The Sincerity Manual

Version 1.0-beta14
Main text written by Tal Liron

July 26, 2015

Contents

Part I
Basic Manual

Introduction

Sincerity is a tool for deploying, installing and bootstrapping software stacks on top of the JVM. It makes these tedious tasks easy, simple and fun.

From the user's perspective, Sincerity makes it easy to install complete products and stacks, or individual modules and libraries, into portable "containers," (page 12) which are nothing more than straightforward file directories. According to your preferences and constraints, you can use either Sincerity's pretty GUI (page 12) or the powerful CLI (page 10).

From the provider's perspective, Sincerity is distribution system: simply host your packages in a "repository" (a simple web site) and let Sincerity do the rest. Configuration and bootstrapping of applications is easily controlled via simple JavaScript code, and Sincerity's growing ecology of plugins makes it especially easy to add features such as centralized logging and robust daemons with surprisingly minimal fuss.

Sincerity was born of many years of experience writing complex software for the JVM. The rest of this chapter summarizes this experience and the problematic reality that made a tool such as Sincerity necessary. If this sounds dreary to you, feel free to skip to the tutorial (page 10) for now, and come back here later!

Principles

After using used Sincerity for a while, you'll wonder how you could ever have lived without it.

Indeed, Sincerity arrives after years of us having to repeat the same development and deployment tasks over and over again for every new project: download, unzip, copy, rename and configure, hopefully while staying organized as to dependencies and versions. Solutions like Maven create their own problems (page 9): "enterprise"-style complexity, enormous XML configuration files, and a Java-centrism that is becoming a burden as more of us are developing for the JVM without Java.

We decided that enough was enough! Sincerity intends:

1. To simplify and unify the **installation** of JVM applications, services and libraries. You should never have to download idiosyncratic distributions and read through pages of installation instructions. We aim for a simpler recipe.

2. To simplify and unify **deployment** via the JVM. It seems that every application and service has its own set of bootstrapping scripts, service wrappers, logging configuration, directory structures, etc. We can't entirely smooth out the quirks, but we can make your deployment experience consistent.

3. To be **language agnostic**. The JVM is no longer the exclusive domain of Java. A rich ecosystem of languages has grown around it, and Sincerity lets you manage installation and deployment without ever having to write or think in Java if you don't want to.

4. To be **culturally agnostic**. The JVM is not the exclusive domain of enterprise applications. If you think and work like an agile technology startup, Sincerity is here for you.

5. To cultivate an **ecosystem**. Three Crickets, the company behind Sincerity, maintains a collection of quality plugins that do all the above, and the list keeps growing. Developing plugins is a piece of cake and you're strongly encouraged to develop your own using the straightforward API. This Sincerity Manual contains everything you need to know to get started.

Lather, Rinse, Repeat

The free software and open source movements have utterly changed how we develop software.

Reusable libraries had existed freely before, but these movements have created a culture of sharing, fueled by viable business models, culminating in an unprecedented wealth of solutions. For any problem you encounter in your everyday development work there is likely a library out there to help you that you can download for free. "Your mileage may vary," as they say: quality may not always be up to snuff, and no warranty is provided, but the source code is included and you can make it better, for yourself and for others. Importantly, it's relatively future-proof to

depend on free software: you can be certain that your license to use the library will not be revoked and that bugs *could* be solved, by you, by the community, or by hired help.

(You do need to worry whether the software breaks any owned patents, but that problem exists for any software, whether it's free or proprietary, from a third party or developed by you.)

This wealth of solutions also creates challenges. There are several packaging, versioning and delivery standards for libraries. And when it comes to platforms and frameworks, there is no standard way to deploy software on top of them. If your project is a composite of many of these, you will find yourself spending a lot of time making sense of these various schemes and integrating them into a system that is maintainable by you in the long run. And if your software is itself modular and redistributable, you will find yourself having to pick one of the many different methods, or inventing one of your own. So, when it comes down to it, while free software can save you a lot of time and effort in terms of development, you end up spending extra effort on integration and maintenance. Annoyingly, you'll find that much of this work is repetitive, unnecessarily so. If you're a programmer used to making code reusable, you'll may such repetitive work especially annoying.

When it comes to the JVM, a few products have been widely adopted that make some of this work easier. However, experience has shown them to have too small a scope: they solve very specific problems, but do not address the complete challenge (page 9). Additionally, since they are already a few years old, they predate the linguistic revolution that is happening in full force on the JVM: no longer is Java the only good choice for leveraging the platform. New and popular languages like Scala, Clojure and Groovy offer a new experience and culture, while Nashorn, Rhino, Jython, JRuby, Quercus and Luaj bring popular languages and their paradigms to the JVM. Indeed, since version 7, the JVM has added support an opcode (invokedynamic) that cannot be normally generated by compiling Java language code: for the first time in its history, the JVM is made for languages that aren't Java.

With that in mind, Sincerity is designed from the ground up with multilingual support, which means that not only is knowledge of Java code never required, but also that the culture of dynamic languages and their standards are intrinsically supported: you can include dependencies from Ruby gems, Python's PyPI repository and PHP's PEAR repository. Moreover, Sincerity has standard plugins that make installing and working with these dynamic languages especially easy.

You might want to jump straight to the tutorial to see how it works, but you're also invited to stay here and look at some of the development and deployment tasks that Sincerity tackles.

Dependency Management and "DLL hell"

How do you get a library working with your application? Let's see:

1. Find the library's web site.
2. Look for the "download" button.
3. Download the latest version: note that you want to write down all versions of all libraries you are using, so that you can handle upgrades and possible conflicts.
4. Open the distribution archive: you want to be organized about this, so that you can find licenses, documentation, etc., later on.
5. You need to make put the jar in your classpath for the following environments:
   (a) Your development environment: you might also want to link source code and documentation if they are available in the distribution.
   (b) Your deployment environment: the application needs it to run, so you to need to somehow include the file in your bootstrapping script.
   (c) Your distribution, assuming you are distributing your application: this is optional, since you might decide not to include this dependency, and to have the users download and install it themselves.
6. There might be configuration files (property files, XML, etc.)
   (a) You might need to make different versions of these for your different environments.
   (b) The configuration files might not be flexible enough for how your application runs, with too much assumed or hardcoded, so you will need to either:
      i. Document this fact for the user to handle on their own.

ii. Generate the configuration files during your application's bootstrapping process.
iii. Patch the library to allow for the flexibility you require.

7. Once in a while you want to check for upgrades, which might mean subscribing to an RSS feed or mailing list, or just reminding yourself to check the web site.

8. The library might have requirements, so you need to make sure to do all the above for them.

The above steps involve a lot of work. And what if you have 20 dependencies?

This is not a new problem, and there are already a few solutions for it. Firstly, there is a straightforward standard for JVM repositories, iBiblio/Maven, which is widely used by many projects. But it requires you to use one of two tools: Ivy, which does a good job of downloading dependencies (and is used internally by Sincerity), but does nothing else, or Maven, which is a sophisticated, heavyweight project management tool with a steep learning curve, and which requires you to work entirely within its domain. We'll compare these tools in more depth to Sincerity later on, but for now let's just say that the former is too limited in scope, and the latter too constraining. There are also various difficulties in configuring these tools: Sincerity "just works," immediately and easily, and also handles bootstrapping and assists in configuration.

There's also the problem of being forced in the JVM bubble: if you're using Jython, JRuby or Quercus, then you have to also work with the repository standards of Python (PyPI), Ruby (gems) and PHP (PEAR). Sincerity is designed to support all of these standards.

Then there's the issue of potential conflicts, a.k.a. "DLL hell": What if one application you're working on requires one version of a specific library, and another application requires another? What if this happens within different parts of the same application? Again, there are standards and tools for this—OSGi and Jigsaw—but they require you to work entirely within the paradigms they enforce. Sincerity doesn't stop you from using them (in fact, it has great support for the Felix OSGi container), but definitely does not force you to play by any special rules. From the bottom up, Sincerity is designed to be as straightforward and universal as possible. See the detailed comparison to OSGi below for more information.

Bootstrapping

The JVM is packaged as a set of command line utilities, plus a few plugins for specialized environments. It does come with one simple way to distribute programs—executable JAR files—but that would only suffice for the most trivial programs.

For anything more complex, you will need to handle bootstrapping your application. This means, at the very least, finding the right JVM on the machine (more than one may be installed), and then loading the application via the "java" tool. Usually, however, it ends up being far more complicated: rummaging through environment variables, detecting the host operating system and environment in order to set specialized JVM flags and load optimized native libraries, and because this is so complex, you'll want to responding to specialized bootstrapping flags set by the user. Indeed, many JVM-based products won't "just run," but will in fact require you to set a host of environment variables first.

All this work happens before the JVM even starts. Thus, it's usually handled by writing a shell script, which is almost always immediately runnable. Depending on how many operating systems you want to support, this may mean, at the very least, writing one for *nix systems and one for Windows systems. This is highly specialized work, and a development project with its own challenges, so some projects choose to avoid scripts and develop native binaries that handle bootstrapping. And then there are installer products that purport to do this all for you.

And what if you want the software to run as a daemon, system service, or cron job?

And what if your software is not just one program, but also contains a set of tools that you also need to bootstrap?

The bottom line is that bootstrapping is very hard to get right, and there are many complicated approaches to it. It's a shame that so many JVM products keep trying to implement the same bootstrapping solutions from scratch. Sincerity streamlines this in two ways: first, by providing you with working shell scripts, and second, by having these scripts delegate the process as soon as possible to a JavaScript program running in the JVM. Once on the JVM, Sincerity offers a range of installable plugins that handle various configuration and deployment tasks, including running the software as a daemon.

Why JavaScript and not a different scripting language? We deal with the question in length below (page 8).

What this means is that most products won't have to do anything beyond what Sincerity offers out of the box, and those with specialized bootstrapping will be able to write portable JavaScript programs, instead of having to deal with complex shell scripting.

Configuration and "XML hell"

Between bootstrapping and reaching full usability, your product has to configure itself. Will you choose a properties file? XML? Something else? And where is the file located?

Well, consider that all the libraries you use had to make their own choices for configuration. A non-trivial JVM product could thus require several configuration files, in different locations, with different configuration rules.

But there's a more serious problem to most of these approaches: they are unnecessarily rigid and static. While there are many advantages to using text files for configuration, the choice of technologies is baffling. Possibly the worst choice is XML. This language, ostensibly a "markup" language, is marking up nothing when used for configuration: it's instead used as a cumbersome format for structured textual data. And it gets far, far worse: XML configuration files are often used *programatically* in the JVM world, to construct JVM classes and call JVM methods. The best known, worst offenders are Log4j and Jetty. There, XML is used as if it were a scripting language, the clumsiest you have ever seen.

The use of XML for configuration is part of what we call "XML hell," which refers to programmers being swamped with countless overly-verbose XML files. XML is also often abused as an interchange format on the Internet, and a descriptor format in much of the JVM enterprise industry. Enough already!

The excuse for this insanity, one would guess, is that the ability to parse XML is standard on many platforms, including the JVM. But, *interpreting* this XML is far more complicated that just parsing it. In essence, parsing a general-purpose XML for something like Jetty involves writing a complete (more likely, not complete enough) scripting language engine. Another excuse for "XML hell" could be part of the general over-enthusiasm with XML, and the untested faith that standardizing on a single format would lead to greater interoperability. Again, this is madness: unless you couple the XML file with the code that can make sense of it, the ability to parse them is of little use.

Another approach, better than XML, is to create a Domain-Specific Language (DSL). But DSLs require a lot of work, both by developers and by users who must learn them.

Sincerity is here to stop the madness: wherever possible, it standardizes on using JavaScript for configuration. (Why JavaScript? We deal with the question in length below.) With JavaScript you can instantiate objects, call methods, insert conditionals and loops using natural programming paradigms, instead of shoe-horning them into XML. At its simplest, a JavaScript configuration file can look identical to a simple properties file: straightforward assignments of values to configuration parameters. But, you also have the option of injecting interpreted code where appropriate. And, of course, it's still just text files that don't need to be compiled, and can even be picked up and re-interpreted at runtime, so you're still absolutely within the "configuration-by-text-file" paradigm.

If you've never tried the "configuration-by-script" approach before, you might be skeptical about its benefits or worried about the extra weight it adds. But Sincerity's JavaScript engine is very lightweight, and we're convinced that once you try this approach, you will avoid all others. For an instructive example, install Sincerity's logging plugin (page 32), and take a look at the logging configuration files. Now compare them to the "official" Log4j formats.

One consequence of this approach is that the line between bootstrapping and configuration gets blurred. They end up as one integrated phase: a bunch of JavaScript programs strapped together. This leads to both simplification and greater flexibility for you. This approach leads to exceptionally dynamic configuration systems that can adopt to any operating environment.

We really hope to see "configuration-by-script" used throughout the JVM world, even for projects that do not want to or cannot use Sincerity.

Logging

The JVM has a few good, widely-used logging APIs, as well as a great glue library—SLF4J—that can bridge between them. But there's quite a bit of work involved in getting all these libraries working together. It seems that every JVM product has its own way of doing this. Logging is important, and can't be relegated to an afterthought: if it's not properly configured and well integrated, it's close to useless.

Sincerity takes logging very seriously: it provides a plugin (page 32) that does much of the work for you, and extensions that further enhance logging. For example, one extension funnels all logs to a centralized MongoDB collection, perfect for distributed cloud deployments. And this system will work with practically *any* JVM library.

Note that logging configuration is handled via the "configuration-by-script" approach mentioned above, and is well integrated with the whole ecology of Sincerity plugins.

Why JavaScript?

There are many great scripting languages for the JVM. Note that by "scripting" here we mean that these languages are immediately runnable from the textual source code. This doesn't have to mean that they are "interpreted": many of these languages compile some or all of your code on-the-fly. So, why has Sincerity standardized on JavaScript, rather than Groovy, Scala, Clojure, Lua, Python, Ruby or others?

There's no single answer, but rather a combination of factors that make JavaScript attractive:

1. JavaScript is very well known. Since its original introduction into web browsers, and universal adoption as a web standard, it has gained an enormous skill share in the industry. There's a wealth of education material available for it.

2. Its implementations are relatively lightweight, in that JavaScript is both fairly minimal linguistically, and also does not have anything like a standard library, of the kind you would find in Python and Ruby. This allows Sincerity to have a much smaller footprint than if it were to use Jython, JRuby or even Groovy. Note that not having a standard library can also be seen as a disadvantage, but in the case of Sincerity it's dealt with in two ways:

   (a) Since we are running JavaScript on the JVM, we have full access to the Java standard library.

   (b) Sincerity comes with the Sincerity JavaScript Library (page 50) a very lightweight framework that makes working with JavaScript on the JVM a little bit easier.

3. JavaScript is very future-proof: not only is it an open standard (where it's called "ECMAScript"), but it is baked into the JVM (from version 8) as the Nashorn engine.

4. JavaScript is actually a *nice* language. It has been the target of a lot of negativity from programmers who had to work with it in browser environments, but we believe the fault is more of the environment (the browser DOM's poor API and many annoying differences between various browser implementations of it) than the language itself. It encourages prototype-oriented programming, which can easily emulate object-oriented programming, as well as other paradigms. In fact, JavaScript shares much of its scoping and function handling with Scheme, a language that is generally admired. You can think of it this way: JavaScript is Scheme with a C-like syntax.

Despite these *general* advantages, you might still prefer to use another scripting language for your own work. Luckily, Sincerity, with the help of the Scripturian library (page 50), will let you write plugins in Python, Ruby, PHP, Lua, Groovy or Clojure. The only disadvantage is that you would have to include the appropriate language engine as a dependency. In the interest of keeping Sincerity and its ecology of plugins lean and mean, we want to encourage the use of JavaScript for plugins that are intended to be shared with the community.

Just to be 100% clear: this preference for JavaScript only applies to Sincerity plugins, configuration scripts, and skeletons: you are definitely welcome to write your application in whatever language you choose. In fact, Sincerity contains great plugins for many popular JVM languages, as well as skeletons for complete language-specific frameworks, such as Django and Rails.

JavaScript vs. Shell Scripting

This section is meant for those of you who are comfortable with shell scripting, and are wary about Sincerity's use of JavaScript for bootstrapping.

