Diligence

Version 1.0-dev12
Main text written by Tal Liron

January 2, 2014

Contents

Assets Service

An "asset" is a common term for statically served files, such as images. Because assets use a lot of bandwidth to download, they are often cached on web browser clients (configured via a "cacheControl" route type in routing.js).

This service generates asset URLs, commonly used in dynamically generated HTML. The URLs are based on a user-defined template, although the default should suffice for most use cases.

The important feature added by this service is the ability to use the asset's base64-encoded cached content digest (usually a SHA-1) in the asset's generated URL. By specifically using this digest as a query param to the URL, two things are accomplished: 1) the URL will still be routed to the resources normally, because query params are not use by the "static" route type, and 2) because the URL is different, web browsers will use a different cache for the asset per client content.

The end result is that you could cache assets in clients for as long as you want (using "farFuture" for "cacheControl") while maintaining the ability to effectively bypass the cache for an asset whenever its content changes.

The asset digests are stored in a "digests.conf" file in the application's root subdirectory. It is a JVM properties file matching asset names to their digests. You can generate this file automatically using the "diligence:digests" Sincerity command.

Usage

Make sure to check out the API documentation for Diligence.Assets.

Authentication Service

This all-important service manages a few systems, which together allow your site to be "logged into" by individual users.

Usage

Make sure to check out the API documentation for Diligence.Authentication.

Users

Users are maintained in a simple MongoDB collection. You can easily attach settings to any user document, which can have any structure and depth you need.

Passwords are hashed many times and stored with a random salt. This good practice makes sure that even if a hacker were to steal your database and hack into a few accounts, they would not be able to use the results of their work to crack the other passwords.

Users are considered "entities" by the authorization service (page 6), such that you can attach permissions to user documents. Users can inherit permissions from groups and from other users.

Though you can maintain the user documents yourself, you can add the registration feature (page 61) to allow individuals to create their own users.

Sessions

When users do log in, they get a cookie with a session ID, which matches a document in the sessions collection. Thus, every conversation is associated with a session. You can store anything you want in the session document.

The authorization service (page 6) can check any operation against the conversation's session to make sure it's permitted.

The service comes with a task to make sure to remove sessions that have not been used for a while. It's a good security feature! (People tend to forget to log out, which can be especially dangerous in public places.)

Caching Per User

This very powerful feature uses a cache key patten handler to inject the currently logged in user ID into the cache key. This lets you cache any /web/dynamic/ or /web/fragment/ resource *per user*, which can do wonders towards helping your site scale. Of course, it does not make sense to cache every dynamic part of a page, but if you can indetify those fragments that look different only for different users then you've achieved a lot.

Authentication Forms

The authentication service comes with a bunch of /web/fragments/ that you can easily drop in to any page. They handle things like logging in, logging out, and showing the currently logged-in user.

Providers

Using Diligence's plug-ins library, the authentication service adds transparent support for 3rd-party authentication providers. Currently supported providers are Facebook, Windows Live, Twitter and OpenID (tested with Google, Yahoo, Myspace, LiveJournal).

Users coming from outside are real users: the first time they log in, a user document is created for them in the collection, and it can join in with settings, permissions, etc. Depending on how your application works, you can treat these users as any other user, or use the authorization service to treat them as "guests" with the ability to do only certain tasks. All 3rd-party users are automatically associated with an authorization group named after their provider. So, you can grant special permissions (or deny permissions) to "facebook."

This useful feature allows your application to be especially welcoming. Studies have shown that typical users think twice when a site requires registration (page 61). People either don't want to invest the effort in registering, or are anxious about yet another copy of their personal data being stored in somebody's database.

Authorization Service

When used with the authentication service (page 5), this service lets you secure your site by allowing only authorized users to access certain resources or perform certain operations.

Usage

Make sure to check out the API documentation for Diligence.Authorization.

Entities and Inheritance

Permissions are associated with "entities," which could be either individual user, from the authorization service (page 6), or groups, which are here stores in a simple MongoDB collection.

Each entity can inherit permissions from any number of other entities, in order. The common use case is for a user to "belong" to a few groups, and inherit their permissions. This lets you centrally manage permissions for large groups of users, and easily change a user's permission profile by changing their groups. Entities can inherit from other entities, and so on.

Permissions will be overriden by the inheritor: for example, if you specifically grant a user permission to edit a certain page, they will have this permission even if the group they inherit from specifically forbids it. The order of inheritance also allows for overriding.

Cascading Permissions

The common practice is to name permissions using a hierarchical dot notation, with each level of depth corresponding to moving into a specific section, resource or operation in your application. In some cases, it may make sense to treat a permission as if it covers all sub-permissions in a hierarchy. Here we call this "cascading permissions."

Backup Service

This service lets you do a live export of your MongoDB databases and collections to JSON, optionally gzipping the output to save space. You can set up your application's "crontab" to have the backup run regularly.

Backups are very fast: large databases can be fully exported in durations measured in seconds or minutes.

You might wonder what advantages this service has over MongoDB's mongodump or mongoexport tools. First, from our experience, the admin tools that come with MongoDB are overly simplistic and unreliable. Otherwise, Diligence's backup service offers the following advantages:

- **Thoughput:** Because we're using the Java MongoDB driver underneath, with its support for connection pooling, we can achieve much higher throughput than the command line tools, which use a single connection and no concurrency. The default is to use 5 threads (and thus 5 connections at most) at once.

- **True JSON:** The mongoexport tool does not export a real JSON array, instead it exports each document as a JSON dict, separating each document with a newline. Diligence exports a standard JSON array, readable from any standard JSON parser.
- **Consistency:** Works with the same MongoDB connection as your application, guaranteeing that you're backing up *exactly* the same data your application sees. This is especially important in a sharded or replica set deployment.
- **Operations:** You don't have to create system scripts to backup your DB. Instead, you can stay in JavaScript and Diligence. You do not even need MongoDB or its command line tools installed.
- **Iterators:** The backup service uses Iterators, so you can transform your data in various ways while backing up, or even include non-MongoDB data.

Usage

Make sure to check out the API documentation for Diligence.Backup.

To export the whole database:

```
document.executeOnce('/diligence/service/backup/')
Diligence.Backup.exportMongoDb({directory: '/tmp/diligence-backup/'})
```

The API further lets you select the MongoDB database and collections you wish to export, otherwise by default it uses the current default database and goes through all collections. You can also set "gzip" to true in order to gzip the resulting files.

To schedule the backup to run every day at 6am, add this to your "crontab":

```
6 <% document.executeOnce('/diligence/service/backup/'); Diligence.Backup.exportMongoDb({di
```

To import a collection:

```
Diligence.Backup.importMongoDbCollection({file: '/tmp/diligence-backup/users.json'})
```

The collection name will be parsed from the filename. If the filename ends with ".gz", it will be assumed to be gzipped and unzipped accordingly. (For example "/tmp/diligence-backup/users.json.gz".) By default the imported documents will be merged into the collection: set the "drop" key true if you want the collection to be dropped before importing.

Documents Service

This service lets you store versioned HTML documents in MongoDB using your choice among several markup languages: Markdown, Textile, Confluence, MediaWiki, TWiki and Trac. It's thus an essential building block for CMS features, such as wikis and blogs.

Every "document" in this service is indeed a single MongoDB document, but internally it composed of versioned "drafts". The last draft (of the highest revision number) represents the current state of the document. For efficiency, the last draft in its own key ("activeDraft"), allowing you to retrieve it from MongoDB without retrieving the whole history of drafts, which is an array. Additionally, each draft is stored both as markup source code and as rendered result, so that rendering only happens once.

MongoDB atomic operations guarantee that even if more than one person is revising a document at the same, no draft will be lost. Only last update to come in, though, will get to set the "activeDraft" key.

Documents are associated with a "site", of which there must be at least one. The Document Service can handle many "sites" at once, each with its own set of documents. The versioning system is designed to be global per each site, meaning that all drafts associated with a site will have *serial* and *unique* revision numbers per that site. This allows time travel: you that you can view the entire state of a site at a given time by fetching only drafts smaller than a certain revision. (This also implies that every draft as its own unique revision number, but there's no easy way in MongoDB to traverse drafts in this order.)

Note the markup rendering is handled by the HTML service (page 22), which you can choose to use directly if you do not need the versioning system.

Usage

Make sure to check out the API documentation for Diligence.Documents.

The API doesn't actually encourage you to access "documents" directly. Instead, you access "drafts" via the document ID and its revision, or simply request the latest draft. As stated above, the API is designed to be very efficient in doing this: whether it's the latest draft you need or a specific older revision, it's a very direct MongoDB fetch.

To fetch the latest draft by the document ID and print out its rendered HTML:

```
<html>
<%
document.executeOnce('/diligence/service/documents/')
var draft = Diligence.Documents.getDraft('4fc4457ae4b030c6611c072f')
%>
<body>
<%= draft.render() %>
</body>
</html>
```

Efficiency note: if that particular draft has already been rendered once, the render() call won't do anything at all, the rendered version having already been fetched. Other options for fetching drafts: you can also call "getLatestDraft" with a maximum revision number, or just call "getDraft" with a specific revision number you want.

To revise a draft, meaning that you will add a new revision to the document:

```
draft.revise('this is the markup source', 'textile')
```

Note that after revision, the draft object is updated with the new information. So you can call "draft.getRevision()" to see the new revision number if you need it. Again, remember that this particular revision number will be unique for the entire "site": no other document or draft will have it.

To create a new document:

```
var site = Diligence.Documents.getSite('4d5595e3f7f2d14d2ab9630f')
var draft = site.createDocument('a new document!', 'textile')
```

Note that "createDocument" returns a draft object, which will be the first and only draft of the document.

As you you can see, the usage is simple and efficient, but the implementation does have some sophistication. It's recommended that you look at the MongoDB collections for "documents" and "sites" to get a sense of how they work together.

Integration

To integrate the Documents Service into your application, use the document ID by called "getDocumentId()" on a draft object, and then store that ID in your own structure. For example, if you're writing a wiki, you might want to associate a wiki page with that document ID. Similarly for a blog entry. And, of course, this is schema-free MongoDB: feel free to add whatever data you need to your "document" documents. You can also inherit the Document Service classes and add the necessary functionality.

An important feature of wiki markup languages is support for special processing of wiki page references, turning them into HTML hyperlinks and possibly creating the page in the wiki. The Documents Service lets you hook in your code to support custom delimiters, so it can output proper links. Example:

```
var rendered = draft.render({
        codes: {
                start: '{{',
                end: '}}',
                fn: function(text) {
                        return '<a href="/link/{0}">{0}</a>'.cast(text)
                }
        }
})
```

You could then insert these custom codes in your markup:

```
This is a link to {{mywikitopic}}.
```

The "codes" key can be an array of several such code processors, and the function can output anything at all, not just links, so you can use it to extend the markup language. In fact, the function can actually do something more substantial than output: you could, for example, save a cross reference to the remote wiki page, or create an empty template for a non-existing page.

Note that custom code processing happens only during the first render: in subsequent calls to "render()" on this draft the "codes" argument will be ignored.

Configuration

If you like, you can avoid specifying the markup language in all the API calls. The default language would then be "textile", but you can change it in your application's "settings.js" by adding something like this to your app.globals:

```
app.globals = {
        ...
        diligence: {
                service: {
                        documents: {
                                defaultLanguage: 'markdown'
                        }
                }
        }
}
```

Events Service

Almost every application framework provides some generic way to listen to and fire one-way messages called "events." By decoupling event producer code from event consumer code, you can allow for a looser, more dynamic code architecture.

Some frameworks go a step beyond simple code decoupling, and treat producers and consumers as separate *components*, in which the producer cannot make any assumptions on the consumer's thread behavior. Consider two extremes: a consumer might respond to events immediately, in thread, possibly tying up the producer's thread in the process. Or, it might allow for events to be queued up, and poll occasionally to handle them. In the latter highly abstracted situations, events are called "messages," and implementations often involve sophisticated middleware to queue messages, persist them, create interdependencies, and make sure they travel from source to destination via repeated attempts, back-off algorithms, notifications to system administrators in case of failure, etc.

One size does not fit all. With Diligence, we wanted to keep events lightweight: we assume that your consumer and producer components are all running inside a Prudence container: either they are explicit or implicit resources running in web request threads, or they are asynchronous tasks. This allows us to optimize for this situation without having to rely on abstracting middleware. Still, more sophisticated, dedicated messaging middleware is out there and available if you need it. We suggest you try RabbitMQ.

That said, the combination of Prudence Hazelcast clusters, MongoDB, and JavaScript's inherent dynamism within the Prudence container allows for a truly scalable event framework. If what you need is asynchrony and scalable distribution, rather than generic decoupling, then Diligence events might be far more useful and simpler than deploying complex middleware.

> The point of an event-driven architecture is that you're relinquishing some control of your code-flow. It's thus hard to know, simply by looking at the code, which parts of it will be triggered when an event is fired. You also need to know what exactly is subscribing and where that listener code is. Decoupling code is a great way to introduce some really difficult bugs into your codebase, and vastly reduce its debuggability. We present this service for your use, but encourage you to think of the costs vs. the benefits in terms of code clarity. Perhaps there is a more straightforward way to solve your problem? If all you need as asynchronicity, then you can also use the Prudence.Task API more directly, allowing you to call *specific* listening code, rather than any generic subscriber. The bottom line is that as great as this service is, we recommend using it with discrimination.

Usage

Make sure to check out the API documentation for Diligence.Events.

In-Thread Events

First, the basics. Here's our "/libraries/politeness/acknowledgements.js":

```
Diligence.Events.subscribe({
        name: 'payments.successful',
        fn: function(name, context) {
                logger.info('User {0} has paid us {1}!', context.username, context.amount)
                Acknowledgements.sendThankYou(context.username)
        }
})
```

Then, to fire the event, somewhere in our payments workflow:

```
document.executeOnce('/politeness/acknowledgements/')
Diligence.Events.fire({
        name: 'payments.successful',
        context: {
                username: user.name,
                id: user.id,
                amount: payment.amount
        }
})
```

For this to work, you have to make sure the firing code has already run the code that hooks up the listeners. Often, a simple document.execute will do the trick, as in this example.

Asynchronous Events

You can easily make the listeners run outside your thread, in fact anywhere in your Prudence cluster. This, of course, is crucial for scalability, because you don't want the listeners holding your web request thread.

For this to work, we need to add something small to our subscription:

```
Diligence.Events.subscribe({
        name: 'payments.successful',
        dependencies: '/politeness/acknowledgements/',
        fn: function(name, context) {
                logger.info('User {0} has paid us {1}!', context.username, context.amount)
                Acknowledgements.sendThankYou(context.username)
        }
})
```

Note that we had to add a "dependencies" key to the listener, to allow it to be called in different contexts. These dependencies are document.executeOnce'd to make sure the calling thread has access to all the code it needs.

Firing it:

```
document.executeOnce('/politeness/acknowledgements/')
Diligence.Events.fire({
        name: 'payments.successful',
        async: true,
        context: {
                username: user.name,
                id: user.id,
                amount: payment.amount
        }
})
```

All we did was add "async: true", and... that's pretty much it. Every listener will run in its own thread within the global pool. You can add a "distributed: true" flag to cause listeners to be executed anywhere in the cluster, and there's where things get really powerful: you can properly scale out your event handling in the cluster, with nothing more than a simple flag.

How does this magic work? It's JavaScript magic: *we're evaluating the serialized listener source code.* The code that fires the event is called as a Prudence task. The task makes sure to run the dependencies and evaluate the JavaScript you stored. Voila. (Serialization and eval will only occur on async events: otherwise, it's a regular function call.)

Concerned about JavaScript eval performance? Generally, it's very fast, and surely whatever overhead is required to parse the JavaScript grammar would be less than any network I/O that a distributed event would involve. If you're really worried about performance, make sure to store as little code as possible in the listener function and quickly delegate to compiled code. For example, your listener can simply call a function from one of the dependency libraries, which are already compiled and at their most efficient.

