The Prudence Manual

Copyright 2009-2010 by Three Crickets LLC
Main text written by Tal Liron

May 25, 2010

Contents

3

Tutorial

Run the Prudence Instance

Prudence comes ready to rumble!

All you need is a Java Virtual Machine (JVM), minimally version 5. Your operating system may already have one. Typing "java" from the command line will usually tell you so. Otherwise, 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.

Try Prudence! Run /bin/run.sh for Unix-like systems (Linux, *BSD and OS X), or /bin/run.bat for Windows. 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. Open your web browser to http://localhost:8080/. You should see the Prudence Administration application, where you can access the demos.

> The /bin/run scripts are there for getting you quickly up and running, however it is strongly recommended that you run Prudence as a daemon in production environments.

Your First Application

Your Prudence instance hosts all applications under the /applications/ directory. To install a new application, simply create a new subdirectory there. The subdirectory name will be used as a default for various things: the base URL, logging, etc. We can change those later.

So, let's create /applications/wackywiki/

If you restart Prudence, you'll see wackywiki listed in the admin application. There's nothing to see there quite yet, though.

> *By the way:* It's perfectly fine to use symbolic links or mounts to put your application subdirectories elsewhere.

A Static Web Page

Create /applications/wackywiki/web/static/. Files you put there will served just like from any static web server. You can put images, HTML files, CSS or anything else. Let's start with a default web page.

/web/static/index.html:

```
<html>
<head>
```

```
         <title>Wacky Wiki</title>
</head>
<body>
         <div>
                   Nothing to see here, for now. Carry on.
         </div>
</body>
</html>
```

Restart Prudence, and browse to http://localhost:8080/wackiwiki/ to see the page.

A Dynamic Web Page

> Important! While Prudence does dynamic web pages well, it really stands
> out from other web frameworks in its support for REST resources, which
> we'll see in the next section. We decided to start this tutorial with web
> pages, because the topic would likely be more familiar to most newcomers
> to Prudence.

Create /applications/wackywiki/web/dynamic/. Unlike /web/static/, files in this direc-
tory must be text files (HTML, XML, plain text, etc.) They are specially processed so
that they can include "scriptlets" of programming code.

Let's move our index.html from /web/static/ to /web/dynamic/ and edit it to add
some Python scriptlets.

/web/dynamic/index.html:

```
<html>
<head>
         <title>Wacky Wiki</title>
</head>
<body>
         <div>
                   It is currently
<%python
from datetime import datetime
now = datetime.now()
print now.strftime('%H:%M')
%>
                   o' clock, and
<%
print now.strftime('%S')
%>
                   seconds
         </div>
</body>
</html>
```

Restart Prudence, and browse to http://localhost:8080/wackiwiki/ to see the page.

You'll notice the <% and %> are used to delimit scriptlets, and that we've added "python" to the opening delimiter of the first scriptlet. Subsequent scriptlets on the page will automatically use the language of the previous scriptlets, so we don't have to repeat "python".

Note, too, that Python's "print" works by adding output to the stream of text where the scriptlet is located.

Final note: Though they are conventionally called "scriptlets," they are not "scripts" in the common meaning of interpreted code. All scriptlets are, in fact, compiled and run as JVM bytecode. That's fast.

Expressions

Let's make our dynamic page more readable by using special "expression scriptlets." They're really just a shorthand for "print."

/web/dynamic/index.html:

```
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
        <div>
<%python
from datetime import datetime
now = datetime.now()
%>
                    It is currently <%= now.strftime('%H:%M') %> o' clock, and <%= n
        </div>
</body>
</html>
```

What happens if you have a coding error in your scriptlets? Let's try and introduce an error on purpose.

/web/dynamic/index.html:

```
... from dratetime import datetime ...
```

If you try to browse to the page, Prudence will show you a detailed debug page, from where you can also access the source code with the troublesome code line highlighted. Once you deploy your application, you can cancel this debug feature. Also, you can show your own, custom, user-friendly error pages. We'll get to that below.

Fragments

Prudence lets you include others documents in-place using a special scriptlet. This makes it very easy to reuse fragments of documents (which can include scriptlets)

throughout your dynamic pages. Common reusbale fragments are page headers, footers, and navigation menus.

By convention in Prudence, fragments are put in /web/fragments/, though you can include them from anywhere, even from your regular /web/dynamic/ or /web/static/ subdirectories. The advantage of keeping them out of those subdiretories is that users won't be able to access the individual fragments directly. Instead, we must explicitly include them in our /web/dynamic/ files.

Let's create a fragment that displays the current time.

/web/fragments/time.html:

```
<%python
from datetime import datetime
now = datetime.now()
%>
It is currently <%= now.strftime('%H:%M') %> o' clock, and <%= now.strftime('%S'
```

/web/dynamic/index.html:

```
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
        <div>
                <%& 'time/' %>
        </div>
</body>
</html>
```

Note that the included scriptlet, which begins with the <%& delimiter, accepts a real Python expression. For example, you can do Pythony things like string interpolation: <%& '%s-%s' % (language, encoding) %>, allowing you to include different fragments according to changing circumstances.

Caching

Dynamically generating HTML and other text is very powerful. You may be wondering, however, how well this scales. Even if the Python code is compiled, it would still need to be run for every user request, right? Prudence supports straightforward per-page caching to greatly help you scale. By changing document.cacheDuration within a scriptlet, you can dynamically change how long Prudence will look for updates in the document. Note that even very short caching times can be a great boost to scalability. For example, if your cache duration is 1 second, it means that your scriptlets would only be run once every second. In computing terms, that's a very long time! Imagine that your popular web site is getting 1000 hits per second...

Finally, you'll be happy to know that by default Prudence also enables client-side caching. It does this by setting a standard HTTP header that tells the client how long the content should be cached. All popular desktop and mobile web browsers recognize

it. This means bandwidth savings for you, because the client will not download content it already has in its cache.

Let's enable a simple 60 second cache.

/web/dynamic/index.html:

```
<% document.cacheDuration = 60000 %>
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
        <div>
                <%& 'time/' %>
        </div>
</body>
</html>
```

To test the above, go to our page and keep refreshing it in your web browser. You will see that the printed time will only be updated once every 10 seconds.