1. You might think that shell scripting would always be more portable than a scripting language running inside the JVM. But, think again: the point of your bootstrapping work is to get *into* the JVM, in order to run your application. If that doesn't work, then your whole application won't run, and portability is moot. Sincerity *does* have shell scripts, but they're designed to delegate to the JVM as soon as they can.

2. You might be concerned about startup delay: starting up the JVM with all the JavaScript engine classes is much slower than starting up a shell script. This is true, no doubt, but since version 7 the JVM is doing better. Also consider that you have to get into the JVM anyway for your application to do anything useful. Still, if your application has a lot of tools that do not always require the JVM, and would be adversely affected by the JavaScript bootstrap times, then by all means write them as shell scripts! You can use all of Sincerity's other features when you need them.

3. Shell scripts treat most of your program as an opaque, black box. But with JavaScript running in the JVM you can call parts of your API before the application truly starts. This can allow for much more powerful, dynamic bootstrapping.

4. JavaScript is likely richer than your shell language. Sure, bash 4.0 and PowerShell are a leap forward compared to what we had 20 years ago, but they're still quite constricting.

Comparisons with Other Solutions

Sincerity vs. Maven

Apache Mavenis a comprehensive solution for managing Java projects, handling building, dependency management and distribution. It contrastingly combines a lot flexibility on the one hand—an open plugin API built on the Plexus IoC container—with deliberate rigidity on the other hand: a strict reactor-based, multi-phase cycle. In particular, Maven's design goes to great lengths to keep you from affecting the order of operations: you are supposed to configure your project, and let Maven decide what to do when. For those used to scripting their build process, this approach may initially seem baffling and restricting. However, there are significant benefits to this approach when working with very large, complex projects: instead of coding and maintaining nightmarishly long build scripts based on dozens of changing environment variables, you can sit back and let Maven analyze the entire operation and then do the right thing.

But, for this to work, you need to play by Maven's rules, and that's where things get tricky. Small deviations from the strict assumptions Maven makes throw you down the rabbit hole of plugins and hacks, as you struggle to shoe-horn a simple procedure into a product that abhors procedure. Specifically, Maven's ideal environment is one in which your versioned modules are written in Java mapped to single jar files. Anything even slightly different becomes painful and hacky.

Both Sincerity and Maven handle downloading dependencies, but other than this apparent overlap these products have different goals and scope. Importantly, they can be very complementary. One way to think of this is that Maven could come first and Sincerity come second: Maven could help you build your project and repositories, while Sincerity would handle your deployment container. Maven won't help you run your application: its output is jars of compiled code, source code or documentation, and it doesn't handle their bootstrapping or runtime configuration. On the other hand, Sincerity does not build your project, nor does it make any assumptions about how its built: you can use Maven, Ant or anything else.

Sincerity vs. OSGi

An "interface" in the JVM lets you create a standard protocol, such that you can plug in various implementations of it—"classes," with "methods" as the entry points—at runtime. The protocol is enforced by the JVM, which will not let you plug in implementations that do not fit the interface. OSGi takes this up a level, by providing a much broader concept of "implementation." The implementation is a "bundle" that can contain any number of classes.

So far so good, but it gets complicated fast. OSGi takes it up one level more: the protocols are published and endorsed by a community of providers, with the idea that different providers (software vendors or departments in a large enterprise) can provide bundles to implement them, which would all work together perfectly. With this broader ambition, "DLL hell" suddenly becomes a far more malevolent enemy: bundles are often black boxes that you cannot easily patch to use a shared version of a dependency. There's thus a real need for a standard solution of runtime code compartmentalization, which OSGi provides via a clever system of classloaders.

... Which, of course, introduces its own set of problems. To get its classloading scheme to work, OSGi requires strict separation of classloading between bundles, which in turn adds subtle and mischievous restrictions to your usual JVM work. This is not entirely bad: working within these limitations does encourage clean, sharp boundaries between your modules, and goes a long way towards reducing classloading confusion. It's not, however, trivial by any means, and all your bundles must be designed with this in mind for OSGi to work properly.

One very useful side effect of having the framework control classloading is that entire bundles can be loaded and unloaded during runtime. Indeed, OSGi defines protocols for starting, stopping and hotswapping services. This is a powerful feature in itself, and is indeed the entire motivation for using OSGi in some cases. (Though, if that's your reason, you might want to look at other, simpler ways to enable hotswapping, rather than embracing the whole of OSGi.)

It's worth noting, however, that there is a more straightforward solution to the problem of "DLL hell": Why not run each "bundle" as a separate process? Each JVM would load its own classes as necessary, and never will they mix or conflict. This makes a lot of sense if you're running a distributed system, since you're already dividing your

software among many machines and processes, and indeed many parts of your application may not be JVM-based at all, and can't be run in a single process anyway. As for starting and stopping your "bundles," the operating system already does a good job of managing processes, so you don't need OSGi's protocol for that. From this perspective, you can see that OSGi is, in effect, creating a virtual operating system *inside* the JVM, where "bundles" are very much like operating system processes.

Indeed, the original target environment for OSGi was precisely one in which all bundles ran in a single process, in shared memory space: it is the world of embedded computing, where the runtime is variously confined, such that you are either limited to a single process due to limited resources or security concerns. In such environments OSGi may be your only good solution for the problem of modularity and pluggable services. Still, OSGi has also proved popular in large enterprise environments, where it allows for modules to be treated more abstractly whether or not they are running in a single process.

Sincerity, in itself, takes the more straightforward approach: such high-level modularity is provided through the notion of "containers," which you can easily create, clone and change, and start and stop as processes, specifically the service plugin (page 34) makes it especially easy to run them as daemons and services. Containers can then talk to each other (and to other services) using whatever technology is appropriate, be it REST, SOAP, message queuing, Hazelcast, etc. That said, OSGi may indeed be appropriate for your project, and Sincerity provides a nice Felix plugin to get you up and running. The point being that Sincerity was designed to be *neutral* to the technology of modularity, introducing no special restrictions for users that do not need them.

Tutorial

Install Sincerity

You need a JVM, at least version 6.

If you're an Ubuntu user, then use our repository! It would do everything for you.

Otherwise, download a Sincerity distribution. If you're download the Zip distribution, unpack the folder, and put it in any standard location, for example:

- Unix: "/opt/sincerity"
- Windows: "C:\Program Files\Sincerity"
- Mac OS X: "Applications"

You can then run the "sincerity" script (Unix and Mac) from there or "sincerity.bat" (Windows).

You might want to add the Sincerity path to your system path, to allow for easy access from the command line. In Linux, you can do this by adding the following line to your user's .bashrc file:

```
PATH=$PATH:/opt/sincerity
```

Working with the Command Line

If you run Sincerity with a Graphical User Interface (GUI) (page 12) using "sincerity gui". However, it's strongly recommended that you learn how to use the command line. Here are the main principles:

1. All Sincerity commands exist within "plugins." The full name of a command is its plugin name, with a colon, and then the command name within the plugin. For example, "container:create" is the "create" command within the "container" plugin. Use "sincerity help" to list all available commands from all plugins. Many commands support command line arguments, both required and optional. See the command documentation for full details.

2. As a short form, you can use only the command name. However, this will only work if there is no ambiguity, meaning that the same command does not exist in more than one plugin. For example, "create" will be equivalent to "container:create" if no other plugin has a "create" command. Also note that the full form of the "help" command is "help:help". (Plugin developers are encouraged not to use command names that would conflict with the core plugin commands, such as "create", "add", "install", etc.)

3. Some Sincerity commands can only be run while pointing to a container. Generally, it's useful to run Sincerity when the current directory is somewhere in the container. There are a few rules to consider:

(a) Sincerity can only point to one container at a time.

(b) You can change the container or explicitly point to one using the "container:use" command (page 19).

(c) Otherwise, Sincerity will attempt to find for a container in the following order:

i. The "sincerity.container.root" JVM property

ii. The "SINCERITY_CONTAINER" environment variable

iii. Search up filesystem tree from current path looking for a directory that has a ".sincerity" subdirectory

4. Sincerity's current set of available plugins, which affects the set of available commands, is a combination of both the plugins available in the Sincerity installation *as well as* those available in the current container.

5. You can chain commands together using ":". Command chains are used extensively in Sincerity.

   (a) Note that a single command chain can change the current container multiple times. For example:

   ```
   sincerity use container1 : install : service start web-server : use container2 : log
   ```

   (b) Considering the above, note also that each time you switch container within a command chain the set of available plugins and commands changes, matching whatever is the current container at the time.

   (c) Also keep in mind that a single command chain is always run *within the same JVM*. Sincerity achieves JVM classworld separation by swapping class loaders when it changes containers. If this behavior is not desired, you should avoid chaining and run your commands using separate "sincerity" command lines.

6. If an argument begins with a "@" character, it will be interpreted as a shortcut, and searched for in the current container's "/configuration/sincerity/shortcuts.conf" file. If found, it will be expanded to the command defined there.

   (a) Expansion to a command chain is allowed, as well as recursive use of shortcuts. For example:

   ```
   sincerity use container1 : @mv : start restlet
   ```

   Would expand to:

   ```
   sincerity use container1 : add mongovision : install : start restlet
   ```

   If the following entry is in "shortcuts.conf":

   ```
   mv = add mongovision : install
   ```

   (b) Some commands support implicit use of shortcuts without requiring the "@" prefix. Specifically, the "dependencies:add" and "repositories:attach" commands will search for shortcuts with the "add#" and "attach#" prefixes respectively. For example:

   ```
   sincerity add mongovision
   ```

   Will expand to:

   ```
   sincerity attach three-crickets : add com.threecrickets.mongovision mongovision
   ```

   If the following entry is in "shortcuts.conf":

   ```
   add#mongovision = attach three-crickets : add com.threecrickets.mongovision mongovisi
   ```

From here, you can continue reading about the core plugins (page 19) to learn about all the essential commands.

Working with the Graphical User Interface (GUI)

Sincerity has a Swing-based GUI that displays information about your container and lets you perform operations on it. It can be used instead of the CLI, though each interface has its own strengths. The GUI is especially useful for displaying data, such as the dependency tree structure.

Sincerity provides the GUI frame, but the contents are provided by plugins. This means that the whole GUI would look differently according to whatever is the current container and what plugins it has installed. For this reason, when you change containers from within the GUI, it will restart.

If you run "sincerity" without any command, it will default to running "shell:console" (page 27). You can also start the GUI via the "shell:gui" command (page 27). For example:

```
sincerity use container1 : gui
```

A richer console, in which you can use full JavaScript, is available via the "jsshell:jsconsole" command (page 27).

If you are designing your own Sincerity plugin, it is strongly recommend that you include GUI support if appropriate via the optional gui() entry point.

Environment Variables

The "sincerity" script will try to find your operating system's default JVM and your Sincerity installation. You can modify its behavior using the following environment variables:

- SINCERITY_HOME: The root of the Sincerity installation to use. If not provided, will automatically discover it according to actual (not symlinked) location of the script file.
- SINCERITY_JAVASCRIPT: To force the JavaScript engine to either "Nashorn" or "Rhino".
- SINCERITY_CONTAINER: The path of the container to use. If not specified, will search up the filesystem tree from the current path. See also the "container:use" command (page 19).
- SINCERITY_DEBUG: An integer specifying the internal debug level. Higher numbers will display more debugging information. Defaults to 0.
- JAVA_HOME: The root of the JVM installation. If not provided, will use a platform-specific heuristic to discover it.
- JAVA_VERSION: Used only in Darwin (Mac OS X). Defaults to "CurrentJDK".
- JVM_LIBRARIES: Extra libraries to add to the classpath.
- JVM_BOOT_LIBRARIES: Extra libraries to prepend to the boot classpath (-Xbootclasspath/p).
- JVM_SWITCHES: Extra switches to add to the JVM command.

Components

Before detailing the core plugins and commands in the next chapter (page 19), it's important that you understand a few basic components:

Container

A set of files implementing a self-contained JVM-based execution environment managed by Sincerity and by you. The container has a root path, under which it may have a directory structure of any depth. Libraries, binary executables, configuration files, temporary work files and logs are all by default stored within the container.

Why such an emphasis on self-containment? One goal is for the container to be deployable anywhere as a whole, simply by copying the directory elsewhere. Another goal is for the container to be a useful playground: you can install and try out various applications and libraries without affecting your operating system. You can undo you work simply by deleting the container's directory.

It is possible and sometimes useful to break this principle of self-containment by using symbolic links.

Below are some a few standard container subdirectories used by the core plugins. Other plugins and skeletons may add more subdirectories.

/.sincerity/ Reserved for Sincerity's internal use. It's most essential use is to mark a directory as a container root.

If you are using a VCS, make sure to commit this hidden directory (page 15).

/cache/ Files put here should be considered deletable without any negative effects.

Two subdirectories are most common: "/cache/sincerity/" is where Sincerity will store information about downloaded dependencies, and "/cache/javascript/" is where Scripturian (page 50) will store its compiled JavaScript code.

You likely do not want to commit this directory to a VCS (page 15).

/logs/ Files put here should be considered deletable without any negative effects. This is used by the logging plugin (page 32).

You likely do not want to commit this directory to a VCS (page 15).

/configuration/ Container-wide configuration files for various libraries are found here. Note that generally Sincerity prefers "configuration by script," so that most of these files will be in JavaScript code. However, some libraries may require XML, property sheets, or other unfortunately idiosyncratic formats.

Various libraries will use their subdirectories here: for example, "/configuration/logging/" for the logging plugin (page 32).

/configuration/sincerity/ Here you can configure your container: repositories, dependencies, installed artifacts, and shortcuts. Note that you usually will not have to edit these files directory: many Sincerity core commands will manipulate these files for you.

For the format of "repositories.conf", see the Ivy documentation for resolvers. For the format of "dependencies.conf", see the Ivy documentation for dependencies.

/libraries/ Sincerity will install dependencies here, but you can also add your own files manually.

Note that the Sincerity installation also has a "/libraries/" subdirectory, which is considered in addition to the one found in your container.

/libraries/jars/ Sincerity will recursively add all Java archives (.jar files) here to the classpath. Those dependencies installed by Sincerity will follow the "/organization/name/version/name.jar" directory structure, for example: "org.slf4j/slf4j-api/1.6.6/slf4j-api.jar". It is *not* required that you follow the same structure for jars you install manually: *all* jars found under this directory will be added.

/libraries/classes/ Sincerity adds this path to the classpath, expecting to find JVM class files (".class"). The directory structure *must* be "/package/sub-package/.../classname.class". For example, the JVM class "org.myorg.Frame" would be in "/org/myorg/Frame.class".

/libraries/javascript/, /libraries/python/, etc. These subdirectories are for libraries for specific programming languages to use directly. Note that these are slightly different from the "/libraries/scripturian/" subdirectory, which also contains programming language libraries, but is intended to use only from within Scripturian (page 50).

/libraries/scripturian/ Sincerity, as well as other products that use Scripturian (page 50), will look for executable documents here (and possibly in other places). Most libraries and frameworks will create their own subdirectory underneath. For example, Prudence (page 46) libraries are under "/libraries/scripturian/prudence-scriptlet-resources/".

/libraries/scripturian/plugins/ This subdirectory is reserved for Sincerity plugins. Each document here represents a single plugin, and each plugin may implement any number of commands.

/libraries/scripturian/installers/ This subdirectory is reserved for Sincerity installers. Installers are run by the "artifacts:install" command (page 23), and are included in some dependencies as a way to execute arbitrary installation tasks. A common use case is for the install hook to manipulate the unpacked files in order to tailor them for the specific environment in which the container is running.

/libraries/web/ Files here are intended to be served over the web as static files, for example: images, HTML files, CSS, etc. Various web servers will look for files here (and possibly in other places). Various client-side web frameworks (such as jQuery, Ext JS) will thus install here, and be made available for various web servers you may have installed.

/programs/ The "delegate:start" (page 25) command will look for Scripturian (page 50) documents to run from here.

/executables/ The "delegate:execute" (page 25) command will look for executables to run from here.

/reference/ Reference material, for use by humans or by software, will be available here.

