The Prudence Manual

Version 1.1
Main text written by Tal Liron

March 26, 2011

Contents

The Case for REST

There's a lot of buzz about REST, but also a lot confusion about what it is and what it's good for. This essay attempts to convey REST's simple essence.

Let's start, then, not at REST, but at an attempt to create a new architecture for building scalable applications. Our goals are for it to be minimal, straightforward, and still have enough features to be productive. We want to learn some lessons from the failures of other, more elaborate and complete architectures.

Let's call ours a "resource-oriented architecture."

Resources

Our base unit is a "resource," which, like an object in object-oriented architectures, encapsulates data with some functionality. However, we've learned from object-orientation that implementing arbitrary interfaces is a recipe for complexity: proxy generation, support for arbitrary types, marshalling, etc. Instead, then, we'll keep it simple and define a limited, unified interface that would be just useful enough.

From our experience with relational databases, we've learned that a tremendous amount of power can be found in "CRUD": Create, Read, Update and Delete. If we support just these operations, our resources will already be very powerful, enjoying the accumulated wisdom and design patterns from the database world.

Identifiers

First, let's start with a way of identifying our resources. We'll define a name-based address space where our resources live. Each resource is "attached" to one or more addresses. We'll allow for "/" as a customary separator to allow for hierarchical addressing schemes. For example:

```
/animal/dog/3/
/animal/cat/12/image/
/animal/cat/12/image/large/
/animal/cat/12/specs/
```

In the above, we've allowed for different kinds of animals, a way of referencing individual animals, and a way of referencing specific aspects of these animals.

Let's now go over CRUD operations in increasing order of complexity.

Delete

"Delete" is the most trivial operation. After sending "delete" to an identifier, we expect it to not exist anymore. Whether sub-resources in our hierarchy can exist or not, we'll leave up to individual implementations. For example, deleting "/animal/cat/12/image" may or may not delete "/animal/cat/12/image/large".

Note that we don't care about atomicity here, because we don't expect anything to happen after our "delete" operation. A million changes can happen to our cat before our command is processed, but they're all forgotten after "delete." (See "update," below, for a small caveat.)

Read

"Read" is a bit more complicated than "delete." Since our resource might be changed by other clients, we want to make sure that there's some kind of way to mark which version we are reading. This will allow us to avoid unnecessary reads if there hasn't been any change.

Thus, we'll need our resource-oriented architecture to support some kind of version tagging feature.

Update

The problem with "update" is that it always references a certain version that we have "read" before. In some cases, though not all, we need some way to make sure that the data we expect to be there hasn't changed since we've last "read" it. Let's call this a "conditional update."

Actually, we've oversimplified our earlier definition of "delete." In some cases, we'd want a "conditional delete" to depend on certain expectations about the data. We might not want the resource deleted in some cases.

So, we'll need our resource-oriented architecture to support a "conditional" operation feature.

Create

This is our most complex operation. Our first problem is that our identifier might not exist yet, or might already be attached to a resource. One approach could be to try identifiers in sequence:

```
Create: /animal/cat/13/ -> Error, already exists
Create: /animal/cat/14/ -> Error, already exists
Create: /animal/cat/15/ -> Error, already exists
...
Create: /animal/cat/302041/ -> Success!
```

Obviously, this is not a scalable solution. Another approach could be to have a helper resource which provides us with the necessary ID:

```
Read: /animal/cat/next/ -> 14
Create: /animal/cat/14/ -> Oops, someone else beat us to 14!
Read: /animal/cat/next/ -> 15
Create: /animal/cat/15/ -> Success!
```

Of course, we can also have "/animal/cat/next" return unique IDs (such as GUIDs) to avoid duplications. If we never create our cat, they will be wasted, though. The main problem with this approach is that it requires two calls per creation: a "read," and then a "create." We can handle this in one call by allowing for "partial" creation, a "create" linked with an intrinsic "read":

```
Create: /animal/cat/ -> We send the data for the cat without the ID, and get back
    the same cat with an ID
```

Other solutions exist, too. The point of this discussion is to show you that "create" is not trivial, but also that solutions to "create" already exist within the resource-oriented architecture we've defined. "Create," though programmatically complex, does not require any new architectural features.

Aggregate Resources

At first glance, handling the problem of getting lots of resources at the same time, thus saving on the number of calls, can trivially be handled by the features we've listed so far. A common solution is to define a "plural" version of the "singular" resource:

```
/animal/cats/
```

A "read" would give us all cats. But what if there are ten million cats? We can support paging. Again, we have a solution within our current feature set, using identifiers for each subset of cats:

```
/animal/cats/100/200/
```

We can define the above to return no more than 100 cats: from the 100th, to the 200th. There's a slight problem in this solution: the burden is on whatever component in our system handles mapping identifiers to resources. This is not terrible, but if we want our system to be more generic, it could help if things like "100 to 200" could be handled by our resource more directly. For convenience, let's implement a simple parameter system for all commands:

```
Read(100, 200): /animal/cats/
```

In the above, our mapping component only needs to know about "/animal/cats". The dumber our mapping component is, the easier it is to implement.

Formats

The problem of supporting multiple formats seems similar, at first glance, to that of aggregate resources. Again, we could potentially solve it with command parameters:

```
Read(UTF-8, Russian): /animal/cat/13/
```

This would give us a Russian, Unicode UTF-8 encoded version of our cat. Looks good, except that there is a potential problem: the client might prefer certain formats, but actually be able to handle others. It's more a matter of preference than any precision. Of course, we can have another resource where all available formats are listed, but this would require an extra call, and also introduce the problem of atomicity—what if the cat changes between these calls? A better solution would be to have the client associate certain preferences per command, have our resource emit its capabilities, with the mapping component in between "negotiating" these two lists. This "negotiation" is a rather simple algorithm to choose the best mutually preferable format.

This would be a simple feature to add to our resource-oriented architecture, which could greatly help to decouple its support for multiple formats from its addressing scheme.

Shared State

Shared state between the client and server is very useful for managing sessions and implementing basic security. Of course, it's quite easy to abuse shared state, too, by treating it as a cache for data. We don't want to encourage that. Instead, we just want a very simple shared state system.

We'll allow for this by attaching small, named, shared state objects to every request and response to a command. Nothing fancy or elaborate. There is a potential security breach here, so we have to trust that all components along the way honor the relationship between client and server, and don't allow other servers access to our shared state.

Summary of Features

So, what do we need?

We need a way to map identifiers to resources. We need support for the four CRUD operations. We need support for "conditional" updates and deletes. We need all operations to support "parameters." We need "negotiation" of formats. And, we need a simple shared state attachment feature.

This list is very easy to implement. It requires very little computing power, and no support for generic, arbitrary additions.

Transactions... Not!

Before we go on, it's worth mentioning one important feature which we did not require: transactions. Transactions are optional, and sometimes core features in many databases and distributed object systems. They can be extremely powerful, as they allow atomicity across an arbitrary number of commands. They are also, however, heavy to implement, as they require considerable shared state between client and server. Powerful as they are, it is possible to live without them. It's possible, for example, to implement this atomicity within a single resource. This would require us to define special resources per type of transaction which we want to support, but it does remove the heavy burden of supporting arbitrary transactions from our architecture. With some small reluctance, then, we'll do without transactions.

Let's Do It!

OK, so now we know what we need, let's go ahead and implement the infrastructure of components to handle our requirements. All we need is stacks for all supported clients, backend stacks for all our potential server platforms, middleware components to handle all the identifier routing, content negotiation, caching of data...

...And thousands of man hours to develop, test, deploy, and integrate. Like any large-scale, enterprise architecture, even trivial requirements have to jump through the usual hoops set up by the sheer scale of the task. Behind every great architecture are the nuts and bolts of the infrastructure.

Wouldn't it be great if the infrastructure already existed?

The Punchline

Well, duh. All the requirements for our resource-oriented architecture are already supported by HTTP:

Our resource identifiers are simple URLs. The CRUD operations are in the four HTTP verbs: PUT, GET, POST and DELETE. "Conditional" and "negotiated" modes are handled by headers, as are "cookies" for shared state. Version stamps are e-tags. Command parameters are query matrixes appended to URLs. It's all there.

Most importantly, the infrastructure for HTTP is already fully deployed world-wide. TCP/IP stacks are part of practically every operating system; wiring, switching and routing are part and parcel; HTTP gateways, firewalls, load balancers, proxies, caches, filters, etc., are stable consumer components; certificate authorities, national laws, international agreements are already in place to support the complex inter-business interaction. Best of all, this available infrastructure is successfully maintained, with minimal down-time, by highly-skilled independent technicians, organizations and component vendors across the world.

> It's important to note a dependency and possible limitation of HTTP: it is bound to TCP/IP. Indeed, all identifiers are URLs: Uniform Resource Locators. In URLs, the first segment is reserved for the domain, either an IP address or a domain name translatable to an IP address. Compare this with the more general URIs (Uniform Resource Identifiers), which do not have this requirement. Though we'll often be tied to HTTP in REST, you'll see the literature attempting, at least, to be more generic. There are definitely use cases for non-HTTP, and even non-TCP/IP addressing schemes. In Prudence, you'll see that it's possible to address internal resources with URIs *that are not URLs*; see document.internal (page 72).

It's All About Infrastructure

The most important lesson to take is the importance of infrastructure, something easily forgotten when planning architecture in ideal, abstract terms. This is why, I believe, Roy Fielding named Chapter 5 of his 2000 dissertation "Representational State Transfer (REST)" rather than, say, "resource-oriented architecture," as we have here. Fielding, one of the authors of the HTTP protocol, was intimately familiar with its challenges, and the name "REST" is intended to point out the key characteristic of its infrastructure: HTTP and similar protocols are designed for transfering lightly annotated data representations. "Resources" are merely logical encapsulations of these representations, depending on a contract between client and server. The infrastructure does not, in itself, do anything in particular to maintain, say, a sensible hierarchy of addresses, the atomicity of CRUD operations, etc. That's up to your implementation. But, representational state transfer—REST—is the mundane, underlying magic that makes it all possible.

To come back to where we started: a resource-oriented architecture requires a REST infrastructure. Practically, the two terms become interchangeable.

The principles of resource-orientation can and are applied in many systems. The word wide web, of course, with its ecology of web browsers, web servers, certificate authorities, etc., is the most obvious model. But other core Internet systems, such as email (SMTP, POP, IMAP), file transfer (FTP, WebDAV) also implement some subset of REST. Your application can do this, too, and enjoy the same potential for scalability as these global, open implementations.

Does REST Scale?

Part of the buzz about REST is that it's an inherently scalable architecture. This is true, but perhaps not in the way that you think.

Consider that there are two uses of the word "scalable":

First, it's **the ability to respond to a growing number of user requests without degradation in response time**, by "simply" adding hardware (horizontal scaling) or replacing it with more powerful hardware (vertical scaling). This is the aspect of scalability that engineers care about. The simple answer is that REST can help, but it doesn't stand out. SOAP, for example, can also do it pretty well. REST aficionados sometimes point out that REST is "stateless," or "session-less," both characteristics that would definitely help scale. But, this is misleading. Protocols might be stateless, but architectures built on top of them don't have to be. For example, we've specifically talked about sessions here, and many web frameworks manage sessions via cookies. Likewise, you can easily make poorly scalable REST. The bottom line is that there's nothing in REST that guarantees scalability in *this* respect. Indeed, engineers coming to REST due to this false lure end up wondering what the big deal is.

The second use of "scalability" comes from the realm of enterprise and project management. It's **the ability of your project to grow in complexity without degradation in your ability to manage it**. And that's REST's beauty—you already have the infrastructure, which is the hardest thing to scale in a project. You don't need to deploy client stacks. You don't need to create and update proxy objects for five different programming languages used in your enterprise. You don't need to deploy incompatible middleware by three different vendors and spend weeks trying to force them to play well together. Why would engineers

care about REST? Precisely because they don't have to: they can focus on application engineering, rather than get bogged down by infrastructure management.

That said, a "resource-oriented architecture" as we defined here is not a bad start for (engineering-wise) scalable systems. Keep your extras lightweight, minimize or eliminate shared state, and encapsulate your resources according to use cases, and you won't, at least, create any immediate obstacles to scaling.

Prudence

Convinced? The best way to understand REST is to experiment with it. You've come to the right place. Start with the tutorial (page 10), and feel free to skip around the documentation and try things out for yourself. You'll find it easy, fun, and powerful enough for you to create large-scale applications that take full advantage of the inherently scalable infrastructure of REST. Happy RESTing!

How to Choose a Flavor?

Prudence was designed around a strong belief in the power of the JVM coupled with the acknowledgment that the Java language does not suit every development project. Thus, Prudence is available in six mouthwatering flavors. Pick the one that works best *for you*!

Python (Succulent!)

Python is a powerful object-oriented language with many high-quality core and 3rd party libraries. It has already proven itself as a capable web programming language, and has many excellent web platforms that make good use of its strengths.

Python presents a unique challenge in a scriptlet environment, due to its reliance on indentation (page 32). However, because HTML is loose with whitespace, it's possible to force the whole file to adhere to Python's scheme. In fact, as many Python enthusiasts would argue, forcing your code to adhere to Python's indentation requirements can go a long way towards making it more readable and manageable.

In Stickstick, the included example application, we show how to use SQLAlchemy as a data backend for Prudence.

> Note: Prudence for Python was built primarily around Jython, but also offers limited support for Jepp if it's installed. For those cases where you need access to a natively-built Python library that won't work on Jython, Jepp lets you run code on the CPython platform.

Ruby (Delectable!)

Ruby can do most of what Python can do and more. A true chameleon, it can adapt to many styles of code and programming. If something can be possible, Ruby allows it and supports it. Unlike Python, it has a very loose and forgiving syntax, which is perfect for scriptlets.

Ruby's Rails platform has revolutionized web programming by offering elegant, powerful alternatives to working directly with HTTP. We hope Ruby web programmers will find in Prudence a refreshing alternative to Rails: elegantly embracing HTTP, instead of avoiding it.

Clojure (Scrumptious!)

Prudence's only functional flavor is a Lisp designed from the ground up for high concurrency. If you're serious about multithreading, Clojure is the way to go. Though new, Clojure is based on one of the oldest programming languages around, and enjoys a long, rich tradition of elegant solutions for tough programming challenges.

Clojure embraces the JVM, but also has a growing collection of nifty "contrib" libraries—all included in Prudence—that make the JVM core more lispy. In the included example application, we show how to use Clojure's SQL library to elegantly access a data backend.

JavaScript (Savory!)

JavaScript (a dialect of ECMAScript) is a sensible choice for "AJAX" and other rich web client applications, because it's the same language used by web browsers and other client platforms. Web developers are already proficient in JavaScript, and can quickly be brought on board to a server-side project. Couple it with JSON and JSON-based document databases, and you're on solid, consistent ground for rapid development. Of all the web programming languages, it's the one most widely deployed and with the most secure future.

It's also an under-appreciated language, and the target of undeserved scorn. Though not as feature-rich as Python or Ruby, JavaScript is still very powerful. Its straightforward closure/prototype mechanisms allow it to support object-orientation, namespaces and other paradigms. Unfortunately, client environments have proven painfully fickle: working with the browser DOM and testing with cross-browser HTML rendering quirks are painful—but these are the not the fault of the language itself. They are also not relevant to server-side development with Prudence. If you've been burned by JavaScript on the client, try it on the server via Prudence: you might be pleasantly surprised.

JavaScript does not have its own core libraries, making it the most minimal Prudence flavor. Instead, it relies on the excellent JVM core.

PHP (Ambrosial!)

PHP is ubiquitous. It's a simple language with the most mature libraries of any web programming language, and programmers are available with years of experience. It's also designed from the ground up as a programming language for the web.

Prudence allows a smooth transition from traditional PHP HTML generation to REST resources. It supports PHP "superglobals" such as $_GET, $_POST, $_COOKIE and $_FILE (but not $_SESSION) to make you feel right at home. It also adds many new features to conventional HTML generation: fine-grained caching, high-performance templating languages, and more.

> Note: Prudence PHP was built around the open source edition of Quercus, which does not feature JVM bytecode compilation as is available in the non-free professional edition. Nevertheless, we found the "non-pro" Quercus to be an excellent performer!

Groovy (Luscious!)

In some ways, Groovy is the best of this bunch. It has all the flexibility of Ruby, but is designed from the ground up to enhance and extend Java. Java programmers would immediately feel at home, while gaining access to far less restrictive programming paradigms. Groovy makes Java... groovy!

All the other Prudence flavors offer JVM interaction, but Groovy does it best. If you know your project will require a lot of work with Java libraries, Groovy is a terrific—and fun!—choice.

Tutorial

The goal of this tutorial is to get you up and running with Prudence's most basic features. Along the way, we'll point you to those sections in the manual where you can explore some topics in depth.

We recommend going through this tutorial using the Firefox browser and the Firebug add-on, which will let you see in detail exactly how Prudence responds to client requests.

First Things First

Get It

Prudence is available in six flavors, so choose one and download it!

A friendly page (page 9) is available to help you decide.

For this tutorial, we'll use the savory JavaScript flavor, under the assumption that JavaScript would be the most widely known. But, let's be clear: the JavaScript in Prudence runs on the *server*, not in the *browser*. Wrap your head around that for a minute... JavaScript is here used *not* for live manipulation of HTML elements, but to do the usual server-side work: communicating with data stores, rendering HTML pages to

send to clients, etc. Of course, in your project, you might be writing JavaScript code to run on the *client*, too. We'll even show you an example of it in this tutorial.

It should be very easy to use any other flavor with this tutorial, since we'll be keeping actual programming to a minimum. Just replace references to "JavaScript" with your flavor of choice.

JVM

The only requirement for Prudence is a Java Virtual Machine (JVM), at least version 5, so make sure you have one installed. An excellent, open-source JVM is available from the OpenJDK project.

> A Java Runtime Environment (JRE) is enough for Prudence. You need a Java Development Kit (JDK) only if you plan to write code in Java. Also, Prudence does not require anything from Java Enterprise Edition (JEE). In fact, you can see Prudence as a RESTful, minimal alternative to developing web applications under JEE.

Fire It Up

Start Prudence with the appropriate "run" script in the /bin/ directory. Use "/bin/run.sh" for Unix-like systems (Linux, *BSD and OS X), and "/bin/run.bat" for Windows. Use "console" as an argument to the script.

Prudence should declare its version and list the installed demo applications. When it announces that it is listening on port 8080, it's ready to go. You can quit Prudence at any time by breaking (CTRL-C).

Open your web browser to http://localhost:8080/. You should see the Prudence Administration application, which will link you to the demos. Also, check out the /logs/ directory for running, ongoing logs.

Further Exploration

- The "console" argument works well enough, but for production environments you're better off running Prudence as a daemon (page 87).
- Learn how to configure Prudence (page 20).
- Prudence logging is flexible and useful (page 84).
- Learn more about the Prudence Administration application (page 86).

Hello, World

The /applications/ directory is where you install your applications. You'll find the demo applications there, but let's create a new one, called "Wacky Wiki". Just create an "/applications/wackywiki/" directory and restart Prudence. The Prudence Administration application should show your application listed under the http://localhost:8080/wackywiki/ URL, but you won't see anything when you click on it quite yet.

Now, let's create our first web page. Create "/applications/wackywiki/web/", and then "/applications/wackywiki/web/static/". By "static" here is meant that the contents do not change per user request. Your /web/static/ directory functions pretty much like a regular web server, from which you can serve images, HTML, CSS files, etc. Let's create a "/web/static/index.html" file:

```
<html>
<body>
        <p>Hello, world! Welcome to Wacky Wiki.</p>
</body>
</html>
```

You should be able to see this page under http://localhost:8080/wackywiki/. If you're using Firebug, you can see that the contents are compressed (using deflate, zip or gzip) and that conditional HTTP is also handled automatically, such that subsequent page visits return the 304 "the document has not been modified" status code. When receiving a 304, the browser will use its cached version instead of downloading it from Prudence.

Further Exploration

- Learn all about configuring and deploying applications (page 25).
- Learn how to configure /web/static/ (page 51).

Hello, Dynamic World

> In this section, we're going to teach you how to dynamically generate web pages. Prudence offers some terrific features for this, such as sophisticated caching, but if you're more interested in how Prudence lets you to easily create RESTful resources, just skip ahead (page 16) and come back here later.

Create a /web/dynamic/ directory and move your "index.html" file to there. Unlike /web/static/, you can put programming language code in these files. Edit "index.html", and add some code:

```
<html>
<body>
        <p>Hello, world! Welcome to Wacky Wiki.</p>
<%
var entry = {title: 'I Love Prudence', contents: 'This is a love letter to Prudence
    '}
%>
        <h2><%= entry.title %></h2>
        <p><%= entry.contents %></p>
</body>
</html>
```

The "<% ... %>" delimiters are used to insert "scriptlets" of pure JavaScript code. The "<%= ... %>" delimiters accept a JavaScript expression, and print it out onto the HTML page.

Refresh your browser, and you should see the dynamically generated page. Hello!

> Make sure you understand the significance of newlines and whitespace in scriptlets (page 32), especially if you're using Python!

Caching

Look at how Firefox communicates with your dynamic page using Firebug: the contents are compressed, but there is no conditional HTTP. That's because Prudence generates a new version of the page per every user request. Since each such version might be different from the last, the default behavior is that clients should never expect their cached version to be correct.

We can change this by adding caching to our page, specifically for 10 seconds (10,000 milliseconds):

```
<html>
<body>
        <p>Hello, world! Welcome to Wacky Wiki.</p>
<%
document.cacheDuration = 10000
var entry = {title: 'I Love Prudence', contents: 'This is a love letter to Prudence
    .'}
%>
        <h2><%= entry.title %></h2>
        <p><%= entry.contents %></p>
</body>
</html>
```

Keep refreshing the page, and you'll see that for every 10 seconds you are getting 304 status codes.

Actually, something far more important is happening: any browser from any computer would see the same version of the page for those 10 seconds. Prudence caches and serves a static version of the page during that interval *and does not run any of your JavaScript code*. We've effectively throttled execution of our code.

Our code didn't do much in this example, but what if it were fetching data for that wiki page from a database? Caching would allow us to serve a very large number of user requests without ever touching the database. We could very precisely state that we want the database accessed only, at the most, every 10 seconds. That's excellent scalability, easily accomplished.

Debugging

For fun, let's purposely introduce an error into our JavaScript code: just type some nonsense into the scriptlet above. When you refresh the page, you'll see a generic error.

Now, tet's enable Prudence's debug page. Create a file named "/applications/wackywiki/settings.js":

```
document.execute('/defaults/application/settings/')
showDebugOnError = true
```

Restart Prudence, and refresh the page. You'll see a lot of useful information to help you understand what went wrong, including a link directly to your source code. Of course, you wouldn't want to enable this debug page for production systems. Moreover, you might want to show a custom error message instead: Prudence can do that.

Further Exploration

- Learn more about scriptlets (page 31).
- We've barely scratched the surface of Prudence's sophisticated caching system (page 37).
- Learn how to accept data from the user via HTML forms (page 39).
- Learn about other application settings (page 27).
- Learn more about the debug page and other debugging features (page 83).
- Learn how to show custom error pages (page 58).

Hello, Fragmented World

As your application grows, you'll likely have a lot of reusable code. Prudence can help you manage it.

Let's create a /web/fragments/ directory, and put the following files in it. Here's "header.html":

```
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
        <h1>Wacky Wiki</h1>
```

Here's "footer.html":

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

And here's "entry.html":

```
<h2><%= entry.title %></h2>
<p><%= entry.contents %></p>
```

We can now edit our original "/web/dynamic/index.html" using special "<%& ... %>" include scriptlets:

```
<%& '/header/' %>
<p>Hello, world! Welcome to Wacky Wiki.</p>
<%
document.cacheDuration = 10000
var entry = {
        title: 'I Love Prudence',
```

```
        contents: 'This is a love letter to Prudence (fragmented).'}
%>
<%& '/entry/' %>
<%& '/footer/' %>
```

We can use the same fragments in other pages, too. Let's create "/web/dynamic/home.html":

```
<%& '/header/' %>
<p>This is the home wiki.</p>
<%
document.cacheDuration = 10000
var entry = {
        title: 'Wiki Home',
        contents: 'This wiki page is empty.'}
%>
<%& '/entry/' %>
<%& '/footer/' %>
```

Note that this page is available under the URL http://localhost:8080/wackywiki/home/. Prudence by default ignores filename extensions for URLs and requires trailing slashes, leading to prettier, more coherent URLs for users.

Caching Fragments

Fragments help organize your code, but they also offer another advantage: because Prudence caches each fragment individually, you can have certain parts of your pages cached longer than others. Moreover, you can alter the cache key to make sure that you are not caching different versions of the fragments per URL. This is a more advanced topic, but we'll provide a quick preview of it example below for our "header" and "footer" fragments. The cache key pattern we will use ensures that the fragment is cached just once per application name ("{an}") and document name ("{dn}"). This is appropriate for our case, because "index.html" and "home.html" never change their content for whatever Wacky Wiki page they're in. If they did change, we could leave the cache pattern at its default, or otherwise tweak it. Maximum efficiency here for maximum scalability.

So, here's the new "header.html":

```
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
<%
document.cacheDuration = 300000
document.cacheKeyPattern = '{an}|{dn}'
%>
```

And "footer.html":

```
</body>
</html>
<%
document.cacheDuration = 300000
document.cacheKeyPattern = '{an}|{dn}'
%>
```

You won't see any significant difference in your application's behavior with your single browser, but for heavy fragments in large production applications serving many thousands of concurrent users the savings can be dramatic.

Libraries

Prudence further helps reusability by easily allowing code libraries. Libraries are source code files with functions, classes, etc., that you'll want to reuse in your scriptlets and in other Prudence code.

Let's create an "/applications/wackywiki/libraries/" directory, and put "wikidata.js" in it:

```
function getWikiEntry(id) {
        // (Note the forced conversion to JavaScript string)
        switch(String(id)) {
                case 'index':
                        return {
                                title: 'I Love Prudence',
                                contents: 'This is a love letter to Prudence (from
                                    the database).'}
                case 'home':
                        return {
                                title: 'Wiki Home',
                                contents: 'This wiki page is empty (from the
                                    database).'}
                case 'todo':
                        return {
                                title: 'Things to Do',
                                contents: 'Nothing to do right now (from the
                                    database).'}
                default:
                        return {
                                title: '',
                                contents: ''}
        }
}
```

(This is just a silly example: in a real application, you'll probably be fetching your wiki entries from a data store.)

Here is our new "index.html" page:

```
<%& '/header/' %>
<p>Hello, world! Welcome to Wacky Wiki.</p>
<%
document.execute('/wikidata/')
var entry = getWikiEntry('index')
%>
<%& '/entry/' %>
<%& '/footer/' %>
```

We'll leave it as an exercize to you to make similar changes to "home.html".

Note the use of "document.execute" here: this is a Prudence API that lets you run code from /libraries/. We're using it here because the JavaScript language doesn't have it's own include facility. However, if you're using Prudence for Python, Ruby, PHP or Clojure, you can use those language's natural inclusion facilities: "import", "require", etc. The /libraries/ directory is automatically added to your inclusion path. Note that Groovy has the same limitation as JavaScript in this respect, so just use "document.execute" there.

Further Exploration

- Learn more about cache key patterns (page 68).
- Learn more about document.execute (page 71).
- Learn more about how Prudence makes your URLs pretty (page 54).

Hello, Captured World

Creating a file per URL can only take you so far. For example, in our Wacky Wiki we had "index.html" and "home.html", but we wouldn't want to create a file per wiki entry. What if we had a million entries?

The pages are generic enough that they could be generated by the same code. And so, Prudence has a feature called "capturing" that lets you use one file to handle many URLs: we can "capture" URLs that fit a generic URL pattern into a single file.

Let's capture our wiki URLs! First, let's create the generic page, "/web/dynamic/wiki.html":

```
<%& '/header/' %>
<p>Hello, world! Welcome to Wacky Wiki.</p>
<%
document.execute('/wikidata/')
var entryID = conversation.locals.get('entryid')
var entry = getWikiEntry(entryID)
%>
<%& '/entry/' %>
<%& '/footer/' %>
```

You see that we're expecting the "conversation locals" to contain the actual entry ID. This will be supplied by Prudence's capturing.

Now, let's capture by creating "/applications/wackywiki/routing.js" in our application's root directory:

```
document.execute('/defaults/application/routing/')
router.capture('/wiki/{entryid}/', '/wiki/')
```

Note that you have to restart Prudence in order for your "routing.js" to take effect. Then, try to access http://localhost:8080/wackywiki/wiki/home/, http://localhost:8080/wackywiki/wiki/index/ and http://localhost:8080/wackywiki/wiki/todo/.

We are capturing all URLs in the form of "wiki/.../" into '/web/dynamic/wiki.html", where the ellipsis could be any URL segment. We send all these URLs to one file, internally named "wiki/". (It's actually our "wiki.html" from above: remember, Prudence always ignores filename extensions and requires a trailing slash.) That URL segment is also captured by Prudence and sent as a special conversation local value named "entryid".

We can also edit our "index.html" to be a simple stub to display the "index" wiki page:

```
<%
conversation.locals.put('entryid', 'index')
document.include('/wiki/')
%>
```

Further Exploration

- Learn more about capturing (page 56).
- Learn more about the "routing.js" file (page 27).
- Learn how to write URI patterns for (page 63).
- Learn more generally about Prudence's elaborate routing system (page 53).
- Learn about conversation locals and other ways of sharing state in Prudence (page 80).

Hello, REST of the World

Introduction: What's REST?

Prudence is designed from the ground up around REST ("Representational State Transfer") principles, a term coined by Roy Fielding in Chapter 5 of his 2000 dissertation. Fielding's analysis perceptively identified the key features of the Internet infrastructure that have made the World Wide Web and email protocols scale so well horizontally. His conclusions have helped us make better use of available technology, and have generally inspired more practical approaches to the problem of sharing state in diverse, distributed environments.

In "The Case for REST" (page 5), we show how REST suggests a "resource-oriented architecture." The "resource" is the *thing* that lies behind the Internet URL. REST resources you are already familiar with include HTML pages, images, videos—everything we see in the World Wide Web. But resources can also be

thought of more logically as blog entries, social networking profiles, inventory items in a store or warehouse, etc. Moreover, you should think about resources as single things with multiple representations. An HTML web page might be one representation, but we can represent the same resources in any format, whether it's meant for direct human consumption, such as PDF or plain text, or a format that can be consumed by software, such as XML, JSON and proprietary binary formats.

The ubiquitous HTTP supports sophisticated negotiation of representational formats between the client and the server, and also multiple operations on resources: in addition to GET and POST, which are commonly used in HTML forms, you can also PUT new resources where none were before and DELETE existing resources. This makes HTTP an attractive protocol for distributed applications—and, of course, it's already there, deployed and ready for you to build upon.

Your First Resource

In Prudence, creating a RESTful resource is as easy as creating a dynamic web page.

Create an "/applications/wackywiki/resources/" directory to host your resources. Then, create an "entry.js" file in it:

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

function handleGet(conversation) {
        return 'Hello, REST of the World!'
}
```

The "media type" mentioned in the code is just an abstract way to refer to MIME types, which are a standard way to name representation formats in the Internet. (Actually, they were originally used to identify the format of email attachments.)

Point your browser to http://localhost:8080/wackywiki/entry/ to see your first resource. Not very interesting, is it? Let's add support for another representation format:

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

function handleGet(conversation) {
        if(conversation.mediaTypeName == 'text/html') {
                return '<html><h1>Hello, <i>REST</i> of the World!</h1></html>'
        }
        return 'Hello, REST of the World!'
}
```

Refresh you browser, and now you should see your resource represented as snazzy HTML. Because desktop web browsers announce, in their HTTP requests, that they *prefer* HTML, and Prudence properly negotiated the correct format from among those we stated in our handleInit function, we could give the client what it wanted.