Stored Listeners

So far so good, but both examples above require you to execute the code that subscribes the listeners before firing the event. Stored listeners remove this requirement by saving the event and its listeners in one of several storage implementations.

For example, let's store our listeners in application.distributedGlobals, so that we can fire the event anywhere in the Prudence cluster:

```
var globalEvents = new Diligence.Events.GlobalsStore(application.distributedGlobals, 'myevent
Diligence.Events.subscribe({
        name: 'payments.successful',
        stores: globalEvents,
        id: 'sendThankYou',
        dependencies: '/politeness/acknowledgements/',
        fn: function(name, context) {
                logger.info('User {0} has paid us {1}!', context.username, context.amount)
                Acknowledgements.sendThankYou(context.username)
        }
})
```

We can also use application.globals or application.sharedGlobals.

One small issue to note when using stored listeners is that storage must support concurrency. One implication of this is that you need to make sure that they are not registered more than once, say by multiple nodes in the cluster, otherwise your listener code would be called multiple times. And that's what the listener "id" field is for. (In fact, the "id" field can also be used for in-thread listeners.) It also might make sense to set up all your stored listeners in your "/startup/" task, but it's not a requirement: you can install listeners whenever necessary and relevant.

Because it's stored, firing the event does not require us to execute the listener code first in our thread. We can remain blissfully unaware of who or what is subscribed to our event:

```
Diligence.Events.fire({
        name: 'payments.successful',
        stores: globalEvents,
        async: true,
        distributed: true,
        context: {
                username: user.name,
                id: user.id,
                amount: payment.amount
        }
})
```

The "stores" param can also be an array, so you can fire the event on listeners from various stores. The in-thread store is in "Diligence.Events.defaultStores", so you can concat that to your custom store if you want to fire the event across all stores. Or, set "Diligence.Events.defaultStore" to your own value.

Persistent Listeners

In the above example, the listeners would have to be re-subscribed when the application restarts, because it cannot guaranteed that application.distributedGlobals would keep its value. (Well, you *can* configure Hazelcast to persist the distributedGlobals map...)

Let's store our listeners in MongoDB, instead (the default is to use the "events" MongoDB collection):

```
Diligence.Events.subscribe({
        name: 'payments.successful',
        stores: new Diligence.Events.MongoDbCollectionStore(),
        id: 'sendThankYou',
        dependencies: '/politeness/acknowledgements/',
        fn: function(name, context) {
                logger.info('User {0} has paid us {1}!', context.username, context.amount)
                Acknowledgements.sendThankYou(context.username)
        }
})
```

Everything is otherwise the same. Neat!

You can also store events inside a specific, arbitrary MongoDB document, using Diligence.Events.MongoDbDocumentStore. This is a great way to keep events and their listeners (and the namespace for events) localized to a specific object without adding external mechanisms and storage.

Finally, you can create your own custom store class to store events anywhere else.

Configuration

You don't have to configure the Events Service, but it is possible to set a few defaults. In your application's "settings.js" add something like this to your app.globals:

```
app.globals = {
        ...
        diligence: {
                service: {
                        events: {
                                defaultAsync: true,
                                defaultDistributed: true,
                                defaultStores: [function() {
                                        document.executeOnce('/diligence/service/events/')
                                        return new Diligence.Events.MongoDbCollectionStore()
                                }]
                        }
                }
        }
}
```

Note the use of function(): this is required in order to allow the Events Service to lazily create the service implementations on demand during runtime.

Forms Service

Forms are an important feature for any GUI application. As for web applications, forms are supported in HTML, but many web applications also use JavaScript to send forms to the server in the background ("AJAX"). Diligence goes a long way towards making it easier for you to use both models, each with its own complexities and subtleties, through a unified API. Allowing for both AJAX and HTML client forms with the same server code makes it easy to support "legacy" clients that can't use AJAX.

Diligence explicitly supports Ext JS Forms, and recommends Ext JS as a client-side framework. See the section on Sencha Integration for full details.

Client-side Validation vs. Server-side Validation

Like all good form frameworks, Diligence's Form Service makes it especially makes it easy to implement form validation, both on the server and the client, using an extensible system of field types. Due to the fact that Diligence is a server-side JavaScript framework, you can actually share the exact same validation code on both the client and the server! This marvelous advantage makes using forms in Diligence less cumbersome as compared to other frameworks.

What are the advantages of each kind of validation? Why you would want both?

- **Server-side validation:** You'll at least want this. It protects against user error, and can return friendly error codes so that the user will know how to correct the form. It's also important for security, to make sure that potentially damaging data will never enter the other parts of your application. For example, you can protect yourself from attacks which try to overflow your database with too much data, or attempts at SQL injection. (MongoDB injection attacks may be possible, too!) Note also that Diligence Forms will automatically catch server-side exceptions, invalidating the form and returning the error to the user, but obviously relying on exceptions is not secure enough.
- **Client-side validation:** Adding this to server-side validation will enhance the user experience by providing fast, instant feedback, thus avoiding an extra round-trip to the server to validate the form data. It will also save you some bandwidth and help you scale. There are two kinds of client-side validation supported by Diligence, which when used together will offer the best user experience:
  - **Validation:** The field's whole value will be tested before allowing the form to be submitted.
  - **Masking:** When entering textual data, this locks the user's text field to only accept allowed characters. For example, if an integer is required, only the characters "0" to "9" and "-" (for negative integers) will be allowed.

Setup

Make sure to check out the API documentation for Diligence.Forms.

Every form is an instance of "Diligence.Forms.Form" or its subclasses. This class inherits "Diligence.REST.Resource," and thus can immediately be hooked to your URI-space. Indeed, much of the Forms Service power comes from such a setup, so we'll go over it here. However, note that is also possible to use the form instance without hooking it up to a URI, as we'll show in "Usage," below.

First, let's configure the URI-space in your application's "routing.js". Add the following to app.routes and app.dispatchers:

```
app.routes = {
        ...
        '/multiply/': '@multiply'
}

app.dispatchers = {
        ...
        javascript: '/manual-resources/'
}
```

We can now configure our resources in "/libraries/manual-resources.js":

```
document.executeOnce('/diligence/service/forms/')

var multiplyForm = {
        fields: {
                first: {
                        type: 'number',
                        label: 'A number',
                        required: true
                },
                second: {
```

```
                type: 'integer',
                label: 'An integer',
                required: true
        }
    },
    process: function(results) {
        if (results.success) {
            results.values.result = Number(results.values.first) Number(results.v
            results.msg = '{first} times {second} equals {result}'.cast(results.v
        }
        else {
            results.msg = 'Invalid!'
        }
    }
}

resources = {
    ...
    multiply: new Diligence.Forms.Form(multiplyForm)
}
```

Let's look more closely at this setup below.

Fields and Validation

Each field has at least a name (the key in the dict) and a type (defaults to "string"). If that's all the information you provide, then no validation will occur: any value, including an empty value, will be accepted.

- **required:** The field cannot be empty, neither a null value nor an empty string will be accepted. Note that the "required" check happens before the "validator" function is called. [TODO error key]
- **validator:** A validating function, meant for both client- and server-side validation. It *must* return true to signify that the value valid. Any other return value will signify invalidity. (See "validation functions," below.)
- **serverValidator:** As "validator", but intended only for server-side validation.
- **clientValidator:** As "validator", but intended only for client-side validation.
- **mask:** A regular expression used for masking. This could be JavaScript literal regular expression, a RegExp object, or a string.
- **serverValidation:** Set to false to override the default for the form.
- **clientValidation:** Set to false to override the default for the form.
- **textKeys:** An array of text pack keys used by validator function. See "Text and Internationalization," below.
- **type:** Instead of providing "validator", "clientValidator", "serverValidator", "mask", "serverValidation", "client-Validation" and "textKeys" for every single field, you can specify a "type" from which these keys will be inherited. Defaults to "string". Note that even if you specify "type", you can override the inherited keys in the field definition.
- **value:** This is a default value assigned to the field when the form is initialized.

Validator Functions

Let's look at such a function in the context of a field definition:

```
first: {
    required: true
    validator: function(value, field, conversation) {
        return value % 1 == 0 ? true : 'Must be an integer'
```

```
        }
}
```

The return value, as stated before must be true to signify a valid value. Otherwise, the value will be considered invalid and the return value will be used as the error message.

The arguments are as follows:

- **value:** The value to be validated, most likely a string.
- **field:** The field definition. This is useful if you are using the same function for multiple fields, and need to validate differently per field. Note that the field definition is framework-dependent. For example, if you are on the server, it will look like the examples above, but if you're on an Ext JS client, then it will use Ext JS's definition. Because we're not using "field" in this example, we supplied just one "validator" function for both the client and the server. However, if you do need to access "field", it may be better to have separate "serverValidator" and "clientValidator" functions.
- **conversation:** The Prudence conversation. Only available on the server.

The function is called with an implicit "this" object, which obviously refers to different objects on the server and the client, but you can expect these fields:

- **form:** The form instance. Only available on the server.
- **textPack:** The currently used text pack. Always available on the server, and available on some clients, such as Ext JS if you use Diligence's Sencha Integration. See "Text and Internationalization," below, for more information.

Through accessing the "field" and "conversation" arguments as well as "this.form", you can do some very sophisticated server-side validation. For example, you can query MongoDB and check against data, check for security authorization, etc. And, of course, you can use similar sophistication for client-side frameworks according to the features they provide.

(At this point, you might be wondering how exactly client-side validator functions get to be called on the client, since we are defining them on the server. We'll talk about that in "Usage," below, but the solution is simple: we send the source code directly as text!)

Types

The Forms Service comes with a few basic types to get you started, all defined under "Diligence.Forms.Types":

- **string:** All values are valid. This is the default type.
- **number:** Valid if the value can be converted into a JavaScript number. Masked for digits, "-" and ".".
- **integer:** Valid if the value can be converted into a JavaScript integer. Masked for digits and "-".
- **email:** Valid if the value is a standard email address. Does no masking.
- **recaptcha:** See reCAPTCHA.

You can also provide your own types:

```
var serviceForm = {
        types: {
                bool: {
                        validator: function(value, field, conversation) {
                                value = String(value).toLowerCase()
                                return (value == 'true') || (value == 'false')
                        }
                }
        }
        fields: {
                enabled: {
                        type: 'bool',
```

```
                label: 'Whether the service is enabled'
            }
            ...
        }
        ...
}
```

Text and Internationalization

If you don't need internationalization, then just use the "label" key in the field definition to set up the text directly. If unspecified, it will default to the field name.

Otherwise, read about the Diligence Internationalization Service to understand how to set it up. We will use the "labelKey" key instead of "label", and also set up the list of other keys we might need using the "textKeys" key:

```
first: {
        labelKey: 'myapp.myform.field.first',
        textKeys: ['myapp.myform.validation.integer.not'],
        required: true
        validator: function(value, field, conversation) {
            return value % 1 == 0 ? true : this.textPack.get('myapp.myform.validation.int
        }
}
```

The above code will work on both the client and the server, because "textKeys" ensures that all those text values are sent to the client.

Processing

Let's look at our processing function again:

```
process: function(results) {
        if (results.success) {
            results.values.result = Number(results.values.first) Number(results.values.se
            results.msg = '{first} times {second} equals {result}'.cast(results.values)
        }
        else {
            results.msg = 'Invalid!'
        }
}
```

The function will be called after validation happens, with "results" being a pre-defined dict, ready for you to modify, with the following keys:

- **results.success:** Will be true if the form data is valid. You can change it to false during processing in order to signify an error to the user. Exceptions thrown in this function will also cause "results.success" to be false.
- **results.values:** A dict of the form values sent from the user. The value keys correspond to the field keys. Note that "results.values" will be deleted if "results.success" is true. The reason is that you should only need the old values if the user needs to correct the form in case of an error. If the form was successful, the form values should be reset. (In the example above we are setting "results.values.result" only for the purpose of the string template cast.)
- **results.msg:** A message to be displayed to the user.
- **results.errors:** A dict of error messages per field, as set by the field validator functions. The error keys correspond to the field keys. This dict will not exist if "results.success" is true when this function is called.

As stated, you can modify any of these results as you need, including settings "results.errors" to extra per-field error messages, beyond what was performed in validation.

Indeed, you can use the processing function to do extra validation, which might have to take into consideration the form as a whole, rather than individual fields. For example, what if a start-date field in the form is set to be

after an end-date field? You can find that out here and set "results.success" to false, with "results.errors.endDate" to a suitable error message.

The return value of this function is ignored.

Usage

If you've set up the resource as instructed above, you should be able to access it at the specified URI. By default, it will only support the HTTP POST operation, for which it expects an entity in the "application/x-www-form-urlencoded" media type, as is used by HTML forms.

Later on, we'll show you below how the Forms Service can help you render an HTML form, complete with validation error messages and internationalization support.

HTML Forms

For now, let's just start with a straightforward, literal HTML example:

```
<html>
<body>
<form action="<%= conversation.pathToBase + '/multiply/?mode=redirect' %>" method="post">
        <p>First value: <input name="first" /></p>
        <p>Second value: <input name="second" /></p>
        <p><input type="submit" value="Multiply!" /></p>
</form>
</body>
</html>
```

You'll notice that added a "mode" query parameter to the action URI. This lets us select one of the following modes of behavior supported by the resource:

- **json:** This is the mode you'll want to use for AJAX, as it returns the form results in JSON format. JSON mode additionally supports the "human=true" query parameter to return the JSON in multiline, indented format. *Note that this is the default mode.*
- **redirect:** After processing, the resource will redirect the client to a new URI. The default is the sending URI, but you can set up specific URIs for success and failure.
- **capture:** As an alternative to a redirect, you can perform a Prudence "capture" of another internal URI. The user will see the URI of the form resource itself, but the content will come from elsewhere. Note that because capturing happens in the same conversation, without a round trip to the client, you can use all the data used during processing. If you do a redirect, the client would be sending a new request and that data would be gone.

When creating your resource instance, you can change the default to be something other than "json" by setting the "mode" key. JSON was chosen as a default because it's easiest to test and produces the least amount of side-effects due to unintentional access to the resource.

Testing Your Form Resource with cURL

cURL is an HTTP command line tool based on the cURL library, available for a great many Unix-like operating systems as well as Windows. It's especially useful for testing RESTful APIs. Here's a quick tutorial to get you started with using cURL with the Forms Service.

Try this command to send a POST to your form:

```
curl --data-urlencode first=5 --data-urlencode second=6 "http://localhost:8080/myapp/multiply
```

Note that using the "data-urlencode" switch will automatically set the method to POST and the entity type to "application/x-www-form-urlencoded."

Because the resource's default mode is JSON, you should get this result:

```
{
        "success": true,
        "msg": "5 times 6 equals 30"
}
```

If you're using AJAX to POST to the resource, then you'll have to parse these JSON results accordingly. See "Processing" above for the exact format of the results.

Also note that this format is immediately usable by Ext JS forms! See Diligence's Ext JS Integration for more details.

You can also use cURL to test redirect mode:

```
curl -v -e "http://my-referring-url" --data-urlencode first=5 --data-urlencode second=6 "http
```

You should see the redirected URL in the "Location" header, as well as an HTTP status of 303.

Redirect Mode

Redirect mode will by default redirect the client to the referring URI, using HTTP status 303 ("See Other").

But, you can explicitly set the redirection URI to something specific in "/libraries/resources.js":

```
var multiplyForm = {
        ...
        redirectUri: '/multiply/results/',
        mode: 'redirect' // we'll make this the default mode (instead of 'json')
}
```

You can also set "redirectSuccessUri" and "redirectFailureUri" separately.

Or, you can set the URI dynamically by setting "results.redirect" in your processing function.