To verify that client-side caching works correctly, we recommend using the free Firefox browser with the Firebug add-on. In the Firebug network panel, you can see which requests are sent to the server. Within those 60 seconds between updates, you will see the that the request for the page returns a 304 status code: "the document has not been modified."

For an even more elaborate experiment, try running two browsers at the same time on different machines, and keep refreshing them. You'll see each client caching individually, and yet the server still returning a different page only after the 60 second cache duration has passed.

Caching is very powerful, but obviously not useful for all pages. For example, if it's important for a page to always show the up-to-date to the moment information, you might not be able to afford even a one second cache. It's useful, then, to know that Prudence's document.cacheDuration works at the level of the actual document, so that each included fragment can set its own cache duration. Just remember that the top document or fragment requests will determine the caching for included fragments, too. If your index.html has a cache duration of 60 seconds, then even if an included fragment has a cache duration of 1 second, it would only appear to update every 60 seconds.

Scriptlets can also dynamically change the cache duration according to changing circumstances. For example, under heavy load, you might want your application to cache for longer periods of time.

HTML Forms

TODO

Let's allow users to edit our wacky wicki.

```
form = prudence.resource.request.resourceRef.queryAsForm
fresh = form.getFirstValue('fresh') == 'true'
```

/web/fragments/wiki.html:

```
<%python
wiki = 'Nothing in the wiki yet.'
%>
Wiki content is: <%= wiki %>
```

/web/dynamic/index.html:

```
<html>
<head>
        <title>Wacky Wiki</title>
</head>
<body>
        <div>
                <%& 'time/' %>
        </div>
        <div>
                <%& 'wiki/' %>
        </div>
</body>
</html>
```

More HTTP

Prudence does a lot of the HTTP work automatically, but you can manipulate the response yourself. For example, to redirect the client:

```
prudence.resource.response.locationRef = prudence.resource.request.resourceRef.b
prudence.statusCode = 303
raise
```

> Note the use of "raise": it's a handy trick to end processing of our page.
> Prudence recognizes that we explicitly changed the status code, so it doesn't
> consider this to be a real error.

You can use also prudence.statusCode to explicitly set an error, for example to send 403 ("Forbidden") if the user is not logged in.

Need more HTTP? Prudence supports HTTP cookies and HTTP authentication challenges. We won't get into that in this tutorial, though.

Templating

You might wonder what other options are available to you other than "python" for scriptlets. You can choose other languages—Ruby, JavaScript, Clojure, PHP, etc.— if you have them installed. But, more importantly, you can choose to use one of the pre-included templating languages, Succint and Velocity.

Templating languages have far less features than any of the above programming languages, but this can be an advantage if you all you need is simple templating. They tend to be more readable for templates. Best of all, you can mix scriptlets from various languages in one page. Here's an example of using Velocity and Python:

/web/fragments/time.html:

```
<%python
from datetime import datetime
now = datetime.now()
clock = now.strftime('%H:%M')
seconds = now.strftime('%S')
%>
<%velocity It is currently $clock o' clock, and $seconds seconds %>
```

A Resource

As stated earlier, where Prudence truly shines is in its support for REST resources. A resource is a piece of data associated with a URL, which clients can optionally create, read, update or delete. An HTML web page is an example of a resource which is usually read, and sometimes supports writing (for handling HTML forms). Prudence lets you create resources that support one or more formats, all create/read/update/delete (CRUD) operations, and advanced caching.

A resource-oriented architecture is perfect for "AJAX" and other rich client platforms (Flash, Silverlight, etc.). Another use is for a scalable backend conduit between servers. Depending on how you look at it, resources are either an online database, an API for your service, or both.

> You might want to take a look at Making the Case for REST for a better understanding of REST and what it can do for you.

Let's start with a resource that can be read for some plain text. Resources are put in the /resources/ subdirectory, and, like /web/dynamic/ and /web/static/, the filename and directory tree define the URL:

/resources/wiki.py:

```
def handle_init(conversation):
        conversation.addMediaTypeByName('text/plain')
def handle_get(conversation):
        return 'Nothing in the wiki yet.'
```

handleInit() always gets called first, before any of the other handler functions.

Try browsing to http://localhost:8080/wackiwiki/wiki/ to see your resource.

Plain text is good for us humans to read. But, we might want to support formats that are more useful for client applications, such as JSON for browser-based JavaScript or Flash's ActiveScript.

Let's add support for JSON to our resource:

/resources/wiki.py:

```
wiki = 'Nothing in the wiki yet.'

def handle_init(conversation):
        conversation.addMediaTypeByName('text/plain')
        conversation.addMediaTypeByName('application/json')

def handle_get(conversation):
        if conversation.mediaTypeByName == 'text/plain':
                return wiki
        else:
                return '{"content": "%s"}' % wiki
```

[Add jQuery to dynamic page to test in browser.]

Just as /web/fragments/ was a useful place for us to put reusable web fragments, we can use /libraries/ for reusable code for resources, as well as dynamic web scriptlets. Let's create a library to handle our wiki:

/libraries/wiki.py:

```
wiki = 'Nothing in the wiki yet.'

def get_wiki():
        return wikiwiki = 'Nothing in the wiki yet.'

def handle_init(conversation):
        conversation.addMediaTypeByName('text/plain')
        conversation.addMediaTypeByName('application/json')

def handle_get(conversation):
        if conversation.mediaTypeByName == 'text/plain':
                return wiki
        else:
                return '{"content": "%s"}' % wiki
```

Let's make use of our library in our resource:

/resources/wiki.py:

```
from wiki import get_wiki

def handle_init(conversation):
        conversation.addMediaTypeByName('text/plain')
        conversation.addMediaTypeByName('application/json')

def handle_get(conversation):
        if conversation.mediaTypeByName == 'text/plain':
                return get_wiki()
        else:
                return '{"content": "%s"}' % get_wiki()
```

[Put, Post, Delete.]