You First *Real* Resource

OK, enough saying hello to the world! Let's create a resource that actually does something useful in Wacky Wiki. We're assuming here that you created the "/libraries/wikidata.js" we introduced previously. Here's our new "entry.js":

```
document.execute('/wikidata/')

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

```
}

function handleGet(conversation) {
        var entryID = conversation.locals.get('entryid')
        var entry = getWikiEntry(entryID)
        switch(String(conversation.mediaTypeName)) {
                case 'application/json':
                        return '{"title":"' + entry.title + '", "contents":"' +
                           entry.contents + '"}'
                case 'text/html':
                        return '<html><h1>' + entry.title + '</h1><p>' + entry.
                           contents + '</p></html>'
                default:
                        return entry.title + '\n' + entry.contents
        }
}
```

For this to work for all wiki entry URLs, we also have to "capture" them, using the method we learned above. Here's our updated "routing.js":

```
document.execute('/defaults/application/routing/')
router.capture('/entry/{entryid}/', '/entry/')
router.capture('/wiki/{entryid}/', '/wiki/')
```

You need to restart Prudence for this new "routing.js" to take effect. Then, you'll be able to see HTML representations of your wiki entries at http://localhost:8080/wackywiki/entry/index/, http://localhost:8080/wackywiki/entry/home/ and http://localhost:8080/wackywiki/entry/todo/.

But, what about those JSON representations? We'll get to them in the next section.

"AJAX"

Modern web browsers are more than consumers of HTML, images and other formats available via plugins (PDF, Flash, etc.). With their internal JavaScript engines, the are adequate RESTful clients. We'll use that feature to access our resource.

Accessing RESTful resources from within web browsers is sometimes called AJAX, which stands for "Active JavaScript and XML." In this example, we're preferring the more compact JSON format over XML, so perhaps we should call it "AJAJ"? Well, the end result is the same.

If you've ever worked with JavaScript in web browsers, you know how painful it is to write code that runs well on all major web browsers. For this example, we'll avoid that pain by using the cross-browser jQuery library, which is freely hosted by Google.

Let's add some AJAX magic to our "index.html":

```
<html>
<head>
        <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/
           jquery/1.4.2/jquery.js"></script>
</head>
<body>
        <p>Enter wiki page name:</p>
        <p><input id="entryid"></input></p>
        <p><button id="fetch">Fetch!</button></p>
        <h2 id="title"></h2>
        <p id="content"></p>
<script>
$(document).ready(function() {
        $('#fetch').click(function() {
                $.ajax({
                        type: 'GET',
                        url: 'entry/' + $('#entryid').val() + '/',
```

```
                        dataType: 'json',
                        success: function(entry) {
                                $('#title').text(entry.title);
                                $('#content').text(entry.contents);
                        }
                });
        });
});
</script>
</body>
</html>
```

We currently support three entry names: “index”, “home” and “todo”. Type any of them into the input box, and click “Fetch!”. You should see the results displayed in your browser. Make sure to use Firebug to see the browser communicating with your resource.

This little example might not seem too exciting, but think of what’s going on here: Prudence is giving the web browser “raw” data, which it can then represent in sophisticated ways. The burden of rendering the user interface has moved from the server to the client, allowing for an improved user experience on modern browsers, and more room for your server to scale.

But there’s a lot more to REST than AJAX: clients for your raw JSON data can be full-blown desktop applications, mobile phone apps, and even other servers on the network (perhaps also powered by Prudence). Taken together, your resources can be considered an API. In fact, a rich, RESTful API can be a valuable Internet product in itself. You can do great things without serving a single web page.

> We’ll repeat it, because it does get confusing: In the code segment above, the JavaScript is running *in your browser*, while in all the previous examples we had JavaScript running *in Prudence*. The language is the same, but the programming environment is vastly different. In the browser, you have access to the DOM and browser services. In Prudence, you have the Prudence API (page 65) as well everything in the JVM. We think it’s nice to be able to write your web application in a single language, whether it’s on the client or the server, but if you find it confusing, go ahead and use any of the other Prudence flavors: Python, Ruby, PHP, Clojure or Groovy.
>
> P.S. You may have noticed that we’ve used semicolons at the end of statements on our client code. Semicolons are optional in the JavaScript language, but many experts recommend you use them anyway *in web browsers*, because many code analysis and compression tools expect them. Since there’s no need to compress your server-side code, we recommend you follow your own preference there. We happen to prefer no semicolons.

Reader Meets Author

Though we won’t cover it in this tutorial, with Prudence—and HTTP, and AJAX—you can also accept data from the client into your resource. Use handlePut to create entirely new resources at the URL identified by the client. For example, Wacky Wiki clients can send you JSON data via a PUT, which you can then parse and save in the database as a new wiki entry. There’s also handlePost, used to update existing resources, and handleDelete for getting rid of them.

Safe and Sound

Worried about unauthorized access to your data? Prudence has a friendly API for cookies, and allows for sophisticated authentication mechanisms and secure connections.

Further Exploration

- Learn all about resources (page 40).
- Learn how to access any RESTful resource from Prudence (page 72).
- Learn about cookies (page 77).
- Learn about secure connections (page 24).

What Next?

The best way to learn Prudence is by creating your own applications. Why not turn Wacky Wiki into the next Internet hit, or into an online discussion board for your organization?

Furthermore, Prudence comes with example code, and even a complete, data-driven, RESTful application: Stickstick. Stickstick is a shared, online bulletin board on which anyone can post notes for everyone else to see. You can move notes around and delete them. The Stickstick source code demonstrates how you can save your resource data into a SQL database, scalable handling of conditional HTTP, and even a few AJAX tricks.

And then there's the Prudence Manual: very detailed, and available online in what we hope is a friendly, easy-to-use format. A PDF of it is included with all Prudence flavors, ready to print out or read on your e-reader.

Finally, make sure to join the Prudence Community online, to ask questions, contribute ideas and see how others are making good use of Prudence. And please report any bugs and make suggestions for improvement!

Before you run off to play with Stickstick, we'll mention just two important features that weren't covered in the tutorial:

Filters

Prudence lets you write code that will filter every incoming and outgoing request. This makes it easy to do a wide variery of general work such as authenticating, sanitizing, transformation/encoding, monitoring, throttling, caching, processing, etc. See filters (page 48) for complete documentation.

Tasks

Prudence lets you write and run tasks asynchronously, and even to schedule them using a crontab-like configuration file. This is very useful for maintenance work such as releasing unused connections and sessions, for sending email reminders and administrative alterts, for monitoring services, etc. See tasks (page 49) for complete documentation.

The Prudence Instance

Here we'll describe the directory and file structure of a Prudence instance, and how to configure it.

Subdirectory Structure

/applications/

This is where you deploy your applications, each occupying a subdirectory. Your application's subdirectory name serves as a useful default name for your application. It is also the default base URL, and the default logging name.

The /applications/ subdirectory is "zip-aware," meaning that zip archives placed here will be automatically expanded.

See configuring applications (page 25) for a complete reference.

/bin/

These shell scripts start up the Prudence instance. Use run.sh for Unix-like operating systems, such as Linux, *BSD and Mac OS X, and run.bat for Windows.

> In production environments, it's best to run Prudence as a daemon (page 87) in Unix-like systems or a service in Windows, via a lightweight wrapper.

What the /bin/ scripts do is:

1. Set the JVM classpath to include all the JARs in /libraries/
2. Start the JVM
3. Delegate to your /instance/ script

/instance/

Your Prudence instance's configuration scripts (page 21) go here. You need, at the very least, a "default.*" script.

/defaults/

The default configuration scripts (page 21) for both the Prudence instance and for applications go here.

/logs/

This is where your rolling logs will appear. Prudence logs are highly configurable and powerful. In particular, your web.log will show all requests hitting your Prudence server, using a standard format that can be consumed and analyzed by many monitoring tools.

See logging (page 84).

/configuration/

There are a few essential configuration files here.

- logging.conf: see logging (page 85)
- wrapper.conf, yajsw.conf: see "Prudence As a Daemon" (page 87)
- velocity.conf: for Velocity, see "Generating HTML" (page 36)
- hazelcast.conf: for Hazelcast, see document.cache (page 69)

/libraries/

Here you will find Prudence's main libraries as well as support libraries. The main /libraries/ subdirectory is for Java archives (JARs), but you may find subdirectories for libraries in other languages, such as /libraries/python/ for Python.

This directory also serves as a common /libraries/ directory for all applications, similarly to how an individual application's /libraries/ directory (page 26) works. You can thus put source code here that you want to share for all your applications.

If you add your own JARs, make sure to edit the scripts in /bin/ to accommodate your additions.

> We've named or renamed all Prudence JARs according to their main Java package prefix, but you do not have to follow this convention.

Configuration Scripts

In Prudence, most instance and application configuration files are written in programming language code, rather than XML or other configuration formats. This method is sometimes called "bootstrapping" or just "scripting." So, rather than referring to them as "configuration files," we'll call them "configuration scripts."

Scripting is powerful. It lets your configuration be dynamic, such that the same configuration script might configure things differently depending on the actual deployment environment and runtime circumstances. For example, you can deploy the same instance to a development environment, in which case your instance would start various debugging processes and logs, and a production environment, in which case it would optimize for performance and scale. You can even have different optimizations for different deployments. For example, a weak cloud-based instance with a single virtual CPU could be use a different HTTP engine from a dedicated box with 8 cores and a lot of RAM. With scripts, you can dynamically test for the presence of installed optional components, etc.

But you can do more with configuration scripts: anything, in fact, that the language and its libraries allow you. You can start external services, log startup errors, notify monitoring daemons, etc.

Prudence's configuration scripts are designed to fall back on default scripts, which are all in the /defaults/ subdirectory. These idea is for these scripts to handle configuration sensibly, as would be appropriate, or good enough, for most common use cases. You can do a lot with Prudence without changing any of these default scripts!

The common way to override any default script is to create your own version, execute the default script first, and then apply your overrides. You can also *not* execute the default script, and instead handle things your own way. You can edit the defaults directly to apply changes across the board: they are all simple, straightforward scripts.

The disadvantage of configuration by script is more limited portability. For example, if you write your configuration scripts in Ruby, they will not work in Prudence for Python—*unless* you manually install Python support. If you want to support all Prudence flavors, you would have to include scripts for all languages.

> Another disadvantage, for now, is that you will need to restart the Prudence instance for changes in your configuration scripts to take effect. This is true for Prudence 1.1: we plan to support "hot" configuration in a future version of Prudence.

For the purposes of this manual, we'll use the "filename.*" for configuration script names. To get the actual script file name, replace the "*" with the filename extension appropriate for your Prudence flavor: "js" for JavaScript, "rb" for Ruby, "py" for Python, etc.

/defaults/

Here you'll find the default scripts used in /instance/ and /applications/. Generally, you won't want to edit the /defaults/ scripts directly, but instead override them there. Prudence tries to make configuration easy by implementing sensible default behavior with a straightforward override mechanism: simply execute the /defaults/ script and then apply your own code. It's always a good idea to at least look at the code in the /defaults/ scripts so you can precisely understand what you're overriding.

Here's an overriding example in the JavaScript flavor. We'll override the default cache installed in the component by creating a "/instance/component.js" script:

```
// Implement defaults
document.execute('/defaults/instance/component/')

// Replace default cache with a Hazelcast-based cache
importClass(com.threecrickets.prudence.cache.HazelcastCache)
component.context.attributes.put('com.threecrickets.prudence.cache',
        new HazelcastCache())
```

Rather than explain the role of individual /defaults/ scripts here, we will refer to them as they are used in the sections below.

/instance/

This is where the Prudence instance gets initialized. The instance is the overall container which manages the servers and the applications.

/instance/default.*

This is the only required instance configuration script, and is the entry point for Prudence. Usually, all it does is simply execute /defaults/instance/, which bootstraps the Prudence instance.

You'd likely prefer to override the other configuration scripts. Override "/instance/default.*" if you want to do things after everything else in the instance has been initialized.

The default script does the following:

1. Prints the Prudence welcome message to the console
2. Sets up logging, including applying /configuration/logging.conf
3. Executes /instance/component/ or /defaults/instance/component/
4. Executes /instance/clients/ or /defaults/instance/clients/
5. Executes /instance/routing/ or /defaults/instance/routing/
6. Executes /instance/servers/ or /defaults/instance/servers/

7. Starts the component
8. Submits bootstrap tasks for execution

/instance/component.*

The "component" in REST terminology refers to the highest-level communication abstraction: the network is made of "components" communicating with other "components." Every Prudence instance is in essence a single component, which can include multiple servers and clients.

Override this script to change the way the component is created. In particular you might want to change the default cache backend or executor.

The default script does the following:

1. Creates a Restlet component
2. Configures its logService
3. Sets its statusService to be a DelegatedStatusService
4. Creates a global, thread pool executor, which is used for application.task (page 66)
5. Creates an InProcessMemoryCache backend, which is used for document.cache (page 69)

/instance/clients.*

Override this script to add more clients to your component.

The default script adds file and HTTP clients. The file client is required for static web support, and also for you to access files on your filesystem as REST resources. The HTTP client is required for you to access external REST resources on the web.

/instance/routing.*

Override this script to change the way the component initializes its routing. Because this script delegates to /instance/hosts/ and to your /applications/, it's more likely that you'll want to override those.

The default script does the following:

1. Executes /instance/hosts/ or /defaults/instances/hosts/
2. Initializes all applications in /applications/ (this list is accessible as "com.threecrickets.prudence.applications" in the component's context)
3. If there are no applications, the Prudence instance is terminated

/instance/hosts.*

Virtual hosting allows you to serve multiple web sites from a single component. You can configure individual applications to attach one or more hosts, and to have different base URLs on each of these hosts. Prudence allows you to create virtual hosts for multiple domain names and HTTP ports, using simple expressions and wildcards. See virtual hosts (page 53).

By default, an "allHosts" virtual host is created, and set as the component's default host, which in turn all applications attach to by default. "allHosts" accepts all incoming requests, to any domain name, to any port.

Override this script to create additional virtual hosts for your component, and to change the default host.

For example, you might want your Prudence component to only serve requests for "www.mysite.org" instead of the permissive "allHosts." Or, you might want to serve multiple web sites with different sets of applications on each.

The default script does the following:

1. Creates the "allHosts" virtual host
2. Sets it as the component's default host

/instance/servers.*

Servers do the low-level work of listening to a port, accepting requests, and returning responses according to supported protocols. Prudence currently supports your choice among various HTTP server technologies: Jetty, Grizzly, Netty and an "internal" connector. To change the HTTP server technology, you have to adds JARs it to /libraries/. See the Restlet documentation for more information.

You can create as many servers as you need to listen on multiple ports. Remember that routing is actually handled by virtual hosts, not the servers. So, if you have servers on ports 8081, 8082, and 8083, and your applications all attach to "allHosts," then the applications will be available on all ports. To limit applications to specific ports, you will need to create additional virtual hosts. See virtual hosts (page 53).

Override this script to change the default port (8080) or add additional servers on other ports.

The default script does the following:

1. Creates an HTTP server on port 8080, with support for the X-FORWARDED-FOR header used by proxies
2. Prints out information about this server

Secure Servers (https) If you are using a load balancer or another kind of proxy (page 89), it makes more sense to handle secure connections there. See the example there on how to do this with Perlbal.

Note, though, that if you want to handle secure requests differently in Prudence, you must allow for some way to distinguish them. This can easily be done by creating a separate non-secure server to which the proxy would send secure requests. You can then use virtual hosts (page 53) or other kinds of routing to send requests to different applications or resources. Alternatively, you could check for the reference (page 74) in code. For example:

```
if(conversation.reference.schemeProtocol.name == 'HTTPS') {
        ...
}
```

Prudence can also handle secure connections itself. Here's an example "/instance/severs.js":

```
// Implement defaults
document.execute('/defaults/instance/servers/')

// Create secure server listening on the default HTTPS port (443)
var secureServer = new Server(Protocol.HTTPS, 443)
secureServer.name = 'secure'
component.servers.add(secureServer)

// Configure it to use our security keys
secureServer.context.parameters.add('keystorePath', '/path/prudence.jks')
secureServer.context.parameters.add('keystorePassword', 'mykeystorepassword')
//secureServer.context.parameters.add('keyPassword', 'mykeypassword')

// Add support for the X-FORWARDED-FOR header used by proxies
secureServer.context.parameters.add('useForwardedForHeader', 'true')
```

See the Restlet Jetty HttpServerHelper documentation for more configuration parameters.

The above configuration script assumes the you have a JKS ("Java keystore") file at "/path/prudence.jks" with your security keys. You can create one with the "keytool" command that is bundled with most JVMs. For example:

```
keytool -keystore /path/prudence.jks -alias mykey -genkey -keyalg RSA
```

When creating the keystore, you will be asked provide a password for it, and you may optionally provide a different password for your key, in which case you need to comment out the relevant line in the "servers.js" above. (The key alias and key password would be kept if you move the key to a different keystore.)

Such self-created keys may be 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 major browsers. Some of these certificate authorities may let your get your key in JKS

format. Otherwise, if they support PKCS12 format, you can use keytool (from version 6) to convert it to JKS. For example:

```
keytool -importkeystore -srcstoretype PKCS12 -srckeystore /path/prudence.pkcs12 -
    destkeystore /path/prudence.jks
```

If you don't even have access to a PKCS12 file, you can create one from your ".key" and ".crt" (or ".pem") files using OpenSSL:

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

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

Prudence Applications

Prudence applications live in the /applications/ subdirectory, with one subdirectory per application. The subdirectories mentioned below should all be considered as subdirectories of an application subdirectory. Deploying an application to a Prudence instance involves nothing more than creating a subdirectory here with the application files.

Deploy by Zip

The /applications/ subdirectory is "zip-aware," meaning that any archive file with the .zip extension placed here will automatically be unzipped when Prudence starts up.

There's no reason to delete the archive: Prudence will only unzip the file again if its modification date changes (it creates an "applications.properties" file to keep track of that). In fact, a good way to restore an application to its defaults if you've made modifications is to delete the subdirectory, but not the archive. The next time Prudence is started, it will unzip the archive again.

> This above is true for Prudence 1.1. A future version of Prudence will feature "live" zip-awareness, so that you will not have to restart the Prudence instance in order to take new or updates zips into account.

Zips can contain pre-packaged single applications, groups of applications, and even patches for individual applications. Zip-awareness makes it very easy to package, deploy and upgrade applications and even large sites containing many applications.

We encourage you to follow our convention and name your deployable zips with versioning information and the ".prudence.zip" extension, for example "wackywiki-1.1.prudence.zip".

Subdirectory Structure

/web/dynamic/

This is where you'll put your dynamic HTML files. By "dynamic" is meant that they are generated on demand, such that each request can potentially produce a different, cacheable result. Prudence has a powerful framework for embedding programming language code into HTML as "scriptlets." (Actually, it supports textual formats other than HTML, such as XML, RSS, JSON, etc.) See generating HTML (page 31).

The names of the files and subdirectories under /web/dynamic/ attach to URLs, with simple intelligence to make it easy for you to create sensible, pretty URL schemes. See pretty URIs (page 54) for more information.

/web/static/

This subdirectory works like a standard "static" web server. Files put here are attached to URLs and accessible to clients as is. Prudence uses non-blocking I/O for high performance, scalable delivery of data to clients.

Like many web servers, MIME types for HTTP headers are automatically assigned according to the filename extension. For example. ".html" files will be sent as "application/html", and ".png" files will be sent as "image/png".

The static web subdirectory will likely "just work" for you as is. See static web (page 51) for extras, such as adding filename-extension-to-MIME-type mappings and compressing JavaScript/CSS.

/web/fragments/

Your dynamic pages in /web/dynamic/ can include any page fragments from here. The advantage of putting them here rather than there is that here they will not be attached to URLs. Fragments are normally hidden from users, visible only if they're included into a page in /web/dynamic/.

Fragments allow you to compose complex pages out of reusable building blocks. Another important use is for fine-grained caching: each fragment has its own caching behavior. See generating HTML (page34).

/resources/

Whereas the /web/dynamic/ subdirectory has HTML (or other textual) files with embedded programming language code, /resources/ is pure code. This is Prudence's most flexible development feature: files here are attached to URLs as REST resources, capable of handling all HTTP verbs and responding with appropriate representations in any format.

From the perspective of web development, consider that if /web/dynamic/ lets you write HTML-based front ends for "thin" clients, such as simple web browsers, /resources/ lets you handle "rich" clients, such as AJAX, Flash and other dynamic platforms.

See resources (page 40) for a complete guide.

/handlers/

Handlers (page 47) work similarly to resources, except that they are much simpler: they are not mapped to URLs. Instead, they must be explicitly used for specific purposes.

/tasks/

Tasks (page 49) are straightforward scripts that can be run on-demand or scheduled to run and specific times. They are straightforward in that, upon execution, the code is simply run as is. There is no particular entry point as with /resources/ or /handlers/.

/libraries/

All your code, whether it's in /resources/, /handlers/, /tasks/ or in scriptlets embedded in /web/dynamic/ and /fragments/, can include code from /libraries/.

Use whatever mechanism is appropriate for your language: "import" for Python or Ruby, "use" for Clojure, etc. For languages that don't have inclusion mechanisms—Groovy, JavaScript—you can use Prudence's inclusion mechanism, document.execute (page 71).

To share source code with all your applications, use the instance's /libraries/ directory (page 21).

/uploads/

Files uploaded via HTML forms will be placed here, if they are larger than a configurable threshold. Otherwise, they are stored in memory. See conversation.form (page 75) for more information.

Configuration Scripts

The application's configuration scripts are in its base subdirectory.

Make sure you read the section on configuration scripts for the Prudence instance (page 21) for general guidelines.

/default.*

You'll rarely need to do it, but you can also override Prudence's default application bootstrap.

Here you can modify an application's filename extensions to media type mappings. An example "default.js":

```
document.execute('/defaults/application/')
applicationInstance.metadataService.addExtension('jsp', MediaType.TEXT_HTML)
```

/settings.*

Here you can override some of Prudence's defaults for your application, such as the subdirectory structure detailed here, the default URLs, include some distribution information, configure the logging name, etc.

You can also add your own runtime settings for your code to use, such as database usernames and passwords.

See the settings section (page 27) for full detail on overrides and default settings.

/routing.*

The settings file gives you some control over the default URLs, but here you can manipulate them extensively. Your routing tools are very powerful, including redirection based on URL patterns, regular expressions, and route scoring.

In particular, this is where you install URL patterns for your resources. For example, you can attach /item/{id}/ to your item resource, and have "id" automatically extracted from the URL.

This is also where you can attach your own custom (non-Prudence) resources to URLs. Actually, anything that's a "restlet" will do, because Prudence uses Restlet for its resource routing. See the section on integrating custom restlets (page 91).

See routing (page 53).

/application.*

Use this to install non-Prudence Restlet applications into Prudence. By default, Prudence creates a PrudenceApplication, but you can override that creation in this file. See the section on integrating Restlets (page 91).

/crontab

This is the configuration file that not a script. It lets you schedule tasks to run at certain Calendrical times. See tasks (49).

Settings

Each application's subdirectory name under the /applications/ directory is used as a default for many settings.

In this manual, we've used setting names in camel case, as is used by JavaScript and Groovy. For Python, Ruby and PHP use lowercase underscore notation, for example: "application_home_url." For Clojure, use lowercase hyphenated notation, for example "application-home-url."

Information

These are for administrative purposes, such as the Prudence administration application, and are also used for the default error message pages.

They are directly available at runtime via application.application (page 66).

applicationName Defaults to the application's subdirectory name.

applicationDescription A short description to help explain what the application is, in case the application name is too obscure.

applicationAuthor The entity (person, corporation) that created the application.

applicationOwner The entity (person, corporation) that owns the rights to the application.

applicationHomeURL The URL (Internet or intranet) of the application. This may be displayed in certain default error messages, as a way to contact the application administrators or technical support. Note that there is only one home URL, even if the application is attached to multiple virtual hosts. In fact, the home URL does not have to be service by this Prudence instance, and can lead to a different site.

applicationContactEmail This may be displayed in certain default error messages, as a way to contact the application administrators or technical support.

Debugging

showDebugOnError Set to true to show debug information on error. See debugging (page 83).

showSourceCodeURL The base URL for showing source code (only relevant when showDebugOnError is true).

minimumTimeBetweenValidityChecks This is the time (in milliseconds) allowed to pass until a script file is tested to see if it was changed. During development, you'd want this to be low, but during production, it should be high in order to avoid unnecessary hits on the filesystem. Defaults to 1000 (1 second).

Logging

applicationLoggerName Defaults to the application's subdirectory name.

Routing

hosts This is a array of array of two elements: the first is the virtual host to which our application will be attached, the second is the base URLs on the hosts. This lets you attach the application to multiple virtual hosts with a different base URL per host.

Specify null for the URL to default to the application's subdirectory name. Use "component.defaultHost" for the default virtual host, as defined by the "hosts" configuration script (page 23).

If not "hosts" setting is specified, it defaults to attach to the default virtual host via the application's subdirectory name. This is equivalent to:

```
hosts = [[component.defaultHost, null]]
```

Here's an example of a more complex "hosts" setting:

```
hosts = [[myorgHost, null], [intranetServiceHost, '/'], [b2bHost, '/b2b/service/']]
```

Note that URLs in the virtual host may overlap as far as hierarchy is concerned. Thus, an application at "/b2b/" can have its own URLs, such as "/b2b/access/", and this would not interfere with the application at "/b2b/service/". The application is routed before URLs internal to the application. See routing (page 53) for a full explanation on how URLs reach their destinations in Prudence.

Resources

Sets up a directory under which you can place script files that implement RESTful resources. The directory structure underneath the base directory is directly linked to the base URL.

resourcesBaseURL Defaults to the root ("/"), meaning that the directory and file structure maps directly to URLs at the application's base URL on the virtual host. See routing (page 53) for more information.

resourcesBasePath Relative to the application's subdirectory. Defaults to "/resources/".

resourcesDefaultName If the URL points to a directory rather than a file, and that directory contains a file with this name, then it will be used. This allows you to use the directory structure to create nice URLs without relying on filenames. Defaults to "default".

resourcesDefrost Set this to true if you want to start to load and compile your resources as soon as Prudence starts. Defaults to true.

resourcesSourceViewable This is so we can see the source code for scripts by adding ?source=true to the URL. You probably wouldn't want this for production environments. Defaults to true.

Dynamic Web

Sets up a directory under which you can place text files that support embedded scriptlets. Note that the generated result can be cached for better performance. See generating HTML (page 31).

dynamicWebBaseURL Defaults to the root ("/"), meaning that the directory and file structure maps directly to URLs at the application's base URL on the virtual host. See routing (page 53) for more information.

dynamicWebBasePath Relative to the application's subdirectory. Defaults to "/web/dynamic/".

dynamicWebDefaultDocument If the URL points to a directory rather than a file, and that directory contains a file with this name, then it will be used. This allows you to use the directory structure to create nice URLs that do not contain filenames. Defaults to "index".

dynamicWebDefrost Set this to true if you want to compile your scriptlets as soon as Prudence starts. Defaults to true.

dynamicWebPreheat Set this to true if you want to load all your dynamic web documents as soon as Prudence starts. Defaults to true.

Also see preheatResources (page 30).

dynamicWebSourceViewable This is so we can see the source code for scripts by adding ?source=true to the URL. You probably wouldn't want this for most applications. Defaults to true.

dynamicWebClientCachingMode If you set server-side caching with document.cacheDuration (page 68), then you can use this setting to define whether client-side caching should be enabled, too:

- 0: Disabled. Client caching headers are not sent.
- 1: Conditional (the default). The client is asked to use conditional mode HTTP to verify that the cache has not changed, via the modificationDate and expirationDate headers. This is a good default, because it generally offers most of the advantages of caching with no risks. See conditional requests for HTML generation (page 39) for a complete discussion.
- 2: Offline. The client is asked to cache without verification, via the maxAge header. This involves some risk: if you ask to cache a page for one week, but then find out that you have a mistake in your application, then users will not see any fix you publish until their local cache expires, which can take up to a week! It's important that you understand the implications before using this mode.

  It's generally safer to apply offline caching for some of your /web/static/ resources, such as graphics and other resources, which, by nature of being static, do not tend to change on their own. See CacheControlFilter (page 51). For /resources/, you have precise control over each header; see conversation.modificationDate, conversation.expirationDate and conversation.tag (page 78).

Static Web

Sets up a directory under which you can place static files of any type. See static web (page 51).

staticWebBaseURL Defaults to the root ("/"), meaning that the directory and file structure maps directly to URLs at the application's base URL on the virtual host. See routing (page 53) for more information.

staticWebBasePath Relative to the application's subdirectory. Defaults to "/web/static/".

staticWebCompress Whether to enable smart compression before sending files over HTTP. Only uncompressed files will be compressed, and only if there are bandwidth savings will the compressed version be used. Defaults to true.

> If you're running Prudence behind a proxy (page 89) that already does compression, and are worried that you're compressing twice, then stop worrying: any decent proxy will know not to compress content that's already compressed.

staticWebDirectoryListingAllowed If the URL points to a directory rather than a file, then this will allow automatic creation of a simple HTML page with a directory listing. Defaults to true.

File Uploads

fileUploadSizeThreshold The size in bytes beyond which uploaded files will be stored to disk. Defaults to zero, meaning that all uploaded files will be stored to disk. See conversation.form (page 75).

Handlers

handlersBasePath Relative to the application's subdirectory. Defaults to "/handlers/".

handlersDefaultName If the handler name points to a directory rather than a file, and that directory contains a file with this name, then it will be used. This allows you to use the directory structure to create nice names without relying on filenames. Defaults to "default".

Tasks

tasksBasePath Relative to the application's subdirectory. Defaults to "/tasks/".

tasksDefaultName If the task name points to a directory rather than a file, and that directory contains a file with this name, then it will be used. This allows you to use the directory structure to create nice names without relying on filenames. Defaults to "default".

Preheater

"Heating up" means that an internal GET verb is sent to them, with the result discarded. This is a good way to ensure that resources, such as database connection pools, are initialized and ready to service user requests as soon as possible.

preheatResources Note that all /resources/ and /web/dynamic/ files are automatically preheated. List additional URLs here that you want heated up as soon as Prudence starts. This is most useful in conjunction with custom routing, such as capturing (page 56).

Also see dynamicWebPreheat (page 29).

Miscellaneous

urlAddTrailingSlash The URLs in this array will automatically be redirected to have a trailing slash added to them if it's missing.

> By default, URLs in /dynamic/web/ and /resources/ have trailing slash *required*. See the trailing slash requirement for routing (page 55) if you wish to disable this.

predefinedGlobals Values set in this map will be available in application.globals (page 65) at runtime. This is useful for custom configuration of your application. For example, you can set resource IP addresses, service usernames and passwords, etc.

predefinedSharedGlobals Values set in this map will be available in application.sharedGlobals (page 65) at runtime. This is useful for custom configuration of your application. For example, you can set resource IP addresses, service usernames and passwords, etc.

Generating HTML

Prudence has excellent support for generating HTML by allowing you to embed programming language code in it, and allowing for superior performance and scalability via a sophisticated, fine-grained caching system.

Note that Prudence can also generate HTML at a "lower" level, via resources (page 40) that return HTML representations.

> While we're titling this section "Generating HTML," all of these methods can be used to generate any textual format, such as XML, JSON, etc.

Files

All files under /web/dynamic/ are assumed to be text files, and are mapped to URLs using the directory and file path, but without the filename extension. For example, the file "/web/dynamic/thread/list.html" would map to a URL like "http://mydomain.org/forum/thread/list/". You can also "capture" (page 56) many URLs to be handled by a single file.

The filename's extension will be used to map the default MIME type for the file. For example, ".html" files will be "text/html", ".json" files will be "application/json", and ".xml" files will be "text/xml". You can easily change these default mapping via application configuration (page 27). Additionally, you can explicitly set the MIME type in a scriptlet (page 31) via conversation.mediaTypeName (page 76).

The above is a very brief overview; see routing (page 53) for an in-depth explanation of how files become URLs.

All text is implicitly assumed to be Unicode encoded as UTF-8. Use conversation.characterSet (page 76) to explicitly translate it to other Unicode and non-Unicode character encodings.

Scriptlets

Programming language code can be embedded in your /web/dynamic/ files, making them all-powerful templates. Either "<% %>" or "<? ?>" delimiters can be used. Note, however, that you can only use one of either in the same file. The first type found will be the one used.

For historical reasons, these embedded bits of code are called "scriptlets." However, even though the word "script" tends to connote interpreted languages, scriptlets are usually compiled, not interpreted.

The standard output stream is the HTTP response. Thus, any "print"-like statements will be sent to the client.

In fact, behind the scenes, all non-scriptlet text is turned into code. It's a simple "print" of the non-scriptlet text into the output stream. Thus, you can freely mix scriptlet and non-scriptlet code:

```
<html>
<body>
<% for(var i = 0; i < 10; i++) { %>
This is line number <% print(i); %> out of 10.<br/>
```

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

The above translates into something like this:

```
print('<html>\n<body>\n');
for(var i = 0; i < 10; i++) {
print('This is line number ');
print(i);
print('out of 10.<br/>');
}
print('</body>\n</html>');
```

Newlines and Whitespace

It's important to remember that non-scriptlet text turns into "print" statements in your scriptlet language: these statements can have an effect on your code blocks, such as if/then blocks, loops, try/catch, closures, etc. Careless use of scriptlets can lead to syntax errors or, worse, to unexpected behavior.

For example, this JavaScript code will generate a syntax error:

```
<% var x = 0 %>You won't see this text!
```

The reason is that it produces the following mangled JavaScript code:

```
var x = 0 print('You won't see this text!')
```

Another example can show possibly unexpected results:

```
<% for(var i = 0; i < 10; i++) %>This line appears 10 times<br>
And so does this line!<br>
```

If you expect the first line to appear 10 times and second line to appear once, you will be disappointed. The reason is that the newline is part of the "print", not part of your scriptlet code. It produces this JavaScript code:

```
for(var i = 0; i < 10; i++) print('This line appears 10 times<br>\nAnd so does this
    line!<br>')
```

Two ways to avoid this problem are:

1. Always put newlines in the beginning and end of your scriptlets, so that it's clear at a glance where statements begin and end:

   ```
   <%
   var x = 0
   %>
   You WILL see this text!
   ```

2. Alternatively, you can explicitly put end-of-statement marks, if your language supports them. For example:

   ```
   <% var x = 0; %>
   You WILL see this text!
   ```

Special Note for Python Python represents a special challenge in a scriptlet environment due to its reliance on significant whitespace for blocks. This means taking special care in mixing scriptlets with text. For example:

```
<%
for x in range(0,10):
  %>
This line appears 10 times<br>
```

```
<%
%>
But this line appears once<br>
```

You'll notice that we put the end delimiter of the first scriptlet after an indentation. This tells Python that whatever follows (the "print" statement for the non-scriptlet text) will be in that for loop. We then used a scriptlet with only a newline to end the for loop.

A more complex example:

```
<%
for x in range(0,10):
  for y in range(0,2):
    %>
This line appears 20 times<br>
<%
  %>
This appears every third line<br>
<%
%>
But this line appears once<br>
```

The trick here was using a scriptlet with an indentation to end the inner for loop, while keeping us in the outer for loop.

Tricky indeed, and definitely not easy to decipher at a glance. It may make more sense to use Python's "print" statement in such situations:

```
<%
for x in range(0,10):
  for y in range(0,2):
    print 'This line appears 20 times<br>'
  print 'This appears every third line<br>'
%>
But this line appears once<br>
```

API

Beyond what's available to your programming language, scriptlets have access to Prudence's elaborate, powerful set of services via its API (page 65).

Code Libraries

Code files you put in your application's /libraries/ subdirectory can be accesses using your language's code inclusion mechanism. Here are examples in various languages. Let's assume that there is a file named "/libraries/util/data.*", which defines a function called getData():

- Python:

  ```
  from util.data import getData
  data = getData()
  ```

- Ruby:

  ```
  require 'util/data.rb'
  $data = get_data
  ```

- Clojure:

  ```
  (use 'util.data)
  (def data (get-data))
  ```

- PHP:

```
require 'util/data.php';
$data = get_data();
```

- JavaScript and Groovy do not have a built-in code inclusion mechanism, but they can use Prudence's document.execute API (page 71):

```
document.execute('/util/data/')
data = getData()
```

Expression Scriptlets

A common idiom is to print out expressions interwoven with non-scriptlet text. The expression scriptlet, marked by an equal sign, can help you reduce clutter. It's equivalent to a simple "print" of the expression. For example, the above example could be shortened into:

```
<html>
<body>
<% for(var i = 0; i < 10; i++) { %>
This is line number <%= i %> out of 10.<br/>
<% } %>
</body>
</html>
```

Expression can be as elaborate as needed. For example, let's show line numbers above 5 in bold:

```
This is line number <%= i>5 ? '<b>' + i + '</b>' : i %> out of 10.<br/>
```

Comment Scriptlets

These scriptlets are ignored. Useful for adding comments, and for "commenting out" code temporarily during development:

```
<html>
<body>
<%#
This whole scriptlet is ignored.

var i = ugh... i don't know. will deal with it later.
%>
</body>
</html>
```

Include Scriptlets and Fragments

You can include other files in-place using an include scriptlet, marked by an ampersand:

```
<html>
<body>
<% for(var i = 0; i < 10; i++) { %>
<%& '/lines/simple/' %>
<% } %>
</body>
</html>
```

The above example assumed a file named "/web/dynamic/lines/simple.html", with the following contents:

```
This is line number <%= i %> out of 10.<br/>
```

Just like an expression scriptlet (page 34) is equivalent to a "print", an include scriptlet is equivalent to calling document.include (page 70). This, too, means that the include expression can be as elaborate as needed:

```
<%& '/lines/' + (i>5 ? 'emphasized/' : 'simple/') %>
```

Separating code into separate files that are then included is a very important technique for two reasons:

1. It makes your application far more manageable by allowing you to reuse code blocks. A common use case is to have special "header.html" and "footer.html" files included by all pages in the application.
2. In Prudence, each document has its own caching characteristics, allowing you to create elaborate caching schemes by splitting a file into smaller files. See caching (page 37) and "Scaling Tips" (page 95).

/web/fragments/ In the example above, our "/web/dynamic/lines/simple.html" file would be exposed via a URL. However, we do not intend this file to served in itself, only as a file to be included in other files.

To avoid external exposure, put files in /web/frament s/ instead of /web/dynamic/. Use is otherwise identical. For example, we'd move our file above to "/web/fragments/lines/simple.html". Files in /web/fragments/ are only available via include scriptlets.

An alternative to using /web/fragments/ is explicit URI hiding (page 56).

Mixing Languages

By default, Prudence will assume your scriptlets to be in the source code of your Prudence flavor. For example, scriptlets in Prudence for JavaScript will be assumed to be JavaScript by default.

However, Prudence lets you use scriptlets of any language by adding the language name after the first scriptlet delimiter. For example, assuming we're using the JavaScript flavor:

```
<html>
<body>
JavaScript:<br/>
<%

// Here's some JavaScript code
for(var i = 0; i < 10; i++) {
        print('Line ' + i + ' of 10<br/>')
}

%>
Python:<br/>
<%python

# Here's some Python code
for i in range(1, 10):
        print 'Line %s of 10<br/>' % i

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

Note that once you "switch" to another language, all subsequent scriptlets will be assumed to be of that language unless you explicitly set them to something else.

Expression and include scriptlets work the same way. For example:

```
<%&javascript 'lines/' + (i>5 ? 'emphasized/' : 'simple/') %>
```

> Note that scriptlets in a specific language will only work if you have the the appropriate language JARs under your /libraries/ directory. The Prudence "Kitchen Sink" Edition comes with all languages supported by Prudence.

Templating Languages

You might think that mixing programming languages is a bad idea in general, and only (painfully) necessary for making use of legacy code. However, it can be a *great* idea if you consider that all Prudence flavors come with two high-performance templating languages, Velocity and Succinct.

By mixing languages, you can write all the more straightforward templating in a templating language, switching to full programming language scriptlets only when you need advanced features. For example, here we'll use JavaScript to set up our data, and then use Velocity to print it out:

```
<html>
<body>
<%
var country = myDatabase.fetchCurrentCountry()
conversation.locals.put('country', country)
%>
<h2>Famous Quotes:</h2>
<%velocity

<p>The rain in $conversation.locals.country stays mainly in the plain.</p>

## Here is a Velocity comment
%>
</body>
</html>
```

Note the use of conversation.locals (page 79) above to pass data between scriptlets of various languages.

Not only is the Velocity code more readable and easier to manage, but it also performs better, is less prone to errors, and more secure. Prudence lets you use the right tool for the job.

Refer to the documentation of Velocity and Succinct to learn more about these templating languages.

In-Flow Scriptlets

One tiny hiccup to be aware of when mixing languages (page 35) is that code blocks can not trivially span several scriptlets in multiple languages. For example, this won't immediately work:

```
<html>
<body>
<%javascript
for(var i = 0; i < 10; i++) {
        conversation.locals.put('i', i)
%>
<%velocity
This is line $conversation.locals.i out of 10.<br/>
%>
<%javascript
}
%>
</body>
</html>
```

The reason is that, by necessity, Prudence runs scriptlets of different languages separately, in sequence. The language switch thus represents a syntactical boundary. The above would cause a syntax error, because we have a for-loop in the first scriptlet that isn't closed.

But, fear not, Prudence lets you solve this problem via the "in-flow" scriptlet, marked by a colon. Here's the exact same code as above, but with the Velocity scriptlet marked with a extra colon:

```
<html>
<body>
<%javascript
for(var i = 0; i < 10; i++) {
```

```
        conversation.locals.put('i', i)
%>
<%:velocity
This is line $conversation.locals.i out of 10.<br/>
%>
<%
}
%>
</body>
</html>
```

This works! But how, you might wonder? Behind the scenes, the in-flow scriptlet is run from within the enclosing language. We thus never leave the enclosing language for the purposes of running through the file. Don't worry about performance here: in-flow scriptlets are compiled only once per file. Also, in-flow scriptlets are treated as regular scriptlets if there is no language switch, making them always safe to use if you're not sure.

One last note: in-flow scriptlets to not change the scriptlet language, so that we did not have to mark the closing bracked scriptlet as JavaScript.

Life

Files and all their scriptlets are only compiled once, when they are first requested by a client. The compiled code is kept in a cache, under your Prudence /cache/ directory. From then on, each request is handled by the compiled version, which is updated only if the original file changes. This cache is maintained even if you restart Prudence.

Unless a valid cache entry exists, the compiled code is executed anew for each request, essentially like a script. This is why global variables are not kept or shared between requests (page 80). Note that there's some minimal setup and cleanup involved in executing the code anew. It's very unlikely that this overhead would meaningfully affect performance, especially with caching in place, but you can avoid it if you use resources, which have a more static life cycle (page 40), instead of /web/dynamic/ pages.

Note that many threads might be running the same code concurrently: make sure you understand the concerns of concurrent programming, as detailed in "Managing State" (page 81).

What if you edit the file? Prudence will automatically pick up your changes and recompile. This happens on the fly, while the application is running. Are you worried that this check would happen for every client request? You can easily control the minimum time interval (page 28) in which Prudence assumes the file is unchanged, and will not check the file for validity.

An updated file would also cause cache entries (page 37) produced by this file to be invalidated.

You can use document.addDependency (page 72) to associate your own life to that of another file. If a dependency is updated, it will force your file to be reloaded.

Caching

You've chosen to generate HTML dynamically because you want requests to be able to have different results. However, sometimes you do not expect results, or parts of the results, to change very often. For example, a home web page might display the local temperature, but it would probably be good enough to update it every half hour, instead of per every request.

Depending on what your scriptlets are doing, dynamically generating a web page can be very costly, and could be a performance and scalability bottleneck for your application under heavy load. You don't want to waste resources to regenerate results that have not changed.

The solution is to cache results. Sometimes caching for even tiny time intervals can make a huge different in the ability of your application to handle load. For example, let's say that in order to fetch the current user count for our home page we need to query a database on the network. Without caching, every client request would result in a service query. Let's say our web page gets 100 hits per second. Caching our home page for a mere 5 seconds would throttle our database queries down to 1 every 5 seconds, vs. 500 every 5 seconds without caching. And users would get a current user count to 5 seconds old. It's a tolerable inaccuracy that generates enormous savings.

This was a trivial example. Truly scalable software requires smart caching everywhere, from the level of file and database access all the way to the generated results. For a full treatment of the topic, see "Scaling Tips" (page 95).

Every /web/dynamic/ and /fragment/ document has its own caching interval, document.cacheDuration (page 68), which you can access and change in scriptlets. For example, here's the home page mentioned above:

```
<html>
<%
document.cacheDuration = 5 * 1000 // five seconds
var userCount = myDatabase.getCurrentUsers().count()
%>
<body>
<p>This is the home page!</p>
<p><%= userCount %> users are currently logged in.</p>
<p><%& '/temperature/' %></p>
</body>
</html>
```

Here is "/web/fragments/temperature.html":

```
<%
document.cacheDuration = 30 * 60 * 1000 // half an hour
var temperature = weatherServer.fetchCurrentTemperature()
%>
The current temperature is <%= temperature %> degrees.
```

Any file that includes the temperature fragment will use a cached result, up to half an hour old. The home page, however, is regenerated every 5 seconds, so that the user count is up to 5 seconds old. Prudence lets you finely control how caching works.

Note that you can change the cache duration dynamically. For example, you might want to increase it if you see that your machine's CPU load is too high, or you might even want to decrease it during "rush hours" where users expect to see especially up-to-date results.

Cache Keys The default cache key used by Prudence is sensible enough that it would not allow for cache conflicts while producing good results. However, Prudence provides many sophisticated tools for customizing cache keys for maximum efficiency. Start with the document.cacheKeyPattern API (page 68), and then take a look at cache key pattern handlers (page 47).

Caching and Compression For further bandwidth and performance improvements with most clients, Prudence compresses your output by default, by setting conversation.encoding (page 77) to the client's preferred encoding. Modern client processors, even on mobile devices, decompress fast enough that the network ends up being the performance bottleneck. The smaller the response, then, the better.

Because the cache key pattern (page 68) automatically gets the encoding added to it, you will always be using the correctly encoded cache entry.

Compression does require some additional computation overhead. Note that if you are caching the output, it will be stored in the cache in its compressed state, such that subsequent cache retrievals would not involve any additional overhead. Additionally, if an uncompressed version already exists in the cache, it will be compressed, cached and used, for optmal efficiency. Thus, the total overhead would likely be insignificant for cached documents, while the bandwidth savings and performance benefits could be considerable. Caching and compression are good friends!

If there are no gains from compression in your application, you can disable Prudence's automatic use of the client's preferred encoding via the "com.threecrickets.prudence.GeneratedTextResource.negotiateEncoding" application global.

> If you're running Prudence behind a proxy (page 89) that already does compression, and are worried that you're compressing twice, then stop worrying: any decent proxy will know not to compress content that's already compressed.

Conditional Requests

HTTP clients, such as web browsers, store downloaded content in their cache, marking it with a modification date and tag according to HTTP headers in the response. Subsequent requests to the same URL will be "conditional," meaning that the client will tell the server what the latest modification date it has. If the server does not have new data, then it returns an empty response with the 304 "not modified" HTTP status, letting the client know that it is safe to use its cached version. This saves both bandwidth and processing resources on the server.

If you're using document.cacheDuration (page 68), then conditional requests are already in effect for you by default. Prudence automatically sets the modification dates accordingly. You can disable this feature via the dynamicWebClientCachingMode setting (page 29).

Client Caching Though clients rely on a local cache for conditional requests, you can provide them with additional caching directives. You can enable this via the the dynamicWebClientCachingMode setting (page 29).

Explicitly setting cache directives has an important side effect: most clients will *not* send conditional HTTP requests for the cached data until the cache entry expires. This allows you to save bandwidth and improves client performance, but at the expense of not being able to update the client upon changes. Use with care.

For even more control of these headers, consider using a resource (page 45) instead.

Handling HTML Forms

Use of HTML's <form> tag is going out of fashion, as more powerful JavaScript-based approaches, such as "AJAX," are used instead to send data from the client to the server and to receive a response. Prudence has excellent support for "AJAX" via resources (page 40). Nevertheless, Prudence also fully supports HTML forms, via the conversation.form API (page 75).

Let's start with a simple form:

```
<html>
<body>
<form name="contact" method="POST">
        <p>Your message to us:</p>
        <p><textarea name="content" cols="50" rows="15"></textarea></p>
        <p>An email address for us to send you a reply:</p>
        <p><input name="email" type="text"></input></p>
        <p><button type="submit">Send</button></p>
</form>
</body>
</html>
```

As it stands, the above form will simply render again when the client clicks "send." Let's add a scriptlet to support the "POST" case:

```
<html>
<body>
<% if(conversation.request.method == 'POST') { %>
Thank you for your message! You will be getting an answer to <%= conversation.form.
    get('email') %> soon.
<% } else { %>
<form name="contact" method="POST">
        <p>Your message to us:</p>
        <p><textarea name="content" cols="50" rows="15"></textarea></p>
        <p>An email address for us to send you a reply:</p>
        <p><input name="email" type="text"></input></p>
        <p><button type="submit">Send</button></p>
</form>
<% } %>
</body>
</html>
```

Pretty simple!

Routing GET and POST You might want to separate the form handler above into two fragments, one for displaying the form and the other for processing it. The main form could then include either fragment according to the value of conversation.request.method.

However, you can also use Prudence's capturing of URI patterns (page 63) to automatically use either file without a main file to switch between the two. Here's an example application "routing.js":

```
document.execute('/defaults/application/routing/')
router.capture('/contact/', '/contact/{m}/')
```

We can then have two files, "contact/GET.html" and "contact/PUT.html" to handle each method for the same "contact/" URL.

Handling POST with a Resource A hybrid solution is to direct the form to a different URL, where you use a resource (page 40) instead of a scriptlet to handle POST. Our example form tag would thus be:

```
<form name="contact" method="POST" action="/send/">
```

And then, under "/resources/send.js":

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

function handlePost(conversation) {
        var email = conversation.form.get('email')
        return '<html><body>' +
                'Thank you for your message! You will be getting an answer to ' +
                   email + ' soon.' +
                '</body></html>'
}
```

Resources

In Prudence, "resources" are encapsulated operational units similar to "objects" in object-oriented architectures. Resources, however, have a stricter structure than objects and define a messaging system more suitable for distributed environments such as the Internet and the World Wide Web. The idea of a "resource-oriented architecture," often called "REST," is covered in depth in "The Case for REST" (page 5).

Files

All files under /resources/ are assumed to be source code files, and are mapped to URLs using the directory and file path, but without the filename extension. For example, the file "/resources/thread/list.js" would map to a URL like "http://mydomain.org/forum/thread/list/". You can also "capture" (page 56) many URLs to be handled by a single file.

The filename's extension will be used to determine the language of the source code.

The above is a very brief overview; see routing (page 53) for an in-depth explanation of how files become URLs.

Life

Files are only compiled once, when they are first requested by a client. The compiled code is kept in a cache, under your Prudence /cache/ directory. From then on, each request is handled by the compiled version, which is updated only if the original file changes. This cache is maintained even if you restart Prudence.

The file itself is executed only once, like a script, upon the first request. If it executes successfully, the remaining state, which contains all the entry points and global variables, is kept in memory. From then on

the entry points are invoked in that state. The file will be executed again only if it changes. Note that this runtime state is *not* cached to disk, like the compiled code, so that restarting Prudence will mean that all your resources will have to be executed before their entry points can be invoked. For this reason, global variables are not kept or shared between such file updates (page 80).

This, then, is the difference between "defrosting" and "preheating" for resources: the former involves making sure that code is compiled and cached, and the latter actually executes the code, and additionally invokes the handleInit (page 43) and handleGet (page 43) entry points.

Note that many threads might be invoking entry points concurrently: make sure you understand the concerns of concurrent programming, as detailed in "Managing State" (page 81).

What if you edit the file? Prudence will automatically pick up your changes and recompile. This happens on the fly, while the application is running. Are you worried that this check would happen for every client request? You can easily control the minimum time interval (page 28) in which Prudence assumes the file is unchanged, and will not check the file for validity.

You can use document.addDependency (page 72) to associate your own life to that of another file. If a dependency is updated, it will force your file to be reloaded.

Source Code

Global Source Code

As is explained in resource "life" (page 40), the first time your source code is loaded, at the time of the first user request, it is executed from start to finish. That means that any code outside of the entry points is run immediately.

Global source code can thus be a good place to initialize various services you need for handling requests. Just be very careful about two things:

- Global variables are not kept or shared between file updates (page 80), which cause the file to be reloaded, recompiled, and thus re-executed. Use application.globals (page 65) instead.
- The re-execution mentioned above can cause your services to be re-initialized. This can be very undesirable, because the service previously initialized may not have been released. It can even lead to errors. It's thus a good idea to first check if the service has already been initialized. Storing references in application.globals (page 65) is a great way to keep track of this.

Consider using the handleInit entry point (page 43) instead of global source code for service initialization. Because handleInit is called for every request, you can make sure to check there if the services you need are up or not.

You may also want to run maintenance tasks (page 49) occasionally to check for and clean up unused services.

API

Beyond what's available to your programming language, resources have access to Prudence's elaborate, powerful set of services via its API (page 65). Note that the "conversation" service (page 74) is provided to entry points as an argument, while the other services are global variables.

Code Libraries

Code files you put in your application's /libraries/ subdirectory can be accesses using your language's code inclusion mechanism. Here are examples in various languages. Let's assume that there is a file named "/libraries/util/data.*", which defines a function called getData().

- Python:

```
from util.data import getData
data = getData()
```

- Ruby:

```
require 'util/data.rb'
$data = get_data
```

- Clojure:

```
(use 'util.data)
(def data (get-data))
```

- PHP:

```
require 'util/data.php';
$data = get_data();
```

- JavaScript and Groovy do not have a built-in code inclusion mechanism, but they can use Prudence's document.execute API (page 71):

```
document.execute('/util/data/')
data = getData()
```

Entry Points

What Are Entry Points?

Each file is a regular source code file, but a few special "entry points" are used by Prudence to connect user requests to your resource. All entry points except one (handleInit) are optional. We'll document each entry point below, but first, here's what an "entry point" means for each flavor of Prudence:

- Python uses global functions:

```
def handle_get(conversation):
        return 'Hello, world!'
```

- Ruby uses global methods:

```
def handle_get conversation
        return 'Hello, world!'
end
```

- Clojure uses functions in the current namespace:

```
(defn handle-get [conversation]
        'Hello, World!')
```

- JavaScript uses global functions:

```
function handleGet(conversation) {
        return 'Hello, world!'
}
```

- PHP uses global functions:

```
function handle_get($conversation) {
        return 'Hello, World!';
}
```

- Groovy uses closures tied to global variables (there are no global methods in Groovy):

```
handleGet = { conversation ->
        return 'Hello, World!'
}
```

Note that the name of the entry point follows typical conventions for each language. In this documentation, we'll use "camel case" (handleGet), but be sure to replace the names following the examples above.

All entry points accept the "conversation" service (page 74) as the sole argument. The other services ("application", "executable", etc.) are available as global variables.

handleInit

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

The main work is to initialize supported media types via conversation.addMediaType (page 79), in order of preference. For example:

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

Prudence handles content negotiation automatically, choosing the best media type according to list of acceptable and preferred formats sent by the client and this list.

But, Why? You might wonder why we add these supported media types dynamically for each request via a call to handleInit, since they are usually always the same for a resource. The reason is that sometimes they may not be the same. In handleInit, you can check for various conditions of the conversation, or even external to the conversation, to decide which media types to support. For example, you might not want to support XHTML for old browsers, but you'd want it at the top of the list for new browsers. Or, you might not be able to support PDF in case an external service is down. In which case, you won't want it on the list at all, in which case content negotiation would choose a different format that the client supports, such as DVI.

So, this gives you a lot flexibility, at no real expense: adding media types per request is extremely lightweight.

handleGet

Handles HTTP "GET" requests.

In a conventional resource-oriented architecture (page 5), clients will not be expecting the resource to be altered in any way by a GET operation.

What you'll usually do here is construct a representation of the resource, possibly according to specific parameters of the request, and return this representation to the client. See the "conversation" API documentation (page 74) for a complete reference. Note especially that if you've captured URI segments (page 56), they'll be available in conversation.locals (page 79).

The following return types are supported:

- Numbers: Returns the number as an HTTP status code to the client, with no other content. Usually used for errors. For example, 404 means "not found." Note that error capturing (page 58) can let you take over and return an appropriate error page to the client.
- Arrays of bytes: Used for returning binary representations. Note that some languages (JavaScript, for example) have their own implementations of arrays, which are not exactly compatible with JVM arrays. In such cases, you have to make sure to return JVM arrays. Internally, Prudence represents these values with a ByteArrayRepresentation.
- Representation instances: You can construct and return a Restlet representation directly.
- Other return values: If the conversation.mediaType (page 76) is "application/java" the value will be wrapped in an ObjectRepresentation instance. Otherwise, it will be converted into a string if it isn't a string already, and returned to the client as a textual representation.

Beyond the return value, you can affect the response sent to the client by the response attributes in the "conversation" service (page 74). In particular, the conversation.modificationDate (page 78) and conversation.tag (page 78) can be used to affect conditional HTTP requests (page 45). For these, you may also consider implementing the handleGetInfo entry point (page 45) for more scalable handling of conditional requests.

handlePost

Handles HTTP "POST" requests.

In a conventional resource-oriented architecture (page 5), POST is the "update" operation (well, not exactly: see note below). Clients will expect the resource to already exist for the POST operation to succeed. That is, a call to GET before the POST may succeed. Clients expect you to return a modified representation, in the selected media type, if the POST succeeded. Subsequent GET operations would then return the same modified representation. A failed POST should not alter the resource.

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

What you'll usually do here is fetch the data, and alter it according to data sent by the client. See the "conversation" API documentation (page 74) for a complete reference. Note especially that if you've captured URI segments (page 56), they'll be available in conversation.locals (page 79).

See handleGet (page 43) for supported return types. In fact, you may want handlePost to share the same code path as handleGet for creating the representation.

> POST is the only HTTP operation that is not "idempotent," which means that multiple *identical* POST operations on a resource *may* yield different results from a single POST operation. This is why web browsers warn you if you try to refresh a web page that is the result of a POST operation. As such, POST is the correct operation to use for manipulations of a resource that *cannot be repeated*. See this blog post by John Calcote for an in-depth explanation.

handlePut

Handles HTTP "PUT" requests.

In a conventional resource-oriented architecture (page 5), PUT is the "create" operation (well, not exactly: see note below). Clients will expect whatever current data exists in the resource to be discarded, and for you to return a representation of the new resource in the selected media type. A failed PUT should not alter the resource.

Note that the entity sent by the client does not have to be identical in format or content to what you return. In fact, it's likely that you will return more information to the client than what was sent.

What you'll usually do here is parse and store the data sent by the client. See the "conversation" API documentation (page 74) for a complete reference. Note especially that if you've captured URI segments (page 56), they'll be available in conversation.locals (page 79).

See handleGet (page 43) for supported return types. In fact, you may want handlePut to share the same code path as handleGet for creating the representation.

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

handleDelete

Handles HTTP "DELETE" requests.

In a conventional resource-oriented architecture (page 5), clients expect subsequent GET operations to fail with a "not found" (404) code. A DELETE should fail with 404 if the resource is not already there; it should *not* silently succeed. A failed DELETE should not alter the resource.

What you'll usually do here is make sure the identified resource exists, and if it does, remove or mark it somehow as deleted. See the "conversation" API documentation (page 74) for a complete reference. Note especially that if you've captured URI segments (page 56), they'll be available in conversation.locals (page 79).

The following return types are supported:

- Numbers: Returns the number as an HTTP status code to the client, with no other content. Usually used for errors. For example, 404 means "not found." Note that error capturing (page 58) can let you take over and return an appropriate error page to the client.
- Null: Signifies success.

Note: It's good practice to always explicitly return null in handleDelete. Some languages return null if no explicit return statement is used. Others, however, return the value of the last executed operation, which could be a number, which would in turn become an HTTP status code for the client. This can lead to some very bizarre bugs, as clients receive apparently random status codes!

handleGetInfo

Handles HTTP "GET" requests *before* handleGet (page 43).

This entry point, if it exists, is called before handleGet in order to provide Prudence with information required for conditional HTTP requests (page 45). Only if conditions are not met—for example if our resource is newer than the version the client has cached, or the tag has changed—does Prudence continue to handleGet. Using handleGetInfo can thus improve on the gains of conditional requests: not only are you saving bandwidth, but you are also avoiding a potentially costly handleGet call.

The following return types are supported:

- Numbers: Considered as Unix timestamps, and converted into the modification date. See conversation.modificationDate (page 78).
- JVM Date instances: The modification date. Refer to the Java API documentation for details.
- Strings: Considered as HTTP tags. See conversation.tag (page 78).
- Tag instances: You can construct and return your own Restlet tag.
- RepresentationInfo instances: You can construct and return your own Restlet representation info.

Note that even if though you can only set either the modification date or the tag by the return value, you can set the other one using conversation.modificationDate (page 78) and conversation.tag (page 78).

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

In the real world... You might be tempted to go ahead and provide a handleGetInfo entry point for every resource you create. This is not necessarily a good practice, for three reasons:

1. It could be that you don't need this optimization. Make sure, first, that you've actually identified a problem with performance or scalability, and that you've traced it to handleGet on this resource.
2. It could be that you won't gain anything from this optimization. Caches and other optimizations along the route between your data and your client might already be doing a great job at keeping handleGet as efficient as it could be. If not, improving them could offer far greater benefits overall than a handleGetInfo.
3. It could be that you'll even hurt your scalability! The reason is that an efficient handleGetInfo implementation would need some mechanism in place to track of data modification, and this mechanism can introduce overhead into your system that causes it to scale worse than without your handleGetInfo.

See "Scaling Tips" (page 93) for a thorough discussion of the problem of scalability.

Conditional Requests

HTTP clients, such as web browsers, store downloaded content in their local cache, marking it with a modification date and tag according to HTTP headers in the response. Subsequent requests to the same URL will be "conditional," meaning that the client will tell the server what the latest modification date it has. If the server does not have new data, then it returns an empty response with the 304 "not modified" HTTP status, letting the client know that it is safe to use its cached version. This saves both bandwidth and processing resources on the server.

To support conditional requests, you have to explicitly set at least one of conversation.modificationDate (page 78) and conversation.tag (page 78). If you implement handleGetInfo (page 45), you should be returning one of these values instead.

Note that these attributes are ignored in case you are constructing and returning your own Representation instance.

Client Caching Though clients rely on a local cache for conditional requests, you can provide them with additional caching directives. In Prudence, you can control the expiration of the client's cached entry with conversation.maxAge (page 78) or conversation.expirationDate (page 79).

Explicitly setting cache directives has an important side effect: most clients will *not* send conditional HTTP requests for the cached data until the cache entry expires. This allows you to save bandwidth and improves client performance, but at the expense of not being able to update the client upon changes. Use with care.

Resources As API

A "resource-oriented architecture" is purposely more minimal than object-oriented, transactional architectures. As discussed in "The Case for REST" (page 5), there are many far-reaching advantages to this minimalism. While it's possible for you to maintain multiple interfaces to your service, it might make more sense to standardize around REST principles throughout your application. A single, uniform API is more maintainable, and less prone to bugs.

Prudence lets you access your resources internally, without having to go through HTTP, using document.internal (page 72). You can furthermore avoid special representations and pass JVM objects through directly by setting conversation.mediaType (page 76) to "application/java".

For example, here's "/resources/testme.js" that nicely supports both external and internal clients:

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

function handleGet(conversation) {
        var data = getData()
        if(conversation.mediaTypeName == 'application/java') {
                return data
        }
        else {
                return JSON.to(data)
        }
}

function getData() {
        return {name: 'test', description: 'This is some test data'}
}
```

We can then access the resource directly, without any JSON encoding/decoding:

```
var data = document.internal('/testme/', 'application/java').get().object
print(data.description)

// Just for comparison, here's how we would to it using JSON encoding/decoding
var data = eval('(' + document.internal('/testme/', 'application/json').get().text
   + ')')
print(data.description)
```

> Note: The "application/java" media type can be used externally (page 72), too. In such a case, the object would be serialized/unserialized over HTTP. This does mean that your data class has to properly support JVM serialization, via the Serializable or Externalizable interfaces.

