The Prudence Manual

Version 2.0-beta11
Main text written by Tal Liron

July 26, 2015

Contents

Part I
Basic Manual

Tutorial

The source code for the examples in the tutorial is available for download.

Up and Running

All you need to run Prudence is a JVM. Version 7 or above is recommended, though version 6 can also be supported. You do not need the “Enterprise Edition” nor even a “JDK”: Prudence needs only the basic JVM runtime.

There are two ways to get Prudence: the easiest way is to download the distribution, which includes most everything and is ready to rumble. Let’s do that now.

But Prudence is modular: later on, you may prefer to assemble your own distribution using the Sincerity packaging and bootstrapping tool.

The Command Line

Actually, the distribution you downloaded comes bundled with Sincerity, which we will use to start Prudence.

Before we do that, let’s quickly familiarize ourselves with Sincerity. Open a command terminal and change to the Prudence directory. We’ll call it our “container.” If you’re using a *nix operating system, you can run the command as “./sincerity”. In Windows it’s “sincerity.bat”.

To see a list of all Sincerity commands:

```
sincerity help
```

For example, let’s print out all installed modules:

```
sincerity dependencies
```

If your operating system supports GUIs, you can start Sincerity without any command to start its GUI (or use the “gui” Sincerity command):

```
sincerity
```

The GUI is useful for visualization, but generally the CLI is better for getting things done. We’ll be using the CLI in this tutorial.

Ready... Set...

Go!

```
sincerity start prudence
```

The command will print out some version information, tell you which applications are installed, and then hold. You should now be able to point your browser to http://localhost:8080 in order to see the Prudence administration application. Or, just go right to http://localhost:8080/stickstick/ and http://localhost:8080/prudence-example/ to see the included examples. Stickstick is an example of an “AJAX” application, implementing sticky notes shared on the web and saved to a relational (SQL) database. It comes with the H2 database, so it does not require a separate database server. The Prudence Example is more general, walking you through Prudence’s basic features in all supported programming languages.

While playing around with the examples, you may also want to look at the logs: they are in the container’s “/logs/” directory. “web.log” is an NCSA-style log of all web requests. Logs prefixed with “prudence-” are created per application, and “common.log” is used for everything else. The logs will automatically roll, so don’t worry about them overflowing your storage.

To quit Prudence, press CTRL+C.

Further Exploration

- Prudence is built on top of Sincerity's Restlet skeleton. The documentation there applies to Prudence, too.
- Sincerity has its own tutorial.
- Of course, you should not run Prudence in a terminal for deployed systems. Instead, you can run Prudence as a robustly monitored system service/daemon (page 148).
- The logging system (page 88) is especially robust. You can use it to centralize logging for your cluster (page 137), and even log directly to a database, such as MongoDB.

First Steps

You've played with the example applications, and now it's time to create your own. Let's make a basic CMS (Content Management System), a web site where users can change the content of pages and create new ones via the web!

Create a new application using the "prudence" Sincerity command:

```
sincerity prudence create cms
```

Restart Prudence, and you should be able to see your new application at http://localhost:8080/cms/. There's not much to see at this point.

Our Application's Subdirectory

Let's take a look at our application. Its files are all under "/component/applications/cms/".

At the root are three JavaScript files used to configure the application. Take a look at "routing.js" and "setting.js". The former defines your application's URI-space, while the latter controls its general behavior. Note that Prudence's (and Sincerity's) configuration philosophy is to use JavaScript wherever possible, which allows for dynamic, flexible configurations that adapt to the environment in which they are run.

Resources

The "/resources/" subdirectory is mapped directly to your URI-space, just like most standard web servers. For example, a file named "/mydir/myfile.html" will be mapped to http://localhost:8080/cms/mydir/myfile.html. Also like standard web servers, "index.html" will be mapped to the directory itself.

Actually, you'll notice that our root index is named "index.t.html". We call that "t" a pre-extension: here, the "t" stands for "template," and tells Prudence that this is a text file that may include one or more sections of code called "scriptlets," which are delimited by "<%" and "%>". This, again, may be familiar to you: it's similar to how PHP, JSP and ASP work.

Under "/resources/style/" you'll see "site.less". LESS is a powerful extended CSS language: Prudence will automatically recognize the ".less" extension and create the CSS for us.

Libraries

Our reusable code is all under the "/libraries/" subdirectory. Specifically, "/libraries/includes/" contains fragments that can be included into our template pages. You'll see that our "index.t.html" uses them. For example:

```
<%& '/header/' %>
```

...includes the file "/libraries/includes/header.html". Note that included files may have scriptlets in them (and can be cached independently).

All the above might seem quite familiar to you, and this is by intention. But the files under "/libraries/dispatched/" may seem strange: they are "low-level" direct implementations of encapsulated RESTful resources. Take a look at "example.js".

Moreover, these kinds of RESTful resources are not directly mapped to URIs like those in the "/resources/" subdirectory. Instead, they are "dispatched" using unique IDs. In Prudence, this paradigm is called "URI/resource separation," and it allows you full flexibility in structuring your code vs. structuring your URI-space.

We'll get into all that as we go along this tutorial.

Scriptlets

Let's look more closely at our "index.t.html" page. The scriptlets are executed, while what's outside the scriptlets is simply output to the HTTP response. You can freely mix regular output and scriptlets:

```
<% for (var x = 0; x < 10; x++) { %>
<p>These are 10 paragraphs.</p>
<% } %>
```

There are also a few shortcut scriptlets. For example, to output an expression:

```
<%= x*2 %>
```

The above is equivalent to:

```
<% print(x*2) %>
```

Also useful is the inclusion scriptlet, which we've already discussed:

```
<%& '/header/' %>
```

The above is equivalent to:

```
<% document.include('/header/'); %>
```

Behind the scenes, Prudence parses "index.t.html" and turns it into JavaScript source code. If you'd like to see the generated code, look at the Scripturian cache, specifically under the container's "/cache/scripturian/container/-component/applications/cms/resources/" directory. These generated files are not actually used by Prudence, and only available for debugging purposes. Disable their creation by configuring "debug" mode to false (page 73).

APIs

That "document" namespace mentioned above is part of the rich, well-documented API provided by Prudence. Other useful namespaces are "conversation," for accessing the request/response, and "application" for shared application-wide services. The complete API reference is available online, though it can be daunting to start there. Instead, continue reading through this tutorial, where we'll explain APIs as we go along, and eventually hit the manual: especially important is the web data chapter (page 51), which goes over much of the interaction with clients.

And remember that you're running on the JVM: the entire JVM standard library is available for you, as well as any other JVM library you install in your container (using Sincerity or manually).

Hello, World

Let's create a new page for our CMS, under our application's "/resources/" subdirectory. We'll call it "page.t.html". We'll start simple:

```
<html>
<body>
        <p>Hello, world. This page will become editable very soon!</p>
</body>
</html>
```

Browse to http://localhost:8080/wiki/page/ to see it. Note that you do not need to restart Prudence if you're only adding or changing pages: Prudence will pick up these changes on-demand, on-the-fly, and make sure to compile and recompile as necessary.

You'll notice that even though the file has a ".t.html" extension, Prudence does not use the extension for the URL. This is because that extension is entirely an implementation concern on your end: there's no reason for the user to have to see it, nor to have it clutter the URL.

Also notice the trailing slash: it's "page/", not "page". *Prudence enforces trailing slashes.* This allows your relative URLs to be clear and unambiguous. For example, to insert an image in our page, we could do this:

```
<img src="../images/logo.png" />
```

Without a trailing slash, you would not need the "../", but *with* the trailing slash, it's *necessary.* If you don't pick one of the two options, handling relative URLs quickly becomes messy. So why have we decided to standardize on trailing slashes? Because it gives you more flexibility: for example, instead of saving our page "page.t.html", we could have saved it as “page/index.t.html”, and the *URL would be identical.* As you will see, the trailing slash principle is used throughout Prudence to allow many such conveniences, for both external URLs and for internal library URIs. You’ll see us constantly ending our URIs in slashes.

Further Exploration

- Learn how to create your own application templates (page 125) for the “prudence create” command.
- Learn how to configure your application (page 70) via its settings.js file.
- Learn about the difference between the “resource mapping” and “URI/resource separation” paradigms (page 25).

Let’s Make a CMS

It’s time to flesh out out “page.t.html”:

```
<html>
<body>
<%
document.require('/prudence/resources/')

if (conversation.request.method.name == 'POST') {
        var form = Prudence.Resources.getForm(conversation, {content: 'string'})
        var content = form.content
        application.globals.put('page', content)
}
else {
        var content = application.globals.get('page') || 'This page is empty.'
}
%>
<div style="border: 1px solid black; padding: 5px 5px 5px 5px">
        <%= content %>
</div>
<form method="post">
        <p>Edit this page:</p>
        <p><textarea name="content" cols="80" rows="20"><%= content %></textarea></p>
        <p><button type="submit">Submit</button></p>
</form>
</body>
</html>
```

Not the most exciting CMS, but it works for editing a single page. Let’s break it down:

- document.require is how we import libraries. (At least for JavaScript: other languages have other import facilities, which you can use instead.) The API will attempt to import libraries from your own application’s “/libraries/” subdirectory first, and if not found there, will use the container’s “/libraries/scripturian/” directory next. You’ll notice that we use a trailing slash in the library URI. The library URIs also match the JavaScript namespaces: “/prudence/resources/” matches Prudence.Resources. This particular namespace is very useful for working with web data. In this case we’re using Prudence.Resources.getForm.
- We’re using the conversation.request API to find out if we’re in an HTTP “POST”. If so, we will extract the “content” form field (as simple text). We will then save it as an “application.global” (page 82). These globals belong to the entire running application, and can be accessed from any page, for any request.
- We then go on to display the content as well as the HTML form for editing the content.

Infinite Editable Pages

The above is just a single page: let's multiply it by *infinity.*

To do this, we'll configure our URI-space by editing the application's routing.js file. To the "app.routes" dict, let's add the following:

```
app.routes = {
        ...
        '/page/*': '/page/'
}
```

What this does is "capture" all incoming URLs that *begin with* "/page/" to exactly our "/page/". The "*" at the end of a URI template signifies a wildcard: anything may follow (except nothing). Restart Prudence for our routing.js changes to take effect.

You'll see that now, indeed, any URL beginning with "/page/" will take us to our single, lonely CMS page. For example, this: http://localhost:8080/cms/page/test/. If you edit one page, the content of all pages would change.

Let's edit our template in "page.t.html" to make it *differ* per incoming URL:

```
<%
document.require('/prudence/resources/')

var id = 'page.' + conversation.wildcard

if (conversation.request.method.name == 'POST') {
        var form = Prudence.Resources.getForm(conversation, {content: 'string'})
        var content = form.content
        application.globals.put(id, content)
}
else {
        var content = application.globals.get(id) || 'This page is empty.'
}
%>
```

And... that's it. Now any URL beginning with "/page/" becomes an editable CMS page. If that page does not exist it, it will be created. Note that we're using "page." as a prefix for our application globals in order to make sure they don't overlap with globals used for other purposes.

The only real "trick" here is the conversation.wildcard API, which gives us the captured wildcard (page 52) we configured in routing.js.

Further Exploration

- Read more about HTML forms (page 56).
- We've barely scratched the surface of what's possible with routing.js. See the URI-space chapter (page 20) for full details.
- There's also, of course, a lot more you can do with templates (page 39), such as reusing fragments and template inheritance.
- Are you a fan of MVC (Model-View-Controller)? It's possible to treat template pages as "views," and moreover you can integrate other templating technologies, such as Jinja2. There's a whole, very detailed chapter about it (page 125).

A Persistent CMS

So far in this tutorial, we've been storing the CMS data in RAM. To store the data on disk, we have a vast array of options: you can use any JVM database driver directly from Prudence.

For this tutorial, we'll be using MongoDB: a document-oriented database with many useful features, such as atomic operations, aggregation and map/reduce, and support for flexible horizontal scalability configurations. It uses JavaScript internally, and thus is a natural fit for web programming in Prudence: you can use JavaScript on the server, JavaScript on the client, and JavaScript in the database. This combo is so popular that we've gone so far as to add especially nice integration with Sincerity/Prudence.

You'll need to install a MongoDB server yourself for this section of the tutorial. If you can't do that right now, don't worry: the rest of the tutorial will work just fine using the in-memory storage we've been using so far, and you can just read through this part.

Let's install the MongoDB driver into our container using a Sincerity command:

```
sincerity add mongodb.javascript : install
```

We can configure the driver by editing our application's settings.js, and adding a section similar to this:

```
app.globals = {
        mongoDb: {
                defaultUris: 'localhost:27017',
                defaultDb: 'cms'
        }
}
```

You can provide full MongoDB connection string URIs, either a single one or an array if you are connecting to a replicaset. The driver will automatically create a thread-safe connection pool upon first use using these settings.

Our new MongoDB-enabled scriptlet:

```
<%
document.require(
        '/prudence/resources/',
        '/mongo-db/')

var id = 'page.' + conversation.wildcard

var collection = new MongoDB.Collection('pages')

if (conversation.request.method.name == 'POST') {
        var form = Prudence.Resources.getForm(conversation, {content: 'string'})
        var content = form.content
        collection.upsert({_id: id}, {$set: {content: content}})
}
else {
        var doc = collection.findOne({_id: id})
        var content = doc ? doc.content : 'This page is empty.'
}
%>
```

Restart Prudence, and check out your new persistent CMS.

Our usage of the MongoDB API is quite trivial here, but in case you're new to it, read up about the $set operation: it guarantees an atomic update of particular fields in our document.

For the rest of this tutorial, we'll continue to use in-memory storage in our examples, but feel free to adapt them to use MongoDB if you already have it set up!

Further Exploration

- Relational (SQL) databases are quite easy to access using JDBC drivers. For a complete Prudence example, see Stickstick. And there are countless frameworks that build abstractions on top of JDBC.
- You can easily administer MongoDB by installing MongoVision into your Prudence container.
- If MongoDB is your cup of tea, check out Diligence, a comprehensive web framework based on Prudence and MongoDB. Especially useful for CMS, Diligence features a documents service, which features versioning and site-wide snapshots.

A Scalable CMS

Caching

As long as we're storing our CMS content in MongoDB or in memory, page rendering will be very fast and light. However, if we were to do more heavy lifting per page—for example, multiple database lookups—then performance would drop accordingly. By smartly caching our pages we can ensure that pages would be rendered only when necessary, allowing for the best possible performance.

Actually, talking about performance is a shorthand for the real issue: even pulling data from a database is usually very fast. The real issue is *scalability*: increased load on the database per user request can limit your ability to support many users. Obviously, it's most efficient to do work *only* when you need it: and that simple principle is the most important tool you have for increasing scalability.

Prudence does caching *very well*. So, though it might be considered an "advanced" topic, it's a crucial one, and worth going over in this tutorial. It's also easy:

```
<%
document.require('/prudence/resources/')

var id = 'page.' + conversation.wildcard
caching.duration = 60000
caching.tags.add(id)
caching.onlyGet = true

if (conversation.request.method.name == 'POST') {
        var form = Prudence.Resources.getForm(conversation, {content: 'string'})
        var content = form.content
        application.globals.put(id, content)
        document.cache.invalidate(id)
}
else {
        var content = application.globals.get(id) || 'This page is empty.'
}
%>
```

What we've done:

- caching.duration is 0 milliseconds by default. We set it to 60 seconds.
- caching.tags allows us to "tag" the cache entry. These tags can then be used to invalidate whole swaths of the cache at once. In this case, we only have one cache entry (the current page) that uses each particular tag.
- caching.onlyGet is set to "true" because we don't want our "POST" requests to use the cache (otherwise they would only be processed for cache misses).
- document.cache.invalidate is used to invalidate a cache tag. In this case, it's our own.

It's easy to see the caching in action via your browser's developer tools. In Firefox, turn on the network monitor by pressing CTRL+SHIFT+Q. In Chromium/Chrome, use CTRL+SHIFT+I and select the "network" tab.

Refresh the page and you will see the "GET" request you just sent. Click on it to see its details, and you will see the response headers returned from the server. The "X-Cache" family of headers will tell you the status of the server-side cache, such as whether it's a "hit" or "miss." You can disable these debug headers by configuring "debug" mode to false (page 73).

However, Prudence also utilizes the *client*-side cache: if you continue refreshing the page from the same browser within the 60-second cache window, the web browser's conditional HTTP requests will return a 304 "not modified" status, which tells the browser that there's no new information on the server, and thus it will efficiently avoid downloading the complete response, using its locally cached value instead.

You've just vastly improved the scalability of your CMS, allowed for a faster user experience, all without ever compromising on the freshness of user data. Prudence caching is pure win.

Note that editing your "page.t.html" file will automatically invalidate the cache, allowing you to instantly see your changes. Actually, this feature extends to the use of includes: if you edit *any* file, it will also invalidate *all* files that include it.

Compression

Prudence will automatically compress responses (using gzip or DEFLATE) if their size in bytes exceeds a certain threshold. Our CMS pages were likely too small, which is why you likely haven't seen compression. You'll usually want that threshold: there are little or no bandwidth savings to be gained by compressing tiny responses, so the extra CPU load required for compression will be wasted. For now, let's lower the threshold, just for demonstration purposes.

Open the application's setting.js and edit the app.settings.compression setting (page 73):

```
app.settings = {
        ...
        compression: {
                sizeThreshold: 0,
                exclude: []
        }
}
```

Restart Prudence for the setting to take effect. With a threshold of zero bytes, *all* responses should now be compressed—assuming the client accepts compression, as indeed web browsers do.

Actually, Prudence integrates compression with caching: both compressed and uncompressed versions of the response are cached, so that compression will not have to be redone. This can result in significant savings for CPU usage in high loads. (And if a client needs a different kind of compression than the one that was stored in the cache—say, gzip was stored, but the client needs DEFLATE—then Prudence will use the uncompressed cache entry, compress it, and then cache. Smart!)

Let's see it in action. Using the web browser's network monitor, you'll notice that the response headers now include "Content-Encoding", which means that our responses are compressed. gzip or DEFLATE will be chosen according to the web browser's preferences. Other than that (and the different response byte size) it should look identical.

To test using cURL, we'll have to do two things: explicitly tell the server that we support compression by sending the "Accept-Encoding" header, and also be ready to uncompress the result in order to display it, using cURL's "compressed" flag:

```
curl --verbose --header 'Accept-Encoding: gzip' --compressed http://localhost:8080/cms/api/te
```

Further Exploration

- Caching is a *big deal*. Read all about it (page 61).
- And caching is just the tip of the scalability iceberg. We treat the topic in depth here (page 159).

A CMS API

Our CMS is currently limited to web *pages*, but we can make it much better by opening it up as a RESTful web *API*. This would allow all kinds of clients to access our CMS, such as dedicated mobile apps, desktop applications, and "rich" web applications (using "AJAX"). It would also allow 3rd-party web sites and services to use our CMS as a service, rather than as an application (in business buzzword land this is called "SaaS").

Hello, API World

As we've seen, templates are Prudence's way of making scalable web pages. The other side of Prudence is manual resources.

We'll want our API in the "/api/" URL, so let's create a "api.m.js" file:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('application/json')
}

function handleGet(conversation) {
        return '{"message": "Hello, API world."}'
}
```

Remember those pre-extensions? "m" stands for "manual." The ".js" extension means that our resource is implemented in JavaScript, though we can end it in ".py", ".rb", etc., if we want to use other supported programming languages.

You *could* test the resource in a web browser by browsing to http://localhost:8080/wiki/api/, however that's not the best way. Also, the web browser only knows how to do "GET" by default. So instead, we recommend testing APIs using cURL.

Let's test our API using this cURL command line:

```
curl http://localhost:8080/cms/api/
```

Hi!

Fleshed Out

We can now fully flesh out our API. It will look quite similar to the template we wrote above:

```
document.require(
        '/prudence/resources/',
        '/sincerity/json/')

function handleInit(conversation) {
        conversation.addMediaTypeByName('application/json')
}

function handleGet(conversation) {
        var id = 'page.' + conversation.wildcard
        var content = application.globals.get(id) || 'This page is empty.'
        return Sincerity.JSON.to({content: content})
}

function handlePut(conversation) {
        var id = 'page.' + conversation.wildcard
        var payload = Prudence.Resources.getEntity(conversation, 'json')
        application.globals.put(id, payload.content)
        document.cache.invalidate(id)
        return Sincerity.JSON.to({content: payload.content})
}

function handleDelete(conversation) {
        var id = 'page.' + conversation.wildcard
        application.globals.remove(id)
        document.cache.invalidate(id)
        return null
}
```

What we did:

- The "handle-" functions are "entry points" into our program, with one per HTTP verb. From Prudence's perspective they are all encapsulated as a single resource class, with each resource instance having its own "conversation" instance.
- handleInit is a bit different in that it dynamically sets up our resource. The MIME type we've added will be used for content negotiation with the client: if several are available, Prudence will automatically select the best type according to the client's preferences. Note that order matters here: the first MIME types you add are preferred over the later ones. For our tutorial, we'll only support JSON responses, though it's possible to support XML and others.
- We've used the Prudence.Resources.getEntity API to get the request payload (page 53). We're expecting the client to send us JSON, but again it's possible to support other formats (page 53).

- Note that we're making sure to invalidate the cache when we change the data: this is to make sure that the CMS pages will be updated accordingly.

We'll also need to change our routing.js to capture the wildcard, again similar to what we did with the template page:

```
app.routes = {
        ...
        '/page/*': '/page/',
        '/api/*': '/api/'
}
```

Restart Prudence for the routing.js change to take effect, and then test it again:

```
curl http://localhost:8080/cms/api/test/
```

Now, let's test an HTTP "PUT" with new page content:

```
curl --request PUT --data '{"content": "New page content!"}' http://localhost:8080/cms/api/te
```

And we can also "DELETE":

```
curl --request DELETE http://localhost:8080/cms/api/test/
curl http://localhost:8080/cms/api/test/
```

While testing with cURL, you can simultaneously test the CMS pages using the browser, as we did earlier. Both the API and the pages see the exact same data, as it's stored in the application.globals.

Cached

Let's add support for caching our API results. Again, it's very similar to what we did with our template, except that caching should be set up in handleInit:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('application/json')
        var id = 'page.' + conversation.wildcard
        caching.duration = 60000
        caching.tags.add(id)
}
```

Note that the cache tag is exactly the same as we're using for template pages: this guarantees that the template pages will be updated even though the CMS is changed from here, and vice versa, even though what is being cached is quite different. Because the cache key depends on the URI, and the URIs for the pages and APIs are different, there will be no conflict.

To see the caching headers when using cURL, add the "verbose" flag:

```
curl --verbose http://localhost:8080/cms/api/test/
curl --verbose --request PUT --data '{"content": "New page content!"}' http://localhost:8080/
```

Further Exploration

- Read all about manual resources (page 36).

Finishing Touches

Here, we'll beef up our CMS a bit, and along the way introduce you to a few more Prudence features.

CSS with LESS

Prudence comes with LESS built in: it's an extended CSS language. LESS greatly increases the power of CSS by allowing for code re-usability, variables and expressions, as well as nesting CSS.

Let's "less up" our CMS pages! First, let's include the CSS in our "page.t.html" by adding an HTML "head" tag:

```
<html>
<head>
        <link rel="stylesheet" type="text/css" href="<%.%>/style/site.css" />
</head>
```

You'll notice our use of "<%.%>": this is a shortcut to printing out the conversation.base API, which returns a *relative* URI path from our currently requested URI to the base URI of the application. For example, if our URI is "http://localhost:8080/cms/page/one/two/three/" then conversation.base would be "../../../..". This frees us from having to hardcode complete URI paths, and guarantees portability even if we move the application to a different URI, attach it to multiple virtual hosts, or run it behind a reverse-proxy (page 142).

Now, let's edit the "/resources/style/site.less" file, which was created for us from the application template. Note that there's also a ".css" file in that directory: Prudence will update it for us when we edit the ".less" file. How about this:

```
@sans-serif: Lucida Sans Unicode, Lucida Grande, Verdana, Arial, sans-serif;
@color1: #EEEEEE;
@color2: #003300;

body {
        font-family: @sans-serif;
        font-size: 12px;
        background-color: @color1;
        color: @color2;
        textarea {
                background-color: @color2;
                color: @color1;
        }
}
```

As a final treat, Prudence can also minify the CSS file for us, in order to save some bandwidth or obfuscate it (Prudence will *also* compress it for us using gzip or DEFLATE, as was mentioned above). To turn on minification, simply change the resource URI to include ".min":

```
<head>
        <link rel="stylesheet" type="text/css" href="<%.%>/style/site.min.css" />
</head>
```

Markdown

Another stylistic flourish would be to use an HTML markup language instead of raw HTML. It's useful for this tutorial, because it will show us how easy it is to use JVM libraries from within Prudence.

First let's install our library using Sincerity. We'll use Pegdown, a JVM implementation of Markdown:

```
sincerity add org.pegdown pegdown : install
```

Now let's edit our "page.t.html" to render using the Pegdown API:

```
<div style="border: 1px solid black; padding: 5px 5px 5px 5px">
        <%= new org.pegdown.PegDownProcessor().markdownToHtml(content) %>
</div>
```

After restarting Prudence we would be able to use Markdown in our CMS. Try to edit a CMS page using this content:

```
This is a title
===============

And a subtitle
--------------

* And these
* Are bullet points
* In a list
```

Internal API

You may have noticed that there's some duplication in our code: both the CMS pages and the CMS API use application.globals to store the content. In this case it's trivial code, but if we moved to database storage, for example, the code would surely grow and be worth encapsulating as an API. One way we could do this is create a reusable library, for example "/libraries/data.js", which we could use both in the CMS pages and the CMS API using "document.require".

But an interesting shortcut is suggested in the fact that *we already have* an encapsulated API: our CMS *web* API. We could simply use that directly. The apparent disadvantage is that it is a web API, and we would have to go through HTTP to call it. However, in Prudence you can call any resource internally, bypassing HTTP entirely, making such requests as fast as any function call.

Let's modify our "page.t.html" for this:

```
<%
document.require('/prudence/resources/')

var id = 'page.' + conversation.wildcard
caching.duration = 60000
caching.tags.add(id)
caching.onlyGet = true

if (conversation.request.method.name == 'POST') {
        var form = Prudence.Resources.getForm(conversation, {content: 'string'})
        var content = form.content
        Prudence.Resources.request({
                uri: '/api/' + conversation.wildcard,
                method: 'put',
                payload: {type: 'json', value: {content: content}}
        })
}
else {
        var data = Prudence.Resources.request({
                uri: '/api/' + conversation.wildcard,
                mediaType: 'application/json'
        })
        var content = data ? data.content : 'This page is empty.'
}
%>
```

We've used the Prudence.Resources.request API to make the request: it will automatically make an *internal* request if the URI begins with "/" (rather than a protocol, such as "http:").

There's actually yet another optimization we can make: we are currently bypassing HTTP, but it's also possible to bypass JSON serialization. That optimization is more advanced, and is fully explained elsewhere (page 116).

Further Exploration

- You don't *have to* use LESS: Prudence can also unify and minify regular CSS, as well as client-side JavaScript (page 48). Also, you may want to read the FAQ (page 100) as to why Prudence supports LESS but not SASS.
- You can use markup languages in scriptlets (page 43).
- Check out the Sincerity markup plugin. It doesn't use Prudence, but you can install it into your container as a utility to allow easy use of markup languages.
- The Prudence.Resources.request API is very powerful, allowing you to easily consume RESTful web APIs.

Not Only JavaScript

The application skeleton uses JavaScript (*server*-side JavaScript; it has nothing to do with the code running inside web browsers), but Prudence also supports Python, Ruby, PHP, Lua, Groovy and Clojure.

PyCMS

For this tutorial, we'll show you how to implement our entire CMS in Python.

First, let's install Python using Sincerity:

```
sincerity add python : install
```

We'll also need a Python JSON library, because our Python engine, Jython, doesn't come with one. simplejson works well with Jython, so let's install it:

```
sincerity easy_install simplejson
```

Now let's rewrite our "page.t.html" using Python instead of JavaScript for our scriptlets:

```
<%py
from org.pegdown import PegDownProcessor

id = 'page.' + conversation.wildcard
caching.duration = 60000
caching.tags.add(id)
caching.onlyGet = True

if conversation.request.method.name == 'POST':
    content = conversation.form['content']
    application.globals[id] = content
    document.cache.invalidate(id)
else:
    content = application.globals[id] or 'This page is empty.'
%>
<div style="border: 1px solid black; padding: 5px 5px 5px 5px">
        <%= PegDownProcessor().markdownToHtml(content) %>
</div>
```

Note the "<%py" delimiter, telling us that from now on scriptlets will be in Python. We can switch back to JavaScript using "<%js", or indeed to any other supported language. You can also change the default from JavaScript to Python if you prefer (page 72).

For our "api.m.js" file, we will have to rename it to "api.m.py". Here's the code in Python:

```
import simplejson

def handle_init(conversation):
    conversation.addMediaTypeByName('application/json')
    id = 'page.' + conversation.wildcard
    caching.duration = 60000
    caching.tags.add(id)

def handle_get(conversation):
    id = 'page.' + conversation.wildcard
    content = application.globals[id] or 'This page is empty.'
    return simplejson.dumps({'content': content})

def handle_put(conversation):
    id = 'page.' + conversation.wildcard
    payload = simplejson.loads(conversation.entity.text)
    application.globals[id] = payload['content']
    document.cache.invalidate(id)
    return simplejson.dumps({'content': payload['content']})

def handle_delete(conversation):
    id = 'page.' + conversation.wildcard
    del application.globals[id]
    document.cache.invalidate(id)
    return None
```

Some things to note:

- Prudence will attempt to use the programming language's naming conventions. For example, JavaScript generally prefers camel-case, while Python uses lowercase-with-underscores: "handleInit" vs. "handle_init". Clojure, for another example, uses lowercase-with-hyphens: "handle-init". However, when you call *from* the language *into* the JVM, the convention depends on the language engine. For our example, Jython requires camel-case. See entry points (page 81).
- When we were using JavaScript we had access to specialized APIs, such as Prudence.Resources.getForm. These APIs are, unfortunately, only for JavaScript: when using other programming languages, you will have to use the "low-level" APIs, such as the conversation.form we've used here. On the other hand, JavaScript suffers from almost no standard library of its own: in Python, for example, you have access to a rich standard library, as well as its wider ecosystem, to make your programming easier.
- We could easily have use *both* Python *and* JavaScript. Why would you want to do that? For example, you might prefer to use JavaScript in scriptlets, because it's more familiar to HTML coders, while having the web API written in Python by a different team. All supported programming languages can live together in a single container.

CljCMS?

Well, to keep this tutorial short, we won't go on rewriting our CMS in every supported programming language... but here's some proof that they *do* work.

Let's install Clojure:

```
sincerity add clojure : install
```

Now let's create a Clojure manual resource, at "/resources/hi.m.clj":

```
(defn handle-init [conversation]
  (.. conversation (addMediaTypeByName "text/plain")))

(defn handle-get [conversation]
  "Hello from Lisp!")
```

Browse to http://localhost:8080/wiki/hi/ to see it.

Further Exploration

- You can even mix scriptlets in several languages on the same template, like magic.

What's Next?

We purposely didn't cover every single Prudence feature in this tutorial: just enough to get you started. Continue reading through the Basic Manual as necessary, refer to the online API documentation, and take a peek at the Advanced Manual to see just how far you can go.

Here are some highlights of topics we haven't covered:

- Prudence has a sophisticated system for handling background tasks (page 102). You can spawn tasks on-demand using APIs, or schedule recurring maintenance tasks using a crontab. In both cases, tasks use a separate thread pool than the one used for handling user requests.
- Prudence supports Swagger (page 94), making it especially easy for clients to consume your RESTful API.

- Runtime debugging of web applications is often difficult. Prudence has a straightforward mechanism for live execution (page 50), allowing you run arbitrary code remotely.
- Clusters (page 137)! When you need to scale horizontally, Prudence provides you with a remarkably seamless solution for sharing state. Even the background task APIs support it: you can farm out your workload anywhere within your Prudence cluster.
- We've already mentioned that Prudence has a very powerful system for defining the URI-space (page 20). You can also configure virtual hosting (page 118) and secure "HTTPS" servers (page 120).
- There's generally a whole lot you can configure (page 117): logging, cache backends, etc.
- We cover deployment strategies in depth (page 140).

The URI-space

The "URI-space" represents the published set of all URIs supported by your server. "Supported" here means that unsupported URIs should return a 404 ("not found") HTTP status code. In other words, they are not in the URI-space.

Importantly, the URI-space can be potentially *infinite*, in that you may support URI templates that match any number of actual URIs (within the limitations of maximum URI length). For example, "/service/{id}/" could match "/service/1/", "/service/23664/", etc., and "/film/*" can match "/film/documentary/mongolia/", "/film/cinema/", etc. All URIs that match these templates belong to your URI-space.

Note that this definition also encompasses the HTTP "PUT" verb, which can be used to create resources (as well as override them). If your server allows for "PUT" at a specific set of URIs, then they are likewise part of your URI-space. In other words, you "support" them.

The URI-space is mostly configured in the application's routing.js file. However, your resource implementations can add their own special limits. For example, for the "/service/{id}/" URI template we can make sure in code that "{id}" would always be a decimal integer (returning 404 otherwise), thus effectively limiting the extent of the URI-space. More generally, Prudence supports "wildcard" URI templates, allowing you to delegate the parsing of the URI remainder entirely to your resource code. This chapter will cover it all.

Make sure to also read the URI-space Architecture article (page 157), which discusses general architectural issues.

routing.js

Routing is configured in your application's routing.js file. The file should configure at least app.routes (page 21) and likely app.hosts (page 31). Add app.errors (page 30), app.dispatchers (page 32), and app.preheat (page 33) if you are using those optional features.

Though routing.js may look a bit like a JSON configuration file, it's important to remember that it's really full JavaScript source! You can include any JavaScript code to dynamically configure your application's routing during the bootstrap process.

Reproduced below is the routing.js used in the "default" application template, demonstrating many of the main route type configurations, including how to chain and nest types. These will be explained in detail in the rest of this chapter.

```
app.routes = {
        '/*': [
                'manual',
                'templates',
                {
                        type: 'cacheControl',
                        mediaTypes: {
                                'image/*': '1m',
                                'text/css': '1m',
                                'application/x-javascript': '1m'
                        },
                        next: {
```

```
                                        type: 'less',
                                        next: 'static'
                                }
                        }
                ],
                '/example1/': '@example', // (dispatched)
                '/example2/': '/example/' // (captured)
}

app.hosts = {
        'default': '/myapp/'
}
```

In these settings, note that time durations are in milliseconds and data sizes in bytes. These can be specified as either numbers or strings (page 71). Examples: "1.5m" is 90000 milliseconds and "1kb" is 1024 bytes.

app.routes

Routes are configured in your application's routing.js, in the app.routes dict.

URI Templates

The *keys* of this dict are *URI templates* (see IETF RFC 6570), which look like URIs, but support the following two features:

- **Variables** are strings wrapped in curly brackets. For example, here is a URI template with two variables: "/profile/{user}/{service}/". The variables will match any text until the next "/". You can access the string values of these variables in your resource as conversation.locals (page 83).
- A **wildcard** can be used as the last character in the URI template. For example, "/archive/*" will match *any* URI that begins with "/archive/". You can access the captured wildcard via the conversation.wildcard API. Note that Prudence will attempt to match *non*-wildcard URI templates first, so a wildcard URI template can be used as a general fallback for URIs.

Route Configurations

The *values* of the app.routes dict are *route configurations*. These are usually defined as JavaScript dicts, where the "type" key is the name of the route type configuration, and the rest of the keys configure the type. During the application's bootstrap process, these dicts are turned in instances of classes in the Prudence.Setup API namespace (the class names have the first character of the type capitalized). The values set in the route type configuration dict are sent to the class constructor.

All route types support the special "hidden" key, which if true specifies that the route is not part of the public URI-space. Prudence will always return a 404 error ("not found") for this match. Note that internal requests always bypass this mechanism (page 115), and so this functionality is useful if you want some URIs available in the internal URI-space but not the public one.

As a shortcut, you can just use a string value (the "type" name) instead of a full dict, however when used this way you must accept the default configuration. We will refer to this as the "short notation" of configuration.

Another useful shortcut: if the type starts with a "!", it is equivalent to setting the "hidden" key to true (this works in both long and short notations). Note that this is different from ending a capture target URI in "!", which would set the "hiddenTarget" key to true: see capture-and-hide (page 26).

There are also special alternate forms for some of the commonly used types, such as JavaScript arrays for the "chain" type. They are detailed below.

We will summarize all the route types briefly here, arranged according to usage categories, and will refer you to the API documentation for a complete reference. Note that some route type configurations allow nesting of further route type configurations.

Resource Route Types

These are routes to a single resource implementation.

"dispatch" or "@" Use the "dispatch" type with an "id" param, or any string starting with the "@" character, to configure a dispatch mapping. For example, {type: 'dispatch', id: 'person'} is identical to '@person'. If you use "@", you can also optionally use a ":" to specify the "dispatcher" param, for example: "@profile:person" is identical to {type: 'dispatch', dispatcher: 'profile', id: 'person'}. If "dispatcher" is not specified, it defaults to "javascript". The unique ID should match a manual resource handled by your dispatcher, otherwise a 404 error ("not found") will result. The "dispatcher" param's value can be any key from the app.dispatchers dict. Handled by the Prudence.Setup.Dispatch class. See the manual resource guide (page 36) for a complete discussion.

> The "manual" route type (page 23) is internally used by Prudence to handle the "dispatch" route type, via a server-side redirect. This introduces two special limitations on its use. First, it means that you *must* have a "manual" if you want to use "dispatch." Second, you must make sure the "manual" always appears *before* any use of "dispatch" in app.routes. For example, if you attach the manual to "/*" in a chain (as in the default application template), and you also want to add a "dispatch" to that chain, you need to put the "manual" *before* the "dispatch" in the chain. Otherwise, you might cause an endless server-side redirect, leading to a stack overflow error. Example of correct use:

```
app.routes = {
        '/*': [
                'manual',
                '@example', // must appear after the manual
                ...
        ],
        ...
}
```

"hidden" or "!" Use the "hidden" or "!" string values to hide a URI template. Prudence will always return a 404 error ("not found") for this match. Note that internal requests always bypass this mechanism (page 115), and so this functionality is useful if you want some URIs available in the internal URI-space but not the public one.

**"resource" or "$..."** Use the "resource" type with a "class" param, or any string starting with the "$" character, to attach a Restlet ServerResource. For example, {type: 'resource', 'class': 'org.myorg.PersonResource'} is identical to '$org.myorg.PersonResource'. This is an easy way to combine Java-written Restlet libraries into your Prudence applications. Handled by the Prudence.Setup.Resource class.

"execute" Use the "execute" type to attach a code execution resource. This powerful (and dangerous) resource executes all POST payloads as Scripturian templates. The standard output of the script will be returned as a response. Because it always execution of arbitrary code, you very likely do not want this resource publicly exposed. If you use it, make sure to protect its URL on publicly available machines! Handled by the Prudence.Setup.Execute class. The "execute" resource is very useful for debugging (page 87).

"status" or "!..." Use the "status" type with a "code" param, or any number starting with a "!" character, to simply return an HTTP status code, doing nothing else. Useful for quick-and-dirty URI bindings. For example, "!401" would mark the URI as "not authorized" to all requests. Handled by the Prudence.Setup.Status class.

Mapping Route Types

You should use a wildcard URI template for all of these route types, because they work by processing the URI remainder.

"static" Use the "static" type to create a static resource handler. By default uses the application's "/resources/" subdirectory chained to the container's "/libraries/web/" subdirectory for its "roots". Note that if you include it in a "chain" with "manual" and/or "templates", then "static" should be the last entry in the chain. Handled by the Prudence.Setup.Static class. See the static resources guide (page 46) for a complete discussion.

“manual” Use the “manual” type to create a manual resource handler. By default uses the application’s “/resources/” subdirectory for its “root”. Important limitation: *All* uses of this class in the same application share the same configuration. Only the first found configuration will take hold and will be shared by other instances. Handled by the Prudence.Setup.Manual class. See the manual resources guide (page 36) for a complete discussion.

“templates” Use the “templates” type to create a template resource handler. By default uses the application’s “/resources/” subdirectory for its “root”. Important limitation: *All* uses of this class in the same application share the same configuration. Only the first found configuration will take hold and will be shared by other instances. Handled by the Prudence.Setup.Templates class. See the template resources guide (page 39) for a complete discussion.

Redirecting Route Types

“capture” or “/...” Use the “capture” type with a “uri” param, or any string starting with the “/” character, to configure a capture. For example, {type: ’capture’, uri: ’/user/profile/’} is identical to ’/user/profile/’. Note that adding a “!” character at the end of the URI (not considered as part of the actual target URI) is a shortcut for *also* hiding the target URI. Capturing-and-hiding (page 26) is indeed a common use case. Handled by the Prudence.Setup.Capture class. See resource capturing (page 28) for a complete discussion.

“redirect” or “>...” Use the “redirect” type with a “uri” param, or any string starting with the “>” character, to asks the client to redirect (repeat its request) to a new URI. For example, {type: ’redirect’, uri: ’http://newsite.org/user/profile/’} is identical to ’>http://newsite.org/user/profile/’. Handled by the Prudence.Setup.Redirect class. See the web data guide (page 55) for a complete discussion, as well as other options for redirection.

“addSlash” Use the “addSlash” type for a permanent client redirect from the URI template to the original URI with a trailing slash added. It’s provides an easy way to enforce trailing slashes in your application. Handled by the Prudence.Setup.AddSlash class.

Combining Route Types

“chain” or “[...]” Use the “chain” type with a “restlets” param (a JavaScript array), or just a JavaScript array, to create a fallback chain. The values of the array can be any route type configuration, allowing for nesting. They will be tested in order: the first value that *doesn’t* return a 404 (“not found”) error will have its value returned. This is very commonly used to combine mapping types, for example: [’manual’, ’templates’, ’static’]. Handled by the Prudence.Setup.Chain class.

“router” Use the “router” type with a “routes” param (a JavaScript dict) to create a router. The values of the dict can be any route type configuration, allowing for nesting. This is in fact how Prudence creates the root router (app.routes). Handled by the Prudence.Setup.Router class.

Filtering Route Types

All these route types require a “next” param for nesting into another route type. See the filtering guide (page 108) for a complete discussion.

“filter” Use the “filter” type with the “library” and “next” params to create a filter. “library” is the document name (from the application’s “/libraries/” subdirectory), while “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.Filter class.

“injector” Use the “injector” type with the “locals” and “next” params to create an injector. An injector is a simple filter that injects preset valued into conversation.locals (page 83), but otherwise has no effect on the conversation. This is useful for inversion of control (IoC): you can use these conversation.locals to alter the behavior of nested route types directly in your routing.js. Handled by the Prudence.Setup.Injector class.

“basicHttpAuthenticator” Use the “basicHttpAuthenticator” with the “credentials”, “realm” and “next” params to require HTTP authentication before allow the request to go through. This straightforward (but weak and inflexible) security mechanism is useful for ensuring that robots, such as search engine crawlers, as well as unauthorized users do not access a URI. Handled by the Prudence.Setup.BasicHttpAuthenticator class. See the HTTP authentication guide (page 112) for a complete discussion.

“cacheControl” Use the “cacheControl” type with a “next” param to create a cache control filter. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.CacheControl class. See the static resources guide (page 47) for a complete discussion.

“cors” Use the “cors” type with a “next” param to create a Cross-Origin Resource Sharing (CORS) filter. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.Cors class. See the CORS guide (page 112) for a complete discussion.

“javaScriptUnifyMinify” Use the “javaScriptUnifyMinify” type with a “next” param to create a JavaScript unify/minify filter. “roots” defaults to your application’s “/resources/scripts/” and your container’s “/libraries/web/scripts/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.JavaScriptUnifyMinify class. See the static resources guide (page 48) for a complete discussion.

“cssUnifyMinify” Use the “cssScriptUnifyMinify” type with a “next” param to create a CSS unify/minify filter. “roots” defaults to your application’s “/resources/style/” and your container’s “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.CssUnifyMinify class. See the static resources guide (page 48) for a complete discussion.

“less” Use the “less” type with a “next” param to create a LESS compiling filter. “roots” defaults to your application’s “/resources/style/” and your container’s “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Setup.Less class. See the static resources guide (page 49) for a complete discussion.

Custom Route Types

With some knowledge of the Restlet library, you can easily create your own custom route types for Prudence:

1. Create a JavaScript class that:
   (a) Implements a create(app, uri) method. The “app” argument is the instance of Prudence.Setup.Application, and the “uri” argument is the URI template to which the route type instance should be attached. The method must return an instance of a Restlet subclass.
   (b) Accepts a single argument, a dict, to the constructor. The dict will be populated by the route type configuration dict in app.routes.
2. Add the class to Prudence.Setup. Remember that the class name begins with an uppercase letter, but will begin with a lowercase letter when referenced in app.routes.

If you like, you can use Sincerity.Classes to create your class (you don’t have to), and also inherit from Prudence.Setup.Restlet.

Here’s a complete example in which we implement a route type, named “see”, that redirects using HTTP status code 303 (“see other”). (Note this same effect can be better achieved using the built-in “redirect” route type, and is here intended merely as an example.)

```
document.require('/sincerity/classes/')

Prudence.Setup.See = Sincerity.Classes.define(function() {
        var Public = {}

        Public._inherit = Prudence.Setup.Restlet
        Public._configure = ['uri']
        Public.create = function(app, uri) {
                importClass(org.restlet.routing.Redirector)
                var redirector = new Redirector(app.context, this.uri, Redirector.MODE_CLIENT
                return redirector
        }

        return Public
}())

app.routes = {
        ...
        '/original-uri/': {type: 'see', uri: 'http://newsite.org/new-uri/'}
}
```

Two Routing Paradigms

Prudence offers *three* built-in techniques for you to support a URI or a URI template, reflecting *two* different routing paradigms:

1. **Resource mapping** (page 25): The filesystem hierarchy under an application's "/resources/" subdirectory is directly mapped to URIs (but not URI templates). Both directory- and file-names are mapped in order of depth. By default, Prudence hides filename extensions from the published URIs, but uses these extensions to extracts MIME-type information for the resources. Also, mapping adds trailing slashes by default, by redirecting URIs without trailing slash to include them (on the client's side). Filesystem mapping provides the most "transparent" management of your URI-space, because you do not need to edit any configuration file: to change URIs, you simply move or rename files and directories.
2. **URI/resource separation:**
   (a) **Resource capturing** (page 28): Capturing lets you map URI templates to fixed URIs, as well as perform other kinds of internal URI rewrites that are invisible to clients, allowing you to provide a published URI-space, which is different from your internal mapping structure. Note that another common use for capturing is to add support for URI templates in resource mapping, as is explained in resource mapping (page 25). This use case does not belong to the URI/resource separation paradigm.
   (b) **Resource dispatching** (page 28): In your application's routing.js you can map URIs or URI templates to a custom ID, which is then dispatched to your resource handling code. Dispatching provides the cleanest and most flexible separation between URIs and their implementation.

When embarking on a new project, you may want to give some thought as which paradigm to use. Generally, URI/resource separation is preferred for larger applications because it allows you more choices for your code organization. However, it does add an extra layer of configuration, and the URI-space is not as transparent as it is for resource mapping. It may make sense to use both paradigms in the same application where appropriate. Read on, and make sure you understand how to use all three routing techniques.

Resource Mapping

Resource mapping is the most straightforward and most familiar technique and paradigm to create your URI-space. It relies on a one-to-one mapping between the filesystem (by default files under your application's "/resources/" subdirectory) to the URI-space. This is how static web servers, as well as the PHP, JSP and ASP platforms usually work.

Prudence differs from the familiar paradigm in three ways:

1. For manual and template resources, Prudence hides filename extensions from the URIs by default. Thus, "/resources/myresource.m.js" would be mapped to "/resources/myresource/". The reasons are two: 1) clients should not have to know about your internal implementation of the resource, and 2) it allows for cleaner and more coherent URIs. Note the filename extensions are used internally by Prudence (differently for manual and template resources). Note that this does not apply to static resources: "/resources/images/logo.png" will be mapped to "/images/logo.png".

