

The Prudence Manual

Version 2.0-dev14

Main text written by Tal Liron

November 10, 2013

Copyright 2009-2013 by Three Crickets LLC.

This work is licensed under a
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Contents

I Basic Manual	6
Tutorial	6
Installing Prudence	6
At a Glance	6
Your First Application	6
Generating HTML	7
A Manual Resource	7
The URI-space	7
routing.js	7
app.routes	8
Two Routing Paradigms	12
Resource Mapping	12
URI/Resource Separation	14
app.hosts	17
app.errors	17
app.dispatchers	18
app.preheat	19
Implementing Resources	19
Manual Resources	20
Scriptlet Resources	23
Static Resources	24
Resource Type Comparison Table	28
Web Data	28
URLs	29
Request Payloads	30
Cookies	31
Redirection	32
HTML Forms	33
Response Payloads	34
External Requests	37
Caching	38
Introduction: Integrated Caching	38
Server-Side Caching	38
Client-Side Caching	38
Two Caching Strategies	39
Configuring Applications	41
settings.js	41
app.settings.description	42
app.settings.errors	42
app.settings.code	43
app.settings.uploads	43
app.settings.mediaTypes	43
app.globals	44

Programming	44
Powered by Scripturian	44
JavaScript	44
Other Languages	44
State	44
APIs	47
Execution Environments	49
Debugging	50
Logging	50
Debug Page	50
Live Execution	50
FAQ	50
Technology	50
Performance and Scalability	52
Errors	53
Licensing	53
 II Advanced Manual	 54
The Internal URI-space	54
Internal Requests	54
Configuring the Internal URI-space	54
Implementing Internal Resources	54
String Interpolation	54
Request URIs	55
Request Attributes	56
Client Attributes	56
Payload Attributes	56
Response Attributes	56
conversation.locals	56
Model-View-Controller (MVC)	57
Background	57
Tutorial	57
View Templates	60
RESTful Models	67
Background Tasks	69
crontab	69
startup	69
Task API	69
Filtering	69
Tutorial	69
Examples	70
Built-in Filters	72
Configuring the Component	73
/component/servers/	73
/component/clients/	73
/component/hosts/	73
/component/services/	73
/component/templates/	74

Servers and Hosts	74
Configuring Servers	74
Configuring Hosts	75
Routing In-Depth	76
Working in a Cluster	77
Distributed Globals	78
Task Farming	78
Deploying	78
The Joys of Sincerity	78
Deployment Strategies	78
Directory Organization	78
Operating System Service	78
Monitoring	78
Security	78
Proxying	79
Deploying Clusters	79
Cache Backends	79
Utilities for Restlet	79
Utility Restlets	79
Client Data	79
Redirection	80
Fallback Routing	80
Resolver Selection	80
Web Filters	80
Upgrading from Prudence 1.1	80
Upgrading Applications	81
Upgrading the Component	81
III Articles	81
The Case for REST	81
Resources	82
Identifiers	82
Delete	82
Read	82
Update	82
Create	82
Aggregate Resources	83
Formats	83
Shared State	84
Summary of Features	84
Transactions... Not!	84
Let's Do It!	84
The Punchline	84
It's All About Infrastructure	85
Does REST Scale?	85
Prudence	85

URI-space Architecture	86
Nouns vs. Verbs	86
Do You Really Need REST?	86
Hierarchies	87
Formats	87
Plural vs. Singular	87
Documenting Your URI-Space	88
Scaling Tips	89
Performance Does Not Equal Scalability	89
Caching	91
Dealing with Lengthy Requests	94
Backend Partitioning	96
Data Backends	98

Part I

Basic Manual

Tutorial

Installing Prudence

Download a distribution
Starting Prudence

Further Exploration

- Start Prudence as a system service
- Logging

At a Glance

/component/

A “component” is the largest logical entity REST. It can encapsulate many servers and clients.

In Prudence, the component is bootstrapped using straightforward JavaScript code, starting with `default.js`. The code makes sure to initialize all your applications, servers and clients, as well as related services, and bind them to your component.

/component/applications/

/component/libraries/scripturian/

/component/libraries/prudence/

/component/libraries/web/

Here you can put static resources that you wish to have shared by all your applications.

It’s a good place to put client-side JavaScript frameworks such as Ext JS and jQuery.

/component/libraries/jars/

This is where JVM libraries are installed. This includes all of Prudence’s dependencies, and you can add your own, too.

You can use JVM APIs from JavaScript almost identically to how they are used in Java.

/cache/

Further Exploration

- Learn about bootstrapping with Sincerity
- Configure your component

Your First Application

The “prudence” Command

/resources/

/libraries/

/libraries/

Further Exploration

- Configure your application

- Managing the URI-space

Generating HTML

Further Exploration

- Textual Resources
- Adding more languages

A Manual Resource

Further Exploration

- Manual Resources

The URI-space

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

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

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

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

Make sure to also read the [URI-space Architecture](#) article (page 86), which discusses general architectural issues.

routing.js

Routing is configured in your application’s `routing.js` file. The file should configure at least [app.routes](#) (page 8) and likely [app.hosts](#) (page 17). Add [app.errors](#) (page 17), [app.dispatchers](#) (page 18), and [app.preheat](#) (page 19) if you are using those optional features.

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

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

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

app.routes = {
  '/*': [
    'manual',
    'scriptlet',
    {
      type: 'cacheControl',
      mediaTypes: {
```

```

        'image/*': '1m',
        'text/css': '1m',
        'application/x-javascript': '1m'
      },
      next: {
        type: 'zuss',
        next: 'static'
      }
    }
  ],
  '/example1/': '@example', // (dispatched)
  '/example2/': '/example/' // (captured)
}