Handlers

Prudence supports a few kinds of "handlers," special purpose code for handling specified mechanisms.

Comparison with Resources

Creating handlers is similar to creating a resource (page 40), except that they are much simpler.

All files under /handlers/ are assumed to be source code files. They are mapped simply to internal URIs, where a trailing slash is optional. Handlers do not have external URIs like resources (page 40). Though there are different kinds of handlers, they are all put in /handlers/, the different being only in which entry points Prudence expects to find in them. It's up to you to use the correct kind of handler for the correct purpose.

The life cycle of handlers is similar to that of resources (page 40), except that no defrosting or preheating is supported.

Entry points (page 42) work a bit differently:

- Like in resources, the "conversation" service (page 74) is provided as the first argument, but only a subset of the API is supported.
- Some entry points support additional arguments.

Cache Key Pattern Handlers

These handlers are used to set conversation.locals (page 79) that are then used for filling in custom document.cacheKeyPattern (page 68) variables. If, and only if, the variable is used in a cache key pattern, then the handler is called. It is expected that the handler would then fill the appropriate conversation.local (page 79) with a value, which is then used to cast the pattern into a value.

Custom variables for sophisticated, powerful caching strategies. For example, by including a user identifier in cache key patterns, you can allow for caching pages or fragments per user.

Global Handlers

One way to use them is to set them globally, such they will be usable in cache key patterns of any document. To do so, in your application's "routing.*" use the cacheKeyPatternHandlers service, which is a map of variable names (without the curly brackets) to handler names. For example, let's have "{un}" and "{ug}" be handled by "/handlers/userinfo.js". Here the "routing.js":

```
cacheKeyPatternHandlers.put('un', 'userinfo/')
cacheKeyPatternHandlers.put('ug', 'userinfo/')
```

Local Handlers

You can also set cache key pattern handlers for specific documents, using document.cacheKeyPatternHandlers (page 68). Local settings will override global settings. For example, here's a scriptlet from a document in /web/dynamic/ or /web/fragments/:

```
document.cacheKeyPatternHandlers.put('un', 'userinfo/')
document.cacheKeyPatternHandlers.put('ug', 'userinfo/')
document.cacheKeyPattern = '{an}|{ptb}|{dn}|{un}'
```

Entry Points

handleCacheKeyPattern The second argument of this entry point, after the "conversation" service (page 74), is a list of the variable names that appear in the document.cacheKeyPattern (page 68), and which you have registered to handle by calling document.cacheKeyPatternHandlers (page 68). The variable names here are not enclosed in curly brackets.

A good technique is to iterate this list and fill in an appropriate conversation.local (page 79) for each variable. He is a "/handlers/userinfo.js" to match the example above:

```
function handleCacheKeyPattern(conversation, variables) {
        var cookie = conversation.getCookie('session')
        var user = getUserFromCookie(user)
        for(var i in variables) {
                var variable = String(variables[i])
                switch(variable) {
                        case 'un':
                                conversation.locals.put('un', user.name)
                                break
                        case 'ug':
                                conversation.locals.put('ug', user.group)
                                break
                }
        }
}

function getUserFromCookie(cookie) {
        ...
        return user
}
```

Note that no return value is expected.

Filters

Filters sit somewhere along the conversation route, and are able to affect the progress of the conversation. See the section on filtering in routing (page 61) for a complete discussion.

Entry Points

Both entry points are optional, though it's quite useless to have a filter with neither.

handleBefore This is called *before* the request is handled by the next handler along the route.

There is likely no response set at this point, so you only have access to request attributes.

Three literal return values are supported, as either a string or a number:

- "continue" or 0: Continue to the next handler
- "skip" or 1: Skip the next handler and continue to our handleAfter entry point (page 48)
- "stop" or 2: Stop our handling altogether (note that previous handlers along the route may still process the conversation before it returns to the client)

Otherwise, you may return an internal URI, which must begin with a "/". This causes the filter to capture to that URI (an internal redirect). This is a powerful feature allowing you to implement complex capturing via filters. For more information and an example, see dynamic capturing in the routing chapter (page 58).

handleAfter This is called *after* the request is handled by the next handler along the route, unless "stop" was returned from our handleBefore (page 48).

At this point, a response has usually been set, unless we returned "skip" from in handleBefore or another filter down the route erased it. You may manipulate the response or even replace it entirely.

Note that handleAfter is called even if the next handler encounters an error. You can test for such an error with conversation.statusCode (page 76). Indeed, one use case for filters is to clean up and release hanging services after errors occur.

No return value is expected.

Tasks

Programming multi-threaded and distributed applications is usually very difficult, but Prudence makes a lot of the work easier via support for tasks, which run in the "background" outside of the client thread pool. Tasks are an important port of your scaling strategy as they allow your application to do work without holding up client threads. For a full discussion, see "Scaling Tips" (page 93).

Tasks are straightforward scripts placed in the application's /tasks/ subdirectory. They are straightforward in that, upon execution, the code is simply run as is. There is no particular entry point as with /resources/ or /handlers/. They can be run on-demand or scheduled to run at specific times and locations.

Tasks are part of an application, and have access to the same "application", "document" and "executable" API services as other application code. Likewise, you can access any code in /libraries/ via document.execute (page 71) or via your language's usual inclusion facility. Unlike /resources/ and /web/dynamic/ code, /tasks/ code has no "conversation" API service (page 74).

The modern JVM, running on modern operating systems and multi-core hardware, will do a very good job at multi-threading, so be confidence in using /tasks/ heavily.

Prudence provides two facilities for scheduling tasks:

On Demand

Use application.task (page 66) to spawn or schedule tasks on demand from your /resources/ or /web/dynamic/ code, or application.distributedTask (page 67) to spawn tasks in your Hazelcast cluster.

In some cases, you may want to atomically update a boolean in application.globals (page 65) or application.distributedGlobals (page 66) to flag that the task has been scheduled, so that you won't schedule it more than once.

At Startup

Startup Task

Prudence will automatically spawn the task called "/startup/" upon startup. Note that it will run after all the other startup tasks (defrosting, preheating) finish, in order to make sure your application is "hot" first. If you need multiple tasks running—usually repeating tasks, such as those that do general maintenance, monitor resource availability, send out email reminders, etc.—you can start them from your startup task using application.task (page 66).

For example, here is an application's "/tasks/startup.js" file:

```
application.logger.info('Scheduling monitor task to repeat every second')
application.task('/monitor/', null, 1000, 1000, false)
```

Note that this initial run of the "/startup/" task will send the string "initial" for the document.context (page 70). This is so you can distinguish between this initial run and possible subequent runs that you spawn yourself.

Custom Startup Tasks

You can also use application.task (page 66) to schedule general tasks in any of the application's configuration scripts (page 25).

Here is an example "default.js" (note the use of "applicationService" instead of "application"):

```
document.execute('/defaults/application/')

applicationService.task('/poll/', {name: 'mypoll', maxDuration: 5000}, 1000, 1000, false)
```

crontab

This facility lets you schedule tasks to run at specific (Gregorian) calendrical times. It is similar to calling application.task (page 66) in the application's configuration scripts, but allows for more succint, calendrical repetition patterns.

To use this facility, place a file with the name "crontab" in your application's base subdirectory. Its format is purposely very similar to that of the crontab configuration file found in many Unix-like operating system: each line starts with scheduling pattern and ends with the task name.

Optionally, you may add more text after the task name and whitespace: anything there is grabbed as a single string and sent as the context to the task, which can be accessed there as document.context (page 70). Because crontab is a text file, only textual contexts may be sent, but you can use JSON, XML or other encodings to create complex contexts.

The scheduling pattern is a series of five whitespace-separated settings. Any of these settings can be "*", signifying that any value would match. Use a slash to match only numbers that divide equally by the number after the slash (can also be used on "*"). Ranges (inclusive) are possible, separated by hyphens. Multiple settings are possible, separated by commas. Multiple patterns are possible, separated by pipes.

1. Minutes of the hour, 0-59.
2. Hour of the day, 0-23.
3. Day of the month, 1-31. The special setting "L" signifies the last day of the month, which varies per month and year.
4. Month of the year, 1-12. Three-letter English month names may be used instead of numbers: "jan", "feb", "mar", etc.
5. Day of the week, 0-6. Three-letter English day names may be used instead of numbers: "sun", "mon", "tue", etc.

Example patterns:

- Every minute: * * * * *
- 11:59pm every Tuesday and Friday: 59 23 * * tue,fri
- Every 5 minutes in the morning, between 5 to 8am, otherwise every 30 minutes: */5 5-7 * * *|*/30 0-4,8-23 * * *
- The same as above, but with one added to all minutes of the hour: 1,6,11,16,21,26,31,36,41,46,51,56 5-7 * * *|1,31 0-4,8-23 * * *

Notes:

1. "crontab" will be checked for changes and parsed once per minute. This means that you can edit this file and have your task scheduling change on the fly without restarting Prudence.
2. You can schedule the same task on multiple lines, which is *not* equivalent to using the pipe: multiple lines means that multiple task instances might be spawned simultaneously if matched on more than one line. Contrarily, using the pipe counts as a single match.
3. The scheduler does not check to see if a task finished running before swapning a new instance of it, so that even if a task is not done yet, but it's time for it to be spawned again, you'll have multiple instances of the task running at the same time. If this is problematic, consider using application.task (page 66) in your confuguration scripts instead, with "false" as the last argument.

You can also set up a special crontab to run arbitrary Java static methods and even non-JVM system processes. Internally, Prudence uses the flexible cron4j library. See its documentation for more information. Here's an example application's "default.js" configuration script to set up a "crontab-system" file:

```
document.execute('/defaults/application/')
scheduler.scheduleFile(new File(applicationBasePath + '/crontab-system'))
```

Static Web

Prudence attaches your application's /web/static/ subdirectory to URLs, making files there available via HTTP GET. In other words, /static/web/ is a "web server." It's a convenient place for storing immutable resources that your clients will need to run your application: images, styles, scripts, etc.

You're likely, though, wondering if /web/static/ is merely a convenience, and if you'd be better off using Apache or other dedicated web servers instead of Prudence to serve your the static files.

Our recommendation is to take that route only if those web servers offer important features that you won't find in Prudence. Remember that Prudence has many powerful features, including URL rewriting and redirection (page 53), smart compression (page 30), and that Prudence's non-blocking I/O does a great job at scaling. You will likely not see significantly better performance or scalability by replacing /web/static/ with standard web servers.

Mapping Files to MIME Types

Files are sent to clients with a MIME type determined according to the filename's extension.

See application configuration (page 27) for instructions on how to configure this mapping.

Replacing Jetty

Jetty can be replaced with Netty or Grizzly, non-blocking I/O HTTP servers with different performance characteristics. See the Restlet documentation for more information.

CacheControlFilter

Prudence handles conditional HTTP automatically for /web/static/: when clients request a new file, they tell Prudence which version they have cached, and Prudence makes sure not to send them the file if they already have the most recent version. Unfortunately, this still does require a short round-trip HTTP request between client and server.

To avoid even these short requests, Prudence comes with a handy filter (page 61) that can selectively set cache control headers according to file type. By default, the filter sets the max age cache header to the "far future" (ten years) for all files. For most web browsers, and many other clients that support local caching, this effectively causes the files to be cached "forever." Thus, the client would only request the file again if the cache entry is deleted due to the cache being full, or if the cache is manually reset.

This can allow for faster page rendering in web browsers, save you in bandwidth costs, and help you scale. Sounds perfect? Unfortunately, it also affects your ability to make changes to files. Ten years means ten years: the client does not expect the file to change in that time. Since you likely cannot ask your users to reset their browser caches every time you make a change, what you can do instead is rename the file on your end: because clients cache files according to their URLs, and filenames are mapped to URLs, the client would consider it a fresh file. Of course, you would also have to update references to the URL in HTML and CSS.

Another trick to force clients to re-request an updated file is to add query strings to the URL. For example, consider this HTML code:

```
<img src="images/logo.png?_=128799073903458">
```

The query string is ignored by Prudence's static web handler, but since it does signify a new URL, the client will use a separate cache entry for it. This lets you keep your filenames, but does require you to alter URLs.

Since the above techniques require considerable discipline from the application maintainers, the cache control filter is not enabled by default in Prudence. The following is an example on how to enable it in an application's "routing.js", while disabling it for file types that are frequently updated:

```
importClass(
        org.restlet.data.MediaType,
        com.threecrickets.prudence.util.CacheControlFilter)

var staticWebCacheControlFilter = new CacheControlFilter(null)
```

```
// Disable caching for CSS and JavaScript files
staticWebCacheControlFilter.maxAgeForMediaType.put(MediaType.TEXT_CSS, -1)
staticWebCacheControlFilter.maxAgeForMediaType.put(MediaType.APPLICATION_JAVASCRIPT
    , -1)

router.filterBase(staticWebBaseURL, staticWebCacheControlFilter, staticWeb)
```

> You can use this filter for /dynamic/web/, too, but you're better off using its built-in support (page 39).

JavaScriptUnifyMinifyFilter

When writing JavaScript code, you likely want to use a lot of spacing, indentation and comments to keep the code clear and manageable. You would likely also want to divide your code into multiple files. However, if this code is meant to be run on the client, this could mean sending several overly large files to it.

Prudence comes with a handy filter (page 61) that lets you "unify" several JavaScript files into one, and to "minify" it so that superfluous comments and whitespace are removed. Bandwidth savings can be significant.

The filter works by detecting a user request for a file named "all.min.js" in a directory which contains ".js" files. If the file does not exist, it collects all the JavaScript files in the directory, unifies them in alphabetical order, and minifies the result into a file called "all.min.js". The filter picks up your changes on the fly, and will only regenerate "all.min.js" if you've made a change to any of the base files. You can control the minimum time required to pass between these checks via a constructor argument (see example below).

If order of source files within the unified file is important, make sure to name so that an alphabetical sort would put them in the right order.

```
importClass(com.threecrickets.prudence.util.JavaScriptUnifyMinifyFilter)

router.filterBase(
        staticWebBaseURL,
        new JavaScriptUnifyMinifyFilter(null, new File(applicationBasePath +
            staticWebBasePath),
                minimumTimeBetweenValidityChecks), staticWeb)
```

An example of loading the unified-minified file from HTML:

```
<html>
<head>
        <script type="text/javascript" src="script/all.min.js"></script>
</head>
</html>
```

CssUnifyMinifyFilter

This filter does for CSS what the JavaScriptUnifyMinifyFilter (page 52) does for JavaScript. Example:

```
importClass(com.threecrickets.prudence.util.CssUnifyMinifyFilter)

router.filterBase(
        staticWebBaseURL,
        new CssUnifyMinifyFilter(null, new File(applicationBasePath +
            staticWebBasePath),
                minimumTimeBetweenValidityChecks), staticWeb)
```

You can use JavaScriptUnifyMinifyFilter and CssUnifyMinifyFilter together. The order of filtering should not matter.

Routing

In Prudence, "routing" refers to the decision-making process by which an incoming client request reaches its handler on the server. Usually, information in the request itself is used to make the decision, such as the URI, cookies, the client type, capabilities and geolocation. But routing can also take server-side and other circumstances into account. For example, a round-robin load-balancing router might send each incoming request to a different handler in sequence. The decision on which handler to use has nothing to do with the a client's particular request.

A request normally goes through many routers before reaching its handler. Filters along the way can change information in the request, which could also affect routing. Filtering should thus also be thought of as a routing tool.

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

When writing applications for Prudence, you will mostly be interested in application-level routing, which we will cover in-depth below. However, to give you a better understanding of how Prudence routing works, let's follow the journey of a request, starting with routing at the instance level.

Instance Routing

Before a request reaches your application, it is routed by your Prudence instance.

Step 1: Servers

Requests come in from servers. Prudence instances have at the minimum one server, but can have more than one. Each server listens at a particular HTTP port, and multiple servers may in turn be restricted to particular network interfaces on your machine. By default, Prudence has a single server that listens to HTTP requests on port 8080 coming in from all network interfaces.

You can configure your servers in "/instance/servers.*" (page 29).

Step 2: The Component

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

Step 3: Virtual Hosts

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

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

You can configure your virtual hosts in "/instance/hosts.*" (page 23).

Step 4: Applications

Using "hosts" in your application's "settings.*" (page 28), you can configure which virtual hosts your application will be attached to, and the base URI for the application on each virtual host. An application can accept requests from several virtual hosts at once.

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

Note that you can create a "nested" URI scheme for your applications. For example, one application might be attached at the root URI at a certain virtual host, "/", while other applications might be at different URIs

beneath the root, "/wackywiki" and "/wackywiki/support/forum". The root application will not "steal" requests from the other applications, because the request is routed to the right application by the virtual host. The fact that the latter URI is the hierarchical descendent of the former makes no difference to the virtual host router.

A Complete Route

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

Our machine has two network interfaces, one facing the Internet and one facing the intranet, and we have two servers to listen on each. This particular request has come in through the external server. The request reachers the component's router.

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

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

The remaining part of the URI, "/thread/12/" will be further routed inside the forum application, as detailed below.

Application Routing

Step 5: Application Handlers

Prudence applications come with default support for three kinds of handlers: resources (in the /resources/ subdirectory), dynamic web pages (in the /web/dynamic/ subdirectory) and static web resources (in the /web/static/ subdirectory). By default, all three handlers are attached at the root URI, "/", of the application (which may vary per virtual host). However, it is possible to change this in your application's "settings.*", see resourcesBaseURL, dynamicWebBaseURL and staticWebBaseURL (page 28). For example, you may want your resources to be routed under "/rest-interface/".

You may ask, for any given request, how can the application's router know which handler to send it to, if all handlers by default assume the same "/" base URL? The answer is that it doesn't. It tries each handler in sequence, and if one handler cannot handle the request, it falls back to the next one. For example, a "style/main.css" URI will be tried first as a resource. If that resource doesn't exist, it will be tried as a dynamic web page. Finally, it will be sent to the static web handler.

Be careful: this system allows for previous handlers in the sequence to supersede later handlers. For example, if you decide to remove scriptlets from a file named /dynamic/web/about.html and move it to /static/web/about.html, make sure to delete the former, or else the latter will not be reached.

Subdirectories and Filenames As URI Segments

The three application handlers—resources, dynamic web pages and static web resources—are all routed by mapping the filesystem structure to a URI. Each subdirectory path or filename is directly translated into a URI segment.

This is exactly the scheme used by most static web servers, and it has the benefit of using a readily-available, easy-to-use hierarchical structure—the filesystem—as a striaghtforward way for creating URIs.

There's one deficiency to this scheme: by directly mapping filenames, it can allow for "ugly" URIs that include filename extensions. For example, you're probably used to seeing many web sites with URLs that end in ".html", ".php" and ".jsp". While these extensions are meaningful to the site developer, they complicate the URIs and expose internal implementation details to outsiders.

Pretty URIs

To allow for prettier URIs, Prudence does a few things:

Filename Extension Hiding Prudence ignores filename extensions for the purpose of mapping to URIs. For example, the file "/web/dynamic/wiki/table-of-contents.html" will be mapped to the URI "/wiki/table-of-contents/".

Default Files In /web/dynamic/ and /web/static/, if a file "index.html" exists in a subdirectory, it is mapped to the URI for the subdirectory itself. In /resources/, the default file is "default.*".

For example, the file "/web/dynamic/wiki/contributors/index.html" is mapped to the URI "/wiki/contributors/". Note that another way to create the same URI is to use "/web/dynamic/wiki/contributors.html" (the ".html" is hidden by Prudence).

Default files are simply another organizational option for you. In very large applications, it can help keep your files well-organized without having any effect on the URIs that exposed externally. You can even change your scheme as your application evolves.

> The name "index.html" is used by web servers for archaic reasons: it was conceived of as a place where you could list the contents, or "index," the subdirectory. These days, however, we tend to have a more general understanding of URIs. By default, we've followed this convention in Prudence. You can change this with the "com.threecrickets.prudence.GeneratedTextResource.defaultName" application attribute.

Trailing Slash Requirement You'll note that we used "/wiki/contributors/" for the above, rather than "/wiki/contributors" (the difference is the trailing slash). This is because Prudence requires trailing slashes by default: trying to access "/wiki/table-of-contents" would permanently redirect to "/wiki/table-of-contents/".

Prudence requires a trailing slash for two main reasons:

1. To keep the URI space consistent, whether you use subdirectories or filenames to create the URI segments.
2. This is Prudence's way of fighting to the "trailing slash" problem, which plagues the use of relative URIs in HTML and CSS.

You can turn off the trailing slash requirement via the "com.threecrickets.prudence.GeneratedTextResource.trailingSlashI
and "com.threecrickets.prudence.DelegatedResource.trailingSlashRequired" application.globals.

> In Prudence 1.1, the filename extension hiding and trailing slash requirement work only for /resources/ and /web/dynamic/ URLs. /web/static/ URLs still map full filenames. A future version of Prudence may extend these features to all resources.

If you'd like a specific non-trailing-slash URL to automatically redirect to the trailing-slash equivalent, use the application's "urlAddTrailingSlash" setting (page 31).

Overriding Pretty URIs You can override all the automatic prettifying techniques mentioned above by explicitly capturing URIs (page 56). For example:

```
router.capture('/sitemap.xml', '/sitemap/')
```

Filename Extensions

Though Prudence hides the filename extensions from the URIs, they do have two important functions:

Filename Extensions and MIME Types Filename extensions define the default media type for "GET" requests to pages in /web/dynamic/ and /web/static/. For example, a ".xml" file will be mapped to the "application/xml" MIME type.

Every application has its own filename extension mapping table (page 27).

Additionally, you can explicitly set the MIME type in a code via conversation.mediaTypeName (page 76), overriding the default.

Filename Extensions and Programming Languages In /resources/, and for configuration scripts, the filename extension tells Prudence the programming language of the source code. Prudence supports ".py", ".rb", ".js", ".php", ".clj" and ".php" files.

Filename Extensions Preference In /resources/, the Prudence flavor you are using will determine which file to use in the case of ambiguity. For example, the Groovy flavor will prefer "process.gv" file even if "process.js" is present. This allows you to write and deploy applications that can run in multiple Prudence flavors. The same is true for configuration scripts.

However, in /web/dynamic/ there is no explicit preference, and behavior is undefined if you have more than one file with the same name but different extension in the same subdirectory. For example, contents.html and contents.xml.

Custom Routing

The straightforward URI routing based on directories and filenames is useful, but you'll likely need other kinds of routing, too. That's what your application's "routing.*" is for (page 27).

> Actually, all of this API can also be called at runtime, allowing you to attach and change routes dynamically while the Prudence instance and its applications are up and running.

Hiding

The following will make sure that the given URI always returns a 404, even if the page or resource exists. For example, in "routing.*" (JavaScript flavor):

```
router.hide('/administration/')
```

The resource will still be available internally, via document.internal (page 72), and for capturing (page 56).

Static Capturing

Prudence lets you "capture" abitrary URI patterns into any internal URI you wish, whether they're implemented as /resources/, /web/dynamic/ or /web/static/. This lets you turn a single URI implementation to into a multitude of URIs, vastly expanding your URI-space.

Static capturing is handled in your application's "routing.*", via the router.capture API. The first argument is the pattern your want to capture, the second is the interal URI to which the request will be redirected. You use curly-bracket-delimited tags in the URI to parse URI segments and store them in conversation.locals (page). For example, in "routing.*" (JavaScript flavor):

```
// Implement defaults
document.execute('/defaults/application/routing/')

router.capture('/forum/help/{topic}/', '/forum/help/')
```

And then, in "/web/dynamic/forum/help.html":

```
<html>
<body>
<p>You are viewing help topic <%= conversation.locals.get('topic') %>.</p>
</body>
</html>
```

Capturing might look like redirection, but in fact it's an *internal* redirection, similar to how the document.internal API works (page 72). The client remains entirely ignorant as to what internal URIs you might be using.

It's important to understand this distinction: the client might be seeing an entirely different URI than your internal one. Thus, if you're using HTML and CSS, you need to make sure that your relative references reach the right place. This is easy in Prudence with the conversation.pathToBase API (page 74), which will use the captive client URI. For example:

```
<html>
<body>
<p>You are viewing help topic <%= conversation.locals.get('topic') %>.</p>
<img src="<%= conversation.pathToBase %>/images/help.png"/>
```

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

Capturing in this manner is often used in conjunction with URI hiding (page 56). By hiding "/forum/help/", users would be able to access "/forum/help/faq/" via HTTP, but not "/forum/help/". This lets you effectively control the exposed URI space. In fact, capturing and hiding is common enough that it has its own shortcut API:

```
router.captureAndHide('/forum/help/{topic}/', '/forum/help/')

The above is identical to:

router.capture('/forum/help/{topic}/', '/forum/help/')
router.hide('/forum/help/')
```

Another feature of internal redirection is that your code can check for internal access and enforce it. As an example, let's implement URI hiding via a scriptlet:

```
<%
if(!conversation.internal) {
        conversation.statusCode = 404 // resource not found
}
%>
<html>
<body>
<p>You are viewing help topic <%= conversation.locals.get('topic') %>.</p>
</body>
</html>
```

Prudence also lets you capture into a different application in the instance. Refer to applications using their subdirectory name. For example:

```
// Implement defaults
document.execute('/defaults/application/routing/')

router.captureOther('/forum/help/{topic}/', 'wackyhelp', '/help/')
```

In addition to parsing URI segments, you can use predefined curly-bracket-delimited variables to route based on attributes of the request, via URI patterns (page 63). For example, we can route with the "{m}" variable (HTTP method) to elegantly handle HTTP forms:

```
router.capture('/message/', '/message/{m}/')
```

We can then have a dynamic web page named "/message/GET.html":

```
<html>
<body>
<form method="post">
        Enter your message:<br/>
        <textarea name="message"></textarea><br/>
        <input type="submit" />
</form>
</body>
</html>
```

and one named "/message/POST.html":

```
<html>
<body>
        <%= conversation.form.get('message') %>
</body>
</html>
```

Another example of capturing with URI patterns:

```
router.capture('/data/client/', '/data/client/{emt}/')
```

Because "{emt}" is the MIME type of data sent by the client, and the slash used in MIME types is compatible with the slash used in filesystem path separation, we can have files such as these:

- /resources/data/client/text/html/default.js
- /resources/data/client/application/json/default.js
- /resources/data/client/default.js (this will be used if no client data is sent, for example in an HTTP GET)

Of course, you might prefer to check the MIME type in code your code via conversation.mediaTypeName (page 76), but the above is a good example of how powerful routing can be with URI patterns.

Dynamic Capturing

Static capturing using patterns is very powerful, but sometimes you need more than patterns: you need to check attributes of the client's request, such as the contents of cookies, or do a database lookup to find out the destination internal URI. To allow for this, Prudence also lets you capture by returning a URI in a filter (page 48).

Here's an example of a filter that redirects to a wiki page only if that page exists in the database. First, let's install the filter (page 61) in our "routing.js":

```
// Implement defaults
document.execute('/defaults/application/routing/')

router.filter(dynamicWebBaseURL, '/page-filter/', dynamicWeb)
```

Here is "/handlers/page-filter.js":

```
function handleBefore(conversation) {
	// Let's look for the page name in remaining part of the URI
	var pageName = String(conversation.reference.getRemainingPart(true, false))

	// Fetch the page from the database
	var page = getPageFromDatabase(pageName)
	if (page) {
		// Cache the fetched page in conversation.locals
		conversation.locals.put('page', page)

		// Capture!
		return '/page/'
	}

	// This was not a page, so continue as usual
	return 'continue'
}
```

You would then have a "/web/dynamic/page.html" that can get the pre-fetched page from conversation.locals.get('page') and renders it.

Capturing Errors

A special case of capturing is for errors. "Errors" can be set explicitly by you: for example, we can set conversation.statusCode (page 76) to 404, as we did above. However, a 500 error occurs automatically for uncaught exceptions in your code.

If you have debug mode enabled, the user would see the special debug page for 500 errors; see debugging (page 83). On a production site, you may instead prefer to capture the 500 error and provide a friendlier page. (You'd also want to test carefully and make sure your code never throws exceptions...)

You can capture errors both at the application level or at the instance level, which the former taking precedence. It may be a good idea to always capture at the instance level, in case applications don't capture for their own custom error pages.

Examples from "/instance/routing.*" (JavaScript flavor):

```
// Implement defaults
document.execute('/defaults/instance/routing/')

// 404 errors
component.statusService.capture(404, 'wackyhelp', '/help/main/',
        component.context)

// 500 errors
component.statusService.captureHostRoot(500, 'wackywiki', '/oops/',
        component.context)
```

Examples from an application's "routing.*":

```
// Implement defaults
document.execute('/defaults/application/routing/')

// 404 errors
applicationInstance.statusService.capture(404, 'wackyhelp', '/help/main/',
        applicationInstance.context)
```

Notes for error capturing:

- You always need to specify the application name (like router.captureOther).
- As with regular capturing, you can hide these pages if you don't want users to be able to access them directly. (Though, if you hide your 404 page, users would still get to it because it's what would be displayed for hidden pages! The difference is that it would appear to users with a 404 status, not a 200 success status.)
- The difference between statusService.capture and statusService.captureHostRoot is in how the base URI is set, which affects the conversation.pathToBase API. statusService.capture uses to the application's base URI on the current virtual host, while statusService.captureHostRoot uses to the virtual host root itself.
- For 500 error capturing, you should prefer a /web/static/ page, which has the least chance of generating an exception and causing a 500 error again, resulting in a loop.