/libraries/wiki.py:

```python
from threading import RLock

wiki = 'Nothing in the wiki yet.'
wiki_lock = RLock()

def get_wiki():
        wiki_lock.acquire()
        try:
                return wiki
        finally:
                wiki_lock.release()

def set_wiki(content):
        wiki_lock.acquire()
        try:
                wiki = content
        finally:
                wiki_lock.release()
```

[Fixing our form handling from earlier using doPost.]

Remember how we supported client-side caching for our dynamic web pages? We have much more control over this process in resources. We can set explicit modification dates and ETags.

Let's give our wiki resources a modification date, so that clients can properly cache it:

/libraries/wiki.py:

```python
from threading import RLock
from datetime import datetime

def to_milliseconds(dt):
        return long(mktime(dt.timetuple()) * 1000)
        wiki = {content: 'Nothing in the wiki yet.', timestamp: datetime.now()}
        wiki_lock = RLock()

def get_wiki():
        wiki_lock.acquire()
        try:
                # Note that we are returning a copy of our wiki dictionary!
                return {content: wiki['content'], timestamp: wiki['timestamp']}
        finally:
                wiki_lock.release()

def set_wiki(content):
        wiki_lock.acquire()
```

```
        try :
                wiki ['content'] = content
                wiki ['timestamp'] = datetime.now()
        finally :
                wiki_lock.release()
```

/resources/wiki.py:

```python
from wiki import get_wiki , set_wiki , to_milliseconds

def handle_init ( conversation ):
        conversation.addMediaTypeByName('text/plain')
        conversation.addMediaTypeByName('application/json')

def handle_get ( conversation ):
        wiki = get_wiki()
        conversation.modificationTimestamp = to_milliseconds(wiki['timestamp'])
        if conversation.mediaTypeByName == 'text/plain':
                return wiki['content']
        else :
                return '{"content": "%s"}' % wiki['content']
```

Support for modification date can go a long way towards saving us bandwidth. However, note that we still have to fetch the entire wiki in order to get its modification date. This could potentially be a costly operation, say, if the wiki is stored in a database.

Prudence lets you optimize this by letting you check only the modification date (or ETag) before actually fetching the resource.

Let's add handleGetInfo():

/resources/wiki.py:

```python
from wiki import get_wiki , set_wiki , to_milliseconds

def handle_init ( conversation ):
        conversation.addMediaTypeByName('text/plain')
        conversation.addMediaTypeByName('application/json')

def handle_get ( conversation ):
        wiki = get_wiki()
        conversation.modificationTimestamp = to_milliseconds(wiki['timestamp'])
        if conversation.mediaTypeByName == 'text/plain':
                return wiki['content']
        else :
                return '{"content": "%s"}' % wiki['content']

def handle_get_info ( conversation ):
        wiki = get_wiki()
        return to_milliseconds(wiki['timestamp'])
```

15

Our trivial example didn't offer any significant savings. But, suppose our resource involved fetching lots of wiki pages at once? We could save computing power if you we only had to fetch a modification date.

As we said earlier, our resources can be seen as an API. Indeed, there's no reason to maintain two sets of APIs, one for "the world" to access via your resources, and one for your application to use internally. You can use easily use your resources internally:

/fragments/wiki.html:

```python
<%python
wiki = document.internal('wiki', 'application/json')
wiki = eval('(' + wiki.text + ')')
%>
Wiki content is: <%= wiki['content'] %>
```

The above is wasteful because we are turning the wiki data into JSON and then back again. For internal API uses, it's possible to support direct access to the object:

[ObjectRepresentation]

For completion, here's an example of how to access our resources from another Prudence-based application:

/fragments/wiki.html:

```python
<%python
wiki = document.external('http://192.168.1.2/wackywiki/wiki', 'application/json'
wiki = eval('(' + wiki.text + ')')
%>
Wiki content is: <%= wiki['content'] %>
```

Advanced Routing

Capturing URL patterns.
    Improve our resource above over the query params.

    By the way: Difference between URL and URI.

Virtual hosting, multiple servers.
    Status handlers—custom 404.
    Just for fun: robots.txt, sitemap.xml.

Other Features

Logging.
    Debug page.
    Defrosting and pre-heating.
    SSL. http://www.restlet.org/documentation/snapshot/jse/ext/index.html?org/restlet/ext/grizzly/HttpsServerHelper.htm
    Advanced representations. (Restlet extensions.)

Instance Configuration

Configuration by Script

Most instance and application configuration files are written in programming language code, rather than XML or other configuration formats. This method is sometimes called "boot-strapping" or just "scripting."

Scripting is powerful. It lets your configuration be dynamic, such that the same configuration script can do different things depending on the actual deployment environment and runtime circumstances. For example, you can deploy the same instance to a development environment, where it will start various debugging processes and logs, and a production environment, where it will 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 might be different from a dedicated box with 8 cores and a lot of RAM. You can dynamically test for the presence of installed optional components, etc.

The disadvantage is portability between Prudence flavors. For example, if you write your scripts in Ruby, they will not work in Prudence for Python—*unless* you manually install Python support.

The default scripts are designed to fall back on other default scripts, which are all in the /defaults/ subdirectory. 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. Or, you can edit the defaults directly to apply changes across the board.

> For changes in your configuration scripts to take effect, you will need to restart the Prudence instance. This is true for Prudence 1.0: we planto support runtime configuration in a future version of Prudence.

/defaults/

Here you'll find defaults used in /instance/ and /applications/. Generally, you'll not want to edit these, but instead override them there. Prudence tries to make configuration easy by using sensible default behavior, and making it easy to override it. However, you can always edit these files directly if necessary.

Rather than explain their use here, we will refer to them as they are used in the sections below.

/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. You can change any of these default easily.

More about configuring applications here.

/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. It simply executes /defaults/instance/.

You'll usually prefer to override the other configuration files below. Override 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

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 tasks for execution

/instance/component.*

The "component" in REST terminology refers to the highest-level communication abstraction. In essence, its all "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 from the defaults.

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, fixed-thread-pool executor (accessible as "prudence.executor" in the component's context)

5. Creates an InProcessMemory cache backend (accessible as "com.threecrickets.prudence.cache" in the component's context)

/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 individual application, it's unlikely that you'll need to override the default behavior.

    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 "applications" in the component's context)

3. If there are no applications, Prudence quits

/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. (Note that actually creating a server to listen on additional ports is a separate matter, dealt with in /instances/servers/*, below.)