/reference/documentation/ Here you'll find reference manuals and API documentation for installed dependencies. Files installed by Sincerity will be placed in "/organization/name/version/" directory structure, for example: "org.slf4j/slf4j-api/1.6.6/".

/reference/licenses/ Here you'll find licenses for installed dependencies. Files installed by Sincerity will be placed in "/organization/name/version/" directory structure, for example: "org.slf4j/slf4j-api/1.6.6/".

See also the "dependencies:licenses" command (page 21).

/reference/sources/ Here you'll find source code for installed dependencies. Files installed by Sincerity will be placed in "/organization/name/version/" directory structure, for example: "org.slf4j/slf4j-api/1.6.6/".

Dependency

A contained, versioned, installable set of files (called "artifacts"), which can in turn have its own list of dependencies. Dependencies are deployable software bundles, representing things like libraries, frameworks, platforms and skeletons, including complete applications and services.

All dependencies in Sincerity are identified by a two-part name, composed of a "group" prefix plus a unique "name" within the group, plus a version specifier. Different dependencies might have their own versioning schemes, but Sincerity is good at guessing these for the purposes of comparing versions.

Note that a dependency can also have none of its own files, and only a list of its own dependencies, allowing for a convenient shortcut for installing several dependencies together. These are sometimes called "meta-dependencies."

You can list all installed dependencies using the "dependencies:dependencies" command (page 21). However, note that dependencies are structured as a nested tree that may be better visualized using the GUI (page 12).

Artifact

There are files within a dependency. Sincerity supports a specific set of artifact types: JVM libraries (jars), language-specific libraries (Python eggs, Ruby gems, PHP packages, etc.), documentation bundles, source code bundles, software licenses, installers, dependency descriptors and more generic "packages" (see below).

You can list all installed artifacts using the "artifacts:artifacts" command (page 23).

Package

This is a special kind of Sincerity-specific jar artifact that can contain other files, and can additionally have special install/uninstall hooks.

Sincerity contains tools to help you easily create your own packages, as well as documentation about the package specification, so that you can manually create your own.

Packages are automatically unpacked using the "artifacts:install" command (page 23), but you can also explicitly unpack them using "packages:unpack" (page 24).

Repository

A store for dependencies and their artifacts. Repositories usually contain indexes of available dependencies and versions. Some repositories also have friendly human-facing web frontends which you can use to search for dependencies. Sincerity supports several repository technologies, and can also help you deploy your own dependencies to them.

You can list all repositories attached to your container using the "repositories:repositories" command (page 20).

Working with a VCS

Because Sincerity containers are all in a single directory tree, it's very easy to use them with Version Control Systems (VCSes), such as git, Mercurial and Subversion.

One quick issue to note is that your "/.sincerity/" directory will often be empty, and many VCSes, such as Git, tend to ignore empty directories. To force Git to commit it, simply add a ".gitignore" file in that directory:

```
touch .sincerity/.gitignore
```

Otherwise, there are two recommended strategies for working with a VCS:

Strategy #1: Commit (Almost) Everything

This strategy is safest in terms of testing and debugging, because it guarantees that all developers and deployments are sharing the exact same files.

You might just want to make sure that you don't commit the deployment-specific directories. Here's an example of a ".gitignore":

```
/cache
/logs
```

The problem with this strategy—and it can be serious—is that distributed VCSes often require you to clone the repository with its entire history. Every time you change a large binary, it will increase the size of the repository. If this happens a lot, the repository can become quite unwieldy.

Some VCS have workarounds for this problem, though they would only work in environments where developers have access to these files via a shared directory. Consider, for example, git-annex for Git.

Strategy #2: Commit Only Your Work

This strategy makes good use of Sincerity and allows for compact repositories. The idea is that users of the repository will just have to run "sincerity install" to fill in all the missing files.

In order for this to work well, you will need some discipline. You will need to have your VCS explicitly ignore all files that are managed by Sincerity. A good way to do this a blanket ignore on all standard container and skeleton directories, and then add exceptions for files you add or change. Care must be taken during the commit phase to make sure that your changes have indeed been committed, and that you have not forgotten to add an exception. If you forget, your changes will not be committed and can be lost.

One small but important issue is that you want to make sure that "/configuration/sincerity/artifacts.conf" file is ignored. This file is managed by Sincerity specifically in order to keep track of files changed during "sincerity install" (and "sincerity unpack").

Here's an example .gitignore for a container based around a Prudence skeleton (page 46):

```
# Ignore everything by default, allowing Sincerity to manage it
/cache
/component
/configuration
/executables
/libraries
/logs
/programs
/reference

# Our applications
```

```
!/component/applications/myapp1
!/component/applications/myapp2

# Modifications to installed applications
!/component/applications/prudence-admin/routing.js

# Our shared libraries
!/libraries/scripturian/minjson.py

# Component modifications and additions
!/component/servers
!/component/services/database

# Logging configuration
!/configuration/logging/appenders/common-file.js

# Sincerity configuration
!/configuration/sincerity/repositories.conf
!/configuration/sincerity/dependencies.conf
!/configuration/sincerity/shortcuts.conf
```

Note how we added exceptions for both new directories added to the container as well as changes to specific files. Also note that the "/configuration/sincerity/artifacts.conf" file is ignored, as required, due to the blanket ignore on "/configuration".

There are two possible disadvantages for this strategy:

First, unless you specify dependency versions precisely for *all* dependencies, every time a user runs "sincerity install" they may get different versions, and thus have a different container. For some testing strategies, this is a disadvantage. However, for more "agile" continuous build strategies, this can actually be seen as an advantage, as it makes sure that you are always at the cutting edge. As long as your tests are run *before* deployment, then this should not be a problem. However, it could still be a problem for coordinating debugging if multiple developers are working on the same VCS repository but are using different versions of dependencies. To work around this potential problem, you can of course maintain your own repository and coordinate its use with the development team, with the same care used for coordinating VCS repository use. Alternatively, for the particular problem of debugging, you can make sure to copy over files from the deployment in which the bug has been discovered, or possibly include a full "sincerity dependencies" dump with the bug report, allowing developers to precisely replicate its environment.

The second problem is that because you need to run "sincerity install", you would potentially be dependent on third-party repositories (Three Crickets, Maven Central, PyPI) to turn your VCS repository into a runnable system. A good solution is to use a repository proxy, such as Nexus (page 38), that would guarantee that you control access to all binaries within your organization, even if the third party repositories fail.

Working with Docker

Machine virtualization brought about a revolution in deployment strategies. And then came LXC, providing a more limited set of features via built-in isolation features in Linux: think chroot, but with filesystem and networking containment. LXC allows for much lighter containers as compared to virtualization.

So lightweight, in fact, that it makes sense to package and distribute applications via LXC. That's exactly what Docker does, by providing an easy-to-use set of tools, standardized packaging, repository management, and a curated catalog of ready-to-run base images. Many workload distribution systems, such as Mesos, support Docker packages, allowing you to deploy applications with exceptional flexibility, robustness, and economical utilization of resources.

(It's also interesting to see LXC encroaching into the data center market, which until now was dominated virtualization: LXD will allow you to manage a cloud of "machines" that are actually LXC containers, offering much greater density on existing hardware. It will even integrate with OpenStack, allowing data centers a smooth transition to this exciting technology.)

Running in Docker

Because Sincerity puts your entire container in one root directory, it's trivial to run your Sincerity container in a Docker image. In this example, we'll create a container with the Prudence example application, and then run it inside the ready-made "java" Docker image:

```
sincerity create /path/to/mycontainer : add prudence.example : install

sudo docker run --rm -it \
-v /path/to/mycontainer/:/opt/mycontainer/ \
-p 8080:8080 \
java:8u45-jre \
/opt/mycontainer/sincerity use /opt/mycontainer/ : start prudence
```

If you haven't used the "java:8u45-jre" Docker image yet, it will have to download it.

In this example, we've mapped our Sincerity container to "/opt/sincerity/" in the Docker image, and Prudence's default HTTP port to a port in the host, so that we could access the site at http://localhost:8080/. We've also enabled an interactive pseudo-TTY ("-it") so that we can press CTRL+C to quit.

The result seems identical to running "normally": and that's the beauty of Docker.

What good is this? Well, for one, it allows you to easily test your Sincerity container in various versions of the JVM without having to install them on your main operating system. But also, Docker can offer tighter security more easily than just, for example, running your Sincerity container under a custom user.

Packaging in Docker

Once you've tested your Sincerity container in Docker, it's time to package it for deployment.

First, create a "Dockerfile" in your container's directory. For our example:

```
FROM java:8u45-jre
MAINTAINER Three Crickets
ADD . /opt/mycontainer/
CMD /opt/mycontainer/sincerity use /opt/mycontainer/ : start prudence
EXPOSE 8080
```

You'll also want to create a ".dockerignore" file (which uses the same syntax as ".gitignore"). For our example:

```
/cache
/logs
```

Now we can build it:

```
sudo docker build -t threecrickets:mycontainer .
```

That's it! It was very fast, because Docker uses a transaction system: our new package is only a small diff over the original image. Running it is very similar to before:

```
sudo docker run -it -p 8080:8080 threecrickets:mycontainer
```

You can also run it in "detached" mode (like a daemon) using "docker run -d". Use "docker ps" to list existing running images, and "docker stop" to stop any.

To save the image into a self-contained, redistributable file:

```
sudo docker save threecrickets:mycontainer | bzip2 > mycontainer.tar.bz2
```

To load it:

```
cat mycontainer.tar.bz2 | bunzip2 | sudo docker load
```

Note that because the image is self-contained, the environment loading it does not need access to the repository where "java:8u45-jre" came from (it essentially includes the JVM). However, because all transactions have GUIDs, it would be identical to having retrieved "java:8u45-jre". So, if that environment were to be running 100 images based on "java:8u45-jre", it would only keep the actual installation once. (You can also "flatten" your image, as if it were a single commit, using "docker export".)

Also note that "save" does not keep the tags, though you can re-tag the image via its ID like so:

```
sudo docker images
...
sudo docker tag ... threecrickets:mycontainer
```

See the Docker documentation for more information about how to work with repositories.

FAQ

Please also refer to the FAQ for Scripturian.

The wrong version of a dependency is being installed. Why, and how do I fix it?

First, diagnose what is going on by viewing the dependency tree, via either the "dependencies:dependencies" command (page 21) or the GUI.

If the problem is with an *explicit* dependency that you added, it could be that it is also being included as an *implicit* dependency with different version restrictions, and Ivy has done its best to resolve the conflict within the restrictions. You can overcome Ivy's compromise by using the "–force" when adding the explicit dependency. For example:

```
sincerity add com.tanukisoftware wrapper-linux 3.5.20 --force
```

If the problem is with an *implicit* dependency, you can override the version by using the "dependencies:override" command (page 22). For example:

```
sincerity override com.tanukisoftware wrapper-linux 3.5.20
```

Another option is to use the "–only" switch when adding the explicit dependency that pulls in the wrong implicit dependency, and then explicitly adding the sub-dependencies in the versions you want. You can, in fact, only use "–only" for *all* your adds, making 100% sure that only explicit dependencies are used.

I'm getting "java.lang.OutOfMemoryError: PermGen space" exceptions!

This is likely because you are chaining several of Sincerity commands together while also using the "heavier" language engines (Jython, JRuby). The easy solution in most cases is simply separating your commands. For example, instead of this:

```
sincerity add rails : install : add django : install : start django
```

Run this:

```
sincerity add rails : install
sincerity add django : install
sincerity start django
```

If you're using the Oracle JVM, you can also increase the PermGen space by setting the JVM_SWITCHES environment variable (page 12) before running Sincerity:

```
JVM_SWITCHES=-XX:MaxPermSize=128m sincerity ...
```

This problem should completely disappear in JVM 8, which removes the PermGen feature entirely.

How do I force the use of Rhino with JVM 8?

There is an environment variable (page 12) for it:

```
SINCERITY_JAVASCRIPT=Rhino sincerity install
```

By default, Sincerity will prefer Nashorn, even if Rhino is also on the classpath.

Part II
Ecosystem

Core Plugins

These are the plugins that come with the Sincerity installation and implement its most essential commands.

Since these commands are used so often, it's a good idea to avoid implementing these command names in your own custom plugins, so that there would never be ambiguity for the essentials. In other words, treat "add", "install", etc. as reserved command names.

We've organized them here in the order by which you'd likely use them.

Optional arguments are marked by square brackets.

Container

Manages Sincerity containers.

container:create

Creates a new container using a container template and points Sincerity to it, making it the new current container.

Arguments

1. **Container root directory**: If the directory does not exist, this command will create a new container there. If the directory already exists and is a container, points Sincerity at it.
2. **[Template name]**: This is the name of a subdirectory under your Sincerity installation's "/templates/" subdirectory. Will default to "default". Sincerity will recursively copy the files from the template into your new container. Use the "templates:templates" command (page 26) to see available templates.

Switches

- **–force**: With this switch, even if the directory already exists, the command would *still* copy the template into it. Note that this might overwrite existing files.

container:use

Changes the current container to which Sincerity is pointing.

Arguments

1. **Container root directory**: The path must point to a valid container root, meaning that it must have a "/.sincerity/" subdirectory.

container:clone

Creates a clone of the current container.

Arguments

1. **Target container root directory**: If the directory does not exist, this command will create a new container there, recursively copying all files from the current container to it. Note that Sincerity will *not* switch to the new container: use the "container:use" command if you need to do that.

Switches

- **–force**: With this switch, even if the target directory already exists, the command would *still* copy the files into it. Note that this might overwrite existing files.

container:clean

This command is the same as "artifacts:uninstall" (page 24) but also deletes the "/cache/" subdirectory.

Repositories

Manages repositories within the current container. Adds a "Repositories" tab to the Sincerity GUI.

The "artifacts:install" (page 23) command searches for dependencies in all attached repositories, *in order*.

Instead of using these commands, you can also edit the container's "/configuration/sincerity/repositories.conf" file directly. See the Ivy documentation for resolvers.

repositories:repositories

List all repositories attached to the current container in order by section.

repositories:attach

Attaches (adds) a repository to the current container. This command modifies the "/configuration/sincerity/repositories.conf" file.

Arguments

1. **Section**: Repositories are searched in the order they are added, but are first ordered by *section*. By default Sincerity containers have two sections: "private" and then "public", in that order. Thus, any repositories you attach to the "private" section will be searched before any repositories attached in the "public" section.
2. **Name**: The repository name must be unique *per its section*.
3. **Type**: Sincerity currently supports two types of repositories: "maven" (you can also use the "ibiblio" alias) and "pypi" (you can also use the "python" alias).

Arguments after the first three depend on the type of repository attached. However, both currently supported types require one additional argument: the repository base URL.

Note that this command supports implicit shortcuts that begin with the "attach#" prefix. For example:

```
sincerity attach maven-central
```

Will expand to:

```
sincerity attach public maven-central maven http://repo1.maven.org/maven2/
```

If the following entry is in "shortcuts.conf":

```
attach#maven-central = attach public maven-central maven http://repo1.maven.org/maven2/
```

To see all available "attach#" shortcuts in your container use the "shortcuts:shortcuts" command.

repositories:detach

Detaches (removes) a repository from the current container. This command modifies the "/configuration/sincerity/repositories.conf" file.

Arguments

1. **Section**: See the "repositories:attach" command.
2. **Name**: See the "repositories:attach" command.

Dependencies

Manages dependencies for the current container. Adds "Dependencies" and "Licenses" tabs to the Sincerity GUI.

Instead of using these commands, you can also edit the container's "/configuration/sincerity/dependencies.conf" file directly. See the Ivy documentation for dependencies.

dependencies:dependencies

Lists all dependencies in the current container as a tree structure. Dependencies that are *not* installed will be listed in parentheses.

For a large dependency tree, it may be easier to use the Sincerity GUI instead of this command.

dependencies:licenses

Lists all licenses per all dependencies in the current container. Note that dependencies may be available via more than one license.

> Please be aware that you should not treat the output of this command as legal advice. Package maintainers do their best to provide you with correct and useful information, but you should yourself investigate the licensing available per each library you use to avoid breaking the law.

Switches

- **–verbose**: By default only the name of the license will be printed. With this switch, the URL will be printed, too, if available.

dependencies:add

Adds a dependency to the current container. *Note that this does not download artifacts*: all it does is modify the "/configuration/sincerity/dependencies.conf" file.