Static Redirection

By "static" here is meant that redirection is configured into the application's "routing.*" script. (You can also call this API on a running application, though you should realize that it essentially changed your application's routing.) "Dynamic" routing, from within /dynamic/web/ or /resources/, is described in the next section (page 60).

Because static redirection supports URI patterns (page 63), this API can actually handle complex "dynamic" redirections. For example, even though most clients will support relative URI paths for redirection, you can force a complete URI using the "{rp}" pattern, which equals is the request URI path.

The following redirection status codes are supported:

- 301, permanent: Most clients will cache this redirection, thus avoiding subsequent requests to the original URI
- 302, client found: This *should* be a temporary redirect, like 307, but many clients treat it as a 303. Best to avoid, unless you enjoy confusion.
- 303, see other: Clients assume that you have processed their request, and should send a GET to the new URI in order to receive the results of the processing. Use with care!

- 307, temporary: The default (because it's the safest)

Note that behind the scenes, static redirection is handled by attaching (page 61) a Redirector restlet.

Here are a few examples (JavaScript):

```
// Implement defaults
document.execute('/defaults/application/routing/')

// Simple redirection
router.redirectClient('/bug/', 'http://wackywiki.org/contact-us/bug/')

// Redirection to add a query to the request URI
router.redirectClient('/forum/', '{rp}?debug=true')

// Permanent redirection using URI segments
router.redirectClient('/contact/{reason}/', 'http://wackywiki.org/contact-us/{
    reason}', 301)

// The above is all Prudence sugar. Here's an example of what happens behind the
    scenes:
importClass(org.restlet.routing.Redirector)
router.attach('/bug/',
        new Redirector(applicationInstance.context, 'http://wackywiki.org/contact-
            us/bug/',
        Redirector.MODE_CLIENT_SEE_OTHER))
```

Dynamic Redirection

You can handle redirection at either /resources/ or /web/dynamic/ with the following API:

- For 301: conversation.response.redirectPermanent
- For 305: conversation.response.redirectSeeOther (use this if you've already processed the client's request, and want to redirect it to a new location in order to GET the results)
- For 307: conversation.response.redirectTemporary

For example:

```
<%
if (!conversation.internal) {
        conversation.response.redirectSeeOther('../contact-us/')
}
conversation.stop()
%>
```

Note that we are checking that the conversation is not internal. As of Prudence 1.1, internal redirects will cause exceptions!

Also, they will likely ignore any data in the redirected response, so you can just return null in /resources/ or empty text in /web/dynamic/. This is the reason we call conversation.stop (page) in the example above.

For other redirections in the 300 family, follow this example:

```
conversation.response.locationRef = 'http://wackywiki.org/report-a-bug/'
conversation.statusCode = 302
conversation.stop()
```

See also static redirection (page 59).

Attaching

This is the lowest-level routing API. It allows you to route URI patterns to any custom "restlet" (a REST conversation handler), as well as Restlet resources (which internally use a Finder restlet). Do your attachments in the application's "routing.*" configuration script. Prudence offers some attachment "sugar" in addition the standard Restlet API.

Here are a few examples (JavaScript):

```
// Implement defaults
document.execute('/defaults/application/routing/')

// Attach a directory instance
importClass(org.restlet.resource.Directory)
router.attach('/forum/info/',
        new Directory(applicationInstance.context,
        'file:///user/info/files/'))
        .matchingMode = Template.MODE_STARTS_WITH

// Or, you can use Prudence sugar (equivalent to MODE_STARTS_WITH)
router.attachBase('/forum/help/',
        new Directory(applicationInstance.context,
        'file:///user/data/files/'))

// More Prudence sugar: attach a resource via its classname
router.attach('/forum/help/{topic}/', 'org.wackywiki.HelpAccessResource')

// Prudence also lets you detach restlet instances
router.detach(staticWeb)
```

Filtering

Instances of filters, which wrap instances of other restlets, can be attached directly to URIs. The filter can then pass control to the "next" restlet, skip other filters along the chain, or immediately stop routing.

There are many use cases for filters. Here are just a few common ones:

- Guards: they check if the client's request has the right authorization (has the right cookie, originated at a white-listed IP address, etc.) before letting it continue. On stopping, they usually set the conversation.statusCode (page 76) to either 401 ("unauthorized") or 402 ("forbidden").
- Sanitizers: they remove harmful data or code from the client request, remove private or offensive material from the response, etc.
- Transformers: they translate, re-encode, decorate or otherwise transform response entities between various formats.
- Monitors: they can log each request passing through, accumulate statistics, alert administrators, etc.
- Throttles: they can refuse to pass requests on if the system is overloaded, if the client has exceeded a quota, etc. On stopping, they usually set the conversation.statusCode (page 76) to 503 ("service unavailable").
- Cachers: they can fetch from a cache according to request attributes, and store in a cache according to response attributes.
- Post-processors: they can find and interpret special codes, scriptlets, headers, etc., in the response as commands to execute.

In addition to the standard Restlet API, Prudence allows for a few useful shortcuts for filtering.

Here's an example of filtering all static web requests through the JavaScriptUnifyMinifyFilter (page 52):

```
importClass(com.threecrickets.prudence.util.JavaScriptUnifyMinifyFilter)

// This will detach staticWeb and attach our filter instead with staticWeb chained
    to it;
// also note that the context for staticWeb will be used for the filter, so we can
    specify
// null for the context in the constructor
// (this is Prudence sugar)

var filter = new JavaScriptUnifyMinifyFilter(null,
        new File(applicationBasePath + staticWebBasePath),
            minimumTimeBetweenValidityChecks)

router.filterBase(staticWebBaseURL, filter, staticWeb)

// The above is equivalent to this:

var filter = new JavaScriptUnifyMinifyFilter(staticWeb.context, staticWeb,
        new File(applicationBasePath + staticWebBasePath),
            minimumTimeBetweenValidityChecks)

router.reattachBase(staticWebBaseURL, staticWeb)
```

You can also filter the entire application by replacing its inbound router with a filter. For example:

```
var myFilter = new MyFilter(applicationInstance.context)

myFilter.next = applicationInstance.inboundRoot
applicationInstance.inboundRoot = myFilter
```

Custom Filters Prudence lets you easily create your own custom filters (page 48). The API for routing through them is almost identical to the above shortcut, except that, instead of a filter instance, you use the internal URI of the handler in the application's /handlers/ subdirectory. For example:

```
router.filterBase(dynamicWebBaseURL, '/remove-foul-language/', applicationInstance.
    context, dynamicWebBase)
```

The above is a useful shortcut, but in some cases you will need access to the actual filter instance. You would then need to explicitly create a DelegatedFilter instance. For example, the above is equivalent to:

```
importClass(com.threecrickets.prudence.DelegatedFilter)

var cleanupFilter = new DelegatedFilter(applicationInstance.context,
        '/remove-foul-language/', filtersDocumentSource, languageManager)

router.filterBase(dynamicWebBaseURL, cleanupFilter, dynamicWeb)
```

Custom filters can also be used for dynamic capturing (page 58).

Built-In Filters Prudence comes with a few useful filters: CacheControlFilter (page 51), JavaScriptUnifyMinifyFilter (page 52), and CssUnifyMinifyFilter (page 52).

Filtering for Authentication Though you can implement an authentication system of your own via custom filters, Restlet comes with a few useful ones.

In this example, we'll filter our entire application through a simple HTTP challenge authenticator. Here's the "routing.js":

```
importClass(
        org.restlet.security.MapVerifier,
```

```
        org.restlet.security.ChallengeAuthenticator,
        org.restlet.data.ChallengeScheme)

// Our authenticator will use this simple verifier
// (better verifiers will load the user/password table from storage, our verify
    against
// network services, such as LDAP)
var verifier = new MapVerifier()
verifier.localSecrets.put('theusername', new java.lang.String('thepassword').
    toCharArray())

// The HTTP challenge authenticator is a filter
var authenticator = new ChallengeAuthenticator(applicationInstance.context,
    ChallengeScheme.HTTP_BASIC, 'You must log in!')
authenticator.verifier = verifier

// Filter root through authenticator
authenticator.next = applicationInstance.inboundRoot
applicationInstance.inboundRoot = authenticator
```

URI Patterns

The custom routing techniques described above support URI patterns, which are URIs with optional curly-bracket-delimited variable names that are replaced by values per every client request. The use of one pattern can thus refer to many URIs.

Note that the same system is used to generate cache keys (page 68), and that some variables can only be used for cache keys (page 64). In fact, many of these variables might be more useful for cache keys than for routing.

For more information, see the Restlet Resolver documentation.

Data Attributes

All these refer to the data ("entity") sent by the client or that you are returning to the client. Lowercase is used for request attributes, uppercase for response attributes. We'll note these as pairs:

- {es} or {ES}: entity size in bytes
- {emt} or {EMT}: entity media type
- {ecs} or {ECS}: entity character set
- {el} or {EL}: entity language
- {ee} or {EE}: entity encoding
- {et} or {ET}: entity tag (HTTP ETag)
- {eed} or {EED}: entity expiration date
- {emd} or {EMD}: entity modification date

Request Attributes

- {d}: date (Unix timestamp)
- {m}: the method (in HTTP, it would be "GET," "POST," "PUT," "DELETE," etc.); see example for handling HTTP forms (page 40)
- {cia}: client IP address
- {ciua}: client upstream IP address (if the request reached us through an upstream load balancer)
- {cig}: client agent name

Response Attributes

- {S}: the status code
- {SIA}: server IP address
- {SIP}: server port number
- {SIG}: server agent name

URIs

We'll use a hyphen to show that you need to add one of the modifiers detailed after this list. For example, "{ri}" is the complete actual URI.

- {p}: the protocol ("http," "https," "ftp," etc.)
- {r-} or {R-}: actual URI (the capital "R" here referes to the response, which may be different from the request if you're redirecting)
- {h-}: virtual host URI
- {o-}: the application's root URI on the virtual host
- {f-}: the referring URI (usually means that the client clicked a hyperlink or was redirected here)

Add the following modifiers to URI values above in order to access the various parts of the URI:

- {-i}: the complete URI
- {-a}: the authority (for URLs, this is the host or IP address)
- {-p}: the path
- {-q}: the query
- {-f}: the fragment
- {-r}: the remaining part of the path
- {-e}: the part of the path relative to the application's root URI

Every URI also has a "base" URI, accessed via the "b" modifier. Usually, this is the application's root URI on the virtual host. You can add the URI modifiers above to this URI. For example: "{rbi}".

Cache Key Variables

The following variables are only available for cache key patterns (page 68), not for routing:

- {dn}: the document name (full path from the Prudence instance root)
- {an}: the application name
- {ptb}: the path to the base, identical to conversation.pathToBase (page 74)

conversation.locals

Variables that aren't any of the above will be assumed to be conversation.locals (page 79). You can thus inject any data you wish into a pattern.

This feature is especially useful with:

- URI capturing (page 56), which automatically parses URIs into conversation.locals
- Cache key pattern handlers (page 47), which can set conversation.locals before the cache key pattern is cast

API

Prudence exposes its API as a set of services to your source code. These services are available in scriptlets in /web/dynamic/, source code in /resources/, /handlers/, /tasks/, and in your configuration scripts.

Some notes for API use in different Prudence flavors:

Python If you're using the Jepp engine, rather than the default Jython engine, you will need to use get- and set- notation to access attributes. For example, use application.getArguments() to access application.arguments in Jepp.

Ruby Our Ruby engine, JRuby, conveniently lets you use the Ruby naming style for API calls. For example, you can use $application.get_global instead of $application.getGlobal.

JavaScript Our JavaScript engine, Rhino, does not provide dictionary access to maps, so you must use get- and put- notation to access map attributes. For example, use application.globals.get('myapp.data.name') rather than application.globals['myapp.data.name'].

Clojure You will need to use get- and set- notation to access attributes. For example, use (.getArguments application) to access application.arguments. You can use Clojure's bean form, for example (bean application), to create a read-only representation of Prudence services.

application

The same "application" service is shared between all code in a single application. Note that there is always a single application instance per application per component, even if the application is attached to several virtual hosts and servers.

The "application" service is a good place to store shared state for the application.

> Though the "application" service of the configuration scripts is different from that of your Prudence application, you can access the Prudence "application" service in that application's "default.*" script as "applicationService". For example, you can access the application logger as "applicationService.logger".

application.globals, application.getGlobal Application globals are general purpose attributes accessible by any code in the application.

Names can be any string, but the convention is to use "." paths to allow for unique "namespaces" that would not overlap with future extensions, expansions or third-party libraries. For example, use "myapp.data.sourceName" rather than "dataSourceName" to avoid conflict.

> Though application.globals is thread safe, it's important to understand how to use it properly. Make sure you read the section on concurrency in "sharing state" (page 81).

application.sharedGlobals, application.getSharedGlobal These are similar to application.globals (page 65), but are in fact shared by all Prudence applications running in the instance.

Note that some secure Prudence deployments may disable sharing between applications, in which case there will be no shared globals. It is thus best to test that they exist before using them:

```
var dbConnection = application.globals.get('db.connection')
if((dbConnection === null) && (application.sharedGlobals !== null)) {
        // Was not in application globals, so try shared globals
        dbConnection = application.sharedGlobals.get('db.connection')
}
```

application.distributedGlobals, application.getDistributedGlobal These are similar to application.globals (page 65), but are in fact shared by all members of the Hazelcast cluster to which we belong.

This is simply a convenience API to access the Hazelcast map named "com.threecrickets.prudence.distributedGlobals".

See application.distributedTask (page 67) for another Hazelcast-specific API.

application.arguments Available only in configuration scripts. This is a list of command-line arguments provided to the Prudence instance script.

application.application This is a reference to the underlying Restlet application instance. Here you can access some information defined in "settings.*", such as application.application.owner, application.application.author, application.application.statusService.contactEmail, etc.

You can configure the underlying application instance via "application.*" (page 27).

application.logger Use the logger to print messages in the log files. The messages are prefixed by the applicationLoggerName setting (page 28), which defaults to the application's subdirectory name.

By default, all log messages from all applications will be sent to prudence.log, but you can change this in /configuration/logging.conf.

See logging.conf (page 85).

application.getSubLogger Uses a logger that inherits the application.logger configuration by default. The name you use will be appended to your base logger name with a ".".

See logging.conf (page 85).

application.getMediaType Utility to get a Restlet MediaType instance from a MIME type name or filename extension.

Note that each application has its own mappings (page 27).

application.task Spawns or schedules in-process asynchronous tasks from the application's /tasks/ subdirectory. See tasks (page 49) for more information, and application.distributedTask (page 67) for a distributed version of this API.

There are five arguments:

1. The document name in /tasks/.
2. The context (or null). This is later available for the task as document.context (page 70).
3. Initial delay in milliseconds, or zero for ASAP.
4. Repeat delay in milliseconds, or zero for no repetition.
5. Whether repetitions are at fixed times (true), or if the repeat delay begins when the task ends (false). This value matters only if the repeat delay is greater than zero. Note that "true" may cause a new instance of the task to be spawned again before the previous one completes.

For example, to schedule a task to run exactly very 10 minutes, starting now:

```
application.task('/email/reminders/', null, 0, 10 * 60 * 1000, true)
```

Note that even if the task is set to run ASAP, with no repetition, it is executed asynchronously on a different thread. If you want to block until that task is completed, use the fact that this method returns a "future" object. Calling get() on this object will block until done. For example:

```
application.task('/database/cleanup/', 'fast', 0, 0, false).get()
```

You can also store this "future" object, and check (without blocking) to see if the task was completed:

```
var cleanup = application.globals.get('myapp.task.cleanup')
if(cleanup && cleanup.done) {
        application.logger.info('Database already clean!')
} else {
        application.globals.put('myapp.task.cleanup', application.task('/database/
            cleanup/', 0, 0, false))
}
```

application.distributedTask Spawns asynchronous tasks on any Prudence instance which is part of the Hazelcast cluster to which we belong (may also be our current instance). Other than the fact that this task may run out-of-process, this API is similar to application.task (page 66).

There are five arguments:

1. The application's full name, per the application setting (page 27).
2. The document name in /tasks/.
3. The context (or null). This is later available for the task as document.context (page 70). Note that for distributed tasks, unlike in-process tasks, the context *must be serializable*. Depending on your object implementation, this may mean having to manually serialize/deserialize the context into a string or another serializable format.
4. Where to run the task:
   (a) Null: let Hazelcast decide
   (b) Member instance: the particular cluster member
   (c) String: the particular cluster member for this key
   (d) Iterable of Member instances: all these members (see the fifth argument, below)
5. Multi-task support: whether the task should run on each member of the cluster (true), or on just one (false). Only applicable if "where" is an iterable of member instances.

See application.distributedGlobals (page 66) for an easy way to make state available to distributed tasks.

document

The "document" service has two distinct uses: first, it represents the file you are in: *this* document. This is where you can access the document's attributes and possibly change them. The second use of this service is for accessing *other* documents. Prudence combines these two uses into one service, but they functionally separate.

In the case of /resources/ and the configuration scripts, "this document" is simply the source code file. In the case /web/dynamic/, it's the whole "text-with-scriptlets" page, so it is shared by all scriptlets on the page, even if they are written in different languages.

This Document

Many of these attributes have to do with caching. Caching is your best tool to make sure your application can scale well. Read more about it in "Scaling Tips" (page 95).

> Note: Passing the "document" global variable as an argument to a function in another document will not let that other document access attributes for the calling document. In fact, all documents in the current thread share the same "document" service, which internally gets and sets attributes for the document in which it is accessed. If you do have a need for a function in one document to manipulate the "document" attributes of another document, you will have to return the values and have code in the owning document explicitly set the attributes there.

document.cacheDuration (only available in /web/dynamic/ and /web/fragments/)

The duration in milleseconds for which the output of this document will be cached. If this value is zero, the default, then caching is disabled. So, you must explicitly set this to a greater than zero value to enable caching. The key used to store and retrieve the cached output is determined by document.cacheKeyPattern (page 68).

See the section on caching in HTML generation (page 37) for more information.

> Note that if you edit the source file such as that it would be reloaded, within the limits defined by the minimumTimeBetweenValidityChecks setting (page 28), then any cache entries would not be used for the next request. This feature lets you see your changes on the fly, even if the document has been cached. Note, too, that updates to documents executed by your source file will also trigger this feature. See document.execute (page 71).

document.cacheKeyPattern (only available in /web/dynamic/ and /web/fragments/)

This lets you control the key that is used to store and retrieve the cached output of the current document. *Note that this is not necessarily the key itself*, but instead a pattern that can contain variables that are set dynamically. See URI patterns (page 63) for a complete list of built-in variables. You can additionally set your own variables using cache key pattern handlers (page 47).

The cache key itself is cast from the cache key pattern for every request reaching your document. This cache key is checked against the cache (page 69), and if a valid cache entry exists there, it is used. Otherwise, your document is executed. For debugging purposes, you can use document.cacheKey (page 68) to output the actual cache key used for the request.

Whatever pattern you use, Prudence will *always* add conversation.encoding (page 77) to the key, because you always want different encodings to be cached separately.

document.cacheKeyPattern is ignored if document.cacheDuration (page 68) is zero.

Cache key patterns are a very powerful feature that lets you easily create different cached versions of documents for different kinds of users and requests, but it's not always trivial to determine the best cache key pattern for every situation. It depends strongly on how you use and cache your fragments.

The default cache key pattern is "{ri}|{dn}|{ptb}", which is a string containing the request identifier (the URI), the document name, the path to base (page 74). An actual key could thus be: "http://mysite.org/wackywiki/main|/common/header.html". This default is sensible, because it makes sure that included fragments are cached invidually. For example, only using "{ri}" would have each included fragment use the same key and override others.

However, though sensible, the default cache key pattern may not be the most efficient. For example, if the header fragment used in the example above never changes per page, then it's wasteful to cache it separately per URI. It would be more efficient to set document.cacheKeyPattern = "{an}|{dn}" or "{an}|{ptb}|{dn}" in a scriptlet in header.html.

> Rule of thumb: Set document.cacheKeyPattern to be as short as you possibly can, but not too short that it won't differentiate between different views. See the section on caching in HTML generation (page 37) and "Scaling Tips" (page 95) for more information on how caching can help you scale.

The default cache key pattern can be changed by setting the "com.threecrickets.prudence.GeneratedTextResource.defaultC
application global.

document.cacheKeyPatternHandlers A map of cache key pattern (page 68) variable names (without the curly brackets) to cache key pattern handlers (page 47) names. Use this to override global handlers, or to otherwise set handlers specific to this document.

See cache key pattern handlers (page 47) for a complete reference.

document.cacheKey This is the actual cache key: document.cacheKeyPattern (page 68), cast for the current conversation. The value is read-only, and made available mostly for debugging purposes.

document.cacheTags (only available in /web/dynamic/ and /web/fragments/)

This is a list of one or more strings that will be attached to the cached output of the page. Any number of tags can be associated with a cache entry. Cache tags are used for document.cache.invalidate (page 70). Note that you can set cache tags to hardcoded strings (for example: "browse-pages") or dynamically generate them using code (for example: "blog-comments-for-entry-" + blogId).

document.cacheTags is ignored if document.cacheDuration (page 68) is zero.

See "Scaling Tips" (page 95) for more information on how cache tags can be used to help you scale.

> Important! Using document.include (page 70), or the <%& .. %> inclusion scriptlet, will cause the cache tags of the included document to be applied to the current document, and to documents that have included the current document, and so on. This is a useful feature: if a fragment included somewhere down the line is invalidated, you likely want the containing document to also be invalidated. If you wish to disable cache tag propagation, prefix the cache tag with an underscore when adding it to document.cacheTags. The tag is *not* stored with the underscore, and so should be referenced *without* the underscore when calling document.cache.invalidate (page 70).

document.cache (note that the same cache instance is accessible in both /web/dynamic/ and /resources/)

Provides access to the cache backend used by this document. Prudence supports a pluggable cache backend mechanism, allowing you to use RAM, disk, database, distributed and other cache systems. It also allows for chaining of various backends together for improved performance and reliability.

Though Prudence automatically caches the output of dynamic HTML and fragments, you can use the cache directly for your own purposes. Cache entries are instances of CacheEntry, which embeds various formatting attributes that you are free to ignore if you don't need them.

Prudence's default cache backend is a 1MB in-process memory cache, but this can be configured. Here's an example of chaining an H2 database cache after the default cache, via the "component.*" configuration script (page 23). The memory cache ensures extremely fast retrieval times, while the database cache makes sure entries will persist even if you restart Prudence:

```
// Implement defaults
document.execute('/defaults/instance/component/')

// Create an H2-database-backed cache chained after the default in-memory cache
importClass(
        com.threecrickets.prudence.cache.H2Cache,
        com.threecrickets.prudence.cache.ChainCache)

var defaultCache = component.context.attributes.get('com.threecrickets.prudence.
    cache')
var chainCache = new ChainCache()
chainCache.caches.add(defaultCache)
chainCache.caches.add(new H2Cache('cache/prudence/prudence'))
component.context.attributes.put('com.threecrickets.prudence.cache', chainCache)
```

Prudence also comes with memcached, Hazelcast and MongoDB cache backends, which can similarly be chained together. There's a full reference in the Java API documentation, but here are simple examples:

```
importClass(
        com.threecrickets.prudence.cache.MemcachedCache,
        com.threecrickets.prudence.cache.HazelcastCache,
        com.threecrickets.prudence.cache.MongoCache)

// The constructor accepts a max size in bytes (defaults to 1MB)
// (Note that you have to make sure that the JVM has been started with a heap
// size large enough for your cache)
component.context.attributes.put('com.threecrickets.prudence.cache',
        new InProcessMemoryCache(1024 * 1024 * 1024 * 1024) // 1GB

// memcached: The constructor accepts a list of nodes (ports required)
```

```
component.context.attributes.put('com.threecrickets.prudence.cache',
        new MemcachedCache('127.0.0.1:11211, 192.168.1.12:11211'))

// Hazelcast: The configuration is in /configuration/hazelcast.conf
component.context.attributes.put('com.threecrickets.prudence.cache',
        new HazelcastCache())

// MongoDB
component.context.attributes.put('com.threecrickets.prudence.cache',
        new MongoCache())
```

document.cache.invalidate This lets you remove entries, zero or many, from your cache at once. It is useful for when your application state changes in such a way that certain pages must be regenerated. The argument is a cache tag, as defined by document.cacheTags (page 69).

A common use case is to invalidate display pages when a user posts new data. For a detailed example, consider a forum hosting site. The home page has a section showing "recent posts in our forums" and additionally each forum has its own front page showing "forum highlights". Both of these query the data backend in order to generate the last, and have a 24-hour cache. You can associate each forum with cache tag "forum-X", where X is the forum number, and associate the home page with all these cache tags. When a user posts a new forum thread in forum X, you just need to call document.cache.invalidate("forum-X") to make sure all associated pages will be regenerated on the next user request.

See "Scaling Tips" (page 95) for more information on how caching can help you scale.

document.source This provides access to the underlying Scripturian DocumentSource instance used for this document.

Scripturian is Prudence's mechanism for loading, caching and executing source code in various languages. By default, Prudence uses a FileDocumentSource. From here, you can access attributes of it, for example: document.source.basePath and document.source.minimumTimeBetweenValidityChecks.

Refer to Scripturian's documentation for a complete reference.

document.invalidateCurrent Forces the current document to be marked as invalid, which would trigger the current and dependent documents to be refreshed. See document.addDependency (page 72) and document.invalidate (page 72).

document.context (only available in /tasks/)

Access to the context optionally sent to the task. See application.task (page 66) and application.distributedTask (page 67).

Other Documents

document.include (only available in /web/dynamic/ and /web/fragments/)

Executes another document "in place," meaning that its output is appended at the location in the document where you call document.include. Global variables, function definitions, class definitions, etc., in the other document would be made available locally.

The included path is an internal URI, not the external URL visible to the world. The URI can be relative to either /web/dynamic/ or to /web/fragments/.

Prudence might compile or otherwise prepare and cache scriptlets in the included document. What "preparation" actually involves depends on the language of the source code. This means that the first time it is included it would be delayed, but subsequent includes would be much faster. To avoid that first-time wait, Prudence supports "defrosting" of your documents when it starts. This is enabled by default for all documents in /web/dynamic/, and would affect included documents from /web/fragments/.

> Calling document.include is equivalent to using the include scriptlet, <%& ... %> (page 34). Internally, the include scriptlet is turned into a regular scriptlet that calls document.include.

There are three main use cases for inclusion:

1. This mechanism allows you to divide your documents into fragments that you can re-use in many documents, helping you manage large applications and keep them consistent. Fragments can include other fragments, those can include others, etc. A common strategy is to separate the document header and footer into fragments and include these in all pages.

2. Because each document fragment can have its own caching properties, fragmentation is also an important strategy for fine-grained caching. It's important to keep in mind, though, that the outermost document's cacheDuration will override all others. If a cached version of a document is used, then it is not executed, which means that document.include calls in it are not executed, too.

3. Scriptlets in the fragments can include re-usable code, such as function and class definitions. You can thus use document.include to include code libraries. You might want to consider, though, using document.execute (page 71) instead, as it will let you use regular source code documents and not have to use scriptlets.

document.execute Executes a program defined by source code in another document. Global variables, function definitions, class definitions, etc., in the other document would be made available locally.

The included path is an internal URI, not the external URL visible to the world. The URI can be relative to either /web/dynamic/ in the case of scriptlets, /resources/ in the case of resources, or to /libraries/ in either case. The /libraries/ subdirectory is indeed the best place to put code usable by all parts of your application.

Prudence might compile or otherwise prepare and cache code in the executed document. What "preparation" actually involves depends on the language of the source code. This means that the first time it is executed it would be delayed, but subsequent execution would be much faster. To avoid that first-time wait, Prudence supports "defrosting" of your documents when it starts. This is enabled by default for all documents in /resources/, and would affect documents they execute from /libraries/.

Executed documents, just like documents exposed at URLs, are reloaded and re-prepared on the fly if the file changes, within the limits of the minimumTimeBetweenValidityChecks setting (page 28). Also, executed documents are automatically added as dependencies, so that if they are reloaded, so will the current document. See document.addDependency (page 72).

> The main difference between document.include and document.execute is that the former expects "text-with-scriptlets" documents, while the latter uses plain source code.

Common use cases:

1. The executed code can be re-usable, such as function and class definitions. This allows you to treat it as a code library. Notes:

   (a) In most cases, you would probably want to use your language's code inclusion mechanism instead of document.execute. For example, use "import" in Python, Ruby and PHP, and "require" in Clojure. The native inclusion mechanism would do a better job at caching code, managing namespaces, avoiding duplication, etc. For example, if you use document.execute in Clojure, then you would have to use defonce rather than def to avoid duplication in case you execute the same document multiple times in the same context.

   (b) For JavaScript and Groovy flavors: Prudence's document.execute is your only option for code inclusion in Prudence, because both JavaScript and Groovy do not have a code inclusion mechanism.

2. The executed code does not have to be in the same language as the calling code. This lets you use multiple languages in your applications, using the strengths of each. Note that languages cannot normally share function and class definitions, but can share state, via mechanisms such as application.globals, if the languages use compatible structures.

3. Use document.execute as an alternative to document.include if you prefer not to use scriptlets. For example, executing "mylibrary.js" might be more readable than including a fragment called mylibrary.html that is all just one JavaScript scriptlet.

document.addDependency Explicitly adds a dependency to this document. The dependency is an internal document URI, identical to what would be used in document.execute (page 71).

If the dependency changes, the current document would be reloaded when accessed. This effect cascades to documents that the dependency depends on. This is useful, because the execution document may contain code that affects the preparation of the current document.

Note that document.execute (page 71) automatically adds the executed document as dependency. You would want to call document.addDependency explicitly only if you are not using document.execute. For example, if you are using your language's built-in library inclusion mechanisms.

document.addFileDependency Like document.addDependency (page 72), except that any file path can be added as a dependency. The contents of such files is ignored.

For constructing the file path, you may want to use document.source.basePath (page 70).

document.invalidate Forces the provided document to be marked as invalid, which would trigger the document and dependent documents to be refreshed. See document.addDependency (page 72) and document.invalidateCurrent (page 70).

document.internal Creates a ClientResource proxy for a resource in the current application, or in other applications in your component. You can perform all the usual REST verbs via the proxy: GET, PUT, POST, DELETE, etc.

The first argument is a URI, which is not relative to any server or virtual host, but to the application root. The second argument is the preferred media type name.

Common use cases:

1. By letting you use your REST resources both internally and externally, document.internal lets you create unified resource API. document.internal directly accesses the resource (not via HTTP), so it is just about as fast as a simple function call, and is definitely scalable. Thus, there may be no need to create a separate set of functions for you to use internally and HTTP resources for external clients to use. A unified API would minimize the possibility of bugs and add coherence to your code, by enforcing a RESTful architecture all around. A second advantage is that you could trivially make your API remote by running it on a different Prudence instance, and using document.external (page 72) instead. This could allow for an easy way to run your application in a cluster, behind a load balancer.
2. You can create unit tests for your resources without having to start an HTTP server.
3. The preheating mechanism uses document.internal to load resources.

   document.internal makes requests via the RIAP (Restlet Internal Access Protocol). You can construct URIs of this pseudo-protocol on your own, like so: "riap://application/myresource/". A call to document.external with this URI would be identical to calling document.internal with "/myresource/". See the Restlet documentation for full information about RIAP and other pseudo-protocols (CLAP, JAR, WAR, etc.).

document.external Creates a ClientResource proxy for a resource. You can perform all the usual REST verbs via the proxy: GET, PUT, POST, DELETE, etc.