app.dispatchers = {
  javascript: '/resources/'
}

```

In these settings, note that time durations are in milliseconds and data sizes in bytes. These can be specified as either numbers or strings (page 42). Examples: “1.5m” is 90000 milliseconds and “1kb” is 1024 bytes.

app.routes

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

URI Templates

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

- **Variables** are strings wrapped in curly brackets. For example, here is a URI template with two variables: “/profile/{user}/{service}/”. The variables will match any text until the next “/”. You can access the string values of these variables in your resource as [conversation.locals](#) (page 45).
- A **wildcard** can be used as the last character in the URI template. For example, “/archive/*” will match *any* URI that begins with “/archive/”. You can access the remainder of the URI via the conversation.reference.remainingPart API. Note that Prudence will attempt to match *non-wildcard* URI templates first, so a wildcard URI template can be used as a general fallback for URIs.

Route Configurations

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

As a shortcut, you can just use a string value (the “type” name) instead of a full dict, however when used this way you must accept the default configuration. There are also special alternate forms for some of the commonly used types, such as “!” for the “hidden” type and JavaScript arrays for the “chain” type.

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

Resource Route Types

These are routes to a single resource implementation.

dispatch or “@” Use the “dispatch” type with an “id” param, or any string starting with the “@” character, to configure a dispatch mapping. For example, {type: 'dispatch', id: 'person'} is identical to '@person'. If you use “@”, you can also optionally use a “.” to specify the “dispatcher” param, for example: “@profile:person” is identical to {type: 'dispatch', dispatcher: 'profile', id: 'person'}. If “dispatcher” is not specified, it defaults to “javascript”. The unique ID should match a manual resource handled by your dispatcher, otherwise a 404 error (“not found”) will result. The “dispatcher” param’s value can be any key from the app.dispatchers dict. Handled by the Prudence.Routing.Dispatch class. See the [manual resource guide \(page 20\)](#) for a complete discussion.

The “manual” route type ([page 9](#)) is internally used by Prudence to handle the “dispatch” route type, via a server-side redirect. This introduces two special limitations on its use. First, it means that you *must* have a “manual” if you want to use “dispatch.” Second, you must make sure the “manual” always appears *before* any use of “dispatch” in app.routes. For example, if you attach the manual to “/*” in a chain (as in the default application template), and you also want to add a “dispatch” to that chain, you need to put the “manual” *before* the “dispatch” in the chain. Otherwise, you might cause an endless server-side redirect, leading to a stack overflow error. Example of correct use:

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

hidden or “!” Use the “hidden” or “!” string values to hide a URI template. Prudence will always return a 404 error (“not found”) for this match. Note that internal requests always bypass this limitation, and so this functionality is useful if you want some URIs available in the internal URI-space but not the public one. This special value is not actually handled by a class, but rather is configured into the current router. See also [capture-and-hide \(page 13\)](#) for another notation.

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

execute Use the “execute” type to attach a code execution resource. This powerful (and dangerous) resource executes all POST payloads as Scripturian text-with-scriptlets documents. The standard output of the script will be returned as a response. Because it always execution of arbitrary code, you very likely do not want this resource publicly exposed. If you use it, make sure to protect its URL on publicly available machines! Handled by the Prudence.Routing.Execute class. The “execute” resource is very useful for [debugging \(page 50\)](#).

Mapping Route Types

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

static Use the “static” type to create a static resource handler. By default uses the application’s “/resources/” subdirectory chained to the container’s “/libraries/web/” subdirectory for its “roots”. Note that if you include it in a “chain” with “manual” and/or “scriptlet”, then “static” should be the last entry in the chain. Handled by the Prudence.Routing.Static class. See the [static resources guide \(page 24\)](#) for a complete discussion.

manual Use the “manual” type to create a static resource handler. By default uses the application’s “/resources/” subdirectory for its “root”. Important limitation: *All* uses of this class in the same application share the same configuration. Only the first found configuration will take hold and will be shared by other instances. Handled by the Prudence.Routing.Manual class. See the [manual resources guide \(page 20\)](#) for a complete discussion.

scriptlet Use the “scriptlet” type to create a static resource handler. By default uses the application’s “/resources/” subdirectory for its “root”. Important limitation: *All* uses of this class in the same application share the same configuration. Only the first found configuration will take hold and will be shared by other instances. Handled by the Prudence.Routing.Scriptlet class. See the [scriptlet resources guide \(page 23\)](#) for a complete discussion.

Redirecting Route Types

capture or “/...” Use the “capture” type with a “uri” param, or any string starting with the “/” character, to configure a capture. For example, {type: ‘capture’, uri: ‘/user/profile/’} is identical to ‘/user/profile/’. Note that adding a “!” character at the end of the URI (not considered as part of the actual target URI) is a shortcut for *also* hiding the target URI. Capturing-and-hiding is a common use case. Handled by the Prudence.Routing.Capture class. See [resource capturing \(page 14\)](#) for a complete discussion.

redirect or “>...” Use the “redirect” type with a “uri” param, or any string starting with the “>” character, to asks the client to redirect (repeat its request) to a new URI. For example, {type: ‘redirect’, uri: ‘http://newsite.org/user/profile/’} is identical to ‘>http://newsite.org/user/profile/’. Handled by the Prudence.Routing.Redirect class. See the [web data guide \(page 32\)](#) for a complete discussion, as well as other options for redirection.

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

Combining Route Types

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

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

Filtering Route Types

All these route types require a “next” param for nesting into another route type. See the [filtering guide \(page 69\)](#) for a complete discussion.

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

injector Use the “injector” type with the “locals” and “next” params to create an injector. An injector is a simple filter that injects preset valued into [conversation.locals \(page 45\)](#), but otherwise has no effect on the conversation. This is useful for Inversion of Control (IoC): you can use these conversation.locals to alter the behavior of nested route types directly in your routing.js. Handled by the Prudence.Routing.Injector class.

httpAuthenticator Use the “httpAuthenticator” with the “credentials”, “realm” and “next” params to require HTTP authentication before allow the request to go through. This straightforward (but weak and inflexible) security mechanism is useful for ensuring that robots, such as search engine crawlers, as well as unauthorized users do not access a URI. Handled by the Prudence.Routing.HttpAuthenticator class. See the [HTTP authentication guide \(page 72\)](#) for a complete discussion.

cacheControl Use the “cacheControl” type with a “next” param to create a cache control filter. “next” is any route type configuration, allowing for nesting. Handled by the `Prudence.Routing.CacheControl` class. See the [static resources guide \(page 25\)](#) for a complete discussion.

javaScriptUnifyMinify Use the “javaScriptUnifyMinify” type with a “next” param to create a JavaScript unify/minify filter. “roots” defaults to your application’s “/resources/scripts/” and your container’s “/libraries/web/scripts/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the `Prudence.Routing.JavaScriptUnifyMinify` class. See the [static resources guide \(page 26\)](#) for a complete discussion.

cssUnifyMinify Use the “cssScriptUnifyMinify” type with a “next” param to create a CSS unify/minify filter. “roots” defaults to your application’s “/resources/style/” and your container’s “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the `Prudence.Routing.CssUnifyMinify` class. See the [static resources guide \(page 26\)](#) for a complete discussion.

zuss Use the “zuss” type with a “next” param to create a ZUSS compiling filter. “roots” defaults to your application’s “/resources/style/” and your container’s “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the `Prudence.Routing.Zuss` class. See the [static resources guide \(page 27\)](#) for a complete discussion.

Custom Route Types

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

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

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

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

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

    Public._inherit = Prudence.Routing.Restlet
    Public._configure = [ 'uri' ]

    Public.create = function(app, uri) {
        importClass(org.restlet.routing.Redirector)
        var redirector = new Redirector(app.context, this.uri, Redirector.MOVED_PERMANENTLY_303)
        return redirector
    }

    return Public
})();

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

Two Routing Paradigms

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

1. **Resource mapping** (page 12): The filesystem hierarchy under an application's `"/resources/"` subdirectory is directly mapped to URIs (but not URI templates). Both directory- and file-names are mapped in order of depth. By default, Prudence hides filename extensions from the published URIs, but uses these extensions to extract MIME-type information for the resources. Also, mapping adds trailing slashes by default, by redirecting URIs without trailing slash to include them (on the client's side). Filesystem mapping provides the most "transparent" management of your URI-space, because you do not need to edit any configuration file: to change URIs, you simply move or rename files and directories.
2. **URI/resource separation:**
 - (a) **Resource capturing** (page 14): Capturing lets you map URI templates to fixed URIs, as well as perform other kinds of internal URI rewrites that are invisible to clients, allowing you to provide a published URI-space, which is different from your internal mapping structure. Note that another common use for capturing is to add support for URI templates in resource mapping, as is explained in [resource mapping](#) (page 12). This use case does not belong to the URI/resource separation paradigm.
 - (b) **Resource dispatching** (page 15): In your application's `routing.js` you can map URIs or URI templates to a custom ID, which is then dispatched to your resource handling code. Dispatching provides the cleanest and most flexible separation between URIs and their implementation.

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

Resource Mapping

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

Prudence differs from the familiar paradigm in three ways:

1. For manual and scriptlet resources, Prudence hides filename extensions from the URIs by default. Thus, `"/resources/myresource.m.js"` would be mapped to `"/resources/myresource/"`. The reasons are two: 1) clients should not have to know about your internal implementation of the resource, and 2) it allows for cleaner and more coherent URIs. Note the filename extensions are used internally by Prudence (differently for manual and scriptlet resources). Note that this does not apply to static resources: `"/resources/images/logo.png"` will be mapped to `"/images/logo.png"`.
2. For manual and scriptlet resources, Prudence by default *requires* trailing slashes for URIs. If clients do not include the trailing slash, they will receive a 404 ("not found") error. Again, the reasons are two: 1) it makes relative URIs always unambiguous, which is especially relevant in HTML and CSS, and 2) it clarifies the extent of URI template variables. As a courtesy to sloppy clients, you can manually add a permanent redirection to a trailing slash, using the ["addSlash" route type](#) (page 10). For example:

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

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

Mapping URI Templates

The problem with resource mapping is that the URIs are “static”: they are only and exactly the the directory and file path. However, this limitation is easily overcome by Prudence’s “capturing” mechanism, which works on URI templates. For example, let’s say you have a scriptlet resource file at “/resources/user/profile.s.html”, but instead of it being mapped to the URI “/user/profile/”, you want to access it via a URI template: “/user/profile/{userId}/”:

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

A URI such as “/user/profile/4431/” would then be internally redirected to the “/user/profile/” URI. Within your “profile.s.html” file, you could then access the captured value as conversation.locals (page 45):

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

We’ve used a scriptlet resource in this example, but capturing can be used for both scriptlet and manual resources. The same conversation.locals API (page 45) is used in both cases.

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

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

That final “!” implies hiding. You of course can also configure capturing and hiding separately, using the “hidden” route type (page 9). The following is equivalent to the above:

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

Dynamic Capturing

URI capturing can actually do more than just capture to a *single* URI: the target URI for a capture is, in fact, *also* a URI template, and can include any of the conversation attributes discussed in the string interpolation guide (page 54). For example:

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

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

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

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

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

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

Limitations of Resource Mapping

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

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

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

URI/Resource Separation

Resource Capturing

Resource capturing, for the purpose of the URI/resource separation paradigm, only makes sense for scriptlet resources. For manual resources, [resource dispatching \(page 15\)](#) provides a similar structural function.

Resource capturing lets you use any public URI for any library scriptlet resource. For example, let’s assume that you have the following files in “/libraries/includes/”: “/database/profile.html”, “/database/preferences.html” and “/cache/session.html”, which you organized in subdirectories according to the technologies used. Your URI-space can be defined thus, using the [“capture” route type \(page 10\)](#):

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

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

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

Note that it’s also possible to [capture wildcards \(page 55\)](#).

Resource Dispatching

Resource dispatching, for the purpose of the URI/resource separation paradigm, only makes sense for manual resources. For scriptlet resources, [resource capturing \(page 14\)](#) provides a similar structural function.

Configuring a dispatch is straightforward. In `routing.js`, use the “[dispatch](#)” route type (page 9), or the “@” shortcut:

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

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

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

Note that, similarly to capturing, you can [interpolate conversation attributes \(page 54\)](#) into the ID. For example, here we will automatically use a different dispatch ID according the protocol, either “session.HTTP” or “session.HTTPS”:

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

You must also configure your dispatchers. There is one dispatcher per programming language. We can configure it like so:

```
app.dispatchers = {
  javascript: '/dispatched /'
}
```

The “/dispatched/” value is the document name to be executed from your application’s “/libraries/” subdirectory. In our case, we must thus also have a “/libraries/dispatched.js” file:

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

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

```

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

```

The dispatcher will execute the above library and look for the “resources” dict, which maps dispatch IDs to resource implementations. In our example we’ve mapped the “session” dispatch ID to a dict, and used simple JavaScript object-oriented programming for the “user” dispatch ID. (Note that the Sincerity.Classes facility offers a comprehensive object-oriented system for JavaScript, but we preferred more straightforward code for this example.)

As you can see, the resources.js file does not refer to URLs, but instead to dispatch IDs, which you can dispatch as you see fit.

Other Programming Languages Resource dispatching is also supported for Python, Ruby, PHP, Lua, Groovy and Clojure. To use them, you must specify the dispatcher together with the dispatch ID in routing.js, and configure that specific dispatcher. For example:

```

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

app.dispatchers = {
  ...
  python: '/resources/'
}

```

Inversion of Control (IoC)

Object-oriented inheritance is one useful way to reuse code while allowing for special implementations. Additionally, Prudence allows for a straightforward Inversion of Control mechanism (page 72).

For both capturing and dispatching, you can inject set values to conversation.locals (page 45). You would need to use the long-form notation to do this. Here are examples for both a capture and a dispatch:

```

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

```

Note that in each case two URI templates are captured/matched to the exact same resource, but the “locals” dict used is different for each. In your resource implementations, you can then allow for different behavior according to the value of the set conversation.local (page 45). For example:

```

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

```



```

    }
}

```

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

app.hosts

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

The default configuration uses a single host, the “default” one, but you can add more, or even not have a single “default” host. Still, note that only a *single* application instance is used, even if attached to multiple hosts: for example, all [application.globals](#) (page 44) are shared no matter which host a request is routed from.

To configure your hosts, and for a more complete discussion, see the [servers and hosts guide](#) (page 75).

The Bare Minimum

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

```

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

```

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

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

The Internal Host

“internal” is a reserved host name, used to represent the [internal URI-space](#) (page 54).

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

```

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

```

Multiple Hosts

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

```

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

```

If you need your application to sometimes behave differently on each host, you *might* be able to *indirectly* discover the host that routed the request using the `conversation.reference.baseRef` API. This will always be the application’s root URI. For example:

```

if (conversation.reference.baseRef.path === '/myapp/') {
  ...
}

```

app.errors

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

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

The keys of the dict are status codes, while the values work like [captures \(page 14\)](#). This means that you can implement your error pages using any kind of resource: manual, scriptlet or static.

If your error resource is mapped under “/resources/”, and you don’t want it exposed, use the [capture-and-hide notation \(page 13\)](#):

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

This is equivalent to explicitly hiding the URI in app.routes using the [“hidden” route type \(page 9\)](#):

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

Note that 500 (“internal server error”) statuses caused by uncaught exceptions can be handled specially by [enabling debug mode \(page 42\)](#).

Warning: If you decide to implement your error pages using a *non*-static resource, you run the risk of the resource code throwing an exception. If that happens, it would cause a 500 status code. If the original error was *not* 500, then this would confuse the client and complicate debugging. However, if the original status code *was* 500, then it could be much worse: the error would trigger your error page to be displayed again, which might throw the exception again... resulting in a stack overflow error. For these reasons, it’s probably a good idea to implement your error pages as static resources, at least for production deployments.

app.dispatchers

The common use of app.dispatchers has already been [discussed above \(page 15\)](#). Here, we’ll delve into more advanced configurations.

Custom Dispatchers

Custom dispatchers are very useful for [integrating alternative templating engines \(page 60\)](#) into Prudence.

Under the hood, resource dispatching is handled by the URI capturing mechanism: the URI is captured to a special manual resource—the “dispatcher”—with an [injected value \(page 72\)](#) specifying the ID of resource to which it should dispatch.

Prudence’s default dispatchers can be found in the “/libraries/scripturian/prudence/dispatchers/” directory of your container. For example, the JavaScript dispatcher is “/libraries/scripturian/prudence/dispatchers/-javascript.js”. You are encouraged to look at the code there in order to understand how dispatching works: it’s quite straightforward delegation of the [entry points of a manual resource \(page 20\)](#).

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

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

The dispatcher code (a standard manual resource) would be in a “/libraries/dispatchers/special-dispatcher.js” file under your application’s subdirectory. You would be able to access the dispatch ID there as the injected “prudence.dispatcher.id” [conversation.local \(page 45\)](#). “customValue1” and “customValue2” would be [application.globals \(page 44\)](#): “prudence.dispatcher.special.customValue1” and “prudence.dispatcher.special.customValue2” respectively.

You can also override the default dispatchers:

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

See the section on [integrating alternative templating engines \(page 60\)](#) for a complete example.

app.preheat

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

Another source for delay is that, if you are [caching pages \(page 38\)](#), as you very well should, then the some cached pages may not exist or be old. The first hit would thus need to generate and cache the page, which is much slower than using the cached version. For very large applications optimized for caching, a “cold” cache can cause serious problems: see the discussion in [Scaling Tips \(page 91\)](#).

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

- **Defrosting:** Prudence will by default parse and compile the code for manual and scriptlet resources under your application’s “/resources/” subdirectory. See [configuring applications \(page 43\)](#) if you wish to turn this feature off.
- **Preheating:** This causes an [internal \(page 54\)](#) “GET” on URIs, which would involve not only compiling the code, but also running it. The response payloads, as well as any errors, are ignored.

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

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

Note that you can’t use URI templates in app.preheat, only explicit URIs. Thus, if you have routed a “/user/{name}/” URI template, you would have to “fill in” the template with a value. In our example, we chose “/user/x/”.

Also note that because preheats are *internal* requests you can use this mechanism to preheat hidden URIs.

If you wish to do more specialized preheating, you may use the [startup task \(page 69\)](#). However, smart use of app.preheat can very effective, and it’s very easy to use.

Implementing Resources

Prudence offers two options for implementing dynamic resources: “manual” resources are “raw,” giving you complete low-level control over the behavior and format of the encapsulated resource, while “scriptlet” resources are simplified and highly optimized for cached textual resources, such as HTML web pages.

Additionally, Prudence offers good support for static resources, just like conventional web servers.

Manual Resources

Mapping vs. Dispatching

TODO

handleInit

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

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

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

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

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

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

handleGet

Handles HTTP "GET" requests.

In a conventional resource-oriented architecture (page 81), clients will not be expecting the resource to be altered in any way by a GET operation.

What you'll usually do here is construct a representation of the resource, possibly according to specific parameters of the request, and return this representation to the client. See the "conversation" API documentation for a complete reference. Note especially that if you've captured URI segments (page 13), they'll be available in conversation.locals (page 45).

The following return types are supported:

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

Beyond the return value, you can affect the response sent to the client by the response attributes in the "conversation" service (page ??). In particular, the conversation.modificationDate (page ??) and conversation.tag (page ??) can be used to affect conditional HTTP requests (page ??). For these, you may also consider implementing the handleGetInfo entry point (page 22) for more scalable handling of conditional requests.

handlePost

Handles HTTP “POST” requests.

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

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

What you’ll usually do here is fetch the data, and alter it according to data sent by the client. See the [“conversation” API documentation \(page ??\)](#) for a complete reference. Note especially that if you’ve [captured URI segments \(page ??\)](#), they’ll be available in [conversation.locals \(page 45\)](#).

See [handleGet \(page 20\)](#) for supported return types. In fact, you may want handlePost to share the same code path as handleGet for creating the representation.

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

handlePut

Handles HTTP “PUT” requests.

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

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

What you’ll usually do here is parse and store the data sent by the client. See the [“conversation” API documentation \(page ??\)](#) for a complete reference. Note especially that if you’ve [captured URI segments \(page ??\)](#), they’ll be available in [conversation.locals \(page 45\)](#).

See [handleGet \(page 20\)](#) for supported return types. In fact, you may want handlePut to share the same code path as handleGet for creating the representation.

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

handleDelete

Handles HTTP “DELETE” requests.

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

What you’ll usually do here is make sure the identified resource exists, and if it does, remove or mark it somehow as deleted. See the [“conversation” API documentation \(page ??\)](#) for a complete reference. Note especially that if you’ve [captured URI segments \(page ??\)](#), they’ll be available in [conversation.locals \(page 45\)](#).

The following return types are supported:

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

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

handleGetInfo

Handles HTTP “GET” requests *before* [handleGet](#) (page 20).

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

The following return types are supported:

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

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

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

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

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

See [“Scaling Tips”](#) (page ??) for a thorough discussion of the problem of scalability.

Controlling the Formats

Negotiated via `handleInit`: the order matters

You can check what was negotiated

But you can set it to whatever you want later

Client-Side Caching

`conversation.modificationDate`, `conversation.tag`

Server-Side Caching

Not supported directly.

Integrating Textual Resources

MVC

Java Resources

Resources

Other Restlets

Scriptlet Resources

Must be mapped. In the future may be dispatched.

Scriptlets

Choice of Programming Language

Templating Languages

Markup Languages

Controlling the Format

Rely on the filename extension (see static resources) or change it in code.

Server-Side Caching

Client-Side Caching

Scriptlet Plugins

On-the-Fly Scriptlet Resources

You can add support for on-the-fly scriptlet resources to your application via the [“execute” route type \(page 9\)](#). This powerful (and dangerous) resource executes all POST payloads as if they were scriptlet resources in the application, and is very useful for [debugging \(page 50\)](#) and maintenance.

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

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

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

Example use with cURL command line:

```
curl -v -d "<% println(1+2) %>" "http://localhost:8080/myapp/execute/"
```

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

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

Where “myscriptfile” would like like a scriptlet resource:

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

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

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

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

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

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

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

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

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

Static Resources

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

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

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

Configuration

Add support for static resources using the “static” route type (page 9) in your `routing.js`, mapping it to a URI template ending in a wildcard:

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

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

```
app.routes = {
  '/*': {
    type: 'static',
    roots: [
      'resources',
      sincerity.container.getLibrariesFile('web')
    ],
    listingAllowed: false,
    negotiate: true,
    compress: true,
    compressThreshold: '1kb',
    compressExclude: []
  }
}
```

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

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



```
    ]
  }
}
```

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

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

MIME Types and Compression

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

Prudence recognizes many common file types by default, but you can add your own mappings in you application’s settings.js, using `app.settings.mediaTypes` (page 43).

When “compress” is *also* true, Prudence will negotiate the best compression format (gzip and zip are supported) and compress on the fly. Only files with a size of at least “compressThreshold” will be compressed. Also, common archive and media formats are excluded from compression, under the assumption that they are already optimally compressed. You can add more exclusions using “compressExclude”.

Client-Side Caching

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

Conditional HTTP is efficient and fast, but you can go one step further and tell clients to avoid even that check. Use the “cacheControl” filter (page 11) before your “static” route type:

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

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

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

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

```

```

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

```

```

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

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

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

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

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

JavaScript and CSS Optimization

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

Prudence’s [“javaScriptUnifyMinify” filter \(page 11\)](#) can help. To configure:

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

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

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

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

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

```

    }
}

```

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

```

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

```

Your HTML files can then include something like this:

```

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

```

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

The “[cssUnifyMinify](#)” filter (page 11) does the same for CSS files, with the default roots being the application's “/resources/style/” subdirectory and the container's “/libraries/web/style/” directory. The relevant files are “all.css” and “all.min.css”. Note, however, that similar functionality is provided by using [ZUSS](#) (page 27).

Here's an example with both filters configured:

```

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

```

Usage in HTML:

```

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

```

ZUSS to CSS

ZUSS is an extended CSS language, inspired by LESS and SASS. It greatly increases the power of CSS by allowing for code re-usability, variables and expressions, as well as nesting CSS. Using the the “[zuss](#)” filter (page 11) you can compile “.zuss” files to CSS, and also apply the same minifier used by “[cssUnifyMinify](#)” (page 11).

To configure:

```

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

```

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

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

```

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

```

The “zuss” filter can be configured with a “roots” param similarly to the [“JavaScriptMinifyUnify”](#) and [“cssUnifyMinify”](#) (page 26).

Other Effects

Because static resources don’t allow for code (unlike manual and scriptlet resources), the way to program your own effects is to add [filters](#) (page 69).

For example, let’s say we want to force all “.pdf” files to be downloadable (by default, web browsers might prefer to display the file using a browser plugin). We’ll be using the technique described [here](#) (page 37) to change the response’s “disposition”.

Here’s our filter code, in “/libraries/filters/download-pdf.js”:

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

To install it in routing.js:

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

Resource Type Comparison Table

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

Web Data

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

Prudence is a minimalist RESTful platform, *not* a data-driven web framework, though such frameworks are built on top of it. Check out our Diligence, which is a full-blown framework based on Prudence and MongoDB. You may also be interested in the [Model-View-Controller \(MVC\) chapter \(page 57\)](#), which guides you through an approach to integrating data backends.

URLs

The simplest way in which a client sends data to the server is via the URL. The main part of the URL is parsed by Prudence and used for [routing \(page 8\)](#), but some of it is left for your own uses.

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

Query Parameters

This is the matrix of parameters after the “?” in the URI.

For example, consider this URL:

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

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

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

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

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

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

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

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

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

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

The Wildcard

If you’ve configured a URI template with a [wildcard \(page 8\)](#) in `routing.js`, you can access the “*” value using `conversation.reference.remainingPart`.

Note that you can also [interpolate the wildcard into the target URI \(page 55\)](#).

Fragments

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

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

But you can also use them in [redirects \(page 32\)](#):

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

Request Payloads

These are used in “POST” and “PUT” verbs.

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

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

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

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

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

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

Note that if the payload comes from a HTML “post” form, better APIs are available (page 33).

MIME Types

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

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

Consumption

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

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

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

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

Parsing Formats

Which formats can Prudence parse, and how well?

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

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

A decent starting point is Restlet’s ecosystem of extensions, which can handle several data formats and conversions. However, these are likely more useful in pure Java Restlet programming, where they can plug into Restlet’s sophisticated annotation-based conversion system. In Prudence, you will usually be applying any generic parsing library to the raw textual or binary data. Still, the Restlet extensions are useful for response payloads (page 35).

Cookies

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

From the Client

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

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

Or use `conversation.cookies` to iterate through all available cookies.

The following attributes are available:

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

To the Client

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

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

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

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

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

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

- `maxAge`: age in seconds, after which the client should delete the cookie; `maxAge=0` deletes the cookie immediately, while `maxAge=-1` (the default) asks the client to keep the cookie only for the duration of the “session” (this is defined by the client; for most web browsers this means that the cookie will be deleted when the browser is closed)
- `secure`: true if the cookie is meant to be used only in secure connections (defaults to false)
- `accessRestricted`: true if the cookie is meant to be used only in authenticated connections (defaults to false)
- `comment`: some clients store this, some discard it

You can ask the client to delete a cookie by calling its “delete” method. Note that this is identical to setting `maxAge=0` and calling “save”.

Security Concerns

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

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

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

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

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

Redirection

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

By Routing

If you need to constantly redirect a specific resource or a URI pattern, you should configure it in your routing.js, using the “redirect” route type (page 10):

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

Note that in this example we interpolated the wildcard (page 55).

By API

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

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

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

In HTML

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