2. For manual and template resources, Prudence by default *requires* trailing slashes for URIs. If clients do not include the trailing slash, they will receive a 404 ("not found") error. Again, the reasons are two: 1) it makes relative URIs always unambiguous, which is especially relevant in HTML and CSS, and 2) it clarifies the extent of URI template variables. As a courtesy to sloppy clients, you can manually add a permanent redirection to a trailing slash, using the "addSlash" route type (page 23). For example:

   ```
   app.routes = {
           ...
           '/main', 'addSlash',
           '/person/profile/{id}': 'addSlash'
   }
   ```

3. This mapped URI-space can be manipulated using URI hiding and capturing, allowing you to support URI templates and rewrite URIs.

Mapping URI Templates

The problem with resource mapping is that the URIs are "static": they are only and exactly the the directory and file path. However, this limitation is easily overcome by Prudence's "capturing" mechanism, which works on URI templates. For example, let's say you have a template resource file at "/resources/user/profile.t.html", but instead of it being mapped to the URI "/user/profile/", you want to access it via a URI template: "/user/profile/{userId}/":

```
app.routes = {
        ...
        '/user/profile/{userId}/': '/user/profile/'
}
```

A URI such as "/user/profile/4431/" would then be internally redirected to the "/user/profile/" URI. Within your "profile.t.html" file, you could then access the captured value as conversation.locals (page 83):

```
<html>
<body>
<p>
User profile for user <%= conversation.locals.get('userId') %>.
</p>
</body>
</html>
```

We've used a template resource in this example, but capturing can be used for both template and manual resources. The same conversation.locals API (page 83) is used in both cases.

Capture-and-Hide You may also want to make sure that "/user/profile/" cannot be accessed *without* the user ID. To capture and hide together you can use the shortcut notation:

```
app.routes = {
        ...
        '/user/profile/{userId}/': '/user/profile/!'
}
```

That final "!" is equivalent to setting "hiddenTarget: true" in the long notation. You may also configure capturing and hiding separately, using the "hidden" route type (page 22). The following is equivalent to the above:

```
app.routes = {
        ...
        '/user/profile/{userId}/': '/user/profile/',
        '/user/profile/': '!'
}
```

Dynamic Capturing

URI capturing can actually do more than just capture to a *single* URI: the target URI for a capture is, in fact, *also* a URI template, and can include any of the conversation attributes discussed in the string interpolation guide (page 113). For example:

```
app.routes = {
        ...
        '/user/{userId}/preferences/': '/database/preferences/{m}/?id={userId}'
}
```

The request method name would then be interpolated into the "{m}", for example it could be "GET" or "POST". It would thus capture to different target URIs depending on the request. So, you could have "/database/preferences/GET.html" and "/database/preferences/POST.html" files in your "/resources/" subdirectory to handle different request methods.

Note that it's also possible to dynamically capture and interpolate the wildcard (page 52), for example:

```
app.routes = {
        ...
        '/user/*': '/database/user/?id={rw}'
}
```

Dynamic Capture-and-Hide Note that if you use the "!" capture-and-hide trick with dynamic capturing, Prudence will hide *any* URI that matches the template. For example:

```
app.routes = {
        ...
        '/user/{userId}/preferences/': '/database/preferences/{m}/!'
}
```

Here, "/database/preferences/GET/" is hidden, but also "/database/preferences/anything/", etc. If you do *not* want this behavior, then you should explicitly hide specific URIs instead:

```
app.routes = {
        ...
        '/user/{userId}/preferences/': '/database/preferences/{m}/',
        '/database/preferences/GET/': '!',
        '/database/preferences/POST/': '!'
}
```

Limitations of Resource Mapping

While resource mapping is very straightforward—one file per resource (or per type of resource if you capture URI templates)—it may be problematic in three ways:

1. In large URI-spaces you may suffer from having too many files. Though you can use "/libraries/" to share code between your resources, mapping still *requires* exactly one file per resource type.
2. Mapped manual resources must have all their entry points (handleInit, handleGet, etc.) defined as global functions. This makes it awkward to use object oriented programming or other kinds of code reuse. If you define your resources as classes, you would have to hook your class instance via the global entry points.
3. The URI-space is your public-facing structure, but your internal implementation may benefit from an entirely different organization. For example, some resources my be backed by a relational database, others by a memory cache, and others by yet another subsystem. It may make sense for you to organize your code according to subsystems, rather than the public URI-space. For this reason, you would want the URI-space configuration to be separate from your code organization.

These problems might not be relevant to your application. But if they are, you may prefer the URI/resource separation paradigm, which can be implemented via resource capturing or dispatching, documented below.

URI/Resource Separation

Resource Capturing

Resource capturing, for the purpose of the URI/resource separation paradigm, only makes sense for template resources. For manual resources, resource dispatching (page 28) provides a similar structural function.

Resource capturing lets you use any public URI for any library template resource. For example, let's assume that you have the following files in "/libraries/includes/": "/database/profile.html", "/database/preferences.html" and "/cache/session.html", which you organized in subdirectories according to the technologies used. Your URI-space can be defined thus, using the "capture" route type (page 23):

```
app.routes = {
        ...
        '/user/{userId}/preferences/': '/database/preferences/',
        '/user/{userId}/profile/': '/database/profile/',
        '/user/{userId}/session/': '/cache/session/'
}
```

Note how the URI-space is organized completely differently from your filesystem: we have full URI/resource separation.

> **Under the hood**: Prudence's capturing mechanism is implemented as server-side redirection (sometimes called "URI rewriting"), with the added ability to use hidden URIs as the destination. It's this added ability that makes capturing useful for URI/resource separation: hidden URIs include both template resource files in your application's "/libraries/includes/" subdirectory as well as URIs routed to the "hidden" route type.

Resource Dispatching

Resource dispatching, for the purpose of the URI/resource separation paradigm, only makes sense for manual resources. For template resources, resource capturing (page 28) provides a similar structural function.

Configuring a dispatch is straightforward. In routing.js, use the "dispatch" route type (page 22), or the "@" shortcut:

```
app.routes = {
        ...
        '/session/{sessionId}/': '@session',
        '/user/{userId}/preferences/': '@user'
}
```

The long-form notation, with all the settings at their defaults, would look like this:

```
app.routes = {
        ...
        '/session/{sessionId}/': {
                type: 'dispatch',
                id: 'session',
                dispatcher: null,
                locals: []
        }
}
```

Note that, similarly to capturing, you can interpolate conversation attributes (page 113) into the ID. For example, here we will automatically use a different dispatch ID according the protocol, either "session.HTTP" or "session.HTTPS":

```
app.routes = {
        ...
        '/session/{sessionId}/': '@session.{p}'
}
```

The Dispatch Map By default, to implement your dispatched resources, create a library named "/dispatched/", which is expected to change a global dict named "resources". That dict maps IDs to implementations.

For our example, let's create a "/libraries/dispatched.js" file:

```
var UserResource = function() {
        this.handleInit = function(conversation) {
                conversation.addMediaTypeByName('text/plain')
        }

        this.handleGet = function(conversation) {
                return 'This is user #' + conversation.locals.get('userId')
        }
}

resources = {
        session: {
                handleInit: function(conversation) {
                        conversation.addMediaTypeByName('text/plain')
                },
                handleGet: function(conversation) {
                        return 'This is session #' + conversation.locals.get('sessionId')
                }
        },
        user: new UserResource()
}
```

We've mapped the "session" dispatch ID to a dict, and used simple JavaScript object-oriented programming for the "user" dispatch ID. (Note that the Sincerity.Classes facility offers a comprehensive object-oriented system for JavaScript, but we preferred more straightforward code for this example.)

As you can see, the resources.js file does not refer to URIs, but instead to dispatch IDs, which you can dispatch as you see fit. This is true URI/resource separation.

These defaults are all configurable: see app.dispatchers (page 32) for more information.

Other Programming Languages Resource dispatching is also supported for Python, Ruby, PHP, Lua, Groovy and Clojure, via alternate dispatchers. To use them, you must specify the dispatcher, which is simply the name of the language in lowercase, when you configure the dispatch. For example:

```
app.routes = {
        ...
        '/session/{sessionId}/': {
                type: 'dispatch',
                id: 'session',
                dispatcher: 'python'
        }
}
```

You can also use the "@" shortcut like so:

```
app.routes = {
        ...
        '/session/{sessionId}/': '@python:session'
}
```

Inversion of Control (IoC)

Object-oriented inheritance is one useful way to reuse code while allowing for special implementations. Additionally, Prudence allows for a straightforward inversion of control mechanism (page 111).

For both capturing and dispatching, you can inject set values to conversation.locals (page 83). You would need to use the long-form notation to do this. Here are examples for both a capture and a dispatch:

```
app.routes = {
        ...
        '/user/{userId}/': {type: 'capture', uri: '/user/', locals: {style: 'simple'}},
        '/user/{userId}/full/': {type: 'capture', uri: '/user/', locals: {style: 'full'}},
        '/user/{userId}/preferences/': {type: 'dispatch', id: 'user', locals: {section: 'pref
        '/user/{userId}/profile/: {type: 'dispatch', id: 'user', locals: {section: 'profile'}
}
```

Note that in each case two URI templates are captured/matched to the exact same resource, but the "locals" dict used is different for each. In your resource implementations, you can then allow for different behavior according to the value of the set conversation.local (page 83). For example:

```
this.handleGet = function(conversation) {
        var section = conversation.locals.get('section')
        if (section == 'preferences') {
                ...
        }
        else if (section == profile') {
                ...
        }
}
```

This allows you to configure your resources in routing.js, rather than at their implementation code. In other words, "control" is "inverted," via value injection.

Pure URI/Resource Separation

The default routing.js makes all the resource types public, and chained to the root URI. At its simplest, it looks like this:

```
app.routes = {
        '/*': [
                'manual',
                'templates',
                'static'
        ]
}
```

However, if you want complete control over URI/Resource separation, you can avoid having these public by hiding them at specific URIs:

```
app.routes = {
        '/_templates/*': '!templates',
        '/_manual/*': '!manual',
        '/_static/*': '!static',

        '/style/*' : '/_static/style/{rw}',
        '/*': '/_templates/page/?id={rw}'
}
```

With the above routing.js, we can then specify exactly which URIs go where, with nothing made public by default. The "_" prefixes are merely a convention in this case to specify an internal-only URI.

app.errors

By default, Prudence will display ready-made error pages if the response's HTTP status code is an error (>=400), but you can override these by settings your own custom pages:

```
app.errors = {
        404: '/errors/not-found/',
        500: '/errors/fail/'
}
```

The keys of the dict are status codes, while the values work like captures (page 28). This means that you can implement your error pages using any kind of resource: manual, template or static.

If your error resource is mapped under "/resources/", and you don't want it exposed, use the capture-and-hide notation (page 26):

```
app.errors = {
        404: '/errors/not-found/!'
}
```

This is equivalent to explicitly hiding the URI in app.routes using the "hidden" route type (page 22):

```
app.routes = {
        ...
        '/errors/not-found/': '!'
}
```

Note that 500 ("internal server error") statuses caused by uncaught exceptions can be handled specially by enabling debug mode (page 72).

> **Warning:** If you decide to implement your error pages using a *non*-static resource, you need to take two things into account. For 404 errors, you must have the resource type handling the page be the *first* handler in a chain in app.routes. For example, if you are using a template resource, then you must have "templates" first. The reason is that a chain works internally by detecting 404 errors and moving on to the next one: if the first one returns a 404 again, this will result in a recursion and will throw a 500 exception. Second, for any non-static error resource, you want to be *very sure* that your code there does not throw an exception. If that happens, it would cause a 500 status code, but if the original error was *not* 500, the result would confuse the user and complicate debugging. However, if the original status code *was* 500, then it could be much worse: the error would trigger your error resource to be displayed again, which might throw the exception again... resulting in a stack overflow error. Due to these pitfalls, it's probably a good idea to implement your error pages as static resources.

It is also possible to set error captures for *all* applications by configuring them into the component (page 123). In such a case, if applications configure their own error captures, those would override the component-wide configuration.

app.hosts

Use app.hosts to assign the application to a base URI on one or more virtual hosts.

The default configuration uses a single host, the "default" one, but you can add more, or even not have a single "default" host. Still, note that only a *single* application instance is used, even if attached to multiple hosts: for example, all application.globals (page 82) are shared no matter which host a request is routed from.

To configure your hosts, and for a more complete discussion, see the component configuration guide (page 118).

The Bare Minimum

You need to attach your application to at least one host to make it public.

```
app.hosts = {
        'default': '/myapp/'
}
```

(Note the required use of quotes around "default", due to it being a reserved keyword in JavaScript.)

If you don't have an app.hosts definition, or if it is empty, then your application *will not be publicly available over HTTP*. However, it will still run, and be available as an internal API.

The Internal Host

"internal" is a reserved host name, used to represent the internal URI-space (page 115).

By default, applications are always attached to the "internal" host with the base URI being the application's subdirectory name. However, you can override this default should you require:

```
app.hosts = {
        'default': '/myapp/',
        internal: '/special/'
}
```

Multiple Hosts

Here's an example of attaching the app to two different virtual hosts, with different base URIs under each:

```
app.hosts = {
        'default': '/myapp/',
        'privatehost': '/admin/'
}
```

If you need your application to sometimes behave differently on each host, use the conversation.host API. Note that it will be null when routed from the internal host. Example:

```
var host = conversation.host
if (Sincerity.Objects.exists(host) && (host.name == 'privatehost')) {
        ...
}
```

app.dispatchers

This section is optional, and is used to change the way your application's dispatching works.

Setting the Default Dispatcher

As seen above in resource dispatching (page 28), if you you do not specify the dispatcher when configuring a "dispatch" route type (page 22), it will default to the "javascript" dispatcher. You may change the default like so:

```
app.routes = {
        '/session/{sessionId}/': '@session'
}

app.dispatchers = {
        'default': 'python'
}
```

In the above, our "@session" dispatch will be using the "python" dispatcher.

Setting the Dispatch Map

We've also seen that by default the dispatcher will attempt to execute your "/dispatched/" library in order to initialize the dispatch map. However, you may change this location per dispatcher like so:

```
app.dispatchers = {
        javascript: '/mylib/mydispatchmap/'
}
```

This is especially useful if you're using more than one dispatcher in your routing, because each dispatcher would need its own resource map. For example:

```
app.routes = {
        ...
        '/user/{userId}/': '@user',
        '/session/{sessionId}/': '@python:session',
        '/page/{pageId}/': '@groovy:page'
}

app.dispatchers = {
        javascript: '/mylib/javascript-dispatchmap/',
        python: '/mylib/python-dispatchmap/',
        groovy: '/mylib/groovy-dispatchmap/'
}
```

Custom Dispatchers

Custom dispatchers are very useful for integrating alternative templating engines (page 129) into Prudence.

Under the hood, resource dispatching is handled by the URI capturing mechanism: the URI is captured to a special manual resource—the "dispatcher"—with an injected value (page 111) specifying the ID of resource to which it should dispatch.

Prudence's default dispatchers can be found in the "/libraries/scripturian/prudence/dispatchers/" directory of your container. For example, the JavaScript dispatcher is "/libraries/scripturian/prudence/dispatchers/-javascript.js". You are encouraged to look at the code there in order to understand how dispatching works: it's quite straightforward delegation of the entry points of a manual resource (page 36).

However, you can also write your own dispatchers to handle different dispatching paradigms. To configure them, you will need to use a long-form for app.dispatchers. For example, here we override the default "javascript" dispatcher:

```
app.dispatchers = {
        ...
        special: {
                dispatcher: '/dispatchers/special-dispatcher/',
                customValue1: 'hello',
                customValue2: 'world'
        }
}
```

The dispatcher code (a standard manual resource) would be in a "/libraries/dispatchers/special-dispatcher.js" file under your application's subdirectory. You would be able to access the dispatch ID there as the injected "prudence.dispatcher.id" conversation.local (page 83). "customValue1" and "customValue2" would be application.globals (page 82): "prudence.dispatcher.special.customValue1" and "prudence.dispatcher.special.customValue2" respectively.

You can also override the default dispatchers:

```
app.dispatchers = {
        ...
        javascript: {
                dispatcher: '/dispatchers/my-javascript-dispatcher/',
                resources: '/resources/'
        }
}
```

See the section on integrating alternative templating engines (page 129) for a complete example.

app.preheat

When a resource gets hit by a request for the first time, there will likely be an initial delay. Your code may have to be compiled, or, if it has been cached, would at least have to be loaded and initialized. But all that should happen very fast: the more serious delay would be caused by initializing subsystems and libraries, such as connecting to databases, logging in to external services, etc.

Another source for delay is that, if you are caching pages (page 62), as you very well should, then the some cached pages may not exist or be old. The first hit would thus need to generate and cache the page, which is much slower than using the cached version. For very large applications optimized for caching, a "cold" cache can cause serious problems: see the discussion in Scaling Tips (page 162).

To avoid the delay, it's recommended that you ready your resources by hitting them as soon as Prudence starts up. This happens in two phases:

- **Defrosting**: Prudence will by default parse and compile the code for manual and template resources under your application's "/resources/" subdirectory. See configuring applications (page 72) if you wish to turn this feature off.
- **Preheating**: This causes an internal (page 115) "GET" on URIs, which would involve not only compiling the code, but also running it. The response payloads, as well as any errors, are ignored.

Your app.preheat is simply an array of relative URIs. As an example, let's preheat the homepage and a few specific resources:

```
app.preheat = [
        '/',
        '/user/x/',
        '/admin/'
]
```

Note that you can't use URI templates in app.preheat, only explicit URIs. Thus, if you have routed a "/user/{name}/" URI template, you would have to "fill in" the template with a value. In our example, we chose "/user/x/". Also note that because preheats are *internal* requests you can use this mechanism to preheat hidden URIs.

Smart use of app.preheat can very effective, and it's very easy to use. Note that you may also use the startup task (page 108) for your own custom preheating.

Both defrosting and preheating are handled in a thread pool, so that they can finish as quickly as possible. Learn how to configure it here (page 124). The duration of the complete process will be announced when you start Prudence. For example:

```
Executing 3488 startup tasks...
Finished all startup tasks in 1.432 seconds.
```

Understanding Routing

In this final section, we'll describe in detail how routing works in Prudence. It can be considered optional, advanced reading.

In Prudence, "routing" refers to the decision-making process by which an incoming client request reaches its server-side handler. Usually, information in the request itself is used to make the decision, such as the URI, cookies, the client type, capabilities and geolocation. But routing can also take server-side and other circumstances into account. For example, a round-robin load-balancing router might send each incoming request to a different handler in sequence.

A request normally goes through many route types before reaching its handler. Filters along the way can change information in the request, which could also affect routing, and indeed filters can be used as routing tools.

This abstract, flexible routing mechanism is one of Prudence's most powerful features, but it's important to understand these basic principles. A common misconception is that routing is based on the hierarchical structure of URIs, such that a child URI's routing is somehow affected by its parent URI. While it's possible to explicitly design your routes hierarchically, routing is primarily to be understood in terms of the order of routers and filters along the way. A parent and child URI could thus use entirely different handlers.

To give you a better understanding of how Prudence routing works, let's follow the journey of a request, starting with routing at the server level.

Step 1: Servers Requests come in from servers (page 120). Each server listens at a particular HTTP or HTTPS port, and multiple servers may in turn be restricted to particular network interfaces on your machine. By default, Prudence has a single server that listens to HTTP requests on port 8080 coming in from all network interfaces.

Step 2: The Component There is only one component per Prudence instance, and *all* servers route to it. This allows Prudence a unified mechanism to deal with all incoming requests.

Step 3: Virtual Hosts The component's router decides which virtual host (page 118) should receive the request. The decision is often made according to the domain name in the URL, but can also take into account which server it came from. Virtual hosting is a tool to let you host multiple sites on the same Prudence instance, but it can be used for more subtle kinds of routing, too.

At the minimum you must have one virtual host. By default, Prudence has one that accepts all incoming requests from all servers. If you have multiple servers and want to treat them differently, you can create a virtual host for each.

Step 4: Applications Using app.hosts (page 31), you can configure which virtual hosts your application will be attached to, as well as the base URI for the application on each virtual host. An application can be configured to accept requests from several virtual hosts.

To put it another way, there's a many-to-many relationship between virtual hosts and applications: one host can have many applications, and the same application can be attached to many hosts.

Note that you can create a "nested" URI scheme for your applications. For example, one application might be attached at the root URI at a certain virtual host, "/", while other applications might be at different URIs beneath the root, "/cms/" and "/cms/support/forum/". The root application will not "steal" requests from the other applications, because the request is routed to the right application by the virtual host. The fact that the latter URI is the hierarchical descendant of the former makes no difference to the virtual host router.

A Complete Route Let's assume a client from the Internet send a request to URI "http://www.wacky.org/cms/support/forum/thread/12/."

Our machine has two network interfaces, one facing the Internet and one facing the intranet, and we have two servers, one for each network adapter. This particular request has come in through the external server. The request then reaches the component's router.

We have a few virtual hosts: one to handle "www.wacky.org", our organization's main site, and another to handle "support.wacky.org", a secure site where registered users can open support tickets.

Our forum application (in the "/applications/forum/" subdirectory) is attached to both virtual hosts, but at different URIs. It's at "www.wacky.org/cms/support/forum/" and at "support.wacky.org/forum/". In this case, our request is routed to the first virtual host. Though there are a few applications installed at this virtual host, our request follows the route to the forum application.

The remaining part of the URI, "/thread/12/" will be further routed inside the forum application, according to route types installed in its routing.js.

Implementing Resources

Prudence offers two options for implementing resources in which the content dynamically changes: "manual" resources are "raw," giving you complete low-level control over the behavior and format of the encapsulated resource, while "template" resources are simplified and highly optimized for cached textual resources, such as HTML web pages. Prudence also provides comprehensive support for static (unchanging) resources, just like conventional web servers.

Programmable Resources

These notes apply to both manual and template resources:

- Resources can be implemented using any of the supported programming languages.
- Uncaught exceptions in any entry point or scriptlet will result in the conversation ending and a 500 ("internal server error") HTTP status code returned to the user. To help you debug these errors, turn on debug mode (page 72), which will enable a detailed debug representation. Otherwise, you can set up a custom error page (page 30), or, in the last resort, a simple error page will be shown, that at least notifies the user that something went wrong.
- Your source code is parsed/compiled on-the-fly *only when necessary.* A reparsing/recompilation is triggered when the file is modified, or when any of its dependent files are modified. To keep track of dependencies at any distance, Prudence internally maintains a dependency tree for each document. Note that checking for file

modification dates involves an operating system API call: the call is usually cached by the OS and very fast, however you can configure the minimum time between validity checks (page 72) if necessary.

- If you've captured a URI template (page 21), you can access the captured parts of the template via conversation.locals (page 83) or conversation.wildcard.

Manual Resources

These are implemented as a set of encapsulated entry points (page 81) in any of the supported programming languages. The entry points all receive the current conversation API namespace as their only argument.

Prudence does the encapsulation for you: it treats the entry points as together belonging to a single logical RESTful resource, and ensures that the same conversation namespace is used for every user request.

There are two ways to define this encapsulation, depending on which routing paradigm (page 25) you're using:

- **Resource mapping**: Here, the entry points are in the global scope, probably defined within the file, though they can be imported via a library or created programmatically (as closures, for example). What an "entry point" means exactly may vary per programming language: it's usually a function, method, closure, etc. See the programming guide (page 81) for examples in all supported languages.
- **Resource dispatching**: The default dispatchers attempt an object-oriented encapsulation, which again varies per programming language. The dispatch ID defines an object instance, which must in turn implement the entry points.

The implementation is in essence the same for both styles. For simplicity, our examples below will be tuned to resource mapping.

Configuration

Add support for manual resources using the "manual" route type (page 23) in your routing.js, mapping it to a URI template ending in a wildcard:

```
app.routes = {
        '/*': 'manual'
}
```

The default configuration for "manual" will map all files from your application's "/resources/" subdirectory with the ".m." pre-extension. Here is the above configuration with all the defaults fleshed out:

```
app.routes = {
        '/*': {
                type: 'manual',
                root: 'resources',
                passThroughs: [],
                preExtension: 'm',
                trailingSlashRequired: true,
                internalUri: '/_manual/',
                clientCachingMode: 'conditional',
                maxClientCachingDuration: -1,
                compress: true
        }
}
```

General configuration for your code is done in setting.js (page 72).

handleInit

This is the only required entry point. It is called once for *every user request*, and *always before* any of the other entry points.

The main work is to initialize supported media types via the conversation.addMediaType APIs, in order of preference. For example:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('application/json')
        conversation.addMediaTypeByName('text/plain')
}
```

Note that you can also add language information:

```
function handleInit(conversation) {
        conversation.addMediaTypeByNameWithLanguage('text/plain', 'en')
        conversation.addMediaTypeByNameWithLanguage('text/plain', 'fr')
}
```

Prudence will use these values for content negotiation, choosing the best media type and language according to list of acceptable and preferred formats sent by the client and this list.

handleInit is also where you should set up caching (page 62), if you're using it:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('application/xml')
        caching.duration = '5s'
        caching.tags.add('blog')
}
```

Dynamic Content Negotiation You might wonder why we add these supported media types via API calls *for each request*, since they are usually always the same for a resource. Why not simply configure them into the resource permanently?

The reason is that they should not always permanent. In handleInit, you can check for various conditions of the conversation, or even external to the conversation, to decide which media types and languages to support. For example, you might not want to support XHTML for old browsers, but you'd want it at the top of the list for new browsers. Or, you might not be able to support PDF in case an external conversion service is down. In which case, you won't want it on the list at all, and instead want content negotiation to choose a different format that the client supports, such as DVI.

So, building your content negotiation table via API gives you a lot flexibility, at no real expense: these API calls are very lightweight.

Note that handleInit is called *even if your resource is cached* (on the server), exactly because you need to set up content negotiation before casting the cache key template (page 63).

handleGet

Handles HTTP "GET" requests.

In a conventional resource-oriented architecture (page 152), clients will not be expecting the resource to be altered in any way by a GET operation.

What you'll usually do here is construct a representation of the resource, possibly according to specific parameters of the request, and then return this representation to the client, possibly with directions for client-side caching (page 66).

There are many kinds of payloads you can return to the client: they are discussed in depth in the web data chapter (page 57). However, here's a reference of the supported return types:

- Numbers: Returns the number as an HTTP status code to the client. If you you wish to set your own return representation, too, you can use conversation.setResponseText or conversation.setResponseBinary. Note that if the status code is an error, then the error capturing (page 30) mechanism may override your response. If you wish to bypass this mechanism, set conversation.statusPassthrough to true.

- Arrays of bytes: Used for returning binary representations. Note that some languages (JavaScript, for example) have their own implementations of arrays, which are not compatible with JVM arrays. In such cases, you have to make sure to return JVM arrays. Internally, Prudence represents these values with a ByteArrayRepresentation.

- Representation instances: You can construct and return a directly.

- Other return values: If the conversation.mediaType is "application/internal" then the value will be wrapped in an InternalRepresentation. Otherwise, it will be converted into a string if it isn't a string already, and returned to the client as a StringRepresentation.

If your resource has been cached (on the server) then handleGet *will not be called.* The cache entry will be returned to the client instead.

Integrating Template Resources It's possible to return template resources as your payload. This could be useful if you're implementing the Model-View-Controller (MVC) pattern. See the MVC chapter (page 125) for complete examples.

handlePost

Handles HTTP "POST" requests.

In a conventional resource-oriented architecture (page 152), POST is the "update" operation (well, not exactly: see note below). Clients will expect the resource to already exist for the POST operation to succeed. That is, a call to GET before the POST may succeed. Clients expect you to return a modified representation, in the selected media type, if the POST succeeded. Subsequent GET operations would then return the same modified representation. A failed POST should not alter the resource: only a success status code should indicate a change. That means that ideally you should roll back changes if the entire operation fails along the way.

Note that the entity sent by the client does not have to be identical in format or content to what you return. In fact, it's likely that the client will send smaller delta updates in a POST, rather than a comprehensive representation.

What you'll usually do here is alter *existing* data according to data sent by the client.

> The most important thing to realize is that POST is the only HTTP operation that is not "idempotent," which means that multiple *identical* POST operations on a resource *may* yield results that are different from that of a single POST operation. This is why web browsers warn you if you try to refresh or go back to a web page that is the result of a POST operation. As such, POST is the correct operation to use for manipulations of a resource that *cannot be repeated.* So, if you're thinking in terms of CRUD, POST can mean *either* "update" or "create": it depends on whether or not "create" is a repeatable operation in your specific data semantics. See this blog post by John Calcote for one explanation.

Return value behavior is identical to that in handleGet (page 37). In fact, you may want handlePost to share the same code path as handleGet for creating the representation.

Note that the default return status for successful operations, 200 ("ok"), is indeed OK. However, it is better to return a 201 ("created") status if indeed the resource was created, and also return the full representation. If you cannot handle the operation at the moment, you should return a 202 ("accepted") status, signifying that the operation has been queued for later.

handlePut

Handles HTTP "PUT" requests.

In a conventional resource-oriented architecture (page 152), PUT is the "create" operation (well, not exactly: see note below). Clients will expect whatever current data exists in the resource to be discarded, and for you to return a representation of the new resource in the selected media type. A failed PUT should not alter the resource.

Note that the entity sent by the client does not have to be identical in format or content to what you return.

What you'll usually do here is parse and store the data sent by the client, *overwriting* data if it already exists.

> PUT, like most HTTP operations, is "idempotent," which means that multiple *identical* PUT operations on a resource are expected to yield the same result as a single PUT operation. PUT should thus *overwrite any existing data.* If you are implementing a "create" operation that *cannot* be repeated, then you should use POST instead. See note in POST (page 38).

See handleGet (page 37) for supported return types. In fact, you may want handlePut to share the same code path as handleGet for creating the representation.

Note that the default return status for successful operations, 200 ("ok"), is indeed OK. However, it is better to return a 201 ("created") status if indeed the resource was created, and also return the full representation. If you cannot handle the operation at the moment, you should return a 202 ("accepted") status, signifying that the operation has been queued for later.

handleDelete

Handles HTTP "DELETE" requests.

In a conventional resource-oriented architecture (page 152), clients expect subsequent GET operations to fail with a 404 ("not found") code. A DELETE should fail with 404 if the resource is not already there; it should *not* silently succeed. A failed DELETE should not alter the resource.

What you'll usually do here is make sure the identified resource exists, and if it does, remove or mark it somehow as deleted

The following return types are supported:

- Null: Signifies success.
- Number: Returns the number as an HTTP status code to the client, with no other content: for example, returning 404 means "not found." Note that error capturing (page 30) can let you take over and return an appropriate error page to the client.

  Though some languages return null if no explicit return statement is used, others return the value of the last executed operation, which could be a number, which would in turn become an HTTP status code for the client. This can lead to some very bizarre bugs, as clients receive apparently random status codes! It's thus good practice to always *explicitly* return null in handleDelete, if only to add clarity to your code's intent.

If you cannot handle the operation at the moment, you should return a 202 ("accepted") status, signifying that the operation has been queued for later.

handleGetInfo

Handles HTTP "GET" requests *before* handleGet (page 37).

This entry point, if it exists, is called before handleGet in order to provide Prudence with information required for conditional HTTP requests (page 65). Only if conditions are not met—for example if our resource is newer than the version the client has cached, or the tag has changed—does Prudence continue to handleGet. Using handleGetInfo can thus improve on the gains of conditional requests: not only are you saving bandwidth, but you are also avoiding a potentially costly handleGet call. Note that if the client is *not* doing a conditional request, then handleGetInfo will *not* be called.

The use of handleGetInfo discussed in detail in the caching chapter (page 67). However, for the sake of completion, here's a reference of the supported return types:

- Null: Means that you wish to continue directly to handleGet.
- Numbers or JVM Date instances: Considered as Unix timestamps, and converted into the modification date (page 66).
- Strings or Tag instances: Considered as HTTP tags (page 66).
- RepresentationInfo instances: Returned as is.

Note that even though you can only return the modification date *or* the tag, it is possible set both together by returning one and setting the other via the APIs (page 65).

If you implement handleGetInfo, you should be returning the same conditional information in your handleGet implementation, so that the client would know how to tag the data. The return value from handleGetInfo does not, in fact, ever get to the client: it is only used internally by Prudence to process conditional requests.

Template Resources

Though especially suitable for web pages (HTML), template resources can be used for any kind of textual asset. They are indeed intended and optimized for arbitrary textual formats: HTML, XML, plain text, etc. The design goal is to make it as easy as possible to generate the text dynamically, by allowing developers and designers to mix static elements with dynamic elements into a single file.

Using the default "scriptlets" parser, this means that the file is static text optionally embedded with delimited programming source code—these are the scriptlets. There are a few built-in shortcut scriptlets for common tasks, and it's also easy to write plugins to implement your own custom shortcuts.

The "scriptlets" parser is intended to be as straightforward and flexible as possible, mimicking the familiar paradigm of PHP/ASP/JSP, which allows for raw code that is executed in-place. However, it's possible to replace this parser with your own should you desire. For example, you might prefer declarative rather than procedural templates (mimicking the structure of HTML/XML), and rather than have opaquely delimited areas, you might prefer structural parsing of XML attributes. The parser system is pluggable and easy to extend.

Whatever parser you use, it will compile the whole file into one or more "programs" that are executed in sequence. Thus, unlike manual resources, template resources have no "entry points," and work in quite a different execution environment, and indeed each "program" can be in a different programming language, allowing for mixed-language templates.

Configuration

Add support for template resources using the "templates" route type (page 23) in your routing.js, mapping it to a URI template ending in a wildcard:

```
app.routes = {
        '/*': 'templates'
}
```

The default configuration for "templates" will map all files from your application's "/resources/" subdirectory with the ".t." pre-extension. Here is the above configuration with all the defaults fleshed out:

```
app.routes = {
        '/*': {
                type: 'templates',
                root: 'resources',
                includeRoot: ['libraries', 'includes'],
                passThroughs: [],
                preExtension: 't',
                trailingSlashRequired: true,
                defaultDocumentName: 'index',
                defaultExtension: 'html',
                clientCachingMode: 'conditional',
                maxClientCachingDuration: -1,
                compress: true
        }
}
```

General configuration for templates is in setting.js (page 73), as are the general configuration for code (page 72).

MIME Types and Compression

Prudence will handle HTTP content negotiation for your template resources, and will assume a single MIME type per resource. That MIME type is determined by the filename extension. For example, a resource named "profile.t.html" will have the "text/html" MIME type.

Prudence recognizes many common file types by default, but you can add your own mappings in you application's settings.js, using app.settings.mediaTypes (page 74).

When "compress" is true, Prudence will negotiate the best compression format (gzip and zip are supported) and compress on the fly. Compression can be configured in settings.js (page 73).

Scriptlets

By default, you can use either "<%...%>" (ASP/JSP-style) or "<?...?>" (PHP-style) scriptlet delimiters in your templates. Note that you can only use one or the other, though, in the same template.

The entire template is turned into a single program (or several programs if you are mixing languages: more on that below): the code between the delimiters is output as-is to the program, while the rest will be printed out. For example, the following template:

```
<% for (var x = 0; x < 10; x++) { %>
<p>This is a "line"</p>
<% } %>
```

Will become the following program behind the scenes:

```
for (var x = 0; x < 10; x++) {
print("<p>This is a \"line\"</p>")
}
```

You can see the programs generated behind the scenes by enabling debug mode (page 73).

> It's important to emphasize that *any* code can be used in templates: you can import libraries, define functions and classes, etc. It's up to you to decide how much and what kind of programming logic to use in templates. There are indeed strict coding disciplines, such as MVC, that forbid anything but visual logic in template. See the MVC chapter (page 125) for a complete discussion.

You can specify the language of the scriptlet in its opening delimiter:

```
<%ruby print 1 + 2 %>
```

Full language names are supported, as above, as well as shortcuts that are usually the common filename extensions for that language: "rb" for Ruby, "py" for Python, "js" for JavaScript, etc.

If the language of a scriptlet is not specified, it will default to the language of the previous scriptlet, so that you only need to specify the language if you are changing languages. If the first scriptlet does not specify a language, it will use the "defaultLanguageTag" setting (page 72). The default for that is JavaScript.

Built-In Shortcuts

Shortcuts use a special marker after the scriptlet's opening delimiter.

To output an expression:

```
<%= x*2 %>
```

The above is equivalent to:

```
<% print(x*2) %>
```

To include another template:

```
<%& '/header/' %>
```

The above is equivalent to:

```
<% document.include('/header/'); %>
```

A shortcut to print out conversation.base:

```
<%.%>
```

Scriptlet comments can be used for human-readable explanations, or to temporarily disable scriptlets:

```
<%# This scriptlet will be ignored by the parser. %>
```

All the above shortcuts work for all supported programming languages, generating source code in that language. To combine a language specification with a shortcut symbol, put the shortcut symbol *first*:

```
<%=rb x*2 %>
```

A few other built-in shortcuts are introduced under inheritance (page 42), and it's also possible create your own shortcuts (page 44).

Fragments

You can compose your templates out of reusable "fragments" by placing them in the "/libraries/includes/" directory, and then including them in other templates using the the "<%&...%>" shortcut (or the document.include API directly). For example, we can create this fragment in "/libraries/includes/lists/simple.t.html":

```
<li><%= line %></li>
```

And then use it like so:

```
<% for (int l in lines) { var line = lines[l] %>
<%& '/lists/simple/' %>
<% } %>
```

Fragments don't have to include scriptlets: they can be purely textual. On the other hand, they can contain sophisticated code to generate their HTML. Fragments can also be nested to any depth: a fragment can include others, those can include others, and so on.

> Fragments in Prudence are not merely "server-side includes": in fact, each fragment is cached separately, with its own cache key and cache duration. This allows you to create sophisticated and extremely efficient fine-grained caching strategies. See the caching chapter (page 61).

To share fragments with *all* applications, put them in your container's "/libraries/prudence-includes/" directory. Files in your application's "/libraries/includes/" will always override the shared versions.

Blocks and Inheritance

This feature is very similar to fragments, but can be understood as its inverse: rather than inserting a fragment into the current location using "<%&...%>", you define blocks that will *only later* be inserted into a template. Unlike fragments, blocks cannot be individually cached, but they can otherwise provide additional flexibility in assembling your page. It's a good idea to use both: blocks when you need design flexibility and fragments when you need fine-grained caching.

There are special scriptlets for blocks. As an example, let's define a block and then include the actual template:

```
<%{ title %>
        <h1>My Title</h1>
<%}%>
<%& '/templates/main/' %>
```

The block definition is enclosed in "<%{...%>" and "<%}%>" scriptlets: instead of being printed out to the page, it will be saved for later use. Indeed, the template above generates no actual output in itself.

Now, let's see the actual template, in "/includes/templates/main.t.html":

```
<html>
<body>
<%== title %>
</body>
</html>
```

The "<%==...%>" scriptlet is a shortcut to print any conversation.local: indeed, our block was simply a temporary capturing of output into a conversation.local. Behind the scenes, blocks are simply calls to document.startCapture and document.endCapture.

But what if you want to give the block a default value in the template? There are special scriptlets for that, too. Let's change our "main.t.html":

```
<html>
<body>
<%[ title %>
        <h1>Default Title</h1>
<%]%>
</body>
</html>
```

The "<%[...%>" and "<%]%>" scriptlets are like a conditional "<%==...%>": the code between them will *not* be executed if the conversation.local is already defined. If it *is* defined, then it will simply be printed out.

Blocks with Arguments You may need blocks that change their content according to external parameters. Nothing magical about these: they are simply functions! For example:

```
<% function header1(content) { %>
<h1><%= content %></h1>
<% } %>
```

To use it, just call it:

```
<% header1('Hello'); %>
```

You can send block content as an argument to the function:

```
<%{ myArgument %>
<a href="/">This HTML code will be sent as an argument</a>
<%}%>
<% header1(conversation.locals.get('myArgument')); %>
```

Templating Languages

Scriptlets can be used not only with programming languages, but also with templating languages. Succinct and Velocity are both supported. Here's an example using Velocity and JavaScript together:

```
<%js
conversation.locals.put('test', 'TEST')
%>
<%velocity
#macro(hello $name)
	<div>Hello, $name!</div>
#end
<div>
	#hello('Mozart')
	#hello('Bach')
	Here is a conversation local: $!conversation.locals.test
</div>
%>
```

As with programming languages, you will need to install the engine first:

```
sincerity add velocity : install
```

You might prefer to use templating engines independently of template resources, as MVC "views." See the MVC chapter (page 127) for a guide.

HTML Markup Languages

You can also render the following HTML markup language via scriptlets: Markdown, Confluence, MediaWiki, TWiki, Trac and Textile. Here's a Markdown example:

```
<html>
<body>
<%md

Our Conference
==============

* The first item of business
* The second item of business

%>
</body>
</html>
```

As with programming languages, you will need to install the engine first:

```
sincerity add org.pegdown pegdown : install
```

Here's a list of package identifiers for all supported languages (note there are two implementation options for Markdown):

Language	Identifier
Markdown	org.pegdown pegdown
Markdown	org.eclipse.mylyn wikitext-markdown
Confluence	org.eclipse.mylyn wikitext-confluence
MediaWiki	org.eclipse.mylyn wikitext-mediawiki
TWiki	org.eclipse.mylyn wikitext-twiki
Trac	org.eclipse.mylyn wikitext-trac
Textile	org.eclipse.mylyn wikitext-textile

PHP

Of all the supported programming languages, PHP is special in that it already has a scriptlet parser. In Prudence, PHP code will work pretty much as is, as it mimics the PHP format. You actually have the choice of using the standard PHP-style delimiters, "<?. . . ?>", or the ASP/JSP-style delimiters instead: "<%. . . %>".

For example:

```
<?php
for($i = 0; $i < 10; $i++) {
          print '<p>' . $i . '</p>';
}
?>
```

PHP is special also in that it is designed for server-side web programming. Thus, though you can use all of Prudence's APIs in PHP, Prudence also explicitly supports many of PHP's predefined variables as a more standard alternative:

```
<?php
print $_GET['id']
?>
```

The convention when programming in PHP is to use ".php" extensions for files, even though the MIME type is "text/html". This is easy to achieve in Prudence by using a ".t.php" for your files, while also mapping the extension to the MIME type in your settings.js (page 74):

```
app.settings = {
          ...
          mediaTypes: {
                    php: 'text/html'
          }
}
```

Scriptlet Plugins

The default "scriptlets" parser comes with various useful shortcuts, but it's easy to create your own.

Configure your scriptlet plugins in settings.js (page 73) using a dict that maps the shortcut codes to the library that will handle them. For this tutorial, we'll define two shortcuts in the same library:

```
app.settings = {
          ...
          templates: {
                    plugins: {
                              '_': '/plugins/custom/',
                              '&?': '/plugins/custom/'
                    }
```

```
		}
}
```

The "_" shortcut will be used to print out localized text strings. The "&?" shortcut will be a conditional include. Let's now implement them in "/libraries/plugins/custom.js":

```
function handleGetScriptlet(code, languageAdapter, content) {
	if (code == '_') {
		return "print(application.globals.get('text.' + conversation.locals.get('loca
	}
	else if (code == '&?') {
		return "if(conversation.locals.get('include')===true) { document.include(" +
	}
	return ''
}
```

As you can see, the "handleGetScriptlet" function returns JavaScript source code that will be embedded into the generated program. You can optionally use the content of the scriptlet via the "content" param. If you wish to support multiple programming languages, you can test for them using "languageAdapter.attributes.get('language.name')".

Let's now use our shortcuts in a template resource:

```
<%
conversation.locals.put('locale', 'en')
conversation.locals.put('include', false)
%>
<p>
	How to say "hello" in your language: <%_ basic.hello %>
</p>
<p>
	This fragment will not be included: <%&? '/hello/' %>
</p>
```

Our "_" shortcut expects certain application.global definitions, so let's define them in our settings.js (page 75):

```
app.globals = {
	text: {
		en: {
			basic: {
				hello: 'Hi!'
			}
		},
		es: {
			basic: {
				hello: '¡Hola!'
			}
		}
	}
}
```

Some things to note about scriptlet plugins:

- It's up to you, of course, to make sure that the code you generate is compilable.
- Plugins are tested for *before* the built-in shortcuts, allowing you to override the built-in ones.
- If you change your plugin code, it will *not* cause all template resources that use it to recompile. To force a recompile, you will need to change the modification date of those files, possibly by using the "touch" tool (on *nix).

Static Resources

Prudence works fine as a static web server: it's fast, supports non-blocking chunking, and has many useful features detailed below.

Of course, there are servers out there that specialize in serving static files and might do a better job, but you might be surprised by how far Prudence can take you.

Note that if Internet scalability is really important to you, it's better to even not use a standard web server at all, but instead rely on a CDN (Content Delivery Network) product or service with true global reach.

Configuration

Add support for static resources using the "static" route type (page 22) in your routing.js, mapping it to a URI template ending in a wildcard:

```
app.routes = {
        '/*': 'static'
}
```

The default configuration for "static" will map all files from your application's "/resources/" subdirectory, *as well as* the container's "/libraries/web/" directory. Here is the above configuration with all the defaults fleshed out:

```
app.routes = {
        '/*': {
                type: 'static',
                roots: [
                        'resources',
                        sincerity.container.getLibrariesFile('web')
                ],
                listingAllowed: false,
                negotiate: true,
                compress: true
        }
}
```

Note that the "roots" (pluralized) param is a shortcut to create a chain of two "static" instances. The above is equivalent to:

```
app.routes = {
        '/*': [
                {type: 'static', root: 'resources'},
                {type: 'static', root: sincerity.container.getLibrariesFile('web')}
        ]
}
```

If you want to also support manual and template resources, make sure to chain "static" after them, so it will catch whatever doesn't have the special ".m." and ".t." pre-extensions:

```
app.routes = {
        '/*': [
                'manual',
                'templates',
                'static'
        ]
}
```

MIME Types and Compression

When "negotiate" is true, Prudence will handle HTTP content negotiation for your static resources, and will assume a single MIME type per resource. That MIME type is determined by the filename extension. For example, a resource named "logo.png" will have the "image/png" MIME type.

Prudence recognizes many common file types by default, but you can add your own mappings in you application's settings.js, using app.settings.mediaTypes (page 74).

When "compress" is *also* true, Prudence will negotiate the best compression format (gzip and zip are supported) and compress on the fly. Compression can be configured in settings.js (page 73).

Client-Side Caching

Prudence adds modification timestamp headers to all static resources, which allow clients, such as web browsers, to cache the contents and use conditional HTTP requests to later check if the cache needs to be refreshed.

Conditional HTTP is efficient and fast, but you can go one step further and tell clients to avoid even that check. Use the "cacheControl" filter (page 24) before your "static" route type:

```
app.routes = {
        '/*': {
                type: 'cacheControl',
                mediaTypes: {
                        'image/*': '10m',
                        'text/css': '10m',
                        'application/x-javascript': '10m'
                },
                next: 'static'
        }
}
```

With the above, Prudence will ask web browsers to cache common image types, CSS and JavaScript for 10 minutes before sending conditional HTTP requests.

Make sure you understand the implications of this: after the client's first hit, for 10 minutes *it will not be able to see changes to that static resource.* The client's web browser would continue using the older version of the resource until its cache expires.

For a full discussion of client-side caching, see the caching chapter (page 65).

Bypassing the Client Cache There is a widely-used trick that lets you use client-side caching while still letting you propagate changes *immediately.* It makes use of the fact that the client cache uses the *complete* URL as the cache key, which *includes the query matrix.* If you use a query param with the URL, the "static" resource will ignore it, but the client will still consider it a new resource in terms of caching. For example, let's say you include an image in an HTML page:

```
<img src="/media/logo.png" />
```

If you made a change to the "logo.png" file, and you want to bypass the client cache, then just change the HTML to this:

```
<img src="/media/logo.png?_=1" />
```

Voila: it's a new URL, so older cached values will not be used. For Prudence, the query makes no difference. You can then simply increase the value of the "_" query param every time you make a change.