The first argument is a full URL. The second argument is the preferred media type name.

Common use cases:

1. You can add scale and redundancy to your internal REST API by running several Prudence instances behind a load balancer, and using document.external instead of document.internal.
2. You can use Prudence to create an internal communication backbone for your enterprise, with various backend services exposing resources to each other. You can expose the same resources to business partners, allowing for "B2B" (business-to-business) services.
3. There are many, many use cases for document.external, and they keep growing as REST is adopted by service providers on the Internet. There are online storage, publishing and content delivery systems, public databases, archives, geolocation services, social networking applications, etc. Perhaps, with Prudence, you will create the next one.

> Important: The semantics of ClientResource require you to explicitly release the response as soon as you're done using it, which will in turn release the thread used to stream the response to you. (Though these threads might eventually be reclaimed by a built-in garbage collection mechanism, leaving them hanging could lead to starvation, such that further uses of document.external will block until threads return to the pool.) It's probably best to employ a try-finally paradigm, where the try clause contains as little code as possible, and the finally clause releases the response. See the example below.

This is a good place to remind you that REST is not just HTTP. By default, Prudence supports http: and file: scheme URIs for document.external, and you can add more protocols via your "/instance/clients.*" configuration script (page 23).

Here's an example of using document.external to read a JSON representation from the filesystem:

```
var fixture
var resource = document.external('file:///myfiles/fixture.json', 'application/json
    ')
try {
        fixture = resource.get().text
}
finally {
        resource.response.release()
}

// Do something with the fixture here...
```

> In case you're interested, Prudence internally uses Apache's HTTP client for high-performance access to external HTTP documents, with full support for secure connections (prefixed with "https://").

document.preferredExtension (only available in configuration scripts)

If multiple files with the same name but different extension exist in the same directory, then this extension will be preferred.

This value is set automatically according the Prudence flavor you are using.

executable

The "executable" is the low-level equivalent of "this document" (page 67). Here you can explore which languages are installed in your Prudence instance, and gain access to their implementation mechanism. You'll likely never need to do any of this in your Prudence application. For more information on executables, see Scripturian, the library that handles code execution for Prudence.

A feature that you might want to use here is the executable globals. These are similar to the application globals (page 65), except that they are global to the entire Prudence instance (in fact, to the virtual machine). It's a good place to store state that you want shared between Prudence applications.

executable.globals, executable.getGlobal These are similar in use to application.globals, but are shared by all applications on the VM. See "sharing state" (page 80) for more information.

executable.context The context is used for communication between the Prudence container and the executable.

- executable.context.writer: direct access to the output writer (writes to a memory buffer in /web/dynamic/, and to standard output in /resources/, /handlers/, /tasks/ and configuration scripts)
- executable.context.exposedVariables: Prudence services are here (application, document, executable, conversation)
- executable.context.attributes: for use by the language engines

executable.manager This is the Scripturian LanguageManager used to create executables in many languages. Here you can query which languages are supported by the current Prudence instance: executable.manager.adapters.

executable.container This is identical to the "document" service (page 67). For example, "document.include" is the same as "executable.container.include".

(Internally, Prudence uses this equivalence to hook the include scriptlet, a Scripturian feature, into Prudence's document.include.)

conversation

The "conversation" represents the request received from the user as well your response to it, hence it's a "conversation." Because Prudence is RESTful, conversations encapsulate exactly a single request and its response. Higher level "session" management, covering an arbitrary number of conversations, is up to you.

Here you can access various aspects of the request: the URI, formatting preferences, client information, and actual data sent with the request (the "entity"). You can likewise set response characteristics.

> Note that in /web/dynamic/ "conversation" is available as a global variable. In /resources/ and /handlers/, it is sent to the handling entry points (functions, closures, etc.) as an argument. Usage is otherwise identical. "conversation" is not available in configuration scripts or in /tasks/.

The Request

conversation.reference This is the URI used by the client. Refer to the Restlet Reference documentation for complete details.

A few useful attributes:

- conversation.reference.identifier: the complete URI
- conversation.reference.path: the URI, not including the domain name and the query matrix
- conversation.reference.segments: a list of URI segments in the path
- conversation.reference.lastSegment: the last segment in the URI path
- conversation.reference.fragment: the URI fragment (whatever follows "#")
- conversation.reference.query: the URI query (whatever follows "?"); you might prefer to use conversation.query (page 75) instead
- conversation.reference.relativeRef: a new reference relative to the base URI (usually the application root URI on the current virtual host)

conversation.pathToBase This is a URI path relative to the base URI, which is usually the application root URI on the current virtual host.

This is very useful to allow for relative references in HTML. It's especially sueful in fragments that you might include at various parts of your application, and for captured URIs (page 56). For example, let's say you have a contact page at /dynamic/web/contact/index.html. The following HTML snippet can be used anywhere in your application:

```
Click <a href="<%= conversation.pathToBase %>/contact/">here</a> to contact us.
```

> Note: If you are caching a document that relies on conversation.pathToBase, you likely also want to include "{ptb}" or some other path reference in the document.cacheKeyPattern (page 68).

conversation.form, conversation.formAll Use these to access data sent via HTML forms, or any other request with a data entity of the "application/x-www-form-urlencoded" MIME type.

conversation.form is a map of form names to values. In case the form has multiple values for the same name, only the last one is mapped.

conversation.formAll is a list of parameters. Use this if you need to support multiple values for the same name.

A value is often a simple string, but in the case of file uploads, it could be a FileParameter instance, with the following attributes:

- data: the raw data as an array of bytes, for uploaded files stored in memory
- file: a JVM File instance, for files stored on disk; uploaded files are stored in the application's /uploads/ subdirectory
- size: in bytes
- mediaTypeName: the MIME type set by the client, such as "image/jpeg"
- mediaType: the MediaType instance equivalent to the above, if available

Note that either "data" or "file" are valid, but not both. One will always be null.

Prudence allows you to keep smaller files in memory. See the fileUploadSizeThreshold setting (page 30).

conversation.query, conversation.queryAll Use these to access data sent via the URI query.

conversation.query is a map of query parameter names to values. In case the URI has multiple values for the same name, only the last one is mapped.

conversation.queryAll is a list of parameters. Use this if you need to support multiple values for the same name.

conversation.entity (only available in HTTP PUT and POST commands)

This is the data entity sent by the client.

In the case of an HTML POST form, it would be more convenient to use conversation.form (page 75) to access the parsed entity data. For other kinds of data, you have to parse the entity data yourself. Before attempting to parse entity data on your own, make sure to look through the Restlet API and its extensive set of plugins for tools to help you parse representations. Plugins exist for many common Internet formats.

The value is a Restlet Representation instance. A few useful representation attributes:

- conversation.entity.size: the size of the data in bytes, or -1 if unknown
- * conversation.entity.text: the data as text (only useful if the data is textual)
- * conversation.entity.reader: an open JVM Reader to the data (only useful if the data is textual)
- * conversation.entity.stream: an open JVM InputStream to the data (useful for binary data)

  Note: Client data is provided as a stream that can only be "consumed" once. Attributes that cause consumption are marked with a "*" above. *Note that conversation.entity.text is one of them!* If you want to access conversation.entity.text more than once, save it to a variable first.

conversation.preferences Provides you with low-level access to the Restlet Variant instance negotiated with the client. You will usually not have to use this, because response attributes (page 76) are set accordingly.

Negotiation happens between the client's preferences, if provided by the client in the request, and our list of supported variations.

In /resources/, this attribute will be null during the handleInit entry point (page 43), precisely because we have not yet set our list of supported variations.

In /web/dynamic/ (and /web/static/), our supported media type will be set according to the filename extension (page 55)

The Response

By default, Prudence initializes all response attributes according to the client's preferences (page 75). However, you may want to explicitly change them. Also note that some clients to not specify preferences, in which case the response attributes will be undefined.

conversation.statusCode, conversation.status This is the HTTP status code returned to the client. The default status code is 200 ("OK").

Prudence will automatically set the status code to 500 ("internal server error") in the case of an unhandled exception in your code. It can also display a debug page (page 83) in such cases. You can also capture error codes (page 58) and display custom error pages to users.

These two variants represent the same value, letting you access the value in different ways.

conversation.statusCode is an HTTP status code as an integer.

conversation.status is the underlying Status instance.

conversation.mediaTypeName, conversation.mediaTypeExtension, conversation.mediaType In /resources/, the "application/java" media type is treated specially (page 43).

These three variants all represent the same value, letting you access the value in different ways.

conversation.mediaTypeName is the MIME representing the media type. MIME (Multipurpose Internet Mail Extensions) is an established web standard for specifying media types. Examples include: "text/plain," "text/html," "application/json," and "application/x-www-form-urlencoded." The exact list of supported MIME types depends on the underlying Restlet implementation.

conversation.mediaTypeExtension is the media type as the default filename extension for the media type. For example, "txt" is equivalent to MIME "text/plain," and "xml" is equivalent to "application/xml." Each application has its own mappings of filename extensions to media types. See also application.getMediaType (page 66), and how to change the mappings for your application (page 27).

conversation.mediaType is the underlying MediaType instance.

The default media type will be set according to conversation.preferences (page 75). Use conversation.request.clientInfo.acceptedMediaTypes to find out more generally which media types the client supports.

conversation.characterSetName, conversation.characterSetShortName, conversation.characterSet These three variants all represent the same value, letting you access the value in different ways.

conversation.characterSetName is ISO's UTC (Universal Character Set) name of the character set. For example, "ISO-8859-1" is the "Latin 1" character set and "UTF-8" is the 8-bit Unicode Transformation Format, "US-ASCII" is ASCII, etc. The exact list of supported ISO names depends on the underlying Restlet implementation.

conversation.characterSetShortName is a shortcut name for the character set. Shortcuts include "ascii," "utf8," and "win" (for the Windows 1252 character set). Restlet handles shortcuts names together with filename extension mappings.(page 27).

conversation.characterSet is the underlying CharacterSet instance.

The default character set will be set according to conversation.preferences (page 75). Use conversation.request.clientInfo.acceptedCharacterSets to find out more generally which character sets the client supports.

> If the client does not specify a preferred character set, then the character set will fall back to a default determined by the "com.threecrickets.prudence.GeneratedTextResource.defaultCharacterSet" application global. It not explicitly set, it will be UTF-8.

conversation.languageName, conversation.language These two variants represent the same value, letting you access the value in different ways.

conversation.languageName is the IETF locale name for the language. Examples include "en" for English, "en-us" for USA English, "fr" for French, etc. The exact list of supported IETF names depends on the underlying Restlet implementation.

conversation.language is the underlying Language instance. Because IETF names are hierarchical, you might prefer to use this as a way to test for containment. For example, conversation.language.includes will tell you that "en-us" is included in "en."

The default language will be set according to conversation.preferences (page 75). Use conversation.request.clientInfo.acceptedLanguages to find out more generally which languages the client supports.

Note that the language can be a null value, and that responses do not have to specify a language.

conversation.encodingName, conversation.encoding In /web/dynamic/, this value is read-only, defined by client. If it is "zip", "gzip" or "deflate," Prudence will compress the output text. Cache entries are stored in their compressed state, such that subsequent cache retrievals will not involve this additional overhead. See caching and encoding (page 38) for a detailed discussion

In /resources/, you can change this value, but all it would do is set the appropriate header in the response. It is up to you to supply Prudence with a correctly encoded representation.

These two variants represent the same value, letting you access the value in different ways.

conversation.encodingName is an internal name used for the encoding. Examples include "zip," "gzip," "compress," and "deflate." The "*" name represents all possible encodings. The exact list of supported names depends on the underlying Restlet implementation.

conversation.encoding is the underlying Encoding instance.

The default encoding will be set according to conversation.preferences (page 75). Use conversation.request.clientInfo.acceptedEncodings to find out more generally which encodings the client supports.

conversation.disposition Access to the response disposition, which lets you define how clients should treat your representation.

conversation.disposition.type can accept the values "attachment", "inline" or "none".

Example:

```
conversation.disposition.type = 'attachment'
conversation.disposition.filename = 'revenue.csv'
```

Cookies

conversation.cookies This is initialized as a list (not a map) of cookies sent from the client. Use conversation.getCookie (page 78) as a shortcut to retrieve a cookie by name.

If you want to ask the client to change any of them, be sure to call save() on the cookie in order to send it in the response. You can also call remove() to ask the client to delete the cookie (no need to call save in that case; internally sets maxAge to zero). Note that you can call save() and remove() as many times as you like, and that only the last changes will be sent in the response.

Note that you can only *ask* a client to change, store cookies, or for them to be used in various. It's up to the client to decide what to do with your requirements. For example, many web browsers allow users to turn off cookie support or filter out certain cookies.

Cookie instances have the following attributes:

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

The following attributes are not received from the client, but you can set them for sending to the client:

- cookie.maxAge: age in seconds, after which the client should delete the cookie. maxAge=0 deletes the cookie immediately, while maxAge=-1 (the default) asks the client to keep the cookie only for the duration of the "session" (this is defined by the client; for most web browsers this means that the cookie will be deleted when the browser is closed).
- cookie.secure: true if the cookie is meant to be used only in secure connections (defaults to false)
- cookie.accessRestricted: true if the cookie is meant to be used only in authenticated connections (defaults to false)
- cookie.comment: some clients store this, some discard it

conversation.getCookie Gets a cookie by its name, or returns null if it doesn't exist. See conversation.cookies (page 77).

conversation.createCookie You must provide the cookie name as an argument. Returns a new cookie instance if the cookie doesn't exist yet, or the existing cookie if it does. The cookie is added to conversation.cookies (page 77).

For new cookies, be sure to call save() on the cookie in order to send it in the response, thus asking the client to create it, or delete() if you want to cancel the creation (in which case nothing will be sent in the response).

Conditional Requests

These attributes are only available for /resources/. See conditional requests (page 45) for use cases.

For /web/dynamic/, these attributes are indirectly set according to document.cacheDuration (page 68) and dynamicWebClientCachingMode setting (page 29).

conversation.modificationTimestamp, conversation.modificationDate These two variants represent the same value, letting you access the value in different ways.

conversation.modificationTimestamp is a long integer value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT ("Unix time"). Once you set conversation.modificationTimestamp for a conversation, you cannot "unset" it—you can only change it to another value. You can, however, set conversation.modificationDate to null instead.

conversation.modificationDate is the underlying JVM Date instance. Refer to the Java API documentation for details.

conversation.httpTag, conversation.tag These two variants represent the same value, letting you access the value in different ways.

conversation.httpTag is an HTTP ETag string. Once you set conversation.httpTag for a conversation, you cannot "unset" it—you can only change it to another value. You can, however, set conversation.tag to null instead.

conversation.tag is the underlying Tag instance.

Client Caching

These attributes are only available for /resources/. See conditional requests and client caching (page 46) for use cases.

For /web/dynamic/, these attributes are indirectly set according to document.cacheDuration (page 68) and dynamicWebClientCachingMode setting (page 29).

conversation.maxAge Clients are asked not to use cached versions of the response entity for more than this number of seconds *and* not to perform conditional requests until then. You can also use conversation.expirationDate (page 79) instead of this value (for most clients, this value supercedes conversation.expirationDate).

Unlike most other attributes, once you set the max age for the conversation, you cannot "unset" it—you can only change it to another value.

A value of -1 is special: it signifies that Prudence should use the "no-cache" HTTP directive instead of "max-age". Though it make look as if it would have the same effect as setting "max-age" to zero, some clients interpret "no-cache" more explicitly and make sure not to keep any local copy of the response.

Cache expiration can be used either alone or in conjunction with conditional requests (page 45). See the dynamicWebClientCachingMode setting (page 29) for further discussion.

conversation.expirationTimestamp, conversation.expirationDate Clients are asked not to use cached versions of the response entity after this date *and* not to perform conditional requests until then. You can also use conversation.maxAge (page 78) instead of this value (for most clients, conversation.maxAge supercedes this value).

Cache expiration can be used either alone or in conjunction with conditional requests (page 45). See the dynamicWebClientCachingMode setting (page 29) for further discussion.

These two variants represent the same value, letting you access the value in different ways.

conversation.expirationTimestamp is a long integer value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT ("Unix time"). Once you set conversation.expirationTimestamp for a conversation, you cannot "unset" it—you can only change it to another value. You can, however, set conversation.expirationDate to null instead.

conversation.expirationDate is the underlying JVM Date instance. Refer to the Java API documentation for details.

Resource Initialization

conversation.addMediaTypeByName, conversation.addMediaTypeByExtension, conversation.addMediaType These three variants all do the same thing, letting you add a media type to the list of media types you can support in the response. After negotiation with the client's preferences, the preferred media type will be found in conversation.preferences (page 75) and in conversation.mediaType (page 76).

You'll want to call one or more of these in a resource's handleInit() entry point (page 43). Order of these calls is important, as it defines order of preference in case the client supports multiple media types.

conversation.addMediaTypeByName accepts the MIME representing the media type. MIME (Multipurpose Internet Mail Extensions) is an established web standard for specifying media types. Examples include: "text/plain," "text/html," "application/json," and "application/x-www-form-urlencoded." The exact list of supported MIME types depends on the underlying Restlet implementation.

conversation.addMediaTypeByExtension accept the media type as the default filename extension for the media type. For example, "txt" is equivalent to MIME "text/plain," and "xml" is equivalent to "application/xml." Each application has its own mappings of filename extensions to media types. See also application.getMediaType (page 66), and how to change the mappings for your application (page 27).

conversation.addMediaType accepts the underlying MediaType instance.

Conversation Flow

conversation.stop Throws an exception, thereby ending execution of your code, and hence the conversation—unless you have deferred it: see conversation.defer (page 80). Note that the client will still get a response, so you can set attributes (conversation.statusCode, conversation.expirationTimestamp, etc.) before calling conversation.stop.

conversation.internal True if the client's request was internal, false if it was external.

Internal requests are usually created in one of two ways:

1. The document.internal API (page 72)
2. URI capturing (page 56)

conversation.locals A map of general purpose attributes that is destroyed at the end of the conversation. Importantly, conversation.locals are maintained even if the conversation has been deferred via conversation.defer (page 80).

See "sharing state" (page 80) for more information.

conversation.defer (only available in /web/dynamic/ and /web/fragments/)

Releases the current conversation thread, and queues handling of this conversation on a separate thread pool. When the conversation turn comes to be handled, it will cause the page to be executed again, but with conversation.deferred (page 80) set to true. Use conversation.locals if you want to pass state for the deferred execution.

Returns true if indeed the conversation has been successfully deferred. Will return false if the conversation is already deferred.

Note that calling conversation.defer does not stop the current execution. You'd likely follow a successful call to conversation.defer with a call to conversation.stop. For example:

```
if(conversation.defer()) {
        conversation.stop()
}
```

> This is an experimental feature in Prudence 1.1. The use of a separate thread pool is only supported when using the internal Restlet connector. For other connectors (such as Jetty, the default), a successful call to defer will cause the page to be executed again in the same thread.

conversation.deferred (only available in /web/dynamic/ and /web/fragments/)

True if the conversation has been deferred via a call to conversation.defer (page 80).

Low-level Access

Use these to access the Restlet instances underlying the conversation. This is useful for features not covered by Prudence's standard API.

For more information, refer to Prudence's Java API documentation and also Restlet's API documentation.

conversation.resource The Restlet resource instance. In the case of /web/dynamic/, this will be Prudence's GeneratedTextResource. In the case of /resources/, this will be DelegatedResource.

conversation.request Equivalent to conversation.resource.request.

conversation.response Equivalent to conversation.resource.response.

Sharing State

Prudence is designed to allow massive concurrency and scalability while at the same time shielding you from the gorier details. However, when it comes to sharing state between different parts of your code, it's critical that you understand Prudence's state services.

Global Variables

You know how local variables work in your programming language: they exist only for the duration of a function call, after which their state is discarded. If you want state to persist beyond the function call, you use a global variable (or a "static" local, which is really a global).

But in Prudence, you cannot expect global variables to persist beyond a user request. To put it another way, you should consider every single user request as a separate "program" with its own global state. See the "life" sections for generating HTML (page 37) and resources (page 40) for more information on when this global state is created and discarded. If you need global variables to persist, you must use application.globals (page 65) or executable.globals (page 73).

Why does Prudence discard your language's globals? This has to do with allowing for concurrency while shielding you from the complexity of having to guarantee the thread-safety of your code. By making each user request a separate "program," you don't have to worry about overlapping shared state, coordinating thread access, etc., for every use of a variable.

The exception to this is code in /resources/, in which language globals *might* persist. To improve performance, Prudence caches the global context for these in memory, with the side effect that your language globals persist beyond a single user request. For various reasons, however, Prudence may reset this global context. You should not rely on this side effect, and instead always use application.globals (page 65) or even executable.globals (page 73).

application.globals vs. executable.globals

You should prefer application.globals. By doing so, you'll minimize interdependencies between applications, and help make each application deployable on its own.

It's best to use executable.globals as an *optional* bridge between applications. Examples:

1. To save resources. For example, if an application detects that a database connection has already been opened by another application in the Prudence instance, and stored in executable.globals, then it could use that connection rather than create a new one. This would only work, of course, if a few applications share the same database.

2. To send messages between applications. This would be necessary if operations in one application could affect another. For example, you could place a task queue in executable.globals, where application could queue required operations. A thread in another application would consume these and act accordingly. Of course, you will have to plan for asynchronous behavior, and especially allow for failure. What happens if the consumer application is down?

Generally, if you find yourself having to rely on executable.globals, ask yourself if your code would be better off encapsulated as a single application. Remember that Prudence has powerful URL routing, support for virtual hosting, etc., letting you easily have one application work in several sites simultaneously

Note for Clojure flavor: All Clojure vars are VM-wide globals equivalent in scope to executable.globals. You usually work with namespaces that Prudence creates on the fly, so they do not persist beyond the execution. However, if you explicitly define a namespace, then you use it as a place for shared state. It will then be up to you to make sure that your namescape doesn't collide with that of another application installed in the Prudence insstance. Though this approach might seem to break our rule of thumb here, of preferring application.globals to executable.globals, it is more idiomatic to Clojure and Lisps more generally.

Concurrency

Though application.globals and executable.globals are thread safe, it's important to understand how to use them properly.

Note for Clojure flavor: Though Clojure goes a long way towards simplifying concurrent programming, it does not solve the problem of concurrent access to global state. You still need to read this section!

For example, this code (Python flavor) is broken:

```
def get_connection()
        data_source = application.globals['myapp.data.source']
        if data_source is None:
                data_source = data_source_factory.create()
                application.globals['myapp.data.source'] = data_source
        return data_source.get_connection()
```

The problem is that in the short interval between comparing the value in the "if" statement and setting the global value in the "then" statement, another thread may have already set the value. Thus, the "data_source" instance you are referring to in the current thread would be different from the "myapp.data.source" global used by other threads.

This may seem like a very rare occurence to you: another thread would have to set the value *exactly* between our comparison and our set. If your application has many concurrent users, and your machine has many CPU cores, it can actually happen quite frequently. And, even if rare, your application has a chance of breaking if just two users use it at the same time!

Use this code instead:

```
def get_connection()
        data_source = application.globals['myapp.data.source']
        if data_source is None:
                data_source = data_source_factory.create()
                data_source = application.getGlobal('myapp.data.source',
                    data_source)
        return data_source.get_connection()
```

The getGlobal call is an atomic compare-and-set operation. It guarantees that the returned value is the unique one.

Optimizing for Performance You may have noticed, in the code above, that if another thread had already set the global value, then our created data source would be discarded. If data source creation is heavy and slow, then this could significantly affect our performance. The only way to guarantee that this would not happen would be to make the entire operation atomic, by synchronizing it with a lock:

Here's an example:

```
def get_connection()
        lock = application.getGlobal('myapp.data.source.lock', RLock())
        lock.acquire()
        try:
                data_source = application.globals['myapp.data.source']
                if data_source is None:
                        data_source = data_source_factory.create()
                        application.globals['myapp.data.source'] = data_source
                return data_source.get_connection()
        finally:
                lock.release()
```

Note that we have to store our RLock as a unique global, too.

Not only is the code above complicated, but synchronization has its own performance penalties, which *might* make this apparent optimization actually perform worse. It's definitely not a good idea to blindly apply this technique: attempt it only if you are experiencing a problem with resource use or performance, and then make sure that you're not making things worse with synchronization.

If all else fails, then Prudence's globals may not be the best solution for your problem. Look into creating an external service (possibly written in Java) to manage global connections for you.

Here's a final version of our get_connection function that lets you control whether to lock access:

```
def get_connection(lock_access=False)
        if lock_access:
                lock = application.getGlobal('myapp.data.source.lock', RLock())
                lock.acquire()
        try:
                data_source = application.globals['myapp.data.source']
                if data_source is None:
                        data_source = data_source_factory.create()
                        if lock_access:
                                application.globals['myapp.data.source'] =
                                    data_source
                        else:
                                data_source = application.getGlobal('myapp.data.
                                    source', data_source)
                return data_source.get_connection()
        finally:
                if lock_access:
                        lock.release()
```

Complicated, isn't it? Unfortunately, complicated code and fine-tuning is the price you must pay in order to support concurrent access, which is the key to Prudence's scalability.

But, don't be discouraged. The standard protocol for using Prudence's globals will likely be good enough for the vast majority of your state-sharing needs.

conversation.locals

These are not "local" in the same way that function locals are. The term "local" here should be read as "local to the conversation" (compare with the term "thread locals"). They are "global" in the sense that they can be accessed by any function in your code, but are "local" in the sense that they persist only for the duration of the user request.

You may ask, then, why you wouldn't want to just use your language globals, which have the same scope and life. conversation.locals have four main uses in Prudence:

1. To easily share conversation-scope state between scriptlets written in different languages.
2. To share state for deferred conversations—see conversation.defer (page 80).
3. Captured URI segments are stored here (page 56).
4. They can be used as document.cacheKeyPattern (page 68) variables, in conjunction with cache key pattern handlers (page 47).

Sharing State Between Prudence Applications and Instances

See application.sharedGlobals (page 65) and application.distributedGlobals (page 66).

Debugging

Logging

Logging (page 84) is the developer's best friend. Use it wisely, and you'll be able to clearly analyze the flows of control, code and data.

The Debug Page

The debug page is returned as a response whenever an un-handled exception is thrown in your code (or, if there is a bug in Prudence!).

> Though very useful during development, you'd probably want to turn it off for production systems. Simply set the showDebugOnError setting (page 28) to false. Note that in that case, you might want to capture errors, sending them to a friendly, apologetic page for the users. See error capturing (page 58).

The debug page shows you the following information about the conversation:

- A stack trace of the exception, with a link to a view of the source code (page 84).
- The reference (URI) used by the client, including the application's root reference, the virtual host reference, the original version of the reference (it might have been altered by filters along the route), and whether it was captured.
- The query of the URI.
- Cookies included in the request.
- Breakdown of the request metadata: media types, character sets, encodings, languages, etc.
- Request conditions, if included.
- Entity data, if available and not yet consumed by your code.

- Caching directives, if included.
- Information about the client: IP address, browser, operating system, user-agent, etc.
- Request attributes, including captured attributes.
- Warnings, if included.
- Underlying JVM stack trace.

Live Viewing of Source Code

With debug mode enabled, Prudence supports live viewing of source code in /web/dynamic/ and /resources/, with syntax coloring provided by Jygments. Links to source code are provided by the debug page, but you can also GET source code directly via:

- Prefixing URIs with "/sourcecode/"; see the showSourceCodeURL setting (page 28). For example, use GET on "/sourcecode/support/forum/" to see the source code for the resource or dynamic page at "/support/forum/".
- Adding a "?source=true" to the query for URIs of /web/dynamic/ pages.

In both cases, you can add "?highlight=n" to the query, when "n" is the line number to highlight.

Breakpoints?

As of Prudence version 1.1, you cannot set breakpoints in your code, unless it's in Java. Future versions of Prudence may allow breakpoints for some flavors, as supported languages add more debugging features. However, we do not feel that this is such a great loss, or that it would adversely affect your ability to develop for Prudence. The combination of robust logging and the debug page can go a long way towards helping you diagnose your problems. Debugging highly concurrent applications, such as network servers, does not work very well with breakpoints, anyway.

Logging

Prudence comes pre-configured with robust logging, based on log4j.

You are encouraged to make use of logging in your applications, but even if you don't, you will still find the logs useful. Prudence's servers, routers, programming languages and other components all send messages to the logs, making them an invaluable tool for debugging, monitoring and understanding how Prudence works.

By default, logs are sent to the /logs/ directory, using configurable rolling log schemes. /logs/web.log records all server requests, using Apache's format, while everything else goes to /logs/prudence.log.

Loggers

The Base Logger

Use application.logger (page 66) to send text messages to the log. The logger name defaults to your application's subdirectory name, but can be configured via the applicationLoggerName setting (page 28).

Sub-Loggers

Large applications might benefit from using more than one logger. Use application.getSubLogger (page 66) with any name you wish. This name will be appended to your base logger name with a ".", and will inherit the base logger's configuration by default. For example, if your base logger is named "wackywiki", then application.getSubLogger('backend') will appear as "wackiwiki.backend" in the log files.

See more on logger inheritance below.

Sending Messages

Whether messages actually are written to the log file depends on your logging configuration. Messages are ranked by levels, and loggers are configured to allow only messages up to a certain level. Smart, consistent use of log levels will increase the debuggability of your application.

The logging methods and their common uses are these (shown for the base logger, but work the same for sub-loggers):

1. application.logger.severe: "Severe" messages are used for unrecoverable errors, alerts about unavailable computing resources, network backends, etc. You'd always want to configure logging to include these messages!
2. application.logger.warning: While not quite severe, the event could still point out a problem that, if left un-handled, might become severe, either now or in the future. Many applications emit copious warnings that can be safely ignored.
3. application.logger.info: These don't report a problem, but instead are used to mark an occurrence of an event. Useful for monitoring and high-level debugging.
4. application.logger.config: Treat these as components of an event that together would constitute a single "info" message. They are meant to show how the event was initialized or released. Useful for low-level debugging of events.
5. application.logger.fine: General purpose, low-level debugging.
6. application.logger.finer: Even lower!
7. application.logger.finest: Lowest of the low!

/configuration/logging.conf

We'll cover the basics here. See log4j documentation for more information.

> The Prudence defaults are mostly at "info" level. You are encouraged to experiment with lower levels in order to see how Prudence's internals function!

Appenders

An "appender" is the service that actually writes log messages. Appenders are configured with properties prefixed with "log4j.appender.X", where X is the name of the appender.

Prudence by default uses two rolling file appenders, one called "web" for web.log, and one called "prudence" for prudence.log. Additionally, a console appender named "console" and a remote appender named "remote" are configured, though they are not used by default.

Loggers

A "logger" is the service to which you send your log messages. It decides whether to write the message according to its level, and if so sends it to one or more appenders. Loggers are configured with properties prefixed with "log4j.logger.X", where X is the name of the logger. Logger names appear in the logs for every message, and are thus useful for organization your log.