```
Go <a href="<%= conversation.base + '/elsewhere/' %>">elsewhere</a>.
```

Server-Side Redirection

In Prudence, this is called “capturing” (page 14) and has particular use cases. (It can indeed be confusing that this functionality is often grouped together with client-side redirection.)

HTML Forms

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

- “get”: The form fields are all turned into query params and appended to the “action” URL. It is important to remember that an HTTP “GET” is *idempotent* and should not be used to store new data, but rather as a way to represent existing data in a particular way. Actually, “get” forms are not that useful, and are mostly an odd legacy from the early days of the World Wide Web, where “GET” was the only the supported HTTP verb on some platforms. See [query parameters \(page 29\)](#) for handling.
- “post”: The form fields are actually encoded in the same way that query params are, but instead of being affixed to the URL, they are sent as the payload with an “application/x-www-form-urlencoded” MIME type. Though you can of course access this payload [directly \(page 30\)](#), it is recommended to use the specialized APIs detailed here.

Example form:

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

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

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

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

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

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

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

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

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

Accepting Uploads

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

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

Prudence has flexible support for handling uploads: you can configure them to be stored in memory, or to disk. See the [application configuration guide \(page 43\)](#).

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

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

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

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

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

```

Response Payloads

This section is mostly applicable to [manual resources \(page 20\)](#), although it can prove useful to affect the textual payloads of [scriptlet resources \(page 23\)](#).

For [static resources \(page 24\)](#), the response payloads are of course the contents of the resource files.

Textual and Binary Payloads

[Scriptlet resources \(page 23\)](#) might seem to always return textual payloads. Actually, by default they will negotiate a compression format, which if selected will result in a binary: the compressed version of the text. But all of that is handled automatically by Prudence for that highly-optimized use case.

For [manual resources \(page 20\)](#), you can return any arbitrary payload by simply returning a value in `handleGet`, `handlePost` or `handlePut`. Both strings and JVM byte arrays are supported. A textual example:

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

A binary example:

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

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

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

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

```
}
```

If you wish to set both the payload *and* the status code, use an API for either one. Here we'll use the `conversation.status` API family:

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

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

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

Binary Streaming Streaming using [background tasks \(page 69\)](#) is not directly supported by Prudence as of version 2.0. However, this feature is planned for a future version, depending on support being added to Restlet.

Restlet Data Extensions

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

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

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

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

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

Or via the `conversation.setResponseText` API shortcut:

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

Overriding the Negotiated Format

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

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

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

function handleGet(conversation) {
    if (conversation.query.get('format') == 'html') {
        conversation.mediaTypeName = 'text/html'
    }
}
```

```

    }
    return '<html><body>My page</body></html>';
}

```

An example of overriding the negotiated language:

```

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

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

```

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

Avoid Serialization for Internal Requests

Your textual and binary representations might be serialized versions of data structures. But if the request is internal (page 54), then it would be unnecessary and wasteful to go through serialization/deserialization. A simple optimization would be to pass the data “as is,” actually very similarly to a return value for a standard function call.

This works using the special “application/java” MIME type. If selected—whether negotiated for or explicitly overridden (page 35)—then return values are indeed sent “as is.” For example:

```

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

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

```

Note that we’ve added support for “application/java” only to internal requests, verified using the `conversation.internal` API. The reason is that we don’t want to allow “application/java” representations over HTTP. (Actually, that *could* work if the object is itself JVM-serializable, but that’s a more advanced use case we won’t deal with here.)

Under the Hood When the MIME type is “application/java”, Prudence is actually wrapping your return value in an `ObjectRepresentation`. You can also construct it explicitly (page 35):

```
return new org.restlet.representation.ObjectRepresentation(data)
```

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

Browser Downloads

You can create browser-friendly downloadable responses using the `conversation.disposition` API. Here's an example using a [scriptlet resource \(page 23\)](#):