This trick works so well that, if you use it, it's recommended that you actually ask clients to cache these resources *forever.* "Forever" is not actually supported, but it's customary to use 10 years in the future as a practical equivalent. Use "farFuture" as a shortcut in "cacheControl":

```
app.routes = {
        '/*': {
                type: 'cacheControl',
                mediaTypes: {
                        'image/*': 'farFuture'
                },
                next: 'static'
        }
}
```

Remembering to increase the query param in all uses of the resource might be too cumbersome and error-prone. Consider using the Diligence Assets service, or something similar, instead: it calculates digests for the resource file contents and use them as the query param. Thus, any change to the file contents will result in a new, unique URL.

> What happens to cache entries that have been marked to expire in 10 years, but are no longer used by your site? They indeed will linger in your client's web browser cache. This isn't too bad: web browsers normally are configured with a maximum cache size and the disk space will be cleared and reused if needed. It's still an inelegant waste, for which unfortunately there is no solution in HTTP and HTML.

JavaScript and CSS Optimization

When writing JavaScript code, you likely want to use a lot of spacing, indentation and comments to keep the code clear and manageable. You would likely also want to divide a large code base among multiple files. Unfortunately, this is not so efficient, because clients must download these files.

Prudence's "javaScriptUnifyMinify" filter (page 24) can help. To configure:

```
app.routes = {
        '/*': {
                type: 'javaScriptUnifyMinify',
                next: 'static'
        }
}
```

The filter will catch URLs ending in "/all.js" or "/all.min.js", the former being unified and the latter unified *and* minified. The contents to be used, by default, will be all the ".js" files under your application's "/resources/scripts/" subdirectory *as well as* those under your container's "/libraries/web/scripts/" directory. The filter writes out the generated "all.js" and "all.min.js" files to "/resources/scripts/", and makes sure to update these files (unifying and minifying again) if any one of the source files are changed.

Note that the files are unified in alphabetical order, so make sure to rename them accordingly if order of execution is important.

Here is the above configuration with all the defaults fleshed out:

```
app.routes = {
        '/*': {
                type: 'javaScriptUnifyMinify',
                roots: [
                        'resources/scripts',
                        sincerity.container.getLibrariesFile('web', 'scripts')
                ],
                next: 'static'
        }
}
```

For a usage example, let's say we have the following three files under the application's subdirectory:

```
/resources/scripts/jquery.js
/resources/scripts/jquery.highlight.js
/resources/scripts/jquery.popup.js
```

Your HTML files can then include something like this:

```
<head>
        ...
        <script src="/scripts/all.min.js"></script>
</head>
```

Note that the first entry in the "roots" array is where the generated "all.js" and "all.min.js" files are stored.

The "cssUnifyMinify" filter (page 24) does the same for CSS files, with the default roots being the application's "/resources/style/" subdirectory and the container's "/libraries/web/style/" directory. The relevant files are "all.css" and "all.min.css". Note, however, that similar functionality is provided by using LESS (page 49).

Here's an example with both filters configured:

```
app.routes = {
	'/*': {
		type: 'javaScriptUnifyMinify',
		next: {
			type: 'cssUnifyMinify',
			next: 'static'
		}
	}
}
```

Usage in HTML:

```
<head>
	...
	<link rel="stylesheet" type="text/css" href="/style/all.min.css" />
</head>
```

LESS to CSS

LESS is an extended CSS language. It greatly increases the power of CSS by allowing for code re-usability, variables and expressions, as well as nesting CSS. Using the the "less" filter (page 24) you can compile ".less" files to CSS, and also apply the same minifier used by "cssUnifyMinify" (page 24).

To configure:

```
app.routes = {
	'/*': {
		type: 'less',
		next: 'static'
	}
}
```

The filter works by catching all URLs ending in ".css" or ".min.css". It will then try to find an equivalent ".less" file, and if it does, will compile it and produce the equivalent ".css" or ".min.css" file. It makes sure to recompile if the source ".less" file is changed.

For a usage example, let's say we have a "/resources/style/dark/main.less" file the application's subdirectory. Your HTML files can then include something like this:

```
<head>
	...
	<link rel="stylesheet" type="text/css" href="/style/dark/main.min.css" />
</head>
```

The "less" filter can be configured with a "roots" param similarly to the "javaScriptMinifyUnify" and "cssUnifyMinify" (page 48).

See the FAQ (page 100) as to why Prudence supports LESS but not SASS.

Other Effects

Because static resources don't allow for code (unlike manual and template resources), the way to program your own effects is to add filters (page 108).

For example, let's say we want to force all ".pdf" files to be downloadable (by default, web browsers might prefer to display the file using a browser plugin). We'll be using the technique described here (page 60) to change the response's "disposition".

Here's our filter code, in "/libraries/filters/download-pdf.js":

```
function handleAfter(conversation) {
	var entity = conversation.response.entity
	if (entity.mediaType.name == 'application/pdf') {
		entity.disposition.type = 'attachment'
	}
}
```

To install it in routing.js:

```
app.routes = {
        '/*': {
                type: 'filter',
                library: '/filters/download-pdf/',
                next: 'static'
        }
}
```

On-the-Fly Resources

You can add support for on-the-fly resources to your application via the "execute" route type (page 22). This powerful—and dangerous—resource executes all POST payloads as if they were template resources in the application, and is very useful for debugging (page 87) and maintenance.

Another possibly exciting use case is to allow an especially rich API: instead of exposing your facilities via a RESTful API, you can allow certain clients full access to, well, everything. It's hard to recommend this usage for most applications due to its severe security risks, as well as limitations to scalability, but for some internally running applications it could prove extremely useful.

> Diligence comes with the console feature, which offers a user-friendly variation of this functionality: it's a web-based mini-IDE that features persistent programs, JavaScript syntax coloring, and log tailing/-filtering.

To install it, modify your application's routing.js and create a route for the "execute" type:

```
app.routes = {
        ...
        '/execute/': 'execute'
}
```

> Because it allows execution of arbitrary code, you very likely do not want its URL publicly exposed. Make sure to protect its URL on publicly available machines!

Example use with cURL command line:

```
curl --data '<% println(1+2) %>' http://localhost:8080/myapp/execute/
```

Note that if you use cURL with a file, you need to send it as binary, otherwise curl will strip your newlines:

```
curl --data-binary @myscriptfile http://localhost:8080/myapp/execute/
```

Where "myscriptfile" could be something like this:

```
<%
document.require('/sincerity/templates/')
println(Hello, {0}'.cast('Linus'))
%>
```

Almost all the usual template resource APIs work (with the exception of caching, which isn't supported):

```
<%
document.require('/sincerity/templates/')
var name = conversation.query.get('name') || 'Linus'
println('Hello, {0}'.cast(name))
%>
```

For the above, you could then POST with a query param:

```
curl --data-binary @myscriptfile 'http://localhost:8080/myapp/execute/?name=Richard'
```

Note that you can, as usual, use scriptlets in any supported programming language:

```
<%python
name = conversation.query['name'] or 'Linus'
print 'Hello, %s' % name
%>
```

Also note that the default response MIME type is “text/plain”, but you can modify it with the conversation.mediaType APIs:

```
<%
document.require('/sincerity/json/')
conversation.mediaTypeName = 'application/json'
println(Sincerity.JSON.to({greeting: 'Hello'}, true))
%>
```

Java Resources

You may prefer to implement some of your manual resources directly in Java, by inheriting ServerResource, or perhaps you are relying a JVM library that already comes with such resources. To use them, set them up in your routing.js via the “resource” route type (page 22).

Some notes:

- Sure, Java can offer better performance than dynamic languages, but it’s very rare for language performance to be the bottleneck: communication with a database server, for example, is orders of magnitude slower. Don’t “drop down” to Java in order to optimize unless you’ve really confirmed that language performance is a problem.
- Prudence’s manual resources are not merely dynamic-language implementations of ServerResource, but also provide extra features such as integrated server-side caching (page 61). If you switch to Java, you would have to implement such features on your own.

Resource Type Comparison Table

	Manual	**Template**	**Static**
Supports URI Mapping	Yes	Yes	Yes
Supports URI Dispatching	Yes	No	No
Filename Extension	Determines programming language	Determines MIME type	Determines MIME type
Filename Pre-extension	*.m.*	*.t.*	n/a
Programming Languages	Determined by filename extension	Determined by scriptlet tags (multiple languages possible per resource)	n/a
Content Negotiation	Manually determined in handleInit; multiple MIME types possible; multiple compression types automatically supported and cached	Single MIME type determined by filename extension; multiple compression types automatically supported and cached	Single MIME type determined by filename extension; multiple compression types automatically supported
Server-Side Caching	Manual (via API)	Automatic (determined by server-side caching)	n/a
Client-Side Caching	Manual (via API)	Automatic (determined by server-side caching)	Can be added with CacheControlFilter

Web Data

This chapter deals with sending and receiving data to and from the client (as well as external servers) via REST, focusing especially on the particulars for HTTP and HTML. It does *not* deal with storing data in backend databases.

Prudence is a minimalist RESTful platform, *not* a data-driven web framework, though such frameworks are built on top of it. Check out our Diligence, which is a full-blown framework based on Prudence and MongoDB. You may also be interested in the Model-View-Controller (MVC) chapter (page 125), which guides you through an approach to integrating data backends.

URLs

The simplest way in which a client sends data to the server is via the URL. The main part of the URL is parsed by Prudence and used for routing (page 21), but some of it is left for your own uses.

Generally, the whole or parts of the request URL can be accessed via the conversation.reference API.

Query Parameters

This is the matrix of parameters after the "?" in the URI.

For example, consider this URL:

```
http://mysite.org/myapp/user?name=Albert%20Einstein&enabled=true
```

Note the "%20" URI encoding for the space character. Query params will be automatically decoded by Prudence.

In JavaScript, you can use Prudence.Resources.getQuery API:

```
document.require('/prudence/resources/')

var query = Prudence.Resources.getQuery(conversation, {
        name: 'string',
        enabled: 'bool'
})
```

In the case of multiple params with the same name, the API would return the first param that matches the name. Otherwise, you can also retrieve all values into an array:

```
var query = Prudence.Resources.getQuery(conversation, {
        name: 'string[]',
        enabled: 'bool'
})
```

Low Level For non-JavaScript you can use the lower-level conversation.query API:

```
var query = {
        name: conversation.query.get('name'),
        enabled: conversation.query.get('enabled') == 'true'
}
```

Use conversation.queryAll if you need to find multiple params with the same name.

Captured Segments

Variables in the URI template (page 21) you configured in routing.js will be captured into conversation.locals (page 83). Note that you can also interpolate the captured variables into the target URI (page 115).

The Wildcard

If you've configured a URI template with a wildcard (page 21) in routing.js, you can access the "*" value using conversation.wildcard. Note that you can also interpolate the wildcard into the target URI (page 114).

Fragments

This is whatever appears after the "#" in the URI. Note that for the web *fragments are only used for response URLs*: those sent from the server to the client. This is enforced: web browsers will normally *strip fragments* before sending URLs to the server, but the server can send them to web browsers. They are commonly used in HTML anchors:

```
<a name="top" /><h1>This is the top!</h1>
<p>Click <a href="#top">here</a> to go to the top</p>
```

But you can also use them in redirects (page 55):

```
conversation.redirectSeeOther(conversation.base + '#top')
```

Request Payloads

These are used in "POST" and "PUT" verbs.

In JavaScript, you can use Prudence.Resources.getEntity API to extract the data in various formats:

```
document.require('/prudence/resources/')

var data = Prudence.Resources.getEntity(conversation, 'json')
```

Otherwise, you can use the lower-level conversation.entity API:

```
document.require('/sincerity/json/')

var text = conversation.entity.text
var data = Sincerity.JSON.from(text)
```

Note that if the payload comes from a HTML "post" form, better APIs are available (page 56).

MIME Types

If you wish to support multiple request payload MIME types, be sure to check before retrieving:

```
var type = conversation.entity.mediaType.name
if (type == 'application/json') {
        var data = Prudence.Resources.getEntity(conversation, 'json')
        ...
} else if (type == 'image/png') {
        var data = Prudence.Resources.getEntity(conversation, 'binary')
        ...
}
```

Consumption

Note that *you can only retrieve the request payload once.* Once the data stream is consumed, its data resources are released. Thus, the following would result in an error:

```
print(conversation.entity.text)
print(conversation.entity.text)
```

The simple solution is retrieve once and store in a variable:

```
var text = conversation.entity.text
print(text)
print(text)
```

Parsing Formats

Which formats can Prudence parse, and how well?

This depends on which programming language you're using: for example, both Python and Ruby both come with basic JSON support in their standard libraries, and Python supports XML, as well. Sincerity provides JavaScript with support for both. Of course, you can install libraries that handle these and other formats, and even use JVM libraries.

For other formats, you may indeed need to add other libraries.

A decent starting point is Restlet's ecosystem of extensions, which can handle several data formats and conversions. However, these are likely more useful in pure Java Restlet programming, where they can plug into Restlet's sophisticated annotation-based conversion system. In Prudence, you will usually be applying any generic parsing library to the raw textual or binary data. Still, the Restlet extensions are useful for response payloads (page 58).

Cookies

Cookies represent a small client-side database, which the server can use to retrieve or store per-client data. Not all clients support cookies, and even those that do (most web browsers) might have the feature disabled, so it's not always a good idea to rely on cookies.

From the Client

Retrieve a specific cookie from those the client sent you according to its name using conversation.getCookie:

```
var session = conversation.getCookie('session')
if (null !== session) {
	print(session.value)
}
```

Or use conversation.cookies to iterate through all available cookies.
The following attributes are available:

- name: (read only)
- version: (integer) per a specific cookie
- value: textual, or text-encoded binary data (note that most clients have strict limits on how much total data is allowed to be stored in all cookies per domain)
- domain: the client should only use the cookie with this domain and its subdomains (web browsers will not let you set a cookie for a domain which is not the domain of the request or a subdomain of it)
- path: the client should only use the cookie with URIs that begin with this path ("/", the default, would mean to use it with all URIs)

To the Client

You can ask that a client modify any of the cookies you've retrieved *from* it, upon a successful response, by calling the "save" method:

```
var session = conversation.getCookie('session')
if (null !== session) {
	session.value = 'newsession'
	session.save()
}
```

Ask the client to create a new cookie using conversation.createCookie:

```
var session = conversation.createCookie('session')
session.value = 'newsession'
session.save()
```

Note that createCookie will retrieve the cookie if it already exists.

When sending cookies *to* the client, you can set the following attributes *in addition* to those mentioned above, but note that you *cannot* retrieve them later:

- maxAge: age in seconds, after which the client should delete the cookie; maxAge=0 deletes the cookie immediately, while maxAge=-1 (the default) asks the client to keep the cookie only for the duration of the "session" (this is defined by the client; for most web browsers this means that the cookie will be deleted when the browser is closed)

- secure: true if the cookie is meant to be used only in secure connections (defaults to false)
- accessRestricted: true if the cookie is meant to be used only in authenticated connections (defaults to false)
- comment: some clients store this, some discard it

You can ask the client to delete a cookie by calling its "remove" method. This is identical to setting maxAge=0 and calling "save".

Security Concerns

You should never store any unencrypted secret data in cookies: though web browsers attempt to "sandbox" cookies, making sure that only the server ("domain") that stored them can retrieve them, they can be hijacked by other means. Better yet, don't store *any* secrets in cookies, even if encrypted, because even encryptions can be hacked. A cautious exception can be made for *short-term* secrets: for example, if you store a session ID in a cookie, make sure to expire it on the server so that it cannot be used later by a hacker.

A separate security concern for users is that cookies can be used to surreptitiously track user activity. This works because any resource on a web page—even an image hosted by an advertising company—can use cookies, and can also track your client's IP address. Using various heuristics it is possible to identify individual users and track parts of their browser history.

Because of these security concerns, it is recommended that you devise a "cookie policy" for users and make it public, assuming you require the use of cookies for your site. In particular, let users know which 3rd-party resources you are including in your web pages that may be storing cookies, and for what purpose.

Cookies are a security concern for you, too: you cannot expect all your clients to be standard, friendly web browsers. Clients might not be honoring your requests for cookie modifications, and might be sending you cookies that you did not ask them to store.

Be careful with cookies! They are a hacker's playground.

Custom Headers

The most commonly used request and response HTTP headers are supported by Prudence's standard APIs. For example: conversation.disposition, conversation.maxAge and conversation.client. However, Prudence also let's you use other headers, including your custom headers, via conversation.requestHeaders and conversation.responseHeaders. Note that you *must* use Prudence's standard APIs for headers if such exist: these APIs will only work for *additional* headers.

These APIs both return a Series object. (You usually won't need to access the elements directly, but in case you do: they are Header objects.) An example of fetching a request header:

```
var host = conversation.requestHeaders.getFirstValue('Host')
```

An example of setting a response header:

```
conversation.responseHeaders.set('X-Pingback', 'http://mysite.org/pingback/')
```

Redirection

Client-side redirection in HTTP is handled via response headers.

By Routing

If you need to constantly redirect a specific resource or a URI template, you should configure it in your routing.js, using the "redirect" route type (page 23):

```
app.routes = {
        ...
        '/images/*': '>/media/{rw}'
}
```

Note that in this example we interpolated the wildcard (page 114).

By API

You can also redirect programmatically by using the conversation.redirectPermament, conversation.redirectSeeOther or conversation.redirectTemporary APIs:

```
conversation.redirectSeeOther(conversation.base + '/help/')
```

Note that if you redirect via API, the client will ignore the response payload if there is one.

In HTML

We're mentioning this here only for completion: via HTML, redirection is handled entirely in the web browser, with no data going to/from the server. A template resource example:

```
Go <a href="<%.%>/elsewhere/">elsewhere</a>.
```

Server-Side Redirection

In Prudence, this is called "capturing" (page 28) and has particular use cases. (It can indeed be confusing that this functionality is often grouped together with client-side redirection.)

HTML Forms

HTML's "form" tag works in two very different modes, depending on the value of its "method" param:

- "get": The form fields are all turned into query params and appended to the "action" URL. It is important to remember that an HTTP "GET" is *idempotent* and should not be used to store new data, but rather as a way to represent existing data in a particular way. Actually, "get" forms are not that useful, and are mostly an odd legacy from the early days of the World Wide Web, where "GET" was the only the supported HTTP verb on some platforms. See query parameters (page 52) for handling.
- "post": The form fields are actually encoded in the same way that query params are, but instead of being affixed to the URL, they are sent as the payload with an "application/x-www-form-urlencoded" MIME type. Though you can of course access this payload directly (page 53), it is recommended to use the specialized APIs detailed here.

Example form:

```
<form action="<%.%>/user/" method="post">
        <p>Name: <input type="text" name="name"></p>
        <p>Enabled: <input type="radio" name="enabled" value="true"></p>
        <p>Disabled: <input type="radio" name="enabled" value="false"></p>
        <p><button type="submit">Send</button></p>
</form>
```

In JavaScript, you can use Prudence.Resources.getForm API:

```
document.require('/prudence/resources/')

var form = Prudence.Resources.getForm(conversation, {
        name: 'string',
        enabled: 'bool'
})
```

In the case of multiple fields with the same name, the API would return the first fields that matches the name. Otherwise, you can also retrieve all values into an array:

```
var form = Prudence.Resources.getForm(conversation, {
        name: 'string[]',
        enabled: 'bool'
})
```

Low Level For non-JavaScript you can use the lower-level conversation.form API family:

```
var form = {
        name: conversation.form.get('name'),
        enabled: conversation.form.get('enabled') == 'true'
}
```

Use conversation.formAll if you need to find multiple fields with the same name.

Accepting Uploads

HTML supports file uploads using forms and the "file" input type. However, the default "application/x-www-form-urlencoded" MIME type for forms will not be able to encode files, so you must change it to "multipart/form-data". For example:

```
<form action="<%.%>/user/" method="post" enctype="multipart/form-data">
        <p>Name: <input type="text" name="name"></p>
        <p>Upload your avatar (an image file): <input name="avatar" type="file" /></p>
        <p><button type="submit">Send</button></p>
</form>
```

Prudence has flexible support for handling uploads: you can configure them to be stored in memory, or to disk. See the application configuration guide (page 73).

You can access the uploaded data using the conversation.form API family. Here's a rather sophisticated example for displaying the uploaded file to the user:

```
<%
var name = conversation.form.get(name')
var tmpAvatar = conversation.form.get('avatar').file

// The metadata service can provide us with a default extension for the media type
var mediaType = conversation.form.get('avatar').mediaType
var extension = application.application.metadataService.getExtension(mediaType)

// We will put all avatars under the "/resources/avatars/" directory, so that they
// can be visible to the world
var avatars = new File(document.source.basePath, 'avatars')
avatars.mkdirs()
var avatar = new File(avatars, name + '.' + extension)

// Move the file to the new location
tmpAvatar.renameTo(avatar)
%>
<p>Here's the avatar you uploaded, <%= name %></p>
<img src="<%.%>/avatars/<%= avatar.name %>" />
```

Response Payloads

This section is mostly applicable to manual resources (page 36), although it can prove useful to affect the textual payloads of template resources (page 39). For static resources (page 46), the response payloads are of course the contents of the resource files.

Two important notes:

- Prudence can automatically cache your response payloads (page 61). Upon a successful cache hit, Prudence will in fact bypass execution of (most of) your code.

- If you are using your resources internally, it's possible to improve performance by avoiding serialization (page 116).

Textual and Binary Payloads

Template resources (page 39) might seem to always return textual payloads. Actually, by default they will negotiate a compression format, which if selected will result in a binary: the compressed version of the text. But all of that is handled automatically by Prudence for that highly-optimized use case.

For manual resources (page 36), you can return any arbitrary payload by simply returning a value in handleGet, handlePost or handlePut. Both strings and JVM byte arrays are supported. A textual example:

```
function handleGet(conversation) {
        return 'My payload'
}
```

A binary example:

```
document.require('/sincerity/jvm/')

function handleGet(conversation) {
        var payload = Sincerity.JVM.newArray(10, 'byte')
        for (var i = 0; i < 10; i++) {
                payload[i] = i
        }
        return payload
}
```

Note that if you return a *number*, it will be treated specially as an HTTP status code. If you wish to return the number as the content of a *textual payload*, simply convert it to a string:

```
function handleGet(conversation) {
        return String(404)
}
```

If you wish to set both the payload *and* the status code, use an API for either one. Here well use the conversation.status API family. Note that if your status code is an error status code, you'll also want to bypass this error page using conversation.statusPassthrough:

```
function handleGet(conversation) {
        conversation.statusCode = 404
        conversation.statusPassthrough = true
        return 'Not found!'
}
```

Alternatively, we can use the conversation.setResponseText or conversation.setResponseBinary:

```
function handleGet(conversation) {
        conversation.setResponseText('Not found!', null, null, null)
        conversation.statusPassthrough
        return 404
}
```

Binary Streaming Streaming using background tasks (page 102) is not directly supported by Prudence as of version 2.0. However, this feature is planned for a future version, depending on support being added to Restlet.

Restlet Data Extensions

Instead of returning a string or a byte array, you can return an instance of any class inheriting from Representation. Restlet comes with a few basic classes to get you started. Here's a rather boring example:

```
function handleGet(conversation) {
        return new org.restlet.representation.StringRepresentation('My payload')
}
```

Where Restlet really shines is in its ecosystem of extensions, which can handle several data formats and conversions. For these extensions to work, you will need to install the appropriate library in your container's "/libraries/jars/" directory, as well as all dependent libraries. Please refer to the Restlet distribution for complete details.

Note that you can also set the response via the conversation.response.entity API:

```
var payload = new org.restlet.representation.StringRepresentation('My payload')
conversation.response.entity = payload
```

Or via the conversation.setResponseText API shortcut:

```
conversation.setResponseText('My payload', null, null, null)
```

Overriding the Negotiated Format

The response payload's MIME type and language have likely been selected for you automatically by Prudence, via HTTP content negotiation, based on the list of preferences you set up in handleInit. However, it's possible to override these values via the conversation.mediaType and conversation.language API families. This should be done sparingly: content negotiation is the preferred RESTful mechanism for determining the response format, and the negotiated values should be honored. However, it could be useful and even necessary to override it if you *cannot* use content negotiation, which might be the case if your clients don't support it, and yet you still want to support multiple formats.

In this example, we'll allow a "format=html" query param to override the negotiated MIME type:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('text/html')
        conversation.addMediaTypeByName('text/plain')
}

function handleGet(conversation) {
        if (conversation.query.get('format') == 'html') {
                conversation.mediaTypeName = 'text/html'
        }
        return conversation.mediaTypeName == 'text/html' ?
                '<html><body>My page</body></html>' :
                'My page'
}
```

An example of overriding the negotiated language:

```
function handleInit(conversation) {
        conversation.addMediaTypeByNameWithLanguage('text/html', 'en')
        conversation.addMediaTypeByNameWithLanguage('text/html', 'fr')
}

function handleGet(conversation) {
        if (conversation.query.get('language') == 'fr') {
                conversation.languageName = 'fr'
        }
        if (conversation.languageName == 'fr') {
                ...
        }
        else {
                ...
        }
}
```

Note that these APIs works just as well for template resources, though again content negotiation should be preferred.

Under the Hood When the MIME type is "application/internal", Prudence is actually wrapping your return value in an InternalRepresentation. You can also construct it explicitly (page 58):

```
return new com.threecrickets.prudence.util.InternalRepresentation(data)
```

Note that, of course, if you return an instance of a class inheriting from Representation, Prudence will detect this and not wrap it again in an InternalRepresentation.

Browser Downloads

You can create browser-friendly downloadable responses using the conversation.disposition API. Here's an example using a manual resource:

```
function handleInit(conversation) {
        conversation.addMediaTypeByName('text/csv')
}

function handleGet(conversation) {
        var csv = 'Item,Cost,Sold,Profit\n'
        csv += 'Keyboard,$10.00,$16.00,$6.00\n'
        csv += 'Monitor,$80.00,$120.00,$40.00\n'
        csv += 'Mouse,$5.00,$7.00,$2.00\n'
        csv += ',,Total,$48.00\n'
        conversation.disposition.type = 'attachment'
        conversation.disposition.filename = 'bill.csv'
        return csv
}
```

Most web browsers would recognize the MIME type and ask the user if they would prefer to either download the file with the suggested "bill.csv" filename, or open it in a supporting application, such as a spreadsheet editor.

> Note that the disposition is *not* cached. If you wish to use this feature, you need to disable caching on the particular resource.

External Requests

Prudence uses the Restlet library to serve RESTful resources, but can also use it to consume them. In fact, the client API nicely mirrors the server API.

Note that Prudence can also handle internal REST requests without going through HTTP or object serialization. There is an entire internal URI-space (page 115) at your fingertips.

> It's not a good idea to send an external request while handling a user request, because it could potentially cause a long delay and hold up the user thread. It would be better to use a background task (page 102). A possible exception is requests to servers that you control yourself, and that represent a subsystem of your application. In that case, you should still use short timeouts (page 61) and fail quickly and gracefully.

For our examples, let's get information about the weather on Mars from MAAS.

In JavaScript, you can use the powerful Prudence.Resources.request API:

```
document.require('/prudence/resources/')
var weather = Prudence.Resources.request({
        uri: 'http://marsweather.ingenology.com/v1/latest/',
        mediaType: 'application/json'
})
if (null !== weather) {
        print('The max temperature on Mars today is ' + weather.report.max_temp + ' degrees')
}
```

The API will automatically convert the response according to the media type. In this case, we requested "application/json", so the textual response will be converted from JSON to JavaScript native data. The API will also automatically follow redirects.

Payloads sent to the server, for the "POST" and "PUT" verbs, are also automatically converted:

```
var newUser = Prudence.Resources.request({
        uri: 'http://mysite.org/user/newton/',
        method: 'put',
        mediaType: 'application/json',
        payload: {
                type: 'json',
                value: {
                        name: 'Isaac',
                        nicknames: ['Izzy', 'Zacky', 'Sir']
                }
        }
})
```

Read the API documentation carefully, as it supports many useful parameters.

Low Level For non-JavaScript you can use the lower-level document.external API:

```
document.require('/sincerity/json/')
var resource = document.external('http://marsweather.ingenology.com/v1/latest/', 'application
result = resource.get()
if (null !== result) {
        weather = Sincerity.JSON.from(result.text)
        print('The max temperature on Mars today is ' + weather.report.max_temp + ' degrees')
}
```

Timeout

Surprisingly, you cannot set the timeout per request, but instead you need to configure the timeout globally for the HTTP client (page 122). This is due to a limitation in Restlet that may be fixed in the future.

Secure Requests

These APIs support secure requests to "https:" servers. Such requests rely on the JVM's built-in authorization mechanism. Like most web browsers, the JVM recognizes the common Internet certificate authorities. This means that if you're using your own self-created keys, that don't use an approved certificate, you need to specify these keys via a "trust store" for the JVM. For an example, see secure servers (page 120) in the configuration chapter.

RESTful Files

The same APIs can be used to easily access resources via the "file:" pseudo-protocol. Let's read a JSON file:

```
var data = Prudence.Resources.request({
        file: '/tmp/weather.json',
        mediaType: 'application/json'
})
```

The above is simply a shortcut to this:

```
var data = Prudence.Resources.request({
        uri: 'file:///tmp/weather.json',
        mediaType: 'application/json'
})
```

You can even "PUT" new file data, and "DELETE" files using this API.

Caching

The State of the Art

Doing caching right is far from trivial: it's much more than just storing data in a key-value store, which is what most web platforms offer you.

Prudence's caching mechanism features the following:

- Template-based cache key generation with support for custom plugins. This allows you full flexibility in caching data that varies per external and internal conditions.
- Fully integrated with client-side caching: uses conditional HTTP requests to make sure clients don't download data they already have, while guaranteeing that they will download newer versions of the data. This enhances the user experience (faster responses) while saving you on bandwidth.
- Allows tagging of cache entries, so that whole swaths of the cache can be invalidated at once just by specifying a tag.
- Tiered caching strategies: allows chaining cache backends in sequence, such that faster backends can be placed before slower ones.
- Prudence caches compressed (gzip and DEFLATE) representations separately, allowing you to save precious CPU cycles on the server.
- Not just pages: Prudence caches your web APIs, too, with all the same features mentioned above.
- For page caching, Prudence caches included fragments individually, allowing for fine-grained control over which parts of the page are cached. With smart use of cache key templates, you can optimize page caching to perfection.

But what's really great about Prudence is how easy it is to use these features: in most cases caching is pretty much automatic. When you need to customize, the API is clear and easy to use.

Server-Side Caching

Five of the caching APIs are in the "caching" namespace, and one is in "application".

For template resources, you may call these APIs anywhere on the page, but for manual resources they should be called in handleInit (page 36).

caching.duration

Specifies the duration of cache entries in milliseconds. Set this to a greater-than-zero value to enable caching on the current resource. The default is zero, meaning that caching is disabled.

You can set this value to a either a number or a string (page 71). For example, "1.5m" is 90000 milliseconds. Note, though, they when you read the value, it will always be numeric (a long integer data type).

Once enabled, every incoming request will have a cache key generated for it based on the cache key template, plus compression information. Prudence will attempt to fetch the cache entry from the cache, and if it's still valid, will display it to the user (this is called a "cache hit"). If there is no cache entry, or it's invalid, Prudence will run the resource as usual (this is called a "cache miss"), and then store a new cache entry via the key.

Compression is handled specially: if the requested compressed cache entry does not exist, then Prudence will attempt to fetch the uncompressed cache entry. If that exists, Prudence would simply compress it and store the compressed version so that compression could be avoided in the future. In the debug headers, this would appear as a "hit;encode" event. Likewise, when storing a new compressed cache entry (during a "miss"), Prudence actually stores both the compressed version as well as the uncompressed version.

See the API documentation for more details.

A Little Bit of Caching Goes a Long Way Remarkably, even a very small cache duration of just a second or two can be immensely beneficial. It will ensure that if you're bombarded with a sudden upsurge of user requests to the resource, your application won't collapse. The cost is often very much worth it: having the "freshness" of your data being delayed by just a few seconds is usually not a big deal.

It's Not Always Worth It It's important to remember that caching is not always faster than fully generating the page. Caching backends are generally very fast, but they still introduce overhead. So, just like in any scenario, avoid premature optimization and benchmark your resources to be sure that caching would indeed improve your performance and scalability.

Caching Forever? You might think that your invalidation scheme is so perfect that there's no reason to ever have your cache entries expire. Well, think again: without a clear expiration time, your cache would continue growing forever. Finite durations thus allow for a way for the cache to recycle.

caching.tags

Tags are simple strings that you can associate with a resource's cache entries, which can then be used to invalidate all entries belonging to a particular tag. You may add as many tags as you wish:

```
caching.tags.add('blog')
caching.tags.add('news.' + newsDate)
```

Note that tags are associated with *all* cache entries based on the resource, whatever their final cache key.
See the API documentation for more details.

caching.keyTemplate

The cache key template is a string with variables delimited in curly brackets that is cast into a cache key per request. The variables are elaborated in the chapter on string interpolation (page 113), and are essentially the same as those used for URI templates. However, Prudence also lets you install plugins (page 64) to support your own specialized template variables. You can debug the cache key template by using the caching debug headers (page 73).
See the API documentation for more details.
Prudence's default cache key template is sensible enough for most scenarios: "{ri}|{dn}|{nmt}|{nl}|{ne}". You can change it in your settings.js (page 73). Let's break it down:

- The "ri" variable is cast to the entire client URI, while "dn" is the document (file) name.
- It's a convention in Prudence, but not a requirement, to use "|" as a separator of cache key elements, because it's a character that won't be used by most template variables.
- "dn" makes sure that dynamic captures (page 27) and other server-side redirections would still be cached uniquely: the same URI might reach a different document depending on an external factor.
- "nmt", "nl" and "ne" all make sure that we have a different key per negotiated format.
- *Pay special attention to "ne"*: if you are supporting compression, you will *need* it in your cache key templates. Because Prudence usually handles compression automatically for you, it's easy to forget this important variable.

The above is a good cache key template, but you may want to modify it. Here are two common reasons:

Per-User Caching A common scenario is for a resource to be generated different accordingly to the logged-in user. You would thus want to include a user identifier in the cache key. To do this, you would likely need to write a plugin (page 64) to interpolate that identifier. You cache key template could then look something like "{ri}|{uid}|{MT}|{L}", where "uid" is handled by your plugin.

Caching Fragments You might be including the same fragment in many pages, but the fragment in fact will be mostly identical. In this case, you can optimize by using a shorter cache key, such that the fragment would be cached only once for all inclusions. You would thus *not* want to use "ri". A simple example would be "{dn}|{MT}|{L}". This can also be used in conjunction with per-user caching: for example, if you want to cache the same fragment per-user, it would be "{dn}|{uid}|{MT}|{L}". Note that fragments are never compressed, so you don't need "{ne}".
Generally, creating the best key template involves a delicate balance between on the one hand making sure that differing data is indeed cached separately, while on the other hand making sure that you're not needlessly caching the same data more than once.

caching.keyTemplatePlugins

This powerful feature allows you to interpolate your own custom values into cache keys. While this does mean that *some* code will be run for every request, even for cache hits, it gives you the opportunity to write efficient, fast code that is used only for handling caching.

A common scenario requiring a key template plugin is to interpolate a user ID. We'd install it like so:

```
caching.keyTemplatePlugins.put('uid', '/plugins/session/')
```

The above means that existence of a "uid" variable in a key template would trigger the invocation of the "/plugins/session/" library to handle it.

Actually, Prudence also allows you to install key template plugins by configuring them in your application's settings.js (page 73). In that case, the plugin would be installed for *all* resources:

```
app.settings = {
        ...
        code: {
                cacheKeyTemplatePlugins: {
                        uid: '/plugins/session/'
                }
        }
}
```

The implementation of the plugin, however, would be the same however we install it. Our plugin would be in "/libraries/plugins/session.js":

```
document.require('/sincerity/objects/')

function handleInterpolation(conversation, variables) {
        for (var v in variables) {
                var variable = variables[v]
                if (variable == 'uid') {
                        var sessionCookie = conversation.getCookie('session')
                        if (Sincerity.Objects.exists(sessionCookie)) {
                                var session = getSession(sessionCookie.value)
                                if (Sincerity.Objects.exists(session)) {
                                        conversation.locals.put('uid', session.getUserId())
                                }
                        }
                }
        }
}

function getSession(sessionId) {
        ...
        return session
}
```

Implementation notes:

- The entry point is "handleInterpolation", and accepts as arguments the current conversation as well as an array of the variables to interpolate. Prudence makes sure to optimize these call by gathering into the array only those variables actually used in the key template. Of course, it would not call your plugin at all if the variables are not present. Also, no cache key casting would be done at all if caching.duration is zero. (This makes it very safe to install the plugin for all resources in setting.js, knowing that it would never be called unless necessary.)
- In this implementation, we're assuming that a cookie named "session" is sent with the session ID. We would then have some functionality in getSession to retrieve a session object.
- The interpolation "trick" is to set up our variable as a conversation.local. Because conversation.locals are interpolated as is (page 115), we've effectively allowed the cache key template to be correctly cast.

See the API documentation for more details.

An Optimization In the above example, we are retrieving the session in order to discover the user ID, an operation that could potentially be costly. Consider that if we have a cache miss, then the session might be retrieved *again* in the implementation of the resource.

It's easy to optimize for this situation by storing the session as a conversation.local, such that it would be available in the resource implementation. We'd modify our above plugin code like so:

```
if (Sincerity.Objects.exists(session)) {
        conversation.locals.put('session', session)
        conversation.locals.put('uid', session.getUserId())
}
```

Then, in our resource implementation, would could check to see if this value is present:

```
var session = conversation.locals.get('session')
if (!Sincerity.Objects.exists(session)) {
        var sessionCookie = conversation.cookies.get('session')
        if (Sincerity.Objects.exists(sessionCookie)) {
                session = getSession(sessionCookie.value)
        }
}
```

caching.key

This is a read-only value, meant purely for debugging purposes. By logging or otherwise displaying it, you can see the cache key that Prudence *would* use for the current resource. *Would* is the key qualifier here: of course, your code displaying the cache key won't actually be run in the case of a cache hit.

Another way to see the cache key is to enable the caching debug headers (page 73).

See the API documentation for more details.

application.cache

You'll most likely want to use this API to invalidate a cache tag:

```
application.cache.invalidate('blog')
```

See the API documentation for more details.

Backends

See the configuration chapter (page 123) for a full guide to configuring your tiered caching backends. Prudence comes with many powerful options.

Client-Side Caching

Many HTTP clients, an in particular web clients, can cache results locally. Actually, HTTP specifies two different caching modes:

- **Conditional mode**: Here, the client caches the downloaded data, but checks with the server to make sure new data is not available. If new data is not available, then the cached data is used, otherwise it is downloaded. Remarkably, this is done in a single step: the HTTP headers returned from the server provide the necessary information about the freshness of the data, and if there is no need to download, then the client will stop there. The request will end with a 304 "not modified" HTTP status code. Prudence provides you with various tools to optimize conditional HTTP, so that you can be sure to do only the minimal amount of work necessary for the check.

- **Offline mode**: Here, the client is told *explicitly* not to check with the server for a certain amount of time. Obviously, this provides the best possible performance and user experience: no network chatter is necessary. But, also obviously, this means that during that period there is no way for your application to push new data to the client. Prudence provides you with tools for using offline mode, but you should use it with care.

Automatic Client-Side Caching

Here's the good news: if you're using server-side caching, then client-side caching in conditional mode is enabled for you automatically, by default, for the cached resources. Moreover, Prudence will compute the expiration times accordingly and specifically per request. For example, if you are caching a particular resource for 5 minutes, and a client tries to access that resource for the first time after 1 minute has passed since the cache entry was stored, then the client will be told to cache the resource for 4 minutes. After those 4 minutes have passed, the client won't need to do a conditional HTTP request: it knows that it would need new data.

Automatic client-side caching applies to both template and manual resources.

Changing the Mode If you wish, you may change the default mode from conditional to offline in your routing.js:

```
app.routes = {
        ...
        {
                type: 'templates',
                clientCachingMode: 'offline',
                maxClientCachingDuration: '30m'
        }
}
```

Note that "maxClientCachingDuration" only has an effect in offline mode: it provides a certain safety cap against too-long cache durations. The default is -1, which means this cap is disabled.

Just make sure you understand the implications of offline mode: you will not be able to push changes to the client for the cached duration. You can also turn off client-side caching by setting "clientCachingMode" to "disabled".

Static Resources You can add automatic client-side caching to static resources, too (page 47).

Manual Client-Side Caching

If you're not using Prudence's automatic caching, you can still benefit from client-side caching by using the APIs.

For conditional mode, you have the option of using modification timestamps and/or tags:

- conversation.modificationTimestamp: The client will compare the modification timestamp it stored with its cached entry to this.
- conversation.tagHttp: The client will compare the tag it stored with its cached entry to this.

For offline mode:

- conversation.maxAge: The client will not contact your server regarding the resource until this number of seconds has passed.
- conversation.expirationTimestamp: Similar to "maxAge", but using an explicit expiration time. Note that if both are set, most clients will treat "maxAge" as superseding "expirationTimestamp". ("maxAge" was introduced in HTTP/1.1, "expirationTimestamp" is from HTTP/1.0.)

Manual Ain't Easy Note that though the APIs are very simple, leveraging them is not trivial, and may require you to design your data structures and subsystems with significant thought towards client-side caching.

For example, storing a modification timestamp within a single database entry is simple enough, but what if your final data is actually the result of a complex query using from several entries? You could potentially use the latest modification date of all of them: but, as you can see, calculating it can quickly get complicated and inefficient. Sometimes it might make sense to actually "bubble" modification dates upwards to all affected database entries as soon as you modify them: it would make your save operations heavier, but it could very well be worth it for greater scalability and an improved user experience.

In some cases, calculating a tag might be less costly than keeping track of modification timestamps. For some data structures, tags may even be provided for you as a byproduct of how they work: key hashes, serial IDs, checksums, etc., are all great candidates for tags.

Optimizing the Server Conditional mode can improve the client experience, but it can improve the server experience, too.

Prudence, using a great feature of the Restlet library, lets you create the conditional HTTP headers and return them to the client *without generating the response*. Thus, *only* if the conditional request continues would your response generation code be called. This feature is internally used by Prudence's automatic caching, but you can also use it yourself in manual resources, using the "handleGetInfo" entry point.

Here's an example—not the most efficient one, but it will demonstrate the flow:

```
function handleGetInfo(conversation) {
        var id = conversation.locals.get('id')
        var data = fetchDataFromDatabase(id)
        conversation.locals.put('data', data)
        return data.getModificationTimestamp()
}

function handleGet(conversation) {
        var data = getData(conversation)
        conversation.modificationTimestamp = data.getModificationTimestamp()
        return Sincerity.JSON.to(data)
}

function getData(conversation) {
        var data = conversation.locals.get('data')
        if (!Sincerity.Objects.exists(data)) {
                var id = conversation.locals.get('id')
                data = fetchDataFromDatabase(id)
        }
        return data
}

function fetchDataFromDatabase(id) {
        ...
}
```

As you can see, we're storing the fetched data in a conversation.local, so that if handleGet is called after handleGetInfo, we would not have to access the database twice.

What have we accomplished in this example? Not that much: all we've done is avoided JSON serialization for those conditional requests that stop at handleGetInfo. A worthwhile little optimization, to be sure, but not one with very dramatic effects. It might be more effective in cases in which we had other heavy processing in handleGet that could be avoided.

The handleGetInfo trick really shines when you have *a shortcut to accessing the modification date or the tag.* Consider as a common example the a way a filesystem works: you can fetch the file modification date with one system API call, without actually opening the file for reading its contents, which would of course be a much costlier operation. Using handleGetInfo with that API would be able to affect a crucial (even necessary!) optimization. Indeed, that's how static resources (page 46) work internally.

But how would you implement this with a database server? Most database servers don't allow for such shortcuts: sure, you *could* only fetch the modification date column from a row for handleGetInfo, but it would be inefficient if soon after you would also need to fetch the rest of the columns. It would be more efficient to just fetch the entire row at once, and so you're back to our non-dramatic optimization from before.

What you could do, however, is cache only the modification dates separately in a specialized backend that is much lighter and faster than the database server, for example Hazelcast or memcached. Here's how it might look:

```
function handleGetInfo(conversation) {
        var id = conversation.locals.get('id')
        var modificationTimestamp = fetchTimestampFromCache(id)
        return Sincerity.Objects.exists(modificationTimestamp) ? modificationTimestamp : null
}
```

```
function handleGet(conversation) {
        var data = fetchDataFromDatabase(conversation)
        conversation.modificationTimestamp = data.getModificationTimestamp()
        storeTimestampInCache(id, data.getModificationTimestamp())
        return Sincerity.JSON.to(data)
}

function fetchDataFromDatabase(id) {
        ...
}

function fetchTimestampFromCache(id) {
        return conversation.distributedGlobals.get('timestamp:' + id)
}

function storeTimestampInCache(id, timestamp) {
        conversation.distributedGlobals.put('timestamp:' + id, timestamp)
}
```

As you can see, the optimization won't be effective unless the cache is warm. Thus, to make it truly effective, you would need a special subsystem to warm up the cache in the background...

Welcome to the world of high-volume web! The solutions for massive scalability are rarely trivial. While Prudence can't provide you with automation for every scenario, at least it provides you with the tools on which you can build comprehensive solutions. With careful planning, you can go very far indeed.

In summary, before you go ahead and provide a handleGetInfo entry point for every resource you create, consider:

1. It could be that you don't need this optimization. Make sure, first, that you've actually identified a problem with performance or scalability, and that you've traced it to handleGet on this resource.
2. It could be that you won't gain anything from this optimization. Caches and other optimizations along the route between your data and your client might already be doing a great job at keeping handleGet as efficient as it could be. If not, improving them could offer far greater benefits overall than a complex handleGetInfo mechanism.
3. It could be that you'll even hurt your scalability! The reason is that an efficient handleGetInfo implementation would need some mechanism in place to track of data modification, and this mechanism can introduce overhead into your system that causes it to scale worse than without your handleGetInfo.

See "Scaling Tips" (page 159) for a thorough discussion of the problem of scalability.

Bypassing the Client-Side Cache

A devilishly useful aspect of client-side caching is that the cache key is the entire URL, *including* the query. This means that by simply adding a query parameter (which you otherwise ignore in your server-side handling), you can force the client to fetch new data, *even when using offline mode caching for that resource.*

Of course, you can't use this trick unless you can control the URLs which the client uses. Luckily, this is exactly what you can do in HTTP: see the static resources guide (page 47) for a comprehensive discussion.

Two Client-Side Caching Strategies

The default Prudence application template is configured with minimal client-side caching, which is suitable for development deployments. However, once you are ready to move your application to production or staging, you will likely want a more robust caching strategy.

We will here present two common strategies, and discuss the pros and cons of each. They are intended as polar opposites, though you may very well choose a strategy somewhere in between.

Paranoid: Short-Term Caching This is a great strategy if you're *not* feeling very confident about managing caching in your application logic. Perhaps you have too many different kinds of pages requiring different caching strategies. Perhaps you can't maintain the strict discipline required for more aggressive caching, due to a quickly changing application structure ("agile"?) or third-party constraints.

If you're in that boat, short-term caching is recommended over no caching at all, because it would still offer better performance and scalability. Because caching is short-term, any mistakes you make won't last for very long, and can quickly be fixed.

How short a term depends on two factors: 1) usage patterns for your web site, and 2) the content update frequency. For example, if a user tends to spend about an hour browsing your site, then a one-hour caching duration makes sense: the client would only have a slightly slower page load at the beginning of the visit.

Our strategy would then be to:

- Use conditional caching mode for manual and template resources. (This is the default.) This would guarantee that we can always push changes to users even when cached.
- Use offline caching mode for static resources commonly used in web pages, for 1 hour. Sure, we won't be able to push changes for those resources in that hour, but by the next time the user visits our site, they should be OK. Within that hour, they would get excellent performance.

Here's an example routing.js:

```
app.routes = {
	'/*': [
		...
		'manual', // clientCachingMode: conditional
		'templates', // clientCachingMode: conditional
		{
			type: 'cacheControl',
			mediaTypes: {
				'image/*': '1h',
				'text/css': '1h',
				'application/x-javascript': '1h'
			},
			next: {
				type: 'less',
				next: 'static'
			}
		}
		...
```

Confident: Long-Term Caching For long-term caching to work, you must have good systems in place for bypassing the cache when necessary:

- For the static resource assets used in your HTML pages, you must be able to change the URLs when the resource contents change. See the discussion in the static resources guide (page 47). With such a system in place, you could cache these resources forever!
- Your HTML template pages can be cached offline, too, but this means two things:
  - You are sure that you won't have to push changes to the client *very* soon. For example, if you allow your home page to be cached offline for an hour, the browser won't have to hit your site at all when returning to the home page! You'd likely still want to set the cache durations for such pages to a short-term time, though, because you would, of course, eventually want to push changes.
  - You can cache HTML offline while *still* allowing yourself an opportunity to push modifications to the client, by using JavaScript (or browser plugins, if you must). For example, even while your home page is cached offline, a JavaScript background routine in it might be pulling data from you and modifying that page accordingly.

Our strategy would then be to:

- Enable offline caching mode for template resources. Per resource, we would have to carefully consider what a reasonable caching duration would be. It should usually be the average length of a user visit to our site, and probably should not exceed 24 hours: at least if the users revisit our page the next day, they'll see our updated changes.
- Set offline caching for static resources commonly used in web pages to the *far future*. This means that we must assume that we can *never* push changes for a URL, and thus *must* change the URLs when the content changes.

Here's an example routing.js:

```
app.routes = {
	'/*': [
		...
		'manual', // clientCachingMode: conditional
		{
			type: 'templates',
			clientCachingMode: 'offline'
		},
		...
		{
			type: 'cacheControl',
			mediaTypes: {
				'image/*': 'farFuture',
				'text/css': 'farFuture',
				'application/x-javascript': 'farFuture'
			},
			next: {
				type: 'less',
				next: 'static'
			}
		},
		...
```

Configuring Applications

Prudence applications live in their own subdirectory under "/component/applications/". The subdirectory *name* itself can be considered a setting, as it is used as a default identifier for the application in various use cases.

Prudence uses "configuration-by-script" almost everywhere: configuration files are true JavaScript source code, meaning that you can do pretty much anything you need during the bootstrap process, allowing for dynamic configurations that adjust to their deployed environments.

> Prudence, as of version 2.0, does not support live re-configuring of applications. You must restart Prudence in order for changed settings to take hold. The one exception is crontab (page 106): changes there are picked up on-the-fly once per minute.

Overview

The subdirectory contains five main configuration files:

- **settings.js**: This required file, detailed in this chapter, includes settings used by Prudence as well as your own custom settings (page 75).
- **routing.js**: This required file defines the application's URI-space (page 20).
- **crontab**: This optional file defines regularly scheduled background tasks (page 106).
- **default.js**: This required file is used to load the other configuration files above. You should not normally need to edit this file, but feel free to examine it to understand the application bootstrapping process.

settings.js

If you use the default template with the "sincerity prudence create" command, you should get a setting.js file that looks something like this:

```
app.settings = {
        description: {
                name: 'myapp',
                description: 'Skeleton for myapp application',
                author: 'The Author',
                owner: 'The Project'
        },
        errors: {
                debug: true,
                homeUrl: 'http://threecrickets.com/prudence/',
                contactEmail: 'info@threecrickets.com'
        },
        code: {
                libraries: ['libraries'],
                defrost: true,
                minimumTimeBetweenValidityChecks: '1s',
                defaultDocumentName: 'default',
                defaultExtension: 'js',
                defaultLanguageTag: 'javascript',
                sourceViewable: true
        },
        templates: {
                debug: true
        },
        caching: {
                debug: true
        },
        uploads: {
                root: 'uploads',
                sizeThreshold: '0kb'
        },
        mediaTypes: {
                php: 'text/html'
        },
        logger: 'myapp'
}
```

Numeric Shortcuts

Time durations are in milliseconds and data sizes in bytes. But these can be specified as either numbers or strings:

- **Durations**: numerically as milliseconds, or using 'ms', 's', 'm', 'h' or 'd' suffixes for milliseconds, seconds, minutes, hours or days. Fractions can be used, and are rounded to the nearest millisecond, for example: "1.5d".
- **Data sizes**: numerically as bytes, or using 'b', 'kb', 'mb', 'gb' or 'tb' suffixes. Magnitude uses the binary rather than decimal system: 1kb = 1024b. Fractions can be used, and are rounded to the nearest byte, for example: "1.5mb".

You can accomplish the same trick for your own code using Sincerity.Localization.toMilliseconds and Sincerity.Localization.toBytes.

app.settings.description

Information here is meant for humans.

- **name**: if not specified, will default to the application's subdirectory name
- **description**: a sentence or a sentence fragment describing the application
- **author**: the name of company or individual who made the application
- **owner**: the project to which the application belongs

This information appears in the Prudence Administration application, and you may access it yourself using the application.application.name, application.application.description, etc., APIs.

app.settings.errors

Configure Prudence's error handling behavior here.

- **debug**: Boolean value; when true enables various useful debugging features detailed below; **should be set to false for production deployments**
- **debugHeader**: sets this HTTP header to "error" when the debug page is displayed; defaults to "X-Debug"; set to null if you don't want the header to be used; only relevant when "debug" is true
- **homeUrl**: a web URL, displayed in the default error page template
- **contactEmail**: an email address, displayed in the default error page template

Note that you can route your own error pages in routing.js (page 30) to replace the default template. If you wish to use the values you set here in your template, use the application.application.statusService.homeRef and application.application.statusService.contactEmail APIs.

Uncaught Exception Debugging

Uncaught exceptions in your code will automatically set the HTTP response status code to 500 ("internal server error"), but here you can configure the content of that response.

When "debug" is true, Prudence will return a very detailed HTML debug page (page 92). Because this might reveal your application's internal data, **make sure to set "debug" to false for production deployments**.

When "debug" is false, the default error template page will be displayed.

app.settings.code

Here you can control how Prudence deals with your code:

- **libraries**: An array of paths where importable libraries will be found. If relative, they will be based on the application's root subdirectory.
- **defrost**: When true will attempt to "defrost" manual and template resources under the "/resources/" subdirectory. Defrosting means pre-parsing and sometimes pre-compiling the code: this allows for faster startup times on first hits to these resources. Note that defrosting is not pre-heating: the former only pre-compiles, the latter actually does a "GET" on your resources, which would ensure that services used by your resources are also warmed up. See app.preheat (page 33). The default value for this setting is true.
- **minimumTimeBetweenValidityChecks**: Scripturian makes sure to reload (and thus re-compile) code if the source files are changed, for which it compares the file's modification dates to the cached values. For high-volume deployments, this might involve constantly checking the filesystem, potentially resulting in performance problems on some operating systems. This value allows you to enforce a delay between these checks. It's a good idea to set it to anything greater than zero. The default value for this setting is 1 second.
- **defaultDocumentName**: When a document name specifies to a directory, Scripturian will internally change the specification to a document with this name in the subdirectory (excluding the extension). For example, if the value is "default" and you are calling "document.require('/mylibrary/')" and "/libraries/mylibrary/" is a directory, the it would specify "/libraries/mylibrary/default.*". The default value for this setting is "default".

- **defaultExtension**: If more than one file in a directory has the same name but different extensions, then this extension will be preferred. For example, if the value is "js" and a directory has both "mylibrary.js" and "mylibrary.py", then the former file will be preferred. The default value for this setting is "js".
- **defaultLanguageTag**: If a scriptlet tag does not specify a language, then this value will be the default. The default value for this setting is "javascript".

app.settings.templates

Configure templates resources (page 39) here.

- **debug**: Boolean value; when true, under your container's "/cache/scripturian/" directory you will see the generated source code for each scriptlet resource. The filenames will include the timestamp for their creation, so you can see all versions of code that were created.
- **parser**: The parser name; allows you to change the default scriptlets parser to your own custom Scripturian parser; defaults to "scriptlets"
- **plugins**: Configure scriptlet plugins (page 44) here

app.settings.caching

Configure caching (page 61) here.

- **debug**: Boolean value; when true, special HTTP response headers will be added for cached resources:
  - **X-Cache**: the caching event; either "hit", "hit;encode" or "miss"
  - **X-Cache-Expiration**: a timestamp in standard HTTP format
  - **X-Cache-Key**: the cache key (see also the caching.key API)
  - **X-Cache-Tags**: comma-separated list of cache tags
- **defaultKeyTemplate**: Allows you to set the default caching.keyTemplate (page 63) for all resources. If not specified, will be "{ri}|{dn}".
- **keyTemplatePlugins**: A dict mapping cache key template variable names to plugin library names. Allows you to install key template plugins for all resources in the application. See the caching.keyTemplatePlugins API (page 64) for more information.

app.settings.compression

Global settings for resource compression. Note that compression can be turned on or off per individual resource type.

- **sizeThreshold**: Responses smaller than this will not be compressed. Defaults to 1024.
- **exclude**: An array of MIME types that will be *additionally* excluded from compression. Note that several common types are automatically excluded, specifically compressed archive and media formats.

app.settings.uploads

Configure file upload behavior (page 57) here.

- **root**: This is where uploaded files are stored. If relative, it will be based on the application's root subdirectory.
- **sizeThreshold**: The file upload mechanism can optimize by caching small files in memory instead of saving them to disk. Only if files are greater in size will be they be stored. Set to zero to save all files to disk.

app.settings.mediaTypes

This dict maps filename extensions to MIME types.

Prudence recognizes many common file types by default: for example, "png" is mapped to "image/png". However, using this setting you can define additional mappings or change the default ones. Note that each filename extension can be mapped to one *and only one* MIME type.

This setting is used mostly for textual (page 39) and static resources (page 46). For example, a template resource named "person.t.html" will have the default "text/html" MIME type (which you can change in scriptlet code via the conversation.mediaType APIs), and a static resource named "logo.png" will have the "image/png" MIME type.

For manual resources, you define their supported MIME types manually in handleInit (page 36). There, you can refer to MIME types directly via the conversation.addMediaTypeByName API, or you can look them up from this setting using the conversation.addMediaTypeByExtension API.

An example:

```
app.settings = {
        ...
        mediaTypes: {
                webm: 'video/webm',
                msh: 'model/mesh'
        }
}
```

> **Note for PHP**: You may notice that the "default" template's settings.js sets the "text/html" MIME type for the "php" extension. The reason for this is that ".php" files you put in resources are usually expected to output HTML. You may change if you require a different behavior.

app.settings.distributed

Configure clusters (page 137) here. These settings are for the usage of Hazelcast.

- **applicationInstance**: The name of the Hazelcast "application" instance. Defaults to "com.threecrickets.prudence.application". Can be accessed at runtime via the application.hazelcastApplicationInstance API.
- **globals**: The name of the Hazelcast map used for the application.distributedGlobals API (page 83). Defaults to "com.threecrickets.prudence.distributedGlobals.[name]", where "name" is the application's subdirectory name.
- **sharedGlobals**: The name of the Hazelcast map used for the application.distributedSharedGlobals API (page 83). Defaults to "com.threecrickets.prudence.distributedSharedGlobals".
- **taskInstanceSharedGlobal**: The name of the application.sharedGlobal (page 82) in which the Hazelcast "task" instance is stored. If there is nothing in that shared global, then the "task" instance will be identical to the "application" instance. Defaults to "com.threecrickets.prudence.hazelcast.taskInstance". Can be accessed at runtime via the application.hazelcastTaskInstance API.
- **executorService**: The name of the Hazelcast executor service used for the distributed task APIs (page 105). Defaults to "default". Can be accessed at runtime via the application.hazelcastExecutorService API.

To learn how to configure general Hazelcast settings, see the configuration guide (page 117).

app.settings.routing

Though routing is configured in routing.js, here we can configure how routing is handled.

- **useForwardedHeaders**: Whether to apply the "X-Forwarded-Proto", "X-Forwarded-Host", and "X-Forwarded-Port" HTTP headers to requests. Defaults to false. You should only set this to true if you are behind a proxy, otherwise clients could use this to manipulate the information. See "forwarded headers" (page 145). When true, installs the ForwardedFilter before your application on all non-internal hosts.

app.settings.logger

Set this string to override the default logger name. If not set, it will be the name of the application's subdirectory. The actual logger name will be this value prefixed with "prudence.".

See the logging chapter (page 88) for a complete discussion.

app.globals

Use this for custom settings for your application: values here will become application.globals (page 82) when your application is running. Note that Prudence also supports localized settings via inversion of control (page 111).

This dict is "flattened" using dot separators. For example, the following:

```
app.globals = {
        database: {
                driver: 'mysql',
                table: {
                        db: 'myapp',
                        name: 'users'
                }
        }
}
```

...would be interpreted as if it were:

```
app.globals = {
        'database.driver': 'mysql',
        'database.table.db': 'myapp',
        'database.table.name': 'users'
}
```

In your code, you would access these values using the application.globals API:

```
var driver = application.globals.get('database.driver')
```

To set application.sharedGlobals (page 82) you can use a custom service (page 123).

Lazy Initialization

Lazy initialization allows you to defer the initialization of an application.global to its actual use within your application. This feature can be used to solve three different problems:

- You wish to set up an application.global with resources that would only be available during runtime, not during the bootstrap process.
- You wish to set up a heavy resource, but instead of slowing down the bootstrap process by initializing it up front, you'd rather initialize it on-demand when actually used for the first time by the running application.
- You wish to simplify the configuration of complex objects: the mechanism allows you to use a simple DSL (Domain-Specific Language), which behind the scenes calls an arbitrary constructor.

  The mechanism relies on evaluation of JavaScript code, and thus is only available for other JavaScript code in your applications. However, similar principles can be used for other languages that support eval. Study the library's code to see how it's done!

Double Dot Lazy globals can be defined using a double-dot key. Here's an example for settings.js:

```
app.globals = {
        services: {
                email: {
                        '..': {
                                dependencies: '/services/email/',
                                name: 'EmailService',
                                config: {
                                        smtp: 'localhost',
                                        origin: 'service@myapp.org'
                                }
                        }
                }
        }
}
```

The lazy configuration has three settings, which are used when the object is initialized:

- **dependencies**: These are called with document.require
- **name**: This is the constructor to call
- **config**: This is an optional argument sent to the constructor; note that it *must* be JSON-serializable

For this example, here's how our "/libraries/services/email.js" file could look:

```
function EmailService(config) {
        this.smtp = config.smtp
        this.origin = config.origin

        this.send = function(destination, message) {
                application.logger.info('Sending an email from ' + config.origin + ' to ' + d
                ...
        }
}
```

To actually access the object in the application, while *possibly* triggering lazy initialization (if it was not initialized already), we need the Prudence.Lazy library:

```
document.require('/prudence/lazy/')

var emailService = Prudence.Lazy.getGlobalEntry(
        'services.email',
        null,
        function(c) { return eval(c)() })

emailService.send('test@myapp.org', 'Hello, this is a test message!')
```

That third argument requires some explanation: it is a closure (anonymous function) that accepts the constructor source code and evaluates it locally. It will be called *only if* the global has *not yet* been initialized. Thus, in all calls except the first, this would be a very fast application.global lookup.

Triple Dot The lazy initialization mechanism can optimize for dicts and arrays of lazy entries using the triple-dot key:

```
app.globals = {
        emailServices: {
                '...': {
                        moderator: {
                                dependencies: '/services/email/', name: 'EmailService',
                                config: {smtp: 'localhost', origin: 'moderator@myapp.org'}
                        },
                        admin: {
                                dependencies: '/services/email/', name: 'EmailService',
                                config: {smtp: 'localhost', origin: 'admin@myapp.org'}
                        }
                }
        },
        nodes: {
                '...': [{
                        dependencies: '/services/nodes', name: 'Node',
                        config: {address: 'node1.myapp.org'}
                }, {
                        dependencies: '/services/nodes', name: 'Node',
                        config: {address: 'node2.myapp.org'}
                }]
        }
}
```

To use them:

```
document.require('/prudence/lazy/')

var emailServices = Prudence.Lazy.getGlobalMap(
        'emailServices',
        null,
        function(c) { return eval(c)() })

emailServices.moderator.send('test@myapp.org', 'Hello, this is a test message!')

var nodes = Prudence.Lazy.getGlobalList(
        'nodes',
        null,
        function(c) { return eval(c)() })
```

Note that for this optimization the entire dict or array will be created at once and stored in a single application.global.

Forcing Re-Initialization If you edit the source file for the lazy-initialized object—say, our "email.js"—you would not see your changes. The reason is that the object has already been initialized and stored in memory: the recompiled functions would not be hooked to the already-existing object (really just a dict in JavaScript).

You can, however, *force* re-initialization using the API:

```
Prudence.Lazy.getGlobalEntry('services.email').reset()
```

This works on dicts and arrays, too:

```
Prudence.Lazy.getGlobalMap('emailServices').reset()
```

To do this live, you can use the live execution mechanism (page 87).

Concurrency Note The lazy mechanism is thread-safe: if you call getGlobalEntry concurrently, it will make sure to store only the *first* initialized object. You can thus be sure that you will always get the same instance of the lazy-initialized object in your application.

However, take note: it is possible that under high-concurrency situations the object would be created more than once, even if subsequent objects after the first are discarded. This might be a problem if your object creation is *very* heavy. Unfortunately, there is no easy way around this: allowing for multiple object creation is key to making the lazy mechanism perform well (it does no locking). If this is a problem for you, you will have to find your own way to avoid simultaneous initialization: possibly locking internally within your object constructor, and checking to make sure that the constructor is not being simultaneously called by another thread.

Programming

Prudence, and the underlying Sincerity bootstrapping tool, allow you to use your choice among several different programming languages on top of the JVM, and moreover they provide you with context-specific, well-documented APIs per relevant execution environment.

Your current programming skills should transfer readily enough, however some principles might be new to you:

- The main execution environments you will be working with in Prudence are inherently multi-threaded, meaning that you must have some basic knowledge of concurrent programming. The API does its best to help you, but it's still crucial that you understand it.
- If you've never used the programming language on the JVM before, you'll might find some pitfalls in terms of interaction with the Prudence and JVM APIs, and these vary per programming language engine. Usually, this has to do with differences in types: strings, numbers, arrays/lists, etc. Additionally, some JVM implementations differ in small but important ways from the reference implementations. Usually, things will work as expected: if not, you should refer to the documentation for the specific language engine.

Powered by Scripturian Scripturian is the magical library that enables much of Prudence and Sincerity: it abstracts away the specifics of the programming language you're using, thus allowing you to transparently switch between programming languages and engines. It compiles your changes on the fly, caches the bytecode, parses templates into code, provides access to the environment's APIs, and does it all while optimizing for high-performance concurrency.

You don't have to understand how Scripturian works to use Prudence, however it's useful to remember that usually Prudence is ignorant as to what programming language you're using: that's all handles by Scripturian.

APIs

Prudence provides you with an especially rich set of APIs. They come in three categories:

- The **Prudence Core APIs** are multilingual: they are implemented via standard JVM classes that can be called from all supported programming languages: JavaScript, Python, Ruby, PHP, Lua, Groovy and Clojure. Indeed, the entire JVM standard APIs can be access in this manner, in addition to any JVM library installed in your container (under "/libraries/jars/").
- **Prudence JavaScript APIs**: these JavaScript wrappers over the Prudence Core APIs make using them a bit easier with JavaScript. Future versions of Prudence may provide similar friendly wrappers for the other supported languages (please contribute!). Until then, non-JavaScript programmers shouldn't be too jealous: there's nothing that these wrappers can do that you can't do with the core APIs, and it's easy to look into the source code in case you want to duplicate the same functionality in your language of choice.
- **Sincerity JavaScript APIs**: most of the supported programming languages have rich standard libraries as well as an ecology of external libraries. JavaScript, however, stands out for having a very meager standard library. To fill in this gap, Sincerity comes with a useful set of JavaScript libraries. Some of these are written pure JavaScript, offering new and useful functionality, while others provide JavaScript-friendly wrappers over standard JVM libraries.

For the sake of coherence and convenience all these APIs are documented together online, with direct links to the source code. The entire documentation uses the JavaScript calling convention, even for the multlingual APIs, however it should be trivial to use the calling convention (page 80) of your language of choice.

You may be further interested in looking up Prudence's low-level API, which is also fully documented online. Also, sometimes the best documentation is the source code itself.

> The development team spends a lot of time meticulously documenting the APIs. Please send us a bug report if you find a mistake, or think that the documentation can be improved!

Prudence Core APIs

These core APIs can be used by any supported programming language. See calling conventions (page 80) for instructions on how to use these APIs from your language of choice.

The APIs consist of four namespaces (objects) that are defined as global variables:

- application: general application services, such as global state, logging and background tasks
- conversation: services related to the current request-and-response; available for manual and template resources, and also for filters
- document: actually incorporates two kinds of services:

  - access to *other* documents, such as executing or requesting them
  - low-level access to the *current* document

- executable: rarely used, low-level access to Scripturian

JavaScript Libraries

The APIs are *only* available for JavaScript.

To use them, you must "document.require" them according to their URI, and then via their namespaces. For example:

```
document.require(
          '/sincerity/json/',
          '/prudence/tasks/)

println(Sincerity.JSON.to({hello:  'world'}))
```

You can find their source code in the "/libraries/scripturian/" directory of your container.

Sincerity JavaScript Library These libraries are intended to fill in for the lack of a rich standard library for the JavaScript language. They are general-purpose and not specific to Prudence. If you've installed Prudence using Sincerity, then you will find them in your Sincerity installation rather than in your Prudence container (under the "/libraries/scripturian/" directory).

- /sincerity/calendar/: useful helpers for JavaScript Date objects
- /sincerity/classes/: a rich and powerful object-oriented programming (OOP) facility, supporting inheritance, private data and methods, and automatic constructors
- /sincerity/cryptography/: JavaScript-friendly wrappers over the JVM's strong cryptographic libraries, allowing for digests/hashing, encryption/decryption, and cryptographically-strong random number generation
- /sincerity/files/: high-performance reading/writing of files, as well as filesystem operations
- /sincerity/iterators/: standardized access to streaming data structures, with plenty of utility classes for stream manipulation
- /sincerity/json/: high-performance, extensible JSON/JavaScript conversion
- /sincerity/jvm/: conversions between JavaScript and JVM data types
- /sincerity/localization/: unit conversions and human-readable formatting of dates and times
- /sincerity/lucene/: JavaScript-friendly wrappers over the Lucene search engine
- /sincerity/mail/: send emails using JavaScript-friendly wrappers over JavaMail with support for message templates
- /sincerity/objects/: many important enhancements to and utilities for working with standard JavaScript objects, such as dicts, arrays and strings
- /sincerity/platform/: utilities specific to the underlying JavaScript engine (Nashorn or Rhino)
- /sincerity/templates/: straightforward and powerful string interpolation
- /sincerity/xml/: parsing and generation of XML

Prudence JavaScript Library These are JavaScript-friendly wrappers over the Prudence Core APIs, as well as other special uses for JavaScript.

- /prudence/logging/: adds automatic support for string interpolation, exception logging, conditional logging, and other convenient utilities; see logging (page 88) for more details
- /prudence/resources/: a rich set of utilities for generated and parsing web data (page 51)
- /prudence/tasks/: JavaScript-friendly spawning and scheduling of background tasks (page 102)

Libraries for Bootstrap and Configuration We're listing these libraries separately, because they are specifically meant to be used for application configuration.

- /sincerity/annotations/: allows you to easily gather specially marked annotations in text files, which can include your source code and templates; specially designed to integrate well with scriptlet comments
- /sincerity/container/: access to the Sincerity container into which Prudence has been installed
- /prudence/lazy/: see lazy initialization (page 75) for more details
- /prudence/setup/: handles the parsing of routing.js (page 20) and settings.js (page 117)

Calling Conventions by Language

```
conversation.redirectSeeOther('http://newsite.org/')
caching.onlyGet = true
application.globals.put('name', 'example') // Nashorn: application.globals['name'] = 'example
```

If you're using JVM 8, with Nashorn as your JavaScript engine, then you can treat JVM maps as dictionaries: "application.globals['myapp.data.name']". However, for JVM 7 and Rhino you must use the get- and put- notation: "application.globals.get('myapp.data.name')". In order to support both engines properly, we recommend using the more cumbersome format used by Rhino.

```
conversation.redirectSeeOther('http://newsite.org/')
caching.onlyGet = True // Jepp: caching.setOnlyGet(True)
application.globals['name'] = 'example' // Jepp: application.getGlobals().put('name', 'exampl
```

If you're using the Jepp engine, rather than the default Jython engine, you will need to use get- and set- notation to access attributes. For example, use "application.getArguments()" to access "application.arguments" in Jepp.

```
$conversation.redirect_see_other 'http://newsite.org/'
$caching.only_get = true
$application.globals['name'] = 'example'
```

Prudence's Ruby engine, JRuby, conveniently lets you use the Ruby naming style for API calls. For example, you can use "$application.get_global" instead of "$application.getGlobal".

```
$conversation->redirectSeeOther('http://newsite.org/');
$cahing->onlyGet = TRUE;
$application->globals['name'] = 'example';
```

```
conversation:redirectSeeOther('http://newsite.org/')
caching:setOnlyGet(true)
application:getGlobals():put('name', 'example')
```

You will need to use the get- and set- notation to access attributes. For example, you must use "conversation:getEntity()" to access "conversation.entity".

```
conversation.redirectSeeOther('http://newsite.org/')
caching.onlyGet = true
application.globals['name'] = 'example'
```

```
(.. conversation redirectSeeOther "http://newsite.org/")
(.setOnlyGet caching true)
(.. application getGlobals (put "name" "example"))
```

You will need to use get- and set- notation to access attributes. For example, use "(.getArguments application)" to access "application.arguments". You can also use Clojure's bean form, for example "(bean application)", to create a read-only representation of Prudence's API services.

Entry Points

Prudence sometimes treats your code as *its* API: for this, it requires you to implement specific "entry points" in your code. What these are, exactly, differs per programming language, as detailed below.

JavaScript Uses global functions, with the camel-case naming convention:

```
function handleGet(conversation) {
        return 'Hello, world!'
}
```

Python Uses global functions, with the lowercase-with-underscores naming convention:

```
def handle_get(conversation):
    return 'Hello, world!'
```

Ruby Uses global methods, with the lowercase-with-underscores naming convention:

```
def handle_get conversation
    return 'Hello, world!'
end
```

PHP Uses global functions, with the lowercase-with-underscores naming convention:

```
function handle_get($conversation) {
        return 'Hello, world!';
}
```

Lua Uses global functions, with the lowercase-with-underscores naming convention:

```
function handle_get (conversation)
        return 'Hello, world!'
end
```

Groovy Uses closures tied to global variables (there are no global methods in Groovy), with the camel-case naming convention:

```
handleGet = { conversation ->
        return 'Hello, world!'
}
```

Clojure Uses functions in the current namespace, with the lowercase-with-dashes naming convention:

```
(defn handle-get [conversation]
        "Hello, world!")
```

State and Scope

Prudence is designed to allow for high concurrency and scalability, while at the same time shielding you from the gorier details. However, it's critical that you understand how to access and manage state and scope with Prudence.

application.globals

The application.globals and application.getGlobal APIs are essential for sharing state across the application: *all* application code can access it using the same API, whether it's a manual resource (page 36), a template resource (page 39), a filter (page 108) or a background task (page 102). Because this means that application.globals may be accessed simultaneous by multiple threads, it's important that you understand how to use them concurrently (page 84).

For Configuration Another important use for application.globals is configuration: you can specify configuration settings as app.globals (page 75) in your settings.js, and then easily access them as application.globals one the application is running.

application.globals vs. Global Variables Be *very careful* with global variables in Prudence: their scope is more limited and more temporary than you might think.

Their behavior is actually a very different depending on the execution environment of your code:

- Manual resources (page 36) and filters (page 108): Your global values will persist *as long as the code is not recompiled.* Anything that will trigger recompilation of your code will also involve recreating a new global context for it, discarding all previously used globals. In between these recompilations, your globals will persist, but it's important to remember that they will be shared between all threads, and as thus should be accessed with concurrency techniques (page 84).
- Template resources (page 39): These are made of programs executed from beginning to end, and as such *every execution uses its own global context.* This means that globals are really "local" to the request, like conversation.locals (page 83). But it also means that you don't have to worry about concurrency when using them.

A good rule of thumb is to assume that global variables *never persist beyond a request*, and to use application.globals whenever persistence is required.

> **Note for Clojure**: Clojure doesn't have true global variables: instead all Vars are bound to a particular namespace. On the other hand, all namespaces are global to JVM: there are no thread-limited scopes. In order to allow for threads to have separate global scopes, Scripturian creates on-the-fly namespaces when necessary (this is very lightweight), each with a unique name. Thus, Vars in Clojure end up behaving exactly the same as those in other programming languages.

application.sharedGlobals

The application.sharedGlobals and application.getSharedGlobal APIs have the same scope as application.globals (page 82), except they are shared between all running applications.

Generally, it's not such a good idea to create interdependencies between applications, however there are cases where it is useful:

- If all running applications are connecting to the same database (or another external resource), it can be a good idea to have them share the same connection pool. This allows you to centrally manage the connections on the JVM.
- You can use shared globals to pass messages between running applications: the shared value can be a LinkedBlockingQueue or other thread-safe message-passing mechanism.
- Shared globals can be used for system-wide configuration settings. A good way to set shared globals is via a custom service (page 118).

If you find yourself using application.sharedGlobals a lot, ask yourself if your code would be better off encapsulated as a single application: remember that Prudence has powerful URI routing, support for virtual hosting (page 118), etc., letting you easily have one application work in several sites simultaneously. In short, there might be a better architecture for what you're trying to do.

> **Note for Clojure**: Actually, Clojure namespaces are identical in scope to application.sharedGlobals (all Prudence applications running in the same JVM share the same Clojure namespaces), so you might prefer to use them because they are more idiomatic. Still, application.sharedGlobals can be useful if you want to share globals with other programming languages.

application.distributedGlobals and application.distributedSharedGlobals

The application.distributedGlobals, application.getDistributedGlobal, application.distributedSharedGlobals and application.getDistributedSharedGlobal APIs work similarly to application.globals (page 82) and application.sharedGlobals (page 82), except that they are shared between all members of a cluster.

See the clusters (page 137) chapter for more information.

executable.globals

The executable.globals and executable.getGlobal APIs have a similar scope to application.sharedGlobals (page 82), except that they can be accessed by any code running on the JVM using Scripturian's GlobalScope API. They're useful if you need to share state with non-Prudence code.

conversation.locals

The conversation.locals are not "local" in the same way that code scope locals are. The term "local" here should be read as "local to the conversation." They are indeed "global" in the sense that they can be accessed by any function in your code, but are "local" in the sense that they persist only for the duration of the user request. (Compare with thread locals, which are also "local" in a specific sense.)

Their primary use is for sharing state among code *along the route*: for example, both your resource code and filters (page 108) will have access to the same conversation.locals per request. It's this essential feature that leads them to being used in many ways throughout Prudence:

- URI template variables are captured (page 21) into conversation.locals.
- They can be interpolated (page 115) into target URIs
- They are used for inversion of control (page 29)
- They are used to inject custom values into cache keys (page 64)
- Template blocks (page 42) are captured into conversation.locals

conversation.locals vs. Global Variables Sometimes, your language's global variables function identically to conversation.locals. Consider this template resource (page 39):

```
<html><body>
<% var name = 'Rambo'; %>
<%& '/hello/ %>
</body></html>
```

Our "/libraries/includes/hello.t.html" can look like this:

```
<p>Hello, <%= name %>!</p>
```

As you can see, the global variable "name" is indeed shared between these two fragments, and there's no advantage to using conversation.locals instead.

But you might have to use conversation.locals if:

- ...you need to share state with a filter (page 108)
- ...you are mixing scriptlets written in different languages (page 43)

Synchronization

Sometimes efficient techniques for handling concurrency (page 84), detailed below, can't be implemented, and the only solution is to ensure single-thread access via synchronization.

> Synchronization is *very bad for scalability*: it works against many of the advantages of using multiple threads, processes and nodes. However, in some situations it may be preferable to the alternatives. Use it responsibly. Especially, you want to avoid using it in request threads: it makes a bit more sense for background task threads (page 102).

Prudence makes synchronization easy via the application.getLock, application.getSharedLock and application.getDistributedSharedLock family of APIs. Here's an example of how to use them:

```
var lock = application.getLock('services.remote')
lock.lock()
try {
        doSomethingAtomicallyWithRemoteService()
}
finally {
        lock.unlock()
}
```

Note our use of try/finally: it's in order to *guarantee* that the lock is released, even if an exception is thrown from our code. Unreleased locks will cause your threads to hang.

Also note that locks will stay in memory, *unless they are distributed.* They take very little memory, but if for some reason you are generating many locks over time, you may want to cull them via the application.locks or application.sharedLocks APIs.

Concurrency

Though application.globals (page 82), application.sharedGlobals (page 82), application.distributedGlobals (page 83) and executable.globals (page 83) are all thread safe, it's important to understand how to use them properly.

> **Note for Clojure**: Though Clojure goes a long way towards simplifying concurrent programming, it does not solve the problem of concurrent access to application.globals. You still need to read this section!

The Problem This code might seem OK to you, but it's subtly broken:

```
function getConnection() {
        var connectionPool = application.globals.get('database.pool')
        if (!Sincerity.Objects.exists(connectionPool)) {
                connectionPool = createPool()
                application.globals.put('database.pool', connectionPool)
        }
        return connectionPool.getConnection()
}
```

The problem is that in the short interval between comparing the value in the "if" statement and setting the global value in the "then" statement, another thread may have already set the value. Thus, the "connectionPool" instance you are referring to in the *current* thread would be different from the "database.pool" application.global used by other threads. The valye is thus not truly shared! In some cases, this would only result in a few extra, unnecessary connection pools being created. But in some cases, when you rely on the uniqueness of the global, this can lead to subtle bugs that will appear only under high concurrency.

> Newcomers to concurrent programming tend to think that these kinds of bugs are very rare: another thread would have to set the value *exactly* between our "if" and our "then." This is wrong: if your application has many concurrent users, and your machine has many CPU cores, it can actually happen quite frequently. And, even if rare, your application has a chance of breaking if *just two users use it at the same time.* This is not a problem you can gloss over, even for simple applications.

A Solution Here's a fixed, concurrently safe version of the above code:

```
function getConnection() {
        return application.getGlobal('database.pool', createPool()).getConnection()
}
```

The application.getGlobal call is an *atomic compare-and-set operation*, which *guarantees* that the returned value is a unique instance.

There are two things worth noting:

- The fixed code is *much* shorter than the broken code! This is due to the application.getGlobal API, which internally actually does several operations for you.
- You can see that createPool is *always* called, even if the internal compare-and-set of application.getGlobal fails. This means that in some cases we will be creating a database connection pool and then immediately discarding it. This is a waste, but it's also a security issue: under very high concurrency, we might be creating a lot of these unnecessary connection pools, which could in fact lead to an overload on our database server. We thus *might* be vulnerable to denial of service (DoS) attacks.

A Better Solution We can significantly reduce the number of times createPool is called by checking to see if the application.global is already set:

```
function getConnection() {
        var connectionPool = application.globals.get('database.pool')
        if (!Sincerity.Objects.exists(connectionPool)) {
                connectionPool = application.getGlobal('database.pool', createPool
                   ())
        }
        return connectionPool.getConnection()
}
```

With this code, we might still have createPool called multiple times if the function is called concurrently while the application.global is still unset. The problem will disappear as soon as the application.global is set, but until then we still have a window of vulnerability.

Yet Another Solution The only way to *guarantee* that createPool is not called more than once is make the entire operation atomic by synchronizing it (page 84):

```
function getConnection() {
        var lock = application.getLock('database.pool')
        lock.lock()
        try {
                var connectionPool = application.globals.get('database.pool')
                if (!Sincerity.Objects.exists(connectionPool)) {
                        connectionPool = application.getGlobal('database.pool',
                           createPool())
                }
                return connectionPool.getConnection()
        }
        finally {
                lock.unlock()
        }
}
```

Not only is the code above complicated, but synchronization introduces performance penalties. It's definitely not a good idea to blindly apply this solution.

Concurrent programming is non-trivial and always involves weighing the pros and cons of various solutions for specific situations. So, hybrid approaches are sometimes the best: choose the right solution according to the context.

Finally, then, here's a version of the above code that will allow us to select if we want to use the lock or not:

```
function getConnection(safe) {
        if (safe) {
                var lock = application.getLock('database.pool')
                lock.lock()
        }
        try {
                var connectionPool = application.globals.get('database.pool')
                if (!Sincerity.Objects.exists(connectionPool)) {
                        connectionPool = application.getGlobal('database.pool',
                            createPool())
                }
                return connectionPool.getConnection()
        }
        finally {
                if (safe) {
                        lock.unlock()
                }
        }
}
```

Execution Environments

This section serves as a summary for advanced programmers who are curious about the differences between the many Scripturian-based execution environments available in Prudence.

Programs vs. Entry Points

These two types of execution environment are very different in terms of programming, scopes and threads.

Programming Program code is executed from beginning to end, like a script. The programmer has the choice of defining functions, classes, etc., but does not have to. Programs can be merely a sequence of statements.

Entry point code is *also* executed *once* from beginning to end, however its intended use is within the defined entry points (page 81). So, while you *can* include statements, just like in a program, it would be problematic if they had any side effects other than setting up the entry points. To understand why, see below.

Scopes and Threads Programs get they own unique global scope every time they are executed, which is discarded when the program ends. In the case of configuration scripts, this is rather trivial: those scripts are all run in a single thread, once, and thus always use a single global scope. However, in a threaded environment—template resources (page 39)—you may have many of these global scopes existing at the same for your the same code. To share state between the threads, you would thus need to use application.globals (page 82) or similar mechanisms.

Entry point code works very differently: the code always uses a *single* global scope shared by *all* threads, and indeed the code is executed from beginning to end only when initialized. However, if a recompilation is triggered (say, if the source code has been modified), then a new global scope will be created, and the code will be executed from beginning to end *again*. The implications of this are discussed above in application.globals vs. global variables (page 82).

Performance Considerations As you may conclude from the above, in threaded environments entry point code is executed from beginning to end much less frequently than program code. This means that if performance is crucial, you should prefer to use entry points over programs. However, there are usually a lot of other factors involved, and indeed the differences between these two execution environment types may not be the important once. For example, even though manual resources are implemented as entry points, there will likely be no performance advantage in using them instead of template resources, even though the latter involves a new global scope per each request. Creating a global scopes is very lightweight for most programming language engines: the differences in performance depend more strongly on what kind of code you chose to execute in the program.

The only Prudence feature that offers a clear choice between these two types is background tasks spawned via API (page 103). There, you can decide as to whether you want to use a program or an entry point.

Comparison Table

Feature	Type
Configuration scripts (page 117)	programs
Manual resources (page 36)	entry points
Template resources (page 39)	programs
Background tasks (page 102)	programs or entry points
Filters (page 108)	entry points
Cache key template plugins (page 64)	entry points
Scriptlet plugins (page 44)	entry points

Debugging

Let's start by admitting that the debugging facilities for dynamic languages on the JVM are lacking: breakpoints are not supported and there are no language-specific runtime inspection tools. However, there are enough other options for debugging in Prudence that you should be able to find all your bugs. Squashing them is up to you!

Live Execution

This immensely powerful (and potentially dangerous) tool (page 50) allows you to execute arbitrary code in live running applications. There are, of course, endless ways in which you can use this feature for debugging. Below are a few common examples.

Dumping Values

Anything you print to standard output will be returned to you:

```
curl --data '<% println(application.globals.get("db.connection")) %>' http://localhost:8080/m
```

You can also dump values to the log (page 88):

```
curl --data '<% application.logger.info(application.globals.get("db.connection")) %>' http://
```

Scheduling Background Tasks

Here's an example of dumping a value to the log every 5 seconds using a background task (page 102). First let's create "program.js":

```
<%
document.require('/prudence/tasks/')
Prudence.Tasks.task({
        fn: function() {
                application.logger.info(application.globals.get('db.connection'))
        },
        repeatEvery: '5s'
})
%>
```

Then execute it:

```
curl --data-binary @program.js http://localhost:8080/myapp/execute/
```

Debug Page

The debug page (page 92) contains a lot of useful debugging information. It's easy to get it just by causing an exception on purpose:

```
curl --data '<% throw null %>' http://localhost:8080/myapp/execute/
```

Logging

Logging is by far the most useful weapon in your debugging arsenal. Learn how to use it and use it well.

Some programming languages—Python and Ruby—have their own standard logging mechanisms, which you are welcome to use if you prefer. However, consider using Prudence's (actually Sincerity's) logging system instead: it is especially powerful and flexible, and moreover is shared by all programming languages running in Prudence.

The default logging configuration (page 90) is production-ready: it will use the following rolling log files under your container's "/logs/" directory:

- "common.log": The rolling log file shared by all applications and libraries. If they don't specify their own, it will default to this (it's configured as attached to the root logger.)
- "application-[name].log": A rolling log file per Prudence application, where "[name]" is the application's subdirectory name.
- "web.log": A rolling log file of all incoming HTTP requests in NCSA-style.
- "service-prudence.log": This will be available if you're running Prudence as a service (page 148). Note that it is configured separately, and does not in fact use JVM logging.

Introduction

It's important to understand that there's a clear separation of concerns between *what* to log and *how* to log it. Programming should be concerned only with the *what*: the use of the API during runtime and the content of messages. Administration should be concerned only with the *how*: configuration of the logging system, which occurs during bootstrap.

From a programming perspective, you only care about "loggers," the entities through which you send log messages. The programmer doesn't actually know where the log messages are going. (While it *is* possible to change the logging configuration via API during runtime, it is not a common practice.)

"Appenders" are the mechanisms that actually write/store/send the messages somewhere, whether it's to a file, to a database, or over the network. *The API is entirely ignorant as to what appenders are attached to which loggers*: appenders are purely a matter of configuration.

API

Loggers are identified by a unique name in your container, and are configured globally.

The logger names are hierarchical, whereby the hierarchy is specified using ".". For example, "myApp.system.mail" is a child logger of "myApp.system". A child logger inherits its parent's configuration, which it can then override. Parents inherit from their parents, etc., up to what is called the "root logger," which is the logger with an empty string name.

Every log message you send has a "log level." The lower the level, the more specific the message. Loggers are configured with a minimum level: they will log only messages *equal to or above* it. So, the lower the minimum log level configured for the logger, the more chatter you should get. The following levels are supported:

- **severe** (1000): These are *very important* messages, usually about errors or other problems. You'd likely want all loggers to include this level. You should use this level sparingly.
- **warning** (900): These are important, but not crucial. Usually used for temporary errors or for problems that do not cause failure.
- **config** (800): These express change of state (configuration), and as such probably don't happen that often. They most likely appear during the initial bootstrapping, but can happen while the application runs if something causes a subsystem to restart or reconfigure itself.
- **info** (700): These informational messages are used for keeping track of the application's runtime health: they don't specify a problem or a change, only report normal activity. This is the most commonly used message.
- **fine** (500): This and the following two levels should be reserved for debugging: you normally won't go beyond "info" unless you're specifically trying to debug a problem, as this would cause a lot of chatter in your logs.
- **finer** (400): More minute debugging.
- **finest** (300): Most minute debugging.

application.logger and application.getSubLogger application.logger gives you access to a unique Logger for the application, with the "prudence." prefix. The name of the logger can be configured in settings.js (page 75), and defaults to the name of the application's subdirectory.

The easiest way to use this API is via the methods named after the log levels:

```
application.logger.info('An info-level message')
application.logger.warning('Something bad happened!')
```

application.getSubLogger can be used to easily access child loggers of the application.logger:

```
application.getSubLogger('email').info('An info-level message')
```

If your application logger name is "cms", then the above logger name would be "prudence.cms.email".

Other Loggers Instead of using application.logger and application.getSubLogger, you can access the logging API directly to retrieve a logger with any name:

```
importClass(java.util.logging.Logger)
var logger = Logger.getLogger('myApp.system.mail')
logger.info('My logger!')
```

/prudence/logging/ If you're using JavaScript, it's recommend that you use this friendly wrapper, which adds many convenient additions over the raw logging API.

Here are examples of simple logging, where you can refer to the log levels using simple strings, or use dedicated methods named after the log levels:

```
document.require('/prudence/logging/')
var logger = new Prudence.Logging.Logger()
logger.log('info', 'An informational message')
logger.fine('A fine message')
logger.warning('A warning')
new Prudence.Logging.Logger('email').info('Hello from the Email subsystem')
```

String interpolation is supported for all methods, using /sincerity/templates/:

```
logger.severe('Hi, {0} and {1}!', 'mom', 'dad')
logger.info('Hi, {name}!', {name: 'mom'})
logger.finest('Hi, {name}!', function(original, key) {
        if (key == 'name') {
                return 'mom'
        }
        return null
})
```

Quick JSON dump, in short form (single line, condensed) or long form (multi-line, indented):

```
var x = {test: 'hello', more: [1, 2, 3]}
logger.dumpShort(x, 'Our object in short form')
logger.dumpLong(x, 'Our object in long form')
logger.dump(x, 'This is an alias for dumpShort')
```

Log exceptions using full or partial stack trace (supports both JavaScript and JVM exceptions):

```
try {
        nothing()
}
catch (e) {
        logger.exception(e)
}
```

Also very useful is automatic timing of sections of code, which will emit a message when the function starts, when it ends with the execution duration (either due to successful completion or due to a uncaught exception):

```
logger.time('The loop', function() {
        var x = 1
        for (var i = 0; i < 100000; i++) {
                x /= 2
        }
})
```

Performance Considerations Log messages will not be written if they are configured to be filtered out. However, your code is still constructing the message and sending it to the logging system, which may involve a performance hit.

The easiest way to optimize is to check for the log level before constructing the message:

```
if (logger.isLoggable(java.util.logging.Level.FINE)) {
        var data = heavyFetchFromDatabase()
        return 'This function was called because the "fine" log level is permitted'
}
```

If you're using the /prudence/logging/ library, as a shortcut you can send an anonymous function:

```
logger.fine(function() {
        var data = heavyFetchFromDatabase()
        return 'This function was called because the "fine" log level is permitted'
})
```

Obviously, this is cumbersome to always follow this format, so it's best used only if constructing the message indeed involves heavy lifting.

Configuration

You configure all your loggers and appenders under your container's "/configuration/logging/" directory, which is handled by Sincerity's logging plugin, so it's recommended you read the documentation there. The plugin has useful add-ons for centralizing logging, for example to a dedicated Log4j server, or directly to MongoDB.

The best way to learn how to configure logging is to go through the default logging configuration. As usual, Prudence and Sincerity use "configuration by script," so in fact you'll be using the /sincerity/log4j/ library.

Configuration Log Levels In order to maintain conformity, the API used throughout Prudence is the standard JVM logging API (often called JULI: "java.util.logging Interface"). *However*, the logging system used by default in Prudence is Sincerity's logging plugin funnels several different logging APIs into Apache Log4j.

> Unfortunately, until the JVM finally included a logging standard, there were already other popular logging APIs. Also unfortunately, the introduced standard is simply not as powerful as Log4j, and we did not want to compromise on power. Fortunately, the SLF4J library allows Sincerity to mimic all these various APIs while implementing them in Log4j, so you get the best of all worlds as well as a uniform system.

The outcome of this is slightly annoying: the log level names used in the configuration files (Log4j) are different from those used in the API (JULI), and indeed the configuration is coarser than the API. Here's how they match up:

Configuration Log Level	API Log Level
fatal	
error	severe
warn	warning
	config
info	info
debug	fine
	finer
trace	finest

So, configuring a logger's level to "info" will mean that it will log the following API levels: "info," "config," "info," "warning" and "severe."

web.log To learn how to configure the message format for this special logger, see the log service (page 123).

Other Loggers Various subsystems within Prudence use their own loggers. The default configuration files set these to reasonable defaults, which you can modify:

- Restlet logs its internal application events to "org.restlet.Application.[name]". You can access this logger in your code using the application.application.logger API. By default, it is attached to the application's appender and set to level "warn".
- Jetty and Hazelcast both feature robust event logging.

Quick-and-Dirty Logging to the Console

Sometimes, in the midst of intensive debugging, setting up proper logging is just too time consuming. It would also be wasteful, because you know you will quickly delete the logging message once you find the bug.

In such situations the most straightforward solution is to output your message to the console. Since often Prudence captures standard output (for example, in template resources), you may need to override any captures and go straight to the console:

```
java.lang.System.out.println('Desperate times call for desperate measures!')
```

Note that there is no console if you're running Prudence as a service/daemon (page 148) (it is captured to the wrapper log). In those cases you either want to switch to running Prudence from the console, or use the logging API instead.

JVM Logging

If you prefer, you can use the JVM's built-in logging instead of the Sincerity plugin. It is less flexible and much less scalable, but may be preferable in some cases. To configure it, create a configuration file, for example "logging.properties", and define the properties of the handlers and loggers:

```
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
java.util.logging.FileHandler.pattern = log.%u.%g.txt
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.SimpleFormatter.format=%1$tb %1$td, %1$tY %1$tl:%1$tM:%1$tS %1$Tp %2$s %4$s
java.util.logging.ConsoleHandler.level = ALL
com.threecrickets.prudence.cache.level = ALL
.level = INFO
```

Then, start Prudence using a switch to point at that configuration:

```
JVM_SWITCHES=-Djava.util.logging.config.file=logging.properties sincerity use mycontainer : s
```

For more information, see the JVM documentation.

Best Practices

There are many different styles of logging, but here are some good rules of thumb:

- Remember that *people* read the logs! This can mean other developers in your team, but it can also be you in the future, at a time when you may forget why you added a particular log message. Because of this, you *always want every log message to be clear in and of itself.* For example, if you're dumping the contents of a variable, prefix it with the name of the variable, as well as the name of the function or file from which it is logged. Otherwise, it would be impossible for others to find out where the dump is coming from and how to disable it.
- Use log levels wisely: if you're adding a logging message purely for debugging, don't set it to "info", otherwise it will remain as distracting chatter during normal operations. Use "fine" instead.
- Some log messages are very specific, and added during debugging, but would be distracting once the bug is solved. In that case make sure to disable the message when you're dong debugging by either switching it to the "finest" level, commenting out the code, or just deleting it. *Make sure not to commit code that outputs distracting log messages.*
- A good, clear log is an invaluable debugging tool. A confusing, messy log is worthless!

Debug Page

Uncaught exceptions will cause Prudence to generate a 500 "internal error" HTTP status. When debug mode (page 72) is enabled, this will come with a generated HTML page full of useful debugging information:

- Error stack trace, with links to the source code
- HTTP request warnings
- HTTP request information, including all headers parsed and organized:
  - URIs
  - Query params
  - Media preferences for content negotiation
  - Conditions for content negotiation
  - Caching directives
  - Client identification
  - Cookies
  - Payload
- conversations.locals (page 83)
- HTTP response attributes
- Application information:
  - application.globals (page 82)
  - application.sharedGlobals (page 82)
  - executable.globals (page 83)
- Scripturian:
  - Executable attributes
  - Execution context attributes
- Low-level JVM stack trace

Obviously, this information may include private data, so you'll want to turn this debug page off for production deployments.

Monitoring

One of the advantages of having Prudence running on the JVM is its powerful built-in monitoring capabilities based on JMX technology. Using the VisualVM tool, you can get a comprehensive view of the JVM's memory use over time, examine garbage collection, threading behavior and performance, and perform basic profiling. It's an invaluable tool for debugging memory leaks and performance bottlenecks.

Threads

Here are a few common thread groups you will see for the Prudence's subsystems:

- qtp: Jetty server's selectors and acceptors
- HttpClient: Jetty client
- cron4j: cron4j
- hz: Hazelcast
- Wrapper: The Sincerity service plugin wrapper
- JXM and RMI: These are used by JMX itself

The "thread dump" button is very useful for getting a picture of what all threads are doing at a particular moment.

Management Beans

If you install the MBeans plugin, you will gain access to useful information and functionality exposed by Prudence's subsystems:

- java.lang: General JVM tools, such as operating system information as well as class loader, memory, compilation and threading statistics; use the "gc" operation in the "Memory" type to cause immediate garbage collection
- java.nio: Some useful I/O statistics
- java.util.logging: Allows you to query the JULI API
- org.apache.logging.log4j2: Available when you are using the Sincerity logging plugin; access to the logging implementation; note that you can also install a dedicated GUI using the JConsole plugin: see the Log4j management documentation for more information
- org.tanukisoftware.wrapper: Available when you are using the Sincerity service plugin; operations allows you to stop/restart the service

Remote JMX

Connecting to a running Prudence JVM over TCP/IP is not so trivial. See the "extras" section in the Sincerity service plugin documentation for a guide.

Memory Leaks

These are sometimes difficult to debug: by the time you get an OutOfMemoryError, it might be impossible to connect over JMX.

jmap and jhat You should familiarize yourself with these two powerful heap analysis tools.

Use jmap to get a live dump from a running JVM:

```
jmap -dump:live,format=b,file=/full/path/to/dumps/prudence.bin <pid>
```

If you're getting connection errors, try the following:

- In Linux, you might need to enable inter-process access for ptrace:

  ```
  echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
  ```

- Try using root access to run jmap
- Try adding the "-F" switch (though it will likely only create a partial dump)

When opening the dump using jhat, you might get an OutOfMemememoryError (again!) if the file is very large. Increase available memory like so:

```
jhat -J-mx2000m prudence.bin
```

Point your web browser to http://localhost:7000/ to see jhat's analysis.

More Logging You can enforce a heap dump to file upon OutOfMemoryError using the following JVM command line switches:

```
JVM_SWITCHES=\
        -XX:+HeapDumpOnOutOfMemoryError \
        -XX:HeapDumpPath=/full/path/to/logs/ \
        sincerity start prudence
```

Likewise, it's a good idea to turn on the garbage collector log:

```
JVM_SWITCHES=\
        -Xloggc:/full/path/to/logs/gc.log \
        -XX:+PrintGCDetails \
        -XX:+PrintTenuringDistribution \
        sincerity start prudence
```

If you're using the Sincerity service plugin, you can enable both of these features by creating a "/configuration/service/jvm/memory-monitoring.conf" file:

```
# Out-of-memory heap dumps
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath={sincerity.container.root}/logs/

# Garbage collection log
-Xloggc:{sincerity.container.root}/logs/gc.log
-XX:+PrintGCDetails
-XX:+PrintTenuringDistribution
```

Describing APIs

An important aspect of managing a RESTful API is documenting it. As with any API documentation, it is useful for humans, both users and developers. But it's also useful for consumption by clients, which can use a properly-formatted description to automatically generate calling interfaces for your APIs.

Prudence allows you to annotate your code with special comments from which a description data can be generated. RESTful description blocks in Prudence must be delimited with "/***" and "*/" (triple-asterix comments). This allows the block to be ignored by the programming language, as a regular comment, while differentiating it from code documentation, which is usually delimited using "/**" and "*/" (double-asterix comments, for example see Sincerity's JsDoc plugin). Within these comments, Prudence will look for special annotation tags beginning with "@", similarly to how the JavaDoc specification works. Annotation values can extend across multiple comment lines, and some require special formats, as detailed below.

Even if you don't intend to support a description technology, adhering to such annotation standards is very useful for documenting your URI-space for humans, and course will allow you to easily support such technologies in the future.

Generating Description Data

The "prudence" Sincerity command has a "describe" sub-command that will automatically gather annotations, parse them, and generate description data:

```
sincerity prudence describe myapp
```

Syntax errors will be reported if discovered.

By default, description files will be written to the application's "description" subdirectory, but you can also specify a different subdirectory name:

```
sincerity prudence describe myapp api-docs
```

Swagger

Prudence supports Swagger (a trademark of Reverb Technologies, Inc.), an emerging standard for RESTful API description with a broad ecosystem of tools and wrappers.

At the very least, you will need a "@swagger" annotation specifying the type for the description block. The documents generated by the "describe" tool (page 94) are fully-compliant JSON, indented for human readability, with a ".json" extension.

Most annotations match the fields in the Swagger specification, version 1.2, and are explained there. However, Prudence automates many values, and additionally provides sensible defaults.

@swagger info

Provides general information about your API.

You can have multiple "@swagger info" blocks in your application, though the parser will merge them all together. It's good practice to place these blocks in your settings.js, routing.js, or both.

```
/***
 * RESTful API for My Application.
 *
 * @swagger info
 * @apiVersion 1.0
 * @basePath http://myapp.org/api
 * @title My Application
 * @description This API provides access to
 *              My Application's resources
 * @license GNU Lesser General Public License 3.0
 * @licenseUrl http://www.gnu.org/licenses/lgpl.html
 * @contact info@threecrickets.com
 */
```

"@swaggerVersion", "@apiVersion" and "@basePath" are used as defaults for the "@swagger api" sections, which you may override there.

Additional annotations will get default values, which you may override:

- @swaggerVersion: 1.2

@swagger api

Represents a group of APIs, often intended to mean a single RESTful resource, though it may be useful to group several resources together in some cases. In Swagger, all APIs within a group are described in a single JSON document.

It's good practice to place these blocks at the top of the source code file for a resource.

```
/***
 * This resource represents a person in the database. Persons have unique
 * IDs defined by integers.
 *
 * @swagger api
 * @description A person
 * @produces application/json text/plain
 * @consumes application/json text/plain
 */
```

Values for "@produces" and "@consumes" may be specified in one annotation, or in several:

```
@produces application/json
@produces text/plain
```

Additional annotations will get default values, which you may override:

- @resourcePath: if not specified, will be automatically generated according to the filename in which this description block appears; Prudence will generate the JSON file according to its value:
  - If it ends with a "/", will be considered a directory name, and Prudence will put a "index.json" file in it.
  - Otherwise, a ".json" extension will be added to the value to create the filename
- @swaggerVersion: from the "@swagger info" block
- @apiVersion: from the "@swagger info" block
- @basePath: from the "@swagger info" block

@swagger operation

Represents a single API call, a RESTful verb on a URL. An operation must be associated with a "@swagger api" block, though if one does not exist it will be inferred and generated for you (though without "@description", "@produces", "@consumes", etc.).

It's good practice to place these blocks right before the appropriate handler code.

```
/***
 * Retrieve a person's data according to its ID.
 *
 * @swagger operation
 * @summary Retrieves a person
 * @notes Accesses the MongoDB database
 * @path /person/{personId}/
 * @method GET
 * @nickname getPerson
 * @parameter path personId string The person's ID
 * @type Person
 * @responseMessage 400 Error Invalid data
 * @responseMessage 404 null Not found
 */
function handleGet(conversation) {
        ...
}
```

The link between a "@swagger operation" and its "@swagger api" is via the "@resourcePath" annotation.

The "@type" annotation matches a "@swagger model" (see below).

You may include none or several "@parameter" annotations, with the following attributes:

1. The parameter type: "path", "query", "body", "header" or "form"
2. Name
3. Type
4. Optional: description

You may include none or several "@responseMessage" annotations, with the following attributes:

1. HTTP status code (an integer)
2. Type: a "@swagger model", or "null" to specify no content
3. Optional: description

Additional annotations will get default values, which you may override:

- @resourcePath: if not specified, will use the value for the previous "@swagger api" block; if there is not such block, it will be automatically generated according to the filename in which this description block appears

@swagger model

Represents a data model. As with "@swagger operation", it is associated with a "@swagger api" block.

```
/***
 * Person model.
 *
 * @swagger model
 * @id Person
 * @property name string
 * @property lastname string
 */
```

You may include none or several “@parameter” annotations, with the following attributes:

1. Name
2. Type

Serving Swagger

You can serve your Swagger description statically (page 46) via routing.js:

```
app.routes = {
        ...
        '/api-docs/*': {
                type: 'static',
                root: 'description'
        }
}
```

However, most Swagger clients will require Cross-Origin Resource Sharing (CORS), which you can support using Prudence’s CORS filter (page 112):

```
app.routes = {
        ...
        '/api-docs/*': {
                type: 'cors',
                allowOrigin: '*',
                next: {
                        type: 'static',
                        root: 'description'
                }
        }
}
```

You can test your Swagger description using the Swagger UI application. A demo is available here.

Working with Swagger Clients

Most Swagger clients will require your API to use Cross-Origin Resource Sharing (CORS), which you can support using Prudence’s CORS filter (page 112). For example, if you’re implementing your API as manual resources (page 36), you can just wrap your entire “manual” route type with the filter:

```
app.routes = {
        ...
        '/*': [
                {
                        type: 'cors',
                        allowOrigin: '*',
                        allowMethods: ['GET', 'POST', 'PUT', 'DELETE'],
                        allowHeaders: 'Content-Type',
                        next: 'manual'
                },
                'templates',
                'static'
        ]
}
```

An additional requirement for CORS is that your resources support the HTTP “OPTIONS” verb. This can be easily done with an empty handler:

```
function handleOptions(conversation) {
        return null
}
```

Of course, instead of using the CORS filter, your handlers can actually set the appropriate CORS headers as is appropriate. See custom headers (page 55) for more information:

```
function handleOptions(conversation) {
        conversation.responseHeaders.set('Access-Control-Allow-Origin', '*')
        conversation.responseHeaders.set('Access-Control-Allow-Methods', 'GET, POST, PUT, DEL
        conversation.responseHeaders.set('Access-Control-Allow-Headers', 'Content-Type')
}
```

FAQ

Please also refer to the FAQs for Sincerity and Scripturian.

Technology

How is Prudence different from Node.js?

Both are server-side platforms for creating network servers using JavaScript. Both have evolved to support package managers: Sincerity for Prudence, npm for Node.js.

But that's pretty much where the similarities end.

Purpose and Architecture Prudence is a comprehensive platform for REST services, such as web pages and RESTful APIs. Node.js is a minimalist platform for asynchronous services, such as streaming video and audio servers. These are *very* different use cases.

First, note that *both* use non-blocking servers at the low level. So they're *both* asynchronous in that particular respect.

On top of the multi-threaded server Prudence builds a RESTful application environment, using Restlet to handle the many intricacies of HTTP. Why multi-threaded? Because generating HTML is logically a *single-event* procedure: at the moment you get the client's request, you generate the HTML content and send it immediately. Thus, via a managed, configurable thread-pool, Prudence can leverage multi-core CPUs as well as highly-concurrent database backends to serve several several user requests simultaneously.

Node.js could not be more different: it is by design *single*-threaded and event-*driven*: requests are never handed simultaneously, and only a single CPU core would ever be used by the HTTP server itself.

Seems odd? Actually, this "raw" architecture makes a lot of sense for streaming applications: as opposed to HTML pages, streams are always *multi*-event procedures, each event generating a "chunk" of the stream that saturates the socket with data. There is no advantage to using more than one thread if a single thread is already taking up all the bandwidth. In fact, thread synchronization could introduce overhead that would slow the server down. Really, your only variable in terms of scalability is the size of the chunks: you'll want them smaller under high load in order to degrade performance fairly among clients. That said, other libraries you might use from Node.js *can and do* use threads: this is useful for CPU-bound workloads such as video encoding.

Node.js is great for its intended use case. It's vastly easier to write event handlers in JavaScript than in C/C++, and JavaScript also makes it easy to bootstrap the server. If you're writing a streaming asynchronous server, we highly recommend Node.js.

> If you like Node.js but JavaScript is not your favorite dynamic language, similarly excellent event-driven platforms are available for other languages: check out Tornado and Twisted for Python, and EventMachine for Ruby.

You're Doing It Wrong That said, it's very odd that Node.js has become a popular platform for the *non*-streaming, data-driven web: the Express framework, for example, provides some minimal RESTful functionality on top of Node.js. But event-driven servers are not designed for REST, and are in fact quite a bad fit: consider that in a single-threaded runtime, if a single event handler hangs, the whole server will hang. Node.js deals with the problem by in effect offloading the thread pools to external libraries, written in C++. For example, Node.js's database drivers handle queries in their own connection thread pools, and push events to Node.js when data is available. Still, in your Node.js JavaScript event handlers, you have to take extra care not to do any time-consuming work: a delay you cause would affect *all* operations waiting their turn on the single-threaded event loop. Thus, to create a properly scalable Node.js application, you must have event handlers with no risky "side effects": that heavy lifting is left to the C++ libraries. The bottom line is that you have an illusion of single-threadedness: you've merely shifted the workload to the drivers.

And even if you did a good job in JavaScript, your implementation will not in *any way* be more scalable for this use case than by using a thread pool: users still need to wait for their requests to complete, and databases still have to return results before you can complete those requests. There's nothing in an event-driven model that changes these essential facts. (It's worth repeating again: both Prudence and Node.js use non-blocking I/O servers; Node.js having an event-driven programming model does not allow it to magically handle more concurrent connections than other platforms can.)

You can add some parallelism by running multiple Node.js processes behind a load balancer, but the problems quickly multiply: each process loads its own version of those C++-written drivers, with its own connection thread pool, which can't be shared with the other Node.js processes. In Prudence, by contrast, the same pool is trivially shared by all requests. (Prudence's threads can also share an in-process memory-based cache backend for the best possible caching performance at the 1st tier.)

It should be clear that Node.js is simply the wrong tool for the job. So, why is Node.js so misused? One reason is that its raw architecture is attractively simple: multi-threaded programming is hard to get right, single-threaded easy. JavaScript, too, is attractive as a language that many programmers already know. So, despite being a problematic web platform, it's one in which you can build web services quickly and with little fuss, and sometimes that's more important than scalability or even robustness (especially if the goal is to create in-house services). But another reason for Node.js' popularity is more worrying: ignorance. People who should know better heard that Node.js is "fast" because it's "asynchronous" and think that would lead to faster page load-times for web browsers and the ability to handle more page hits. That's a very wrong conclusion. You can do great REST in Node.js, but would have to work against the platform's limitations for the scenario.

We believe that Prudence is a much more sensible choice for the RESTful web. Beyond the basic multi-threaded model, also consider Prudence's many features aimed specifically at RESTful scalability, such as integrated caching (page 61), full control of conditional HTTP (page 39), and clusters (page 137) for scaling horizontally in the "cloud." Check out our Scaling Tips article, too, which is useful even if you don't choose Prudence.

Technology and Ecosystem Prudence was designed specifically for the JVM, to provide you with access to its rich and high-quality ecosystem, to leverage its excellent concurrency libraries and monitoring/profiling capabilities, and to be portable and integrative. The JVM platform is very mature and reliable, as are many of the libraries that Prudence uses, such as Jetty, Restlet and Hazelcast. It's easier to connect C/C++ libraries to Node.js, but on the other hand it's easier to write and deploy extensions in Java/Scala/Groovy/Clojure for Prudence.

And Prudence is not just JavaScript: it supports many dynamic languages running on top of the JVM—Python, Ruby, PHP, Lua, Groovy and Clojure—as well as their respective ecosystems. That said, it does give special love to JavaScript: Sincerity comes with a rich foundation library, offering essentials such as OOP and string interpolation, as well as friendly wrappers for powerful JVM services, such as concurrent collections and cryptography.

Node.js is JavaScript-centric, and though it has a much younger and narrower ecosystem, it is vibrant and quickly growing.

No Benchmarks for You In terms of sheer computational performance, Node.js has done well to leverage the "browser wars," which have resulted in very performative JavaScript interpreters and JIT compilers. However, it's worth remembering that these engines have been optimized for web browser environments, not servers, and thus have very limited support for threading. By contrast, Prudence can run JavaScript on your choice of either Nashorn or Rhino, which can both use the JVM's excellent concurrency management. Nashorn promises excellent performance, on par with Java code in some cases. Rhino is not as fast, but still performs well and is very mature.

But a comparative benchmark would make little sense. Node.js' single-threaded model *really needs* blazing-fast language performance, as it directly affects its scalability. Prudence's multi-threaded model and RESTful expectations mean that it's rarely CPU-bound: for example, you spend orders of magnitude more time waiting for database backends to respond than for functions to be called. For the web page use case, smart architecture—and smart caching—are *far* more important for scalability than language engine performance.

Note, too, that all the performance-critical parts of Prudence are written in Java, just as they are written in C++ for Node.js. There should be no noticeable performance differences in the basic request handling performance.

And it's important to remember—and repeat to yourself as mantra—that *performance does not equal scalability.* Prudence's default HTTP engine is Jetty, which is designed from the ground-up for scalable web services, and it behaves smartly under high volumes, even if it might not seem so under trivial and misleading localized benchmarks. It's also easy to replace Jetty with other servers (page 100).

Conclusion Choose the right tool for the job! Node.js is a great choice for streaming and streaming-like services, and Prudence—we hope you'll agree—is a great choice for web services and RESTful APIs.

How is REST better than RPC?

Well, it's not *necessarily* better. One important advantage is that it works inside the already-existing, already-deployed infrastructure of the World Wide Web, meaning that you can immediately benefit from a wide range of optimized functionality. We discuss this in greater detail in The Case for REST (page 152).

But we don't advocate REST for every project. See this discussion (page 157) for one reason why RPC may be more appropriate. Choose the right tool for the job!

Why does Prudence support LESS instead of SASS?

Prudence implements LESS using the Less4j library. Unfortunately, there is no SASS implementation that would directly work on the JVM: we could potentially run it using JRuby, but that is simply too big a dependency for a single feature.

However, nothing is stopping you from running SASS from the command line to create your CSS.

An alternative to LESS is ZUSS: smaller, but not quite as comprehensive. Code to support it is included in Prudence, though you must install the ZUSS Jar yourself.

How do I use other HTTP servers instead of Jetty?

We recommend Jetty and install it by default, but in some cases it may be worth using alternatives: for example, you may want to experiment with different performance characteristics for certain deployments or make comparative benchmarks.

Fortunately, Restlet decouples its API from the server library, and comes with extensions to connect to several server libraries in addition to Jetty. You can install these easily with Sincerity, though note that you would need to exclude the Jetty extension if you do so: Restlet does not currently run if several server extensions are in the classpath at the same time.

For example, let's use the Simple Framework instead of Jetty:

```
sincerity add prudence.example :
        add restlet.simple :
        exclude org.restlet.jse org.restlet.ext.jetty9 :
        exclude org.eclipse.jetty jetty-server :
        install :
        start prudence
```

If Restlet does not find any server extension in the classpath, it will default to its own "internal" server, which is adequate to simple deployments and for testing:

```
sincerity add prudence.example :
        exclude org.restlet.jse org.restlet.ext.jetty9 :
        exclude org.eclipse.jetty jetty-server :
        install :
        start prudence
```

Performance and Scalability

How well does Prudence perform? How well does it scale?

First, recognize that there are two common uses for the term "scale." REST is often referred to as an inherently scalable architecture, but that has more to do with project management than technical effectiveness. This difference is addressed in the "The Case for REST" (page 152).

From the perspective of the ability to respond to user requests, there are three aspects to consider:

1. Serving HTTP Prudence comes with Jetty, an HTTP server based on the JVM's non-blocking I/O API. Jetty handles concurrent HTTP requests very well, and serves static files at scales comparable to popular HTTP servers.

2. Generating HTML Prudence implements what might be the most sophisticated caching system (page 61) of any web development framework. Caching is truly the key to scalable software. See "Scaling Tips" (page 159) for a comprehensive discussion of the role of caching.

3. Running code There may be a delay when starting up a specific language engine in Prudence for the first time in an application, as it loads and initializes itself. Then, there may be a delay when accessing a dynamic web page or resource for the first time, or after it has been changed, as it might require compilation. Once it's up and running, though, your code performs and scale very well—as well as you've written it. You need to understand concurrency and make sure you make good choices to handle coordination between threads accessing the same data. If all is good, your code will actually perform better throughout the life of the application. The JVM learns and adapts as it runs, and performance can improve the more the application is used.

All language engines supported of Prudence are generally very fast. In some cases, the JVM language implementations are faster than their "native" equivalents. This is demonstrable for Python, Ruby and PHP. The reason is that the JVM, soon reaching version 8, is a very mature virtual machine, and incorporates decades-worth of optimizations for live production environments.

If you are performing CPU-intensive or time-sensitive tasks, then it's best to profile these code segments precisely. Exact performance characteristics depend on the language and engine used. The Bechmarks Game can give you some comparisons of different language engines running high-computation programs. In any case, if you have a piece of intensive code that really needs to perform well, it's probably best to write it in Java and access it from the your language. You can even write it in C or assembly, and have it linked to Java via JNI.

If you're not doing intensive computation, then don't worry too much about your language being "slow." It's been shown that for the vast majority of web applications, the performance of the web programming language is rarely the bottleneck. The deciding factors are the usually performance of the backend data-driving technologies and architectures.

I heard REST is very scalable. Is this true? Does this mean Prudence is "web scale"?

Yes, if you know what you're doing. See "The Case for REST" (page 152) and "Scaling Tips" (page 159) for in-depth discussions.

The bottom line is that it's very easy to make your application scale poorly, whatever technology or architecture you use, and that Prudence, in embracing REST and the JVM, can more easily allow for best-practice scalable architectures than most other web platforms.

That might not be very reassuring, but it's a fact of software and hardware architecture right now. Achieving massive scale is challenging.

Errors

How to avoid the "Adapter not available for language: xml" parsing exception for XML files?

The problem is that the XML header confuses Scripturian, Prudence's language parser, which considers the "<?" a possible scriptlet delimiter:

```
<?xml version='1.0' encoding='UTF-8'?>
```

The simple solution is to force Scripturian to use the "<%" for the page via an empty scriptlet, ignoring all "<?":

```
<% %><?xml version='1.0' encoding='UTF-8'?>
```

Licensing

> The author is not a lawyer. This is not legal advice, but a personal, and possibly wrong interpretation. The wording of the license itself supersedes anything written here.

Does the LGPL mean I can't use Prudence unless my product is open sourced?

The GPL family of licenses restrict your ability to *redistribute* software, not to use it. You are free to use Prudence as you please within your organization, even if you're using it to serve public web sites—though with no warranty nor an implicit guarantee of support from the copyright holder, Three Crickets LLC.

The GPL would thus only be an issue if you're selling, or even giving away, a product that *would include* Prudence.

And note that Prudence uses the *Lesser* GPL, which has even fewer restrictions on redistribution than the regular GPL. Essentially, as long as you do not alter Prudence in any way, you can include Prudence in any product, *even if it is not free.* With one exception: Prudence uses version 3 of the Lesser GPL, which requires your product to not restrict users' ownership of data via schemes such as DRM if Prudence is to be included in its distribution.

Even if your product does not qualify for including Prudence in it, you always have the option of distributing your product without Prudence, and instructing your customers to download and install Prudence on their own.

We understand that in some cases open sourcing your product is impossible, and passing the burden to the users is cumbersome. As a last resort, we offer you a commercial license as an alternative to the GPL. Please contact Three Crickets for details.

Three Crickets, the original developers of Prudence, are not trying to force you to purchase it. That is not our business model, and we furthermore find such trickery bad for building trusting relationships. Instead, we hope to encourage you to 1) pay Three Crickets for consultation, support and development services for Prudence, and to 2) consider releasing your own product as free software, thereby truly sharing your innovation with all of society.

Why the LGPL and not the GPL?

The Lesser GPL used to be called the "Library GPL," and was originally drafted for glibc. It is meant for special cases in which the full GPL could limit the adoption of a product, which would be self-defeating. The assumption is that there are many alternatives with fewer restrictions on distribution.

In the case of the Linux project, the full GPL has done a wonderful job at convincing vendors to open source their code in order to ship their products with Linux inside. However, it doesn't seem likely that they would do the same for Prudence. There are so many great web development platforms out there with fewer restrictions.

Note that the LGPL version 3 has a clause allowing you to "upgrade" Prudence to the full GPL for inclusion in your GPL-ed product. This is a terrific feature, and another reason to love this excellent license.

Part II
Advanced Manual

Background Tasks

The primary workload of a web platform is in handling user requests, and in Prudence these are handled in a configurable thread pool managed by the web server (usually Jetty). However, "primary" does not mean "only" or even "most": your application may be doing *lots* of other work both to serve users and to keep itself running properly. In Prudence, we call these "background tasks." They are run in a separate thread pool, and can even be "farmed out" in the cluster (page 137).

The common use cases are:

- Doing work in connection to user requests that *can* happen outside of requests: sending email notifications, updating a statistics database, etc. Even if this happens a bit later than the user request, the user experience would not suffer. By performing this work in the background, you can ensure that the request thread is freed as quickly as possible, which can go a long way towards improving your scalability.

- Doing work for users that *cannot* or *should not* be done within a request thread: encoding videos, interacting with 3rd-party services, performing long searches, etc. All of these could hang the request thread for far too long, which could severely limit your scalability. In these cases you would need to find some way to let the user know that the work they needed is done. Often, results are stored in a database. (Diligence's Progress Service is a generic tool for handling these scenarios.)

- Maintenance tasks unrelated to user requests: cleaning up idle sessions, pruning caches and unused results, sending out digest emails, etc. These tasks are often scheduled to be run on a regular basis: daily, every 5 minutes, etc.

In order to support these diverse use cases, Prudence allows for several ways to schedule and spawn background tasks.

Implementing Tasks

Prudence supports two ways to implement tasks. Note that in both cases, you may implement the task in any supported programming language: it doesn't matter which language calls the API and which language implements the task.

As Programs These tasks are executed from beginning to end. They can, of course, include libraries, define functions and classes, etc. You may optionally send the task an arbitrary "context," which will be accessible via the document.context API.

A simple example:

```
var count = document.context
for (var i = 0; i < count; i++) {
        application.logger.info('#' + i)
}
```

As Entry Points in Programs These tasks are loaded into memory once: Prudence will call a specified entry point every time the task is spawned. The "context" will be provided as an entry point argument. Additionally, entry points allow you to return a value to the caller.

A simple example:

```
function myEntryPoint(context) {
        var count = document.context
        for (var i = 0; i < count; i++) {
                application.logger.info('#' + i)
        }
        return 'finished'
}
```

Performance Considerations Generally, tasks implemented as entry points will be spawned faster (page 86). So, the rule of thumb should be:

- Implement your task as a program if it is supposed to run only once in a while.
- Implement your task as an entry point if it is frequently used.

APIs for Spawning and Scheduling

We'll discuss the higher-level API below (page 105). However, it's useful to start with the lower-level API, so you can better understand the many options.

Libraries

To spawn and/or schedule code in your application's "/libraries/" subdirectory, use the application.executeTask API. As a first example, let's spawn a program task:

```
application.executeTask(
        'cms', '/tasks/hello/', null,
        {name: 'Michael'},
        0, 0, false)
```

Our program would be in "/libraries/tasks/hello.js":

```
application.logger.info('Hello, ' + document.context.name)
```

The first argument to executeTask is the application's name on the internal host (page 31). If you leave it as null, it would default to the current application. The second is the library URI. The third is the entry point name (not used in this example), and the fourth is the optional context.

The final three arguments are for scheduling:

- **delay**: Milliseconds before starting the task; zero means ASAP
- **repeatEvery**: Milliseconds after which the task will be repeated; zero means no repetitions
- **fixedRepeat**: Boolean; not used when "repeatEvery" is zero; if true, "repeatEvery" will be fixed according to the clock; if false, "repeatEvery" will be counted from when each task finishes executing

For our second example, let's use an entry point:

```
application.executeTask(
        null, '/tasks/hello/', 'sayHello',
        {name: 'Michael'},
        0, 0, false)
```

Our program with its entry point:

```
function sayHello(context) {
        application.logger.info('Hello, ' + context.name)
}
```

Literal Scriptlet Code

To spawn and/or schedule literal scriptlet source code as a task, use the application.codeTask API. Note that the source code must provided as scriptlets, identical to the format of template resources (page 39):

```
application.codeTask(
        null, "<% application.logger.info('Hello, ' + document.context.name) %>",
        {name: 'Michael'},
        0, 0, false)
```

The arguments are similar to executeTask, except that literal source code is provided instead of a library name, and there is no entry point name.

This API is useful for generating the task's source code on demand.

The scriptlet format makes it possible to run tasks in any supported programming language:

```
application.codeTask(
        null, "<%python application.logger.info('Hello from Python, ' + document.context) %>
        'Michael',
        0, 0, false)
```

Canceling Tasks

All the APIs return a JVM Future instance, which you can use to cancel the task:

```
var future = application.codeTask(...)
future.cancel(true)
```

Return Values

As mentioned above, entry points can return values to the caller. This is also handled via the Future:

```
var future = application.codeTask(...)
var r = future.get(500, java.util.concurrent.TimeUnit.MILLISECONDS)
```

> Generally, it's not a very good idea to use the returned Future. The advantage of background tasks is in allowing you to release the current thread, but if you block waiting for a task to complete, then you will be doing the opposite. A better way to return values from background tasks store them in a database and attempt to fetch them later, as in the Diligence's Progress Service. However, if you keep the block times very short, the Future has its uses: for example, it provides an easy way to call code in one programming language from another.

Distributed Task APIs

When working with a Prudence clusters (page 137), you can spawn tasks on other nodes in the cluster. This feature enables you to easily create specialized task farms (page 139) for flexible, scalable deployments.

The task APIs have distributed versions: application.distributedExecuteTask and application.distributedCodeTask. The difference is that the distributed versions don't have the three scheduling arguments: you can't delay or repeat distributed tasks. On the other hand, you have two extra arguments for optionally hinting where in the cluster you would want them executed:

- **where**: If this is a string, it is interpreted as a comma-separated list of node tags (page 139). When "multi" is false, it will select the first node that matches any of the required tags. When "multi" is true, it will select *all* nodes that match *any* of the tags. A null value here means to let Hazelcast decide: when "multi" is true, this would mean *all* nodes.
- **multi**: Boolean; when false, will spawn on only *one* member in the cluster; when true, will spawn on *all* members specified; note that when "multi" is true, the return value is a map of Hazelcast Member instances to their appropriate Future instances

In this example, we don't care when in the cluster the task will be executed once:

```
application.distributedCodeTask(
        null, "<% application.logger.info('Hello, ' + document.context.name) %>",
        {name: 'Michael'},
        null, false)
```

In this example, we'll spawn a maintenance task on all nodes:

```
application.distributedExecuteTask(
        'maintenance', '/tasks/cleanup/', null,
        'now',
        null, true)
```

Note that, of course, the application "maintenance" as well as the library "/tasks/cleanup/" have to be present on all nodes in the cluster.

Serialization For distributed tasks, your sent contexts as well as your returned values must be serializable in order for them to be transferred over the network. If you're using JavaScript, this likely means sticking to primitive types: strings and numbers. However, you can serialize the data yourself: say, into JSON when spawning the task, and then from JSON in the task implementation. (The high-level API does this for you.)

High-Level API

If you're using JavaScript, you can use the Prudence.Tasks.task as a shortcut to all the APIs mentioned above. An example:

```
document.require('/prudence/tasks/')
Prudence.Tasks.task({
        uri: '/tasks/cleanup',
        application: 'maintenance'
})
```

It comes with some sweet JavaScript sugar. For example, you can directly spawn functions:

```
function cleanup(context) {
        application.logger.info('Cleaning up: ' + context.time)
}

Prudence.Tasks.task({
        fn: cleanup,
        context: {time: 'now'},
        json: true,
        distributed: true
})
```

Behind the scenes, the above actually serializes the function source code, and calls application.distributedCodeTask (so JavaScript stack closure won't work). The "json: true" param adds some useful magic: it will serialize the context, and then wrap code to deserialize the context around task code. So, the above will work just fine as a distributed task.

(While convenient, it's generally more efficient to invoke an entry point in a library than to serialize function code like so.)

Here's an example of blocking until we get a result:

```
var future = Prudence.Tasks.task({
        uri: '/tasks/math/',
        entryPoint: 'multiply',
        context: [5, 6, 8],
        pure: true,
        block: '1s'
})
print(future.get())
```

Note the "pure: true" param that forces the API to send the context as is: otherwise it will send it as a string to ensure support for serialization. (JavaScript data structures are not, unfortunately, serializable.)

In case you're curious, he's the task for that example:

```
function multiply(elements) {
        var r = 1
        for (var e in elements) {
                r *= elements[e]
        }
        return r
}
```

An Even Lower-Level API

If you have some knowledge of Java programming, you may access the task executor directly via the application.executor API.

Application crontab

Prudence supports crontab files that mimic the format used by that ubiquitous scheduling program.

This facility lets you schedule tasks to run at specific (Gregorian) calendrical times. It works similarly to calling application.codeTask (page 104) with the repetition params, but allows for more succinct, calendrical repetition patterns. Also, the facility is always on, as long as your Prudence container is running: you do not have to call an API to enable it.

To use this facility, place a file with the name "crontab" in your application's base subdirectory. Each line of the file starts with scheduling pattern and ends with the task name. Empty lines and comments beginning with "#" are ignored. Example:

```
* * * * * /tasks/every-minute/
59 23 * * tue,fri <% application.getSubLogger('scheduled').info('It is Tue or Fri, 11:59PM')
```

Notes:

- The crontab files will be checked for changes and parsed once per minute. This means that you can edit this file and have your task scheduling change on the fly without restarting Prudence.
- The scheduler does not check to see if a task finished running before spawning a new instance of it, so that even if a task is not done yet, but it's time for it to be spawned again, you'll have multiple instances of the task running at the same time. If this is problematic, consider using the task APIs (page 103) instead, with the "fixedRepeat" param set to false.

Sending a Context

Optionally, you may add more text after the task name and whitespace: anything there is grabbed as a single string and sent as the context to the task, which can be accessed there using the document.context API. Because crontab is a text file, only textual contexts may be sent, but you can use JSON, XML or other encodings to create complex contexts.

For example:

```
* * * * * /tasks/every-minute/ {"message": "This is a JSON context"}
```

Scheduling Patterns

The scheduling pattern is a series of five settings separated by whitespace:

1. **Minutes** of the hour, 0-59
2. **Hour** of the day, 0-23
3. **Day of the month**, 1-31; the special setting "L" signifies the last day of the month, which varies per month and year
4. **Month** of the year, 1-12; three-letter English month names may be used instead of numbers: "jan", "feb", "mar", etc.
5. **Day of the week**, 0-6; three-letter English day names may be used instead of numbers: "sun", "mon", "tue", etc.

The following rules apply:

- Any of these settings can be "*", signifying that *every* value would match: every minute, every hour, every day, etc.
- Use a slash to match only numbers that divide equally by the number after the slash (can also be used on "*")
- Ranges (inclusive) are possible, separated by hyphens
- Multiple values per setting are possible, separated by commas
- Multiple whole patterns are possible, separated by pipes ("|": a logical "or")

Note that you can schedule the same task on multiple lines, which is *not* equivalent to using the pipe: multiple lines means that multiple task instances might be spawned simultaneously if matched on more than one line. Contrarily, using the pipe counts as a *single* match.

Example Patterns Every minute:

```
* * * * *
```

11:59pm every Tuesday and Friday:

```
59 23 * * tue,fri
```

Every 5 minutes in the morning, between 5 to 8am, otherwise every 30 minutes:

```
*/5 5-7 * * *|*/30 0-4,8-23 * * *
```

The same as above, but with one added to all minutes of the hour:

```
1,6,11,16,21,26,31,36,41,46,51,56 5-7 * * *|1,31 0-4,8-23 * * *
```

System crontab

You can also set up a special crontab to run arbitrary Java static methods and non-JVM system processes, just like with the system cron, by creating a "/component/crontab" file.

Here's an example:

```
0 5 * * * sol.exe
0,30 * * * * OUT:C:\ping.txt ping 10.9.43.55
0,30 4 * * * "OUT:C:\Documents and Settings\Carlo\ping.txt" ping 10.9.43.55
0 3 * * * ENV:JAVA_HOME=C:\jdks\1.4.2_15 DIR:C:\myproject OUT:C:\myproject\build.
    log C:\myproject\build.bat "Nightly Build"
0 4 * * * java:mypackage.MyClass#startApplication myOption1 myOption2
```

The format is different from the application crontabs: see the cron4j documentation for complete details.

Like the application crontabs, it will be enabled as long as the Prudence container is running, and can be edited at runtime.

crontab APIs

For direct access to the crontab, use application.taskCollector for the current application's ApplicationTaskCollector, and application.scheduler for the component-wide cron4j Scheduler.

These APIs let you modify the crontab in memory. However, note that if you edit your crontab, the task table will be reset and reloaded, losing the changes you made via the API.

/startup/

It's often useful to schedule a task to be run as soon as the application starts: to initialize resources, turn on subsystems, do initial testing, etc.

Upon startup, Prudence will automatically spawn "/startup/" as a background task. So, you can create a file called "/libraries/startup.js", "/libraries/startup.py", "/libraries/startup/default.js", etc. For example, here's "/libraries/startup.js":

```
application.logger.info('Our application started!')
```

Tweaking

Learn how to configure the size of the thread pool for task APIs here (page 124).

For crontab configuration, see here (page 125).

Filters

In Prudence, "filters" are route types (page 23) used to add effects to resources. Though you can often code the same effect directly into a resource, filters are decoupled from resources, allowing you to reuse an effect on many resources, which is especially useful with mapping route types (page 22).

Furthermore, filters are the *only* way to add effects to static resources (page 46), which cannot themselves be programmed.

Because it's easy to enable and disable filters just by editing routing.js, filters are often used to add debugging and testing effects.

Tutorial

Filters are implemented similarly to manual resources (page 36): as source code files (in any supported language) with either or both filter entry points: handleBefore and handleAfter. The former is called before all requests reach the next (downstream) route type, and the latter is called on the way back, after the downstream finishes its work.

Let's start with configuring our filter, using the "filter" route type (page 23) in routing.js. We'll put it in front of all our main mapping resources:

```
app.routes = {
        ...
        '/*': {
                type: 'filter',
                library: '/my-filter/',
                next: [
                        'manual',
                        'templates',
                        'static'
                ]
        }
}
```

Now let's create the actual filter in "/libraries/my-filter.js". We'll start with a trivial implementation of the handleAfter entry point (page 81):

```
function handleAfter(conversation) {
        application.logger.info('We got a request for a resource at: ' + conversation.referen
}
```

Our filter doesn't do much yet, but it's easy to test that the code is being called by looking at the log. Of course you can do many other things here, as detailed in the examples below. Most of the conversation APIs are available to you, including the redirection APIs.

Note that while you need to restart your application for the changes in routing.js to take hold, you are free to edit my-filter.js and have the changes picked up on-the-fly.

handleBefore is a bit more sophisticated than handleAfter, in that it also requires a return value:

```
function handleBefore(conversation) {
        application.logger.info('We got a request for a resource at: ' + conversation.referen
        return 'continue'
}
```

Three literal return values are supported, as either a string or a number:

- **"continue" or 0**: Continue to the "next" handler
- **"skip" or 1**: Skip the "next" route type and go immediately to our own handleAfter
- **"stop" or 2**: Stop *our* handling altogether: the same as "skip," but handleAfter is *not* called (note that if a filter was installed *before* us, it would still be called)

Again, you may define both a handleBefore and a handleAfter in the same filter.

Examples

Changing the Request

It may be useful to change user requests for testing purposes. Specifically, we can affect content negotiation by changing the accepted formats declared by the user.

For example, let's say we want to disable compression for all resources, even if clients declare that they are capable of handling it:

```
function handleBefore(conversation) {
        conversation.client.acceptedEncodings.clear()
        return 'continue'
}
```

Easy! Note that this would be *much* harder to achieve retroactively, by changing the response: we would have to decompress all compressed responses. Some effects are much better implemented in handleBefore.

Overriding the Response

Filters can be useful for overriding the response under certain conditions.

The following example always sets the response to a web page displaying "blocked!", unless a special "admin" cookie (page 54) is used with a magic value. It can be used to make sure that certain resources are unavailable for users who are not administrators:

```
function handleAfter(conversation) {
        if (!isAuthorized(conversation)) {
                var content = '<html><body>' + conversation.reference + ' is blocked to you!<
                conversation.setResponseText(content, 'text/html', 'en', 'UTF-8')
        }
}

function isAuthorized(conversation) {
        var cookie = conversation.getCookie('admin')
        return (null !== cookie) && (cookie.value == 'magic123')
}
```

Another, simpler trick, would be to redirect the response:

```
function handleAfter(conversation) {
        if (!isAuthorized(conversation)) {
                conversation.redirectSeeOther(conversation.base + '/blocked/')
        }
}
```

> **Note on *changing* the response:** You might think that filters could be useful to affect the content of responses, for example to "filter out" data from HTML pages. Actually, Prudence filters are *not* a good way to do this, because there's no guarantee that response payloads returned from downstream resources are textual, even if the content is text: they could very well be compressed (gzip) and also chunked. You would then need to decode, disassemble, make your changes, and then reassemble such responses, which is neither trivial nor efficient. *Content* filtering should best be handled at the level of the resource code itself, *before* the response payload is created.

Side Effects

Filters don't have to change anything about the request or the response. They can be useful for gathering statistics or other debugging information.

In this example, we'll gather statistics about agent self-identification: specifically web browser product names and operating systems (via the conversation.client API):

```
document.require('/sincerity/templates/')

importClass(
        java.util.concurrent.ConcurrentHashMap,
        java.util.concurrent.atomic.AtomicInteger)

var logger = application.getSubLogger('statistics')

function handleBefore(conversation) {
        var agent = conversation.client.agentName
        var os = conversation.client.agentAttributes.get('osData')

        getCounter('agent', agent).incrementAndGet()
        getCounter('os', os).incrementAndGet()

        logger.info('Agent stats: ' + application.globals.get('counters.agent'))
        logger.info('OS stats: ' + application.globals.get('counters.os'))
        return 'continue'
}

function getCounter(section, name) {
        var counters = application.getGlobal('counters.' + section, new ConcurrentHashMap())
        var counter = counters.get(name)
        if (null === counter) {
                counter = new AtomicInteger()
                var existing = counters.putIfAbsent(name, counter)
                if (null !== existing) {
                        counter = existing
                }
        }
        return counter
}
```

Here we're storing statistics in memory and sending them to the log, but for your uses you might prefer to store them in a database using atomic operations.

Built-in Filters

Prudence comes with a few built-in filters, each with its own route type (page 23). Many of them are useful specifically with static resources (page 46), and are discussed in that chapter. A few others are more generally useful, and are discussed here.

Inversion of Control (IoC) via Injection

You already know that you can configure parts of your application via application.global presets (page 75). Globals, of course, affect the entire application. However, you may sometimes need *local* configurations: the ability to a specific instance of a resources differently from others. That's where the "injector" route type (page 23) comes in.

Note that because IoC is most often used together with capturing and dispatching, there is a shortcut notation to apply to the "capture" and "dispatch" route types (page 29). However, you can also use injection independently. An example for routing.js:

```
app.routes = {
        ...
        '/user/{name}/': {
                type: 'injector',
                locals: {
                        deployment: 'production'
                },
                next: '@user'
        }
}
```

To access the injected value in your resource code, simply use the conversation.locals API:

```
var deployment = conversation.locals.get('deployment')
if (deployment == 'production') {
        ...
}
```

> You can inject any kind of object using an injector, though keep in mind that the native types of JavaScript may not be easily accessible in other programming languages. For example, if you're injecting a dict or an array, it would not be automatically converted to, say, a Python dict or vector. However, primitive types such as strings and numbers would be OK for all supported languages.

Templates Note that injectors specially recognize Template instances and casts them before injecting. This allows you to interpolate conversation attributes (page 113) into strings:

```
app.routes = {
        ...
        '/user/{name}/': {
                type: 'injector',
                locals: {
                        protocol: new org.restlet.routing.Template('{p}'),
                        deployment: 'production'
                },
                next: '@user'
        }
}
```

(The above example is not that useful: you can just as easily access the protocol using conversation.request.protocol.name.)

HTTP Authentication

You can implement simple HTTP authentication using the "basicHttpAuthenticator" route type (page 24):

```
app.routes = {
        ...
        '/*': {
                type: 'basicHttpAuthenticator',
                realm: 'Authorized users only!',
                credentials: {
                        moderator: 'moderatorpassword',
                        admin: 'adminpassword'
                },
                next: [
                        'manual',
                        'templates'
                ]
        }
}
```

Note that the implementation relies on basic authentication (BA), which is unencrypted. It is thus strongly recommended that you use it only with HTTPS (page 120).

Cross-Origin Resource Sharing (CORS)

You can add Cross-Origin Resource Sharing (CORS) headers using the "cors" route type (page 24):