This should go without saying, but client redirections means that a whole new HTTP GET request will be sent by the client, such that all your conversation data will be gone. Of course, often the resulting page should depend on the result of form processing. There are two good strategies for handling this:

- Because you can set the URI dynamically in "results.redirect", you can create a special kind of results view. For example, let's say you are implementing a search form (like Google's search engine page), which should redirect the user to the search results. You could redirect to a URI which includes the search results, for example in the URI query string. For example, searching for the phrase "cool apps" could end up redirecting to something like this: "http://myapp.org/search/?terms=cool+apps". In "/mapped/search.d.html" you would then unpack the terms and display the correct results. (You likely want to cache the search results for a while for the best user experience!)

- Another option is set a cookie, using Prudence's "conversation.createCookie" API, which you can then read in the redirected page using "conversation.cookies". Cookies are great if the result is very specific to the user, but note that bookmarks to the result URL would display something different if the cookie does not exist.

Capture Mode

Capture mode may seem similar to redirect mode: you supply a new URI which gets displayed to the client. The difference is that "redirection" happens on the server, rather than the client. That means that the URI for the client will remain the same. This is more efficient in that an extra round trip from the client is avoided. However, it creates serious problems for bookmarking: the result URI ends up being the same as the form URI. Think carefully about the pros and cons of each approach in terms of what would provide the best user experience. (Also see manual mode, below, which is similar in behavior to capture mode.)

You can access the form and the captured page using "Diligence.Forms.getCapturedForm" and "Diligence.Forms.getCapturedResults". This API will only work in a captured page. Let's see how this works by creating a "/mapped/multiply/results.d.html" for our results:

```
<html>
<body>
<%
document.executeOnce('/diligence/service/forms/')
```

```
var form = Diligence.Forms.getCapturedForm(conversation)
var results = Diligence.Forms.getCapturedResults(conversation)

if (results && results.success) {
%>
<p><%= results.msg %></p>
<% } else { %>
<form method="post">
        <p>First value: <input name="first" /></p>
        <p>Second value: <input name="second" /></p>
        <p><input type="submit" value="Multiply!" /></p>
</form>
<% } %>
</body>
</html>
```

We can specify the capture URI when we create the resource, in "/libraries/resources.js":

```
var multiplyForm = {
        ...
        captureUri: '/multiply/results/',
        mode: 'capture' // we'll make this the default mode (instead of 'json')
}
```

You can also set "captureSuccessUri" and "captureFailureUri" separately.

Or, you can set the URI dynamically by setting "results.capture" in your processing function.

Finally, while it's not entirely necessary, you can hide the URI. This will guarantee that it's only available for capturing, but the user won't be able to reach it by entering the URL in their browser. You do this in your application's "routing.js":

```
app.routes = {
        ...
        '/multiply/results/': 'hidden'
}
```

Manual Mode

If you go back to the code for the simple HTML form we've provided above, you might wonder if having the form as a separate resource is necessary. While it does provide a cleaner separation between the form processing resource and the HTML view resource, it would be more efficient if we could avoid that extra client redirect and do the processing and viewing in the same resource.

Before we consider if this is a good idea or not, let's see how this would be easily done with Diligence:

```
<html>
<body>
<%
document.executeOnce('/diligence/service/forms/')

var form = Diligence.Forms.getForm('/multiply/')
var results = form.handle(conversation)
if (results && results.success) {
%>
<p><%= results.msg %></p>
<% } else { %>
<form method="post">
        <p>First value: <input name="first" /></p>
        <p>Second value: <input name="second" /></p>
        <p><input type="submit" value="Multiply!" /></p>
</form>
```

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

A few points to explain:

- "Diligence.Forms.getForm" is a very useful function. It works by doing an internal GET on the URI to fetch the form instance. We could have also avoided setting up the instance in "resources.js" as well as routing it in "routing.js", and instead simply have created the "Diligence.Forms.Form" instance here. But this lets us use the instance both as a resource and in manual mode, as we've done here.
- The "handle" method will validate and process the form, but only if the conversation is a POST. If it's not processed, it will will return null.
- Note how we're displaying different content according to whether the processing was successful or not.

So, is manual mode a good idea or not? If can provide a straightforward, quick-and-dirty way to implement a form. Compact, too: you can create the instance, do all the processing, and put all the view code in a single file. There's no need to set up routing for a resource.

But, there are a few disadvantages:

- The code is not very easy to follow or debug. The same page is doing three different things: 1) displaying the form, 2) displaying errors, and 3) displaying the results of a successful post. (You could put each view in a different included fragment, but would lose the compactness.)
- This also means that caching logic for the page my be difficult if not impossible to do efficiently.
- A single URI with multiple uses can be confusing for users. If they bookmark the result "page," but try to go to it again at a later time, it would display an unfilled form, because it's the same page. This is problematic for all POSTed HTML forms: it's always a good idea to redirect the user to a book-markable URI that responds correctly to an HTTP GET.

You can mitigate some of these problems by using capture mode instead. Capture mode will let you use a separate page for results, which can be cached (on the server, at least: a POST will never cache on the client), while keeping the URI the same.

Low-Level Manual Mode

So, this "mode" actually does not use the Diligence Forms Service at all, instead it relies directly on the Prudence API. We thought it would be a good idea to include it here for the sake of completion. Sometimes, even manual mode may not be quick-and-dirty enough! Note that validation is very, very basic: if the value cannot be converted, you will simply get a null.

Here's how it would look:

```
<html>
<body>
<%
document.executeOnce('/prudence/resources/')
document.executeOnce('/sincerity/objects/')
document.executeOnce('/sincerity/templates/')

var form
if (conversation.request.method.name == 'POST') {
        form = Prudence.Resources.getForm(conversation, {
                first: 'float',
                second: 'int'
        })
}

if (form && Sincerity.Objects.exists(form.first) && Sincerity.Objects.exists(form.second)) {
        form.result = Number(form.first) Number(form.second)
```

```
%>
<p><%= '{first} times {second} equals {result}'.cast(form) %></p>
<% } else { %>
<form method="post">
        <p>First value: <input name="first" /></p>
        <p>Second value: <input name="second" /></p>
        <p><input type="submit" value="Multiply!" /></p>
</form>
<% } %>
</body>
</html>
```

Rendering an Internationalized HTML Form

In all the above examples, we explicitly entered the HTML for the form and its fields. But, Diligence Forms can also generate the HTML for you, and moreover use the Internationalization Service, in conjunction with the Authorization Service, to render the correct text for the user's preferred language.

The rendered HTML is very straightforward: it's a simple <input> tag when using "htmlText" (or a <textarea> tag when usng "htmlTextArea"), with a connected <label> prepended. If the field failed validation then an extra is appended with the validation error message. Furthermore, in case of validation error, all tags for the field will get the "error" class, allowing you to use CSS in order to stylize validation errors.

You should add the "results" of the form if you have them (they are available in capture mode and manual mode) to the method calls. This will render errors properly, and also set the values of the form to the previous values, making it easier for the user to correct the form.

Here's an example using manual mode, which also uses CSS to stylize form errors:

```
<html>
<head>
<style>
form input.error {
        border: 1px solid red;
}
form div.error {
        color: red;
        display: inline;
        padding-left: 5px;
}
</style>
</head>
<body>
<%
document.executeOnce('/diligence/service/forms/')

var form = Diligence.Forms.getForm('/multiply/')
var results = form.handle(conversation)
%>
<form method="post">
        <div><%= form.htmlText({name: 'first', conversation: conversation, results: results})
        <div><%= form.htmlText({name: 'second', conversation: conversation, results: results}
        <div><input type="submit" value="Multiply!" /></div>
</form>
</body>
</html>
```

HTML Service

This service supports two uses:

1. Generating structured, internationalized, sanitized HTML code.
2. Consuming HTML and parsing it using a jQuery-like syntax. For this we rely internally on the jsoup library.

Usage

Make sure to check out the API documentation for Diligence.HTML. Also useful is the API documentation for Sincerity.XML.

From JSON to HTML

The most general API is "build", which accepts a JSON structure and turns it into HTML:

```
<%
print(Diligence.HTML.build({
        _tag: 'div',
        _children: [
        ]
}))
%>
```

The generated HTML is:

```
<div>
</div>
```

All text is properly escaped as appropriate for HTML content and HTML tag attributes. Note that keys beginning with "_" are treated specially, as explained below.

The library also contains shortcuts for simple HTML elements, like so:

```
<%= Diligence.HTML.img({src: 'http://threecrickets.com/media/three-crickets/prudence-small.pn
```

Special Attributes

Internationalization

Parsing HTML

Other Utilities

Sanitizing

Internationalization Service

This is a straightforward but powerful service that lets you render text by key from "text packs" per locale.

A single application can load many text packs simultaneously, such that every user could see text in their preferred language, if you support it. Text packs can be cached in memory (in the application globals) once loaded, while giving you control over the cache duration in case you want to enable on-the-fly editing of text packs.

Importantly, this service supports bi-directionality (left-to-right or right-to-left languages) by keeping track of the direction of *every single key*. This is crucial, because you may have to render left-to-right and right-to-left text on the same page, and you want to make sure that each key is rendered correctly.

Text packs can inherit each other, making it easy to manage many text packs with a common base, or to merge text packs from different sources into one. For example, you can you have a general English text pack, and the a British English text pack, which inherits the general English text pack and only overrides those keys that are different. Directionality of keys is maintained: if a right-to-left Arabic text pack inherits an English text pack, those left-to-right keys from the English text pack will stay left-to-right.

Setup

Text packs are looked for first JSON files and then in a MongoDB collection called "textpacks". You can combine text packs from both, and inherit either from the other.

The text pack is a dict that must include at least a "text" key, with a structure of any depth, and optionally a "direction" key, which could be either "ltr" (the default, for left-to-right, the default), or "rtl" (for right-to-left languages). Additionally, you can add an "inherits" key, which can be either a single locale specification or an array of locale specifications, which specifies which text packs should be merged into this one. The values of the inheriting text pack will always override those from the inherited text packs.

Locale Specifications

In all the following examples, whenever you need to specify a locale you can specify it either as a string signifying the language or in full form, with "language", "country" and "variant" keys. For example, these two locale specifications would be considered equivalent:

```
"en" == {"language": "en"}
```

But this locale would be different:

```
{"language": "en", "country": "nz"}
```

As MongoDB Documents

Text packs will be found in the collection called "textpacks". They have the same structure as the JSON files, but must also have a "locale" key, with the locale specification as detailed above. Here's an example document:

```
{
        _id: ObjectId("4d6803e6ddfe99e799c7b809"),
        "locale": {
                "language": "en",
                "country": "nz"
        },
        "direction": "ltr",
        "inherit": "en",
        "text": {
                "application": {
                        "myapp": {
                                "time": "It is now {now}"
                        }
                }
        }
}
```

Again we'll emphasize: even though this text pack is defined in MongoDB, it can inherit the "en" text pack defined in the JSON file.

You'll usually prefer one method or the other, but it might make sense to use both: for example, a default text pack can be hard-coded for your application, to allow it to function even if MongoDB is not available.

As JSON Files

If stored in files, the name of the file must be in the form "[locale].json". For example, for the English locale it is "en.json". If the locale has country and variant specifications, they are added with underscores. For example, English/New Zealand would be "en_nz.json".

An example "en.json" file:

```
{
        "direction": "ltr",
        "text": {
                "application": {
                        "myapp": {
```

```
                        "time": "It is now {now}"
                    }
                }
            }
}
```

Per-User Text Packs

See the "Authentication Service."

Usage

Make sure to check out the API documentation for Diligence.Internationalization.

Here's an example:

```
<%
document.executeOnce('/diligence/service/internationalization/')
var textPack = Diligence.Internationalization.getPack('en')
%>
<p dir="<%= textPack.getDirection('application.myapp.time') %>">
        <%= textPack.get('application.myapp.time', {now: new Date()}) %>
</p>

var textPack = Diligence.Internationalization.getCurrentPack(conversation)
```

The "get" method will automatically cast templates. In this case, our text is a template in the form of "It is now {now}". The "getDirection" method will return either "ltr" or "rtl" according to the directionality of that specific key.

Attaching a Text Pack to the Conversation

In many cases, you would not want to specify the locale explicitly, but instead would want it loaded from, say, the logged-in user's stored preferences. In that case, you can store the selected locale in the conversation.locals as "diligence.service.internationalization.pack", or use this shortcut:

```
textPack.setCurrent(conversation)
```

And then retrieve it like so:

```
var textPack = Diligence.Internationalization.getCurrentPack(conversation)
```

Many of Diligence's other services and features rely on this API call, so make sure to set up the conversation.local appropriately if you want them to support internationalization.

Configuration

In your application's "settings.js", add something like this to your app.globals:

```
app.globals = {
        ...
        diligence: {
                service: {
                        internationalization: {
                                defaultLocale: 'en',
                                cacheDuration: 10000, // in milliseconds; if 0 (the default)
                                path: Sincerity.Container.getFileFromHere('textpacks') // opt
                        }
                }
        }
}
```

It would then look for ".json" files in the "/textpacks/" directory under your application's main directory.

To signify the locale in full form during configuration, make sure to use the "." key to avoid flattening of the dict (see Sincerity.Objects.flatten). For example:

```
defaultLocale: {.: {language: 'en', country: 'nz'}}
```

Cache Service

The Prudence platform already provides excellent caching for your generated HTML, with a lot of control over cache keys. It also provides you with an API to access the cache backend directly. But, that is a very special purpose cache highly optimized for that particular task.

With the Diligence Cache Service, we are providing you with a general purpose caching mechanism, letting you store anything MongoDB can take, again with full control over key generation. Moreover, the Cache Service lets you easily wrap arbitrary JavaScript functions, so that you can transparently cache their results.

Usage

Make sure to check out the API documentation for Diligence.Cache.

A simple example:

```
document.executeOnce('/diligence/service/cache/')
document.executeOnce('/sincerity/jvm/')

var cache = new Diligence.Cache('result')

var getResult = function(userId) {
        Sincerity.JVM.sleep(1000)
        return {
                userId: userId,
                randomValue: Math.random()
        }
}
.cache(cache, 10000, 'result.')

var result = getResult(123)
```

A few notes:

- Our "getResult" function here is very silly, and purposely delays for 1 second. However, it could easily do very real things: for example, a slow map-reduce query on MongoDB, fetching data from an external service or site, etc.
- We here cache the result for 10 seconds, meaning that only once every 10 seconds would the function actually be called. In all other cases, the last cached result will be retrieved from the MongoDB collection. It should go without saying, but: this works in high-concurrency, so any number of threads and nodes would be using the same cached value.
- The data must be compatible with MongoDB. This includes anything that works with MongoDB's extended JSON format.
- We here use a simple string prefix ("result.") to generate our cache key. The service will automatically add the function arguments to the cache key, so in this case our cache key will be "result.123". However, you can supply a function instead of a string, which would return the final cache key as the string using whatever logic you need. An implication of this is that you can use a single cache collection to store results of numerous functions, as long as you make sure that the final cache keys don't overlap.
- The library overrides the JavaScript function prototype, adding the "cache" method to it. The Diligence.Cache API also has methods that offer more flexibility. For example, it can let you set advanced logging, so that you can see how the cache is working. See the API documentation for full details.

- The service removes expired entries only when you try to access them. If it's important for you to save space and remove *all* expired cache entries, you might want to call the Diligence.Cache.prune method regularly. You can do this in your "crontab" file. Here's an example of doing so every 15 minutes:

```
/15    <% document.executeOnce('/diligence/service/cache/'); new Diligence.Cache('result')
```

Linkback Service

"Linkbacks" are a way to add cross-referencing to hyperlinks: if I link to another page on another site, I can let that other site know that I am linking to it, and then that other site can choose to display a link back to my site. This can be useful for users, as it lets them quickly find relevant sites. But, it's probably more important in terms of SEO: the more links you have, the higher your page's rank will be in search engines. And if you can get a link to your site on a popular site, all the better.

Because linkbacks require trust and mutuality, there are especially popular in the blogosphere, where bloggers often work with each other (sometimes antagonistically!) to create more hits, and thus generate more revenue.