The reason this command doesn't install files is that installation requires a resolution phase that goes over all dependencies and their sub-dependencies and selected the highest possible versions of dependencies. Use "artifacts:install" (page 23) to download and install artifacts. It will also delete artifacts no longer used in the revised dependency tree.

Arguments

1. **Group**: The dependency group name. This is sometimes also called an "organization," though it may be a bit misleading, because a group can refer to a set of products within an organization. Group names tend to follow the Java package naming format. For example, Prudence's group name is "com.threecrickets.prudence". Unfortunately, group names are not standardized and many projects follow their own conventions.
2. **Name**: This is the name of the dependency within the group. It is usually a simple string, possibly with dashes, the project. For example, the name for the "Prudence Example" application within the Prudence group is "prudence-example".
3. **[Version]**: If you do not specify a specific version (or use the special "latest" string), Sincerity will resolve for the highest available version. Sincerity supports range specifications for versions. For example. "[1.0,2.0[" will match versions that are greater than or equal to 1.0 but lesser than 2.0.

Switches

- **–only**: Ignores all implicit dependencies of this dependency
- **–force**: Forces the specified version, even if a different version is preferred by a different dependency

  Important: Sincerity uses Ivy's dynamic revision format for versions, which look similar to Maven's but is in fact interpreted quite differently. This is a cause for many mistakes in using version constraints in Sincerity!

Note that this command supports implicit shortcuts that begin with the "add#" prefix. For example:

```
sincerity add velocity 1.7
```

Will expand to:

```
sincerity attach three-crickets : add org.apache.velocity velocity 1.7
```

If the following entry is in "shortcuts.conf":

```
add#velocity = attach three-crickets : add com.org.apache.velocity velocity
```

To see all available "add#" shortcuts in your container use the "shortcuts:shortcuts" command.

dependencies:revise

Allows you to change the version of a previously added dependency. The format is identical to "dependencies:add" (page 21): the difference is that a new dependency cannot be added with this command, only revised. For example:

```
sincerity add org.apache.velocity velocity 1.7
sincerity revise org.apache.velocity velocity latest
```

dependencies:remove

Removes a dependency from the current container. *Note that this does not delete installed artifacts*: all it does is modify the "/configuration/sincerity/dependencies.conf" file.

Use "artifacts:install" (page 23) or "artifacts:prune" (page 24) to delete artifacts no longer used in the revised dependency tree, or "artifacts:uninstall" (page 24) to delete all artifacts.

Arguments

1. **Group**: The dependency group name. This is sometimes also called an "organization," though it may be a bit misleading, because a group can refer to a set of products within an organization. Group names tend to follow the Java package naming format. For example, Prudence's group name is "com.threecrickets.prudence". Unfortunately, group names are not standardized and many projects follow their own conventions.
2. **Name**: This is the name of the dependency within the group. It is usually a simple string, possibly with dashes, the project. For example, the name for the "Prudence Example" application within the Prudence group is "prudence-example".

For example:

```
sincerity add org.apache.velocity velocity 1.7
sincerity remove org.apache.velocity velocity
```

dependencies:exclude

Forcibly excludes an implicit dependency from being downloaded. The format is identical to "dependencies:remove" (page 22). For example:

```
sincerity exclude org.apache.velocity velocity
```

dependencies:override

Overrides the version of an implicit dependency. Note that this does not actually add the dependency. If the dependency is not in the tree, then the override has no effect. The format is identical to "dependencies:add" (page 21):

```
sincerity override org.apache.velocity velocity 1.6
```

dependencies:freeze

Overrides the versions of all explicit and implicit dependencies to be their *currently installed* versions. This ensure that future runs of "artifacts:install" (page 23) will result in exactly the same version installations.

dependencies:reset

Removes *all* dependencies from the current container. *Note that this does not delete installed artifacts*: all it does is empty the "/configuration/sincerity/dependencies.conf" file. Use "artifacts:uninstall" (page 24) to delete installed files.

Artifacts

Manages artifacts in the current container. Adds an "Artifacts" tab to the Sincerity GUI.

artifacts:artifacts

Lists artifacts available for each dependency of the current container. If the dependency is *not* installed, it will be listed in parentheses.

Switches

- **–packages**: Shows artifacts within packages (by default these are *not* shown).
- **–verbose**: In non-verbose mode (the default) only the type of artifact is shown. In verbose mode you'll see the complete relative path to the artifact as well as its size in bytes.

artifacts:install

This powerful command downloads and installs artifacts belonging to the current container's dependencies and their sub-dependencies from the online repositories to which the container is attached. It should thus be used *after* "dependencies:add" (page 21) and "repositories:attach" (page 20) have been used. This command also handles upgrades and resolves dependency version conflicts.

Installation happens in seven phases:

1. **Checking**: First, it searches for your dependencies in all attached repositories *in order*. It uses the "/configuration/sincerity/dependencies.conf" and "/configuration/sincerity/repositories.conf" files as a starting point. Though you can edit these directly (or copy them from elsewhere), you may prefer to use the "dependencies:add" and "repositories:attach" commands to manipulate them instead.
2. **Meta-data**: When found, the meta-data (in Maven this is a .pom file) for the package is downloaded and stored in a local cache (under "/cache/sincerity/packages/").
3. **Recursion**: If your dependency has sub-dependencies, they are added. Phases 1 to 3 are repeated for each.
4. **Resolution**: Now that we have a complete dependency tree, it will be "resolved." This means that duplicate dependencies will be skipped and highest possible versions for dependencies will be selected. Note that upgrades are handled by this phase: if a newer version of a certain dependency is found, it will be selected instead of the previously installed one. Rarely, this phase may fail with an error due to version conflicts that cannot be resolved.
5. **Download/Delete**: The dependencies selected for installation in the resolution phase will be downloaded, and those that were previously installed and are no longer needed (for an upgrade) will be deleted. You can use the "dependencies:dependencies" command to see the whole dependency tree, including dependencies that were selected to not be installed by the resolution phase. A more detailed report will be made available in "/cache/sincerity/resolution/threecrickets-sincerity-container-default.xml". Note that this XML report uses XSL and CSS to make it nicely readable in a web browser.
6. **Unpack**: Installed packages will be unpacked. This is identical to the "packages:unpack" command, and may cause new artifacts to appear in your container, as well as arbitrary code to be executed via installer hooks in a package. It will not by default overwrite existing files.
7. **Prune**: Unused artifacts will be deleted, *unless you have changed them*. This is identical to the "artifacts:prune" command.

Under the hood, Sincerity relies on Ivy to handle phases 1 to 5, and it may be useful to refer to its documentation if you require specialized configuration and handling.

Note that remote repositories introduce consider delay for phases 1, 2 and 5. Furthermore, the more repositories you attach, the longer phase 1 will take, as each repository is checked in sequence. For these reasons, as well as saving you from network/server failure by 3rd party providers, it is strongly recommend that you run a local proxy for the repositories you use. You can install one easily with Sincerity using the Nexus skeleton.

Switches

- **–overwrite**: This affects phase 6. See "packages:unpack" (page 24) for more information.
- **–verify**: This affects phase 6. See "packages:unpack" (page 24) for more information.

artifacts:uninstall

This command deletes all artifacts installed by "artifacts:install" (page 23), *unless these artifacts have been changed since they've been installed.* This behavior ensures that you do not lose your custom work. If a package has an uninstaller hook, it will be executed after its artifacts are deleted.

This command is useful for leaving your container clean of any dependencies. Though the artifacts are deleted, they are still added to your container. Thus, this command is entirely reversible by issuing a "artifacts:install" command.

Also see the "container:clean" command (page 20).

artifacts:prune

Deletes artifacts that were previously installed by "artifacts:install" (page 23) but for which their dependencies no longer exist. *Artifacts that were changed since installation will not be deleted.* This behavior ensures that you do not lose your custom work.

You usually would not have to run this command by itself, because it is part of "artifacts:install". However, it may be useful in case you are manipulating the contents of "/libraries/jars/" manually.

Packages

Manages packages in the current container.

packages:unpack

Unpacks all Sincerity packages in "/libraries/jars/". If a package has an installer hook, it will be executed after its artifacts are unpacked.

You usually would not have to run this command by itself, because it is part of "artifacts:install" (page 23). However, it may be useful in case you are manipulating the contents of "/libraries/jars/" manually.

It is also useful in case you've made various changes to unpacked artifacts and want to restore them to their initial unpacked state. A good way to do this is to delete all the files that you want to restore and then run "sincerity unpack".

Arguments

1. **[Filter]**: Currently unused.

Switches

- **–overwrite**: By default the command will not overwrite existing files, unless these files were previously installed by Sincerity and have not been modified since. This behavior ensures that you do not lose your custom work. However, you can override this behavior using this switch. *Be careful:* this will overwrite files unpacked by *all* packages. If you only want to overwrite a select few files, it is best to delete them and then run "unpack" *without* this switch.
- **–verify**: Verifies that artifacts have been unpacked correctly. Slower but safer.

Delegate

Manages entry points into the current container, and is the primary means to run applications in it. Adds a "Programs" tab to the Sincerity GUI.

delegate:main

Calls the main() method within a JVM class. The class may exist anywhere within the current container, within the Sincerity installation, or elsewhere in the JVM classpath.

Arguments

1. **Classname**: This is the fully qualified JVM class name. For example, "org.myapplication.Service". Note that the class has to have a method named "main" with the correct signature (public, returns void, with an array of strings as its only argument).

Additional command arguments after the first will be sent as arguments to the main() method.

delegate:start

Executes a Scripturian (page 50) document.

Though Sincerity comes with support for JavaScript, documents can be written in any installed programming language that support Scripturian. The document extension will tell Scripturian which language engine to use: ".js" for JavaScript, ".py" for Python, ".rb" for Ruby, etc. Use the "delegate:languages" command (page 26) to list all supported languages.

Your code will have full access to Sincerity's execution environment and API. See the chapter on Programming for more information.

Scripturian will store compiled code in the "/cache/" subdirectory, speeding up subsequent runs. For example, JavaScript classes will be stored under "/cache/javascript/". You can safely delete this files.

Arguments

1. **Document name**: The argument is a document name, either beginning with a "/" and relative to the current container root, or a simple string specifying a filename in the "/programs/" subdirectory. Use the "delegate:programs" command to list all files under "/programs/". As usual with Scripturian, filename extensions should not be used. If the name points to a directory, then a file named "default" (with the appropriate programming language extension) in that directory will be executed. For example, "sincerity start /component/" would execute "/component/default.js" from within the current container, while "sincerity start component" would execute "/programs/component.js".

Additional command arguments after the first will be ignored by Sincerity, but will be forwarded to the program and can be accessed from within its code using the application.arguments API.

delegate:execute

This command starts a new process, which would be a child process of Sincerity. Standard output and input from the child process are piped to the current standard output and input.

This command is useful not only for integrating non-JVM code into Sincerity, but also for hashtag support, allowing you to incorporate dynamic language scripts, for Python, Ruby, etc. Since Sincerity controls the environment of the child process, it can guarantee that environment variables and other properties are set according to the current container.

Arguments

1. **Executable name**: The argument is a filename relative to the current container's "/executables/" subdirectory. The file must be executable by the underlying operating system. On *nix this includes support for hashtag script files.

Switches

- **–background**: By default this command will block until the child process exits. However, using this switch Sincerity will not block and continue processing its command chain. Note that this would *not* stop the child process from ending when the Sincerity parent process ends.

delegate:programs

Lists all available programs in the current container (documents in the "/programs/" subdirectory). Use the "delegate:start" command (page 25) to start them.

delegate:languages

Lists all languages installed in the current container that support Scripturian (page 50).

Templates

Manages templates in the Sincerity installation. Adds a "Templates" tab to the Sincerity GUI.

Templates are used by the "container:create" command (page 19) to initialize new containers.

templates:templates

Lists all templates available in the Sincerity installation.

templates:templatize

Turns the current container into a Sincerity template. This works by simply recursively copying the current container into your Sincerity installation's "/templates/" subdirectory. Note that you must have write permissions there in order for this to work.

Note that you can manipulate the "/templates/" subdirectory directly. This command is merely for convenience.

Arguments

1. **Template name**: A new subdirectory to be created under your Sincerity installation's "/templates/" subdirectory. Note that this command will *not* copy over an existing template! If the directory already exists, you will get an error. You must manually delete the directory if you want to change an existing template using this command.

Shortcuts

Manages shortcuts for the current container. Adds a "Shortcuts" tab to the Sincerity GUI.

Your shortcuts are defined in your container's "/configuration/sincerity/shortcuts.conf" file.

shortcuts:shortcuts

Lists all available shortcuts in the current container.

Help

Provides general information about your Sincerity installation. Adds a "Commands" tab to the Sincerity GUI.

help:version

Lists Sincerity version information. This includes the numerical version and the build timestamp. An example of output:

```
built=Jun 18 2013, 15:28:46, TZ+0800
version=1.0-dev5
```

help:help

Lists all available Sincerity commands (in full form) from all available Sincerity plugins. This includes both plugins installed in the current container and those available in the Sincerity installation.

help:verbosity

If no argument is provided, prints out the current Sincerity output verbosity. If an argument is provided (integer >=0) then changes the current verbosity. Note that the default verbosity is 1, and you can change the verbosity several times within a chained Sincerity command.

Verbosity is interpreted individually by individual commands, though 0 usually means "silent," 1 means "only important messages" and 2 means "quite chatty." Higher values usually include more minute debugging information.

Note that verbosity is *only* used to control messages to standard output and standard error. Configuring logging should be done separately, via the logging plugin (page 32).

Shell

User interfaces to Sincerity.

shell:console

A straightforward console in which you can run Sincerity commands. The console supports basic command completion using the TAB key and persistent command history using the UP and DOWN keys.

Use "exit" (or CTRL+C) to exit the console. Use "reset" to reset the command history. The history is available in the "/cache/shell/console.history" file.

See "jsshell:jsconsole" (page 27) for a richer console, in which you can use full JavaScript code.

Switches

- **–script**=: If present, the console will load this script file, run it one line at a time, and then exit. Empty lines and lines beginning with a "#" (comments) will be ignored. In the script, you may separate commands via ":" or a newline, with the same final effect.

shell:gui

Starts the Sincerity Sincerity GUI (page 12). The GUI will go through all available plugins and try to call the optional gui() entry point if they have them, allowing plugins to enhance the GUI as is appropriate.

Note that this command blocks until the GUI is shut down.

Switches

- **–ui**=: Let's you change the Swing look-and-feel. Look-and-feels supported on most JVMs are: "metal" and "nimbus". Note that if no look-and-feel is specified, or the specified look-and-feel is not found, then Sincerity will attempt to default to the native look-and-feel, *unless the native platform is GTK*. We found the GTK look-and-feel to be so riddled with bugs that we decided to spare you from it.

JavaScript Shell

User interfaces to Sincerity using JavaScript.

jsshell:jsconsole

A JavaScript console in which you can run JavaScript code, with full access to all JavaScript and JVM libraries in the container. The console supports basic command completion using the TAB key and persistent command history using the UP and DOWN keys.

As a shortcut, any line beginning with a ":" will execute a Sincerity command, similar to using the basic "shell:console" (page 27).

Use "exit" (or CTRL+C) to exit the console. Use "reset" to reset the command history. The history is available in the "/cache/jsshell/jsconsole.history" file.

Switches

- **–script**=: If present, the console will load this script file, run it *all at once*, and then exit. Note that you cannot use the ":" shortcut to run Sincerity commands here, because this file is pure JavaScript. However, you can run Sincerity commands using sincerity.run(...) calls.

Java

Support for the Java programming language.

java:compile

Compiles Java source files (".java") into JVM class files (".class") using the current container's classpath.

Note that you must have a full JDK to use this command: a JRE usually does not come with a Java compiler.

Arguments

1. **[Source directory]**: Recursively compiles all ".java" files in this directory (relative to the container root). Defaults to "/libraries/java/".
2. **[Classes directory]**: Output ".class" files here. Defaults to "/libraries/classes/". Note that Sincerity will always include this directory in its classpath, so it may be a good idea to keep this default.