```
app.routes = {
        ...
        '/*': {
                type: 'cors',
                allowOrigin: '*',
                allowMethods: ['GET', 'POST'],
                allowHeaders: ['Content-Type', 'Last-Modified', 'Expires'],
                maxAge: 'farFuture',
                next: [
                        'manual',
                        'templates'
                ]
        }
}
```

Note that "magAge" is in seconds, and must be greater than zero. You can specify it as as either a number or a string (page 71), or use "farFuture" as a shortcut for 10 years.

String Interpolation

Template variables, delimited by curly brackets, can be used to interpolate strings for three use cases:

- Captured URI targets, via the "capture" route type (page 23)
- Redirection URI targets, via the "redirect" route type (page 23)
- Cache key templates (page 63)

Prudence supports many built-in interpolation variables, extracted from the conversation attributes and summarized below. See also the related Restlet API documentation.

Note that for cache key templates, it's possible to create your own interpolation variables using plugins (page 64).

Request URIs

These variables are composed of a prefix and a suffix. The prefix specifies which URI you are referring to, while the suffix specifies the part of that URI. For example, the prefix "{r-}" can be combined with the suffix "{-i}" for "{ri}", to specify the complete request URI.

Prefixes

- {r-}: actual URI (reference)
- {h-}: virtual host URI
- {o-}: the application's root URI on the current virtual host
- {f-}: the referring URI (sent by some clients: usually means that the client clicked a hyperlink or was redirected here from elsewhere)

Suffixes

- {-i}: the complete URI (identifier)
- {-h}: the host identifier (protocol + authority)
- {-a}: the authority (for URLs, this is the host or IP address)
- {-p}: the path (everything after the authority)
- {-w}: the wildcard (remaining part of the path after the base URI, *not including* the query)
- {-r}: the remaining part of the path after the base URI, *including* the query
- {-e}: a relative path from the URI to the application's base URI (note that this is a constructed value, not merely a string extracted from the URI)
- {-q}: the query (everything after the "?")
- {-f}: the fragment (the tag after the "#"; note that web browsers handle fragments internally and *never* send them to the server, however fragments may exist in URIs sent *from* the server: see the "{R-}" variable mentioned below)

Base URIs

Every URI also has a "base" version of it: in the case of wildcard URI templates, it is the URI before the wildcard begins. Otherwise it is usually the application's root URI on the virtual host. It is used in the "{-r}" and "{-e}" suffixes above.

To refer to the base URI directly, use the special "{-b-}" infix, to which you would still need to add one of the above suffixes. For example, "{rbi}" refers to the complete base URI of the actual URI.

Relative URIs

For cache key templates only, you can also use the "{cb}" variable. It is equivalent to calling the conversation.base API.

Interpolating the Wildcard

This is a common use case for redirection, so even though it's included in the documentation above, it's worth emphasizing. According to the rules, the "*" would be the "{rw}" variable. For example:

```
app.routes = {
        ...
        '/assets/*': '/files/media/{rw}'
}
```

The above would capture a URI such as "/assets/images/logo.png" to "/files/media/images/logo.png".

The wildcard can also be accessed via API (page 52).

Request Attributes

- {p}: the protocol ("http," "https," "ftp," etc.)
- {m}: the method (in HTTP, it would be "GET," "POST," "PUT," "DELETE," etc.)
- {d}: date (in the RFC1123 format used by HTTP)

Client Attributes

- {cia}: client IP address
- {ciua}: client upstream IP address (if the request reached us through an upstream load balancer)
- {cig}: client agent name (for example, and identifier for the browser)

Payload Attributes

All these refer to the payload ("entity") sent by the client.

- {es}: entity size (in bytes)
- {emt}: entity media (MIME) type
- {ecs}: entity character set
- {el}: entity language
- {ee}: entity encoding
- {et}: entity tag (HTTP ETag)
- {eed}: entity expiration date
- {emd}: entity modification date

Negotiated Attributes

These are the result of content negotiation, and are used specifically for cache key templates.

- {nmt}: negotiated media (MIME) type
- {nl}: negotiated language
- {ne}: negotiated encoding

Implementation Attributes

These are used specifically for cache key templates.

- {dn}: document name
- {an}: application name

Response Attributes

These attributes are not normally used in Prudence—they are not used in routing nor in cache key templates—but are mentioned here for completion. They are all in uppercase to differentiate them from the request variables:

- {S}: the HTTP status code
- {SIA}: server IP address
- {SIP}: server port number
- {SIG}: server agent name
- {R-}: the redirection URI (see "Request URIs" above for a list of suffixes, which must also be in uppercase)

Additionally, all the entity attributes can be used in uppercase to correspond to the response entity. For example, "{ES}" for the response entity size, "{EMT}" for the response media type, etc.

conversation.locals

As we've seen in the app.routes guide (page 21), URI templates delimited by curly brackets can be used to parse incoming request URIs and extract the values into conversation.locals (page 83). For example, a "/person/{id}/" URI template will match the "/person/linus/" URI and extract "linus" into the "id" conversation.local.

But you can also do the opposite: interpolate the values that were extracted from the matched URI template. An example of redirection (page 55) that both extracts and interpolates:

```
app.routes = {
        ...
        "/person/{id}/": ">http://newsite.org/profile/?id={id}"
}
```

The Internal URI-space

Who says that you need HTTP, or any kind of networking, for REST? The principles are themselves applicable and suitable to in-memory communication, and represent an attractive architectural paradigm for APIs: attractive especially because it can work *both* locally and over the wire. This useful feature gives you considerable deployment flexibility: you can easily export a whole API layer to another running instance in the cluster (page 137). For an example of this, see the MVC chapter (page 136).

The RIAP Pseudo-Protocol

In Prudence, internal REST is straightforwardly supported by specifying the "RIAP" pseudo-protocol in URIs. RIAP stands for "Restlet Internal Access Protocol". There are two common formats for RIAP URIs:

- "riap://application/*": This will route to the *current* application.
- "riap://component/{application}/*": This will use the component's internal router, to which applications attach by default using their subdirectory name, allowing you to send requests to any application. You change the default name by configuring the "internal" host in app.hosts (page 31).

In both cases the wildcard will exactly match the URI templates you've mapped in app.routes (page 21).

Internal Requests

You can use RIAP URIs everywhere a full URI is used in Prudence, for example, when using the Prudence.Resources.request and document.external APIs. See the section on external requests (page 60) for examples.

However, the APIs can also implicitly create these RIAP URIs for you. For example:

```
document.require('/prudence/resources/')
var weather = Prudence.Resources.request({
        uri: '/weather/',
        mediaType: 'application/json'
})
```

Prudence.Resources.request will automatically assume that URIs beginning with "/" are internal, and thus set the "internal" param to true. The URI used would in fact be an RIAP URI: "riap://application/weather/".

To access a different application:

```
var weather = Prudence.Resources.request({
        uri: '/weather/',
        internal: 'weatherapp'
        mediaType: 'application/json'
})
```

The actual URI would be "riap://component/weatherapp/weather/".

Low Level For non-JavaScript you can use the lower-level document.internal and document.internalOther APIs:

```
document.require('/sincerity/json')
var resource = document.internal('/weather/', 'application/json')
result = resource.get()
if (null !== result) {
        weather = Sincerity.JSON.from(result.text)
}
```

To access a different application:

```
resource = document.internalOther('weatherapp', '/weather/', 'application/json')
```

Private URI-space

Internal requests bypass the hiding mechanism in routing.js (page 22). Thus, if you want some URIs to only be usable internally, simply hide them from the public URI-space.

Avoiding Serialization for Internal Requests

The web data chapter covers the creation of response payloads (page 57). There is, however, a possible optimization for internal requests. If your textual and binary representations are serialized versions of data structures, it would be unnecessary and wasteful to go through serialization/deserialization for an internal request. A simple optimization would be to pass the data "as is," which is actually very similar to a return value for a mundane function call.

This works using the special "application/internal" MIME type. If selected—whether negotiated for or explicitly overridden (page 59)—then return values are indeed sent "as is." For example:

```
function handleInit(conversation) {
	conversation.addMediaTypeByName('application/json')
	if (conversation.internal) {
		conversation.addMediaTypeByName('application/internal')
	}
}

function handleGet(conversation) {
	var data = ...
	return conversation.mediaTypeName == 'application/internal' ?
		data : Sincerity.JSON.to(data)
}
```

Note that we've added support for "application/internal" only to internal requests, verified using the conversation.internal API. The reason is that we don't want to allow "application/internal" representations over HTTP. (Actually, that *could* work if the object is itself JVM-serializable, but that's a more advanced use case we won't deal with here.)

Configuration

Your Prudence container's "/component/" directory has various subdirectories in which you can configure it.

Prudence uses "configuration-by-script" almost everywhere: configuration files are true JavaScript source code, meaning that you can do pretty much anything you need during the bootstrap process, allowing for dynamic configurations that adjust to their deployed environments.

> Prudence, as of version 2.0, does not support live re-configuration. You must restart Prudence in order for changed settings to take hold. The one exception is the system crontab (page 108): changes there are picked up on-the-fly once per minute.

/configuration/logging/

Used by the Sincerity logging plugin. Configure system-wide logging here, using Apache log4j.

/configuration/sincerity/

Used by Sincerity to manage installation of packages in your Prudence container. You usually won't be editing this files directly, instead using "sincerity" commands to manipulate it. However, take special note of artifacts.conf if you are committing your container to a VCS.

/configuration/hazelcast/

Configure Hazelcast here, if you are running in a cluster (page 137). Note that the configuration is actually loaded by the distributed service (page 125).

You have two options for configuration:

- By script, at "/configuration/hazelcast/". This is the default method, and is recommended.
- By standard XML file, at "/configuration/hazelcast.conf". This method is provided for compatibility, and it's generally preferable to use the by-script configuration. An example file is provided for you at "/configuration/hazelcast.alt.conf". If you rename this file to "hazelcast.conf", then it will be used *instead* of the by-script configuration.

For your convenience, the entire com.hazelcast.config package is already imported for configuration-by-script. See the Hazelcast configuration guide for a general overview.

The default configuration creates a single Hazelcast instance belonging to an "application" cluster, and the default configuration of applications points them to use this instance, though each application gets its own Hazelcast map to handles it application.distributedGlobals API (page 83). However, you can configure applications to use any Hazelcast instance, as well as change which shared objects to use, in their settings.js (page 74). This allows each application to belong to a different Hazelcast cluster.

Likewise, if you are using Hazelcast as a cache backend (page 123), it will default to use the Hazelcast "application" instance. However, you can set the cache backend to use its own Hazelcast instance in its constructor.

There are good reasons why you might want a more complex configuration: for example, if you're doing task farming (page 139), you might want the "task" nodes to be separate from the "application" nodes. A commented-out suggested configuration for this is included—see the clusters chapter for a detailed explanation.

/component/

The Prudence component is bootstrapped here. If you wish to understand exactly how it works, take a look at "/component/default.js".

sharedGlobals

In all the "/component/" configuration files, you have access to a global "sharedGlobals" JavaScript dict. Values you set here will become application.sharedGlobals (page 82) once the component is started. This dict works similarly to app.globals in settings.js (page 75); it is likewise "flattened."

For an example, here we set a database connection pool as a shared global, using a custom service (page 123) defined in "/component/services/database/default.js":

```
sharedGlobals.database = sharedGlobals.database || {}
sharedGlobals.database.pool = createPool()
```

Initializers

In all the "/component/" configuration files, you have access to a global "initializers" JavaScript array. Any function you add to this array will be executed *after* the component is started, in the order in which they were added. Here's a trivial usage example:

```
initializers.push(function() {
        println('This is my trivial initializer!')
})
```

/component/hosts/

Restlet has excellent virtual host support. There is a many-to-many relationship routing between servers, hosts and applications, allowing you considerable flexibility in binding your URI-spaces. For example, you can easily have a single Prudence container (running in a single JVM instance) managing several sites at once, with several applications, on several domains, on several servers.

Define your virtual hosts as ".js" files under "/component/hosts/". A minimal host definition would like this:

```
var host = new org.restlet.routing.VirtualHost(component.context)
host.name = 'privatehost'
component.hosts.add(host)
```

The "host.name" param exactly matches the string used in app.hosts (page 31) per each application.

A virtual host can route according to domain name, and incoming server IP address and port assignment:

```
host.resourceScheme = [string]
host.resourceDomain = [string]
host.resourcePort = [string]
host.serverAddress = [string]
host.serverPort = [string]
host.hostScheme = [string]
host.hostDomain = [string]
host.hostPort = [string]
```

An example of a virtual host for a specific domain name:

```
var host = new org.restlet.routing.VirtualHost(component.context)
host.name = 'other'
host.resourceDomain = 'otherdomain.org'
component.hosts.add(host)
```

If you do this, you will likely want to set "resourceDomain" for the default host to a specific domain, too.
Some notes:

- "*" wildcards are supported for all of these properties. If you do not explicitly set a property value, it will default to "*".
- "resourceScheme", "resourceDomain" and "resourcePort" refer to the actual incoming URI. Thus "resourcePort" would be meaningful *only if* URIs explicitly include a port number, a rather rare situation. To match a host to a specific *server* you would likely want to use "serverPort".
- "hostScheme", "hostDomain" and "hostPort" are for matching the "Host" HTTP header used by load balancers and other proxies (page 145).

The Hungry Host

The "default" host that comes with the Prudence skeleton doesn't configure any routing limitations, meaning that *all* incoming requests are routed (equivalent to setting all properties at "*"). We call such a host "hungry," because it will "eat" any request coming its way, denying any other virtual hosts the opportunity to route them. Thus, if you want to use more than one host, you have two options:

1. Don't use a hungry host: delete the "default.js" file or comment out the "component.hosts.add" call in it.
2. Make sure that the hungry host is the last one, so that it acts as a fallback when other hosts don't match incoming requests. Because host files are executed in alphabetical order, a good way to ensure that the hungry host is last is to number your host filenames. For example:

- "1-public.js"
- "2-private.js"
- "3-default.js" (this is the hungry host)

Deploying Multiple Sites

Using the virtual hosts and application model, Prudence can let you manage several sites using a single Prudence installation (a "container"). But is this always a good idea?

Advantages of Using a Single Container

1. Possibly simpler deployment: you are using a single base directory for the entire project, which might be easier for you. Because all configuration is done by JavaScript inside the container, it is very flexible.
2. Less memory use than running multiple JVMs.
3. Shared memory: you can use application.sharedGlobals (page 82) to share state between applications.

Advantages of Using Multiple Containers

1. Possibly simpler deployment: several base directories can mean separate code/distribution repositories, which might be easier for you. You'll configure routing between them at your load balancer (page 142).
2. Robustness: crashes/deadlocks/memory leaks in one VM won't affect others. With this in mind, it may even be worth having each *single application* running in its own JVM/container.
3. Run-time flexibility: you can restart the JVM for one container without affecting others that are running.

There is no performance advantage in either scenario. Everything in Prudence is designed around high-concurrency and threading, and generally threads are managed by the OS globally.

Well, there are caveats to that statement: Linux can group threads per running process for purposes of prioritization, but this is mostly used for desktop applications. The feature could possibly be useful when running several Prudence containers, if you want to guarantee high thread priority to one of the containers over the others. This kind of tweaking would only effect *very* high concurrency and highly CPU-bound deployments.

/component/servers/

Define your servers as ".js" files under "/component/servers/". At the minimum, you must specify a protocol and a port. Here's an example definition for an HTTP server, "http.js":

```
var server = new Server(Protocol.HTTP, 8080)
server.name = 'myserver'
component.servers.add(server)
```

The server name is optional, and used for debugging.

An important configuration is to bind a server to a specific IP address, in case your machine has more than one IP address:

```
server.address = [string]
```

There are many configuration parameters for Jetty, the HTTP engine, which you can set as parameters in the server's context. Here we'll increase the size of the thread pool and lower the idle timeout:

```
server.context.parameters.set('threadPool.minThreads', '50')
server.context.parameters.set('threadPool.maxThreads', '300')
server.context.parameters.set('connector.idleTimeout', '10000')
```

For a complete list of available configuration parameters, see JettyHttpServerHelper. *Make sure to check the documentation for all the parent classes*, because they are inherited. Note that parameters are always set as strings, even if they are interpreted as other types.

Load Balancing Are your servers running behind a load balancer? See the explanation here (page 146) as to why you would want to add this:

```
server.context.parameters.set('useForwardedForHeader', 'true')
```

Secure Servers (HTTPS)

If you are using a load balancer (page 142), it may make sense to handle secure connections there. But Prudence can also handle secure (HTTPS) connections itself. Here's an example configuration for "/component/servers/https.js":

```
var server = new Server(Protocol.HTTPS, 8082)
server.name = 'secure'
component.servers.add(server)

// Configure it to use our security keys
server.context.parameters.set('keystorePath', '/path/prudence.jks')
server.context.parameters.set('keystorePassword', 'mykeystorepassword')
//server.context.parameters.set('keyPassword', 'mykeypassword')
```

See DefaultSslContextFactory for all security configuration parameters.

Security Keys The above configuration assumes the you have a Java KeyStore (JKS) file at "/path/prudence.jks" containing your security key. You can create a key using the "keytool" utility that is bundled with most JDKs. For example:

```
keytool -keystore /path/prudence.jks -alias mykey -genkey -keyalg RSA
```

When creating the keystore, you will be asked provide a password for it, and you may optionally provide a password for your key, too, in which case you need to comment out the relevant line in the example above. (The key alias and key password would be transferred together with the key if you move it to a different keystore.)

Note that if you want to make client requests to a server that uses such a self-created key, you will need your client to recognize that key. If this is done from the JVM, this means setting the "javax.net.ssl.trustStore" JVM property. For example, if you're using Prudence's request API (page 60), you will need to start Prudence like so:

```
JVM_SWITCHES=-Djavax.net.ssl.trustStore=/path/prudence.jks sincerity start prudence
```

Such self-created keys are useful for controlled intranet environments, in which you can provide clients with the public key, but for Internet applications you will likely want a key created by one of the "certificate authorities" trusted by most web browsers. Some of these certificate authorities may conveniently let you download a key in JKS format. Otherwise, if they support PKCS12 format, you can use keytool (only JVM version 6 and later) to convert PKCS12 to JKS. For example:

```
keytool -importkeystore -srcstoretype PKCS12 -srckeystore /path/prudence.pkcs12 -
    destkeystore /path/prudence.jks
```

If your certificate authority won't even let you download PKCS12 file, you can create one from your ".key" and ".crt" (or ".pem") files using OpenSSL:

```
openssl pkcs12 -inkey /path/mykey.key -in /path/mykey.crt -export -out /path/
    prudence.pkcs12
```

(Note that in this case you *must* give your new PKCS12 a non-empty password, or else keytool will fail with an unhelpful error message.)

HTTP/2 Jetty adds full HTTP/2 support to your secure servers, bringing an improved user experience and a lighter load on the backend. See the Sincerity documentation for details on how to enable it.

API It's sometimes necessary to support HTTPS specially in your implementation. One useful strategy is to create separate applications for HTTP and HTTPS, and then attach them to different virtual hosts (page 118), one for each protocol (the "resourceScheme" parameter). However, if the application behaves mostly the same for HTTP and HTTPS, but differs only in a few specific resources, it may be useful to check for HTTPS programmatically, via the conversation.reference.schemeProtocol API. For example:

```
if (conversation.reference.scheme == 'https') {
        ...
}
```

Other Server Engines

Prudence, by default, uses Jetty 9.3 as its server engine. Jetty is mature, performant and eminently scalable, and we highly recommend it for production environments.

However, it's possible to replace Jetty with a different engine should you require. To do this, you must remove the Jetty Restlet 9 connector from your container, and install a different connector instead, as well as its dependencies.

To make sure Jetty 9 is excluded from the next installation, you can use the following Sincerity command:

```
sincerity exclude org.restlet.jse org.restlet.ext.restlet.jetty9 :
        exclude org.eclipse.jetty jetty-security :
        exclude org.eclipse.jetty jetty-client
```

As of Restlet 2.3, three alternatives are available:

Jetty 9.2 This is the recommended alternative to Jetty 9.3 if you cannot use JVM 8 or above, and are limited to JVM 7. To install it:

```
sincerity add org.restlet.jse org.restlet.ext.jetty : install
```

The configuration parameters are documented here.

Simple Simple Framework, which also works on JVM 6, is a lighter alternative to Jetty. Its documentation makes some controversial claims about its improved scalability in comparison to Jetty, but we encourage you to verify them for yourself. To install it:

```
sincerity add org.restlet.jse org.restlet.ext.simple : install
```

The configuration parameters are documented here.

Restlet's Internal Server Engine Will be used if no other connector is installed. While the internal engine may be adequate for testing, we found that it suffers from stability issues, and do not recommend it for production environments.

/component/clients/

Client connectors have two main use cases:

- External requests (page 60). Most often you will use "http:" and "https:" connectors, but you might also need "file:", "ftp:", WebDAV extensions and/or others.
- Static resources (page 46) internally *require* a "file:" client connector.

To add a client, add a ".js" file to "/component/clients/". For example, here's a minimal configuration for an HTTP client, "http.js":

```
importClass(org.restlet.data.Protocol)
var client = component.clients.add(Protocol.HTTP)
```

Clients are configured by setting parameters in their context:

```
client.context.parameters.set('socketTimeout', '10000')
```

For a complete list of available configuration parameters, see HttpClientHelper. *Make sure to check the documentation for all the parent classes*, because they are inherited. Note that parameters are always set as strings, even if they are interpreted as other types.

(That link was for Jetty 9; use this link if you are using Apache HttpClient on JVM 6.)

Other Client Engines

Prudence, by default, uses Jetty 9 as its client engine. Jetty is mature, performant and eminently scalable, and we highly recommend it for production environments.

However, it's possible to replace Jetty with a different engine should you require. To do this, you must remove the Jetty Restlet 9 connector from your container, and install a different connector instead, as well as its dependencies.

To make sure Jetty 9 is excluded from the next installation, you can use the following Sincerity command:

```
sincerity exclude org.restlet.jse restlet-jetty9 :
        exclude org.eclipse.jetty jetty-security :
        exclude org.eclipse.jetty jetty-client
```

As of Restlet 2.2, three alternatives are available:

Apache HttpClient Apache HttpClient is the recommended alternative to Jetty 9 if you cannot use JVM 7 or above, and are limited to JVM 6. To install it:

```
sincerity add org.restlet.jse restlet-httpclient : install
```

The configuration parameters are documented here.

URLConnection This engine uses the java.net.URLConnection class included in the JVM. It does not scale well, however it does the job, and even supports FTP connections. To install it:

```
sincerity add org.restlet.jse restlet-net : install
```

The configuration parameters are documented here for HTTP, and here for FTP.

Restlet's Internal Client Engine Will be used if no other connector is installed. While the internal engine may be adequate for testing, we found that it suffers from stability issues, and do not recommend it for production environments.

/component/services/

Services are run *after* the component is configured but *before* it is started. Several services are required to support various Prudence features, but you may freely add your own subdirectories here, to support your own features and subsystems.

Remember that your custom services have access to "sharedGlobals" (page 118) and "initializers" (page 118).

/component/services/log

Prudence supports NCSA-style logging of all incoming client requests, to all servers. By default, we just configure the logger name here (the default is "web"):

```
component.logService.loggerName = 'web'
```

You may also further configure the log format using the string interpolation variables (page 113):

```
component.logService.responseLogFormat = '{d}\t{cia}\t{ri}\t{S}'
```

See the Restlet documentation for more information.

To learn how configure this logger and connect it to an appender, refer to the documentation for Sincerity's logging plugin. By default, Prudence will use a rolling file appender to "/logs/web.log".

/component/services/prudence/status

Here you may configure system-wide custom error pages. Though applications may define their own custom error pages using app.errors (page 30), you can set them up here, too. Note that app.errors takes precedence over definitions here: you may thus treat this feature as a fallback option for when applications do not handle errors themselves.

An example definition:

```
statusService.capture(404, 'myapp', '/404/', component.context)
```

The capture API uses an application's internal name (page 31), allowing you to implement the error page in any installed application. Note that this API does not let you capture-and-hide the target URI, e.g. "/404/!". If you wish to hide it, you must do so in the application's app.routes.

/component/services/prudence/caching

Configure the caching (page 61) backends here.

In "/services/prudence/caching/default.js", a ChainCache instance is set up as the main cache implementation. This allows you to create a tiered cache. Each tier should be added in order: the first tier added to the chain is the first one from which Prudence will fetch, so it usually should be the fastest backend. When storing entries in the cache, *all* tiers will be invoked.

Configure your tiered backends under "/services/prudence/caching/backends/", making sure that their filenames are in alphabetical order. By default, Prudence calls these "backend.1.js", "backend.2.js", etc., though you may your use your own alphabetic scheme.

In-Process Memory Cache By default, Prudence sets up an InProcessMemoryCache as the first tier. Its default max size is 1MB, however it's recommended to increase this size according to your machine's available RAM. Note that if you're limiting the JVM's RAM usage in some way (for example, if you're using the Sincerity service plugin), then you want to make sure that the JVM has enough room for your cache as well as its normal operational requirements.

Though the in-process memory cache offers the best possible performance, it's of course limited in size. For *very* large web sites, it might be too small to be effective: if cache entries keep being discarded to make room for others, it will not be helping you much as a 1st tier backend. Make sure to monitor your usage carefully to see how often cache entries are discarded. Otherwise, good alternatives for the 1st tier would be Hazelcast or memcached.

Other Cache Backends The following cache backends are all supported in Prudence:

- Hazelcast: If you're already running in a cluster (page 137), enabling a Hazelcast-based cache backend is a great idea, because you've already deployed it. This cache backend is in essence similar to the in-process memory cache, except that you will be pooling together the RAM from all running JVMs in the cluster. Note that it's possible to add lazy persistence plugins to Hazelcast, to make sure your data is stored on disk. Note that it doesn't make much sense to use the in-process memory cache together with Hazelcast: choose one or the other. See the API documentation.
- memcached: For *very* large web sites, even your pooled RAM in the cluster may not be enough. In that case, you may consider creating a separate memcached-based cluster, entirely devoted to caching. An advantage of memached is that it's very standard and widely supported, so you may also be able to use your cache cluster for other systems. Note that though memcached is not persistent by design, there exist compatible alternatives, such as Tarantool, that allow lazy persistence of your cached data to disk. See the API documentation.
- MongoDB: If you're using MongoDB to store your data, it makes perfect sense to also use it as a cache, as it performs very well and provides you with instant persistence, as well as support for truly enormous caches. You can also use a capped collection for even better performance, at the expense of limiting your cache size. Also, Prudence can store its cache entries as structured MongoDB documents, making it very easy to debug or otherwise collect statistics directly from the cache collection. See the API documentation.
- SQL: This is a great choice if you're using a relational (SQL) database to store your data. Advocates of "NoSQL" like to claim that relational databases are "slow," but that's nonsense: Prudence doesn't use transactions for its implementation, and performance should be excellent, no worse than "NoSQL," especially with some careful tuning. It's definitely fine as a 2nd-tier cache. Practically any database can be supported via JDBC, though we also have a special implementation for the H2 database, making it easy to test locally, because H2 can run inside Prudence's JVM. See the API documentation (also for H2).

Easy Installation We've made it easy to install the dependencies for all the supported cache backends via Sincerity packages and shortcuts. The packages will also install an example configuration, as a 2nd-level tier (after the default in-process memory cache), in the file "/services/prudence/caching/backends/backend.2.js". Here's a table of the shortcuts, as well as the full package identifiers:

Backend	Shortcut	Identifier
Hazelcast	prudence.cache.hazelcast	com.threecrickets.prudence prudence-cache-hazelcast
memached	prudence.cache.memached	com.threecrickets.prudence prudence-cache-memcached
MongoDB	prudence.cache.mongodb	com.threecrickets.prudence prudence-cache-mongodb
H2	prudence.cache.h2	com.threecrickets.prudence prudence-cache-h2

For example, to install the H2 cache backend in the second tier:

```
sincerity add prudence.cache.h2 : install
```

The default configuration will put the H2 files under "/cache/prudence/cache/".

/component/services/prudence/startup

This service provides you with access to a global "startupTasks" JavaScript array. Any object implementing Callable or Runnable that you add to this array will be executed *after* the component configured but *before* is started in a multi-threaded task pool. You can configure the thread pool here.

This feature is used by the defrost/preheat feature (page 33).

/component/services/prudence/executor

This service configures the thread pool used for background task execution (page 102). You may change the size of the thread pool here, or otherwise install specialized implementations. By default Prudence uses (number of CPU cores * 2 + 1) for the pool size, a formula which offers good performance under high loads for common network-bound scenarios, though you may want to decrease this size for heavy CPU-bound workloads.

See the Executors API for a few other built-in options. Note, however, that your implementation must support the ScheduledExecutorService interface if you wish to support task scheduling.

The executor can be accessed via the application.executor API.

/component/services/prudence/scheduler

The scheduler is used to handle the crontab feature (page 106). By default, Prudence here sets up a scheduler for the component, accessible via the application.scheduler API, and also installs the system-wide crontab (page 108). You may edit this file to add other specialized crontabs, or otherwise set up scheduled tasks.

See the cron4j Scheduler documentation for more options.

/component/services/prudence/distributed

This service's job is to load the Hazelcast configuration (page 117).

/component/services/prudence/singleton

Prudence assumes a single Restlet Component instance. If for some reason you have a more complex setup, you can configure Prudence's initialization here.

/component/services/prudence/version

Provides access to Prudence, Restlet and Jetty versions, and prints out the welcoming message. There's not much to configure here, but feel free to examine the code!

/component/templates/

The "prudence create" Sincerity command copies and adapts the application template under "/component/templates/default/". You can create your own templates in that directory, and use their name as a second argument to the "prudence create" command, e.g.: "sincerity prudence create cms mytemplate". Likely, you'd want to copy the default template and modify it.

The mechanism interpolates the "${APPLICATION}" string in any of these files to be the application name argument you supplied as the first argument to "prudence create".

Model-View-Controller (MVC)

The source code for the examples in this chapter is available for download.

Background

The model-view-controller (MVC) family of architectural patterns has had great influence over user-interface programming and even design. At its core is the idea that the "model" (the data) and the "view" (the user interface) should be decoupled and isolated. This essentially good idea allows each layer to be optimized and tested on its own. It also allows the secondary benefit of easier refactoring in the future, in case one of the layers needs to be replaced with a different technology, a not uncommon requirement.

Forms, Forms, Forms The problem is that you need an intermediary: a middle layer. For this reason, "classic" MVC, based around a thick controller layer, isn't as popular as it used to be: the controller does *as much as possible* in order to automate the development of user interfaces for very large applications that require constant maintenance and tweaking. This kind of MVC thus handles form validation, binding of form fields to database columns, and even form flows.

Forms, forms, as far as the eye can see. MVC was, and still is, the domain of "enterprise" user interfaces. It's telling that the manipulation of the data model is called "business logic": the use case for MVC is big business for big businesses. At its best, MVC makes hundreds of forms easier to maintain in the long run. At its worst, programmers drown in an ocean of controller configuration files, fighting against opaque layers of APIs that can only do what they were programmed to do, but not necessarily what is needed for a sensible UI.

Outside of the big business world, UI implementations use more flexible derivatives of MVC, such as Model-View-Presenter (MVP). The "presenter" is not an opaque layer, but rather is implemented directly by the programmer in code, often by inheriting classes that provide the basic functionality while allowing for customization. Depending on the variation you're using, the "business logic" might even be in the presenter rather than the model. Still, implementing MVP, like MVC, often comes from the same anxiety about mixing model and view.

MVC and the Web These two do not seem an obvious match. The web is RESTful, such that the user interface (the "view") is no different from data (the "model"): both are RESTful resources, implemented similarly. In other words, in REST *the model is the view.*

Well, that's only really and entirely true for the "classic" web. Using JavaScript and other in-browser plugins, we get a "rich" web that acts no differently from desktop applications. The backend remains RESTful, essentially the "model," with controllers/presenters as well as views implemented entirely client-side. You can do full-blown, conventional MVC with the "rich" web.

MVC, however, has found inroads into the *classic* web: there exist many frameworks that treat web pages as a pure "view," an approach they go so far as to enforce by allowing you to embed only limited templating code into your HTML. Some of these frameworks even allow you to configure the form flow, which they then use to generate an opaque URI-space for you, and can even sabotage the browser "back" button to enforce that flow. (MVC automation at its finest! Or worst. . .)

The impetus for these brutally extreme measures is similar to the one with which we started: a desire to decouple the user interface from everything else. HTML is the realm of web designers, not programmers, and mixing the work of both professions into a single file presents project management challenges. However, there's a productive distance between cleaning up HTML pages and full-blown MVC, which unfortunately not enough frameworks explore. And actually, not everything called "MVC" really is MVC.

So what are we left with in Prudence? As you'll see, Prudence supports a straightforward MVP architecture while still adhering to RESTful principles. Read on, and consider if it would benefit your project to use it. You do *not* have to. Our recommendation is to use what works best for you and your development team.

Tutorial

Models

You should implement this layer as is appropriate to the database technology and schema you are using. Object-oriented architecture are common, but of course not necessary. The model layer as a whole should live in your "/libraries/" subdirectory. For this example, let's put it under "/libraries/models/".

Do you want the "business logic" to live in the model layer? If so, your classes should be of a somewhat higher level of abstraction above the actual data structure. If you prefer the models to more directly represent the data, then you have the option of putting the "business logic" in your presenters instead.

For our example, let's implement a simple in-memory model, as "/libraries/models/user.js":

```
var Models = Models || {}

/**
 * Retrieve a person model from the database.
 */
Models.getPerson = function(name) {
	var person = new Models.Person()
	person.setUsername(name)
	return person
}

Models.Person = function() {
	this.getUsername = function() {
		return this.username
	}

	this.setUsername = function(username) {
		this.username = username
	}

	this.getComments = function() {
		return this.comments
	}

	this.comments = new Models.Messages()
}

Models.Messages = function() {
        this.get = function() {
                return this.messages
        }

        this.add = function(message) {
                this.messages.append(message)
        }

        this.messages = ['This is a test.', 'This is also a test.']
}
```

Views

In Prudence, these are hidden template resources (page 39). For this example, let's put them under "/libraries/includes/views/".

If you prefer to use templating languages for your views, Velocity and Succinct are supported (page 43). Your designers may also find it useful to use the supported HTML markup languages (page 43). Even if you prefer templating, you can still "drop down" to dynamic languages, such as JavaScript (server-side), when useful: Prudence allows you to easily mix and match scriplets in different languages. If you do so, take special note of the nifty in-flow tag.

> There are some who shudder at the thought of mixing dynamic languages and HTML. This likely comes from bad experience with poorly-designed PHP/JSP/ASP applications, where everything gets mixed together into the "view" file. If you're afraid of losing control, then you can simply make yourself a rule that only templating languages are allowed in template resources. It's purely a matter of project management discipline. We recommend, however, relaxing some of that extremism: for example, you can make the rule that no "business logic" should appear together with HTML, while still allowing some flexibility for using server-side JavaScript, but *only* for UI-related work. Still unconvinced? We'll show you below how to use your favorite pure templating engine with Prudence (page 129).

The required data will be injected into the view by the presenter as an "object" POST payload, available via the conversation.entity API. We'll detail below how that happens. For now, here's our example view, "/libraries/includes/views/user/comments.html":

```
<html>
<%
var context = conversation.entity.object
var person = context.person
var comments = person.getComments().get()
%>
<body>
        <p>These are the comments for user: <%= person.getUsername() %></p>
        <table>
<% for (var c in comments) { %>
                <tr><td><%= comments[c] %></td></tr>
<% } %>
        </table>
        <p>You may add a comment here:</p>
        <form>
                <input name="comment" />
                <input type="submit" />
        </form>
</body>
</html>
```

Presenters

In Prudence, these are the resources that are actually exposed in the URI-space, while the views remain hidden. The presenter retrieves the appropriate view and presents it to the user.

You can use either manual or template resources as your presenters. However, manual resources (page 36) offer a bit more flexibility, so we will choose them for our example. Our example presenter is in "/resources/user/comments.m.js":

```
document.require(
        '/models/user/',
        '/prudence/resources/',
        '/sincerity/objects/')

function handleInit(conversation) {
        conversation.addMediaTypeByName('text/html')
}

function handleGet(conversation) {
        var name = conversation.locals.get('name')
        var person = Models.getPerson(name)
        return getView('user/comments', {person: person})
}

function handlePost(conversation) {
        var name = conversation.locals.get('name')
        var comment = conversation.form.get('comment')
        var person = Models.getPerson(name)
        person.getComments().add(comment)
        return getView('user/comments', {person: person})
}

function getView(view, context) {
        var page = Prudence.Resources.request({
                uri: '/views/' + view + '/',
                internal: true,
                method: 'post',
                mediaType: 'text/*',
                payload: {
                        type: 'object',
                        value: context
                }
        })
        return Sincerity.Objects.exists(page) ? page : 404
}
```

To keep the example succinct, we're only making use of a single view in this presenter, though it should be clear that you can use any appropriate logic here to retrieve any view using getView.

getView is where the MVC "magic" happens, but as you can see it's really nothing more than an internal request. We're specifically using two special features of internal requests:

- We can request URIs that are hidden: in this case, anything under "/libraries/includes/".
- We can transfer "POST" payloads directly using the "object" type: see (page 116).

We'll remind you also that internal requests are *fast*. They emphatically do not use HTTP, and "object"-type payloads are *not* serialized.

Here's our routing.js entry for the presenter:

```
app.routes = {
        ...
        '/user/{name}/comments/': '/user/comments/!'
}
```

Note the use of capture-and-hide (page 26).

Voila. Test your new MVC application by pointing your web browser to "/user/Linus/comments/" under your application's base URL.

Implications for Caching

You have two options for implementing caching (page 61):

- You can cache the presenter, by simply adding a caching.duration (page 62) directive in handleInit. However, this would mean that all views would be cached using the same parameters, which may not be flexible enough.
- You can cache the views. Your presenter logic will always be called for every request, but the views may be fetched from the cache. This will allow every view to use its own caching parameters. However, you should take care to remember that the request hitting the template resource is the *internal* one, *not* the external one, which actually hits the presenter. If there are attributes of the external request that you want to use for the cache key template, then you must transfer them manually.

View Templates

One size does not fit all. Almost every web framework comes with its own solution to templating, with its own idiosyncratic syntax and set of features, manifesting its own templating philosophy. As you've probably picked up, Prudence's philosophy is that the programmer knows best: scriptlets should be able do *anything*, and the programmer doesn't need to be "protected" from bad decisions via a dumbed-down, sandboxed templating domain language.

There are two common counter-arguments, which we don't think are very convincing.

The first is that the people designing the templates might not, in fact, know best: they might not be proficient programmers, but instead web designers who specialize in HTML/CSS coding and testing. They would be able to deal with a few inserted template codes, but not a full-blown programming language. The "real" programmers would be writing the controllers/presenters, and injecting values into the templates according to the web designers' needs. This argument carries less validity than it used to: proficient web designers these days need to know JavaScript, and if they can handle client-side JavaScript, they should be able to handle server-side JavaScript, too. Will they need to learn some new things? Yes, but learning a new templating language is no trivial task, either.

The second counter-argument is about discipline: even competent programmers might be tempted to make "shortcuts," and insert "business logic" into what should be purely a "view." This would short-circuit the MVC separation and create hard-to-manage "spaghetti" code. A restricted templating language could, then, enforce this discipline. This seems like a brutal solution: programmers get annoyed if their own platforms don't trust them, and in any case can circumvent these restrictions by writing plugins that would then do what they need. But the real issue is that discipline should be handled as a social issue of project management, not by tools.

In any case, we won't force our philosophy on you: Prudence has built in support for two templating engines (page 43), and it's easy to plug in a wide range of alternative templating engines into Prudence. If you're familiar and comfortable with a particular one, use it. We'll guide you in this section.

There are many templating engines you can use. The best performing and most minimal solutions are pure JVM libraries: StringTemplate, Thymeleaf, Snippetory and Chunk. However, popular ones use other languages, for example Jinja2 for Python. Below are examples per both types.

The technique we'll show for using both types is the same: writing a custom dispatcher (page 33), so make sure you understand dispatching before you continue to read.

StringTemplate Example

For this example, we chose StringTemplate: it's very minimal, and stringently espouses a philosophy entirely opposite to Prudence's: proof that Prudence is not forcing you into a paradigm! We'll of course use the Java/JVM port, though note that StringTemplate is also ported to other languages.

It's available on Maven Central, so you can install it in your container using Sincerity:

```
sincerity attach maven-central : add org.antlr ST4 : install
```

Otherwise, you can also download the binary Jar from the StringTemplate site and place it in your container's "/libraries/jars/" directory.

Here's our application's "/libraries/dispatchers/st.js":

```
document.require('/sincerity/objects/')

function handleInit(conversation) {
        conversation.addMediaTypeByName('text/html')
}

function handlePost(conversation) {
        if (!conversation.internal) {
                return 404
        }
        var id = String(conversation.locals.get('com.threecrickets.prudence.dispatcher.id'))
        if (id.endsWith('/')) {
                id = id.substring(0, id.length - 1)
        }
        var st = getDir().getInstanceOf(id)
        if (!Sincerity.Objects.exists(st)) {
                return 404
        }
        if (Sincerity.Objects.exists(conversation.entity)) {
                var context = conversation.entity.object
                if (Sincerity.Objects.exists(conversation.context)) {
                        for (var key in context) {
                                var value = context[key]
                                if (Sincerity.Objects.isArray(value)) {
                                        for (var v in value) {
                                                st.add(key, value[v])
                                        }
                                }
                                else {
                                        st.add(key, value)
                                }
                        }
                }
        }
        return st.render()
}

function getDir() {
        var dir = new org.stringtemplate.v4.STRawGroupDir(application.root + '/libraries/view
        dir.delimiterStartChar = '$'
        dir.delimiterStopChar = '$'
        return dir
}
```

The StringTemplate API is very straightforward and this code should be easy to follow. Notes:

- We've allowed only internal requests through: we want to hide this dispatcher from the public URI space because it should only accessed by our presenters.
- We're stripping trailing slashes from the ID because STRawGroupDir doesn't support them.
- We've switched to the older "$" delimiters because the default "<" and ">" delimiters are awkward to use with HTML.
- STRawGroupDir does not pick up template file changes on-the-fly, so we're recreating it per request. Because

it caches compiled templates, it would be more efficient to make it a global, but they you would have to find an alternative way for invalidating it for live application edits.

Now for our routing.js:

```
app.routes = {
          ...
          '/views/*': '@st:{rw}'
}

app.dispatchers = {
          ...
          st: {
                    dispatcher: '/dispatchers/st/'
          }
}
```

See how we've interpolated the wildcard (page 114) into "{rw}": this means that a URI such as "/views/hello/" would translate to the ID "hello".

Let's create our template, "/libraries/views/user/comments.st":

```
<html>
<body>
          <p>These are the comments for user: $username$</p>
          <table>
                    $comments:{c | <tr><td>$c$</td></tr>}$
          </table>
</body>
</html>
```

Note the use of an anonymous template (a lambda). As an alternative, we can also use named templates, which we can to group into reusable libraries. This is easy to do with group files. With that, here's an alternative definition of the above, saved as "/libraries/views/user.stg":

```
comments(username, comments) ::= <<
<html>
<body>
          <p>These are the comments for user: $username$</p>
          <table>
                    $comments:row()$
          </table>
</body>
</html>
>>

row(content) ::= <<
<tr><td>$content$</td></tr>
>>
```

Note that STRawGroupDir treats these ".stg" files as if they were a directory with multiple files when you look up an ID, so the URI to access "comments" would be the same in both cases: "/views/users/comments/".

Finally, our presenters would work the same as in the MVC tutorial (page 128). The only change would be to flatten out the contexts for StringTemplate to use:

```
var person = Models.getPerson(name)
...
return getView('user/comments', {
          username: person.getUsername(),
          comments: person.getComments().get()
})
```

That's it!

Implications for Caching You can set caching on your presenter, but unfortunately you can't set different caching parameters per view. StringTemplate's brutal rejection of any kind of programming logic in templates means that you can't "call" anything from within a template, not even to change a parameter.

In our next example, we'll be using a more flexible engine that allows for more integration with Prudence features.

Jinja2 Example

Jinja2 is an embeddable engine that mimics the well-known template syntax of the Django framework. We'll go over its basic integration into Prudence, and also show you how to write Jinja2 custom tags to easily take advantage of Prudence's caching mechanism (page 61).

First we need to install Python and Jinja2 in our container:

```
sincerity add python : install : easy_install Jinja2==2.6 simplejson
```

We're also installing the simplejson library, because Jython doesn't come with the JSON support we'll need (more on that later).

> Note that Jinja2 version 2.7 doesn't work in Jython (might be fixed for version 2.8), but version 2.6 does, so that's what we use here.

For our dispatcher, we'll do something a bit different from before: because we want to support caching of templates, we will want the actual template renderer as a template resource (page 39). The dispatcher, then, will simply delegate to that template resource. Another change is that we'll be writing it all in Python, so we can call the Jinja2 API. Let's start with "/libraries/dispatchers/jinja.py":

```python
import simplejson, urllib
from com.threecrickets.prudence.util import InternalRepresentation

def handle_init(conversation):
    conversation.addMediaTypeByName('text/html')

def handle_post(conversation):
    if not conversation.internal:
        return 404
    id = conversation.locals['com.threecrickets.prudence.dispatcher.id']
    if id[-1] == '/':
        id = id[0:-1]
    id += '.html'
    context = {}
    if conversation.entity:
        if conversation.entity.mediaType.name == 'application/internal':
            context = conversation.entity.object
        else:
            context = conversation.entity.text
            if context:
                context = simplejson.loads(context)
    payload = InternalRepresentation({
        'context': context,
        'uri': str(conversation.reference),
        'base_uri': str(conversation.reference.baseRef)})
    resource = document.internal('/jinja-template/' + urllib.quote(id, '') + '/', 'text/html'
    result = resource.post(payload)
    if not result:
        return 404
    return result.text
```

Notes:

- We've allowed only internal requests through: we want to hide this dispatcher from the public URI space because it should only be accessed by our presenters.

- Jinja2's FileSystemLoader requires the full filename, so we're stripping trailing slashes and adding ".html".
- We're forwarding a few useful attributes of the request: the original URI and the original base URI. We'll show you later how to use those for our custom tags.
- We're supporting "application/internal" payloads (page 116), but also JSON payloads. Avoiding serialization is fine for Python-to-Python calls, but if we call from another programming language—say, JavaScript—the native structures are incompatible. Thus, we're allowing the use of JSON as an interchange format. On the other side, when we send the payload to "/jinja-template/", it's fine to send a raw object, because it's Python-to-Python.

Our template resource is "/resources/jinja-template.t.html":

```
<%python
import urllib
from jinja2 import Environment, FileSystemLoader
from jinja2.exceptions import TemplateNotFound
from os import sep

id = urllib.unquote(conversation.locals['id'])
payload = conversation.entity.object
context = payload['context']

loader = application.globals['jinaj2.loader']
if not loader:
    loader = FileSystemLoader(application.root.path + sep + 'libraries' + sep + 'views')
    loader = application.getGlobal('jinja2.loader', loader)
env = Environment(loader=loader)

try:
    template = env.get_template(id)
    print template.render(context)
except TemplateNotFound:
    conversation.statusCode = 404
%>
```

The Jinja2 API is very straightforward and this code should be easy to follow. Note that we're caching the FileSystemLoader as an application.global: because it can pick up our changes on-the-fly and cache them, it's eminently reusable.

Now for our routing.js:

```
app.routes = {
        ...
        '/views/*': '@jinja:{rw}',
        '/jinja-template/{id}/': '/jinja-template/!'
}

app.dispatchers = {
        ...
        jinja: {
                dispatcher: '/dispatchers/jinja/'
        }
}
```

See how we've interpolated the wildcard (page 114) into "{rw}": this means that a URI such as "/views/hello/" would translate to the ID "hello/".

Let's create our template, "/libraries/views/user/comments.html":