    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 various HTTP

servers: Grizzly, Jetty, Netty and an "internal" connector. (To change the HTTP server technology, you have to add it to /libraries/.)

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.

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 used by proxies

2. Prints out information about this server

/bin/

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 in Unix-like systems or a service in Windows, via a lightweight wrapper. See Prudence As a Daemon.

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

   Note that if you're using JVM version 6 or above, you can use wildcards for specifying the classpath in run.sh and run.bat, instead of having to list individual JAR files.

/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 consumes and analyzed by software.

More on logging here.

/configuration/

There are a few essential configuration files here. Most important is logging.conf, but you will also find some optional files to help you configure Prudence to run as a daemon.

/libraries/

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

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

Application Configuration

Each application's subdirectory name under the /applications/ directory is used as a default for many things. Under that subdirectory is a straightforward structure, detailed below.

/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 result. Prudence has a powerful framework for embedding programming language code into HTML as "scriptlets," with support for caching, as well as textual formats other than HTML, such as XML, RSS, JSON, etc. See generating HTML.

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.

/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 client.

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 for details on configuring it, for example, in order to add/change filename-extension-to-MIME-type mappings.

/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 and be available to users.

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

/resources/

Whereas the /web/dynamic/ subdirectory has HTML files with embedded programming language code, /resources/ is pure code. This is Prudence's most flexible development feature: files here are attached as REST resources, capable of handling all HTTP verbs and responding with appropriate representations in many different formats.

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.

/libraries/

All your code, whether its in /resources/ or scriptlets 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.

Configuration Scripts

The application's configuration scripts are in its base subdirectory. See instance configuration for more details about configuration scripts.

/default.*

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

/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 Settings, below, 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. More on integrating custom restlets here.

More on routing later.

/application.*

Use this to install non-Prudence Restlet applications into Prudence. By default, Prudence creates a basic Restlet application, but you can override that creation in this file. More on this topic here.

Settings

Information

These are for administrative purposes only.

applicationName   Defaults to the application's subdirectory name

applicationDescription

applicationAuthor

applicationOwner

applicationHomeURL

applicationContactEmail

Debugging

showDebugOnError   Set to true to show debug information on error

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

Logging

applicationLoggerName   Defaults to the application's subdirectory name

Hosts

hosts   This is a vector of vectors 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. Specify null for the URL to default to the application's directory name.

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

resourcesBasePath

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.

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

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 most applications.

resourcesMinimumTimeBetweenValidityChecks   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.

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.

dynamicWebBaseURL

dynamicWebBasePath

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.

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

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

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.

dynamicWebMinimumTimeBetweenValidityChecks   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.

Static Web

Sets up a directory under which you can place static files of any type. // Servers like Grizzly and Jetty can use non-blocking I/O to stream static files efficiently to clients.

staticWebBaseURL

staticWebBasePath

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

Preheater

preheatResources   List resources here that you want heated up as soon as Prudence starts.

URL Manipulation

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

Runtime Attributes

runtimeAttributes

Generating HTML

Prudence has good support for generating HTML by allowing you to embed programming language code in it. Despite some unique and useful features, Prudence is not so much different in this than other platforms that support dynamic HTML generation. Where Prudence stands out is in its support for REST resources, which we'll deal with in the next section. Go ahead and skip to there if HTML generation bores you!

Files

Files under /web/dynamic/ are all assumed to be text files. While we're titling this section "Generating HTML," these files can, in fact, be in any textual format. The filename's extension will be used to map the default MIME type for the file, though you can easily change it in code.

> Create a file named index.html under /web/dynamic/. Open your web browser to http://localhost:8080/hello/. You should see your file rendered in the browser. Note that you did not have to enter index.html, though it would also work. Additionally, just index would work: in the /web/dynamic/ and /resources/ subdirectories, Prudence considers filename extensions to be optional for URLs.

Scriptlets

Programming language code can be embedded within your /web/dynamic/ files using either <% %> or <? ?>. Note, however, that you can only use one of either in the same file.

For historical reasons, these embedded bits of code are called "scriptlets." However, they are usually compiled, not interpreted.

Blocks can span several scriptlets (as long as you are using the same programming language: see below). For example, this will work as expected:

You can achieve the same result by printing out the text in Python code:

In fact, behind the scenes, all non-scriptlet text is turned into code. It's a simple print of the non-scriptlet text.

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. For example:

Just remember that behind the scenes the expression scriptlet is turned into a regular scriptlet, So, you need to follow all the rules of Python expressions.

Mixing Languages

By default, Prudence will assume your scriptlets to be Python code. However, Prudence lets you scriptlets of various languages by adding the language name after the first scriptlet delimiter. For example:

> You need the appropriate language JARs under your /libs/ directory for this to work.

Once you change the language, all subsequent scriptlets will default to that language. Make sure to change back if you need to.

You might think that mixing programming languages is a bad idea in general, and only necessary for making use of legacy code. However, it can be a great idea if you consider that Prudence comes with high-performance templating languages, both Succinct and Velocity. This lets you write all the more straightforward templating as

Succinct scriptlets, switching to Python scriptlets only when you need Python features. For example:

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

One tiny hiccup to be aware of is that blocks can no longer trivially span several scriptlets. For example, this won't immediately work:

The reason is that, by necessity, Prudence must run scriptlets of different languages separately, in sequence. The language switch thus represents a boundary, But, fear not, Prudence lets you solve this problem via the "in-flow" scriptlet, marked by a colon:

This works! But how, you might wonder? Behind the scenes, the in-flow scriptlet is run from within the enclosing language. We thus do 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.

> Prudence's terrific handling of scriptlets in multiple languages is internally handled by the Scripturian library. Scripturian was originally developed for Prudence, but its use is generic and useful enough that it merited separating it into a separate project. With Scripturian, you can easily add scriptlet power to your Java applications.

Includes

Life

As mentioned, files are only compiled once, when they are first requested by a client. From then on, each request is handled by the compiled version.

What if you edit the file? Prudence can 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 disable this feature for production deployments, or control the minimum time interval in which Prudence assumes the file is unchanged.