Unfortunately, there's no single standard for linbacking, and all of them are rather cumbersome. Luckily, Diligence does most of the work for you: it features clients and servers for both the Trackback and Pingback specs. As a server, it lets you accept these linkbacks from other sites, respond properly to the remote clients, and register the linkback in a MongoDB collection. As a client, it lets you auto-discover trackback and pingback URLs on remote pages, and do the necessary handshaking.

Pingback is by far the more complicated spec: it requires XML-RPC (we are using Diligence's RPC service for it), and also suggests that you make sure that the other site is indeed linking to you before registering. Trackback is more lightweight, but allows telling the target site more information about how you are linking them.

Usage

Make sure to check out the API documentation for Diligence.Linkback.

Integrating Linkbacks into Your Product

Diligence does a lot for you, but the burden is still on your to understand these non-trivial technologies well enough to integrate them properly into your application. On this page, we're featuring a rather elaborate example of how linkbacks work on this page for the purpose of demonstration. Much of this can be automated for your application: for example, in a blogging application, you might want to go over every new blog post and try out all the links on the page with Diligence.Linkback.discover to see if they support linkbacks, and then to do the linkback automatically without any user interaction. Or, you might prefer to have users explicitly click on a "linkback" feature. Diligence gives you the tools, making it as easy as possible for you to do the rest.

How to Linkback *from* This Page?

1. Link First, we need to make sure that we actually have a link to the remote site on our page. Here's a really simple form that lets you add links to this page:

```
<form id="add" method="POST">
<p>
        <%= Diligence.HTML.input({name: 'addPageUri', size: 70}, {_content: 'Page URL:'}) %>
</p>
<p>
        <%= Diligence.HTML.submit({value: 'Add Link'}) %>
</p>
</form>
<form id="clear" method="POST">
        <input type="hidden" name="clearPageUris" value="true" />
        <p>
                <%= Diligence.HTML.submit({value: 'Clear Link List'}) %>
        </p>
</form>
```

```
<p>
        Currently linked pages:
<% for (var i = links.iterator(); i.hasNext(); ) { var link = i.next(); %>
        <a href="<%= link %>">link</a>
<% } %>
</p>
```

2. Auto-Discovery We support auto-discovery of trackback and pingback URLs, so you can first try to just enter the linked URL. Make sure it's one of the links you've added above! Pingback will be preferred if both Trackback and Pingback are supported by the page.

3. Or Use Explicit Linkback URLs In case that doesn't work, you might also have to enter an explicit trackback or pingback URL posted on that page:

```
<p>
        <%= Diligence.HTML.input({name: 'trackbackUri', size: 70}, {_content: 'Trackback URL:
</p>
```

(Note that you do not need to enter the page URL with trackback, but you do need it it with pingback)

```
<form>
<p>
        <%= Diligence.HTML.input({name: 'pingbackUri', size: 70}, {_content: 'Pingback URL:'}
</p>
<p>
        <%= Diligence.HTML.submit({value: 'Linkback'}) %>
</p>
</form>
<% if (message) { %>
<p>
<span style="color: red;"><%= message %></span>
</p>
<% } %>
```

How to Linkback *to* This Page?

This page contains information about its trackback and pingback URLs. In case your software doesn't support auto-discovery of these, and you need to enter them explicitly, they are:

```
<p>
        <%= Diligence.HTML.input({value: Diligence.Linkback.getTrackbackUri(conversation.refe
</p>
<p>
        <%= Diligence.HTML.input({value: Diligence.Linkback.getPingbackUri(), readonly: 'read
</p>
```

Nonces Service

This is a straightforward implementation of number-used-once, or "nonce," using MongoDB atomic operations.

It allows you to issue a unique number, which you can then "check." The check will work once *and only once* for any issued nonce, across all nodes accessing the same MongoDB database. Furthermore, every issued nonce is given an expiration time, after which it will be considered invalid.

Nonces are often used in authentication schemes, where tokens, meant to be used only once, are purposely issued for short time periods in order to minimize security risks.

Usage

Make sure to check out the API documentation for Diligence.Nonces.

The API is very simple. To issue a 60-second nonce:

```
document.executeOnce('/diligence/service/nonces/')
var nonce = Diligence.Nonces.create(60 1000)
```

To check a nonce:

```
if (!Diligence.Nonces.check(nonce)) {
        print('Your token is invalid! Perhaps it was expired? Try logging in again.')
}
```

Note that the nonces used in the API are *strings*, which are hexadecimal representations of big integers. Strings are preferable in this use case, because you can be certain that precision will not be lost across various conversions and serializations. If you really need a non-hexadecimal representation, you can convert it a nonce using the following:

```
var nonceInteger = new java.math.BigInteger(nonce, 16)
print(nonceInteger) // this will print a decimal representation of the nonce
```

Configuration

The service removes expired nonces only when you check them. If it's important for you to save space and remove *all* expired nonces, you might want to call the Diligence.Nonces.prune method regularly. You can do this in your "crontab" file. Here's an example of doing so every 15 minutes:

```
/15   <% document.executeOnce('/diligence/service/nonces/'); Diligence.Nonces.prune(); %>
```

Notification Service

Sending out email from your application can quickly become difficult to manage when you have hundreds of thousands of emails to send out. But Diligence's Notification Service is here to help! Some key features:

- The implementation is optimized for high concurrency, making good use of MongoDB's atomic update features. This means that it's easy to scale: you can have many nodes all sending queued notices at the same time. They won't interfere with each other and there's no fear of having the same email sent more than once.
- It supports subscription channels: you can send a notice to the channel, and it would then be sent to all subscribers. This greatly minimizes the load on MongoDB. Moreover, you can use a notice template such that each subscriber gets a personalized email. Of course, you can also send direct notices to a single addressee.
- Automatic handling of daily and weekly digests for subscribers who prefer not to get individual emails. This works by merging notices into a digest document at scheduled times.
- You don't have to use email: the service implementation is pluggable, allowing you support other kinds of mailboxes if they make sense. For example, you might want to have an internal messaging feature for your application. The implementation is configured per subscriber, so you can support different kinds of mailboxes quite transparently.
- Supports both plain text and mixed-media HTML email.

Note that Diligence connects to but is *not* itself an SMTP server. SMTP servers are complex beasts in their own right: they must handle errors and retries, queuing of outgoing messages, as well as incoming ones if they are configured for relaying or for mailboxes. It's a good idea to keep that separate from your main application. We like Postfix, a mature SMTP server that offers excellent scalability and security.

If you want your application to *receive* email, which is quite a different task than relaying it onward, then we can recommend the SubEtha SMTP library. If there's interest, we may incorporate it into Diligence directly in the future.

Usage

Make sure to check out the API documentation for Diligence.Notification.

Here's an example of two ways for queuing a notice, the first by a direct address, and the second to all subscribers of a channel:

```
document.executeOnce('/diligence/service/notification/')
Diligence.Notification.queueForAddress('Email', 'email@myorg.org', {subject: 'The Subject', t
Diligence.Notification.queueForChannel('main', {subject: 'The Subject', text: 'The content.'}
```

The first option doesn't require any subscription: it uses "Email" as the implementation (see "configuration," below), with the second argument being an identifier for that implementation (in this case, simply an email address). The second option queues the notice on the channel named "main". To add a subscription, you can do the following:

```
Diligence.Notification.subscribe('main', {service: 'Email', address: 'email@myorg.org', mode:
```

The "mode" key can be "immediate", "daily" or "weekly", with the latter two modes for digests. You don't need to create the channel itself: adding at least one subscription will automatically do that.

In the above examples we've sent plain text emails. To add HTML, add an "html" key. Note that if you use "html" you need to *also* add "text" to specify the plain text version. This is very good practice: not all email clients support HTML, and if they don't your HTML will be unreadable without a plain text fallback.

It might be useful to make use of the Sincerity.Mail.MessageTemplate class, which lets you store messages in text packs. For more information on text packs, see the Internationalization Service.

Configuration

In your application's "settings.js" you want to make sure to enable lazy configuration:

```
document.executeOnce('/prudence/lazy/')
```

And then add something like this to your app.globals:

```
app.globals = {
        ...
        diligence: {
                service: {
                        notification: {
                                services: {
                                        '.': Prudence.Lazy.build({
                                                Email: {
                                                        dependencies: '/diligence/service/not
                                                        name: 'Diligence.Notification.EmailSe
                                                        config: {
                                                                from: 'myaddress@mymail.org',
                                                                site: 'Diligence Example'
                                                        }
                                                }
                                        })
                                }
                        }
                }
        }
}
```

Note the use of Prudence.Lazy.build: this allows the Notification Service to lazily create the email implementation on demand during runtime. The key, "Email", will be used in subscriptions, as in the examples above. Note that it is case-sensitive. Within the lazy configuration, the "name" key is the class to instantiate, the "config" is sent to the class constructor, and values in the "dependencies" key are used for "document.executeOnce". Also note the use of the "." key to avoid flattening of the resulting lazy build (see Sincerity.Objects.flatten).

If you want to write your own service implementations, see the source code for the Diligence.Notification.EmailService.

To set up the background tasks for sending out queued notices, add something like the following to your application's "crontab":

```
<% document.executeOnce('/diligence/service/notification/'); Diligence.Notification.sendQu
4 <% document.executeOnce('/diligence/service/notification/'); Diligence.Notification.sendQ
5 0 <% document.executeOnce('/diligence/service/notification/'); Diligence.Notification.sen
```

The above will check for and send regular notices every minute, send daily digests at 4am, and send weekly digests every Sunday at 5am. As stated above, you can have this same "crontab" running on many nodes. Because the implementation relies on MongoDB's atomic updates, you can be sure that notices will not be sent more than once.

Progress Service

If you've read Prudence's Scaling Tips article, you know that for potentially long-running tasks you want to release web request threads as soon as possible, and notify the user in some way as to when the task is finished. This service helps you do exactly that.

For a use case example, consider an application that searches for flight information using several databases and services. The search can take many seconds, if not minutes! Of course, you do not want to hold up a web request thread and have the browser spin while the search is going on, so you turn to Diligence's Progress Service.

It works like this: you create a "process," which is stored in a MongoDB document, and you can asynchronously mark when certain "milestones" are completed, including the final completion of the whole process. Processes can be associated with a user, which allows you to use the authorization service (page 6) to allow only that user access to the process' status, and also to allow the user to query all processes associated with them.

The service supports two ways of letting the user know the status of the process. The first is for short-term processes: a drop-in fragment that simply shows the current status of the process and uses browser JavaScript to refresh the page every few seconds. The user would see milestones along the way to completion, if there are any, and eventually be redirected to another page when the process completes (or fails!).

For longer running processes, you cannot expect the user to wait in front of the web browsers. In these cases, the Progress Service uses the notification service (page 28) to notify the user about milestones, success and failure. Additionally, we provide a drop-in fragment that would allow the user to see the current state of the process on the web, and another one that lets the user access all processes associated with them.

Usage

Make sure to check out the API documentation for Diligence.Progress.

Trivial Example

This fake process will simply do nothing until its expiration:

```
document.executeOnce('/diligence/service/progress/')

var process = Diligence.Progress.startProcess({
        description: 'Searching for your flights...',
        maxDuration: 20 1000,
        redirect: conversation.reference
})

process.redirectWait(conversation, application)
```

That final redirectWait call will send the user to a "please wait" page which will show "Searching for your flights..." as the text, and have a progress bar. The page will automatically refresh and show ongoing progress. After 20 seconds of this, it will redirect back to this page. Note that you can specify different redirect URIs for success, error, timeouts, etc.

The "please wait" page is in "/diligence/service/progress/wait/". If you don't have it in your "/fragments/" then a default page will be used, which is in your container's "/libraries/prudence/" directory. You can use that as a template for your own custom page.

Example with Milestones

You can launch a task from within startProcess, which in turns call the Prudence.Tasks API:

```
var searchString = 'flight #1234'
var process = Diligence.Progress.startProcess({
        description: 'Searching for your flights...',
        maxDuration: 60 1000,
        redirect: '/flight/results/',
        task: {
                name: '/flight/search/',
                searchString: searchString, // this is our custom field
                distributed: true
        }
})
```

Our "/libraries/flights/search.js" would look like this:

```
document.executeOnce('/diligence/service/processing/')

var process = Diligence.Progress.getProcess()
if (process && process.isActive()) {
        var task = process.getTask()
        var milestone = process.getLastMilestone()
        switch (milestone.name) {
                case 'started':
                        process.addMilestone({name: 'ours', description: 'Searching our fligh
                        var found = searchOurDatabase(task.searchString)
                        if (found) {
                                process.addMilestone({name: 'done'})
                        } else {
                                Prudence.Tasks.task(task)
                        }
                        break
                case 'ours':
                        process.addMilestone({name: 'partners', description: 'Searching our p
                        var found = searchPartnerDatabases(task.searchString)
                        if (found) {
                                process.addMilestone({name: 'done'})
                        } else {
                                process.addMilestone({name: 'failed'})
                        }
                        break
        }
}
```

Notes:

- The "Diligence.Progress.getProcess()" API works here only because we launched the task from within startProcess. (It works by putting the process ID in the task context.)
- The first milestone is always "started", and the last one is always "done". The name "failed" is reserved for failed processes, and like "done" will mark the process as inactive. Otherwise, you can set any milestone name you wish.
- You'll also see that we've handled each milestone as a new execution of the task. "process.getTask()" returns a copy of the arguments sent to the last Prudence.Tasks.task call, so we can simply call it again with the same arguments.
- Breaking up our work into separate tasks allows for better concurrency: we're not holding on the thread at once longer than makes sense. Also note that if the task is distributed, each milestone could be executed in a different node in the cluster.

- This method and also makes sure that a milestone will not be executed if a process expires (isActive would return false).

Reattempts

A common use case for the processing service is in dealing with an unreliable action that might actually succeed after a few attempts. You'd thus want to let the user wait until a certain maximum duration, and keep retrying every few seconds in the background until the action succeeds.

The Progress Service automates much of this using the "maxAttempts" key in "task":

```
var ipAddressOfRemoteLocation = '1.2.3.4'
var process = Diligence.Progress.startProcess({
        description: 'Attemping to connect you to remote location {0}...'.cast(ipAddressOfRem
        maxDuration: 5 60 1000,
        redirect: '/remote/connected/',
        task: {
                name: '/remote/connect/',
                maxAttempts: 10, // for reattempts
                delay: 5000, // between reattempts
                remoteLocation: ipAddressOfRemoteLocation // this is our custom field
        }
})
```

Our "/libraries/remote/connect.js" would look something like this:

```
document.executeOnce('/diligence/service/progress/')
var process = Diligence.Progress.getProcess()
if (process) {
        process.attempt(function(process) {
                document.executeOnce('/mylibrary/connections/')
                return connectRemote(process.getTask().remoteLocation)
        })
}
```

Notes:

- The process.attempt call doest most of the work: it makes sure to call the task again if there's still time before the process expires and the maximum number of attempts has not been exceeded, waiting the appropriate delay before each attempt. Your function just has to make sure to return true if the attempt has succeeded.
- Each attempt will get a milestone name in the form of "attempt #X" where X starts at 1.
- If the maximum number of attempts has been reached, the milestone will be set to "failed".
- Reattempts are logged, to help you debug problems.

REST Service

The REST Service makes it easy to create a RESTful API layer over your MongoDB database. It's powerful enough that it may be in itself the primary reason why you wish to use Diligence.

While there are tools to do this automatically—and the REST Service does have an automatic mode, too—the true power of this service is in its customizability. You can insert your own code anywhere in the resources to do special processing, for anything from data validation, through constraint enforcement, to security authorization and high-level business logic.

Moreover, the Prudence platform lets you access this RESTful layer internally, without any HTTP communication or serialization, so that you can use this layer as your primary data access layer API, both internally and for other services. There's no reason to create a separate API for internal vs. external use. This architecture also makes it trivial to separate your data processing nodes from your application logic nodes, should you ever want to do so.