```
<html>
<body>
```

```
        <p>These are the comments for user: {{username}}</p>
        <table>
{% for comment in comments %}
                <tr><td>{{comment}}</td></tr>
{% endfor %}
        </table>
</body>
</html>
```

Finally, we need to make two changes to our presenter (page 128). First, we need to isend JSON payloads (if we were writing it in Python, we could optimize by sending "object" payloads):

```
return getView('user/comments', {
        username: person.getUsername(),
        comments: person.getComments().get()
})

function getView(view, context) {
        var page = Prudence.Resources.request({
                uri: '/views/' + view + '/',
                internal: true,
                method: 'post',
                mediaType: 'text/*',
                payload: {
                        type: 'json',
                        value: context
                }
        })
        return Sincerity.Objects.exists(page) ? page : 404
}
```

And we have to "flatten" the model in order to make JSON-able:

```
var person = Models.getPerson(name)
...
return getView('user/comments', {
        username: person.getUsername(),
        comments: person.getComments().get()
})
```

That's it!

Custom Tags It's fairly easy to add custom tags to Jinja2. Let's add some to support Prudence caching, as well as other useful Prudence values. Here's "/libraries/jinja_extensions.py":

```
from jinja2 import nodes
from jinja2.ext import Extension
from org.restlet.data import Reference

class Prudence(Extension):
    # a set of names that trigger the extension
    tags = set(['current_uri', 'application_uri', 'to_base', 'cache_duration', 'cache_tags'])

    def __init__(self, environment):
        super(Prudence, self).__init__(environment)

        # add the defaults to the environment
        environment.extend(
            prudence_caching=None,
            prudence_uri=None,
            prudence_base_uri=None
        )

    def parse(self, parser):
        token = parser.stream.next()
        tag = token.value
        lineno = token.lineno

        if tag == 'current_uri':
            return _literal(self.environment.prudence_uri, lineno)

        elif tag == 'application_uri':
            return _literal(self.environment.prudence_base_uri, lineno)

        elif tag == 'to_base':
            base = Reference(self.environment.prudence_base_uri)
            reference = Reference(base, self.environment.prudence_uri)

            # reverse relative path to the base
            relative = base.getRelativeRef(reference).path

            return _literal(relative, lineno)

        elif tag == 'cache_duration':
            duration = parser.parse_expression().as_const()
            self.environment.prudence_caching.duration = duration

        elif tag == 'cache_tags':
            tags = [parser.parse_expression().as_const()]
            while parser.stream.skip_if('comma'):
                tags.append(parser.parse_expression().as_const())

            cache_tags = self.environment.prudence_caching.tags
            for tag in tags:
                cache_tags.add(tag)

        return _literal('', lineno)

def _print(text, lineno):
    return nodes.Output([nodes.TemplateData(text)]).set_lineno(lineno)
```

We'll then modify our "/resources/jinja-template.t.html" to use our extension, and set it up using the attributes forwarded from the dispatcher.:

```
env = Environment(loader=loader, extensions=['jinja_extensions.Prudence'])
env.prudence_caching = caching
env.prudence_uri = payload['uri']
env.prudence_base_uri = payload['base_uri']
```

Here's a simple template to test the extensions:

```
<html>
{% cache_duration 5000 %}
{% cache_tags 'tag1', 'tag2' %}
<body>
<p>This page is cached for 5 seconds.</p>
<p><b>current_uri</b>: {% current_uri %}</p>
<p><b>application_uri</b>: {% application_uri %}</p>
<p><b>to_base</b>: {% to_base %}</p>
```

```
</body>
</html>
```

RESTful Models

In our MVC tutorial above, we've implemented our models as classes (OOP). However, it may make sense to implement them as RESTful resources instead.

Doing so allows for powerful deployment flexibility: it would be possible to decouple the model layer entirely, over HTTP. For example, you could have "model servers" running at one data center, close to the database servers, while "presentation servers" run elsewhere, providing the direct responses to users. In this scenario, the presenters would be calling the models using secured HTTP requests, instead of function calls.

Even if you're not planning for such flexibility at the moment, it might still be a good idea to allow for it in the future. Until then, you could optimize by treating the model layer as an internal API (page 115), which makes it about as fast as function calls.

There are two potential downsides to a RESTful model layer. First, there's the added programming complexity: it's easier to create a class than a resource. Second, RESTful resources are limited to four verbs: though GET/POST/PUT/DELETE might be enough for most CRUD operations, it can prove harder to design a RESTful URI-space for complex "business logic."

A good compromise, if necessary, can be to still use HTTP to access models, just not RESTfully: use Remote Procedure Call (RPC) instead. We discuss this option in the URI-space architecture tips (page 157).

Tutorial

For simplicity, we'll use a mapped resource, "/resources/models/person.m.js":

```
document.require(
        '/models/user/',
        '/sincerity/json')

function handleInit(conversation) {
        conversation.addMediaTypeByName('application/json')
        if (conversation.internal) {
                conversation.addMediaTypeByName('application/internal')
        }
}

function handleGet(conversation) {
        var name = conversation.locals.get('name')
        var person = Models.getPerson(name)
        var result = {
                username: person.getUsername(),
                comments: person.getComments().get()
        }
        return conversation.mediaTypeName == 'application/internal' ? result : Sincerity.JSON
}

function handlePost(conversation) {
        var name = conversation.locals.get('name')
        var payload = Sincerity.JSON.from(conversation.entity.text)
        if (payload.comment) {
                var person = Models.getPerson(name)
                person.getComments().add(comment)
                var result = {
                        username: person.getUsername(),
                        comments: person.getComments().get()
                }
                return conversation.mediaTypeName == 'application/internal' ? result : Sincer
        }
        return 400
}
```

Note that we've optimized for internal requests (page 116).

We would then modify our presenter like so:

```
function handleGet(conversation) {
        var name = conversation.locals.get('name')
        var person = getModel('person/' + encodeURIComponent(name))
        return getView('comments', {person: person})
}

function handlePost(conversation) {
        var name = conversation.locals.get('name')
        var comment = conversation.form.get('comment')
        var person = postModel('person/' + encodeURIComponent(name), {comment: comment})
        return getView('comments', {person: person})
}

function getModel(model) {
        return Prudence.Resources.request({
                uri: '/models/' + model + '/',
                internal: true
        })
}

function postModel(model, payload) {
        return Prudence.Resources.request({
                uri: '/models/' + model + '/',
                internal: true,
                method: 'post'
                payload: {
                        type: 'object',
                        value: payload
                }
        })
}
```

We've assumed an internal request here, but it's easy to change it to an external request if the model layer runs elsewhere on the network.

Finally, here's our addition to routing.js, using capture-and-hide (page 26):

```
app.routes = {
        ...
        '/models/person/{name}/': '/models/person/!'
}
```

Clusters

Multiple JVMs running Prudence, on several machines or on a single one, may share certain resources. The running instances do not have to be identical ("redundant"), though in some deployment scenarios (page 140) they should be.

Many of the clustering features in Prudence rely on the excellent Hazelcast library: Hazelcast allows Prudence nodes to to share state and locks and to run tasks remotely. Its auto-discovery feature is especially useful for flexible cloud deployments.

You don't have to dig deep into Hazelcast: the core features work out-of-the-box with Prudence, and are fully explained in this chapter. However, if you're serious about clustering, it's a good idea to study Hazelcast and see what more, perhaps, it can do for you. Especially, you may want to tweak the default configuration (page 117), for example to add backups, change eviction policies, create node groups, enable SSL, encryption, etc.

Shared State

Easily share in-memory data between all nodes and applications in the cluster using the application.distributedGlobals and application.distributedSharedGlobals API family (page 83).

Concurrency

Like the other globals, distributed globals support concurrently atomic operations (page 84). In the following example, we make sure that the default value is only ever set once:

```
var defaultPerson = {name: 'Linus', role: 'boss'}
var person =
        Sincerity.JSON.from(application.getDistributedGlobal('person', Sincerity.
           JSON.to(defaultPerson)))
```

Serializability

Distributed globals *must be serializable*. If they aren't, you will get an exception when Hazelcast attempts to move the data between nodes.

If you're using JavaScript, this unfortunately means that you are limited to primitive types. One easy way to get around this is to serialize via JSON:

```
document.require('/sincerity/json/')
var person = {name: 'Linus', role: 'boss'}
application.distributedGlobals.put('person', Sincerity.JSON.to(person))
person = Sincerity.JSON.from(application.distributedGlobals.get('person'))
println(person.name)
```

Configuration

You can configure the distributed globals in "/configuration/hazelcast/application/globals.js" (page 117), as a map named "com.threecrickets.prudence.distributedGlobals.[name]", where "name" is the application name. You can also change the name of that map in settings.js (page 74).

Cluster-Wide Synchronization

Use the application.getDistributedSharedLock API (page 84) to synchronize access to resources for the entire cluster:

```
var lock = application.getDistributedSharedLock('services.remote')
lock.lock()
try {
        doSomethingAtomicallyWithRemoteService()
}
finally {
        lock.unlock()
}
```

This apparently simple tool offers a reliable and thus powerful guarantee for atomicity across entire deployments, even very large ones. It can replace the need to use a separate, dedicated synchronization tool, such as is provided Apache ZooKeeper. Just make sure you understand the detrimental effect its use could have for your scalability.

See the Hazelcast documentation for more information on distributed locks.

Task Farms

The distributed task APIs (page 105) let you spawn tasks anywhere, everywhere, or on specific nodes in the cluster.

This powerful feature is only really useful in a heterogeneous cluster: if all your nodes are the same, and all of them are answering user requests behind a load balancer (page 142), then it's hard to see what advantage you would get by spawning background tasks on a node other than the one that answered the request. However, if you have a *separate* set of nodes specifically designated to running tasks, or even multiple sets dedicated to tasks of different kinds, then you gain considerable control over your deployment strategies. You can scale out your "web nodes" if you need to handle more user requests, and separately scale out your "task nodes," "cache nodes," "database nodes," etc. This deployment strategy is key to cost-efficient use of your resources.

Prudence supports two ways to create such a heterogeneous cluster.

Tagging Nodes

You can "tag" each node in your cluster with one or more custom strings. Many nodes may share the same tag, or you may create unique tags for some nodes in order to refer to them, and only them, directly.

To tag nodes, edit their "/configuration/hazelcast/default/default.js", and set the "com.threecrickets.prudence.tags" attribute to a comma-separated list of tags (every node can be associated to one or more tags). As an example, let's give our node two tags, "video-encoding" and "backup":

```
config.memberAttributeConfig.setStringAttribute('com.threecrickets.prudence.tags', 'video-enc
```

We can then use the distributed task APIs (page 105) to execute a single task on any one of the "video-encoding" nodes:

```
Prudence.Tasks.task({
        uri: '/reencode-video/,
        context: {filename: 'great_movie.avi', resolution: {w: 600, h: 400}},
        json: true,
        distributed: true,
        where: 'video-encoding'
})
```

Or, we can execute a backup operation on *all* the "backup" nodes, by setting the "multi" param to true:

```
Prudence.Tasks.task({
        uri: '/backup/,
        distributed: true,
        multi: true,
        where: 'backup'
})
```

Note that "where" can also be a comma-separated list.

Separate Cluster for Tasks

In a more complex deployment, you may want your task farm as an entirely separate Hazelcast cluster. For example, your task farm may be running in an entirely different data center, and you do not need or want it to share state with the application cluster.

By default, Prudence executes all tasks on one Hazelcast instance, but it allows you to configure a separate instance for tasks. To enable this scenario, Prudence comes with commented-out configurations (page 117) for both the "task" nodes (the servers) and for the nodes that will be spawning the tasks (the clients):

- "/configuration/hazelcast/task/1-server.js": Enable this if you want this node to be a full member of the "task" cluster, which means it will be able to accept and run distributed tasks. Note that the node would still be a member of the "application" cluster (unless you disable its configuration explicitly), meaning it will be able to share state (page 138) with the application nodes.
- "/configuration/hazelcast/task/2-client.js": Enable this if you want this node to be able to spawn tasks in the "task" cluster. This creates a lightweight HazelcastClient for the "tasks" cluster, which is not an actual member.

It normally doesn't make sense to have both "1-server.js" and "2-client.js" enabled on the same node: a server is already a full member, and doesn't have to also be a lightweight client. However, it can be useful to enable both for the purpose of testing the complete loop in a single container: it *will* work. (That's why there are numeric prefixes for the filenames: this makes sure they are initialized in the correct order if both are enabled.)

Once configured, you can use the task APIs as usual on both the client and server nodes, but the tasks will only run on the server nodes. Note that you can still use node tags (page 139) for the server nodes by editing "1-server.js".

For more options for partitioning clusters in Hazelcast, see its grouping feature.

Shared Cache

Obviously, in a cluster you want to use a shared cache backend, and even implement a tiered caching strategy. The configuration chapter (page 123) details your many built-in options.

Centralized Logging

There are four possible strategies for handling logging in a cluster, each with its own advantages:

- Let every node keep its own log files, which is the default configuration in Prudence. This makes it easier to debug problems specific to each node. However, in a load-balancing scenario (page 142) it can be very hard to follow user activities over time, because each request might be logged on a different node.
- Centralize all logging. There are actually two ways to achieve this in Prudence:
  - Log to a database. Sincerity's logging plugin comes with powerful support for logging to MongoDB.
  - Run a dedicated logging node (a Apache Log4j server). This is again explicitly supported by Sincerity's logging plugin: it allows all your nodes to send their log messages to the logging node over the network. The logging node will be doing the actual message writing.
- It's easy to log both locally and centrally, immediately giving you the benefits of both worlds, at the cost of some wasted resources due to the duplication.
- A hybrid approach can be the best idea: some loggers might be stored locally, others might write to the central log.

Deployment

"Deployment" here refers to getting your Prudence applications running so that users can, well, use them. The challenge is that the development environment is often quite different from the production, staging and testing environments, and indeed it's the point where development work must integrate with systems administration and operations.

Approaching deployment can quickly mire you into a comparison of ideologies of project management: some prefer Continuous Integration (CI) and agile methods, others prefer more careful deployment by humans according to step-by-step plans. We'll bypass the ideological discussion here, and deal specifically with the technical possibilities and tools. It's up to you to decide which deployment technologies fit best with or best enable your project management ideology.

Deployment Strategies

File Synchronization

Sometimes the best strategies are the most straightforward.

Because your entire Prudence installation is contained in one directory, you can simply copy it from your development environment to your deployment environments. Even better, you can use a synchronization tool like rsync, which will efficiently copy only the updated/new files. Best of all, you can use a two-way synchronization tool like Unison, which allows on-the-fly changes you make at the deployment environment to synchronize back to your development environment. Both tools mentioned use compression and batch transfers for speed and can run over SSH for security.

Actually, the deployment origin does not have to be a single programmer's development environment: you can create a dedicated deploying environment from which to deploy to all nodes and have it shared among a team of programmers.

Example Let's see how this is done using Unison. First, let's create a profile in our development environment. We'll store it in "~/.unison/production.prf":

```
root = /path/to/prudence
root = ssh://node1.mysite.org//path/to/prudence
root = ssh://node2.mysite.org//path/to/prudence
ignore = Path cache
ignore = Path logs
ignore = Path component/applications/stickstick/data
ignore = Path .git
ignore = Path .gitignore
```

Note how easy it is to include several nodes in a single profile. Also note that likely want to exclude syncing a few localized directories: in this example, we're ignoring an extra data path and also files used by our version control system, Git.

To synchronize with the above profile, run this:

```
unison production -batch
```

Of course, you can create additional profiles, for example "staging.prf" for your staging environment.

Version Control Systems (VCS)

If you're already using a VCS, why not use it to deploy your applications, too? In many ways, this is as straightforward as file synchronization, though there are a few important advantages and disadvantages:

- You don't need a deploying origin: every node can update itself.
- You can automatically upgrade/downgrade simply by checking out a revision, tag or branch.
- Distributed VCSs (such as Git and Mercurial) normally require a clone of the entire repository, including its history. This can be quite heavy and redundant. One solution is to do a "shallow clone," which avoids the history. However, shallow clones have several limitations that might make them difficult to use.
- VCSs don't deal well with large binary files. For development work, there are good solutions, such as git-annex for Git. However, these solutions don't solve the problem for deployment: you still need to get those binaries to your deployed environment somehow.
- Allowing VCS access from your deployed environment might be an unacceptable security risk.

You can also adopt a hybrid strategy: use VCS to deploy the main application code, and install the other parts of it (including Prudence itself) via some other means.

Be sure to read the Sincerity tutorial, which gives you a few suggestions to using a VCS with Prudence, which apply to development as well as to deployment.

Packaging

You can encapsulate your entire Prudence container, or individual applications and services, into deployable, versioned, interdependent, signed packages.

Though configuring and creating the packages is the hard part, deploying them is often very easy. Packaging's big advantage in how easy and safe it makes uninstalling, upgrading and downgrading processes. The strategy indeed reveals many of its advantages when it is used modularly: it makes it possible to install/upgrade/downgrade only specific applications or services, while leaving the rest of the deployment intact. Different types of nodes could thus be installed as assemblages (meta-packages) of particular packages. Finally, careful management of dependencies can be used to ensure that the package has everything it needs to run properly.

Modularity has a huge cost in terms of project management complexity, which should not be underestimated. However, if you're already managing your project as separate modules, with their own roadmaps and version progression, it can make a lot of sense to deploy it that way, too.

There are many packaging standards and tools out there, but we'll mention a few that you are especially likely to use with Prudence.

Docker Because everything is one directory, and the only requirement is a JVM (actually, just a JRE), it's trivial to package your Prudence containers in Docker. See the Sincerity Manual for instructions.

Maven Apache Maven is a comprehensive (and highly complex) project management tool for the JVM, especially targeted at the Java language and related technologies. Whether or not you use the Maven tool itself, its repository format (also known as iBiblio) has become the *de facto* standard for JVM packaging.

The Sincerity tool, on which Prudence is itself distributed, uses the Maven repository format, but adds a few important (and optional) features to its packaging specification, namely the ability to unpack archives into the container, and to run install/uninstall hooks for each package. We recommend Sincerity for Maven-type package deployment: it will handle not only your own packages, but also Prudence itself, as well as other Sincerity plugins and add-ons.

You can package and publish your packages using the Maven tool: the Sincerity packaging documentation includes a template configuration for Maven, which you can easily modify for your own packages (and bypass Maven's notorious learning curve). Alternatively, you can use easier tools like Gradle and Ivy.

Maintaining and managing the repository is easy enough. At its most straightforward, you can simply host the repository's filesystem via a web server. However, there are also several powerful tools and hosted solutions offering many useful features, such as proxying of other repositories. Sonatype's Nexus is especially easy to install and get running using Sincerity. Another great option is JFrog's Artifactory.

Debian and RPM If you are deploying to nodes based on a Linux-based operating system, then you're are likely already using a packaging system: either Debian or RPM. Using the native packaging system for your own deployment gives you the very useful advantage of allowing for dependencies to OS packages, as well as having a single, unified packaging system for *everything.* At the very least, for example, you'll want your Prudence packages to depend on a JVM. By creating a meta-package for each node type, you can then install and upgrade entire nodes, starting from a freshly installed operating system, by simply installing a single package.

Debian is the the more complex of the two standards: it's actually not just a file format, but part of a comprehensive, integrated operating system build system, requiring several highly specific configuration files per package. It might be easier to use more minimal tools for packaging your Prudence applications: we recommend jdeb for Debian and Redline for RPM, which both run on the JVM and can be integrated into Ant builds.

Another important advantage of using Debian or RPM is that you can integrate your Prudence packages into comprehensive infrastructure management and orchestration tools. There is a great variety among these: some are tied to specific operating systems, some are hosted, some proprietary. If you're using Ubuntu, you can use Juju and Landscape. For RedHat and CentOS, you can use YADT, which can also be used for your build process.

Load Balancing and Proxies

One of the great advantages of REST architectures (page 152) is that they're trivial, in an architectural sense, to scale horizontally: any number of identical nodes can sit comfortably behind a load balancer. Because each REST request is self-contained, it doesn't matter which node handles which request.

> Well, that's a bit idealized. Actually, requests are not themselves identical: some might need access shared resources, such as databases and task farms (page 139), and in complex applications it may make sense to have different kinds of nodes answering different kinds of requests, or at the very least it may be important to route certain requests to certain nodes that would do a better job at servicing the request (for example, if they are nearer to the specific resources the request needs). Your routing needs might be quite sophisticated. See a more comprehensive discussion of the "partitioning" problem in the scaling tips article (page 166). Nevertheless, for simpler applications load balancing is indeed trivial, and ready-made products, services and algorithms will fit most use cases.

Clusters

You don't *have to* enable clusters (page 137) in order to create a load-balanced Prudence deployment. However, you *can* use the cluster features to allow for powerful cooperation between nodes. In particular, they can share a Hazelcast cache backend (page 123).

Also take a look at the task farming (page 139) feature: you can run a separate task farm cluster without the application nodes forming a cluster themselves.

Choices

Good load balancers do more than just scale: they allow for robustness by removing problematic nodes from the pool, either because of errors or because of poor performance. Often, you can configure the various thresholds and the behavior of the back-off algorithms.

If you're deploying to a hosted "cloud" environment, it could be that your host provides a load-balancing solution. It's often a good choice: these load balancers will likely perform better and be more reliable than those running inside a virtual host. However, they may not be flexible (or trustworthy) enough for your needs. It's easy enough to install your own load balancer using a wide range of products: we'll provide examples for using two popular solutions below.

> Who should handle SSL? Prudence can handle SSL perfectly fine on its own (page 120). However, when using a load balancer, you may have the option of "terminating" SSL there. The problem with terminating SSL early is that you have unencrypted packets moving between the load balancer and your nodes, which is a security risk. You definitely want SSL going all the way to Prudence if you're deployed in an environment you can't trust. Otherwise, terminating early is often recommended, as it can offer better utilization of your application node resources, and allow for simpler deployments. Both examples below demonstrate how to terminate SSL at the load balancer.

Nginx

Nginx is a popular general-purpose web server, which has several high-quality modules. It's a good choice if you need other features in addition to load balancing, but also works fine as a standalone load balancer. Refer to the documentation for the proxy and upstream modules for a complete reference.

Here's a simple configuration:

```
http {
  server {
    listen 80;
    location / {
      proxy_pass http://prudence;
    }
  }

  server {
    listen 443 ssl spdy;
    ssl on;
    ssl_certificate_key /etc/ssl/server.key;
    ssl_certificate /etc/ssl/server.crt;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-G
    ssl_prefer_server_ciphers on;
    ssl_session_timeout 5m;
    ssl_session_cache shared:SSL:5m;
    location / {
      proxy_pass http://secure_prudence;
    }
  }

  upstream prudence {
    server node1.myapp.org:8080;
    server node2.myapp.org:8080;
    server node3.myapp.org:8080;
  }
  upstream secure_prudence {
    server node1.myapp.org:8081;
    server node2.myapp.org:8081;
    server node3.myapp.org:8081;
  }
}
```

Nginx offers some useful routing features. For example, you can gives nodes "weights," where a higher weight means that more requests will be sent to that node:

```
upstream prudence {
  ip_hash;
  server node1.myapp.org:8080 weight=1;
  server node2.myapp.org:8080 weight=2;
  server node3.myapp.org:8080 weight=4;
}
```

Or you can enable client IP-based routing, so that requests from a particular client will always go to the same node:

```
upstream prudence {
  ip_hash;
  server node1.myapp.org:8080;
  server node2.myapp.org:8080;
  server node3.myapp.org:8080;
}
```

Perlbal

Perlbal is a minimalist web server dedicated solely to load balancing. It offers very few configuration options, but is eminently hackable due to being written in crisp Perl. It is recommended for users who don't like fiddling with knobs or who appreciate single-purpose tools.

Here's an example "perlbal.conf":

```
CREATE POOL pool
  SET nodefile = /etc/perlbal/nodes

CREATE POOL secure_pool
  SET nodefile = /etc/perlbal/secure_nodes

# HTTP
CREATE SERVICE balancer
  SET listen          = 0.0.0.0:80
  SET role            = reverse_proxy
  SET pool            = pool
  SET verify_backend  = on

# HTTPS
CREATE SERVICE secure_balancer
  SET listen          = 0.0.0.0:443
  SET role            = reverse_proxy
  SET pool            = secure_pool
  SET verify_backend  = on
  SET enable_ssl      = on
  SET ssl_key_file    = /etc/ssl/server.key
  SET ssl_cert_file   = /etc/ssl/server.crt
  # The following is recommended to work around a bug in older versions of IE
  # (the default is ALL:!LOW:!EXP)
  SET ssl_cipher_list = ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+
     eNULL

# Internal management port
CREATE SERVICE mgmt
  SET role   = management
  SET listen = 127.0.0.1:60000

ENABLE balancer
ENABLE secure_balancer
ENABLE mgmt
```

The "nodes" file is a list of IP addresses (not hostnames!) with ports. We'll add three Prudence instances running at the default server port:

```
192.168.1.10:8080
192.168.1.11:8080
192.168.1.12:8080
```

The "secure_nodes" file is the same, but uses our separate server port:

```
192.168.1.10:8081
192.168.1.11:8081
192.168.1.12:8081
```

If the node files are edited, Perlbal will pick up their changes on the fly.

Web Data

The load balancer handles user requests *instead* of your Prudence instances. (This is sometimes called, from the client's perspective, a "reverse" proxy.) This means that request headers might be modified *before* they reach you, and response headers *after* they leave you.

Your Host One thorny issue is that the host and port (and scheme, if you are proxying https to http) you get are different from those the client sent. For example, the client might have made a request to "https://myapp.org/myapp/" (at port 443), but it reaches your node as "http://192.168.1.10:8081/myapp/".

In terms of routing your URI-space, this is not a problem: Prudence always uses the URL in the original request. However, you might care about which server *you* are on, for example if you need to access local services.

Usefully, the standard HTTP/1.1 "Host" header can be used here: your load balancer will likely set it to be your server. In Prudence, you can access its parsed value via the conversation.request.hostRef API:

```
var hostRef = conversation.request.hostRef
var host = hostRef.hostDomain // '192.168.1.10'
var port = hostRef.hostPort // 8081
var protocol = hostRef.scheme // 'http'
```

The same value is used for virtual host configuration (page 118).

In some cases you might want the load balancer to work transparently, leaving the original "Host" header intact. This behavior is sometimes configurable in load balancers.

For example, in Nginx:

```
location / {
  proxy_pass http://prudence;
  proxy_set_header Host $host;
}
```

Forwarded Headers One problem with the standard "Host" header is that it only contains the host and port, but not the scheme. If you're proxying https to http, and are setting the "Host" header to work transparently, you will nevertheless lose the original scheme.

Though the HTTP/1.1 specification does not have a solution to this problem, we can use a widely supported *de facto* standard, first introduced in Squid: the “X-Forwarded-Proto” header. Also, “X-Forwarded-Host” (and/or “X-Forwarded-Port”) can be used *instead* of “Host”, allowing you to retain the “Host” of the proxy without overwriting it.

You can enable support of these headers in Prudence per application by setting “app.settings.routing.useForwardedHeaders” (page 74) to true in your application’s settings.js:

```
app.settings = {
        ...
        routing: {
                useForwardedHeaders: true
        }
}
```

By default, Prudence does *not* enable the interpretation of these headers, because if you’re *not* behind a proxy, it would allow clients to manipulate the information.

Note that you can also use the ForwardedFilter directly for more fine-grained control over which requests will use these headers.

Your load balancer must also be configured to set these headers. For example, in Nginx:

```
location / {
  proxy_pass http://prudence;
  proxy_set_header X-Forwarded-Proto $scheme;
  proxy_set_header X-Forwarded-Port $server_port;
}
```

Client IP Address Prudence’s conversation.client.upstreamAddress API identifies the request’s client, however, in a load-balancing scenario, the client is actually the load balancer itself. This is tricky: there actually might be various components (load balancers, caches, gateways) along the way, so how can the original IP address be preserved?

We can use the “X-Forwarded-For” header (a *de facto* standard), which is a comma-separated list of all client IP addresses in order. Each component along the way can append itself to before forwarding onward. The first IP address would thus be the original client.

The default server configuration (page 120) in Prudence does *not* enable the interpretation of this header, because if you’re *not* behind a proxy, it would allow clients to manipulate the information. To enable it, uncomment or add this line in “/component/servers/http.js”:

```
server.context.parameters.set('useForwardedForHeader', 'true')
```

See the relevant Restlet documentation here and here. Also note that you can use conversation.client.forwardedAddresses API to access the complete list.

Your load balancer must also be configured to set “X-Forwarded-For”. For example, in Nginx:

```
location / {
  proxy_pass http://prudence;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Adaptable Configurations

The Sincerity tool, which is used to bootstrap Prudence, gives you a lot deployment power.

In particular, the "configuration-by-script" principle means that you don’t have to create separate configuration files for different deployment environments: you can use the same deployed files for development, staging, testing and production.

This depends on the scripts discovering in what environment they are running, and performing the appropriate configuration. Here are a few suggested discovery methods:

By System Configuration File You can configure your node via a single file. For example, let's create "/etc/deployment":

```
DEPLOYMENT=production
```

In our Prudence bootstrapping scripts, we can parse the file safely like so:

```
document.require('/sincerity/jvm/')
var deployment = 'staging'
try {
deployment = Sincerity.JVM.fromProperties(Sincerity.JVM.loadProperties('/etc/deployment')).DE
} catch(x) {}
println('Deployment: ' + deployment)

if (deployment == 'development') {
        ...
}
```

Note that we are defaulting to "staging" in case the file doesn't exist. Defaulting to "development" might be risky: development deployments usually reveal too much data, or otherwise provide security overrides.

The advantage of this file format is that it's very easy to parse in other languages, so you can use it to configure non-Prudence components, too. Here's an example in bash:

```
if [ -f /etc/deployment ]; then
        . /etc/deployment
        echo "Deployment: $DEPLOYMENT"
else
        DEPLOYMENT=staging
        echo "Deployment: staging (default)"
fi
```

By IP Address Your deployment environment might be known according to the local IP address or subnetwork. It's possible to lookup the IP address in a table, or other parse parts of it (the subnetwork).

To retrieve it, the following *might* work:

```
var address = java.net.InetAddress.localHost.hostAddress
```

This API won't work as expected in some local configurations, instead returning the loopback address, "127.0.0.1". Also, some nodes may have multiple IP addresses. To iterate *all* local IP addresses, use this code:

```
var addresses = []
for (var e = java.net.NetworkInterface.networkInterfaces; e.hasMoreElements(); ) {
        var interface = e.nextElement()
        for (var ee = interface.inetAddresses; ee.hasMoreElements(); ) {
                var address = ee.nextElement()
                if (!address.loopbackAddress) {
                        addresses.push(address.hostAddress)
                }
        }
}
```

By "Cloud" API If you're running in a "cloud" environment, your host likely provides you with an API service to discover your node's name, group it belongs to, etc., making it easy to determine your deployment environment.

Many cloud environments support the OpenStack API, which you can access easily directly via RESTful requests (page 60) or a dedicated JVM wrapper. Specifically, its identity service can be used to retrieve information about the current node.

For example, let's list the tenants of our node:

```
document.require('/prudence/resources/')
var openstackBaseUri = ...
var tenants = Prudence.Resources.request({
        uri: openstackBaseUri + 'v2.0/tenants',
        mediaType: 'application/json'
})
for (var t in tenants) {
        var tenant = tenants[t]
        println(tenant.name)
}
```

Note that the above code will only work from a running Prudence application: during bootstrapping, you will have to use a different method to make REST requests, for example the dedicated JVM wrapper.

Operating System Service (Daemon)

In production environments it's recommend to have Prudence installed as an operating system service ("daemon"), which allows you to use standard tools to start, stop and monitor its status. It will also guarantee that Prudence starts as quickly as possible when the system starts, reducing downtime in the cases of restarts.

In Prudence this is best handled by installing the Sincerity service plugin, which adds a powerful native "wrapper" service around the JVM. The documentation there also provides you with examples of how to install it in your operating system.

Note that the wrapper uses its own logging system, separate from the one used by Prudence (page 88), though the default configurations of both will work well together.

Security

Securing your deployment is important whether you're running an Internet site or a local intranet site. Actually, "security" is an umbrella term for various aspects dealing with, ahem, unintended use of your site:

- You want to make sure that outsiders do not have access to your secret data.
- You want to make sure that outsiders, and possibly even *you*, don't have access to *other users'* data, for which you are the trusted caretaker.
- You want to make sure that outsiders cannot take modify your site's operation, essentially hijacking it for their own uses, while using your users' confidence in you for their own purposes.
- You want to make sure that outsiders cannot stop your site from operating. This is often called a Denial of Service (DoS) attack.

It's crucial that you study this topic well or hire an expert to handle it for you. It's astounding how often big, "trusted" companies fail miserably in securing their applications. Don't wait until the fire starts to put it out: prevent it from ever happening.

Unfortunately, it's not a simple problem field: attacks are getting more and more sophisticated, using sheer computational prowess to break through encryption, "rainbow" attacks to uncover user passwords, and finding clever loopholes for injecting code into your application or database runtime. We definitely can't cover all aspects of security here, especially because it's a moving target. But we'll cover some essentials that are directly related to Prudence and the deployment technologies mentioned in this chapter.

> Is Prudence less secure, as an open source product, than proprietary alternatives? To be honest, there's something scary about revealing all your cards to the hackers. At the same time, these cards are also seen by the community of users, like you, who have an interest in plugging security holes as soon as they're discovered. With closed-source software, ignorance is bliss: you have to rely on indirect knowledge to evaluate just how secure it is. And if a loophole is discovered, you have to rely on others to fix it. With Prudence, you don't have to wait for us: we've gone to great lengths not only to share the code, but also to make it as easy as possible for you to build your own patched-up Prudence. When all the factors are considered, we believe that open source is the better choice in the long run for those who care about security.

Locked-Down User

When applications are hacked, your only line of defense is the operating system. And let's put is plainly: all applications have exploits. Thus, especially because it's so easy, there's no excuse not to implement basic operating system security.

The first thing you should do is create a special user that will spawn and thus own the Prudence process. If you're running Prudence as a service (page 148), then you should install that service via that user.

Then, *lock that user down.* All operating systems let your control file access per user, so make sure the user can only exact *only* the files it needs. Most operating systems are too promiscuous by default, so make sure that nothing important is readable by your designated user.

But accessing the files is only the tip of the iceberg: you want Mandatory Access Control (MAC), too, to limit the user's ability to execute processes it shouldn't. If you're deploying on Linux, consider using AppArmor, which allows for simple profiles to configure the use of Linux's many security features.

Firewall

Whether you're running a cluster (page 137) or a single server, there's no reason unused ports should be open to the world, and to mischief. Get yourself a firewall.

If you're deploying on Linux, iptables (a netfiler module) is the standard solution. However, it can be quite daunting to manage directly, so consider using a higher-level tool instead: we recommend the Uncomplicated Firewall (UFW), which allows for per-application profiles, and comes with sensible iptables defaults.

Which ports should you leave open?

HTTP and HTTPS The standard and default Internet ports for these are 80 and 443 respectively. Of course, you are free to use other ports for your deployment, for which there might be security advantages.

Cluster Are you running your nodes as a cluster (page 137)? The default port used by Hazelcast is 5701, though it's easy to change it.

Cache Backends Do you need access to a shared cache backend (page 123)?

- Hazelcast: See cluster, above: the default port is 5701.
- memcached: The default port is 11211.
- MongoDB: The default port is 27018.
- SQL: This varies, refer to your product's documentation.

HTTPS

Enabling TLS/SSL encryption for HTTP is *usually* more about protecting your user data than protecting your own. And indeed, this service you provide to your users is costly in terms of CPU cycles it requires and certification authorities you must pay for generally trusted certificates.

HTTPS also has an entirely different use case, specifically when it's used with privately issued certificates: you gain a powerful authentication barrier. Only users who own the certificate would be able to access the protected resources.

You have the option of handling SSL directly in Prudence (page 120) or "terminating" at the load balancer (page 142).

HTTP Authentication

Prudence supports HTTP basic authentication (page 112), which is in turn supported by most web browsers and other HTTP clients. Coupled with HTTPS (in order to ensure that the password is transferred securely to you), it's actually not such a bad way to secure your resources. Web browsers will cache user credentials for the duration of a "session," which usually means until the web browser is closed.

> HTTP authentication is secure enough when coupled with HTTPS, but it's not a scalable way to support many users and sessions. If you need a comprehensive solution, consider Diligence's authentication service.

Quarantine the Admins

Many sites allow for "administrative" logins that have special privileges and abilities. Probably, these abilities would be utterly destructive in the wrong hands.

It thus makes a lot of sense to quarantine these users and their needs into a separate security domains: it should be harder for an admin to login than for a regular user. All defenses should be up: HTTPS with private certificates, a separate cluster firewalled from the regular application nodes, and sharing *only* the resources that are absolutely necessary for admin functionality.

Utilities for Restlet

If you are a Restlet Java programmer, Prudence may still be of use to you. The Prudence standalone Jar has several well-documented classes useful for any Restlet application. They're all in the "com.threecrickets.prudence.util" package, and introduced below.

Also, remember that can use Sincerity without Prudence: Sincerity's Restlet skeleton gives you all the benefits of JavaScript-based configuration and bootstrapping for your Java Restlet applications. Use Java where it matters, JavaScript where it doesn't: don't waste your time recompiling Java when all you need is some external, deployment-related features. Edit your scripts on the fly, and you're done.

Utility Restlets

We wish these general-purpose utilities existed in the standard Restlet library!

- CacheControlFilter: A Filter that adds cache control directives to responses. Great for wrapping around a Directory.
- CorsFilter: A Filter that add Cross-Origin Resource Sharing (CORS) response headers.
- CustomEncoder: A more localized alternative to using the EncoderService. Place it as a Filter before any restlet.
- ForwardedFilter: A Filter that applies the X-Forwarded-Proto, X-Forwarded-Host, and X-Forwarded-Port HTTP headers to the request's references.
- Injector: A Filter that adds values to the request attributes before moving to the next restlet. It allows for a straightforward implementation of IoC (Inversion of Control). Automatically supports casting of Template instances.
- StatusRestlet: A restlet that always sets a specific status and does nothing else.
- VirtualHostInjector: A Filter that sets the virtual host as a request attribute.

Client Data

These classes add no new functionality, but make working with some client data a bit easier.

- CompressedStringRepresentation: This is a ByteArrayRepresentation that can be constructed using text and an encoding, which it then compresses into bytes according the encoding. This is an alternative to using an Encoder filter, allowing you direct control over and access to the final representation.
- ConversationCookie: A modifiable extension of a regular Cookie. Tracks modifications, and upon calling save() stores them as a CookieSetting, likely in the Response. Also supports cookie deletion via remove().
- FormWithFiles: A form that can parse MediaType.MULTIPART_FORM_DATA entities by accepting file uploads. Files will appear as parameters of type FileParameter.
- InternalRepresentation: A representation used to transfer internal objects. Uses a custom "application/internal" media type.

Redirection

Restlet's server-side redirection works by creating a new request. Unfortunately, this means that some information from the original request is lost. Prudence includes a set of classes that work together to preserve the original URI, which we here call the "captured" URI.

- CapturingRedirector: A Redirector that keeps track of the "captured" (original) reference.
- NormalizingRedirector: A Redirector that normalizes relative paths. This may be unnecessary in future versions of Restlet: see Restlet issue 238.

Fallback Routing

"Fallback" is a powerful new routing paradigm introduced in Prudence that lets you attach multiple restlets to a single route.

- Fallback: A restlet that delegates Restlet.handle(Request, Response) to a series of targets in sequence, stopping at the first target that satisfies the condition of wasHandled. This is very useful for allowing multiple restlets a chance to handle a request, while "falling back" to subsequent restlets when those "fail."
- FallbackRouter: A Router that takes care to bunch identical routes under Fallback restlets.

Resolver Selection

Restlet does not provide an easy way to use different template variable resolver instances (see Restlet issue 798 for more information). We've created new implementations of a few of the core classes that let you choose which resolver to use.

- ResolvingTemplate: A Template that allows control over which Resolver instances it will use.
- ResolvingRedirector: A Redirector that uses ResolvingTemplate. It also allows another useful feature: turning off the cleaning of headers during server-side redirections.
- ResolvingRouter: A Router that uses ResolvingTemplate for all routes.

Web Filters

A set of Filter classes for web technologies.

- CssUnifyMinifyFilter: Automatically unifies and/or compresses CSS source files, saving them as a single file. Unifying them allows clients to retrieve the CSS via one request rather than many. Compressing them makes their retrieval faster. Compression is done via CSSMin.
- JavaScriptUnifyMinifyFilter: Automatically unifies and/or compresses JavaScript source files, saving them as a single file. Unifying them allows clients to retrieve the JavaScript via one request rather than many. Compressing them makes their retrieval faster. Compression is done via John Reilly's Java port of Douglas Crockford's JSMin.
- LessFilter: Automatically parses LESS code and renders CSS using the Less4j library. Also supports minifying files, if the ".min.css" extension is used.
- ZussFilter: Automatically parses ZUSS code and renders CSS. Also supports minifying files, if the ".min.css" extension is used.

Other Utilities

- CustomAuthenticatorHelper: Makes it easy to register custom authentication schemes.
- EnhancedCallResolver: Improves on the default Restlet CallResolver.

Upgrading from Prudence 1.1

Prudence 1.1 did not use Sincerity: instead, it was a self-contained container with everything in the box. This meant it could also not be modular, and instead supported several distributions ("flavors") per supported programming language. Rather than standardizing on a single language for bootstrapping code, and indeed the project maintained a separate set of bootstrapping code for each supported language.

This was not only cumbersome in terms of documentation and maintenance, but it also made it hard to port applications between "flavors."

With the move to Sincerity in Prudence 2.0, it was possible to make Prudence more minimal as well as more modular, as Sincerity handles the bootstrapping and installation of supported languages. Though Sincerity can ostensibly run bootstrapping scripts in any Scripturian-supported language, it standardizes on JavaScript in order to maintain focus and portability. The bottom line is that if you used non-JavaScript flavors of Prudence 1.1, you will need to use JavaScript for your bootstrapping scripts, even if your application code (resources, scriptlets, tasks, etc.) is written in a different language.

To be 100% clear: *all "flavors" supported in Prudence 1.1 are still supported in Prudence 2.0*, and your application code will likely not even have to change. You *only* need (or rather, are recommended) to use JavaScript for bootstrapping.

Upgrading Applications

There are no significant API changes between Prudence 1.1 and Prudence 2.0. However, the bootstrapping and configuration has been completely overhauled. You will likely need to take a few minutes to rewrite your settings.js, routing.js, etc. Here is a step-by-step checklist:

1. Start with a new application based on the default template.
   (a) Rename old application (add "-old"), for example: "myapp-old"
   (b) Use the "prudence" Sincerity tool to create a new application for your application name:
   ```
   sincerity prudence create myapp
   ```
2. Copy over individual settings from settings.js, using the new manual (page 70) to find equivalences.
3. Redo your routing.js, using the new manual (page 117) to find equivalences. Prudence 2.0 uses a far more powerful and clearer routing configuration.
4. Rename "/resources/" files to add a ".m." pre-extension (they are now called "manual resources"). Under Unix-like operation systems, you can rename the all files in the tree via a Perl expression using something like this:
   ```
   find . -name "*.js" -exec rename -v 's/\.js$/\.m.js/i' {} \;
   ```
5. Rename "/web/dynamic/" files to add a ".t." pre-extension (they are now called "template resources"). Under Unix-like operation systems, you can rename the all files in the tree via a Perl expression using something like this:
   ```
   find . -name "*.html" -exec rename -v 's/\.html$/\.t.html/i' {} \;
   ```
6. Merge "/web/dynamic/" and "/web/static/" into "/resources/".
7. Move "/web/fragments/" to "/libraries/includes/".

Part III
Articles

The Case for REST

There's a lot of buzz about REST, but also a lot confusion about what it is and what it's good for. This essay attempts to convey REST's simple essence.

Let's start, then, not at REST, but at an attempt to create a new architecture for building scalable applications. Our goals are for it to be minimal, straightforward, and still have enough features to be productive. We want to learn some lessons from the failures of other, more elaborate and complicated architectures.

Let's call ours a "resource-oriented architecture."

Resources

Our base unit is a "resource," which, like an object in object-oriented architectures, encapsulates data with some functionality. However, we've learned from object-orientation that implementing arbitrary interfaces is a recipe for complexity: proxy generation, support for arbitrary types, marshaling, etc. And then we would need middleware to do all that heavy lifting for us. So, let's instead keep it simple and define a limited, unified interface that would be just useful enough.

From our experience with relational databases, we've learned that tremendous power can be found in "CRUD": Create, Read, Update and Delete. If we support just these operations, our resources will already be very powerful, enjoying the accumulated wisdom and design patterns from the database world.

Identifiers

Let's start with a way of uniquely identifying our resources. We'll define a name-based address space where our resources live. Each resource is "attached" to one or more addresses. We'll allow for "/" as a customary separator to allow for hierarchical addressing schemes. For example:

```
/animal/dog/3/
/animal/cat/12/image/
/animal/cat/12/image/large/
/animal/cat/12/specs/
```

In the above, we've allowed for different kinds of animals, a way of referencing individual animals, and a way of referencing specific aspects of these animals.

Let's now go over CRUD operations in increasing order of complexity.

Delete

"Delete" is the most trivial operation. After sending "delete" to an identifier, we expect it to not exist anymore. Whether sub-resources in our hierarchy can exist or not, we'll leave up to individual implementations. For example, deleting "/animal/cat/12/image" may or may not delete "/animal/cat/12/image/large".

Note that we don't care about atomicity here, because we don't expect anything to happen after our "delete" operation. A million changes can happen to our cat before our command is processed, but they're all forgotten after "delete." (See "update," below, for a small caveat.)

Read

"Read" is a bit more complicated than "delete." Since our resource might be changed by other clients, too, we want to make sure that there's some kind of way to mark which version we are reading. This will allow us to avoid unnecessary reads if there hasn't been any change.

Thus, we'll need our resource-oriented architecture to support some kind of version tagging feature.

Update

The problem with "update" is that it always references a certain version that we have "read" before. In some cases, though not all, we need some way to make sure that the data we expect to be there hasn't changed since we've last "read" it. Let's call this a "conditional update." (In databases, this is called a "compare-and-set" atomic operation.)

Actually, we've oversimplified our earlier definition of "delete." In some cases, we'd want a "conditional delete" to depend on certain expectations about the data. We might not want the resource deleted in some cases.

We'll need our resource-oriented architecture to support a general "conditional" operation feature.

Create

This is our most complex operation. Our first problem is that our identifier might not exist yet, or might already be attached to a resource. One approach could be to try identifiers in sequence:

```
Create: /animal/cat/13/ -> Error, already exists
Create: /animal/cat/14/ -> Error, already exists
Create: /animal/cat/15/ -> Error, already exists
...
Create: /animal/cat/302041/ -> Success!
```

Obviously, this is not a scalable solution. Another approach could be to have a helper resource which provides us with the necessary ID:

```
Read: /animal/cat/next/ -> 14
Create: /animal/cat/14/ -> Oops, someone else beat us to 14!
Read: /animal/cat/next/ -> 15
Create: /animal/cat/15/ -> Success!
```

Of course, we can also have "/animal/cat/next/" return unique IDs (such as GUIDs) to avoid duplications. If we never create our cat, they will be wasted, though. The main problem with this approach is that it requires two calls per creation: a "read," and then a "create." We can handle this in one call by allowing for "partial" creation, a "create" linked with an intrinsic "read":

```
Create: /animal/cat/ -> We send the data for the cat without the ID, and get back
    the same cat with an ID
```

Other solutions exist, too. The point of this discussion is to show you that "create" is not trivial, but also that solutions to "create" already exist within the resource-oriented architecture we've defined. "Create," though programmatically complex, does not require any additional architectural features.

Aggregate Resources

At first glance, handling the problem of getting lots of resources at the same time, thus saving on the number of calls, can trivially be handled by the features we've listed so far. A common solution is to define a "plural" version of the "singular" resource:

```
/animal/cats/
```

A "read" would give us all cats. But what if there are ten million cats? We can support paging. Again, we have a solution within our current feature set, using identifiers for each subset of cats:

```
/animal/cats/100/200/
```

We can define the above to return no more than 100 cats: from the 100th, to the 200th. There's a slight problem in this solution: the burden is on whatever component in our system handles mapping identifiers to resources. This is not terrible, but if we want our system to be more generic, it could help if things like "100 to 200" could be handled by our resource more directly. For convenience, let's implement a simple parameter system for all commands:

```
Read(100, 200): /animal/cats/
```

In the above, our mapping component only needs to know about "/animal/cats". The dumber our mapping component is, the easier it is to implement.