Language Plugins

These plugins add a language engine to your container. In some cases, this also means support for standard tools that come with the language distribution, such as a CLI, a REPL, and tools for compilation and packaging.

Most of these language engines support the Scripturian (page 50) standard, meaning that with a language plugin installed you can:

- Write Scripturian programs and Sincerity programs in this language. For example, with Python installed, you can write a "/programs/fish.py" program and start it via "sincerity start fish". Note that the service plugin (page 34) can also be used to run programs as daemons or services.
- Write Sincerity plugins in this language. For example, with Ruby installed, you can write a "/libraries/scripturian/plugins/fish.rb" plugin.

The "delegate:languages" command (page 26) will list all Scripturian-supported languages in the container.

JavaScript Plugin

Though JavaScript was originally designed to be run in web browsers, it is a powerful general-purpose C-syntax language with Scheme-like closures that supports many programming paradigms, and has proved useful and popular outside the browser. Sincerity runs JavaScript code via either Nashorn (available from JVM 8) or Rhino.

> Note: *You do not need this plugin to install JavaScript support in a Sincerity container.* All it does is provide you with a new command to get easy access to a JavaScript shell.

To install:

```
sincerity add javascript : install
```

To start a shell:

```
sincerity javascript
```

Fleshing Out

The shell can run script files and also evaluate inline scripts as arguments. Use "sincerity javascript -h" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add javascript : install : javascript -e "print('Hello, world'
```

Note that this "javascript" does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run JavaScript files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

Python Plugin

Python is a general-purpose multi-paradigm dynamic language with an exceptionally clean syntax and a rich ecosystem. Sincerity implements Python via Jython, and also has limited support for Jepp.

To install:

```
sincerity add python : install
```

To start a shell:

```
sincerity python
```

> Ruby note: Due to conflicts in their implementations, you cannot currently use the Python and Ruby plugins in the same container.

Fleshing Out

The shell can run script files and also evaluate inline scripts as arguments. Use "sincerity python -h" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add python : install : python -c "print 'Hello, world'"
```

Note that this "python" command does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run Python files in Sincerity's Scripturian environment use the "delegate:start" command.

Python has a very extensive ecosystem hosted on PyPI (a.k.a. "The Cheese Factory") in "egg" format. You can install libraries, frameworks and applications into your Sincerity container using a special version of "easy_install" included in this plugin as a Sincerity command. For example, let's install Beej's Flickr API:

```
sincerity easy_install flickrapi
```

Eggs will be installed into your container under the "/libraries/python/Lib/site-packages/" subdirectory.

Note that not all software written for CPython runs well on the Jython engine. See the software's documentation for more details.

The Sincerity Python plugin also include a "python" command (under "/executables/python") to allow for proper integration with Python software that starts Python subprocesses. You can run this command directly, and even use it with a shebang for executable files. For example, this file is executable:

```
#!/path/to/mycontainer/executables/python
print 'hello world'
```

You can also place such files in your "/executables/" subdirectory and run them using Sincerity's "delegate:execute" command (page 25).

Ruby Plugin

Ruby is a general-purpose multi-paradigm dynamic language with a exceptionally full set of features and a rich ecosystem.

Sincerity implements Ruby via JRuby, an exceptionally robust implementation.

To install:

```
sincerity add ruby : install
```

To start a shell:

```
sincerity ruby
```

> Python note: Due to conflicts in their implementations, you cannot currently use the Python and Ruby plugins in the same container.

Fleshing Out

The shell can run script files and also evaluate inline scripts as arguments. Use "sincerity ruby -h" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add ruby : install : ruby -e "puts 'Hello, world'"
```

Note that this "ruby" command does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run Ruby files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

Ruby has a very extensive ecosystem hosted on RubyGems in "gem" format. You can install libraries, frameworks and applications into your Sincerity container using a version of "gem" included in this plugin as a Sincerity command. For example, let's install Flickraw, an API for accessing Flickr:

```
sincerity gem install flickraw
```

Gems will be installed into your container under the "/libraries/ruby/lib/ruby/gems/shared/" subdirectory.

Note that not all software written for Ruby runs well on the JRuby engine (though in some cases it may actually run *better* in JRuby). See the software's documentation for more details.

Other standard Ruby commands supported by the plugin are: "ast", "irb", "rake", "rdoc", "ri" and "testrb".

The Sincerity Ruby plugin makes sure that the execution environment will work with the JRuby ecosystem. Specifically, JRuby executable files start with the "env" shebang, for example:

```
#!/usr/bin/env jruby
puts 'Hello, world'
```

You can place such files in your "/executables/" subdirectory and run them using Sincerity's "delegate:execute" command (page 25).

PHP Plugin

Though PHP was designed for generating web pages, it is also useful as a general-purpose templating language. Sincerity implements PHP via Quercus. Note that the free version of Quercus is included, but you may easily swap it for a purchased professional release if you have it.

To install:

```
sincerity add php : install
```

To start a shell:

```
sincerity php
```

Fleshing Out

The shell can run script files provided as arguments. Use "sincerity php -h" for more information. For example, let's create a file named "test.php":

```
<?php
print "Hello, World!\n";
?>
```

And then run it like so:

```
sincerity create mycontainer : add php : install : php test.php
```

Note that this "php" command does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run PHP files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

PHP has a very extensive ecosystem hosted on PEAR, often in PHP archive (.phar) format. Though Sincerity does not yet support PEAR directly, you can install PEAR libraries using standard PHP and then copy them over to your Sincerity container.

Lua Plugin

Lua is an especially lightweight multi-paradigm dynamic language, which shares many features with JavaScript, but is nevertheless simpler to implement due to its minimalist design. The simple implementation allows for a register- rather than stack-based virtual machine and famously fast performance. Sincerity implements Lua via Luaj, which outperforms even the standard Lua in many situations and allows integration with JVM libraries.

To install:

```
sincerity add lua : install
```

To start a shell:

```
sincerity lua
```

Fleshing Out

The shell can execute Lua files provides as arguments, and also evaluate inline scripts as arguments. Use "sincerity lua -h" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add lua : install : lua -e "print 'Hello, world'"
```

Note that this "lua" command does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run Lua files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

Additionally, the plugin supports a "luac" command to compile Lua source files into portable Lua bytecode, and a "luajc" command to compile into JVM classes.

Groovy Plugin

Groovy is a dynamic language with a syntax familiar to Java programmers, but with features inspired by Python, Ruby and Smalltalk. It provides exceptionally good integration with libraries written in Java, such that any JVM library is immediately also a Groovy library.

To install:

```
sincerity add groovy : install
```

To start shell:

```
sincerity groovy
```

Note that the Groovy plugin requires at least JVM 7 by default, because it depends on the invokedynamic version of Groovy. If you need to run on JVM 6, you can switch to the non-invokedynamic version with the following command:

```
sincerity exclude org.codehaus.groovy groovy-indy : add org.codehaus.groovy groovy : install
```

Fleshing Out

The shell can run script files and also evaluate inline scripts as arguments. Use "sincerity groovy" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add groovy : install : groovy -e "println 'Hello, world'"
```

Note that this "groovy" command does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run Groovy files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

Clojure Plugin

Clojure is a modern Lisp designed for concurrency and performance. It is a superbly expressive language that supports robust functional programming as well as other paradigms.

To install:

```
sincerity add clojure : install
```

To start a REPL:

```
sincerity clojure
```

Fleshing Out

The REPL can run script files and also evaluate inline scripts as arguments. Use "sincerity clojure -h" to see the command's possible arguments. An example of an inline script:

```
sincerity create mycontainer : add clojure : install : clojure -e '(println "Hello, World")'
```

Note that the REPL does not use Scripturian (page 50), nor does it have access to any Sincerity APIs. To run Clojure files in Sincerity's Scripturian environment use the "delegate:start" command (page 25).

Clojure has a very extensive ecosystem hosted on Clojars. It is a standard Maven-type repository that is naturally supported by Sincerity, and attached by default if you use the "add clojure" shortcut. For example, let's install the flickr-clj, an API to access Flickr:

```
sincerity add clojure : add org.clojars.stanistan flickr-clj : install
```

A shortcut is also available for attaching Clojars explicitly:

```
sincerity attach clojars : attach maven-central
```

Note that many Clojars libraries also rely on Maven Central, so it's a good idea to attach it as well. Both are attached when you use the "add clojure" shortcut.

Feature Plugins

Sincerity Standalone Plugin

See also the redistribution plugin for a different approach to distribution.

Logging Plugin

This plugin makes it exceptionally easy to unify and configure your logging across a diverse set of technologies and dependencies. In most cases, simply installing this plugin into your container should handle all logging with sensible defaults. Should you need to customize and configure logging, you'll find Sincerity's scheme especially flexible and powerful.

Though the JVM includes a standard logging API, in the "java.util.logging Interface" ("JULI") package, the greater JVM ecology has adopted a few incompatible standards. In particular, Apache Log4j, which was the inspiration for JULI, enjoys broad support and a more robust implementation. Especially since Log4j 2.0, it provides state-of-the-art scalability for high loads using innovative asynchronous handling. We've thus preferred to use Log4j for our actual implementation, and rely on the excellent SLF4J library for bridging JULI to it. SLF4J has become popular enough that several libraries support it directly, so that we can avoid even the minimal overhead introduced by bridging.

To install:

```
sincerity add logging : install
```

To initialize logging, you can execute the "logging" command from within your programs. An example in JavaScript:

```
// Will do nothing if the logging plugin is not installed:
try { sincerity.run('logging:logging') } catch(x) {}
```

You can also test logging simply using the "log" command, which sends an "info" level message to the "sincerity" logger:

```
sincerity add logging : install : log "This is a test!"
```

Logs will appear under the "/logs/" directory. By default, all loggers are appended to "/logs/common.log", which is a rolling log file with a size of 5MB per file, and a maximum of 10 files.

> **Note for Restlet users:** If you're using the Restlet skeleton, it's recommended to install the Restlet skeleton logging add-on (page 45), which adds a Restlet extension library that provides direct chute to SLF4J.
>
> **High CPU usage?** On some rare combinations of operating systems and JVMs, Log4j 2.0 (well, actually the LMAX Disruptor library it uses) may use too much CPU time, even when idle. You can

reduce CPU usage in these cases, at the expense of affecting the high-scalability profile, by using the following JVM switch:

```
-DAsyncLoggerConfig.WaitStrategy=Block
```

Fleshing Out

"Officially," Log4j configuration is based either on JVM properties files or XML. Both are hardcoded, inflexible and difficult to scale. Sincerity's logging plugin instead uses a powerful JavaScript-based scheme, which allows you to dynamically configure your loggers according to your operating environment.

Configure your loggers under "/configuration/logging/loggers/" and your appenders under "/configuration/logging/appenders/". Any JavaScript file you add to these directories will be executed upon logging initialization. Take a look at the defaults to get a sense of how this works: the /sincerity/log4j/ library makes it especially easy to use.

For example, here's a definition of a rolling file appender:

```
var logFile = sincerity.container.getLogsFile('main.log')
logFile.parentFile.mkdirs()

configuration.rollingFileAppender({
        name: 'main',
        layout: {
                pattern: '%d: %-5p [%c] %m%n'
        },
        fileName: String(logFile),
        filePattern: String(logFile) + '.%i',
        policy: {
                size: '5MB'
        },
        strategy: {
                min: '1',
                max: '9'
        }
})
```

Note that you can also use "official" Log4j configuration if you are more comfortable with it. If the file "/configuration/logging.conf" is present, it will be used. This file can either be a properties file, a JSON file, or an XML file (in which case it must begin with the "<?xml" header). The plugin comes with an example "logging.conf" named "logging.alt.conf", which you can rename to "logging.conf" if you wish to use it.

The logging plugin also comes with a simple "log" command to test your logging configuration. Example usage:

```
sincerity log "Hello, log!"
```

Extras

In distributed environments, such as grids and clouds, you may prefer to centralize your logging. To aid this common use case, Sincerity comes with two logging server solutions.

Log4j Server You can create a simple Log4j TCP-based socket server, which comes with the logging plugin:

```
sincerity create logserver : add logging : install : start log4j-server
```

Note that we're creating the Log4j server in a separate container, and starting it as a separate process.

On this Log4j server, you want to configure your *actual* appenders. Then, on all your client processes, you want to disable all the appenders except the socket appender ("/configuration/logging/appenders/socket.js"). Uncomment all the code there to enable it and make it the default root appender.

The result is that all logging messages will be sent from the clients to the server, where they will be actually logged.

It's recommend to run the Log4j server with Sincerity's service plugin (page 34):

```
sincerity create logserver : add logging : add service : install : service log4j-server start
```

MongoDB Appender You can install a MongoDB-backed appender:

```
sincerity add logging.mongodb : install
```

To configure the MongoDB connection, edit "/configuration/logging/appenders/common-mongo-db.js". By default, it connects to localhost at the default port (27017) without security, and logs to database "logs", collection "common".

It is strongly recommended that you use a capped collection for your log. This guarantees both excellent write performance as well as automatic rolling. You can create it from the "mongo" shell tool like so:

```
db.createCollection('common', {capped: true, size: 100000})
```

Or, convert an existing collection to capped:

```
db.runCommand({'convertToCapped': 'common', size: 100000})
```

This plugin also provides you with a very useful tool to "tail" your central log (works only with capped collections):

```
sincerity logtail
```

Press CTRL+C to quit. To test that this works, open another terminal and send a log message:

```
sincerity log "This is a test!"
```

You can provide "logtail" with the MongoDB connection parameters:

```
sincerity logtail --uri=localhost:27017 --username=admin --password=admin123 --db=logs --coll
```

Service Plugin

This plugin lets you easily start and control any program as a daemon or service running in the background. This is achieved using Tanuki Software's excellent Java Service Wrapper (JSW). JSW deploys a native process to monitor your daemon's health, is able to detect failures and hangs, and restarts in such cases. It supports many configuration options to control the JVM process, as well as JMX-based management for the wrapper. While you can start any program using the "sincerity start" command, it is strongly recommended that you use this plugin instead for production environments. It's so well-designed, we wish it were included in the JVM!

JSW runs on an impressive array of JVM-capable operating systems: Linux, Mac OS X, Windows, Solaris, AIX, FreeBSD, HPUX, z/OS and z/Linux, supporting several 32-bit and 64-bit machine architectures for each. Of course, you do not want to install support for *all* of these platforms in your container, and so this plugin cleverly detects the underlying operating system and downloads the necessary native libraries on-demand the first time it is run. An error message will be displayed on unsupported platforms.

To install:

```
sincerity add service : install
```

To start a program as a daemon:

```
sincerity service myprogram start
```

The above assumes that you have a "/programs/myprogram.js" file. (Programs can be written in languages other than JavaScript if they are installed in your container.) See the service wrapper's log at "/logs/service-myprogram.log".

To stop the daemon:

```
sincerity service myprogram stop
```

To restart it:

```
sincerity service myprogram restart
```

To check its status:

```
sincerity service myprogram status
```

Additionally, you can run the wrapper in "console mode," which outputs the wrapper's log to the console, and lets you easily stop it using CTRL+C. This is very useful for testing and debugging:

```
sincerity service myprogram console
```

> Note that Sincerity uses the Community Edition of JSW, which is licensed under the GPL (v2). Make sure that you understand the special implications of this license if you intend to redistribute your product. Furthermore, some Windows platforms (64bit x86 and Itanium) supported by the Standard/Professional Editions are not supported by the Community Edition. A commercial license is available for purchase without these limitations. See the license guide for more information.

Fleshing Out

This plugin generates some parts of the JSW configuration on the fly, but it can furthermore merge your custom settings into this configuration. To do so, edit "/configuration/service/service.conf". In particular, you might want to control the memory profile of your JVM, or configure the wrapper's logging (which works independently of JVM logging). See the JSW documentation for a complete guide.

Additionally, this plugin provides a flexible way for you to send arguments to the wrapped JVM. Any files under "/configuration/service/jvm/" with a ".conf" extension will be merged and added. The plugin comes installed with a few sensible defaults, and additionally other plugins may add their own ".conf" files to support the service plugin. These ".conf" files all support string interpolation using any JVM system property or environment variable. For example, here's a way to add garbage collection logging:

```
-Xloggc:{sincerity.container.root}/logs/gc.log
-XX:+PrintGCDetails
-XX:+PrintTenuringDistribution
```

It may furthermore be useful to run your Sincerity service as an operating system service. On Unix-like systems, you can use a "system init script." Below is a script template you may use, meant for the Restlet skeleton (page 44). It adds a special "###" comment block used by Linux's Standard Base (LSB) specification. Let's name it "/etc/init.d/restlet":

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:          restlet
# Required-Start:    $local_fs $remote_fs $network $syslog
# Required-Stop:     $local_fs $remote_fs $network $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: starts the Restlet component
# Description:       starts the Restlet component using start-stop-daemon
### END INIT INFO

SINCERITY=/path/to/sincerity/sincerity
CONTAINER=/path/to/container
SERVICE=restlet
OWNER=myuser

COMMAND=$1

sudo -u "$OWNER" "$SINCERITY" use "$CONTAINER" : service "$SERVICE" "$COMMAND"
exit 0
```

For consistency, make sure to change the "Provides:" entry to match the name of the file.