You do not have to define all your loggers in logging.conf. Any logger name can be used by your application. If it is not found in logging.conf, then default attributes are inherited. Inheritence works by treating logger names hierarchically: if you do not specify a certain attribute for a logger, then its parent logger is used. The nameless root logger, configured with the "log4j.rootLogger" prefix, defines defaults for all loggers.

You can configure your application's logger and sub-loggers. For example, if your application is named "wackywiki," you can set its maximum logging level thus:

```
log4j.logger.wackywiki=WARN
```

Log levels in logging.conf are named a bit differently from the commands used in application.logger:

1. application.logger.severe: ERROR

2. application.logger.warning: WARN
3. application.logger.info: INFO
4. application.logger.config: DEBUG
5. application.logger.fine: DEBUG
6. application.logger.finer: DEBUG
7. application.logger.finest: TRACE (or ALL)

   The differences are due to the preponderance of logging solutions for the JVM, which are used in some of Prudence's underlying libraries. We hope to streamline this further in a future version of Prudence.

You can also add or change appenders for your loggers. For example, to send wackywiki messages to the console appender:

```
log4j.logger.wackywiki=WARN, console
```

Separate Logs Per Application

We'll create a new appender for each new log file we need. In this example, we'll just copy the "prudence" appender with a new name:

```
log4j.appender.wackywiki=org.apache.log4j.RollingFileAppender
log4j.appender.wackywiki.File=logs/wackywiki.log
log4j.appender.wackywiki.MaxFileSize=5MB
log4j.appender.wackywiki.MaxBackupIndex=9
log4j.appender.wackywiki.layout=org.apache.log4j.PatternLayout
log4j.appender.wackywiki.layout.ConversionPattern=%d: %-5p [%c] %m%n
```

Then, we'll direct our application logger to use this appender:

```
log4j.logger.wackywiki=WARN, wackywiki
```

Analyzing /logs/web.log

You can throw Prudence's web.log into almost any Apache log file analyzer. Here's an example using the ubiquitous Analog:

```
analog \
-C'LOGFORMAT (%Y-%m-%d\t%h:%n:%j\t%S\t%u\t%j\t%j\t%j\t%r\t%q\t%c\t%b\t%j\t%T\t%v\t%
    B\t%f)' \
-C'LOCALCHARTDIR local/images/' \
-C'CHARTDIR images/' \
-C'HOSTNAME "mysite.org"' \
logs/web.log \
-Oapplications/myapp/web/static/analog/index.html
```

Administration

Prudence comes with the "Prudence Administration Application." As of Prudence 1.1, this is a simple application that lets you see the servers, virtual hosts and applications in the Prudence instance, and lets you start and stop them. Future versions of Prudence will build on this foundation, allowing for more runtime control, configuration and monitoring.

Installation

Prudence comes with this application installed as "prudence-admin," bound to the default virtual host at the root URL.

Customization

Changing the Root URL

Let's say we want prudence-admin at the "/admin/" URL on the default host, and the "/prudence/admin/" URL on myHost. We'll edit its "settings.*". Here's an example for the JavaScript flavor:

```
hosts = [[component.defaultHost, '/admin/'], [myHost, '/prudence/admin/']]
```

Requiring a Password

As an example, we'll route the application's root router through an HTTP authentication filter (page 53).

We'll create a "routing.js" file (for the JavaScript flavor):

```
// Implement defaults
document.execute('/defaults/application/routing/')

importClass(
        org.restlet.security.ChallengeAuthenticator,
        org.restlet.security.MapVerifier,
        org.restlet.data.ChallengeScheme);

// Create an authenticator
var verifier = new MapVerifier()
verifier.localSecrets.put('admin', new java.lang.String('opensesame').toCharArray()
    )
var authenticator = new ChallengeAuthenticator(applicationInstance.context,
        ChallengeScheme.HTTP_BASIC, 'Prudence Administration')
authenticator.verifier = verifier

// Put authenticator before root
authenticator.next = applicationInstance.inboundRoot
applicationInstance.inboundRoot = authenticator
```

Prudence As a Daemon

Prudence comes with built-in support Apache Commons Daemon. However, in production environments, you might prefer to use one of the daemon *wrappers*, which provide process monitoring and control. For example, if Prudence's JVM crashes for some reason, hangs, grabs too much CPU or RAM, the wrapper can automatically shut it down and even restart it.

Though Prudence comes with the necessary configuration files for wrappers, it does not include the actual wrappers, which you will need to install separately. The reason we haven't included them is different for each wrapper: Tanuki's JSW uses a restrictive license (GPL) that will not allow us to distribute it with Prudence, and YAJSW is just far too big.

Apache Commons Daemon

The /bin/ scripts (page 20) allow you to "start", "stop", "restart" and check the running "status" for the Prudence daemon (called a "service" in Windows). The daemon will write its log to "/logs/run.log".

Though you need root privileges (sudo) to start or stop the daemon, the daemon itself will attempt to run under user "prudence-X", where "X" is the name of the flavor you are using. For example, "prudence-python" is used for the Python flavor. If that user doesn't exist, the daemon will run under *your* user.

In any case, it's up to you to make sure that the daemon's user has the necessary read and write privileges for the Prudence files. In particular, it needs to be able to write to /logs/ and to /cache/.

The scripts work by detecting your running environment and attempting to use the appropriate Apache Commons Daemon binary. Prudence comes with binaries for Linux, Solaris, Darwin (OS X) and Windows, for both 32bit and 64bit architectures.

Notes for Linux

- We maintain a repository for Ubuntu where everything just works.
- The /bin/run.sh script can be installed as an /etc/init.d/ script, so it can also be run via Upstart, like so:

```
sudo service prudence-X start
```

- If you get an error about a missing JAVA_HOME, you can set it like so:

```
sudo JAVA_HOME=/var/lib/jvm bin/run.sh start
```

- Newer Linux operating systems might give you an error about a missing "libcap.so.1" library. In such cases, you will need to install your operating system's Apache Commons Daemon package. In Ubuntu, for example, it is package "jsvc". The /bin/run.sh script will always try to use your operating system's jsvc before falling back to the binary that comes with Prudence.

Notes for Windows

- You need to "install" the service before you can "start" it. Use "uninstall" if you need to remove it.
- Note that Windows requires your service executable (in the /commons-daemon/ subdirectory) to be on a local drive. It will fail on network drives.
- Prudence is installed as a standard Windows service, so you can also use Control Panel > Administrative Tools > Services or the "net" command line tool to start and stop it. For example:

```
net start PrudenceX
```

- You can edit the settings using "settings".
- Windows does not support "status", but you can use "monitor" to add a nice little tray icon that lets you manage the daemon.

JSW

JSW is written in C rather than Java, making it much more lightweight than YAJSW. It supports many operating systems.

See /configuration/wrapper.conf for a sample JSW configuration, which also configures the wrapper to log to /logs/wrapper.log.

In particular, you'll want to set the following configuration settings:

- wrapper.working.dir: Set this to your Prudence installation's base directory
- wrapper.java.command: Set this to the JVM runtime (the "java" command) you'll want to use for Prudence
- wrapper.java.maxmemory: Set this according to your deployment environment. More memory translates to better performance, and more room for the in-process memory cache.
- wrapper.java.initmemory: A reasonably high value here can help speed up Prudence's startup time.

The copyright for JSW is held by its original developer, Tanuki Software. For the first few versions Tanuki released JSW under a permissive license, making it popular in many open source projects. However, since version 3.2 it has been distributed under the GPL 2.0 (and also via a commercial license). We applaud Tanuki's commitment to open source, and are big fans of the GPL. However, the GPL makes it impossible to distribute JSW with open source projects using less restrictive licenses, such as Prudence. Many projects have kept distributing version 3.2 of JSW, which is now out of date and missing bug fixes. Note that Prudence's license, the LGPL 3.0, has a clause allowing you to upgrade it to GPL, which would allow you to distribute JSW with it. Doing this would in turn require you to distribute you own product under the GPL.

YAJSW

YAJSW is written in 100% Java, using JNA to handle the native operating-system-dependent features.
See /configuration/yajsw.conf for a sample JAJSW configuration. The wrapper will log to /logs/yajsw.log. Change yasjw.conf in the same way as described for wrapper.conf, above.

HTTP Proxy

There's nothing special about how Prudence handles HTTP, and it can work easily behind any reverse proxy. This lets you easily unite Prudence with other web servers or run it behind a load balancer. Though it's not unique to Prudence, we thought to add this section to the manual in order to get you up and running quickly with this useful scenario.

Perlbal

You can run many instances of Prudence behind a load balancer. This offers fault tolerance, maintenance options, and the possibility of dramatically scaling up the number of requests you can support. Your application can tolerate failure of any number of instances, as long as you have one running, because load balancers will automatically route to working instances. Similarly, load balancing allows you to bring some instances down for maintenance while keeping your application up and running.

Scaling up can be straightforward: simply add more and more instances behind the load balancer, which will make sure to distribute requests among them, while monitoring their responsiveness to accommodate for how well they handle their load. More complex systems can involve different kinds of instances, with the load balancer being in charge of routing requests to the appropriate pool of instances. This "partitioning" can be according to features (one pool handles chat room, one pool handles file downloads), geography (one pool handles England, one pool handles France), or other clever ways to keep the whole system efficient and responsive. See Scaling Tips (page 93) for an in-depth treatment.

There are many great load balancers out there, but we especially like Perlbal. Here's an example perlbal.conf in which we use Perlbal to handle secure connections for us. See secure servers (page 24) on how to configure Prudence to handle secure connections directly.

```
CREATE POOL pool
  SET nodefile = /etc/perlbal/nodes

CREATE POOL secure_pool
  SET nodefile = /etc/perlbal/secure_nodes

# HTTP
CREATE SERVICE balancer
  SET listen          = 0.0.0.0:80
  SET role            = reverse_proxy
  SET pool            = pool
  SET verify_backend  = on

# HTTPS
CREATE SERVICE secure_balancer
```

```
  SET listen          = 0.0.0.0:443
  SET role            = reverse_proxy
  SET pool            = secure_pool
  SET verify_backend  = on
  SET enable_ssl      = on
  SET ssl_key_file    = /etc/perlbal/server.key
  SET ssl_cert_file   = /etc/perlbal/server.crt
  # This is recommended to workaround a bug in older versions of IE
  # (the default is ALL:!LOW:!EXP)
  SET ssl_cipher_list = ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+
      eNULL

# Internal management port
CREATE SERVICE mgmt
  SET role    = management
  SET listen = 127.0.0.1:60000

ENABLE balancer
ENABLE secure_balancer
ENABLE mgmt
```

The nodes file is a list of IP addresses (not hostnames!) with ports. We'll add three Prudence instances running at the default server port:

```
192.168.1.10:8080
192.168.1.11:8080
192.168.1.12:8080
```

The secure_nodes file is the same for SSL connections. Let's have our Prudence instances run a separate server at port 8081 for secure requests:

```
192.168.1.10:8081
192.168.1.11:8081
192.168.1.12:8081
```

In our Prudence instances, we'll use "/instance/servers.*" (page 24) to create a server at port 8081 (JavaScript flavor):

```
document.execute('/defaults/instance/servers/')

var secureServer = new Server(Protocol.HTTP, 8081)
secureServer.name = 'secure'
component.servers.add(secureServer)
```

And we'll use "/instance/hosts.*" (page 23) to create separate virtual hosts for each port:

```
importClass(org.restlet.routing.VirtualHost)

var host = new VirtualHost(component.context)
host.name = 'default'
host.hostPort = '8080'
component.hosts.add(host)

var secureHost = new VirtualHost(component.context)
secureHost.name = 'secure'
secureHost.hostPort = '8081'
component.hosts.add(secureHost)

component.defaultHost = host
```

By default, an application would bind to the default host. To bind an application to the secure virtual host, set the hosts settings (page 28) in its /settings.*:

```
hosts = [[secureHost, '/myapp/']]
```

Apache

Apache's HTTP server is often called the "Swiss army knife of the Internet" for how well it manipulates URLs and routes HTTP requests. Prudence already does powerful URI-based routing (page), including virtual hosting (page), meaning that you probably won't need Apache for that.

Where you might want to use Apache is as a container environment for other application platforms, such as mod_php and mod_wsgi. If you have no choice but to run Apache as your front end, it is straightforward to set it to route to Prudence via reverse proxy.

Here's a sample Apache configuration that proxies all URLs beginning with "/prudence/" to a local Prudence instance running at the default server port. For demonstration purposes, we'll also explicitly block proxying of "/prudence/images/" URLs so that Apache would handle serving them (from directory /var/www/prudence/images) instead of Prudence. Though, note that Prudence can serve static files just fine (page). Apache's mod_proxy and mod_proxy_http must be enabled for this configuration to work:

```
<VirtualHost *:80>
        DocumentRoot /var/www

        <Proxy *>
                Order deny,allow
                Allow from all
        </Proxy>

        ProxyRequests Off
        ProxyPreserveHost On
        ProxyStatus On

        # Note that the order of ProxyPass statements matters
        ProxyPass /prudence/images/ !
        ProxyPass /prudence/ http://localhost:8080/ retry=0
</VirtualHost>
```

Prudence As a Restlet Container

Why use Prudence for a Restlet application that already works?

Though applications can be written in Prudence without a single line of Java code, Prudence also acts as a useful container for existing Restlet applications, Restlet resources or restlets written in Java.

Prudence makes it easy to handle the bootstrapping and routing (page 27) of your applications, and the Administration Application (page 86), debugging (page 83) and logging (page 84) features make it easier to deploy and manage multiple applications together. These issues have more to do with your application's configuration, rather than its functionality, and it can be useful to handle them outside of the Java build process, using live, dynamic languages in simple text source that you can modify on the fly. Deploying your Restlet application to a Prudence instance can be as simple as copying a zip file to a Prudence container.

This need is also fulfilled by servlet and Java Enterprise Edition (JEE) containers, such as Tomcat, Resin and JBoss. Indeed, Restlet has a JEE edition, and good support for servlets. However, if all you need is a deployment container, Prudence can serve as a straightforward, pure REST alternative to JEE.

Some people also look to JEE containers for their support of Java Server Pages (JSP). We urge you to take a good look at Prudence's dynamic web (page 31) support. It may surpass JSP for your purposes. In particular, it is based on Restlet, which you already know and love, giving you the entire Restlet API at your fingertips. It also lets you use many wonderful languages other than Java for scriptlets. For simpler templating, Velocity and Succinct are also built in and immediately ready to use.

Summary

A 100% Restlet-based alternative to servlet/JEE containers. (Requires only Restlet JSE.)

1. Easy deployment
    (a) Configuration scripts (page 21): avoid weird (XML) configuration formats and start your Restlet component exactly as you want (your choice from six languages)
    (b) The default scripts already handle virtual hosting (page 23), multiple servers (page 24) and internal routing (page 46)
    (c) Designed from the ground-up to handle multiple applications (page 25) on the same component
    (d) Admin application (page 86) for live management of components
    (e) Logging is pre-configured (page 84) and "just works," including an Apache-compatible web log (page 86)
    (f) Single zip-file application deployment (page 25) (like WAR files in JEE)
2. Easy prototyping of REST resources (page 40)
    (a) Your choice among 6 languages (page 9)
    (b) Code is compiled, cached and loaded on the fly (page 40)
    (c) Rich debug page (page 83) shows errors and source code (page 84)
    (d) When you're happy with it, you can rewrite it as a ServerResource in Java
3. Powerful HTML generation platform (page 31), like JSP/ASP/PHP (again, 100% Restlet-based)
    (a) Your choice from six languages, including mixing languages (page 35) and templating engines (page 36) (Velocity, Succinct) on one page
    (b) Code is compiled, cached and loaded on the fly (page 37)
    (c) RAM/database/Hazelcast/memcached server-side caching (page 69), using Restlet's URI templating language for cache key generation (page 68)
    (d) Straightforward support for client-side caching (page 51)
    (e) Asynchronous processing (page 80)
    (f) Easily accept uploaded files (page 75)
    (g) Rich debug page (page 83) shows errors and source code (page 84)
4. Restlet sugar (also available as a standalone JAR)
    (a) Fallback router (attach multiple MODE_STARTS_WITH restlets to the same base URI)
    (b) URI "capturing" (page 56)
    (c) JavaScript and CSS unify-and-minify filters (page 52)
    (d) Delegated status service for diverting to custom pages (404, 500 errors, etc.)
    (e) Rich DebugRepresentation
    (f) Cache backend abstraction, designed for storing StringRepresentations
    (g) Easier file uploads (slightly higher-level than the Restlet FileUpload extension)
    (h) ConversationCookie (combines Cookie and CookieSetting)
    (i) Filter to selectively add CacheControl (to Directory, for example)

Custom Resources and Restlets

Use your application's "routing.*" (page 27) to attach your resources, or otherwise manage routing. Example (JavaScript flavor):

```
// Prudence defaults
document.execute('/defaults/application/routing/')

// MyOrg resources
router.attach('/data/item/{id}', classLoader.loadClass('org.myorg.ItemResource'))
router.attach('/data/items', classLoader.loadClass('org.myorg.ItemsResource'))
```

You can also change Prudence's default routing by detaching and re-attaching routes:

```
importClass(com.threecrickets.prudence.util.JavaScriptUnifyMinifyFilter)

router.detach(staticWeb)
router.attachBase(
        staticWebBaseURL,
        new JavaScriptUnifyMinifyFilter(application.context, staticWeb, new File(
            applicationBasePath + staticWebBasePath),
                10000))
```

Custom Application

By default, Prudence creates an instance of the standard Restlet Application class. Use your application's "application.*" (page 27) to override this, and create and configure your own application. Example:

```
// MyOrgApplication
importClass(org.myorg.MyOrgApplication)
var application = new MyOrgApplication()

// Install our custom tunnel service
importClass(org.myorg.MyOrgTunnelService)
application.tunnelService = new MyOrgTunnelService(MyOrgTunnelService.MODE_QUERY)

// These attributes are specific to the MyOrgApplication class
application.databaseURI = 'mysql://localhost/myorg'
application.useTransactions = true
```

Scaling Tips

Scalability is the ability to respond to a growing number of user requests without degradation in response time. Two variables influence it: 1) your total number of threads and 2) the time it takes each thread to process a request. Increasing the number of threads seems straightforward: you can keep adding more machines behind load balancers. However, the two variables are tied, as there are diminishing returns and even reversals: beyond a certain point, time per request can actually grow longer as you add threads and machines.

Let's ignore the first variable here, because the challenge of getting more machines is mostly financial. It's the second that you can do something about as an engineer.

If you want your application to handle many concurrent users, then you're fighting this fact: a request will get queued in the best case or discarded in the worst case if there is no thread available to serve it. Your challenge is to make sure that a thread is always available. And it's not easy, as you'll find out as you read through this article. Minimizing the time per request becomes an architectural challenge that encompasses the entire structure of your application

Performance Does Not Equal Scalability

Performance does not equal scalability. Performance does not equal scalability. Performance does not equal scalability.

Get it? Performance does not equal scalability.

This is an important mantra for two reasons:

1. Performant Can Mean Less Scalable

Optimizing for performance can adversely affect your scalability. The reason is contextual: when you optimize for performance, you often work in an isolated context, specifically so you can accurately measure response times and fine-tune them. For example, making sure that a specific SQL query is fast would involve just running that query. A full-blown experiment involving millions of users doing various operations on your application would make it very hard to accurately measure and optimize the query. Unfortunately, by working in an isolated context you cannot easily see how your efforts would affect other parts of an application. To do so would require a lot of experience and imagination. To continue our example, in order to optimize your one SQL query you might create an index. That index might need to be synchronized with many servers in your cluster. And that synchronization overhead, in turn, could seriously affect your ability to scale. Congratulations! You've made one query run fast in a situation that never happens in real life, and you've brought your web site to a halt.

One way to try to get around this is to fake scale. Tools such as JMeter, Siege and ApacheBench can create "load." They also create unfounded confidence in engineers. If you simulate 10,000 users bombarding a single web page, then you're, as before, working in an isolated context. All you've done is add concurrency to your performance optimization measurements. Your application pathways might work optimally in these situations, but this might very well be due to the fact that the system is not doing anything else. Add those "other" operations in, and you might get worse site capacity than you did before "optimizing."

2. Wasted Effort

Even if you don't adversely affect your scalability through optimizing for performance, you might be making no gains, either. No harm done? Well, plenty of harm, maybe. Optimizing for performance might waste a lot of development time and money. This effort would be better spent on work that could actually help scalability.

And, perhaps more seriously, it demonstrates a fundamental misunderstanding of the problem field. If you don't know what your problems are, you'll never be able to solve them.

Pitfalls

Study the problem field carefully. Understand the challenges and potential pitfalls. You don't have to apply every single scalability strategy up-front, but at least make sure you're not making a fatal mistake, such as binding yourself strongly to a technology or product with poor scalability. A bad decision can mean that when you need to scale up in the future, no amount of money and engineering effort would be able to save you before you lose customers and tarnish your brand.

Moreover, be very careful of blindly applying "successful" strategies used and recommended by others to your product. What worked for them might not work for you. In fact, there's a chance that their strategy doesn't even work for them, and they just think it did because of a combination of seemingly unrelated factors. The realm of web scalability is still young, full of guesswork, intuition and magical thinking. Even the experts are often making it up as they're going along.

Generally, be very suspicious of products or technologies being touted as "faster" than others. *"Fast" doesn't say anything about the ability to scale.* Is a certain database engine "fast"? That's important for certain applications, no doubt. But maybe the database is missing important clustering features, such that it would be a poor choice for scalable applications. Does a certain programming language execute faster than another? That's great if you're doing video compression, but speed of execution might not have have noticeable effect on scalability. Web applications mostly do I/O, not computation. The same web application might have very similar performance characteristics whether it's written in C++ or PHP.

Moreover, if the faster language is difficult to work with, has poor debugging tools, limited integration with web technologies, then it would slow down your work and your ability to scale.

> Speed of execution can actually help scalability in its financial aspect: If your application servers are constantly at maximum CPU load, then a faster execution platform would let you cram more web threads into each server. This will help you reduce costs. For example, see Facebook's HipHop: they saved millions by translating their PHP code to C. Because Prudence is built on the fast JVM platform, you're in good hands in this respect. Note, however, that there's a potential pitfall to high performance: more threads threads per machine would also mean more RAM requirements per machine, which also costs money. Crunch the numbers and make sure that you're actually saving money by increasing performance. Once again, performance does not equal scalability.

That last point about programming languages is worth some elaboration. Beyond how well your chosen technologies perform, it's important to evaluate them in terms to how easy they are to manage. Large web sites are large projects, involving large teams of people and large amounts of money. That's difficult enough to coordinate. You want the technology to present you with as few extra managerial challenges as possible.

Beware especially of languages and platforms described as "agile," as if they somehow embody the spirit of the popular Agile Manifesto. Often, "agile" seems to emphasize the following features: forgiveness for syntax slips, light or no type checking, automatic memory management and automatic concurrency—all features that seem to speed up development, but could just as well be used for sloppy, error-prone, hard-to-debug, and hard-to-fix code, slowing down development in the long run. If you're reading this article, then your goal is likely not to create a quick demo, but a stable application with a long, evolving life span.

Ignore the buzzwords ("productivity", "fast"), and instead make sure you're choosing technology that you can control, instead of technology that will control you.

We discuss this topic some more in "The Case for Rest" (page 5). By building on the existing web infrastructure, Prudence can make large Internet projects easier to manage.

Analysis

Be especially careful of applying a solution before you know if you even have a problem.

How to identify your scalability bottlenecks? You can create simulations and measurements of scalability rather than performance. You need to model actual user behavior patterns, allow for a diversity of such behaviors to happen concurrently, and replicate this diversity on a massive scale.

Creating such a simulation is a difficult and expensive, as is monitoring and interpreting the results and identifying potential bottlenecks. This is the main reason for the lack of good data and good judgment about how to scale. Most of what we know comes from tweaking real live web sites, which either comes at the expense of user experience, or allows for very limited experimentation. Your best bet is to hire a team who's already been through this before.

Optimizing for Scalability

In summary, your architectural objective is to increase concurrency, not necessarily performance. Optimizing for concurrency means breaking up tasks into as many pieces as possible, and possibly even breaking requests into smaller pieces. We'll cover numerous strategies here, from frontend to backend. Meanwhile, feel free to frame these inspirational slogans on your wall:

> Requests are hot potatoes: Pass them on!

And:

> It's better to have many short requests than one long one.

Caching

Retrieving from a cache can be orders of magnitude faster than dynamically processing a request. It's your most powerful tool for increasing concurrency.

Caching, however, is only effective is there's something in the cache. It's pointless to cache fragments that appear only to one user on only one page that they won't return to. On the other hand, there may very well be fragments on the page that will recur often. If you design your page carefully to allow for fragmentation, you will reap the benefits of fine-grained caching. Remember, though, that the outermost fragment's expiration defines the expiration of the included fragments. It's thus good practice to define no caching on the page itself, and only to cache fragments.

In your plan for fine-grained caching, take special care to isolate those fragments that cannot be cached, and cache everything around them.

Make sure to change Prudence's cache key (page 68) to fit the lowest common denominator: you want as many possible requests to use the already-cached data, rather than generating new data. Note that, by default, Prudence includes the request URI in the cache key . Fragments, though, may very well appear identically in many different URIs. You would thus not want the URI as part of their cache key.

Cache aggressively, but also take cache validation seriously. Make good use of Prudence's cache tags (page 69) to allow you to invalidate portions of the cache that should be updated as data changes. Note, though, that every time you invalidate you will lose caching benefits. If possible, make sure that your cache tags don't cover too many pages. Invalidate only those entries that really need to be invalidated.

(It's sad that many popular web sites do cache validation so poorly. Users have come to expect that sometimes they see wrong, outdated data on a page, sometimes mixed with up-to-date data. The problem is usually solved within minutes, or after a few browser refreshes, but please do strive for a better user experience in your web site!)

If you're using a deferred task handler (page 99), you might want to invalidate tagged cache entries when tasks are done. Consider creating a special internal API that lets the task handler call back to your application to do this.

How long should you cache? As long as the user can bear! In a perfect world, of limitless computing resources, all pages would always be generated freshly per request. In a great many cases, however, there is no harm at all if users see some data that's a few hours or a few days old.

> Note that even very small cache durations can make a big difference in application stability. Consider it the maximum throttle for load. For example, a huge sudden peak of user load, or even a denial-of-service (DOS) attack, might overrun your thread pool. However, a cache duration of just 1 second would mean that your page would never be generated more than once every second. You are instantly protected against a destructive scenario.

Cache Warming

Caches work best when they are "warm," meaning that they are full of data ready to be retrieved.

A "cold" cache is not only useless, but it can also lead indirectly to a serious problem. If your site has been optimized for a warm cache, starting from cold could significantly strain your performance, as your application servers struggle to generate all pages and fragments from scratch. Users would be getting slow response times until the cache is significantly warm. Worse, your system could crash under the sudden extra load.

There are two strategies to deal with cold caches. The first is to allow your cache to be persistent, so that if you restart the cache system it retains the same warmth it had before. This happens automatically with database-backed caches (page 98). The second strategy is to deliberately warm up the cache in preparation for user requests.

Consider creating a special external process or processes to do so. Here are some tips:

1. Consider mechanisms to make sure that your warmer does not overload your system or take too much bandwidth from actual users. The best warmers are adaptive, changing their load according to what the servers can handle. Otherwise, consider shutting down your site for a certain amount of time until the cache is sufficiently warm.

2. If the scope is very large, you will have to pick and choose which pages to warm up. You would want to choose only the most popular pages, in which case you might need a system to record and measure popularity. For example, for a blog, it's not enough just to warm up, say, the last two weeks of blog posts, because a blog post from a year ago might be very popular at the moment. Effective warming would require you to find out how many times certain blog posts were hit in the past two weeks. It might make sense to embed this auditing ability into the cache backend itself.

Pre-Filling the Cache

If there are thousands of ways in which users can organize a data view, and each of these views is particular to one user, then it may make little sense to cache them individually, because individual schemes would hardly ever be re-used. You'll just be filling up the cache with useless entries.

Take a closer look, though:

1. It may be that of the thousands of organization schemes only a few are commonly used, so it's worth caching the output of just those.
2. It could be that these schemes are similar enough to each other that you could generate them all in one operation, and save them each separately in the cache. Even if cache entries will barely be used, if they're cheap to create, it still might be worth creating them.

This leads us to an important point:

> Prudence is a "frontend" platform, in that it does not specify which data backend, if at all, you should use. Its cache, however, is general purpose, and you can store in it anything that you can encode as a string.

Let's take as a pre-filling example a tree data structure in which branches can be visually opened and closed. Additionally, according to user permissions different parts of the tree may be hidden. Sounds too complicated to cache all the view combinations? Well, consider that you can trigger, upon any change to the tree data structure, a function that loops through all the different iterations of the tree recursively and saves a view of each of them to the cache. The cache keys can be something like "branch1+.branch2-.branch3+", with "+" signifying "-" whether the branch is visually open or closed. You can use similar +'s and -'s for permissions, and create views per permission combinations. Later, when users with specific permissions request different views of the tree, no problem: all possibilities were already pre-filled. You might end up having to generate and cache thousands of views at once, but the difference between generating one view and generating thousands of views may be quite small, because the majority of the duration is spend communicating with the database backend.

If generating thousands of views takes too long for the duration of a single request, another option is to generate them on a separate thread. Even if it takes a few minutes to generate all the many, many tree views combinations, it might be OK in your application for views to be a few minutes out-of-date. Consider that the scalability benefits can be very significant: you generate views only *once* for the entire system, while millions of concurrent users do a simple retrieval from the cache.

Caching the Data Backend

Pre-filling the cache can take you very far. It is, however, quite complicated to implement, and can be ineffective if data changes too frequently or if the cache has to constantly be updated. Also, it's hard to scale the pre-filling to *millions* of fragments.

If we go back to our tree example above, the problem was that it was too costly to fetch the entire tree from the database. But what if we cache the tree itself? In that case, it would be very quick to generate any view of the tree on-demand. Instead of caching the view, we'd be caching the data, and achieving the same scalability gains.

Easy, right? So why not cache *all* our data structures? The reason is that it's very difficult to do this correctly beyond trivial examples. Data structures tend to have complex interrelationships (one-to-many, many-to-many, foreign keys, recursive tree structures, graphs, etc.) such that a change in data at one point of the structure may alter various others in particular ways. For example, consider a calendar database, and that you're caching individual days with all their events. Weekly calendar views are then generated on the fly (and quickly) for users according to what kinds of events they want to see in their personal calendars. What happens if a user adds a recurring event that happens every Monday? You'll need to make sure that all Mondays currently cached would be invalidated, which might mean tagging all these as "monday" using Prudence's cache tags. This requires a specific caching strategy for a specific application.

By all means, cache your data structures if you can't easily cache your output, but be aware of the challenge!

> Prudence's sister project, Diligence, is designed specifically to solve this problem. It not only caches your data structures, but it validates them in memory using your coded logic, instead of invalidating them and forcing them to be re-fetched from the database. It supports data structures commonly used with relational databases, pluggable storage technologies, high-performance resource pooling and throttling, and natural integration with Prudence. Together, Diligence and Prudence form a solid platform for building scalable, data-backed web applications. At the time of this writing, Diligence is still under development. We hope to release it as open source soon, so stay tuned!

Cache Backends

Your cache backend can become a bottleneck to scalability if 1) it can't handle the amount of data you are storing, or 2) it can't respond quickly enough to cache fetching.