> Your compiled file can be run by multiple threads concurrently. It is up to you to avoid race conditions and guarantee thread safety.

Caching

You've chosen to use dynamically-generated HTML because you want request to be able to have different results. However, sometimes you do not expect 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 hour, instead of per 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 energy and resources to generate results that do not change.

The solution is to cache results. Sometimes even caching for 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 local temperature for our web page we need to query a service 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. Even just caching our home page for 5 seconds would throttle our service queries down to 1 every 5 seconds, vs. 500 every 5 seconds without caching. And users would get temperature readings up to 5 seconds old. It's a negligible usability with 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.

Prudence handles caching of results. Every file has its own caching interval, which you can access and change via scriptlet:

Note that you can change the cache interval 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.

By using includes, you can fine-tune your caching strategy even more. Since each file has its own caching interval, you can fragment your page into parts with different caching sensitivity. For example, you can put the temperature reading in its own fragment, with a high cache interval, with another fragment being a "number of visitors today" counter with no cache interval, always up-to-date. The home page itself could have its own interval, or none. This setup can help you think of caching problems independently, while allowing for subtle overall schemes.

TODO: default caching time in application context

Streaming

Sometimes the client expects a slow, and long response from your application. For example, a list of bank account transactions for the past year might be hundreds of rows in length. The client does not want to wait until you've produced the entire result, but can start consuming results as they come.

Streaming mode lets send results to your client as you produce them.

There are risks to streaming mode. If your scriptlet code fails along the way, the stream will stop and the client might get broken results, For example, an HTML table might not get its closing </table> tag, leading to a broken rendering on a web browser. Of course, this is a general risk, and can happen due to connection failures along the way. It's just something you need to be extra careful about in streaming mode. So, make sure you catch all exceptions and gracefully finish the request for the client in case of error.

HTML forms

The files in /web/dynamic/ are meant to be sent to the clients, in response to an HTTP GET. Later on, we'll look at files under /resources/, which can respond to all HTTP verbs. However, there is one case in which you might want to respond to HTTP POST and PUT directly in your /web/dynamic/ files: HTML form submissions. Though

hardly ideal, HTML forms are the most universally supported technique of getting input from clients, and are easy to set up.

Prudence's solution is trivial: it attaches your file to HTTP POST and PUT in addition to HTTP GET. This leaves it up to you to handle each method accordingly. Here is an example:

Handling form responses via scriplets can make your files hard to read, as they turn into a patchwork of various results all overlapped in one file. This can be cleaned up via includes:

Still, it might make more sense to remove all this form handling logic from /web/-dynamic/ to /resources/. We'll show you an example of it in the next section.

Mapping and Changing MIME Types

See this.

By extension Programmatically (text, XML, JSON)

Resources

Resources as API.

There's no need to maintain a separate "internal API," with REST resources serving as an "external API." Your resources are available internally, without having to use HTTP. A single, uniform API is more maintainable, and less prone to bugs. Additionally, the internal connectivity is an easy way to test your resources. See below for internal access to your resources.

Life

CRUD

An example of handleGet in Python:

```python
def handleGet():
        id = int(prudence.resource.request.attributes.get('id'))
        session = get_session()
        try:
                note = session.query(Note).filter_by(id=id).one()
        except NoResultFound:
                return 404 finally: session.close()
        prudence.modificationTimestamp = note.timestamp
        if prudence.mediaTypeName == 'application/json':
                return json.write(note.to_dict())
        elif if prudence.mediaTypeName == 'application/xml':
                return dict_to_xml(note.to_dict())
        else:
                return 415
```

29

Implied Representation

Error Codes

Explicit Representation

Conditional HTTP

getInfo(), modification dates and ETags

Concurrency Concerns

Mixing Languages

Accessing Resources Internally

Prudence has very easy-to-use REST client, and a rich API for creating and manipulation data representations. You can use it to access your own resources, as well as resources from other applications in your Prudence installation. Examples:

```
importClass(org.restlet.resource.ClientResource);

// A resource in our application
var r = new ClientResource('riap://application/hosts');
print(r.get().text);

// A resource from another application on our host
var id = 9;
var r = new ClientResource('riap://host/stickstick/note/' + id + '/');
var note = JSON.parse(String(r.get(MediaType.APPLICATION_JSON).text));
```

Accessing External REST Services

The client API mentioned above can also be used to access external resources from anywhere on the web.
    Examples:

```
importClass(org.restlet.resource.ClientResource);

// An external resource
var r = new ClientResource('http://threecrickets/prudence/data/rhino');
print(r.get().text);
```

Note that for the above to work, you need HTTP configured in your clients file:

```
component.clients.add(Protocol.HTTP);
```

Representations

MIME Types

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 redirecting, rewriting and routing, and that Prudence's non-blocking I/O actually does a great job at serving files. You will likely not see many performance or scalability gains replacing /web/static/ with standard Apache.

Mapping MIME Types

```
application.metadataService.addExtension('php', MediaType.TEXT_HTML)
```

Replacing Grizzly

Grizzly can be replaced with Netty, a non-blocking I/O HTTP server with slightly different performance characteristics, and Jetty, which offers similar performance, but with many more features. Or, use any other connector supported by Restlet.

CacheControlFilter

JavaScriptUnifyMinifyFilter

CssUnifyMinifyFilter

Routing

Prudence can actually run several servers at the same time, on multiple IP addresses and ports, as well as handle several web sites via virtual hosting.

"Routing" in Prudence refers to how URLs are attached to resources. In fact, this is often not a simple one-to-one relationship, and different routes may be selected according to changing circumstances.

Directories As URL Segments

Prudence provides straightforward, hierarchical, live routing between your directory structure and URLs. Simply put, file paths are mapped directly to URLs. Files added at runtime will automatically create new URLs for them. To keep the URLs pretty, filename extensions are optional. The filename extension, though serves two purposes:

In the case of code in /resources/, /libraries/ or other places, the extension selects the language engine which will run the code. For example, "search.py" will be run using Python, "search.gv" will be Groovy, "search.clj" will be Clojure, etc.

In the case of files in /web/dynamic/ and /web/static/, it helps select the default MIME type for representations. This selection is based on the meta data service, which you can customize (see static web). Embedded language code in /web/dynamic/ files can also programatically change the returns MIME types (see resources).