Even without customization via code, out of the box you get the following features:

- The default format immediately supports Ext JS's RESTful data stores. Attach any MongoDB collection to an editable grid widget in a web browser! See the Sencha Integration manual for more information.
- Automatic content negotiation with support for JSON and XML formats, as well as a human-readable HTML format perfect for debugging via browsers. The HTML format even allows simple editing of your content. (Note, though, that if you want a full-fledged web frontend for your MongoDB data, you're better off with MongoVision, which is easily installable side-by-side with your Diligence application.)
- Pagination for traversing collections of any size.
- Choose which document fields you want to expose, and extract sub-documents from your main document.
- Apply straightforward "modes," which let you transform MongoDB's extended JSON format into simpler primitives. For example, "{$date: 1234}" would become "1234".

There are a lot of details below, but you shouldn't be intimidated by them. You do not have to learn every single feature of the REST Service in order to use it. In just a few lines of code, you can setup a whole RESTful layer automatically that will "just work" for many use cases.

Setup

Make sure to check out the API documentation for Diligence.REST.

Manual Setup

We'll start with manual configuration, because it will help you better understand how the REST Service works.

First, let's configure the URI-space in your application's "routing.js". Add the following to app.routes and app.dispatchers:

```
app.routes = {
        ...
        '/data/users/{id}/': '@users',
        '/data/users/':      '@users.plural'
}

app.dispatchers = {
        ...
        javascript: '/manual-resources/'
}
```

We can now configure our resources in "/libraries/manual-resources.js":

```
document.executeOnce('/diligence/service/rest/')

resources = {
        ...
        users:       new Diligence.REST.MongoDbResource({name: 'users'}),
        'users.plural': new Diligence.REST.MongoDbResource({name: 'users', plural: true})
}
```

Automatic Setup

The REST Service can do all the above automatically for you, which is especially useful if you have lots of collections, or if you keep adding collections and want resources for them to be added automatically. Note that this automation does not occur dynamically while your application is running: you have to restart for this to work.

In your application's "routing.js".

```
MongoDB = null
document.execute('/mongo-db/')

document.executeOnce('/diligence/service/rest/')
```

```
app.routes = {
        ...
}

Sincerity.Objects.merge(app.routes, Diligence.REST.createMongoDbRoutes({prefix: '/data/'}))
```

> Important! The first two lines of code make sure that MongoDB is re-initialized before proceeding, so that we can be sure to avoid using the default MongoDB initialization in other applications. This is good practice when using Diligence in any initialization script.

In "/libraries/resources.js", we just need this:

```
document.executeOnce('/diligence/service/rest/')

resources = {
        ...
}

Sincerity.Objects.merge(resources, Diligence.REST.createMongoDbResources())
```

You can also specify exactly which collections you want created:

```
Diligence.REST.createMongoDbResources({collections: ['users','notices','documents']})
```

Custom Queries

Sometimes you may be using a single MongoDB collection as a container for documents of several different types, and you would want them exposed as a separate URI-space.

The REST Service allows for this via a simple querying language. To illustrate it, lets first look at what the default query is for singular resources, if no query is provided by you:

```
resources = {
        ...
        users: new Diligence.REST.MongoDbResource({
                name: 'users',
                query: {_id: {$oid: '{id}'}}
        })
}

app.routes = {
        ...
        '/data/users/{id}/': {type: 'implicit', id: 'users'}
}
```

The "query" key is in MongoDB's extended JSON format, and is used for the MongoDB "find" operation. The values are all cast using the conversation.locals, which, if you remember how to do Prudence routing, are extracted from the URI template. Let's look at this slowly:

1. If a "/data/users/123/" URI is accessed with a GET operation, the "123" will be extracted from the URI template. The effect will be as if we called:

   ```
   conversation.locals.put('id', '123')
   ```

2. All the values in our resource's "query" value are cast using conversation.locals. So, our final query will be:

   ```
   {_id: {$oid: '123'}}
   ```

3. The REST Service will use the above query for a "find" operation:

```
var data = collection.findOne({_id: {$oid: '123'}})
```

(Note that the "$oid" in MongoDB's extended JSON becomes an ObjectId in BSON.)

Knowing this, you can then set the "query" any way you like. You can use values extracted from conversation.locals, or any literal value. For example, let's create a URI-space for users of type "admin", to be accessed :

```
resources = {
        ...
        admins: new Diligence.REST.MongoDbResource({
                name: 'users',
                query: {name: '{name}'}, {type: 'admin'}}
        }),
        'admins.plural': new Diligence.REST.MongoDbResource({
                name: 'users',
                query: {type: 'admin'},
                plural: true
        })
}

app.routes = {
        ...
        '/data/admins/{name}/': {type: 'implicit', id: 'admins'},
        '/data/admins/':    {type: 'implicit', id: 'admins.plural'}
}
```

As a convenience, you can also add custom values to be cast using the "values" key. These will be merged with values from conversation.locals:

```
new Diligence.REST.MongoDbResource({
        name: 'users',
        query: {name: '{name}'}, {type: '{type}'}},
        values: {type: 'admin'}
})
```

This allows for nice reusability when you create your own extended classes: you can share one query among many subclasses.

Custom Extraction

By default, the REST Service will extract and return the entire MongoDB document, but you can customize this quite powerfully, even to allow you to access sub-documents inside a document.

First off, you can simply choose the fields you want:

```
new Diligence.REST.MongoDbResource({
        name: 'users',
        fields: ['name', 'email', 'address']
})
```

The "fields" key will be used at the level of MongoDB's driver, so that unused data won't even be retrieved from the database.

You can go further and extract sub-fields:

```
resources = {
        ...
        'users.email': new Diligence.REST.MongoDbResource({
                name: 'users',
                fields: 'email',
                extract: 'email'
        }
```

```
})

app.routes = {
        ...
        '/data/users/{id}/email': {type: 'implicit', id: 'users.email'},
}
```

The result of a GET would be only a string of the email address. An example in JSON:

```
"myemail@mail.org"
```

Without the "extract", the representation would be this:

```
{
        "_id": {
                "$oid": "4e057e94e799a23b0f581d7d"
        },
        "email": "myemail@mail.org"
}
```

> Important! Not all client JSON parsers can deal with JSON data that is not a dict or an array. If you are extracting data that is not a dict or an array, you may need to implement your own special parsing.

With "extract" you can go further and even provide an array that will be extracted in order. For example:

```
resources = {
        ...
        'users.groups': new Diligence.REST.MongoDbResource({
                name: 'users',
                fields: 'authorization',
                extract: ['authorization', 'entities']
        }
})

app.routes = {
        ...
        '/data/users/{id}/groups': {type: 'implicit', id: 'users.groups'},
}
```

The above actually uses the data structure used by Diligence's Authorization Service to retrieve the security groups. The result of a GET would be an array. An example in JSON:

```
["users", "admins"]
```

Finally, you can do your own custom extraction, by providing a function:

```
new Diligence.REST.MongoDbResource({
        name: 'users',
        fields: 'authorization',
        extract: function(doc) {
                return doc.authorization.entities.join(',')
        }
})
```

Custom Modes

You can set up your own custom modes like so:

```
new Diligence.REST.MongoDbResource({
        name: 'users',
        modes: {
                flat: function(data) {
```

```
                return Sincerity.Objects.flatten(data)
            }
        }
})
```

See "Usage" below for information on how to use modes.

Overriding

There are two ways to override the default behavior: 1) inherit the Diligence.MongoDbResource class using the Sincerity.Classes API, or 2) monkey-patch the instances. The former method is more reusable, but the latter method works just as well and is easier if you just need to customize a single resource. Example of monkey-patching:

```
resources = {
        ...
        users: new Diligence.REST.MongoDbResource({name: 'users'})
}

resources.users.doDelete = function(conversation) {
        ...
        // Call overridden method
        arguments.callee.overridden.call(this, conversation)
}
```

Using this method you can even monkey-patch instances created automatically after a call to "Diligence.REST.createMongoDbResources()".

In-Memory Data

The REST Service does not have to use MongoDB to store data: it also supports storing data in memory, even shared memory distributed in the Prudence cluster.

This is useful if you don't need persistent storage in MongoDB (the data is considered volatile) and is also useful for creating mock data for testing. The URI-space otherwise behaves exactly the same as if it were attached to MongoDB collections. Performance, of course, should be better than if you were accessing MongoDB. On the other, your storage size is limited to your RAM. So, while this feature is not a replacement for using MongoDB, it can be quite useful in various scenarios.

Let's modify our example from above to use in-memory resources:

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

var users = {
        '4e057e94e799a23b0f581d7d': {
                _id: '4e057e94e799a23b0f581d7d',
                name: 'newton',
                lastSeen: new Date()
        },
        '4e057e94e799a23b0f581d7e': {
                _id: '4e057e94e799a23b0f581d7e',
                name: 'sagan',
                lastSeen: new Date()
        }
}

var usersMap = Sincerity.JVM.toMap(users, true)

resources = {
        ...
        users:       new Diligence.REST.InMemoryResource({name: 'users', documents: usersMap}),
        'users.plural': new Diligence.REST.InMemoryResource({name: 'users', documents: usersM
}
```

Note that we translated the "users" dict into a thread-safe JVM map. We could have also just sent the "users" dict directly to the "InMemoryResource" constructor, which can create the map for us. But, since we have *two* resources, the singular and the plural, and we want them to share the same map, we have created this map ourselves.

What if you're in a Prudence cluster, and want all nodes to share the same in-memory data? Let's modify our code:

```
resources = {
        ...
        users:          new Diligence.REST.DistributedResource({name: 'users', documents: users}),
        'users.plural': new Diligence.REST.DistributedResource({name: 'users', documents: use
}
```

The code is even simpler than the "InMemoryResource" code (no need to create "usersMap"), but requires some explanation:

- The "name" field will be used as the name of the Hazelcast map. You can configure this map by name in the Hazelcast configuration, otherwise it will use the Hazelcast defaults for new maps.
- The data from the "documents" field will be copied into the Hazelcast *only once* and *only if the map is already empty*. Thus, it should be thought of as your initialization data: the first time a resource is set up for that map, from anywhere in the cluster, this data will be copied in. From then on, for the life of the cluster, "documents" will be ignored. Thus, if you want to re-initialize the map, you will need to either restart your whole cluster, or programmatically set the data. (The Diligence Console would be very useful for that.)
- Note that we are serializing data using JSON into the distributed map. The performance hit should be minimal, but it's important to remember that only your data must be extended-JSON-compatible. (The "InMemoryResource" doesn't have this restriction.)

Usage

Resource Characteristics

All resources support the following URI query parameters:

- **format**: You can use this to specify the exact format you want, overriding any HTTP content negotiation. This is useful for testing and debugging, but can also help you in dealing with HTTP clients that can't easily set headers. Accepted values are "json", "xml" and "html". Note that when accessing resources internally, no serialization happens, and "format" is unnecessary.
- **human**: Setting this to "true" will further help your debugging, as it will return nicely indented, multiline JSON or XML representations.
- **mode:** "Modes" are simple functions that are applied to all documents in order to transform the final representation. The REST Service comes with a few useful modes, but you can easily create your own, just make sure to hook them to the instance using the "modes" key. The query parameter value will be mapped to a key in this dict. Note that you can provide multiple "mode" values, in which case all mode functions will be called in order. Provided modes:
  - **primitive:** This converts MongoDB extended values into simpler JSON structures. For example, "{timestamp: {$date: 12345}}" will become "{timestamp: 12345}".
  - **string**: This converts all JSON values into strings. It's a good way to overcome various number accuracy issues, especially when dealing with PHP clients.
  - **stringid:** Converts only the "_id" field to a string, in case it's a BSON ObjectId. Some clients, such as Ext JS, cannot deal with ID values that are dicts.

An example URI with all the above parameters:

```
/data/users/4e057e94e799a23b0f581d7d/?format=json&human=true&mode=primitive&mode=string
```

As for payloads, in POST and PUT operations, note that by default they must be in JSON, even if you are representing the result in XML or HTML. The reason is that there is no obvious way to translate XML to the final JSON format needed by MongoDB. If you do need to support XML payloads, you can override "handlePost" and "handlePut" to do this yourself according to your specifications.

Singular Resources

The REST Service will by default extract the "{id}" pattern in the URI into a MongoDB ObjectID for the document "_id" field. For example, if your route is "/data/users/{id}/", then "/data/users/4e057e94e799a23b0f581d7d/" would refer to the user document with that "_id."

Requests to the URI always return 404 if the document does not exist. Further notes:

- **POST:** All keys of the payload will be used for a "$set" in a MongoDB "findAndModify" operation, and the modified document will be returned. If you include an "_id" key in the payload it will be removed, because the ID in the URI takes precedence.
- **PUT:** The payload will become a simple MongoDB "save" operation, which is an upsert, meaning it would either create a new resource or replace the existing one. If you include an "_id" key in the payload it will be removed, because the ID in the URI takes precedence. Note that if you want to create a new resource, it's up to you to make sure the the id is unique, otherwise you will get an HTTP 409 error (conflict). You can generate a unique ID by calling MongoDB.newId(). Example for generating a unique URI using templates:

```
'/data/users/{0}/'.cast(MongoDB.newId())
```

Plural Resources

The plural resource is a bit more complex. The returned representations include a "total" key, counting the size of the collection, and a "documents" key, containing an array of specific documents. For example:

```
{
        "total": 1092,
        "documents": [
                {"_id": {"$oid": "4e057f2ae799a23b0f581d7f" }, ... }
                ...
        ]
}
```

The following additional query parameters are supported for pagination, controlling which documents are included in the "documents" array:

- **start:** The index from which to start collecting documents. By default it will be 0.
- **limit:** The maximum number of documents to return.

The "documents" array can definitely be empty if your "start" and "limit" values are not satisfied.

Further notes:

- **POST:** This lets you update many documents at once. Your payload should be an array of values that would be sent via the singular resource POST, as described above, *however* you must also include an "_id" for each value. The response will include all documents after their modification.
- **PUT:** This is how you add documents to your MongoDB collection. Simply provide an array of values, and they will become MongoDB "insert" operations. The response will include "_id" fields on all your documents, if you did not set them yourself.
- **DELETE:** This is a MongoDB "remove" operation, *not* a "drop".

Accessing Your Resources over the Web

All your resources support the HTML format, so you can easily access them via a web browser. For example, this link: http://localhost:8080/diligence-example/data/users/4e057e94e799a23b0f581d7d/.

This view supports simple editing of your resources: you can POST, PUT any resource using JSON or XML payloads, or DELETE them. It's a great way to test and debug your resources.

You can customize this view as you please: just create "/diligence/service/rest/singular.html" and "/diligence/service/rest/plural.html" files in your "/fragments/" directory. You can start with the default files under your container's "/libraries/prudence/" directory as a template.

Accessing Your Resources with the API

The Prudence.Resources API makes it very easy to access your resources, whether internally or on a different node. See the API documentation for full details, otherwise here we'll provide you with a quick tutorial for using it with the REST Service.

Let's start with the internal use case:

```
document.executeOnce('/prudence/resources/')
var user = Prudence.Resources.request({
        uri: '/data/users/4e057e94e799a23b0f581d7d/',
        internal: true
})
print(user.name)
```

> Again, we'll emphasize that when accessing the API internally neither HTTP nor serialization are involved. The data is never converted to JSON, instead it's extracted directly from MongoDB's BSON to JavaScript's internal data structure, exactly as if you were using the MongoDB API directly. There's obviously some overhead added by the Prudence platform and the REST Service, but it should be very minimal, especially when compared to the network fetch from MongoDB. In short, performance concerns should not stop you from using the REST Service in this fashion.

Accessing remote resources is almost identical, though obviously HTTP and JSON (or XML) are involved. As an example, we can try to access our local resource via HTTP:

```
var user = Prudence.Resources.request({
        uri: 'http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/',
        mediaType: 'application/json'
})
print(user.name)
```

Of course, the URI can point to anywhere on the network, or the Internet. Note that we had to explicitly specify our preferred media type, because our resource supports several different formats.

The API can be used for all REST methods:

```
var user = Prudence.Resources.request({
        uri: '/data/users/4e057e94e799a23b0f581d7d/',
        internal: true,
        method: 'post',
        payload: {
                value: {email: 'newemail@mysite.org'}
        }
})
```

Remotely, the REST methods are actual HTTP verbs:

```
var user = Prudence.Resources.request({
        uri: 'http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/',
        mediaType: 'application/json'
        method: 'post',
        payload: {
                type: 'json',
                value: {email: 'newemail@mysite.org'}
        }
})
```

We'll finish off this short tutorial by showing you that for every request you can also set query params:

```
var users = Prudence.Resources.request({
        uri: '/data/users/',
        internal: true,
        query: {
```

```
                start: 5,
                limit: 3
        }
})
print(users[0].name)
```

Accessing Your Resources with cURL

cURL is an HTTP command line tool based on the cURL library, available for a great many Unix-like operating systems as well as Windows. It's especially useful for testing RESTful APIs. Here's a quick tutorial to get you started with using cURL with the REST Service.

First, a few GET commands to try:

```
curl "http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/?human=true"
curl "http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/?format=xml&human=true"
curl "http://localhost:8080/myapp/data/users/?limit=3&human=true"
```

You can send a payload using the "-d" switch, which also sets the HTTP verb to POST. For example, this will modify the email of a user:

```
curl -d '{"email":"newemail@mysite.org"}' "http://localhost:8080/myapp/data/users/4e057e94e79
```

When using "-d", you can also start your payload with "@" to signify that you want to send the contents of a file, in this case "data.json":

```
curl -d @data.json "http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/?human=tr
```

To set the HTTP verb explicitly, use "-X". Here we'll create a new user:

```
curl -X PUT -d @data.json "http://localhost:8080/myapp/data/users/?human=true"
```

And now we'll delete a user:

```
curl -X DELETE "http://localhost:8080/myapp/data/users/4e057e94e799a23b0f581d7d/"
```

With the "-h" switch, you can also send HTTP headers in raw form:

```
curl -H "Accept: application/xml" "http://localhost:8080/myapp/data/users/4e057e94e799a23b0f5
```

Finally, add the "-v" switch to print out the outgoing and incoming headers.

Extension

TODO

Extended MongoDbResource

Extending IterableResource

RPC Service

The RPC (Remote Procedure Call) Service provides robust, elegant support for various versions of the JSON-RPC and XML-RPC specifications, including support for batch processing for JSON-RPC 2.0. It's powerful enough that it may be in itself the primary reason why you wish to use Diligence.

In most cases, all you need to do is hookup your JavaScript functions to a URI, and let the RPC Service do the rest. All error codes, system APIs and type conversions will be properly handled.

As a bonus, the RPC Service also includes a nice client utility for calling JSON-RPC and XML-RPC.

> Is RPC a good idea? We're inclined to say: no. REST is a much more scalable and robust pattern, all things considered. REST uses all the power of HTTP to provide client-cacheable representations. RPC, on the other hand, supports only HTTP POST, the only *non*-idempotent HTTP operation, which can never be cached. However, RPC may be necessary for communication with other services and clients, so you might not have a choice. And, sometimes, it's just the most straightforward, quick-and-dirty solution to a problem. Especially with the Diligence RPC Service, it's so easy to just allow clients to call functions on the server, that sometimes you might prefer it to designing a RESTful URI-space. So be it! Just make sure you understand the pros and cons of you choice.

Setup

Make sure to check out the API documentation for Diligence.RPC.

First, let's configure the URI-space in your application's "routing.js". Add the following to app.routes and app.dispatchers:

```
app.routes = {
        ...
        '/calc/': '@calc'
}

app.dispatchers = {
        ...
        javascript: '/manual-resources/'
}
```

We can now configure our resources in "/libraries/manual-resources.js":

```
document.executeOnce('/diligence/service/rpc/')

var Calc = {
        multiply: function(x, y) {
                return x y
        }
}

resources = {
        ...
        calc: new Diligence.RPC.Resource({namespaces: {Calc: Calc}})
}
```

And... that's pretty much it! You can now call your methods using JSON-RPC or XML-RPC.

Namespaces

The key of the namespace is prefixed with a period before all method identifiers. So, our method above would be identified as "Calc.multiply".

However, if you do not want this prefix, you can use the special "." key, which here means the root namespace:

```
resources = {
        ...
        calc: new Diligence.RPC.Resource({'.': Calc})
}
```

The method would now be identified simply as "multiply".

If you don't need the namespaces feature at all, you can use the following shortcut (note the "namespace" key, singular):

```
resources = {
        ...
        calc: new Diligence.RPC.Resource({namespace: Calc})
}
```

Long Form

You have some more control over the exported functions, should you need it. The long form of creating namespaces is like so:

```
var Calc = {
        multiply: {
                fn: function(x, y) {
```

```
                return x y
            },
            arity: 2
        }
}
```

The "artity" key counts how many arguments the function requires. If it's not there, the RPC Service will count them from the function spec. However, this won't work if you access JavaScript "arguments" directly, hence this long form exists.

System Namespace

The "system" namespace is reserved for parts of the RPC protocols. The RPC Service implements these for you:

- **system.getCapabilities**
- **system.listMethods**
- **system.methodSignature**
- **system.methodHelp:** By default, this will just show the method name, but in the long form definition you can add a "help" key to set this as you need.

Scope

When your function is called, the "this" will be automatically populated with the following keys:

- **definition:** Your long-form function definition (short-form function definitions will be expanded into the long form)
- **namespace:** The original namespace object you supplied
- **resource:** The Diligence.RPC.Resource instance
- **conversation:** The Prudence conversation of the call
- **call:** The RPC call object, as sent from the client

The "method" key is useful in that you can add anything you want to the method object. For a rather silly example:

```
var Calc = {
        multiply: {
                fn: function(x, y) {
                        return x y this.definition.multiplyAll
                },
                multiplyAll: 100
        }
}
```

One special key is reserved: "scope". Use it to override "this" to be any value you desire:

```
var Calc = {
        multiply: {
                fn: function(x, y) {
                        return x y this
                },
                scope: 100
        }
}
```

If you are using JavaScript object oriented programming, you might want "this" to always just be the namespace object itself. In that case, you can use the "objects" key instead of the "namespaces" key when creating your Diligence.RPC.Resource constructor. It works the same way as a namespace except that the scope will be the object itself for all method calls:

```
// This is a class
var Calc = function(multiplyAll) {
        this.multiplyAll = multiplyAll

        this.multiply = function(x, y) {
                return x y this.multiplyAll
        }
}

resources = {
        ...
        calc: new Diligence.RPC.Resource({objects: {Calc: new Calc(100)}})
}
```

You can mix "namespaces" and "objects" in the same constructor. Also note that you can also use "object" (singular) in the same way as "namespace" (singular).

Fault Codes

If your function throws an exception, the RPC Service will return a ServerError fault code with the exception string as the message.

However, you can also return specific XML-RPC fault codes (the same code numbers are used by JSON-RPC):

- By throwing a number (all fault codes are negative numbers). You can use the convenient constants in "Diligence.Fault". For example:

  ```
  throw Diligence.Fault.InvalidParams
  ```

- By throwing a dict with both the fault code and the message. For example:

  ```
  throw {code: Diligence.Fault.InvalidParams, message: 'Cannot divide by 0!'}
  ```

Usage

URI Query Parameters

- **type:** The resource will automatically determine whether it should work in JSON-RPC or XML-RPC according to the media type of the incoming payload, or if that's not available, the preferred media type for the returned representation. Unfortunately, some clients don't or can't set either. In that case, you can set the type explicitly in the URI, with either "json" or "xml" as values.
- **human:** Set this to "true" to generate multiline, indented human-readable results (both for JSON and XML). Great for debugging.

Calling RPC with the API

The RPC Service includes a useful RPC client function, "Diligence.RPC.request". It's essentially a wrapper over the Prudence.Resources API that builds the payload for you and nicely unpacks the results. The results will always be in JSON-RPC's format, even if you are using XML-RPC. This allows for uniform processing on your end.

Here's an example of an internal call using JSON-RPC:

```
document.executeOnce('/diligence/service/rpc/')

var result = Diligence.RPC.request({
        uri: '/calc/',
        internal: true,
        name: 'Calc.multiply',
        params: [5, 6],
        id: 'abc',
        protocol: 'json'
```

```
})

if (result.error) {
        print('Error: ' + result.error.message)
}
else {
        print(result.result)
}
```

For XML-RPC, simply set "protocol" to "xml". If not provided, it defaults to "json". Note that the result will also include that "protocol" key you provided, in case you need to know which protocol was used.

Generally, if you have the option to use JSON-RPC, you should prefer it. XML serialization incurs an extra overhead in JavaScript.

Calling RPC with cURL

cURL is an HTTP command line tool based on the cURL library, available for a great many Unix-like operating systems as well as Windows. It's especially useful for testing RESTful APIs. Here's a quick tutorial to get you started with using cURL with the RPC Service.

First, let's create our payload. With a text editor, create a file named "rpc.json" and paste this:

```
{
        "jsonrpc": "2.0",
        "method": "Calc.multiply",
        "params": [2, 3],
        "id": "abc"
}
```

You can send a payload using the "-d" switch, which also sets the HTTP verb to POST. When using "-d", you can also start your payload with "@" to signify that you want to send the contents of a file:

```
curl -d @rpc.json "http://localhost:8080/myapp/calc/?type=json&human=true"
```

You should get this result:

```
{
        "id": "abc",
        "result": "6",
        "error": null,
        "jsonrpc": "2.0"
}
```

Calling RPC from Web Browsers

Many client-side JavaScript frameworks include support for RPC, but if all you need is a straightforward, self-contained library, we recommend jsonrpcjs.

Search Service

Usage

Make sure to check out the API documentation for Diligence.Search.

Serials Service

This straightforward service generates unique integers in a series, using MongoDB atomic operations. No number in a specific series will ever be generated again. This service is thus useful for generating integer IDs.

Note that uniqueness is only guaranteed by the intactness of the MongoDB database. If you somehow lose it and have to start over, there's a chance you would regenerate IDs that have already been used. If you need unique IDs that don't have this limitation, you'll want to use GUIDs instead.

Usage

Make sure to check out the API documentation for Diligence.Serials.

Usage is very simple:

```
document.executeOnce('/diligence/service/serials/')
var id = Diligence.Serials.next('person')
```

Each series is stored as a single document in the "serials" MongoDB collection. By default, the method will create the series document if it does not yet exist, initializing it with the number 1.

Syndication Service

Usage

Make sure to check out the API documentation for Diligence.Syndication.

Links

The module contains a simple /web/fragments/ drop-in that adds links recognizes by all major browsers, and another drop-in for the "syndication" button, using the *de facto* standard icon.

Gravatar Integration

Gravatar is a popular service for managing user avatars and simple profile pages by associating them with email addresses.

It makes users happy, because they can manage their avatars for many, many services in one place. The user's email is hashed so that it is not made publicly available, unless the user chooses to put them explicitly on their profile.

It makes site owners happy, because they can display avatars for users without having to store them or otherwise manage them. Additionally, new users would have their avatar immediately displayed without any effort on their part, and users do not like effort. If you're using the authentication service (page 5) in association with the registration feature (page 61), then you already have an email address for the user, and can immediately fetch their avatar from Gravatar.

Worried about forcing users to use an external service? Then make Gravatar an optional fallback. Provide users with a way to manage avatars on your site directly, and only default to Gravatar.

Usage

Make sure to check out the API documentation for Diligence.Gravatar.

Just enter an email address, and let the Diligence magic happen.

The avatar above is hyperlinked to their Gravatar profile page. And here's the complete JSON dump of their profile:

PayPal Integration

Usage

Make sure to check out the API documentation for Diligence.PayPal.

Sencha Integration

Ext JS and Sencha Touch are both large JavaScript frameworks in their own right, and Diligence supports many of their features. For this reason, we've divided the section for Sencha Integration into several sub-sections. Still, you'll want to start here, where we go over some general usage applicable to all features.

After that, go ahead and read the sections for the following integration features:

- Grids
- Trees
- Charts
- Forms
- Ext Direct

Usage

Make sure to check out the API documentation for Diligence.Sencha.

To include Ext JS in your HTML page, you'll want to insert a scriptlet, resulting in a page template similar to this:

```
<html>
<head>
...
<%
document.executeOnce('/diligence/integration/frontend/sencha/')
Diligence.Sencha.extJsHead(conversation, 'ext-all-gray')
%>
</head>
<body>
...
</body>
<script type="text/javascript">
Ext.onReady(function() {
...
});
</script>
</html>
```

Notes:

- The "extJsHead" method uses "conversation.pathToBase" to make sure that the correct relative URL is inserted. Be aware of this if you intend to cache that fragment for all URLs.
- The second argument is the theme: it will be "ext-all" if not provided.
- This also includes Diligence's Ext JS client-side helper library. You don't have to use it, but it can make your life easier. You can find it under "/libraries/web/scripts/diligence/integration/ext-js.js". The library enhances Ext JS via:
  - JSON Readers and Writers that support MongoDB's extended JSON format. This will allow you to automatically translate $date, $long and other JSON extensions. Even without using MongoDB, this is a very useful format.
  - A data Proxy that automatically uses the extended JSON Reader and Writer, and builds URLs in Diligence's default structure.

Sencha Integration: Grids

Ext JS's grid widget may be its most powerful feature. It supports editing, paging and endless scrolling, with lots of room for customization. Grids offer a familiar and powerful UI for traversing large amounts of structured data. Diligence offers excellent server-side support for this astounding client widget: in a few lines of code, you can hook up an editable grid widget to a MongoDB collection.

Despite being one of Diligence's most immediately impressive features, this is going to be a rather short manual chapter! The reason is that the heavy lifting is done by the REST Service. The URI-space created by the REST

Service is compatible with Ext JS, so there's not much more to do other than hook up the grid using client-side JavaScript.

What we're going to do here is give a quick tutorial for using Ext JS grids with Diligence.

Setup

See the REST Service. Resources created there are immediately attachable to Ext JS grids.

Usage

Make sure to check out the server-side API documentation for Diligence.Sencha and the client-side API documentation for Ext JS.

A full tutorial of Ext JS grids is beyond what we can do in this Manual, but here are is a quick overview of the components as they apply to Diligence:

- You start by creating a "Model" class, which is a template for your "records," represented by your grid rows. Each model has a list of typed fields (the default is a plain string) which imply client-side translation and validation. You can further create your custom fields. For Diligence, it's important that you include the "_id" field and also set "idProperty" to be that field. If you don't explicitly set "idProperty," Ext JS will not be able to save individual records. Also not that Ext JS requires the idProperty to be a primitive, so we are using the "stringid" mode for the Diligence REST Service in order to make sure we get strings, not MongoDB ObjectIds.

- The model also defines a "Proxy," which is Ext JS's extensible connector class. Proxies are in charge of loading and saving the data. In this case, we are using a "diligence" proxy type. This is a custom type that we have defined in Diligence's Ext JS helper library. It's rather simple, and you are free to use the "ajax" proxy type instead with the modifications we've made there. The "diligence" proxy is configured to automatically support MongoDB's extended JSON notation and also use Diligence's URL style. We've additionally set the "root" property for the reader to "documents".

- The "Store" is an intermediary class between the model and the grid. It handles caching of model instances ("records") in memory, paging, pre-fetching, etc. By default it will use the proxy we defined in our model.

- Finally, there's the grid panel. Though we've defined "fields" in our model, we must define "columns" in our grid that map onto the fields. In many cases we'll be doing a one-to-one mapping, but you can create custom columns that transform the model in various ways, for example combining fields into a single column, or having a column that is derived from other fields. You do not have to have a column for every field. (Indeed, you'd likely not want to have the "_id" field visible.)

- By default, the grid is not editable, but we can add the "CellEditing" plugin to handle that. Every column can define its own editor, which can handle user-side validation beyond what is offered by the model. Ext JS comes with many powerful editing widgets, and of course you can create your own.

- In this example, we've also added a paging toolbar to the grid, and hooked it up to use the same store as the grid. As the store is paged by the toolbar, it fires events that update the current grid view.

That should be enough to get you started. Here's how the code looks:

```
<html>
<head>
<%
document.executeOnce('/diligence/integration/frontend/sencha/')
Diligence.Sencha.extJsHead(conversation)
%>
</head>
<body>
        <div id="grid"></div>
</body>
<script type="text/javascript">
Ext.onReady(function() {
```

```
var pageSize = 15;

Ext.define('User', {
        extend: 'Ext.data.Model',
        fields: [
                '_id',
                'name',
                {name: 'lastSeen', type: 'date'}
        ],
        idProperty: '_id',
        proxy: {
                type: 'diligence',
                url: '<%= conversation.pathToBase %>/data/users/'
        }
});

var store = Ext.create('Ext.data.Store', {
        model: 'User',
        pageSize: pageSize,
        autoSync: true,
        autoLoad: true
});

Ext.create('Ext.grid.Panel', {
        store: store,
        columns: [{
                dataIndex: 'name',
                header: 'Name',
                editor: 'textfield'
        }, {
                dataIndex: 'lastSeen',
                xtype: 'datecolumn',
                format: 'm/d/y, H:i',
                header: 'Last Seen',
                editor: {
                        xtype: 'datefield',
                        format: 'm/d/y, H:i'
                }
        }],
        forceFit: true,
        selType: 'cellmodel',
        plugins: [
                Ext.create('Ext.grid.plugin.CellEditing', {clicksToEdit: 2})
        ],
        dockedItems: [{
                dock: 'bottom',
                xtype: 'pagingtoolbar',
                store: store,
        }],
        renderTo: 'grid',
        style: {
                margin: 'auto'
        },
        width: 500,
        height: 370
});
```

```
});
</script>
</html>
```

Sencha Integration: Trees

Ext JS's tree widget is quite powerful, and gives you a lot of control over the visual presentation, supporting complex nodes and multi-column displays. Though it's not in itself editable, it integrates with Ext JS's drag-and-drop model, which you can hook up into your custom editing model. Diligence offers excellent server-side support for it: in a few lines of code, you can hook up a tree widget to a document MongoDB collection, and use MongoDB DBRefs to expand the tree into other documents.

Setup

Make sure to check out the server-side API documentation for Diligence.Sencha and the client-side API documentation for Ext JS.

The Ext JS's tree requires a rather specific JSON data representation, so we've inherited the resource class in the REST Service to support it, with classes "Diligence.Sencha.TreeResource" and "Diligence.Sencha.MongoDbTreeResource." You might want to start by reading the REST Service manual chapter.

In your "/routing.js", add the following to app.routes and app.dispatchers:

```
app.routes = {
        ...
        '/data/textpack/{id}/': '@textpack'
}

app.dispatchers = {
        ...
        javascript: '/manual-resources/'
}
```

Note the "{id}" variable in the URI pattern: the resource expects the node ID to appear there.

Now add a "MongoDbTreeResource" to your "/libraries/manual-resources.js":

```
document.executeOnce('/diligence/integration/frontend/sencha/')

resources = {
        ...
        textpack: new Diligence.Sencha.MongoDbTreeResource({collection: 'textpacks'})
}
```

Custom Queries

In the above example, the "id" segment in the URI will be used for a MongoDB "findOne" operation in the collection for the document "_id", and the entire document (minus the "_id" field) will be used for the tree data. However, Diligence allows you to customize this data search and extraction:

```
resources = {
        ...
        textpack: new Diligence.Sencha.MongoDbTreeResource({collection: 'textpacks', query: {
}
```

The "query" key will be used for the MongoDB "findOne" operation, and the "field" key specifies which field in the document contains the tree data.

Custom Text

By default, the text for each node will be the key for tree folders and the stringified value for tree leaves. But, you can customize this by overriding the "getNodeText" method:

```
resources = {
        ...
        textpack: new Diligence.Sencha.MongoDbTreeResource({
                collection: 'textpacks',
                query: {locale: 'en'},
                field: 'text',
                getNodeText: function(id, node) {
                        return typeof node == 'string' ? id + ': ' + node : id
                }
        })
}
```

The "id" argument is the key, while the "node" argument is null for tree folders or the value for tree leaves.

Data Structure

The expected document data structure is quite straightforward: a series of nested dicts, for which non-dict keys become tree leaves. For example:

```
{
        "_id": {
                "$oid": "4d474457f9e399e7e05e1269"
        },
        "text": {
                "application": {
                        "title": "MyApp",
                        "description": "This is an important application"
                }
        },
        "locale": "en"
}
```

Here, "text" and "application" will both become tree folders, while "title", "description" and "locale" will become tree leaves. The "_id" field will be ignored by "MongoDbTreeResource".

Multi-Document Data Structure

The tree data can be split among several documents using MongoDB DBRefs. Diligence will fetch the referred document and use the "field" key, if it was set, to retrieve a specific field. This can continue recursively to any depth.

Let's add a DBRef (using MongoDB's extended JSON notation, via the "$ref" key) to another document in our collection:

```
{
        "_id": {
                "$oid": "4d474457f9e399e7e05e1269"
        },
        "text": {
                "application": {
                        "title": "MyApp",
                        "description": "This is an important application",
                        "more": {
                                "$ref": "textpacks",
                                "$id": "4d6831f97c6c99e71b8eaf0e"
                        }
```

```
                }
        },
        "locale": "en"
}
```

The DBRef node will appear in the tree as a non-expanded folder so that the user will have to explicitly expand it in order to fetch the nodes underneath. If you require all nodes to be expanded, you can call "expandAll" on the tree after it is loaded.

In-Memory Data

As with the REST Service, you can also avoid MongoDB and create an in-memory tree resource:

```
var textpack = {
        application: {
                title: 'MyApp',
                description: 'This is an important application'
        }
}

resources = {
        ...
        textpack: new Diligence.Sencha.InMemroyTreeResource({tree: textpack})
}
```

Note that there is no distributed version of this, because it's unnecessary: the tree data is read-only, so there's no reason to synchronize the data across the cluster.

Usage

URI-space

Tree widgets are read-only, so the "MongoDbTreeResource" is significantly simpler to implement than "MongoDbResource". It only handles HTTP GET. Moreover, since this resource is designed for Ext JS, it only supports JSON, not XML. The only URI query parameter supported is "human=true", to return multiline, indented JSON representations.

What is a bit more complicated here is the node ID pattern. To support the recursive nature of the tree, the node ID is constructed using the path of the node starting at the root, with "/" as a separator. The root node is simply "/". (These constants are configurable.)

To show how this works, let's lay out all the node IDs from the example data structure provided above:

```
/
/text
/text/application
/text/application/title
/text/application/description
/locale
```

Note that when include the node ID in the URI, you have to URI-encode it. The URI-code for a "/" is "%2f". As an example, let's fetch a node using cURL in the command line:

```
curl "http://localhost:8080/myapp/data/textpack/%2ftext%2fapplication/?human=true"
```

> If you are using Apache to reverse-proxy to your server, you may find that it does not proxy URLs with a "%2f". To solve this problem, you need to add the "AllowEncodedSlashes NoDecode" directive, and also add the "nocanon" attribute to your "ProxyPass" directive. For more information, see this discussion.

Tree Widget

A full tutorial of Ext JS trees is beyond what we can do in this Manual, but here are is a quick overview of the components as they apply to Diligence:

- The "TreeStore" is manages data for the tree. It handles caching of tree node instances in memory. Note that we've set "defaultRootId" to "/", instead of the default "root". This is to match Diligence's path-based node ID pattern (see above).
- The store also defines a "Proxy," which is Ext JS's extensible connector class. Proxies are in charge of loading the data. In this case, we are using a "diligence" proxy type. This is a custom type that we have defined in Diligence's Ext JS helper library. It's rather simple, and you are free to use the "ajax" proxy type instead with the modifications we've made there. The "diligence" proxy is configured to automatically support MongoDB's extended JSON notation and also use Diligence's URL style. We've additionally set the "root" property for the reader to "documents" (where the node's children will be found).
- Finally, there's the tree panel, which is linked to the store.

That should be enough to get you started. Here's how the code looks:

```
<html>
<head>
<%
document.executeOnce('/diligence/integration/frontend/sencha/')
Diligence.Sencha.extJsHead(conversation)
%>
</head>
<body>
        <div id="tree"></div>
</body>
<script type="text/javascript">
Ext.onReady(function() {
        var store = Ext.create('Ext.data.TreeStore', {
                proxy: {
                        type: 'diligence',
                        url: '<%= conversation.pathToBase %>/data/textpack/'
                },
                defaultRootId: '/',
                autoLoad: true
        });

        Ext.create('Ext.tree.Panel', {
                store: store,
                autoScroll: true,
                useArrows: true,
                rootVisible: false,
                renderTo: 'tree',
                style: {
                        margin: 'auto'
                },
                width: 500,
                height: 400
        });
});
</script>
</html>
```

Sencha Integration: Charts

TODO

Usage

TODO

Sencha Integration: Forms

Setup

We're using the Diligence Forms Service, so follow the instructions there.

The different is that you should use the "Diligence.Sencha.Form" class instead of "Diligence.Forms.Form". The former class extends the latter class with an extra method to better integrate with Ext JS.

So, in "/libraries/resources.js":

```
document.executeOnce('/diligence/integration/frontend/sencha/')

...

resources = {
        ...
        multiply: new Diligence.Sencha.Form(multiplyForm)
}
```

Usage

Configuring the Form Fields

The "toExtJs" method lets you generate the correct client-side source code for configuring fields for the Ext JS form:

```
<%
document.executeOnce('/diligence/service/forms/')
var form = Diligence.Forms.getForm('/multiply/')
%>
<script type="text/javascript">
var fields = <%= form.toExtJs(conversation) %>;
...
</script>
```

Note the difference between server-side and client-side JavaScript here!

The field configurations include the following, if they were set up for the field:

- The masking regular expression.
- The client-side validation function.
- Internationalization text strings for field labels and possible client-side validation error messages.

You can explicitly disable these like so:

```
var fields = <%= form.toExtJs(conversation, {clientValidation: false, clientMasking: false}) %>;
```

Internationalization will use text pack stored in the conversation, or you can set one explicitly:

```
<%
var textPack = Diligence.Internationalization.getPack('fr')
%>
var fields = <%= form.toExtJs(conversation, {textPack: textPack}) %>;
```

AJAX Forms

A full tutorial of Ext JS forms is beyond what we can do in this Manual, but here are is a quick example of how you could create an AJAX form to use with Diligence:

```
<html>
<head>
<%
document.executeOnce('/diligence/integration/frontend/sencha/')
Diligence.Sencha.extJsHead(conversation)
%>
</head>
<body>
</body>
<script type="text/javascript">
<%
document.executeOnce('/diligence/service/forms/')
var form = Diligence.Forms.getForm('/multiply/')
%>
var fields =<%= form.toExtJs(conversation) %>;

Ext.onReady(function() {
        Ext.create('Ext.window.Window', {
                title: 'MyForm',
                width: 350,
                items: {
                        xtype: 'form',
                        url: '<%= conversation.pathToBase %>/multiply/?mode=json',
                        border: false,
                        bodyCls: 'x-border-layout-ct', // Uses the neutral background color
                        bodyPadding: 10,
                        layout: 'anchor',
                        defaults: {
                                anchor: '100%'
                        },
                        defaultType: 'textfield',
                        items: fields,
                        buttons: [{
                                text: 'Submit',
                                disabled: true,
                                formBind: true,
                                handler: function() {
                                        var form = this.up('form').getForm();
                                        if (form.isValid()) {
                                                form.submit({
                                                        success: function(form, action) {
                                                                Ext.Msg.alert('Success!', act
                                                        },
                                                        failure: function(form, action) {
                                                                Ext.Msg.alert('Failure!', act
                                                        }
                                                });
                                        }
                                }
                        }]
        }).show();
});
</script>
```

```
</html>
```

Standard Forms

Ext JS can also perform a standard submission instead of using AJAX. The result is that you get the nice GUI of Ext JS, including client-side validation, but as far as the server is concerned, the behavior is like the standard HTML <form> mechanism.

Why would want to do this? Honestly, with Diligence handling AJAX forms for you, it's hard to imagine a use case. Nevertheless, we'll tell you how to do this, for completion's sake.

Let's use the Diligence Form Service's manual mode:

```
<script type="text/javascript">
<%
document.executeOnce('/diligence/service/forms/')
var form = Diligence.Forms.getForm('/multiply/')
var results = form.handle(conversation)
%>
var fields = <%= form.toExtJs(conversation, {results: results}) %>;
...
</script>
```

Note how we added "results" to "toExtJs". This makes sure that the fields will be initialized with the previous form submission values, and also the correct error codes for field validation.

Ext Direct Forms

Finally, Ext JS forms can also use Ext Direct, Sencha's RPC mechanism, which is nicely supported by Diligence, instead of the regular AJAX mode. Going this route means that you will not use the Diligence Forms Service at all, and use the Diligence RPC Service instead.

We recommend using the Diligence Forms Service if you can, because it will give you fuller control over field validation. However, Ext Direct might be nice to use if you already are using it a lot and have everything set up for it. In any case, Ext Direct is fully supported, and since it's also based on AJAX, the user experience is pretty much the same.

You will need to add an extra attribute when setting up Ext Direct, to make sure that it supports form submission, and also return the results in the appropriate format. Here's an example "/libraries/resources.js", similar to the one for the RPC Service:

```
document.executeOnce('/diligence/integration/frontend/sencha/')

var Calc = {
        multiply: {
                fn: function(x, y) {
                        return {
                                success: true,
                                msg: '{0} times {1} is {2}'.cast(x, y, x y)
                        }
                },
                extDirect: {
                        formHandler: true
                }
        }
}

resources = {
        ...
        calc: new Diligence.Sencha.DirectResource({name: 'MyApp', namespaces: {Calc: Calc}})
}
```

Then, on the client you would create your form *after* initializing Ext Direct like so:

```
<script type="text/javascript">
function openForm() {
        Ext.create('Ext.window.Window', {
                title: 'MyForm',
                width: 350,
                items: {
                        xtype: 'form',
                        api: {
                                submit: MyApp.Calc.multiply
                        },
                        ...
                }
        }).show();
}

Ext.onReady(function() {
        Ext.Ajax.request({
                url: '<%= conversation.pathToBase %>/calc/',
                method: 'GET',
                disableCaching: false,
                success: function(response) {
                        var provider = Ext.decode(response.responseText);
                        Ext.Direct.addProvider(provider);
                        openForm();
                },
        });
});
</script>
```

Note that instead of supplying a "url" key to the form configuration, we use "api" and hook the "submit" key to our Ext Direct method. Ext JS will take care of the rest.

Sencha Integration: Ext Direct

Diligence makes it trivial to support Ext Direct, Sencha's straightforward RPC protocol. Ext Direct it has excellent support in Ext JS and Sencha Touch: the frameworks generate a client-side namespace for you with asynchronous methods equivalent to those on the server. All you have to do is call them! Operations are batched for maximum efficiency, and errors are handled as elegantly as can be.

Diligence actually takes Ext Direct one step ahead in letting you automatically generate the API configuration on the server. A "GET" to the resource will retrieve the JSON needed to configure the client-side provider. We show this in detail under "Usage," below.

Ext Direct's functionality is practically identical to that JSON-RPC, but the protocol is incompatible. It may be unfortunate that Sencha decided not to use that better-known protocol, but in any case Diligence supports both.

Setup

Make sure to check out the server-side API documentation for Diligence.Sencha and Diligence.RPC, as well as the client-side API documentation for Ext Direct.

Ext Direct setup is almost identical to RPC Service setup, so make sure you read the section there.

One small difference is in how Ext Direct handles namespaces. First of all, you cannot have an empty namespace (the "." namespace in JSON-RPC). And, second, you can optionally set up a client-side namespace, using the "name" key. Here's an example "/libraries/resources.js", similar to the one for the RPC Service:

```
document.executeOnce('/diligence/integration/frontend/sencha/')

var Calc = {
        multiply: function(x, y) {
```

```
                return x y
        }
}

resources = {
        ...
        calc: new Diligence.Sencha.DirectResource({name: 'MyApp', namespaces: {Calc: Calc}})
}
```

It is also possible to set Ext Direct method attributes using the long-form method definition with the "extDirect" key.

Ext JS Forms

Ext Direct can be used to respond to Ext JS form submissions. To do so, we need to set the "formHandler" attribute and also return an appropriate response:

```
var Calc = {
        multiply: {
                fn: function(x, y) {
                        return {
                                success: true,
                                msg: '{0} times {1} is {2}'.cast(x, y, x y)
                        }
                },
                extDirect: {
                        formHandler: true
                }
        }
}
```

See the section on Ext JS Forms for more information on usage. Note furthermore that Diligence supports all of Ext JS's form submission mechanisms.

Usage

See the Ext JS documentation for full details on the client-side API. Otherwise, here's a quick tutorial, which also shows you how to fetch the provider configuration from the resource.

Here's an example of a dynamic web page, say "direct.d.html":

```
<html>
<head>
<%
document.executeOnce('/diligence/integration/frontend/sencha/')
Diligence.Sencha.extJsHead(conversation)
%>
</head>
<body>
</body>
<script>
function init() {
        MyApp.Calc.multiply(2, 3, function(provider, response) {
                if (response.type == 'exception') {
                        Ext.Msg.alert('Multiplication', 'Exception: ' + response.message);
                }
                else {
                        Ext.Msg.alert('Multiplication', response.result);
                }
        });
```

```
}

Ext.Ajax.request({
        url: '<%= conversation.pathToBase %>/calc/',
        method: 'GET',
        disableCaching: false,
        success: function(response) {
                var provider = Ext.decode(response.responseText);
                Ext.Direct.addProvider(provider);
                init();
        },
        failure: function(response) {
                console.log(response);
        }
});
</script>
</html>
```

Some notes:

- Make sure you understand the difference between the *server*-side JavaScript (between the "<%" and "%>" delimiters) and the *client*-side JavaScript (between the "<script>" and "</script>" delimiters)!
- We are using "Ext.Ajax.request" to do a "GET" on our resource. It will return the JSON needed for the call to "Ext.Direct.addProvider". Here's how it would look in our example:

  ```
  {
          "actions": {
                  "Calc": [{
                          "name": "multiply",
                          "len": 2
                  }]
          },
          "namespace": "MyApp"
  }
  ```

  You can avoid that "Ext.Ajax.request" call by simply copying and pasting that JSON into your client-side source code. This extra call is simply a convenience allowing you to modify the server-side code without worrying about also having to update the client-side code accordingly. You might prefer to keep this extra call during development, and then freeze it for production code.
- We are disabling the default "disableCaching" mode in "Ext.Ajax.request". Ext JS disabled caching by default in order to better deal with servers that do not handle REST properly. Since Prudence does this for us, there's no reason to avoid client-side caching if it's possible.
- The last argument for any Ext Direct method is a callback that is called when the server returns a response. It's cumbersome, but that's the price you pay for asynchronous remote calls! Also note that you want to properly handle server and network failures.

Blog Feature

TODO

Usage

Make sure to check out the API documentation for Diligence.Blog.

Console Feature

TODO

Usage

Make sure to check out the API documentation for Diligence.Console.

Contact Us Feature

This simple feature contains a /web/fragment/ that displays an HTML form with a CAPTCHA that allows users to send a message on a specific notification service (page 28) channel. System administrators or others subscribing to the channel would then receive it. Straightforward!

Note that a different form is displayed depending on whether the user is logged in. Logged-in users will not have to enter their email address or pass the CAPTCHA. We already know they are legit, by virtue of having logged in!

The originating IP address is included in the email.

Usage

Make sure to check out the API documentation for Diligence.ContactUs.

Discussion Feature

This feature lets you attach a "forum" to any MongoDB document. It could be a Page from the Wiki Feature, a blog post, or just anything in your application. Of course, permissions apply, and you can allow, for example, for registered users to post new threads and have "visitor" users (Facebook, Twitter, etc.) only the right to comment. The discussion is threaded, in that comments can have any level of depth. It's very easy to drop in, and makes a lot of web application features instantly sociable.

Usage

Make sure to check out the API documentation for Diligence.Discussion.

Editable Graph Structures in MongoDB

If you'll take a look at Diligence's Ext JS tree integration, you'll see it's pretty neat. It's *literally* neat because the trees for Ext-JS are immutable, and easily stored in a MongoDB document, which can hold a structure of arbitrary depth. However, if you want your tree to change by multiple users and threads, document databases such as MongoDB begin to show some of their limitations. (Graph databases, such as neo4j, are of course perfect for this use case.)

Nevertheless, it's not impossible, and can get excellent all-around performance for mutability. How is this solved for the Discussion Feature? MongoDB's atomic operations do not support such recursion, so we needed a different method. You can see ideas on the MongoDB trees page.

After some consideration, we used a variation of the "materialized paths" pattern. We have the forum posts stored as plain array, with each having a path as well as a parent field. We parse this document on load, to give it a tree-like structure more amenable to work with. The flat storage structure, however, allows for easy use of MongoDB's atomic update operation. For each post, we store a "nextResponse" running serial. We update it atomically with $inc for each new post, to make sure it's unique, and append that number to the parent's path to create the new path. We then add the new response using MongoDB's $push. The result is that any number of users can respond at the same time to the same forum, and each response takes only two MongoDB write operations, only one of which waits for the response. We're guaranteed atomicity and uniqueness of each path ID.

A graph DB would do this better, but the real comparison would be to a relational database. Just two writes, but the whole forum is read with one read. We think this counts as a smashing success!

You'll notice a rule of thumb we've applied here, useful in general when working with MongoDB: if in relational database you always want your tables to be normalized, in document databases your goal is to use as few documents as possible. In this case, the entire forum is embedded into one document (together with the document's other

data, if there is any). The document limit in MongoDB is 4MB, easily adequate for such discussions. But, what if you want a more open forum, with no limitations on size? Well, the Discussion Feature also comes with a forum implementation that stores each thread in post in its own document, or even each post in its own document. All use the same API. Mix and match for the best performance and growth ability suitable for your needs.

Registration Feature

This complement to the authentication service (page 5) uses a two-step process to allow new users to register to your application. As is common, it expects users to have a personal email address, which will be used to both confirm the identity of the user and to communicate with the user when they are not logged in.

The feature contains a /web/fragment/ HTML form with a CAPTCHA, which collects the user's email, username, password and possibly some personal information. The form will be valid only of the username is not already in use.

If the form is valid, the user is created but not yet activated. An email is sent to the user with a unique, impossible-to-guess URL, which can be used only once. If they click on that link, the user is activated.

The feature allows for not-yet-activated users to be automatically deleted after a certain time. This would release the username for others to use.

Usage

Make sure to check out the API documentation for Diligence.Registration.

SEO Feature

This feature helps you comply with a few *de facto* search engine standards to improve your interaction with them, specifically robots.txt and sitemap.xml.

At first glance, there's nothing very sophisticated about these standards, and you might be tempted to create the required text files manually and then serve them statically. However, large applications with many URLs can easily have unwieldy site maps. This Diligence feature helps you create them and manage them fairly automatically. It supports *very, very* large site maps.

Usage

Make sure to check out the API documentation for Diligence.SEO.

The Goods

robots.txt

Search engines expect to find this resource right at the root of your domain. Its plain text content tells them where to find your sitemap URL, and can also control the crawling of your domain.

Your robots.txt will likely not be very dynamic. Because it matches URLs starting with stated URLs, it can easily cover large sections of your site, and require infrequent tweaking.

When would you need a lot of robots.txt tweaking? A common case for large sites is that public resources are deprecated or otherwise cancelled. In such cases you still want to keep them up for reference, and to allow hyperlinks elsewhere on the web to still be able to reach them—there's SEO value in that. But, you do not want these resources to appear in search engines and confuse users (you want them to find the new, better resources). A robots.txt exclusion would do the trick.

sitemap.xml

If your robots.txt doesn't state otherwise, then this resource will also be at the root of your domain. Its XML content can either list URLs directly, or, more commonly, act as the primary index of other XML files called URL sets.

Search engines do take site maps seriously. A carefully maintained site map would help them keep up to date with your dynamic site, in turn helping to get human searchers to the page they want (or the page you want them to want...). It's likely this would indirectly and directly improve your ranking, too.

Dynamic or Static?

URL sets can grow to be very large (think: Wikipedia), so search engines have put limits on file size: 50,000 URLs per file and 10MB, uncompressed. That's right, you're allowed to compress your site map files with gzip to save bandwidth. There doesn't seem to be a limit on the number of files you can serve, so potentially your site map can be as big as needed.

Diligence supports two ways of generating site map resources: dynamic (via /web/fragments/) and static (via /web/static/). Dynamic is the default, and should be fine for small web sites. It generates robots.txt and sitemap.xml on demand, using Prudence's standard caching to keep things smooth and fast.

But, dynamic mode does not support more than 50,000 URLs per URL set. What's more, it generates these within the HTTP request thread. So, you definitely do not want to use dynamic mode for large sites, or sites which are slow to generate the URL sets! If you do, each time you get hit by a search engine for the site map (can happen several times a day for "hot" sites!) a web request thread will be tied up for the length of time it takes to generate the huge URL set. There are two problems for this: first and worst, the search engine may penalize you for being so slow, and second, even if you are caching aggressively, it means that you will occasionally have one *very* heavy request, breaking the ironclad rules laid out in Prudence's Scaling Tips article.

Static mode can support URL sets of any size: it works by generating all required files in an asynchronous Prudence task so that they can take as much time as necessary, without tying up any user thread. You can set the task to run via Prudence's crontab: once a day, twice a day, etc. The task makes sure to split URL sets into "pages" of 50,000 URLs max, and to gzip compress them. It even makes sure to generate them in a separate spool directory, and then swap them all at once, so that search engines hitting your site exactly during site map generation don't see a partial, inconsistent picture. And it all happens asynchronously, using Diligence tasks, so that multiple URL sets can be generated simultaneously. And, of course, since they are plain old files, you can also host them outside of Prudence.

Note that robots.txt is always generated dynamically: its size limit is 100KB, which should be manageable. The implication is that you can't go crazy with very large lists of exclusions/inclusions. If this is an issue, you can use meta tags instead.

Instruction Manual

Every application in your Prudence instance can have its own URL sets, but it only makes sense for the root application to have both robots.txt and sitemap.xml. We'll start our guide with an application that is *not* the at root, because it's simpler.

From our settings.js:

```
predefinedGlobals = Sincerity.Objects.flatten({
        diligence: {
                feature: {
                        seo: {
                                domains: [{
                                        rootUri: 'http://localhost:8080'
                                }, {
                                        rootUri: 'http://threecrickets.com'
                                }],
                                locations: [{
                                        name: 'the-real-thing',
                                        domains: ['http://localhost:8080', 'http://threecrick
                                        locations: ['/happy/', '/this/', '/is/', '/working/']
                                        exclusions: ['/diligence/media/', '/diligence/style/'
                                        inclusions: ['/diligence/media/name/'],
                                        factory: 'Explicit'
                                }, {
                                        name: 'test',
                                        domains: ['http://localhost:8080'],
                                        factory: 'Fake',
                                        dependency: '/about/feature/seo/fake-locations/'
                                }]
                        }
```

```
                }
        }
})
```

Note the two arrays: domains and locations. There is a many-to-many connection between them, such your application can support many domains, many location groups, and apply different locations groups to different domains. This is because Prudence allows for multiple virtual hosting, so that each application may very well be running on different domains at the same time, and may want to present itself differently to search engines on each domain.

If you don't need to support virtual hosting, ignore the domains array and domains parameters: it will be assumed that your locations are to be applied to all domains.

You then route the SEO resources for the application in its routing.js:

```
document.executeOnce('/diligence/feature/seo/')
Diligence.SEO.routing()
```

Locations

Locations are configured using Diligence's plug-in library, which uses the factory pattern to generate plug-ins. In our first locations config, we used the "Explicit" factory, which is built-in to the SEO feature. This lets use explicitly list our locations as arrays within the config. Obviously, this is useful only for very small sites with a known list of URLs.

The "name" field is important: this becomes exactly the name of the URL set as it appears in the site map. As for exclusions and inclusions: they are lumped into robots.txt.

More interesting is our second locations config: it uses our own factory, which we called "Fake". This factory generates lots and lots (300,000) fake locations, and is useful for testing out very large site maps. (Bottom line: it takes about 7 seconds to generate the complete, gzip-compressed 7-page site map for that many URLs.) It's also a good example for you to use to create your own location factories.

The key to factory success is understanding Iterators: as long as you keep your iterator properly fed, you should be able to scale to site maps of scary sizes.

One more thing to note is that each locations config will be executed simultaneously on its own tasks thread, and this is true for all locations configs on all applications which you include in your root application, as detailed below.

The Root Application

At minimum, the settings.js of the root application should look something like this:

```
predefinedGlobals = Sincerity.Objects.flatten({
        diligence: {
                feature: {
                        seo: {
                                domains: [{
                                        rootUri: 'http://localhost:8080',
                                        applications: [{
                                                name: 'My Application',
                                                internalName: 'myapp'
                                        }],
                                        delaySeconds: 100,
                                        dynamic: false,
                                        staticRelativePath: 'sitemap-local',
                                        workRelativePath: 'sitemap-local'
                                }]
                        }
                }
        }
})
```

You'll see that we added a few more fields to our domain config: beyond the root URI, we are also configuring our robots.txt here, which we will be hosting, and configuring the paths to use for static generation. The static path is relative to the application's /web/static/ directory, while the work path will be under your application's root directory's "work" subdirectory. Alternatively, you can use "staticPath" or "workPath" to provide absolute paths. For example, you might prefer to use "workPath: '/tmp/sitemap"'.

Note that these paths are per domain: if you hosting multiple domains via virtual hosting, each site map should go to a different path. Via a simple filter we make sure that each domain gets it correct site map. Thus, the outside world doesn't actually see these static subdirectories: the URI space for the site map all appears, publicly, at the root.

The truly magical field is "applications": this is an array of application names for which locations will be added to this domain. The URL sets for each application for this will be merged into the main site map, and its exclusions/inclusions will be merged into robots.txt. It's up to you to make sure that URL set names from all applications don't overlap, since their files are all moved into the same static directory.

The root application can also have its own "locations" field, which will also be merged in. We omitted it in this example for simplicity.

To have your site map generated regularly, put something like the following in your application's crontab (as a single line). In this example, we're having our site map generator run every day at 4:00AM:

```
0 4 * * * /diligence/eval/ document.executeOnce('/diligence/feature/seo/'); SEO.getDomain('http://
```

You then route the SEO resources for the root application in its routing.js:

```
document.executeOnce('/diligence/feature/seo/')
Diligence.SEO.routing(true)
```

Well, one tiny little convenience here: though you do need to install the routes in your root application, you are free to host the SEO resources on another app (works via the magic of Prudence's router.captureOther). So, we can call SEO.install(true, 'myapp').

... And do all of the SEO stuff on myapp, even though it's not at root. The root application really doesn't have to do anything else.

Optionally, you can also register the ".gz" extension to serve the gzip MIME type. Search engines would not really care, but it makes your URI-space more correct and debuggable. Do this in the application's default.js:

```
document.executeOnce('/diligence/feature/seo/')
Diligence.SEO.registerExtensions()
```

And that's pretty much it!

Shopping Cart Feature

TODO

Usage

Make sure to check out the API documentation for Diligence.ShoppingCart.

Wiki Feature

TODO

Usage

Make sure to check out the API documentation for Diligence.Wiki.