You can start/stop this service by running the above script directly, for example: "sudo /etc/init.d/restlet start". On some operating systems, you may also use your "service" command, for example: "sudo service restlet start".

Make sure to edit the variables to point to the paths on your system. Note that the "OWNER" user will be used to run your service, and that for security reasons you are strongly advised *not* to use "root": it is best to create a

special user for your service, and to set its permissions according to only what it needs: read access to Sincerity and the container, and write access to the container's "/cache/", "/logs/" and other relevant directories.

To make your service start automatically when the system starts depends on your operating system. The above script should work on most Linux-based operating systems due to the "###" comment block. To process this file and the block, effectively installing the service into the operating system, run "sudo update-rc.d restlet defaults". That command uses "insserv" internally, so run "man insserv" to get documentation for the comment block format.

Extras

JMX is a powerful technology for remote monitoring and management of your JVM and applications. Local JMX (using pipes) is automatically supported, however you may also need remote JMX over the network. To add it:

```
sincerity add service.remote-jmx : install
```

The default configuration is adequate for accessing JMX via SSH tunneling, which is very secure. To create the tunnel, use the "-L" switch of SSH when connecting to your remote server:

```
ssh -L 1650:localhost:1650 -L 1651:localhost:1651 mysite.org
```

Note that we actually create *two* tunnels, one for JMX on port 1650 and one for the RMI registry on port 1651. With the tunnel in place, start VisualVM (it's included with the JDK), choose "Add JMX Connection," and use "localhost:1650" for the connection string. Note that you do *not* want to use "Add Remote Host": tunneling makes the remote host appear local. Of course, you will need to keep the tunnel open for as long as you're connected with VisualVM.

If this is your first time using VisualVM, it is recommended that you install its "VisualVM-MBean" plugin, which will, among other things, allows you to access JSW's bean ("org.tanukisoftware.wrapper") for remotely restarting your service.

You can configure remote JMX by editing "/configuration/service/jvm/remote-jmx.conf". For example, you may change the port numbers, enable authentication, and also SSL if you prefer it to SSH tunneling.

> **Important note if you are using a version of the JVM prior to 7u4:** Unfortunately, old versions of the JVM do not support the "com.sun.management.jmxremote.rmi.port" property, without which the RMI registry port is assigned randomly, thus making it difficult to use SSH tunneling. To solve this problem, this Sincerity add-on comes with a special "firewall-friendly-agent" library that allows for this functionality. You must specifically enable it in "/configuration/service/jvm/remote-jmx.conf".

Redistribution Plugin

This plugin lets you package your Sincerity container for convenient redistribution. See the Sincerity standalone plugin for a different approach to distribution.

Currently, it supports creating a powerful cross-platform JVM-based graphical installer, using the excellent IzPack library. In the future, we hope to support additional distribution media.

To install:

```
sincerity add redistribution : install
```

To use:

```
sincerity izpack [application name] [version (optional, defaults to "1.0")]
```

For example, in a single command let's create an installable Nexus repository manager with a few plugins:

```
sincerity create nexus : add nexus : add logging : add service : add redistribution : install
```

The result will be an "installer.jar" file in your container's root directory, which you can distribute. To install your application from this jar:

```
java -jar installer.jar
```

(Note that on some desktop environments double-clicking this file would also run it.)

The installed directory will contain a convenient "uninstaller.jar".

Fleshing Out

To change the license, edit "/configuration/izpack/license.txt".

The "/configuration/izpack/installer.xml" included with this plugin has sensible defaults that should work fine for many use cases, but you'll likely want to customize it. By default, it merges your Sincerity install in, and excludes IzPack itself, as well as the "/cache/" and "/logs/" directory. This guarantees that it would "just work" cleanly on any JVM with no pre-requisites.

Please see the IzPack documentation for full details. IzPack is very powerful, and can let you create modular, flexible distributions.

Markup Plugin

Need to quickly render markup text into HTML? Markdown, Confluence, MediaWiki, Twiki, Trac, Textile and Bugzilla Textile are all supported by this plugin. Markdown is supported by the Pegdown engine, and the rest by Mylyn WikiText. (While useful in itself, this plugin is intended to serve as a code example for using these libraries.)

The rendering engines themselves are not at first installed: the plugin will make sure that the engine you need is available, and install it if it's not, on demand.

To install:

```
sincerity add markup : install
```

To use:

```
sincerity render [language] [marked up source path] [rendered output path]
```

For example:

```
sincerity render markdown README.md readme.html
```

Batik SVG Plugin

Need to quickly render SVG into PDF, PNG or JPEG? This plugin uses Apache Batik to do so. (While useful in itself, this plugin is intended to serve as a code example for using Batik.)

To install:

```
sincerity add batik : install
```

To use:

```
sincerity render [SVG source path] [rendered output path]
```

The output path extension will determine the output type. For example:

```
sincerity render test.svg test.pdf
```

JsDoc Plugin

Uses JsDoc Toolkit.

"jsdoc.sincerity"

See JsDoc template (page 50).

Skeletons

You've most likely come to Sincerity for the skeletons: they provide the easiest way to get started with all kinds of frameworks, servers and libraries, while Sincerity lets you easily add more features, more libraries and more languages as your project grows.

Web Platforms

- Prudence (page 46)
- Restlet (page 44)
- Jetty: static web (page 42)
- Jetty: servlet/JSP container (page 43)

Web Frameworks

- Diligence (page 47)
- Rails (page 47)
- Django (page 48)

Databases

- OrientDB (page 40)
- H2 (page 41)

Middleware

- Hadoop (page 40)
- Solr (page 39)
- Felix (page 46)
- Nexus (page 38)

Nexus Skeleton

Sonatype's Nexus repository manager is a recommended companion for Sincerity. At its most basic, it provides you with a proxy for accessing remote repositories, such as the Three Crickets repository in which many Sincerity packages are stored. Accessing repositories via a proxy provides you with much better performance and reliability. Nexus is a very powerful tool, and learning how to use it well will can go a long way towards improving your Sincerity experience.

With Sincerity, it's a piece of cake to install a working Nexus instance:

```
sincerity add nexus : install
```

Give this a minute or two: Nexus has a *lot* of dependencies, though most are tiny.

To start the server:

```
sincerity start jetty
```

The default port is 8080, so point your browser to http://localhost:8080 to see your new Nexus repository manager. The default user is "admin" with password "admin123". You probably want to log in and change that password. Nexus provides a rich web-based interface and includes excellent documentation.

Note that the Nexus skeleton relies on the standard Jetty servlet skeleton, to which you can indeed install other "contexts" (web applications).

Fleshing Out

You may want to change the default port from 8080, which you can do by editing "/server/connectors/default.js".

Otherwise, the default configuration should be quite sensible. It includes support for the standard repositories used by Sincerity, in addition to the Nexus defaults. Logging has also been configured to adhere to Sincerity's container structure, so that logs will appear under "/logs/". Note that Nexus itself will not use Sincerity's logging plugin (page 32), but you can configure Nexus logging right in the user interface.

Extras

Two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add logging : add service : install
```

Note that the Nexus application uses its own logging implementation, which must be configured internally. However, the logging plugin (page 32) will be put to good use by the containing Jetty server.

The following command will install a Nexus repository with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add nexus : add logging : add service : install : service jett
```

To stop it:

```
sincerity use mycontainer : service jetty stop
```

Solr Skeleton

Apache Solr is a popular distributed textual search platform. It runs on the JVM and relies on the excellent Lucene library for indexing and searching, but is accessed via simple network APIs, making it perfect for distributed deployments and heavy loads. Client libraries are available for many platforms, and are even integrated into the backends of many web development frameworks, such as Django and Ruby on Rails.

With Sincerity, it's a piece of cake to install a working Solr instance:

```
sincerity add solr : install
```

To start the server:

```
sincerity start jetty
```

The default port is 8080, so point your browser to http://localhost:8080/solr/admin/ to see the main Solr administration page.

Fleshing Out

The skeleton comes with the example configuration supplied with the official Solr distribution, and should serve as a good starting point for the majority of project. The configuration is available under "/configuration/solr/conf/", and indexing and other data is stored in "/data/solr/".

Solr is very configurable, both in terms of performance fine-tuning and language analysis and indexing. It also enjoys a range of useful plugins. See the official site for more information on fleshing out your skeleton.

Extras

The logging plugin (page 32) is strongly recommended. To install it:

```
sincerity add service : install
```

Note that the logging plugin (page 32) is already included in the skeleton, because Solr relies on SLF4J.

The following command will install a Solr server with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add solr : add service : install : service jetty start
```

To stop it:

```
sincerity use mycontainer : service jetty stop
```

Hadoop Skeleton

Apache Hadoop is a powerful platform for distributed computing, well known for its scalable distributed filesystem and popular map-reduce module. It provides the underlying infrastructure for several data storage and analysis platforms, such as Cassandra, HBase, Hive and Pig.

Hadoop runs best on Linux, where it relies on native libraries. This skeleton detects the underlying architecture and downloads the necessary native libraries on-demand.

With Sincerity, it's a piece of cake to install a working Hadoop instance. Note that we need to format the node first:

```
sincerity add hadoop : install : hadoop namenode -format
```

(Note that the "namenode -format" command exits the JVM when done, so you cannot chain more commands after it.)

Then, to start the node:

```
sincerity hadoop start
```

If this is the only node in your Hadoop cluster, you will need to wait about 30 seconds for the services to fully initialize. To test copying files to and from the Hadoop filesystem:

```
sincerity hadoop fs -put myfile.txt test.txt
sincerity hadoop fs -get test.txt test.txt
```

To stop the node:

```
sincerity hadoop stop
```

To see the status of the node services:

```
sincerity hadoop status
```

Note that Hadoop uses the logging plugin (page 32) to manage the node services.

Fleshing Out

The skeleton comes with a plugin that supports the full list of Hadoop commands, and additionally supports "start", "stop" and "status" to manage the services.

All logs are under "/logs/", and the data is stored in "/data/".

To configure your instance, see "/configuration/hadoop/". The default configuration is based on that of the official Hadoop distribution on ports 8000 (name node) and 8001 (job tracker). The logging configuration is based on the logging plugin (page 32), but its essential setup has likewise been copied over from the official distribution.

OrientDB Skeleton

OrientDB is a powerful document- and graph-oriented ("NoSQL") database server designed for scalability. As a graph database, it supports the entire Tinkerpop stack, including the Gremlin graph traversal language, allowing you to easily port your application between different database implementations. For users needing features from traditional RDBMS, OrientDB also supports SQL and allows enforcing schemas on your collections.

If you're interested in a more traditional RDBMS, check out Sincerity's H2 skeleton.

The Sincerity OrientDB skeleton makes it easy to set up and run a single OrientDB instance, which can run on its own or as a node in a multi-master cluster. To install an OrientDB instance:

```
sincerity add orientdb : install
```

To start the server:

```
sincerity start orientdb
```

The default web port is 2480, so point your browser to http://localhost:2480/studio/ to see the OrientDB Studio application.

To start the console:

```
sincerity console
```

In the console, to connect to the demo "tinkerpop" database:

```
orientdb> connect remote:localhost/tinkerpop admin admin
orientdb> gremlin g.V[1]
```

Fleshing Out

The OrientDB plugin also supports a "gremlin" command to get a pure Gremlin console, though note you can also run Gremlin code in the general OrientDB console by prefixing it with the "gremlin" command.

To configure your instance, start with "/configuration/orientdb/server.conf" (XML). The default "server.conf" also references "database.conf" (JSON) and "hazelcast.conf" (XML).

Additionally, "properties.conf" (properties sheet) can be used to set JVM system properties used by OrientDB.

Databases will be stored in the "/databases/" directory in your Sincerity container.

Extras

Two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add logging : add service : install
```

The following command will install an OrientDB node with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add orientdb : add logging : add service : install : service o
```

To stop it:

```
sincerity use mycontainer : service orientdb stop
```

H2 Skeleton

H2 is a lightweight-yet-powerful relational database management system (RDBMS). It can run both as a standalone server (supporting a PostgreSQL compatibility mode), or embedded in your JVM program.

If you're interested in non-relational ("NoSQL") databases, check out Sincerity's OrientDB skeleton.

The Sincerity H2 skeleton is specifically designed to make it easy to run H2 in standalone server mode. To install an H2 instance:

```
sincerity add h2 : install
```

To start the server:

```
sincerity start h2
```

The default web port is 8082, so point your browser to http://localhost:8082/ to see the H2 Console application. Note that the web console application is useful not just for H2: it be used to connect to any JDBC URI, as long as you have the JDBC driver installed in your Sincerity container.

Fleshing Out

The H2 plugin supports all the tools that come with H2. You can use "sincerity help" to get a list of them. For example, to create a cluster:

```
sincerity create-cluster \
        -urlSource jdbc:h2:tcp://localhost:9101/test \
        -urlTarget jdbc:h2:tcp://localhost:9102/test \
        -user sa \
        -serverList localhost:9101,localhost:9102
```

To configure your server, see "/configuration/h2/server.conf". Lines that are not empty and do not begin with "#" will be added as command line arguments to the "server" tool. In fact, you can create ".conf" files for all the H2 tools if you wish to set default command arguments for them. For example, "create-cluster.conf".

By default, databases will be stored in the "/databases/" directory in your Sincerity container. However, note that H2's JDBC URI allows you to access database stored anywhere in the filesystem. If this is a security concern, you may want to consider running the H2 server in a locked-down operating system user.

Extras

Two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add logging : add service : install
```

The following command will install an H2 database server with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add h2 : add logging : add service : install : service h2 star
```

To stop it:

```
sincerity use mycontainer : service h2 stop
```

Jetty Web Server Skeleton

Need a web server for static files? No problem:

```
sincerity add jetty.web : install
```

Jetty is a very robust, modular web server with excellent asynchronous performance, and lots of features and extensions. With this skeleton we've provided you with the lightweight, bare minimum dependencies to serve just static files for a single web site.

To start you server:

```
sincerity start jetty
```

The default port is 8080, so point your browser to http://localhost:8080 to see the default welcoming page.

Jetty allows for much more sophistication than just serving a single web site, and for that we've provided a separate skeleton: "jetty.servlet". That skeleton supports multiple "contexts" under the server, as well as configuration of connectors, and of course servlets and web applications packaged as WAR files.

Additionally, Jetty is a recommended connector for Restlet. It's available as a skeleton add-on, "restlet.jetty".

> The Jetty skeletons use Jetty 9.3, which requires a JVM of at least version 8.

Fleshing Out

Just put your files under the container's "/web/" directory, using the usual rules for web servers: URLs are mapped to file paths under "/web/", and directory URLs are mapped to "index.html" files in that directory. MIME types are automatically guessed according to the common filename extensions.

You can configure the server by editing "/configuration/jetty/default.js". For example, you can change the port, enable SSL, and also HTTP/2. Note that support for SPDY must be added as an "extra" (see below).

For SSL, the example comes with a self-signed key stored in a Java KeyStore (JKS) at "/configuration/jetty/-jetty.jks". You should use it only for testing! Otherwise, you will want to create or import your own key using the "keytool" utility that is bundled with most JDKs. Here's how to create a new, unique key:

```
keytool -keystore jetty.jks -alias jetty -genkey -keyalg RSA
```

Such self-created keys are useful for controlled intranet environments, in which you can provide clients with the public key, but for Internet applications you will likely want a key created by one of the "certificate authorities" trusted by most web browsers. Some of these certificate authorities may conveniently let you download a key in JKS format. Otherwise, if they support PKCS12 format, you can use keytool (only JVM version 6 and later) to convert PKCS12 to JKS. For example:

```
keytool -importkeystore -srcstoretype PKCS12 -srckeystore mysite.pkcs12 -
    destkeystore jetty.jks
```

If your certificate authority won't even let you download PKCS12 file, you can create one from your ".key" and ".crt" (or ".pem") files using OpenSSL:

```
openssl pkcs12 -inkey /path/mykey.key -in /path/mykey.crt -export -out mysite.
    pkcs12
```

(Note that in this case you *must* give your new PKCS12 a non-empty password, or else keytool will fail with an unhelpful error message.)

Extras

Two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add logging : add service : install
```

It is also possible to add HTTP/2 support: this protocol, supported by many web browsers, can improve the user experience as well as reduce server load when using "https".