```
Item , Cost , Sold , Profit
Keyboard , $10.00 , $16.00 , $6.00
Monitor , $80.00 , $120.00 , $40.00
Mouse , $5.00 , $7.00 , $2.00
, , Total , $48.00
<%
conversation.mediaTypeName = 'text/csv'
conversation.disposition.type = 'attachment'
conversation.disposition.filename = 'bill.csv'
%>
```

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

External Requests

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

Note that Prudence can also handle [internal REST requests \(page 54\)](#) without going through HTTP or object serialization. There is an entire [internal URI-space \(page 54\)](#) at your fingertips.

It's not a good idea to send an external request while handling a user request, because it could potentially cause a long delay and hold up the user thread. It would be better to use a [background task \(page 69\)](#). A possible exception is requests to servers that you control yourself, and that represent a subsystem of your application. In that case, you should still use [short timeouts \(page 38\)](#) and fail gracefully.

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

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

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

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

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

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

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

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

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

Timeout

Surprisingly, you cannot set the timeout per request, but instead you need to configure the timeout globally for the [HTTP client \(page 73\)](#). The reason for this is that the client works asynchronously, via a thread pool that is initialized upon startup. Because you don't know which thread will handle your request, you can't set a specific timeout for it.

RESTful Files

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

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

The above is simply a shortcut to this:

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

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

Caching

Introduction: Integrated Caching

Server-Side Caching

TODO: write the manual.

The APIs:

`document.cacheDuration`

`document.cacheTags`

`document.cacheKeyPattern`

`document.cacheKey`

Client-Side Caching

See [static resources \(page 25\)](#).

Conditional Requests

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

To support conditional requests, you have to explicitly set at least one of `conversation.modificationDate` (page ??) and `conversation.tag` (page ??). If you implement `handleGetInfo` (page 22), you should be returning one of these values instead.

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

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

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

Two Caching Strategies

The default Prudence application template is configured for very minimal caching, which is suitable for development deployments. However, once you are ready to move your application to production or staging, you will likely want a more robust caching strategy.

We will here present two common strategies, and discuss the pros and cons of each.

Cautious: Short-Term Caching

This is a great strategy if you’re *not* feeling very confident about managing caching in your application logic. Perhaps you have too many different kinds of pages requiring different caching strategies. Perhaps you can’t maintain the strict discipline required for more aggressive caching, due to a quickly changing application structure (“agile”?) or third-party constraints.

If you’re in that boat, short-term caching is recommended over no caching at all, because it would still offer better performance and scalability. Because caching is short-term, any mistakes you make won’t last for very long, and can quickly be fixed.

How short a term depends on two factors: 1) usage patterns for your web site, and 2) the content update frequency.

If a user tends to spend about an hour browsing your site, then a one-hour duration
routing.js:

```
app.routes = {
  '/*': [
    ...
    {
      type: 'scriptlet',
      clientCachingMode: 'conditional' // this is actually the default
    },
    ...
    {
      type: 'cacheControl',
      mediaTypes: {
        'image/png': '1h',
        'image/gif': '1h',
        'image/jpeg': '1h',
        'text/css': '1h',
        'application/x-javascript': '1h'
      },
    },
  ],
}
```

```

        next: {
            type: 'zuss',
            next: 'static'
        }
    }
    ...

```

settings.js:

```

document.require('/sincerity/localization/')

app.globals = {
    ...
    cacheDuration = '1m'.toMilliseconds()
}

```

Selectively Disabling Caching

Confident: Indefinite Caching

```

app.routes = {
    '/*': [
        ...
        {
            type: 'scriptlet',
            clientCachingMode: 'offline',
            maxClientCachingDuration: '5m'.toMilliseconds()
        },
        ...
        {
            type: 'cacheControl',
            mediaTypes: {
                'image/png': 'farFuture',
                'image/gif': 'farFuture',
                'image/jpeg': 'farFuture',
                'text/css': 'farFuture',
                'application/x-javascript': 'farFuture'
            },
            next: {
                type: 'zuss',
                next: 'static'
            }
        },
        ...
    ]
}

```

Paranoid: No Caching

```

app.routes = {
    '/*': [
        ...
        {
            type: 'scriptlet',
            clientCachingMode: 'disabled'
        },
        ...
        {
            type: 'zuss',

```



```

        next: 'static'
    }
    ...

```

Configuring Applications

Prudence application's live in their own subdirectory under “/component/applications/”. The subdirectory name itself can be considered a setting, as it is used as a default identifier for the application in various use cases.

Prudence uses “configuration-by-script” almost everywhere: configuration files are true JavaScript source code, meaning that you can do pretty much anything you need during the bootstrap process, allowing for dynamic configurations that adjust to their working environments.

Prudence, as of version 2.0, does not support live re-configuring of applications. You must restart Prudence in order for changed settings to take hold. The one exception is crontab (page 69): changes there are picked up on-the-fly once per minute.

Overview

The subdirectory contains five main configuration files:

- **settings.js**: This required file, detailed in this chapter, includes settings used by Prudence as well as your own custom settings (page 44).
- **routing.js**: This required file defines the application's URI-space (page 7).
- **crontab**: This optional file defines regularly scheduled background tasks (page 69).
- **default.js**: This required file is used to load the other configuration files above. You should not normally need to edit this file, but feel free to examine it to understand the application bootstrapping process.

settings.js

If you use the default template with the “sincerity prudence create” command, you should get a setting.js file that looks something like this:

```

app.settings = {
  description: {
    name: 'myapp',
    description: 'Skeleton for myapp application',
    author: 'The Author',
    owner: 'The Project'
  },
  errors: {
    debug: true,
    homeUrl: 'http://threecrickets.com/prudence/', // Only used when debug=false
    contactEmail: 'info@threecrickets.com' // Only used when debug=false
  },
  code: {
    libraries: ['libraries'], // Handlers and tasks will be found here
    defrost: true,
    minimumTimeBetweenValidityChecks: '1s',
    defaultDocumentName: 'default',
    defaultExtension: 'js',
    defaultLanguageTag: 'javascript',
    sourceViewable: true
  },
  uploads: {
    root: 'uploads',
    sizeThreshold: '0kb'
  },
}

```