> When Prudence URLs are attached to files in /web/dynamic/ and /resources/, the filename extension is optional. Prudence simply grabs the first file it finds with that name. So, generally, get used to not have more than one file with the same name in the same directory, even if they have different extensions.
>
> Prudence routing rule: route by filename, format/language by extension

default.*

URI Templates

See this.

Virtual Hosts

Attaching

Redirecting

Capturing

Capturing Errors

```
// 404 handler
component.statusService.redirect(404, '{hi}/404.html', component.context);
```

API

Prudence exposes its API as services to your source code, whether its in scriptlets in /web/dynamic/, complete source code in /resources/, or your configuration scripts.

application

The same "application" service is shared between all code in a single application. Note this there is a single application instance per component, even if the application is routed on several virtual hosts and servers. The "application" service is a good place to store shared state for the application.

application.globals, getGlobal, setGlobal

application.arguments (for configuration scripts)

application.application

application.logger

application.getMediaType

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:**

document.source

document.source.basePath

document.cache

document.cache.invalidate

document.cacheKey

document.cacheDuration

document.cacheGroups

Other documents:

document.include (/web/dynamic/)

document.defaultLanguageTag (for include in startup scripts)

document.internal

document.external

executable

The "executable" is the low-level equivalent of "this document" (see above). 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 (see above), 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.context

executable.context.writer

executable.context.attributes

executable.context.exposedVariables

executable.manager

executable.container = the document!

executable.globals, getGlobal, setGlobal

conversation

The "conversation" represents the request received from the user. This includes your response to the request, hence it's a "conversation." 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.

conversation.resource

conversation.request

conversation.response

conversation.form, formAll

conversation.uriValues

conversation.characterSet Name, Extension

conversation.language Name, Extension

conversation.mediaType Name, Extension

conversation.expirationDate/Timestamp

conversation.modificationDate/Timestamp

conversation.tag httpTag

conversation.maxAge

conversation.statusCode

conversation.variant

conversation.entity

conversation.internal?

conversation.addMediaType ByName, ByExtension

conversation.deferred? (/web/dynamic/)

conversation.defer (/web/dynamic/)

conversation.locals

conversation.kaboom

Managing State

Application Globals
    Executable Globals
    Conversation Locals

Debugging

TODO

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 finds the logs useful. Prudence's servers, routers, language engines and other components all send messages to the logs, making them an invaluable tool for debugging, monitoring and understanding.

By default, logs are sent to the /logs/ directory, using a configurable rolling log scheme. web.log records all server requests, using Apache's format, while everything else goes to prudence.log.

Sending Messages

logging.conf

Appenders and Loggers

Separate Logs Per Application

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
-C'LOCALCHARTDIR local/images/' \
-C'CHARTDIR images/' \
-C'HOSTNAME "mysite.org"' \
log/web.log \
-Oapplications/myapp/web/static/analog/index.html
```

Administration

This is a simple app that lets you see the servers, virtual hosts and applications in the Prudence instance, and lets you start and stop them.

Prudence As a Daemon

In production environments, it's best to run Prudence as a daemon in Unix-like systems or a service in Windows, via a lightweight wrapper. This will provide proper 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 down and even restart it.

While Prudence comes with the necessary configuration files for this, it does not include the actual wrapper, which you will need to install manually. There are two reasons for this: YAJSW, our preferred wrapper, is too big a download as of now, and including it in Prudence make it too heavy. As for Tanuki's JSW, its license will not allow us to distribute it with Prudence. It does not, however, bar you from downloading and installing it yourself.

> Ready for a short history lesson?
>
> The copyright for JSW is held by its developer, Tanuki Software. Up to version 3.2, Tanuki released JSW under a very permissive license, making it popular in many open source projects. However, it has since 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.
>
> For less restrictive open source projects, the gap has been filled by YAJSW, which seeks not only to be a drop-in replacement for JSW, but also to go beyond it with many additional features.

YAJSW

YAJSW is written in 100% Java, using JNA to handle the native operating-system-dependent features. It is a powerful tool with many great features.

Monitoring via JMX

JSW

JSW is written in C rather than Java, making it more lightweight than YAJSW. Nevertheless, it has binaries for many operating systems.

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 documentation 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 to dramatically scale 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.

There are many great load balancers out there, but we especially like Perlbal. Not only is it easy to use and configure, but it also adds a cool feature: instances behind Perlbal can ask it to reroute specific requests. [In the future, we will show you how to do this in Prudence.]

Apache

Apache is often called the "Swiss army knife of the Internet" for how well it manipulates URLs and routes HTTP requests. Prudence already does powerful routing, including virtual hosting, and that this may be good enough for you. However, Apache is also used a container environment for application platforms as modules, 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 as a reverse proxy.

Prudence As a Restlet Container

Prudence brings the power of REST and the JVM to programmers in other languages, but has a lot to offer to Java/Restlet developers.

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 just restlets, written in Java or other JVM languages.

Why use Prudence for a Restlet application that already works?

Prudence makes it easy to handle the bootstrapping and routing of your applications, and the Prudence administration application, debugging and logging 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 plopping in your jar.

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.

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 support. It may likely surpass JSP for your purposes. In particular, it is based on Restlet, which you already know and love, with the entire Restlet API at your fingertips. It also lets you use many wonderful languages other than Java for scriptlets. For example, we at Three Crickets, are mad about Clojure. And, of course, Velocity and Succinct are built in.

Summary

A 100% Restlet-based alternative to servlet/JEE containers. (Requires only Restlet SE.)

1. Easy deployment

    (a) Boot scripts: avoid weird configuration formats and start your Restlet component exactly as you want (in your choice of scripting language)

    (b) Default scripts already handle things like virtual hosting, multiple servers and internal routing

    (c) Designed from the ground-up to handle multiple apps on the same component

    (d) Admin app for realtime management of server(s)

    (e) Logging is pre-configured and "just works," including Apache-compatible web log

    (f) Single-file "plop" deployment (like WAR files)

2. Easy prototyping of REST resources

    (a) In the language of your choice

    (b) Code is compiled, cached and loaded on-the-fly

    (c) Rich debug page for runtime errors

    (d) When you're happy with it, you can rewrite it as a ServerResource in Java

3. Powerful HTML generation platform, like JSP/ASP/PHP (again, 100% Restlet-based)

    (a) In the language of your choice, including mixing languages and template engines (Velocity, Succinct) on one page

(b) Code is compiled, cached and loaded on-the-fly

(c) RAM/db server-side caching (uses Restlet's URI template language for cache key generation)

(d) Straightforward support for client-side caching

(e) Easily "stream" output (Restlet async)

(f) Easily accept uploaded files

(g) Rich debug page for runtime errors

4. Restlet sugar

(a) Fallback router (attach multiple MODE_STARTS_WITH restlets to the same base URI)

(b) URI "capturing" (internal redirection; let the URI-space you expose to the world be different from the one you use internally)

(c) JavaScript and CSS unify-and-minify filter

(d) Delegated status service for displaying custom pages for error codes (404, 500, etc.)

(e) Rich debugging representation

(f) Caching system, with pluggable backends, designed for StringRepresenta-tions

(g) Easy file uploads (slightly higher-level than the Restlet FileUpload exten-sion)

Custom Resources

Use your application's routing file to attach your resources, or otherwise manage rout-ing. Example:

```
// 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:

```
// Wrap static web in JavaScript minifying filter
importClass(org.myorg.JavaScriptMinifyFilter);
router.detach(staticWeb);
router.attach(
        fixURL(staticWebBaseURL),
        new JavaScriptMinifyFilter(application.context, staticWeb, File(applicati
        .matchingMode = Template.MODE_STARTS_WITH;
```

Custom Application

By default, Prudence creates an instance of the standard Restlet Application class. Use your application's application file 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 MyOrgApplication
application.databaseURI = 'mysql://localhost/myorg';
application.useTransactions = true;
```

How to Choose a Flavor?

Python (Succulent!)

Python is a powerful object-oriented language, far richer in features than JavaScript and PHP, with many high-quality core and 3rd party libraries. Python has already proven itself as a web programming language with many excellent platforms.

Python presents a unique challenge in a scriptlet environment, due to its reliance on indentation. 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 the included example application we show how to use SQLAlchemy as a data backend for Prudence.

> Note: Prudence 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 scaling, Clojure is the way to go. Though new, Clojure is based on one of the oldest programming languages around, and enjoys a 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. In the included example application, we show how to use Clojure's SQL library to access a data backend the Clojure way.

Three Crickets ♥ Clojure.

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. JavaScript lets you to write server-side and client-side code in the same language, and have both sides share code. Couple it with JSON, and you're on solid ground for rapid development. Of all the web programming languages, it's the one most widely deployed and with the most secure future.

JavaScript is an under-appreciated language. Though not as feature-rich as Python or Ruby, it's still very powerful. Its straightforward closure/prototype mechanisms allow it to support object-orientation, namespaces and other paradigms. It's also been the target of much un-deserved angst due to the fickleness of in-browser development. Working with the DOM, cross-browser HTML rendering quirks—none of these are the faults of the language itself, and none of these are relevant to server-side development with Prudence.

JavaScript does not have its own core libraries, making it the most minimal Prudence flavor. But this isn't necessarily a deficiency: instead, it uses the excellent JVM core.

PHP (Ambrosial!)

PHPis, of course, 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 it 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.

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

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. The essay attempts to convey its simple essence.

Let's start, then, not at REST, but at an attempt to create a new architecture for building scalabale 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 enormous complexity. Instead, then, we'll keep it simple and define a limited interface that would still be 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 discriminating 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/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 order of complexity.

Delete

"Delete" is the most trivial operation. After sending "delete" to an identifier, we expect it to not exist anymore. Whether or not sub-resources in our hierarchy can exist or not, we'd 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 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 we expect our resource to change, 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.

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.

We'll need our resource-oriented architecture to support a "conditional" feature.

Create

This is our most complex operation. Our first problem is that our identifier might not exist yet. 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 IDs that are never used and 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:

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

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

The above would 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 this 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 "/animal/cats". It can be very dumb, and easy 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. We would not want a series of wasteful operations to happen until one succeeds. 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. A better solution would be to have the client associate certain preferences per command, have our resource emit its capabilities, with mapping component in between "negotiating" these two lists, via a simple algorithm, and 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.

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 worldwide. 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. Importantly, this 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 non-URL kinds of URIs.

It's All About Infrastructure

The most important lesson to take from this experience is the importance of infrastructure. This is why, I believe, Roy Fielding named Chapter 5 of his 2000 dissertation "Representational State Transfer" 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 is intended to point out the key characteristic of its infrastructure: it's all about the transfer of 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 put it succinctly, 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 the above.

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 term "scalable":

First, it's the ability to respond to a growing number of user requests without degradation in response time, by "simply" adding hardware. 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 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, 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 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 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 Prudence tutorial, 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!

Scaling Tips

If you want your application to handle many concurrent users, then you are 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.

Two variables influence this: 1) your total number of threads and 2) the time it takes each thread to process a request. Increasing the number of threads is straightforward: you keep adding more and more machines behind load balancers. The only challenge in this respect is financial. On the other hand, the time per request has little to do with your computer power. In fact, for various reasons it can grow longer as you keep adding threads. Minimizing the time per request is a design challenge.

Your design objective is to increase concurrency, not necessarily performance. Optimizing for concurrency means breaking up tasks to as many pieces as possible, and possibly even breaking requests into smaller pieces. We'll cover numerous strategies here.

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.

Finally, a favorite:

> Performance does not equal scalability.
> Performance does not equal scalability.
> Performance does not equal scalability.

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 cacheKey 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 cacheKey. Fragments, though, may very well appear identically in many different URIs. You would thus not want the URI as part of their cacheKey.

Cache aggressively, but also take cache validation seriously. Make good use of Prudence's cacheGroups 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 cacheGroups don't include 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 deferrable task handler, you might want to invalidate cache groups 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.

Necessary Tasks

The user can't continue without the task being resolved.

If the necessary task is deterministically fast, you can do all processing in the request iself.

If not, you should queue the task on a handling service and return a "please wait" page to the user. It woule 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. The client will poll until the task status is marked "done," after which they will be redirected somewhere else. 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").

Implementing a handling service is by no means 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 tast to complete. The bottom line, though, is you're vastly increasing concurrency and your ability to scale. And, you're actually improving the user experience: they would get a feedback on what's going on rather than having their browsers spin, waiting for their requests to complete.

Deferrable Tasks

It's OK if the task occurs later.

As with necessary tasks, you can queue these with a task handling service, but it's much simpler because you don't have to keep track of status or display it to the user. This allows you to use a simpler—and more scalable—task handling service.