Before you start worrying about this, consider that it's a rare problem to have. Even if you are caching millions of pages and fragments, a simple relational-database-backed cache, such as Prudence's SqlCache implementations, could handle this just fine. A key/value table is the most trivial workload for relational databases, and it's also easy to shard (page 102). Relational database are usually very good at caching these tables in their memory and responding optimally to read requests. Prudence even lets you chain caches together to create tiers: an in-process memory cache in front of a SQL cache would ensure that many requests don't even reach the SQL backend.

High concurrency can also be handled very well by this solution. Despite any limits to the number of concurrent connections you can maintain to the database, each request is handled very quickly, and it would require *very* high loads to saturate. The math is straightforward: with a 10ms average retrieval time (very pessimistic!) and a maximum of 10 concurrent database connections (again, pessimistic!) you can handle 1,000 cache hits per second. A real environment would likely provide results orders of magnitude better.

The nice thing about this solution is that it uses the infrastructure you already have: the database.

But, what if you need to handle *millions* of cache hits per second? First, let us congratulate you for your global popularity. Second, there is a simple solution: distributed memory caches. Prudence comes with Hazelcast and support for memcached, which both offer much better scalability than database backends. Because the cache is in memory, you lose the ability to easily persist your cache and keep it warm: restarting your cache nodes will effectively reset them. There are workarounds—for example, parts of the cache can be persisted to a second database-backed cache tier—but this is a significant feature to lose.

> Actually, Hazelcast offers fail-safe, live backups. While it's not quite as permanent as a database, it might be good enough for your needs. And memcached has various plugins that allow for real database persistence, though using them would require you to deal with the scalability challenges of database backends (page 103).

You'll see many web frameworks out there that support a distributed memory cache (usually memcached) and recommend you use it ("it's fast!" they claim, except that it can be slower per request than optimized databases, and that anyway performance does not equal scalability). We'd urge you to consider that advice carefully: keeping your cache warm is a challenge made much easier if you can store it in a persistent backend, and database backends can take you very far in scale without adding a new infrastructure to your deployment. It's good to know, though, that Prudence's support for Hazelcast and memcached is there to help you in case you reach the popularity levels of LiveJournal, Facebook, YouTube, Twitter, etc.

Client-Side Caching

Modern web browsers support client-side caching, a feature meant to improve the user experience and save bandwidth costs. A site that makes good use of client-side caching will appear to work fast for users, and will also help to increase your site's popularity index with search engines.

Optimizing the user experience is not the topic of this article: our job here is to make sure your site doesn't degrade its performance as load increases. However, client-side caching can indirectly help you scale by reducing the number of hits you have to take in order for your application to work.

> Actually, doing a poor job with client-side caching can help you scale: users will hate your site and stop using it—voila, less hits you have to deal with. OK, that was a joke!

Generally, Prudence handles client-side caching automatically. If you cache a page, then headers will be set to ask the client to cache for the same length of time. By default, conditional mode is used: every time the client tries to view a page, it will make a request to make sure that nothing has changed since their last request to the page. In case nothing has changed, no content is returned.

You can also turn on "offline caching" mode, in which the client will avoid even that quick request. Why not enable offline caching by default? Because it involves some risk: if you ask to cache a page for one week, but then find out that you have a mistake in your application, then users will not see any fix you publish until their local cache expires, which can take up to a week! It's important that you you understand the implications before using this mode. See the dynamicWebClientCachingMode application setting (page 29).

It's generally safer to apply offline caching to your static resources, such as graphics and other resources. A general custom is to ask the client to cache these "forever" (10 years), and then, if you need to update a file, you simply create a new one with a new URL, and have all your HTML refer to the new version. Because clients cache according to URL, their cached for the old version will simply not be ignored. See CacheControlFilter (page 51). There, you'll also see some more tricks Prudence offers you to help optimize the user experience, such as unifying/minimizing client-side JavaScript and CSS.

Upstream Caching

If you need to quickly scale a web site that has not been designed for caching, a band-aid is available: upstream caches, such as Varnish, NCache and even Squid. For archaic reasons, these are called "reverse proxy" caches, but they really work more like filters. According to attributes in the user request (URL, cookies, etc.), they decide whether to fetch and send a cached version of the response, or to allow the request to continue to your application servers.

The crucial use case is archaic, too. If you're using an old web framework in which you cannot implement caching logic yourself, or cannot plug in to a good cache backend, then these upstream caches can do it for you.

They are problematic in two ways:

1. Decoupling caching logic from your application means losing many features. For example, invalidating portions of the cache is difficult if not impossible. It's because of upstream caching, indeed, that so many web sites do a poor job at showing up-to-date information.

2. Filtering actually implements a kind of partitioning, but one that is vertical rather than horizontal. In horizontal partitioning, a "switch" decides to send requests to one cluster of servers or another. Within each cluster, you can control capacity and scale. But in vertical partitioning, the "filter" handles requests internally. Not only is the "filter" more complex and vulnerable than a "switch" as a frontend connector to the world, but you've also complicated your ability to control the capacity of the caching layer. It's embedded inside your frontend, rather than being another cluster of servers. We'll delve into backend partitioning (page 101) below.

Unfortunately, there is a use case relevant for newer web frameworks, too: if you've designed your application poorly, and you have many requests that could take a long time to complete, then your thread pools could get saturated when many users are concurrently making those requests. When saturated, you cannot handle even the super-quick cache requests. An upstream cache band-aid could, at least, keep serving its cached pages, even though your application servers are at full capacity. This creates an illusion of scalability: some users will see your web site behaving fine, while others will see it hanging.

The real solution would be to re-factor your application so that it does not have long requests, guaranteeing that you're never too saturated to handle tiny requests. Below are tips on how to do this.

Dealing with Lengthy Requests

One size does not fit all: you will want to use different strategies to deal with different kinds of tasks.

Deferrable Tasks

Deferrable tasks are tasks that can be resolved later, without impeding on the user's ability to continue using the application.

If the deferrable task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service. Prudence's tasks (page 49) implementation is a great solution for deferrable tasks, as it lets you run tasks on other threads or even distribute them in a Hazelcast cluster.

Deferring tasks does present a challenge to the user experience: What do you do if the task fails and the user needs to know about it? One solution can be to send a warning email or other kind of message to the user. Another solution could be to have your client constantly poll in the background (via "AJAX") to see if there are any error messages, which in turn might require you to keep a queue of such error messages per user.

Before you decide on deferring a task, think carefully of the user experience: for example, users might be constantly refreshing a web page waiting to see the results of their operation. Perhaps the task you thought you can defer should actually be considered necessary (see below)?

Necessary Tasks

Necessary tasks are tasks that must be resolved before the user can continue using the application.

If the necessary task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service and return a "please wait" page to the user.

It would be nice to add a progress bar or some other kind of estimation of how long it would take for the task to be done, with a maximum duraton set after which the task should be considered to have failed. The client would poll until the task status is marked "done," after which they would be redirected back to the application flow. Each polling request sent by the client could likely be processed very quickly, so this this strategy effectively breaks the task into many small requests ("It's better to have many short requests than one long one").

Prudence's tasks (page 49) implementation is a good start for creating such a mechanism: however, it would be up to you to create a "please wait" mechanism, as well as a way to track the tasks' progress and deal with failure.

Implementing such a handling service is not trivial. It adds a new component to your architecture, one that also has to be made to scale. One can also argue that it adversely affects user experience by adding overhead, delaying the time it takes for the task to complete. The bottom line, though, is you're vastly increasing concurrency and your ability to scale. And, you're improving the user experience in one respect: they would get a feedback on what's going on rather than having their browsers spin, waiting for their requests to complete.

File Uploads

These are potentially very long requests that you cannot break into smaller tasks, because they depend entirely on the client. As such, they present a unique challenge to scalability.

Fortunately, Prudence handles client requests via non-blocking I/O, meaning that large file uploads will not hold on to a single thread for the duration of the upload. See conversation.form (page 75).

Unfortunately, many concurrent uploads will still saturate your threads. If your application relies on frequent file uploads, you are advised to handle such requests on separate Prudence instances, so that uploads won't stop your application from handling other web requests. You may also consider using a third-party service specializing in file storage and web uploads.

Asynchronous Request Processing

Having the client poll until a task is completed lets you break up a task into multiple requests and increase concurrency. Another strategy is to break an *individual request* into pieces. While you're processing the request and preparing the response, you can free the web thread to handle other requests. When you're ready to deliver content, you raise a signal, and the next available web thread takes care of sending your response to the client. You can continue doing this indefinitely until the response is complete. From the client's perspective it's a single request: a web browser, for example, would spin until the request was completed.

You might be adding some extra time overhead for the thread-switching on your end, but the benefits for scalability are obvious: you are increasing concurrency by shortening the time you are holding on to web threads.

For web services that deliver heavy content, such as images, video, audio, it's absolutely necessary. Without it, a single user could tie up a thread for minutes, if not hours. You would still get degraded performance if you have more concurrent users than you have threads, but at least degradation will be shared among users. Without asynchronous processing, each user would tie up one thread, and when that finite resource is used up, more users won't be able to access your service.

Even for lightweight content such as HTML web pages, asynchronous processing can be a good tactic for increasing concurrency. For example, if you need to fetch data from a backend with non-deterministic response time, it's best to free the web thread until you actually have content available for the response.

It's not a good idea to do this for every page. While it's better to have many short requests instead of one long one, it's obviously better to have one short request rather than many short ones. Which web requests are good candidates for asynchronous processing?

1. Requests for which processing is made of independent operations. (They'll likely be required to work in sequence, but if they can be processed in parallel, even better!)

2. Requests that must access backend services with non-deterministic response times.

And, even for #2, if the service can take a *very* long time to respond, consider that it might be better to queue the task on a task handler and give proper feedback to the user.

And so, after this lengthy discussion, it turns out that there aren't that many places where asynchronous processing can help you scale. Caching is far more useful.

As of version 1.1, Prudence has limited support for asynchronous processing, via conversation.defer (page 80). Better support is planned for a future version.

Backend Partitioning

You can keep adding more nodes behind a load balancer insofar as each request does not have to access shared state. Useful web applications, however, are likely data-driven, requiring considerable state.

If the challenge in handling web requests is cutting down the length of request, then that of backends is the struggle against degraded performance as you add new nodes to your database cluster. These nodes have to synchronize their state with each other, and that synchronization overhead increases exponentially. There's a definite point of diminishing returns.

The backend is one place where high-performance hardware can help. Ten expensive, powerful machines might be equal in total power to forty cheap machines, but they require a quarter of the synchronization overhead, giving you more elbow room to scale up. Fewer nodes is better.

But CPUs can only take you so far.

Partitioning is as useful to backend scaling as caching is to web request scaling. Rather than having one big cluster of identical nodes, you would have several smaller, independent clusters. This lets you add nodes to each cluster without spreading synchronization overhead everywhere. The more partitions you can create, the better you'll be able to scale.

Partitioning can happen in various components of your application, such as application servers, the caching system, task queues, etc. However, it is most effective, and most complicated to implement, for databases. Our discussion will thus focus on relational (SQL) databases. Other systems would likely require simpler subsets of these strategies.

Reads vs. Writes

This simple partitioning scheme greatly reduces synchronization overhead. Read-only servers will never send data to the writable servers. Also, knowing that they don't have to handle writes means you can optimize their configurations for aggressive caching.

(In fact, some database synchronization systems will only let you create this kind of cluster, providing you with one "master" writable node and several read-only "slaves." They force you to partition!)

Another nice thing about read/write partitioning is that you can easily add it to all the other strategies. Any cluster can thus be divided into two.

Of course, for web services that are heavily balanced towards writes, this is not an effective strategy. For example, if you are implementing an auditing service that is constantly being bombarded by incoming data, but is only queried once in a while, then an extra read-only node won't help you scale.

Note that one feature you lose is the ability to have a transaction in which a write *might* happen, because a transaction cannot contain both a read-only node and a write-only node. If you must have atomicity, you will have to do your transaction on the writable cluster, or have two transactions: one to lookup and see if you need to change the data, and the second to perform the change—while first checking again that data didn't change since the previous transaction. Too much of this obviously lessens the effectiveness of read/write partitioning.

By Feature

The most obvious and effective partitioning scheme is by feature. Your site might offer different kinds of services that are functionally independent of each other, even though they are displayed to users as united. Behind the scenes, each feature uses a different set of tables. The rule of thumb is trivial: if you can put the tables in separate databases, then you can put these databases in separate clusters.

One concern in feature-based partitioning is that there are a few tables that still need to be shared. For example, even though the features are separate, they all depend on user settings that are stored in one table.

The good news is that it can be cheap to synchronize just this one table between all clusters. Especially if this table doesn't change often—how often do you get new users signing up for your service?—then synchronization overhead will be minimal.

If your database system doesn't let you synchronize individual tables, then you can do it in your code by writing to all clusters at the same time.

Partitioning by feature is terrific in that it lets you partition other parts of the stack, too. For example, you can also use a different set of web servers for each feature.

Also consider that some features might be candidates for using a "NoSQL" database (page 103). Choose the best backend per feature.

By Section

Another kind of partitioning is sometimes called "sharding." It involves splitting up tables into sections that can be placed in different databases. Some databases support sharding as part of their synchronization strategy, but you can also implement it in your code. The great thing about sharding is that it lets you create as many shards (and clusters) as you want. It's the key to the truly large scale.

Unfortunately, like partitioning by feature, sharding is not always possible. You need to also shard all related tables, so that queries can be self-contained within each shard. It's thus most appropriate for one-to-many data hierarchies. For example, if your application is a blog that supports comments, then you put some blogs and their comments on one shard, and others in another shard. However, if, say, you have a feature where blog posts can refer to other arbitrary blog posts, then querying for those would have to cross shard boundaries.

The best way to see where sharding is possible is to draw a diagram of your table relationships. Places in the diagram which look like individual trees—trunks spreading out into branches and twigs—are good candidates for sharding.

How to decide which data goes in which shard?

Sometimes the best strategy is arbitrary. For example, put all the even-numbered IDs in one shard, and the odd-numbered ones in another. This allows for straightforward growth because you can just switch it to division by three if you want three shards.

Another strategy might seem obvious: If you're running a site which shows different sets of data to different users, then why not implement it as essentially separate sites? For example, a social networking site strictly organized around individual cities could have separate database clusters per city.

A "region" can be geographical, but also topical. For example, a site hosting dance-related discussion forums might have one cluster for ballet and one for tango. A "region" can also refer to user types. For example, your social networking site could be partitioned according to age groups.

The only limitation is queries. You can still let users access profiles in other regions, but cross-regional relational queries won't be possible. Depending on what your application does, this could be a reasonable solution.

A great side-benefit to geographical partitioning is that you can host your servers at data centers within the geographical location, leading to better user experiences. Regional partitioning is useful even for "NoSQL" databases.

Coding Tips for Partitioning

If you organize your code well, it would be very easy to implement partitioning. You simply assign different database operations to use different connection pools. If it's by feature, then you can hard code it for those features. If it's sharding, then you add a switch before each operation telling it which connection pool to use.

For example:

```
def get_blogger_profile(user_id):
        connection = blogger_pool.get_connection()
        ...
        connection.close()

def get_blog_post_and_comments(blog_post_id):
        shard_id = object.id % 3
        connection = blog_pools[shard_id].get_connection()
```

```
...
connection.close()
```

Unfortunately, some programming practices make such an effective, clean organization difficult.

Some developers prefer to use ORMs (object-relational mappers) rather than access the database directly. Many ORMs do not easily allow for partitioning, either because they support only a single database connection pool, or because they don't allow your objects to be easily shared between connections.

For example, your logic might require you to retrieve an "object" from the database, and only then decide if you need to alter it or not. If you're doing read/write partitioning, then you obviously want to read from the read partition. Some ORMs, though, have the object tied so strongly to an internal connection object that you can't trivially read it from one connection and save it into another. You'd either have to read the object initially from the write partition, minimizing the usefulness of read/write partitioning, or re-read it from the write partition when you realize you need to alter it, causing unnecessary overhead. (Note that you'll need to do this anyway if you need the write to happen in a transaction.)

Object oriented design is also problematic in a more general sense. The first principle of object orientation is "encapsulation," putting your code and data structure in one place: the class. This might make sense for business logic, but, for the purposes of re-factoring your data backend for partitioning or other strategies, you really don't want the data access code to be spread out among dozens of classes in your application. You want it all in one place, preferably even one source code file. It would let you plug in a whole new data backend strategy by replacing this source code file. For data-driven web development, you are better off not being too object oriented.

Even more generally speaking, organizing code together by mechanism or technology, rather than by "object" encapsulation, will let you apply all kinds of re-factorizations more easily, especially if you manage to decouple your application's data structures from any library-specific data structures.

Data Backends

Relational (SQL) databases such as MySQL were, for decades, the backbone of the web. They were originally developed as minimal alternatives to enterprise database servers such as Oracle Database and IBM's DB2. Their modest feature set allowed for better performance, smaller footprints, and low investment costs—perfect for web applications. The free software LAMP stack (Linux, Apache, MySQL and PHP) *was* the web.

Relational databases require a lot of synchronization overhead for clusters, limiting their scalability. Though partitioning can take you far, using a "NoSQL" database could take you even further.

Graph Databases

If your relational data structure contains arbitrary-depth relationships or many "generic" relationships forced into a relational model, then consider using a graph database instead. Not only will traversing your data be faster, but also the database structure will allow for more efficient performance. The implications for scalability can be dramatic.

Social networking applications are often used as examples of graph structures, but there are many others: forums with threaded and cross-referenced discussions, semantic knowledge bases, warehouse and parts management, music "genomes," user-tagged media sharing sites, and many science and engineering applications.

Though fast, querying a complex graph can be difficult to prototype. Fortunately, the Gremlin and SPARQL languages do for graphs what SQL does for relational databases. Your query becomes coherent and portable.

A popular graph database is Neo4j, and it's especially easy to use with Prudence. Because it's JVM-based, you can access it internally from Prudence. It also has embedded bindings for many of Prudence's supported languages, and supports a network REST interface which you can easily access via Prudence's document.external.

Document Databases

If your data contains mostly "documents"—self-contained records with few relationships to other documents—then consider a document database.

Document databases allow for straightforward distribution and very fine-grained replication, requiring considerably less overhead than relational and graph databases. Document databases are as scalable as data storage gets: variants are used by all the super-massive Internet services.

The cost of this scalability is the loss of your ability to do relational queries of your data. Instead, you'll be using distributed map/reduce, or rely on an external indexing service. These are powerful tools, but they do not match relational queries in sheer speed of complex queries. Implementing something as simple as a many-to-many connection, the bread-and-butter of relational databases, requires some specialization. Document databases shine at listing, sorting and searching through extremely large catalogs of documents.

Candidate applications include online retail, blogs, wikis, archives, newspapers, contact lists, calendars, photo galleries, dating profiles... This is a long list, but by no means exhaustive of all that is possible in web applications. Many useful applications cannot be reduced to sets of lightly interconnected "documents" without giving up a lot of useful functionality. For example, merely adding social networking capabilities to a dating site would require complex relations that might be better handled with a graph database.

A popular document database is MongoDB. Though document-based, it has a few basic relational features that might be just good enough for your needs. Another is CouchDB, which is a truly distributed database. With CouchDB it's trivial to replicate and synchronize data with clients' desktops or mobile devices, and to distribute it to partners. It also supports a REST interface which you can easily access via Prudence's document.external. And, both MongoDB and CouchDB use JavaScript extensively, making it natural to use with Prudence's JavaScript flavor.

> We've started a project to better integrate Prudence JavaScript with MongoDB. It is included in the "Savory JavaScript" edition.

Column Databases

These can be considered as subsets of document databases. The "document," in this case, is required to have an especially simple, one-dimensional structure.

This requirement allows optimization for a truly massive scale.

Column databases occupy the "cloud" market niche: they allow massive companies like Google and Amazon to cheaply offer database storage and services for third parties. See Google's Datastore (based on Bigtable) and Amazon's SimpleDB (based on Dynamo; actually, Dynamo is a "key/value" database, which is even more opaque than a column database).

Though you can run your own column database via open source projects like Cassandra (originally developed by/for Facebook) and HBase, the document databases mentioned above offer richer document structures and more features. Consider column databases only if you need truly massive scale, or if you want to make use of the cheap storage offered by "cloud" vendors.

Best of All Worlds

Of course, consider that it's very possible to use both SQL and "NoSQL" (graph, document, column) databases together for different parts of your application. See backend partitioning (page 101).

Under the Hood

Prudence brings together many open source libraries, some of which were designed specifically for Prudence.

Consider this as Prudence's "acknowledgments" page. Hundreds of people have worked on these libraries, and we're grateful to all of them for sharing their hard work, for embracing open source licensing, and for adhering to design principles that allow reuse of their work in other projects, such as Prudence.

The JVM

How wonderful that the best-performing, most robust, secure and widely ported virtual machine is now open source? How wonderful that you can use it with JavaScript, Python, Ruby, Clojure, PHP and others languages?

We strongly recommend the JVM for enterprise and scalable Internet applications, even if you're not particularly fond of Java-the-programming-language. Treat Java, if you will, as the low machine-level language:

Java is to the JVM as C is to Unix. You "drop down" to Java only if you have to do some machine-level work. Otherwise, use the higher-level languages.

Scripturian

Scripturian is Prudence's "special sauce": a small, magical library that makes sure that your code runs well on the JVM and can handle the concurrency introduced by HTTP requests and Restlet. It is developed in tandem with Prudence.

Our premise was this: to make Prudence applications easy to deploy, they could not be standard Java applications. The cycle of compilation and packaging required by Java is unnecessarily cumbersome. Although we could have implemented something like the on-the-fly Java compilation done in JSP, we felt that, if that's the route to go, many exciting choices open up besides Java, and that these languages are more relevant to Prudence's goals.

Unfortunately, we found that integrating JVM languages into Prudence was anything but trivial. Each implementation had its own idea of what integration could mean. We tried to standardize on JSR-223 (the Java scripting standard), but found that adherence to the specification was inconsistent, and that the specification itself is vague, especially when it comes to threading. We hacked and hacked and hacked. We even submitted patches to fix broken implementations of various languages. All in all, we probably spent more time on this than on any other aspect of Prudence.

The result is an API more abstract than JSR-223, but with a clear threading model. Under the hood, Scripturian contains many tricks and mechanisms to make each language work correctly and well, but you don't have to worry about any of it: Scripturian just works.

Jython, JRuby, Clojure, Rhino, Quercus, Groovy

Prudence would hardly be as exciting if you had to use Java.

These open source language engines have allowed us to extend the power of Prudence, REST and the JVM to languages outside of Java. Some of these engines are large, complex projects, and are in fact the biggest libraries included in Prudence. We strongly recommend you join in the communities surrounding the language engines corresponding to your favorite flavor of Prudence.

Restlet

Prudence went through many in-house transformations before aligning itself strongly with Restlet. First, we experimented with Facelets, but ended up giving up on JSF, its complex lifecycle, and on the promise of component-based web development in general. Then, we designed REST architectures using servlets and Succinct templates, but found it awkward to force servlets into a REST architecture. Discovering Restlet was a breath of fresh air.

Restlet's super-powers are three:

1. It's a clean abstraction of HTTP requests, responses and headers over best-of-breed HTTP engines, such as Jetty, Grizzly and Netty. Automatically gain the scalable advantages of non-blocking I/O and even asynchronous request handling (which will be even better supported in Restlet 2.1). Restlet transparently handles conditional requests, content negotiation, and other complicated HTTP mechanisms.

2. Powerful URI routing and manipulation. Expose your service to users and APIs with elegance and coherence. Make sure the URI reaches its destination, with support for virtual hosting, rewriting, templating, and other useful real-world features. Restlet is truly the Swiss army knife of URIs.

3. Straightforward data representation and consumption through a diverse set of extensions. Expose your data using any standard format, and even convert it on the fly. Easily parse data received from clients.

Restlet is a great library, with a great ecosystem of extensions. In embracing it, though, we missed some of the advantages of having a servlet container: easy deployment and configuration, centralized logging, etc. We also missed having JSP at our fingertips to quickly push out dynamic HTML.

Prudence is meant to fill in these gaps.

Special thanks go to Noelios, and especially Jérôme Louvel, for cultivating such a vibrant community around Restlet. It's a solid foundation.

Succinct

Succinct, like Scripturian, started as a part of Prudence, and was in fact one of its earliest components. It has since branched out into an independent library. We created it because we wanted straightforward, scalable templating built in to Prudence, and were unsatisfied by other open source offerings. We think you might like it. (If not, Prudence fully supports Velocity.)

Jygments

Yet another Prudence side-project...

For Prudence's debug mode, we wanted good syntax highlighting for viewing source code. We found nothing adequate enough in the Java world, though we fell in love with Pygments. For a while, we ran Pygments in Prudence via Jython, but found it too heavy for this particular use case. Thus, Jygments was born as a port of Pygments to Java.

H2

We're great fans of this lean and mean database engine! It has allowed us to distribute Prudence with a fully-functioning data-drive demo application, without any external dependencies. We're proud to introduce H2, through Prudence, to more people, and we believe you'll find it fast enough, reliable enough, and flexible enough for many production environments.

Hazelcast

This library is a dream come true: distributed, fault-tolerant implementations of the JVM's standard collection interfaces, with distributed task queues thrown into the box. We are confident that Hazelcast will help many Prudence users scale their applications easily and elegantly.

FAQ

REST

Why are plural URL forms for aggregate resources (/animal/cats/) preferred over singular forms (/animal/cat/)?

You'll see RESTful implementations that use either convention. The advantage of using the singular form is that you have less addresses, and what some people would call a more elegant scheme:

```
/animal/cat/12 -> Just one cat
/animal/cat/ -> All cats
```

Why add another URL when a single one is enough to do the work? One reason is that you can help the client avoid potential errors. For example, the client probably uses a variable to hold the ID of the cat and then constructs the URL dynamically. But, what if the client forgets to check for null IDs? It might then construct a URL in the form "/animal/cat//" which would then successfully access *all* cats. This can cause unintended consequences and be difficult to debug. If, however, we used this scheme:

```
/animal/cat/12 -> Just one cat
/animal/cats/ -> All cats
```

...then the form "/animal/cat//" would route to our singular cat resource, which would indeed not find the cat and return the expected, debuggable 404 error. From this example, we can extract a good rule of thumb: clearly separate URLs at their base by usage, so that mistakes cannot happen. More addresses means more debuggability.

Languages

How to avoid the "Adapter not available for language: xml" parsing exception for XML files?

The problem is that the XML header confuses Scripturian, Prudence's langauge parser, which considers the "<?" a possible scriptlet delimeter:

```
<?xml version='1.0' encoding='UTF-8'?>
```

The simple solution is to force Scripturian to use the "<%" for the page via an empty scriptlet, ignoring all "<?":

```
<% %><?xml version='1.0' encoding='UTF-8'?>
```

Scalability

I heard REST is very scalable. Is this true? Does this mean Prudence can support many millions of users?

Yes, if you know what you're doing. See "The Case for REST" (page 5) and "Scaling Tips" (page 93) for in-depth discussions.

The bottom line is that it's very easy to make your application scale poorly, whatever technology or architecture you use, and that Prudence, in embracing REST and the JVM, can more easily allow for best-practice scalable architectures than most other web platforms.

That's not very reassuring, but it's a fact of software and hardware architecture right now. Achieving massive scale is challenging.

Performance

How well does Prudence perform? How well does it scale?

First, recognize that there are two common uses for the term "scale." REST is often referred to as an inherently scalable architecture, but that has more to do with project management than technical performance. This difference is addressed in the "The Case for REST" (page 5).

From the perspective of the ability to respond to user requests, there are three aspects to consider:

1. Serving HTTP Prudence comes with Jetty, an HTTP server based on the JVM's non-blocking I/O API. Jetty handles concurrent HTTP requests very well, and serves static files at scales comparable to popular HTTP servers.

2. Generating HTML Prudence implements what might be the most sophisticated caching system of any web development framework. Caching is truly the key to scalable software. See "Scaling Tips" (page 95) for a comprehensive discussion.

3. Running code There may be a delay when starting up a specific language engine in Prudence for the first time in an application, as it loads and initializes itself. Then, there may be a delay when accessing a dynamic web page or resource for the first time, or after it has been changed, as it might require compilation. Once it's up and running, though, your code performs and scale very well—as well as you've written it. You need to understand concurrency and make sure you make good choices to handle coordination between threads accessing the same data. If all is good, your code will actually perform better throughout the life of the application. The JVM learns and adapts as it runs, and performance can improve the more the application is used.

All flavors of Prudence are generally very fast. In some cases, the JVM language implementations are faster than their "native" equivalents. This is demonstrable for Python, Ruby and PHP. The reason is that the JVM, soon reaching version 7, is a very mature virtual machine, and incorporates decades-worth of optimiziations for live production environments.

If you are performing CPU-intensive or time-sensitive tasks, then it's best to profile these code segments precisely. Exact performance characteristics depend on the language and engine used. The Bechmarks Game can give you some comparisons of different language engines running high-computation programs. In any

case, if you have a piece of intensive code that really needs to perform well, it's probably best to write it in Java and access it from the your language. You can even write it in C or assembly, and have it linked to Java via JNI.

If you're not doing intensive computation, then don't worry too much about your language being "slow." It's been shown that for the vast majority of web applications, the performance of the web programming language is rarely the bottleneck. The deciding factors are the usually performance of the backend data-driving technologies and architectures.

Licensing

> The author is not a lawyer. This is not legal advice, but a personal, and possibly wrong interpretation. The wording of the license itself supersedes anything written here.

Does the LGPL mean I can't use Prudence unless my product is open sourced?

The GPL family of licenses restrict your ability to *redistribute* software, not to use it. You are free to use Prudence as you please within your organization, even if you're using it to serve public web sites—though with no warranty nor an implicit guarantee of support from the copyright holder, Three Crickets LLC.

The GPL would thus only be an issue if you're selling, or even giving away, a product that *would include* Prudence.

In fact, Prudence uses the Lesser GPL, which has even less restrictions on redistribution than the regular GPL. Essentially, as long as you do not alter Prudence in any way, you can include Prudence in any product, even if it is not free. (With one exception: Prudence uses version 3 of the Lesser GPL, which requires your product to not restrict users' ownership of data via schemes such as DRM if Prudence is to be included in its distribution.)

Even if your product does not qualify for including Prudence in it, you always have the option of distributing your product without Prudence, and instructing your customers to download and install Prudence on their own.

We understand that in some cases open sourcing your product is impossible, and passing the burden to the users is cumbersome. As a last resort, we offer you a commercial license as an alternative to the GPL. Please contact Three Crickets for details.

Three Crickets, the original developers of Prudence, are not trying to force you to purchase it. Instead, they hope to encourage you to 1) pay Three Crickets for consultation, support and development services for Prudence, which is our business model for Prudence, and to 2) consider releasing your own product as free software, thereby truly sharing your innovation with all of society.

Why the LGPL and not the GPL?

The Lesser GPL used to be called the "Library GPL," and was originally drafted for glibc. It is meant for special cases in which the full GPL could severely limit the adoption of a product, which would be self-defeating. The assumption is that there are many alternatives with less restrictions on distribution.

In the case of Linux, the full GPL has done a wonderful job at convincing vendors to open source their code in order to ship their products with Linux inside. However, it doesn't seem likely that they would do the same for Prudence.

Note that the LGPL version 3 has a clause allowing you to "upgrade" Prudence to the full GPL for inclusion in your GPL-ed product. This is a terrific feature, and another reason to love this excellent license.