To install support for HTTP/2, as well as the required ALPN support:

```
sincerity add jetty.http2 : install
```

To enable ALPN, you need to specify your "alpn-boot.jar" in the JVM_BOOT_LIBRARIES environment variable (page 12), for example:

```
JVM_BOOT_LIBRARIES=/path/to/mycontainer/libraries/jars/org.mortbay.jetty.alpn/alpn-boot/8.1.3
sincerity : start jetty
```

It's a bit awkward, but necessary due to the way ALPN is securely implemented in the JVM.

The following command will install a web server with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add jetty.web : add logging : add service : install : service
```

To stop it:

```
sincerity use mycontainer : service jetty stop
```

Jetty Servlet/JSP Skeleton

Servlets let you generate dynamic content for a web site, usually using the Java language. There is a very large ecosystem of free servlets out there, including complete frameworks, that can help you develop dynamic applications.

To install a bare servlet skeleton, based on Jetty:

```
sincerity add jetty.servlet : install
```

To start you server:

```
sincerity start jetty
```

The default port is 8080, so point your browser to http://localhost:8080. But, you won't see anything yet: this is a bare skeleton waiting for you to add your application to it. You might want to start by installing "jetty.servlet.example" first.

Note that if you only intend to install Jetty as a simple web server for static files, then you can use a simpler skeleton: "jetty.web".

As useful as servlets are, we recommend you take a look at the Restlet skeleton (page 44) if you want to build a dynamic web application in Java. And Restlet can use Jetty as its underlying connector.

And why stop there? Prudence (page 46) builds on Restlet, letting you do all of that and more with your choice of JavaScript, Python, Ruby, PHP, Lua, Groovy or Clojure. (Disclosure: Prudence has also been created by Three Crickets.)

> The Jetty skeletons use Jetty 9.3, which requires a JVM of at least version 8.

Fleshing Out

Jetty's official distribution (which doesn't rely on Sincerity... yet) is a perfect example of why Sincerity needs to exist. "Official" Jetty configuration is a morass of XML files that effectively duplicate what a lightweight scripting language, like JavaScript does far more comprehensibly and with far greater power. If you're switching from "official" Jetty, then you're in for a treat, as well as a sigh of relief.

Configuration is handled similarly to the Jetty web server skeleton (page 42): in the same way, you can add SSL and SPDY support.

Though Jetty can use the logging plugin (see below), it also supports its own internal logging mechanism for the web (NCSA-style) log. To configure it, see "/server/services/web-log.js". By default, these logs will appear under the "/logs/web/" directory, and will be named according to the date.

Extras

Two add-ons are available: "jetty.servlet.jsp" adds support for JSP (JavaServer Pages), and "jetty.servlet.jmx" adds JMX support to your Jetty server, allowing you to manage it via VisualVM or JConsole. To install both add-ons:

```
sincerity add jetty.servlet.jsp : add jetty.servlet.jmx : install
```

(You can test JSP support in the example WAR below, at http://localhost:8080/test/jsp/.)

It is also possible to add SPDY support: this protocol, supported by many web browsers, can improve the user experience as well as reduce server load when using "https". For instructions, see the Jetty web server skeleton (page 43).

A nice example of a Jetty server with multiple contexts is also provided, which includes a static web server, a servlet container, and a web application installed as a WAR file:

```
sincerity add jetty.servlet.example : install
```

(You can install this on its own, and it will pull in the basic skeleton as a dependency.)

The example "/server/contexts/servlet-example/" is the most elaborate: it shows you how you can drop in Java source code for your servlets and have them compiled as the server starts.

Additionally, two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add logging : add service : install
```

The following command will install the servlet examples with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add jetty.servlet.example : add logging : add service : instal
```

To stop it:

```
sincerity use mycontainer : service jetty stop
```

Restlet Skeleton

The Restlet library ("Restlet" is a registered trademark of Restlet S.A.S.) lets you dynamically generate web content, but it goes beyond just responding to client requests: it lets you map RESTful resources to URIs, while handling all the tricky HTTP mechanics involved (content negotiation, conditional HTTP) and providing full, rich abstractions for routing, filtering and data presentation.

To install the minimal skeleton:

```
sincerity add restlet : install
```

To start your Restlet component and its servers:

```
sincerity start restlet
```

The default port is 8080, so point your browser to http://localhost:8080.

Restlet, on its own, requires you to code in Java, but Prudence (page 46) builds on Restlet, letting you do all of the above with your choice of JavaScript, Python, Ruby, PHP, Lua, Groovy or Clojure. (Disclosure: Prudence has also been created by Three Crickets.)

Fleshing Out

While Restlet requires you to write your resources in Java, there is no reason for your bootstrapping code—the code that assembles your component, servers, clients, hosts and routes—to be so rigid. The API for boostrapping your component is simple and elegant enough, but without Sincerity you would have to likely have to write it in Java, or implement your own bootstrapping mechanism or use a DSL.

JavaScript, Sincerity's natural language, provides a lightweight solution, and one that does not require you to recompile anything when all you want to change is your configuration. Of course, once your component is up and running, JavaScript plays no more role. We mention that in case you're worried about performance, though you shouldn't be: the language engine is likely not the source of any bottlenecks in your application's live performance.

The skeleton follows the network structure of Restlet, which in turn closely adheres to Roy Fielding's original terminology for Representational State Transfer (REST):

The "/component/" directory is the basis for your REST component.

Under "/component/servers/" you can create files for HTTP servers bound to your component. See the API documentation. The default is a single HTTP server is created on port 8080, but you can create additional servers. The technology used for the servers is called a "connector," and is pluggable in Restlet. Connectors cannot be selected by API calls; rather, they are installed automatically if they are discovered in the classpath. By default, the Sincerity skeleton for Restlet relies on Restlet's internal connector, but it is not recommended for production applications. See "Extras" (page 45) on how to install other connectors.

A quick and easy way to change the port for the default server is to set the environment variable is "RESTLET_PORT" (or the "restlet.port" JVM property):

```
RESTLET_PORT=80 sincerity start restlet
```

Under "/component/hosts/" you can create files for virtual hosts. See the API documentation. The default host has no filters, meaning that *all* requests from *all* servers will be routed to it. If you need several virtual hosts, you will want to make the default host less inclusive, or do away with a default host entirely. (The default host is merely a Restlet convenience and is not required for a component.) Applications can be attached to one or more hosts (see below).

Under "/component/clients/" you can create files for clients supported by your component. See the API documentation. As with servers, client technologies are "connectors" installed on the classpath. Each connector handles a specific URI protocol, such as "http:", "https:" and "file:". The skeleton defines no clients by default, but you can create files here for each client you need. Install the Restlet example (page 45) to see usage of a "file:" client. (The "file:" client is required internally by the Restlet Directory resource.) Note that the Restlet internal connector can handle "http:", but not "https:". To add support for "https:", you can install the Apache HttpClient connector (page 45).

The "/component/services/" is used to configure Restlet services, such as ConnegService, TunnelService, EncoderService, etc., but can be used for any additional work to be done before applications are configured. By default only the LogService is configured.

Finally, "/component/applications/" is where you can create your Restlet applications. See the API documentation. Though you can attach applications directly to your component, it is recommended that you attach them to virtual hosts, even if it's just the default host, as it allows you more routing flexibility. Also, though there is no requirement to do so, most Restlet applications will probably have a Router as their inbound root. It is crucial that you understand how routing works in Restlet: from server, through host, through application, through router, and finally to your RESTful resources. Please refer to the Restlet documentation for full details. Note that the skeleton does not include any application by default, but one is available for you to install (page 45).

It may be useful during development to start only a few select applications. This can be done by providing the application directory names you wish to start as arguments to the "start restlet" command:

```
sincerity start restlet restlet-example myapp
```

Alternatively, you can set the "RESTLET_APPLICATIONS" environment variable (or the "restlet.applications" JVM property) to a comma-separated list of application directory names:

```
RESTLET_APPLICATIONS=restlet-example,myapp sincerity start restlet
```

Extras

A simple example Restlet application, with a custom resource as well as static content:

```
sincerity add restlet.example : install
```

(You can install this on its own, and it will pull in the basic skeleton as a dependency.)

The example at "/component/applications/example/" shows you how you can drop in Java source code for your resources and have it compiled automatically.

The skeleton does not install any connectors by default, relying instead on the default Restlet connectors. To install the Jetty server connector, you have the choice of either Jetty 9.3 (requires JVM 8) or Jetty 9.2 (requires JVM 7). For 9.3:

```
sincerity add restlet.jetty : install
```

For 9.2:

```
sincerity add restlet.jetty.legacy : install
```

It is possible to add HTTP/2 support to Jetty 9.3: this protocol, supported by many web browsers, can improve the user experience as well as reduce server load when using "https". For instructions, see the Jetty web server skeleton (page 43).

To install the Apache HttpClient connector:

```
sincerity add restlet.httpclient : install
```

Other shortcuts include "restlet.simple" (the Simple Framework server connector).

Additionally, two plugins are strongly recommended: logging (page 32) and service (page 34). To install them:

```
sincerity add restlet.logging : add service : install
```

(Note that "restlet.logging" is used here in preference over Sincerity's "logging" plugin. The former depends on the latter, but adds a Restlet library that provides a direct chute to SLF4J, which is more efficient than bridging.)

The following command will install the Restlet example with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add restlet.example : add restlet.jetty : add restlet.logging
```

To stop it:

```
sincerity use mycontainer : service restlet stop
```

Felix Skeleton

Apache Felix is flexible, straightforward OSGi (R4) container.

To install:

```
sincerity add felix : install
```

To start the Gogo console:

```
sincerity felix
```

(You can also use "sincerity gogo" instead.) As an example, let's install the web console via Gogo:

```
install http://archive.apache.org/dist/felix/org.apache.felix.http.jetty-2.2.0.jar
start 5
install http://archive.apache.org/dist/felix/org.apache.felix.webconsole-3.1.8.jar
start 6
```

In this example you may need to change the IDs in the "start" command to match the bundle IDs that Gogo reports. Then, point your browser to http://localhost:8080/system/console/. The default user is "admin" with password "admin".

Prudence Skeleton

Prudence is a platform on which you can build scalable web frontends and network services. It lets you write your server-side code in JavaScript, Python, Ruby, PHP, Lua, Groovy or Clojure. Though minimalistic, Prudence addresses real-world, practical web development needs, from virtual hosting and URI rewriting to state-of-the-art server- and client-side caching. Your applications can support rich clients (AJAX), thin clients (pure HTML), and happy mixes between the two.

Prudence is distributed exclusively as a Sincerity skeleton with a large collection of tightly integrated add-ons. It is an extension of the Restlet skeleton (page 44), so the documentation there applies here. It is, in turn, the underlying platform for Diligence (page 47).

Since version 2.0, Prudence is designed from the ground-up around Sincerity, and such provides the primary example for how Sincerity can reform product distribution.

> Historically, it was actually the other way around. Sincerity was designed by Three Crickets precisely in order to make Prudence 2.0 sanely modular, building on many lessons learned while deploying Prudence 1.0 and 1.1. It was clear during development that there was nothing in the proposed solution that was specific to Prudence. And so Sincerity was born as a generic tool useful for many JVM projects.

Quick start to see the Prudence example:

```
sincerity add prudence.example : install : start prudence
```

And then browse to http://localhost:8080/.

Diligence Skeleton

Diligence lets you develop scalable data-driven web applications in server-side JavaScript, using MongoDB as its data provider and Prudence (page 46) as its RESTful base. It features strong integration with client-side "AJAX," notably Ext JS and Sencha Touch, and clean-room integration with Facebook, Twitter, Google, etc. Services include a scalable email notification system, robust sitemap generation (with special support for *very* large sites), authentication and authorization, and support for several markup languages.

Diligence is distributed exclusively as a Sincerity skeleton. It is an extension of the Prudence skeleton (page 46) and the Restlet skeleton (page 44), so the documentation there applies here.

(Disclosure: Like Sincerity and Prudence, Diligence is developed by Three Crickets. The three products together form a powerful web application stack on top of the JVM.)

Quick start to see the Diligence example:

```
sincerity add diligence.example : install : start prudence
```

And then browse to http://localhost:8080/diligence-example/. Note that the example expects an unprotected MongoDB instance running at localhost.

Rails Skeleton

Ruby on Rails, or just "Rails," is a popular web development framework for the Ruby programming language. It combines a traditional MVC approach with a RESTful orientation backed by relational database stores (MySQL, Postgres). Rails enjoys the elegant, often-imitated, ActiveRecord ORM, and a powerful "scaffolding" feature that automatically generates models, views and controllers to which you can add your code.

Rails is known to work very well on the JVM, but it can sometimes be painful to install everything and get it running. The Sincerity skeleton can do it all for you with one command:

```
sincerity add rails : install
```

This may take a few minutes: Rails is quite massive.

To start you server:

```
sincerity start rails
```

The default port is 3000, so point your browser to http://localhost:3000.

If you're looking for a more strictly RESTful, minimalist alternative to Rails, while sticking to Ruby, take a look at Prudence (page 46). (Disclosure: Prudence has also been created by Three Crickets.)

Fleshing Out

The skeleton will create an application for you under "/app/", so you don't have to run "rails new" to create one. Indeed, the correct way to start a new Rails project in Sincerity is simply to create a new container for it. That's the whole point of Sincerity!

The skeleton comes with a plugin to handle the "rails" tool for you, similarly to how Sincerity's Ruby plugin adds commands for common Ruby tools, such as "gem" and "rake". The benefit of this approach is that you do not have to explicitly change to the "/app/" directory to run the tool, and indeed you can chain it as is usual with Sincerity commands. It should work identically to the usual "rails" command: simple prefix "sincerity" to it. Examples:

```
sincerity rails generate controller home index
sincerity rails generate scaffold Post name:string title:string content:text
sincerity rake db:migrate
```

Or as one Sincerity command:

```
sincerity use mycontainer : rails generate controller home index : rails generate scaffold Po
```

A quick note: Ruby is a bit sluggish to start up on the JVM, which you will notice when running "rails". However, don't let this worry you: once it's up and running, your Rails application will perform marvelously.

And that's it: from here on, it's all standard Rails goodness. You can go ahead with the tutorial, skipping step 3.2 ("Creating the Blog Application").

MySQL, PostgreSQL and SQLite are all supported out of the box, identically to how Rails works on other platforms.

If you need to access the Rails source code, you'll find it under "/libraries/ruby/lib/ruby/gems/1.8/gems/", which is where all Ruby gems will be installed in your container.