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 is deferrable should actually be considered necessary?

Uploads

File uploads are very long tasks 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.

Unfortunately, many concurrent uploads will still saturate your threads. If your application relies on 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.

Asynchronous 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 be 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 that do a few independent operations.

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 asynchronous processing is not such a useful strategy for increasing concurrency in web applications. Caching is far more useful.

As of version 1.0, Prudence has limited support for asynchronous processing, via conversation.defer.

Data Backends

You can keeping 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.

If the challenge in handling web requests is cutting down the length of request, then that of relational data backends is the struggle against degraded performance as you add new nodes to your database cluster. These nodes have to be synchronized with each other, and that synchronization overhead increases exponentially. There's a definite point of diminishing returns.

Fortunately, there are workable strategies for scaling data backends. Unfortunately, they all have limitations, and must be carefully selected for your application's needs.

Graph Databases

If your relational data structure contains a arbitrary-depth relationships or "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 scalability. The improvement over relational databases 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, 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 perfect for Prudence. Because it's JVM-based, you can access it internally from Prudence. It also supports a 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 almost limitlessly scalable.

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. 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.

Best of All Worlds

Of course, consider that it's very possible to use both SQL and "NoSQL" (graph, document) databases together for different parts of your application.

Powering Up

The database 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.

Partitioning

Partitioning is as useful to relational database scaling as caching is to web request scaling.

Rather than having one big cluster of identical databases, you will have several smaller clusters of different, independent databases. 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.

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 them 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. Choose the best backend for each 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 straight-forward 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

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 hardcode 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 rather than access the database directly. Unfortunately, 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 refactoring 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, irganizing code together by mechanism, rather than by data "class," will let you apply all kinds of refactorizations more easily, especially if you manage to decouple your application data structures from any library-specific data structures.

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.

Restlet

Prudence went through many 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. Clean abstraction of HTTP requests, responses and headers over best-of-breed engines, such as Grizzly, Jetty 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 labor.

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.

The JVM

How wonderful that the best-performing, most robust 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.

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.

Scripturian

To make Prudence applications easy to deploy, they could not be traditional Java. The cycle of compilation and packaging is unnecessarily cumbersome. Though we could have implemented something like the on-the-fly Java compilation done in JSP, but we felt that, if that's the route to go, many choices open up besides Java, and that these languages are more relevant to Prudence's goals.

There are many terrific JVM-based languages out there. Unfortunately, we found that integrating these languages into Prudence was anything but trivial. Each implementation had its own idea of what was acceptable support for embedding. We tried to standardize on JSR-223 (the Java scripting standard), but found that adherence to the spec was inconsistent. 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.

Since we think software other than Prudence can make use of our hard-won gains, we've abstracted and separated our language integration code as the Scripturian library.

Succinct

Succinct also started as a part of Prudence (one of its earliest components) that 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'll like it.

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.

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

Why mix languages?

What are "in-flow" scriptlets?

Can different languages share state?

Can languages call each other's functions?

Concurrency

Should I be worried?

How to make my code thread safe?

Performance

Which Prudence flavor performs best?

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 address in the Making the Case for REST.

From the perspective of the ability to respond to user requests, there are three aspects to consider:

1. Serving HTTP   Prudence comes with Grizzly, an HTTP server based on the JVM's non-blocking I/O API. Grizzly handles concurrent HTTP requests very well, and serves static files at scales comparable to popular HTTP servers. See the tutorial for more information.

2. Generating HTML   Prudence supports two modes for generating HTML (and other textual formats), each with its own performance characteristics:

Caching mode: First, the entire document is run, with its output sent into a buffer. This buffer is then cached, and *only then* sent to the client. This is the default mode and recommended for most documents. Scriptlets can be used to control the duration of the document's individual cache.

Streaming mode: Output is sent to the client *while* the document runs. This is recommended for documents that need to output a very large amount of text, which might take a long time, or that might otherwise encounter slow-downs while running. In either case, you want the client to receive ongoing output. The output of the document is not cached. Scriptlets can switch between modes according to changing circumstances. For example, to increase caching duration during heavy loads, to decrease it during periods where data changes often, or to stream in the case of an expected large output. See the tutorial for more information.

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.

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.

Scalability

Do you have any tips?

Yes. :)

Licensing

(The author is not a lawyer. This is not a legal advice, but a personal interpretation. The wording of the license itself superceds 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 Three Crickets, the copyright holder).

The GPL would thus only be an issue if you're selling, or even giving away, a product that *would include* Prudence.

Prudence uses the Lesser GPL, which has even less restrictions on redistribution than the regular GPL. As long as you do not alter Prudence in any way, you can include Prudence in any product, free or non-free. (Actually, Prudence uses version 3 of the Lesser GPL, which requires your product, even if it's not free software, to at least not restrict users' ownership of data via schemes such as DRM if you want to include Prudence 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.

Three Crickets, the original developers of Prudence, are not trying to force you to purchase it. Instead, they hope to encourage you 1) to pay Three Crickets for consultation, support and development services for Prudence, and 2) to release your own product as free software, thereby truly sharing your innovation with all of society.

We understand that in some cases open sourcing your product is impossible. As a last resort, we offer you a commercial license. Please contact us for details.

Why the LGPL and not the GPL?

The Lesser GPL used to be called the "Library GPL," and was originally drafted for glibc. It represents a certain admission of defeat: there are so many alternatives to our library out there, that you might not consider using our library under GPL.

In the case of Linux, the GPL has done a wonderful job in convincing vendors to open source their code in order to ship their products with Linux inside. It just 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.