Formats

The problem of supporting multiple formats seems similar, at first glance, to that of aggregate resources. Again, we could potentially solve it with command parameters:

```
Read(UTF-8, Russian): /animal/cat/13/
```

This would give us a Russian, Unicode UTF-8 encoded version of our cat. Looks good, except that there is a potential problem: the client might prefer certain formats, but actually be able to handle others. It's more a matter of preference than any precision. Of course, we can have another resource where all available formats are listed, but this would require an extra call, and also introduce the problem of atomicity—what if the cat changes between these calls? A better solution would be to have the client associate certain preferences per command, have our resource emit its capabilities, with the mapping component in between "negotiating" these two lists. This "negotiation" is a rather simple algorithm to choose the best mutually preferable format.

This would be a simple feature to add to our resource-oriented architecture, which could greatly help to decouple its support for multiple formats from its addressing scheme.

Shared State

Shared state between the client and server is very useful for managing sessions and implementing basic security. Of course, it's quite easy to abuse shared state, too, by treating it as a cache for data. We don't want to encourage that. Instead, we just want a very simple shared state system.

We'll allow for this by attaching small, named, shared state objects to every request and response to a command. Nothing fancy or elaborate. There is a potential security breach here, so we have to trust that all components along the way honor the relationship between client and server, and don't allow other servers access to our shared state.

Summary of Features

So, what do we need?

We need a way to map identifiers to resources. We need support for the four CRUD operations. We need support for "conditional" updates and deletes. We need all operations to support "parameters." We need "negotiation" of formats. And, we need a simple shared state attachment feature.

This list is very easy to implement. It requires very little computing power, and no support for generic, arbitrary additions.

Transactions... Not!

Before we go on, it's worth mentioning one important feature which we did not require: transactions. Transactions are optional, and sometimes core features in many databases and distributed object systems. They can be extremely powerful, as they allow atomicity across an arbitrary number of commands. They are also, however, heavy to implement, as they require considerable shared state between client and server. Powerful as they are, it is possible to live without them. For example, we can implement complex atomicity schemes ourselves within a single resource. This puts some burden on us, but it does remove the heavy burden of supporting arbitrary transactions from our architecture. With some small reluctance, then, we'll do without transactions.

Let's Do It!

OK, so now we know what we need, let's go ahead and implement the infrastructure of components to handle our requirements. All we need is stacks for all supported clients, backend stacks for all our potential server platforms, middleware components to handle all the identifier routing, content negotiation, caching of data...

...And thousands of man hours to develop, test, deploy, and integrate. Like any large-scale, enterprise architecture, even trivial requirements have to jump through the usual hoops set up by the sheer scale of the task. Behind every great architecture are the nuts and bolts of the infrastructure.

Wouldn't it be great if the infrastructure already existed?

The Punchline

Well, duh. *All* the requirements for our resource-oriented architecture are already supported by HTTP:

Our resource identifiers are URLs. The CRUD operations are in the four HTTP verbs: PUT, GET, POST and DELETE. "Conditional" and "negotiated" modes are handled by headers, as are "cookies" for shared state. Version stamps are e-tags and timestamps. Command parameters are query matrices appended to URLs. It's all there.

Most importantly, the infrastructure for HTTP is already fully *deployed* world-wide. TCP/IP stacks are part of practically every operating system; wiring, switching and routing are part and parcel; HTTP gateways, firewalls, load balancers, proxies, caches, filters, etc., are stable consumer components; certificate authorities, national laws, international agreements are already in place to support the complex inter-business interaction. Best of all, this available infrastructure is successfully maintained, with minimal down-time, by highly-skilled independent technicians, organizations and component vendors across the world.

It's important to note a dependency and possible limitation of HTTP: it is bound to TCP/IP. Indeed, all identifiers are URLs: Uniform Resource Locators. In URLs, the first segment is reserved for the domain, either an IP address or a domain name translatable to an IP address. Compare this with the more general URIs (Uniform Resource Identifiers), which do not have this requirement. Though we'll often be tied to HTTP in REST, you'll see the literature attempting, at least, to be more generic. There are definitely use cases for non-HTTP, and even non-TCP/IP addressing schemes. In Prudence, it's possible to address internal resources with URIs *that are not URLs*; see internal APIs (page **??**).

It's All About Infrastructure

The most important lesson to take from this exercise is the importance of infrastructure, something easily forgotten when planning architecture in ideal, abstract terms. This is why, I believe, Roy Fielding named Chapter 5 of his 2000 dissertation "Representational State Transfer (REST)" rather than, say, "resource-oriented architecture," as we have here. Fielding, one of the authors of the HTTP protocol, was intimately familiar with its deployment challenges, and the name "REST" is intended to point out the key characteristic of its infrastructure: HTTP and similar protocols are designed for transferring lightly annotated data representations, nothing more. "Resources" are merely logical encapsulations of these representations, depending on a contract between client and server. The infrastructure does not, in itself, do anything in particular to maintain, say, a sensible hierarchy of addresses, abitrary atomicity of CRUD operations, etc. That's up to your implementation. But, representational state transfer—REST—is the mundane, underlying magic that makes it all possible.

To come back to where we started: a resource-oriented architecture requires a REST infrastructure. In practice, the two terms become interchangeable.

The principles of resource-orientation can and are applied in many systems. The word wide web, of course, with its ecology of web browsers, web servers, certificate authorities, etc., is the most obvious model. But other core Internet systems, such as email (SMTP, POP, IMAP), file transfer (FTP, WebDAV) also implement some subset of REST. Your application can do this, too, and enjoy the same potential for scalability as these global, open implementations.

Does REST Scale?

Part of the buzz about REST is that it's an inherently scalable architecture. This is true, but perhaps not in the way that you think.

Consider that there are two uses of the word "scalable":

First, it's **the ability to respond to a growing number of user requests without degradation in response time**, by "simply" adding hardware (horizontal scaling) or replacing it with more powerful hardware (vertical scaling). This is the aspect of scalability that engineers care about. The simple answer is that REST can help, but it doesn't stand out. SOAP, for example, can also do it pretty well. REST aficionados sometimes point out that REST is "stateless," or "session-less," both characteristics that would definitely help scale. But, this is misleading. Protocols might be stateless, but architectures built on top of them don't have to be. For example, we've specifically talked about sessions here, and many web frameworks manage sessions via cookies. On the other hand, you can easily make poorly scalable REST. The bottom line is that there's nothing in REST that guarantees scalability in *this* respect. Indeed, engineers coming to REST due to this false lure end up wondering what the big deal is. We wrote a whole article for Scaling Tips (page 159), which is indeed not specifically about REST.

The second use of "scalability" comes from the realm of enterprise and project management. It's **the ability of your project to grow in complexity without degradation in your ability to manage it**. And that's REST's beauty—you already have the infrastructure, which is the hardest thing to scale in a project. You don't need to deploy client stacks. You don't need to create and update proxy objects for five different programming languages used in your enterprise. You don't need to deploy incompatible middleware by three different vendors and spend weeks trying to force them to play well together. Why would engineers care about REST? Precisely because they don't have to: they can focus on application engineering, rather than get bogged down by infrastructure management.

That said, a "resource-oriented architecture" as we defined here is not a bad start for—engineering-wise—scalable systems. Keep your extras lightweight, minimize or eliminate shared state, and encapsulate your resources according to use cases, and you won't, at least immediately, create any obstacles to scaling.

Prudence

Convinced? The best way to understand REST is to experiment with it. You've come to the right place. Start with the tutorial (page 6), and feel free to skip around the documentation and try things out for yourself. We're sure that you'll find it easy, fun, and powerful enough for you to create large-scale applications that take full advantage of the inherently scalable infrastructure of REST.

URI-space Architecture

REST does not standardize URI-spaces, and indeed has little to say about URI design. However, it does *imply* a preference for certain architectural principles. We go over much of the impetus in The Case for REST article (page 152), and suggest you start there.

It's a good idea to think very carefully about your URI-space: a good design will in turn help you define well-encapsulated RESTful resources. Below are some topics to consider.

Nouns vs. Verbs

It's useful to think of URIs as syntactic *nouns*, a grammatical counterpart to HTTP's *verbs*. To put it simply: make sure that you do not include verbs in your URIs. Examples:

- Good: "/service/{id}/status/"
- Bad: "/service/{id}/start/", "/service/{id}/stop/"

What is wrong with verbs in URIs?

One potential problem is clarity. Which HTTP verb should be used on a verb URI? Do you need to POST, PUT or DELETE to "/service/{id}/stop/" in order to stop the service? Of course, you can support all and document this, but it won't be immediately obvious to the user.

A second potential problem is that you need to keep increasing the size of your URI-space the more actions of this sort you want to support. This means more files, more classes, and generally more code. Handling these operations *inside* a single resource would just mean a simple "if" or "switch" statement and an extra method.

A third, more serious potential problem is idempotency. The idempotent verbs PUT and DELETE may be optimized by the HTTP infrastructure (for example, a smart load balancer) such that requests arrive more than once: this is allowed by the very definition of idempotency. However, your intended operations *may not be* semantically idempotent. For example, if a "stop" is sent to an already-stopped service, it may return an "already stopped" 500 error. In this case, if the infrastructure indeed allows for two "stop" commands to come through, then the user may get an error even though the operation succeeded for the first "stop." There's an easy way around this: simply allow *only* POST, the non-idempotent verb, for all such operations. The infrastructure would never allow more than request to come through per POST. However, if you enforce the use of POST, you will lose the ability of the infrastructure to optimize for non-idempotency. POST is the least scalable HTTP verb.

The bottom line is that if you standardize on only using nouns for your URIs, you will avoid many of these semantic pitfalls.

Also: beware of gerunds! A URI such as "/service/{id}/stopping/" is may be a noun, but allows for some verb-related problems to creep in.

Do You Really Need REST?

In the above section, it was suggested that you prefer nouns to verbs. However, this preference may be too constraining for your project. Your application may be very command-oriented, such that you will end up with a very small set of "noun" URIs that need to support a large amount of commands, meaning less clarity and more cluttered code.

REST's best feature it it's tiny set of tightly defined verbs: GET, POST, PUT, DELETE. The entire infrastructure is highly optimized around them: load balancers, caches, browsers, gateways, etc., all know how best to handle each of these for maximum scalability and reliability. But, it's entirely possible that your needs cannot be easily satisfied by just four verbs.

And that's OK. REST is not always the best solution for APIs.

Instead, take a look at RPC (Remote Procedure Call) mechanisms. The Diligence framework, based on Prudence, provides easy and robust support for both JSON-RPC and XML-RPC in its RPC Service as well as Ext Direct in its Sencha Integration, allowing you to hook a JavaScript function on the server directly to a URI. In terms of HTTP, these protocols all use HTTP POST, and do not leverage the HTTP infrastructure as well as a more fully RESTful API. But, one size does not fit all, and an RPC-based solution may prove a better match for your project.

It's also perfectly possible to use *both* REST and RPC in your project. Use each approach where it is most appropriate.

Hierarchical URIs

It's entirely a matter of convention that the use of "/" in URIs implies hierarchy. Historically, the convention was likely imported from filesystem paths, where a name before a "/" signifies a directory rather than a file.

This convention is useful because it's very familiar to users, but beyond that it implies a few semantic properties that can add clarity and power to your resource design. There are two possible principles you may consider:

1. A descendant resource *belongs to* its ancestor, such that resources have cascading relationships in the hierarchy. This implies two rules:
   (a) Operations on a resource *may* affect descendants. This rule is most obvious when applied to the DELETE verb: for example, if you delete "/user/{id}/", then it is expected that the resources at "/user/{id}/profile/" and "/user/{id}/preferences/" also be deleted. A PUT, too, would also affect the descendant resources.
   (b) Operations on a resource *should not* affect ancestors. In other words, a descendant's state is isolated from its ancestors. For example, if I send a POST to "/user/{id}/profile/", the representation at "/user/{id}/" should remain unaltered.
2. A descendant resource *belongs to* its ancestor and also represents *an aspect of* its ancestor, such that operations on a resource can be fine-tuned to particular aspects of it. This implies three rules:
   (a) Descendant representations *are included* in ancestor representations. For example, a GET on "/service/{id}/" would include information about the status that you would see if you GET on "/service/{id}/status/". The latter URI makes it easier for the client to direct operations at the status aspect.
   (b) Operations on a resource *may* affect descendants. See above.
   (c) Operations on a resource *will* affect ancestors. This is the *opposite* of the above: the descendant's state is *not isolated* from its ancestors. For example, a POST to "/service/{id}/status/" would surely also affect "/service/{id}/", which includes the status.

You can see from the difference between rule 1.b and 2.c. that it's important to carefully define the *nature* of your hierarchical relationships. Unlike filesystem directory hierarchies, in a URI-space there is no single standard or obvious interpretation of what of a hierarchical relationship would mean. But unless you clarify it for yourself, it cannot be clear to your users.

Formats Are Not Aspects

A format should not be considered "an aspect" in the sense used in principle 2. For example, "/service/{id}/html/" would not be a good way to support an HTML format for "/service/{id}/". The reason is that you would be allowing for more than one URI for the same encapsulated resource, creating confusion for users. For example, it's not immediately clear what would happen if they DELETE "/service/{id}/html/". Would that just remove the ability to represent the service as HTML? Or delete the service itself?

Supporting multiple formats is best handled with content negotiation, within the REST architecture. If further formatting is required, URI query parameters can be used. For example: "/service/{id}/?indent=2" might return a JSON representation with 2-space indentation.

Plural vs. Singular

You'll see RESTful implementations that use either convention. The advantage of using the singular form is that you have less addresses, and what some people would call a more elegant scheme:

```
/animal/cat/12/ -> Just one cat
/animal/cat/ -> All cats
```

Why add another URL format when a single one is enough to do the work? One reason is that you can help the client avoid potential errors. For example, the client probably uses a variable to hold the ID of the cat and then constructs the URL dynamically. But, what if the client forgets to check for empty IDs? It might then construct a URL in the form "/animal/cat//" which would then successfully access *all* cats. This can cause unintended consequences and be difficult to debug. If, however, we used this scheme:

```
/animal/cat/12/ -> Just one cat
/animal/cats/ -> All cats
```

...then the form "/animal/cat//" would route to our singular cat resource, which would indeed not find the "empty" cat and return the expected, debuggable 404 error.

From this example, we can extract a good rule of thumb: *clearly separate URI templates by usage*, so that mistakes cannot happen. More URI types means more debuggability.

Scaling Tips

Scalability is the ability to respond to a growing number of user requests without degradation in response time. Two variables influence it: 1) your total number of threads and 2) the time it takes each thread to process a request. Increasing the number of threads seems straightforward: you can keep adding more machines behind load balancers. However, the two variables are tied, as there are diminishing returns and even reversals: beyond a certain point, time per request can actually grow longer as you add threads and machines.

Let's ignore the first variable here, because the challenge of getting more machines is mostly financial. It's the second that you can do something about as an engineer.

If you want your application to handle many concurrent users, then you're fighting this fact: a request will get queued in the best case or discarded in the worst case if there is no thread available to serve it. Your challenge is to make sure that a thread is always available. And it's not easy, as you'll find out as you read through this article. Minimizing the time per request becomes an architectural challenge that encompasses the entire structure of your application

Performance Does Not Equal Scalability

Performance does not equal scalability. Performance does not equal scalability. Performance does not equal scalability.

Get it? Performance does not equal scalability.

This is an important mantra for two reasons:

1. Performant Can Mean Less Scalable

Optimizing for performance can adversely affect your scalability. The reason is contextual: when you optimize for performance, you often work in an isolated context, specifically so you can accurately measure response times and fine-tune them. For example, making sure that a specific SQL query is fast would involve just running that query. A full-blown experiment involving millions of users doing various operations on your application would make it very hard to accurately measure and optimize the query. Unfortunately, by working in an isolated context you cannot easily see how your efforts would affect other parts of an application. To do so would require a lot of experience and imagination. To continue our example, in order to optimize your one SQL query you might create an index. That index might need to be synchronized with many servers in your cluster. And that synchronization overhead, in turn, could seriously affect your ability to scale. Congratulations! You've made one query run fast in a situation that never happens in real life, and you've brought your web site to a halt.

One way to try to get around this is to fake scale. Tools such as JMeter, Siege and ApacheBench can create "load." They also create unfounded confidence in engineers. If you simulate 10,000 users bombarding a single web page, then you're, as before, working in an isolated context. All you've done is add concurrency to your performance optimization measurements. Your application pathways might work optimally in these situations, but this might very well be due to the fact that the system is not doing anything else. Add those "other" operations in, and you might get worse site capacity than you did before "optimizing."

(The folk who make Jetty have an interesting discussion of this.)

2. Wasted Effort

Even if you don't adversely affect your scalability through optimizing for performance, you might be making no gains, either. No harm done? Well, plenty of harm, maybe. Optimizing for performance might waste a lot of development time and money. This effort would be better spent on work that could actually help scalability.

And, perhaps more seriously, it demonstrates a fundamental misunderstanding of the problem field. If you don't know what your problems are, you'll never be able to solve them.

Pitfalls

Study the problem field carefully. Understand the challenges and potential pitfalls. You don't have to apply every single scalability strategy up-front, but at least make sure you're not making a fatal mistake, such as binding yourself strongly to a technology or product with poor scalability. A bad decision can mean that when you need to scale up in the future, no amount of money and engineering effort would be able to save you before you lose customers and tarnish your brand.

Moreover, be very careful of blindly applying "successful" strategies used and recommended by others to your product. What worked for them might not work for you. In fact, there's a chance that their strategy doesn't even work for them, and they just think it did because of a combination of seemingly unrelated factors. The realm of web scalability is still young, full of guesswork, intuition and magical thinking. Even the experts are often making it up as they're going along.

Generally, be very suspicious of products or technologies being touted as "faster" than others. *"Fast" doesn't say anything about the ability to scale.* Is a certain database engine "fast"? That's important for certain applications, no doubt. But maybe the database is missing important clustering features, such that it would be a poor choice for scalable applications. Does a certain programming language execute faster than another? That's great if you're doing video compression, but speed of execution might not have a noticeable effect on scalability. Web applications mostly do I/O, not computation. The same web application might have very similar performance characteristics whether it's written in C++ or PHP.

Moreover, if the faster language is difficult to work with, has poor debugging tools, limited integration with web technologies, then it would slow down your work and your ability to scale.

> Speed of execution can actually help scalability in its financial aspect: If your application servers are constantly at maximum CPU load, then a faster execution platform would let you cram more web threads into each server. This could help you reduce costs. For example, see Facebook's HipHop: they saved millions by translating their PHP code to C. Because Prudence is built on the fast JVM platform, you're in good hands in this respect. Note, however, that there's a potential pitfall to high performance: more threads per machine would also mean more RAM requirements per machine, which also costs money. Crunch the numbers and make sure that you're actually saving money by increasing performance. Once again, performance does not equal scalability.

That last point about programming languages is worth some elaboration. Beyond how well your chosen technologies perform, it's important to evaluate them in terms to how easy they are to manage. Large web sites are large projects, involving large teams of people and large amounts of money. That's difficult enough to coordinate. You want the technology to present you with as few extra managerial challenges as possible.

Beware especially of languages and platforms described as "agile," as if they somehow embody the spirit of the popular Agile Manifesto. Often, "agile" seems to emphasize the following features: forgiveness for syntax slips, light or no type checking, automatic memory management and automatic concurrency—all features that seem to speed up development, but could just as well be used for sloppy, error-prone, hard-to-debug, and hard-to-fix code, slowing down development in the long run. If you're reading this article, then your goal is likely not to create a quick demo, but a stable application with a long, evolving life span.

Ignore the buzzwords ("productivity", "fast"), and instead make sure you're choosing technology that you can control, instead of technology that will control you.

We discuss this topic some more in "The Case for REST" (page 152). By building on the existing web infrastructure, Prudence can make large Internet projects easier to manage.

Analysis

Be especially careful of applying a solution before you know if you even have a problem.

How to identify your scalability bottlenecks? You can create simulations and measurements of scalability rather than performance. You need to model actual user behavior patterns, allow for a diversity of such behaviors to happen concurrently, and replicate this diversity on a massive scale.

Creating such a simulation is a difficult and expensive, as is monitoring and interpreting the results and identifying potential bottlenecks. This is the main reason for the lack of good data and good judgment about how to scale. Most of what we know comes from tweaking real live web sites, which either comes at the expense of user experience, or allows for very limited experimentation. Your best bet is to hire a team who's already been through this before.

Optimizing for Scalability

In summary, your architectural objective is to increase concurrency, not necessarily performance. Optimizing for concurrency means breaking up tasks into as many pieces as possible, and possibly even breaking requests into smaller pieces. We'll cover numerous strategies here, from frontend to backend. Meanwhile, feel free to frame these inspirational slogans on your wall:

> Requests are hot potatoes: Pass them on!

And:

> It's better to have many short requests than one long one.

Caching

Retrieving from a cache can be orders of magnitude faster than dynamically processing a request. It's your most powerful tool for increasing concurrency.

Caching, however, is only effective is there's something in the cache. It's pointless to cache fragments that appear only to one user on only one page that they won't return to. On the other hand, there may very well be fragments on the page that will recur often. If you design your page carefully to allow for fragmentation, you will reap the benefits of fine-grained caching. Remember, though, that the outermost fragment's expiration defines the expiration of the included fragments. It's thus good practice to define no caching on the page itself, and only to cache fragments.

In your plan for fine-grained caching with Prudence, take special care to isolate those fragments that cannot be cached, and cache everything around them.

Make sure to change the cache key template (page 63) to fit the lowest common denominator: you want as many possible requests to use the already-cached data, rather than generating new data. Note that, by default, Prudence includes the request URI in the cache key. Fragments, though, may very well appear identically in many different URIs. You would thus not want the URI as part of their cache key.

Cache aggressively, but also take cache validation seriously. Make good use of Prudence's cache tags (page 63) to allow you to invalidate portions of the cache that should be updated as data changes. Note, though, that every time you invalidate you will lose caching benefits. If possible, make sure that your cache tags don't cover too many pages. Invalidate only those entries that really need to be invalidated.

(It's sad that many popular web sites do cache validation so poorly. Users have come to expect that sometimes they see wrong, outdated data on a page, sometimes mixed with up-to-date data. The problem is usually solved within minutes, or after a few browser refreshes, but please do strive for a better user experience in your web site!)

If you're using a background task (page 165), you might want to invalidate tagged cache entries when tasks are done. Consider creating a special internal API that lets the task handler call back to your application to do this.

How long should you cache? As long as the user can bear! In a perfect world, of limitless computing resources, all pages would always be generated freshly per request. In a great many cases, however, there is no harm at all if users see some data that's a few hours or a few days old.

> Note that even very small cache durations can make a big difference in application stability. Consider it the maximum throttle for load. For example, a huge sudden peak of user load, or even a denial-of-service (DOS) attack, might overrun your thread pool. However, a cache duration of just 1 second would mean that your page would never be generated more than once every second. You are instantly protected against a destructive scenario.

Cache Warming

Caches work best when they are "warm," meaning that they are full of data ready to be retrieved.

A "cold" cache is not only useless, but it can also lead indirectly to a serious problem. If your site has been optimized for a warm cache, starting from cold could significantly strain your performance, as your application servers struggle to generate all pages and fragments from scratch. Users would be getting slow response times until the cache is significantly warm. Worse, your system could crash under the sudden extra load.

There are two strategies to deal with cold caches. The first is to allow your cache to be persistent, so that if you restart the cache system it retains the same warmth it had before. This happens automatically with database-backed caches (page 163). The second strategy is to deliberately warm up the cache in preparation for user requests.

Consider creating a special external process or processes to do so. Here are some tips:

1. Consider mechanisms to make sure that your warmer does not overload your system or take too much bandwidth from actual users. The best warmers are adaptive, changing their load according to what the servers can handle. Otherwise, consider shutting down your site for a certain amount of time until the cache is sufficiently warm.

2. If the scope is very large, you will have to pick and choose which pages to warm up. In Prudence, this is supported via app.preheat (page 33). You would want to choose only the most popular pages, in which case you might need a system to record and measure popularity. For example, for a blog, it's not enough just to warm up, say, the last two weeks of blog posts, because a blog post from a year ago might be very popular at the moment. Effective warming would require you to find out how many times certain blog posts were hit in the past two weeks. It might make sense to embed this auditing ability into the cache backend itself.

Pre-Filling the Cache

If there are thousands of ways in which users can organize a data view, and each of these views is particular to one user, then it may make little sense to cache them individually, because individual schemes would hardly ever be re-used. You'll just be filling up the cache with useless entries.

Take a closer look, though:

1. It may be that of the thousands of organization schemes only a few are commonly used, so it's worth caching the output of just those.

2. It could be that these schemes are similar enough to each other that you could generate them all in one operation, and save them each separately in the cache. Even if cache entries will barely be used, if they're cheap to create, it still might be worth creating them.

This leads us to an important point:

> Prudence is a "frontend" platform, in that it does not specify which data backend, if at all, you should use. Its cache, however, is general purpose, and you can store in it anything that you can encode as a string.

Let's take as a pre-filling example a tree data structure in which branches can be visually opened and closed. Additionally, according to user permissions different parts of the tree may be hidden. Sounds too complicated to cache all the view combinations? Well, consider that you can trigger, upon any change to the tree data structure, a function that loops through all the different iterations of the tree recursively and saves a view of each of them to the cache. The cache keys can be something like "branch1+.branch2-.branch3+", with "+" and "-" signifying whether the branch is visually open or closed. You can use similar +'s and -'s for permissions, and create views per permission combinations. Later, when users with specific permissions request different views of the tree, no problem: all possibilities were already pre-filled. You might end up having to generate and cache thousands of views at once, but the difference between generating one view and generating thousands of views may be quite small, because the majority of that duration is spent communicating with the database backend.

If generating thousands of views takes too long for the duration of a single request, another option is to generate them on a separate thread. Even if it takes a few minutes to generate all the many, many tree view combinations, it might be OK in your application for views to be a few minutes out-of-date. Consider that the scalability benefits can be very significant: you generate views only *once* for the entire system, while millions of concurrent users do a simple retrieval from the cache.

Caching the Data Backend

Pre-filling the cache can take you very far. It is, however, quite complicated to implement, and can be ineffective if data changes too frequently or if the cache has to constantly be updated. Also, it's hard to scale the pre-filling to *millions* of fragments.

If we go back to our tree example above, the problem was that it was too costly to fetch the entire tree from the database. But what if we cache the tree itself? In that case, it would be very quick to generate any view of the tree on-demand. Instead of caching the view, we'd be caching the data, and achieving the same scalability gains.

Easy, right? So why not cache *all* our data structures? The reason is that it's very difficult to do this correctly beyond trivial examples. Data structures tend to have complex interrelationships (one-to-many, many-to-many, foreign keys, recursive tree structures, graphs, etc.) such that a change in data at one point of the structure may alter various others in particular ways. For example, consider a calendar database, and that you're caching individual days with all their events. Weekly calendar views are then generated on the fly (and quickly) for users according to what kinds of events they want to see in their personal calendars. What happens if a user adds a recurring event that happens every Monday? You'll need to make sure that all Mondays currently cached would be invalidated, which might mean tagging all these as "monday" using Prudence's cache tags. This requires a specific caching strategy for a specific application.

By all means, cache your data structures if you can't easily cache your output, but be aware of the challenge!

Cache Backends

Your cache backend can become a bottleneck to scalability if 1) it can't handle the amount of data you are storing, or 2) it can't respond quickly enough to cache fetching.

Before you start worrying about this, consider that it's a rare problem to have. Even if you are caching millions of pages and fragments, a simple relational-database-backed cache, such as Prudence's SqlCache implementations, could handle this just fine. A key/value table is the most trivial workload for relational databases, and it's also easy to shard (page 167). Relational database are usually very good at caching these tables in their memory and responding optimally to read requests. Prudence even lets you chain caches together to create tiers: an in-process memory cache in front of a SQL cache would ensure that many requests don't even reach the SQL backend.

High concurrency can also be handled very well by this solution. Despite any limits to the number of concurrent connections you can maintain to the database, each request is handled very quickly, and it would require *very* high loads to saturate. The math is straightforward: with a 10ms average retrieval time (very pessimistic!) and a maximum of 10 concurrent database connections (again, pessimistic!) you can handle 1,000 cache hits per second. A real environment would likely provide results orders of magnitude better.

The nice thing about this solution is that it uses the infrastructure you already have: the database.

But, what if you need to handle *millions* of cache hits per second? First, let us congratulate you for your global popularity. Second, there is a simple solution: distributed memory caches. Prudence comes with Hazelcast and support for memcached, which both offer much better scalability than database backends. Because the cache is in memory, you lose the ability to easily persist your cache and keep it warm: restarting your cache nodes will effectively reset them. There are workarounds—for example, parts of the cache can be persisted to a second database-backed cache tier—but this is a significant feature to lose.

> Actually, Hazelcast offers fail-safe, live backups. While it's not quite as permanent as a database, it might be good enough for your needs. And memcached has various plugins that allow for real database persistence, though using them would require you to deal with the scalability challenges of database backends (page 168).

You'll see many web frameworks out there that support a distributed memory cache (usually memcached) and recommend you use it ("it's fast!" they claim, except that it can be slower per request than optimized databases, and that anyway performance does not equal scalability). We'd urge you to consider that advice carefully: keeping your cache warm is a challenge made much easier if you can store it in a persistent backend, and database backends can take you very far in scale without adding a new infrastructure to your deployment. It's good to know, though, that Prudence's support for Hazelcast and memcached is there to help you in case you reach the popularity levels of LiveJournal, Facebook, YouTube, Twitter, etc.

Client-Side Caching

Modern web browsers support client-side caching, a feature meant to improve the user experience and save bandwidth costs. A site that makes good use of client-side caching will appear to work fast for users, and will also help to increase your site's popularity index with search engines.

Optimizing the user experience is not the topic of this article: our job here is to make sure your site doesn't degrade its performance as load increases. However, client-side caching can indirectly help you scale by reducing the number of hits you have to take in order for your application to work.

> Actually, doing a poor job with client-side caching can help you scale: users will hate your site and stop using it—voila, less hits you have to deal with. OK, that was a joke!

Generally, Prudence handles client-side caching automatically. If you cache a page, then headers will be set to ask the client to cache for the same length of time. By default, conditional mode is used: every time the client tries to view a page, it will make a request to make sure that nothing has changed since their last request to the page. In case nothing has changed, no content is returned.

You can also turn on "offline caching" mode, in which the client will avoid even that quick request. Why not enable offline caching by default? Because it involves some risk: if you ask to cache a page for one week, but then find out that you have a mistake in your application, then users will not see any fix you publish until their local cache expires, which can take up to a week! It's important that you you understand the implications before using this mode. See the caching guide (page 65) for a complete discussion.

It's generally safer to apply offline caching to your static resources, such as graphics and other resources. A general custom is to ask the client to cache these "forever" (10 years), and then, if you need to update a file, you simply create a new one with a new URL, and have all your HTML refer to the new version. Because clients cache according to URL, their cached for the old version will simply not be ignored. See static resources guide (page 47). There, you'll also see some more tricks Prudence offers you to help optimize the user experience, such as unifying/minimizing client-side JavaScript and CSS.

Upstream Caching

If you need to quickly scale a web site that has not been designed for caching, a band-aid is available: upstream caches, such as Varnish, NCache and even Squid. For archaic reasons, these are sometimes called "reverse proxy" caches, but they really work more like filters: according to attributes in the user request (URL, cookies, etc.), they decide whether to fetch and send a cached version of the response, or to allow the request to continue to your application servers.

The crucial use case is archaic, too. If you're using an old web framework in which you cannot implement caching logic yourself, or cannot plug in to a good cache backend, then these upstream caches can do it for you.

They are problematic in two ways:

1. Decoupling caching logic from your application means losing many features. For example, invalidating portions of the cache is difficult if not impossible. It's because of upstream caching, indeed, that so many web sites do a poor job at showing up-to-date information.

2. Filtering actually implements a kind of partitioning, but one that is vertical rather than horizontal. In horizontal partitioning, a "switch" decides to send requests to one cluster of servers or another. Within each cluster, you can control capacity and scale. But in vertical partitioning, the "filter" handles requests internally. Not only is the "filter" more complex and vulnerable than a "switch" as a frontend connector to the world, but you've also complicated your ability to control the capacity of the caching layer. It's embedded inside your frontend, rather than being another cluster of servers. We'll delve into backend partitioning (page 166) below.

Unfortunately, there is a use case relevant for newer web frameworks, too: if you've designed your application poorly, and you have many requests that could take a long time to complete, then your thread pools could get saturated when many users are concurrently making those requests. When saturated, you cannot handle even the super-quick cache requests. An upstream cache band-aid could, at least, keep serving its cached pages, even though your application servers are at full capacity. This creates an illusion of scalability: some users will see your web site behaving fine, while others will see it hanging.

The real solution would be to re-factor your application so that it does not have long requests, guaranteeing that you're never too saturated to handle tiny requests. Below are tips on how to do this.

Dealing with Lengthy Requests

One size does not fit all: you will want to use different strategies to deal with different kinds of tasks.

Deferrable Tasks

Deferrable tasks are tasks that can be resolved later, without impeding on the user's ability to continue using the application.

If the deferrable task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service. Prudence's background tasks (page 102) implementation is a great solution for deferrable tasks, as it lets you run tasks on other threads or even distribute them in a Hazelcast cluster.

Deferring tasks does present a challenge to the user experience: What do you do if the task fails and the user needs to know about it? One solution can be to send a warning email or other kind of message to the user. Another solution could be to have your client constantly poll in the background (via "AJAX") to see if there are any error messages, which in turn might require you to keep a queue of such error messages per user.

Before you decide on deferring a task, think carefully of the user experience: for example, users might be constantly refreshing a web page waiting to see the results of their operation. Perhaps the task you thought you can defer should actually be considered necessary (see below)?

Necessary Tasks

Necessary tasks are tasks that must be resolved before the user can continue using the application.

If the necessary task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service and return a "please wait" page to the user.

It would be nice to add a progress bar or some other kind of estimation of how long it would take for the task to be done, with a maximum duration set after which the task should be considered to have failed. The client would poll until the task status is marked "done," after which they would be redirected back to the application flow. Each polling request sent by the client could likely be processed very quickly, so this this strategy effectively breaks the task into many small requests ("It's better to have many short requests than one long one").

Prudence's background tasks (page 102) implementation is a good start for creating such a mechanism: however, it would be up to you to create a "please wait" mechanism, as well as a way to track the tasks' progress and deal with failure.

Implementing such a handling service is not trivial. It adds a new component to your architecture, one that also has to be made to scale. One can also argue that it adversely affects user experience by adding overhead, delaying the time it takes for the task to complete. The bottom line, though, is you're vastly increasing concurrency and your ability to scale. And, you're improving the user experience in one respect: they would get a feedback on what's going on rather than having their browsers spin, waiting for their requests to complete.

File Uploads

These are potentially very long requests that you cannot break into smaller tasks, because they depend entirely on the client. As such, they present a unique challenge to scalability.

Fortunately, Prudence handles client requests via non-blocking I/O, meaning that large file uploads will not hold on to a single thread for the duration of the upload. See accepting uploads (page 57).

Unfortunately, many concurrent uploads will still saturate your threads. If your application relies on frequent file uploads, you are advised to handle such requests on separate Prudence instances, so that uploads won't stop your application from handling other web requests. You may also consider using a third-party service specializing in file storage and web uploads.

Asynchronous Request Processing

Having the client poll until a task is completed lets you break up a task into multiple requests and increase concurrency. Another strategy is to break an *individual request* into pieces. While you're processing the request and preparing the response, you can free the web thread to handle other requests. When you're ready to deliver content, you raise a signal, and the next available web thread takes care of sending your response to the client. You can continue doing this indefinitely until the response is complete. From the client's perspective it's a single request: a web browser, for example, would spin until the request was completed.

You might be adding some extra time overhead for the thread-switching on your end, but the benefits for scalability are obvious: you are increasing concurrency by shortening the time you are holding on to web threads.

For web services that deliver heavy content, such as images, video, audio, it's absolutely necessary. Without it, a single user could tie up a thread for minutes, if not hours. You would still get degraded performance if you have more concurrent users than you have threads, but at least degradation will be shared among users. Without asynchronous processing, each user would tie up one thread, and when that finite resource is used up, more users won't be able to access your service.

Even for lightweight content such as HTML web pages, asynchronous processing can be a good tactic for increasing concurrency. For example, if you need to fetch data from a backend with non-deterministic response time, it's best to free the web thread until you actually have content available for the response.

It's not a good idea to do this for every page. While it's better to have many short requests instead of one long one, it's obviously better to have one short request rather than many short ones. Which web requests are good candidates for asynchronous processing?

1. Requests for which processing is made of independent operations. (They'll likely be required to work in sequence, but if they can be processed in parallel, even better!)

2. Requests that must access backend services with non-deterministic response times.

And, even for #2, if the service can take a *very* long time to respond, consider that it might be better to queue the task on a task handler and give proper feedback to the user.

And so, after this lengthy discussion, it turns out that there aren't that many places where asynchronous processing can help you scale. Caching is far more useful.

As of Prudence 2.0, there is no support for asynchronous processing. However, it is planned for a future release, depending on proper support being included in Restlet.

Backend Partitioning

You can keep adding more nodes behind a load balancer insofar as each request does not have to access shared state. Useful web applications, however, are likely data-driven, requiring considerable state.

If the challenge in handling web requests is cutting down the length of request, then that of backends is the struggle against degraded performance as you add new nodes to your database cluster. These nodes have to synchronize their state with each other, and that synchronization overhead increases exponentially. There's a definite point of diminishing returns.

The backend is one place where high-performance hardware can help. Ten expensive, powerful machines might be equal in total power to forty cheap machines, but they require a quarter of the synchronization overhead, giving you more elbow room to scale up. Fewer nodes means better scalablity.

But CPUs can only take you so far.

Partitioning is as useful to backend scaling as caching is to web request scaling. Rather than having one big cluster of identical nodes, you would have several smaller, independent clusters. This lets you add nodes to each cluster without spreading synchronization overhead everywhere. The more partitions you can create, the better you'll be able to scale.

Partitioning can happen in various components of your application, such as application servers, the caching system, task queues, etc. However, it is most effective, and most complicated to implement, for databases. Our discussion will thus focus on relational (SQL) databases. Other systems would likely require simpler subsets of these strategies.

Reads vs. Writes

This simple partitioning scheme greatly reduces synchronization overhead. Read-only servers will never send data to the writable servers. Also, knowing that they don't have to handle writes means you can optimize their configurations for aggressive caching.

(In fact, some database synchronization systems will only let you create this kind of cluster, providing you with one "master" writable node and several read-only "slaves." They force you to partition!)

Another nice thing about read/write partitioning is that you can easily add it to all the other strategies. Any cluster can thus be divided into two.

Of course, for web services that are heavily balanced towards writes, this is not an effective strategy. For example, if you are implementing an auditing service that is constantly being bombarded by incoming data, but is only queried once in a while, then an extra read-only node won't help you scale.

Note that one feature you lose is the ability to have a transaction in which a write *might* happen, because a transaction cannot contain both a read-only node and a write-only node. If you must have atomicity, you will have to do your transaction on the writable cluster, or have two transactions: one to lookup and see if you need to change the data, and the second to perform the change—while first checking again that data didn't change since the previous transaction. Too much of this obviously lessens the effectiveness of read/write partitioning.

By Feature

The most obvious and effective partitioning scheme is by feature. Your site might offer different kinds of services that are functionally independent of each other, even though they are displayed to users as united. Behind the scenes, each feature uses a different set of tables. The rule of thumb is trivial: if you can put the tables in separate databases, then you can put these databases in separate clusters.

One concern in feature-based partitioning is that there are a few tables that still need to be shared. For example, even though the features are separate, they all depend on user settings that are stored in one table.

The good news is that it can be cheap to synchronize just this one table between all clusters. Especially if this table doesn't change often—how often do you get new users signing up for your service?—then synchronization overhead will be minimal.

If your database system doesn't let you synchronize individual tables, then you can do it in your code by writing to all clusters at the same time.

Partitioning by feature is terrific in that it lets you partition other parts of the stack, too. For example, you can also use a different set of web servers for each feature.

Also consider that some features might be candidates for using a "NoSQL" database (page 168). Choose the best backend per feature.

By Section

Another kind of partitioning is sometimes called "sharding." It involves splitting up tables into sections that can be placed in different databases. Some databases support sharding as part of their synchronization strategy, but you can also implement it in your code. The great thing about sharding is that it lets you create as many shards (and clusters) as you want. It's the key to the truly large scale.

Unfortunately, like partitioning by feature, sharding is not always possible. You need to also shard all related tables, so that queries can be self-contained within each shard. It's thus most appropriate for one-to-many data hierarchies. For example, if your application is a blog that supports comments, then you put some blogs and their comments on one shard, and others in another shard. However, if, say, you have a feature where blog posts can refer to other arbitrary blog posts, then querying for those would have to cross shard boundaries.

The best way to see where sharding is possible is to draw a diagram of your table relationships. Places in the diagram which look like individual trees—trunks spreading out into branches and twigs—are good candidates for sharding.

How to decide which data goes in which shard?

Sometimes the best strategy is arbitrary. For example, put all the even-numbered IDs in one shard, and the odd-numbered ones in another. This allows for straightforward growth because you can just switch it to division by three if you want three shards.

Another strategy might seem obvious: If you're running a site which shows different sets of data to different users, then why not implement it as essentially separate sites? For example, a social networking site strictly organized around individual cities could have separate database clusters per city.

A "region" can be geographical, but also topical. For example, a site hosting dance-related discussion forums might have one cluster for ballet and one for tango. A "region" can also refer to user types. For example, your social networking site could be partitioned according to age groups.

The only limitation is queries. You can still let users access profiles in other regions, but cross-regional relational queries won't be possible. Depending on what your application does, this could be a reasonable solution.

A great side-benefit to geographical partitioning is that you can host your servers at data centers within the geographical location, leading to better user experiences. Regional partitioning is useful even for "NoSQL" databases.

Coding Tips for Partitioning

If you organize your code well, it would be very easy to implement partitioning. You simply assign different database operations to use different connection pools. If it's by feature, then you can hard code it for those features. If it's sharding, then you add a switch before each operation telling it which connection pool to use.

For example:

```
def get_blogger_profile(user_id):
    connection = blogger_pool.get_connection()
    ...
    connection.close()

def get_blog_post_and_comments(blog_post_id):
    shard_id = object.id % 3
    connection = blog_pools[shard_id].get_connection()
    ...
    connection.close()
```

Unfortunately, some programming practices make such an effective, clean organization difficult.

Some developers prefer to use ORMs (object-relational mappers) rather than access the database directly. Many ORMs do not easily allow for partitioning, either because they support only a single database connection pool, or because they don't allow your objects to be easily shared between connections.

For example, your logic might require you to retrieve an "object" from the database, and only then decide if you need to alter it or not. If you're doing read/write partitioning, then you obviously want to read from the read partition. Some ORMs, though, have the object tied so strongly to an internal connection object that you can't trivially read it from one connection and save it into another. You'd either have to read the object initially from the write partition, minimizing the usefulness of read/write partitioning, or re-read it from the write partition when you realize you need to alter it, causing unnecessary overhead. (Note that you'll need to do this anyway if you need the write to happen in a transaction.)

Object oriented design is also problematic in a more general sense. The first principle of object orientation is "encapsulation," putting your code and data structure in one place: the class. This might make sense for business logic, but, for the purposes of re-factoring your data backend for partitioning or other strategies, you really don't want the data access code to be spread out among dozens of classes in your application. You want it all in one place, preferably even one source code file. It would let you plug in a whole new data backend strategy by replacing this source code file. For data-driven web development, you are better off not being too object oriented.

Even more generally speaking, organizing code together by mechanism or technology, rather than by "object" encapsulation, will let you apply all kinds of re-factorizations more easily, especially if you manage to decouple your application's data structures from any library-specific data structures.

Data Backends

Relational (SQL) databases such as MySQL were, for decades, the backbone of the web. They were originally developed as minimal alternatives to enterprise database servers such as Oracle Database and IBM's DB2. Their modest feature set allowed for better performance, smaller footprints, and low investment costs—perfect for web applications. The free software LAMP stack (Linux, Apache, MySQL and PHP) *was* the web.

Relational databases require a lot of synchronization overhead for clusters, limiting their scalability. Though partitioning can take you far, using a "NoSQL" database could take you even further.

Graph Databases

If your relational data structure contains arbitrary-depth relationships or many "generic" relationships forced into a relational model, then consider using a graph database instead. Not only will traversing your data be faster, but also the database structure will allow for more efficient performance. The implications for scalability can be dramatic.

Social networking applications are often used as examples of graph structures, but there are many others: forums with threaded and cross-referenced discussions, semantic knowledge bases, warehouse and parts management, music "genomes," user-tagged media sharing sites, and many science and engineering applications.

Though fast, querying a complex graph can be difficult to prototype. Fortunately, the Gremlin and SPARQL languages do for graphs what SQL does for relational databases. Your query becomes coherent and portable.

A popular graph database is Neo4j, and it's especially easy to use with Prudence. Because it's JVM-based, you can access it internally from Prudence. It also has embedded bindings for many of Prudence's supported languages, and supports a network REST interface which you can easily access via Prudence's document.external API.

Document Databases

If your data is composed mostly of "documents"—self-contained records with few relationships to other documents—then consider a document database.

Document databases allow for straightforward distribution and very fine-grained replication, requiring considerably less overhead than relational and graph databases. Document databases are as scalable as data storage gets: variants are used by all the super-massive Internet services.

The cost of this scalability is the loss of your ability to do relational queries of your data. Instead, you'll be using distributed map/reduce, or rely on an indexing service. These are powerful tools, but they do not match relational queries in sheer flexibility of complex queries. Implementing something as simple as a many-to-many connection, the bread-and-butter of relational databases, is non-trivial in document databases. Where document databases shine is at listing, sorting and searching through very large catalogs of documents.

Candidate applications include online retail, blogs, CMS, archives, newspapers, contact lists, calendars, photo galleries, dating profiles... The list is actually quite long, making document databases very attractive for many products. But, it's important to always be aware of their limitations: for example, merely adding social networking capabilities to a dating site would require complex relations that might be better handled with a graph database. The combination of a document database with a graph database might, in fact, be enough to remove any benefit a relational database could bring.

A popular document database is MongoDB. Though document-based, it has a few basic relational features that might be just good enough for your needs. OrientDB is interesting because it handles does both documents and graphs, and is entirely JVM-based. Another is CouchDB, which is a truly distributed database. With CouchDB it's trivial to replicate and synchronize data with clients' desktops or mobile devices, and to distribute it to partners. It also supports a REST interface which you can easily access via Prudence's document.external API. And, both MongoDB and CouchDB use JavaScript extensively, making it natural to use with Prudence's JavaScript flavor.

We've started a project to better integrate Prudence JavaScript with MongoDB.

Column Databases

These can be considered as subsets of document databases. The "document," in this case, is required to have an especially simple, one-dimensional structure.

This requirement allows optimization for a truly massive scale.

Column databases occupy the "cloud" market niche: they allow companies like Google and Amazon to offer cheap database storage and services for third parties. See Google's Datastore (based on Bigtable) and Amazon's SimpleDB (based on Dynamo; actually, Dynamo is a "key/value" database, which is even more opaque than a column database).

Though you can run your own column database via open source projects like Cassandra (originally developed by/for Facebook), HBase and Redis, the document databases mentioned above offer richer document structures and more features. Consider column databases only if you need truly massive scale, or if you want to make use of the cheap storage offered by "cloud" vendors.

Best of All Worlds

Of course, consider that it's very possible to use both SQL and "NoSQL" (graph, document, column) databases together for different parts of your application. See backend partitioning (page 166).