Extras

Though the "rails" tool does support a daemon mode, Sincerity's logging plugin (page 32) is far more powerful and is strongly recommended. To install:

```
sincerity add service : install
```

The following command will install the Rails skeleton with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add rails : add service : install : service rails start
```

To stop it:

```
sincerity use mycontainer : service rails stop
```

Note that Sincerity's logging plugin (page 32) won't do you much good out of the box, because Rails uses Ruby's logging system, not the JVM's. However, it should be easy implement your own Ruby logger that delegates to standard JVM logging if that seems exciting to you.

Django Skeleton

Django is a popular web development framework for the Python programming language. It relies on a traditional MVC approach backed by relational database stores (MySQL, Postgres). Django enjoys a large ecosystem of drop-in features and snippets, but already provides many features right out of the box. Much the appeal of Django is the Python programming language: elegant, clean and supported by what must be the friendliest and most welcoming community of any programming language.

There are many advantages for running Django on the JVM instead of on the CPython reference platform: great performance, much improved scalability (there is no GIL in Jython), as well as access to any JVM library *in addition* to Python libraries. Of course, Sincerity makes it extremely easy and transparent to add both kinds of libraries as dependencies.

If you're looking for RESTful, minimalist alternative to Django, while sticking to Python, take a look at Prudence (page 46). (Disclosure: Prudence has also been created by Three Crickets.)

Django can be difficult to install and get running on Jython, but of course it's trivial with Sincerity:

```
sincerity add django : install
```

This may take a few minutes: Django is quite massive!

To start you server:

```
sincerity start django
```

The default port is 8000, so point your browser to http://localhost:8000.

Fleshing Out

The skeleton already has a minimal project ready for you under "/project/", so you don't have to run "django-admin.py startproject" to create one. Indeed, the correct way to start a new Django project in Sincerity is simply to create a new container for it. That's the whole point of Sincerity!

However, if you need to access "django-admin.py", it is located under your "/executables/" directory, so:

```
sincerity execute django-admin.py
```

Much of the work with Django involves running "manage.py", which in this skeleton is located under "/project/manage.py". You can run it easily, from anywhere in the container, with a handy plugin:

```
sincerity manage
```

Note that Python is a bit sluggish to start up on the JVM, which you will notice when running "manage". However, don't let this worry you: once it's up and running, your Django application will perform very well.

And that's it: from here on, it's all standard Django goodness. You can go ahead with the tutorial, skipping the short "Creating a project" step.

Well, just one quick note: the database backend uses JDBC drivers (the JVM's relational database interface) instead of Python drivers, so the database engine names in your "settings.py" are a little bit different than in the official tutorial. You'll see the supported options commented in "settings.py". JDBC drivers for MySQL and

PostgreSQL are included in the skeleton, but you must install the Oracle JDBC driver on your own. Also note that SQLite is not supported at this time.

If you need to access the Django source code, you'll find it under "/libraries/python/Lib/site-packages/", which is where all Python libraries will be installed in your container.

Extras

Adding the service plugin (page 34) is strongly recommended. To install:

```
sincerity add service : install
```

The following command will install the Django skeleton with the recommended plugins into a Sincerity container created in the current directory, and then start it a service:

```
sincerity create mycontainer : add django : add service : install : service django start
```

To stop it:

```
sincerity use mycontainer : service django stop
```

Note that Sincerity's logging plugin (page 32) won't do you much good out of the box, because Django uses Python's logging system, not the JVM's. However, it should be easy implement your own Python logger that delegates to standard JVM logging if that seems exciting to you.

> **OutOfMemoryError?** Installing and starting Django in the same Sincerity command may exhaust your JVM's PermGen space. Try installing and starting via separate commands. For more tips, see the FAQ (page 18).

LWJGL Skeleton

The JVM is growing in popularity as a platform for game designers, due to its ability to easily have the game run on many operating systems, as well as in browsers. Much of this growth is due to the excellent LWJGL library, which makes easy to use hardware-accelerated features, such as 3D graphics and 3D sound, and to accept input from gaming controllers. LWJGL relies on native extensions to the JVM, and supports Linux, Windows, Mac OS X and Solaris. (This author's favorite game, Minecraft, is based on it!)

To install the barebones skeleton:

```
sincerity add lwjgl : install
```

To start your game:

```
sincerity start lwjgl
```

This "lwjgl" program will detect your operating system, install the relevant native binaries into the container (if they aren't already installed), and then start the "game" program... except that with this barebones skeleton, there is no game to start (page 50).

Fleshing Out

Create a "/programs/game.js" that starts up your game. If you want your game to be written only in Java, this likely means delegating to your main class, like this:

```
sincerity.run('delegate:main', ['org.mycoolgame.Main'])
```

However, don't rule out writing your game in JavaScript, or the host of other languages easily installable in Sincerity! You can even "drop down" to Java when you some low-level work, and keep the main game logic in a higher-level language.

This is especially useful if you want to provide a way for the community to provide plugins for your game: it would make it easier for novice programmers to contribute, and also allow such plugins to be distributed as simple text files. If you go this route, consider using Scripturian (page 50) to allow high-performance, muli-threaded integration of the language engines.

Extras

For something to play with, see the "lwjgl.example" add-on. It includes a simple Space Invaders clone:

```
sincerity add lwjgl.example : install : start lwjgl
```

Libraries

The Sincerity JsDoc Template

TODO

```
sincerity add jsdoc.sincerity : install
```

Note that you need to host the documentation via HTTP. File does not work.
See JsDoc plugin (page 37).

MongoDB JavaScript Driver

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

Part III
Advanced Manual

Programming

Scripturian

TODO
See link.

The Sincerity JavaScript Library

Sincerity relies on JavaScript for bootstrapping and plugins, and while JavaScript does not have a standard library, you do have access to the entire JVM standard library.

Still, this isn't quite good enough: using JVM libraries works, but they do involve using paradigms that have not been optimized for JavaScript.

For Sincerity, we decided that we can do better, and so we present you with a collection of useful code called the Sincerity JavaScript Library. We should point out from the start that this is not a general-purpose JavaScript library: it relies on the JVM libraries, and only works in the JVM. Included are also optimizations specific to the Nashorn and Rhino JVM JavaScript engines.

What follows is a general introduction to the library. See the API documentation for full details. Also make sure to check out Sincerity's JsDoc plugin which makes it easy for you to generate similar documentation for your own JavaScript codebase.

Note that the Sincerity Foundation Library is used by at least two other JavaScript frameworks: the Prudence JavaScript Library, and the Diligence Framework, which builds on the Prudence JavaScript Library.

Objects Enhanced support for standard JavaScript types: strings, arrays, dicts and dates. This library monkey-patches the standard types with many useful methods. See the Sincerity.Objects API documentation.

Classes This straightforward-but-powerful library lets you use the object-oriented programming (OOP) paradigm in JavaScript. It lets you define classes with public and private members, inherit classes, and even provides a mechanism for generation of constructors. Generally, the Sincerity JavaScript Library does not use OOP indiscriminately: classes are used only when they make sense and add elegance. See the Sincerity.Classes API documentation.

Iterators Iterators let you write coherent code that can efficiently comprehend and operate on sequences of any size. Its design borrows stylistically from functional programming languages. See the Sincerity.Iterators API documentation.

Files Low-level access to the filesystem, including high-performance reading and writing of files using memory-mapped files. See the Sincerity.Files API documentation.

Templates Straightforward and flexible string interpolation. See the Sincerity.Templates API documentation. Example:

```
println('Hello, {user}'.cast({user: 'Sincerity'}))
```

JSON High-performance JSON parsing and rendering using the JSON JVM library, which is written in Java. See the Sincerity.JSON API documentation.

XML High-performance XML parsing and rendering using the standard JVM libraries. See the Sincerity.Objects API documentationSincerity.XML API documentation.

Calendar Enhancements to JavaScript's standard Date type. See the Sincerity.Calendar API documentation.

Localization Easy access to the JVM's localization libraries, including formatting for dates, times and currencies. See the Sincerity.Localization API documentation.

Cryptography Easy access to the JVM's cryptography libraries, including shortcuts for common hashing, encryption and decryption tasks. Sincerity.Cryptography API documentation

JVM Easy conversions between JVM and JavaScript types, and also access to a few operating system services. Sincerity.Localization API documentationSincerity.JVM API documentation

Validation A general-purpose user input validation library for commonly used types, such as numbers and email addresses. Sincerity.Validation API documentation

Mail Easy access to JavaMail, including sending of mixed-media plain-text/HTML emails. Uses the templates library to let you easily create email templates. Sincerity.Mail API documentation

Lucene Easy access to the Lucene search engine. Supports the iterators library, so you can easily index very large collections of documents. Sincerity.Lucene API documentation

Platform Access to features of the Nashorn and Rhino JavaScript engines, such as the call stack and exception details. Sincerity.Platform API documentation

Extending Sincerity

Developing Plugins

TODO

Make sure you understand that dependencies may be installed in arbitrary order.

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

try {
        document.require('/mongo-db/')
} catch(x) { /* the dependency may not have been installed yet! */ }
```

Eclipse Integration

TODO

Installing

http://repository.threecrickets.com/eclipse/

Preferences

Using internal or external Sincerity installation.

Sincerity Projects

Converting to Sincerity

Adds the Sincerity nature.

Sincerity Classpath

Java projects only.

Sincerity Launch Configurations

Choose the program or URI.

Debugging

Breakpoints in Java Code

Breakpoints in non-Java Code

Packaging

There are two main reasons you would want to create Sincerity packages:

1. You've created a useful skeleton, skeleton add-on or plugin, which you would like to share with others for use in their Sincerity containers. A package, of course, is the most natural way to do so. You could then host your package on your own repository, or submit it for inclusion in other public repositories.

2. Packages are very useful for deploying your application internally, especially in ephemeral "cloud" environments. Programmers working on different modules could package their results, using a clear versioning system. You would then host the packages in your own private repository, using Nexus or even a plain directory. Deployment, including upgrades, would thus involve nothing more than running "sincerity install" on the relevant containers. It also allows easy downgrading of applications, or setting modules to specific versions for testing and debugging.

   Note that "packaging" here refers specifically to creating Sincerity packages, which you can then install into Sincerity containers as dependencies. If what you want is to distribute the entire container, then see the Distribution Plugin, and also the Sincerity Runtime Plugin.

The Sincerity Packaging Plugin

... does not exist yet, as of Sincerity 1.0. This is something on our roadmap, and technically entirely viable. The idea is to allow for a friendly GUI, as well as a strong CLI.

Until then, you can use Maven, as detailed below. It's slightly awkward, in that it requires editing complex XML files, but for the purpose of creating simple packages it should be very straightforward.

How to Create a Sincerity Package Using Maven

It's relatively easy to use Apache Maven to create a Sincerity package, with the help of the maven-assembly-plugin.

You can start with the following "pom.xml" file as a skeleton:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
        xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v
        <modelVersion>4.0.0</modelVersion>

        <groupId>org.myorg.myapp</groupId>
        <artifactId>myapp</artifactId>
        <version>1.0.0</version>
        <packaging>pom</packaging>

        <name>My Cool Application</name>
        <description>This is an application packaged for use with Sincerity.</description>

        <dependencies>
                <dependency>
                        <groupId>com.threecrickets.savory</groupId>
                        <artifactId>savory-framework</artifactId>
                        <version>1.0-beta1</version>
                </dependency>
        </dependencies>

        <build>
                <directory>cache</directory>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-assembly-plugin</artifactId>
                                <version>2.2.1</version>
                                <executions>
                                        <execution>
                                                <id>jar</id>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>single</goal>
                                                </goals>
                                                <configuration>
                                                        <appendAssemblyId>false</appendAssemb
                                                        <archive>
                                                                <manifestEntries>
                                                                        <Package-Folders>pack
                                                                </manifestEntries>
                                                        </archive>
                                                        <descriptors>
                                                                <descriptor>package.xml</desc
                                                        </descriptors>
                                                </configuration>
                                        </execution>
                                </executions>
                        </plugin>
                </plugins>
        </build>
```

```
</project>
```

Some things you'll want to customize:

- You can add as many dependencies as you like. Note that they can plain JVM jars, Sincerity packages, Python packages, Ruby gems, etc.: anything supported by Sincerity. In this case, we are including the Savory Framework, which is a Sincerity package (which in turn has dependencies). You can also have no dependencies at all.
- Under <manifestEntries> you can add anything that adheres to the packaging specification (page 55). For example, you may want to call a package installation script, like so:

  ```
  <Package-Installer>com.threecrickets.sincerity.Sincerity delegate:start /libraries/script
  ```

  (If you do so, you'll need a "libraries/scripturian/installers/myapp.js" file in your package, otherwise Sincerity will report an error when trying to install it.)
- The <directory> is a work directory used by Maven for creating your final package. You may want to specify it as "/tmp". It is relative to the location of the "pom.xml" file.
- If you want to share your package in a public repository, you'd likely want to add additional information about your package. Consult the Maven pom.xml guide for more options.

You will also need to create a "package.xml" file in the same directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<assembly
        xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.

        <id>jar</id>
        <formats>
                <format>jar</format>
        </formats>
        <baseDirectory>package</baseDirectory>
        <fileSets>
                <fileSet>
                        <directory>path-to-package</directory>
                        <outputDirectory>.</outputDirectory>
                        <includes>
                                <include></include>
                        </includes>
                </fileSet>
        </fileSets>
</assembly>
```

You'll want to change "path-to-package" to point to the base of your distribution directory. Note that it is relative to the location of "package.xml" file, and must have the final directory structure you want in the Sincerity container into which your package will be installed.

Note that you can create much more complex <fileSets> than this one. Consult the assembly descriptor format documentation for more information.

You can now build and deploy your package into a local file repository by running the following Maven command from the directory in which you have your "pom.xml" file:

```
mvn deploy -DaltDeploymentRepository=myrepo::default::file:/path-to-local-repository/
```

Note that if this is the first time you've run Maven, it will take some time to download all the necessary plugins it needs. Consequent runs will be much faster.

Of course, you can also deploy to a repository server, such as Nexus, which you can easily install with Sincerity's Nexus skeleton. You can also configure Maven to always use a default target repository for deployment.

Maven is a complex tool that can do a whole lot more than this, but this should get you started.

[TODO: Add note about support for -SNAPSHOT]

Repositories

TODO

Sincerity can work with a any arbitrary repository for which it has the supported technology. That said, here's an overview of some repositories that you are most likely to work with:

The Three Crickets Repository

See link.

iBiblio/Maven Repositories

Python and PyPI (a.k.a. "The Cheese Factory")

Ruby and Gems

PHP and PEAR

Community Repositories

Specifications

Sincerity Packages

Packages are collections of artifacts. They are defined using special tags in standard JVM resource manifests. Additionally, packages support special install/uninstall hooks for calling arbitrary entry points, allowing for custom behavior. Indeed, a package can include no artifacts, and only implement these hooks.

Packages allow you to work around various limitations in repositories such as iBiblio/Maven, in which the smallest deployable unit is a Jar. The package specification allows you to include as many files as you need in a single Jar, greatly simplifying your deployment scheme.

Note that two different ways are supported for specifying artifacts: they can specified as files, thus referring to actual zipped entries with the Jar file in which the manifest resides, or that can be specified as general resources, in which case they will be general resource URLs to be loaded by the classloader, and thus they can reside anywhere in the classpath.

Also note what "volatile" means in this context: a "volatile" artifact is one that should be installed once and only once. This means that subsequent attempts to install the package, beyond the first, should ignore these artifacts. This is useful for marking configuration files, example files, and other files that the user should be allow to delete without worrying that they would reappear on every change to the dependency structure.

The Manifest

Supported manifest tags:

- **Package-Files**: a comma separated list of file paths within this Jar.
- **Package-Folders:** a comma separated list of folder paths within this Jar. Specifies all artifacts under these folders, recursively.
- **Package-Resources**: a comma separated list of resource paths to be retrieved via the classloader.
- **Package-Volatile-Files**: all these artifacts will be marked as volatile.
- **Package-Volatile-Folders**: all artifacts under these paths will be marked as volatile.
- **Package-Installer**: specifies a class name which has a main() entry point. Simple string arguments can be optionally appended, separated by spaces. The installer will be called when the package is to be installed, *after* all artifacts have been unpacked. Any thrown exception would cause installation to fail.
- **Package-Uninstaller**: specifies a class name which has a main() entry point. Simple string arguments can be optionally appended, separated by spaces. The uninstaller will be called when the package is to be uninstalled.

For example, here is a "/META-INF/MANIFEST.MF" file:

```
Manifest-Version: 1.0
Package-Folders: package
```

All packaged files would be expected under the "/package/" directory inside the Jar.
Note that manifests can often be automatically created by packaging tools. See the Maven example (page 53).