```

        mediaTypes: {
            php: 'text/html'
        }
    }
}

```

Numeric Shortcuts

Time durations are in milliseconds and data sizes in bytes. But these can be specified as either numbers or strings:

- Durations: numerically as milliseconds, or using 'ms', 's', 'm', 'h' or 'd' suffixes for milliseconds, seconds, minutes, hours or days. Fractions can be used, and are rounded to the nearest millisecond, for example: "1.5d".
- Data sizes: numerically as bytes, or using 'b', 'kb', 'mb', 'gb' or 'tb' suffixes. Magnitude uses the binary rather than decimal system: 1kb = 1024b. Fractions can be used, and are rounded to the nearest byte, for example: "1.5mb".

You can accomplish the same trick for your own code using `Sincerity.Localization.toMilliseconds` and `Sincerity.Localization.toBytes`.

app.settings.description

Information here is meant for humans. It appears in the Prudence Administration application.

app.settings.errors

Setting "debug" to true here enables various useful debugging features.

Exception Handling

Uncaught exceptions in your application will automatically set the HTTP response status code to 500 ("internal server error"), but here you can configure the content of the response.

When "debug" is true, Prudence will return a very detailed HTML [debug page \(page 50\)](#). Because this can reveal your application's internal data, **make sure to set "debug" to false for production deployments**.

When "debug" is false, a generic HTML page will be returned instead, using "homeUrl" and "contactEmail" in its template. Note that you can also route your own custom error pages using [app.errors in routing.js \(page 17\)](#).

Caching

When "debug" is true, special HTTP response headers will be added to caching [scriptlet resources \(page 23\)](#):

- **X-Cache**: the cache event, either "hit" or "miss"
- **X-Cache-Expiration**: a timestamp in standard HTTP format
- **X-Cache-Key**: the cache key
- **X-Cache-Tags**: comma-separated list of cache tags

Scriptlet

When "debug" is true, scriptlet resource debugging is enabled: under your container's `/cache/scripturian/` directory you will see the generated source code for each resource.

app.settings.code

Here you can control how Prudence deals with your code:

- **libraries:** An array of paths where importable libraries will be found. If relative, they will be based on the application's root subdirectory.
- **defrost:** When true will attempt to “defrost” manual and scriptlet resources under the “/resources/” subdirectory. Defrosting means pre-parsing and sometimes pre-compiling the code: this allows for faster startup times on first hits to these resources. Note that defrosting is *not* pre-heating: the former only pre-compiles, the latter actually does a “GET” on your resources, which would ensure that services used by your resources are also warmed up. See [app.preheat](#) (page 19).
- **minimumTimeBetweenValidityChecks:** Scripturian makes sure to reload (and thus re-compile) code if the source files are changed, for which it compares the file's modification dates to the cached values. For high-volume deployments, this might involve constantly checking the filesystem, potentially resulting in performance problems on some operating systems. This value allows you to enforce a delay between these checks. It's a good idea to set it to anything greater than zero.
- **defaultDocumentName:** When a document name specifies to a directory, Scripturian will internally change the specification to a document with this name in the subdirectory (excluding the extension). Example, if the value is “default” and you are calling “document.require('/mylibrary/')” and “/libraries/mylibrary/” is a directory, the it would specify “/libraries/mylibrary/default.*”.
- **defaultExtension:** If more than one file in a directory has the same name but different extensions, then this extension will be preferred. For example, if the value is “js” and a directory has both “mylibrary.js” and “mylibrary.py”, then the former file will be preferred.
- **defaultLanguageTag:** If a scriptlet tag does not specify a language, then this value will be the default.

app.settings.uploads

Configure [file upload behavior](#) (page 33) here:

- **root:** This is where uploaded files are stored. If relative, it will be based on the application's root subdirectory.
- **sizeThreshold:** The file upload mechanism can optimize by caching small files in memory instead of saving them to disk. Only if files are greater in size will they be stored. Set to zero to save all files to disk.

app.settings.mediaTypes

This dict maps filename extensions to MIME types.

Prudence recognizes many common file types by default: for example, “png” is mapped to “image/png”. However, using this setting you can define additional mappings or change the default ones. Note that each filename extension can be mapped to one *and only one* MIME type.

This setting is used mostly for scriptlet and [static resources](#) (page 25). For example, a scriptlet resource named “person.s.html” will have the default “text/html” MIME type (which you can change in scriptlet code via the conversation.mediaType APIs), and a static resource named “logo.png” will have the “image/png” MIME type.

For manual resources, you define their supported MIME types manually in [handleInit](#) (page 20). There, you can refer to MIME types directly via the conversation.addMediaTypeByName API, or you can look them up from this setting using the conversation.addMediaTypeByExtension API.

An example:

```
app.settings = {  
    ...  
    mediaTypes: {  
        webm: 'video/webm',  
        msh: 'model/mesh'  
    }  
}
```

Note for PHP: You may notice that the “default” template’s settings.js sets the “text/html” MIME type for the “php” extension. The reason for this is that “.php” files you put in resources are usually expected to output HTML. You may change if you require a different behavior.

app.globals

Use this for custom settings for your application: values here will become application.globals (page 44) when your application is running. Note that Prudence also supports localized settings via inversion of control (page 72).

This dict is “flattened” using dot separators. For example, the following:

```
app.globals = {
  database: {
    driver: 'mysql',
    table: {
      db: 'myapp',
      name: 'users'
    }
  }
}
```

... would be interpreted as if it were:

```
app.globals = {
  'database.driver': 'mysql',
  'database.table.db': 'myapp',
  'database.table.name': 'users'
}
```

In your code, you would access these values using the application.globals API:

```
var driver = application.globals.get('database.driver')
```

Lazy Initialization

This is an advanced topic.

TODO

Programming

Powered by Scripturian

JavaScript

Other Languages

State

Prudence is designed to allow massive concurrency and scalability while at the same time shielding you from the gorier details. However, when it comes to sharing state between different parts of your code, it’s critical that you understand Prudence’s state services.

application.globals

TODO

See app.globals (page 44).

conversation.locals

These are not “local” in the same way that code scope locals are. The term “local” here should be read as “local to the conversation.” They are “global” in the sense that they can be accessed by any function in your code, but are “local” in the sense that they persist only for the duration of the user request. (Compare with “thread locals” in the JVM, which are also “local” in a specific sense.)

You may ask, then, why you wouldn’t want to just use your language globals, which have similar scope and life. `conversation.locals` have three main uses in Prudence:

1. To easily share conversation-scope state between scriptlets written in different languages.
2. To share state for deferred conversations—see [conversation.defer \(page ??\)](#). In such cases, your language’s globals would not persist across the thread boundaries.
3. They are Prudence’s general mechanism for sending state to your code in a conversation. For example, [captured URI segments are stored here \(page ??\)](#) as well as [document.cacheKeyPattern \(page ??\)](#) variables.

Global Variables

You know how local variables work in your programming language: they exist only for the duration of a function call, after which their state is discarded. If you want state to persist beyond the function call, you use a global variable (or a “static” local, which is really a global).

But in Prudence, you cannot expect global variables to persist beyond a user request. To put it another way, you should consider every single user request as a separate “program” with its own global state. See the “life” sections for [generating HTML \(page ??\)](#) and [resources \(page ??\)](#) for more information on when this global state is created and discarded. If you need global variables to persist, you must use [application.globals \(page 44\)](#), [application.sharedGlobals \(page ??\)](#) or even [application.distributedGlobals \(page ??\)](#).

Why does Prudence discard your language’s globals? This has to do with allowing for concurrency while shielding you from the complexity of having to guarantee the thread-safety of your code. By making each user request a separate “program,” you don’t have to worry about overlapping shared state, coordinating thread access, etc., for every use of a variable.

The exception to this is code in `/resources/`, in which language globals *might* persist. To improve performance, Prudence caches the global context for these in memory, with the side effect that your language globals persist beyond a single user request. For various reasons, however, Prudence may reset this global context. You should not rely on this side effect, and instead always use [application.globals \(page 44\)](#).

application.globals vs. application.sharedGlobals

The rule of thumb is to always prefer to use [application.globals \(page 44\)](#). By doing so, you’ll minimize interdependencies between applications, and help make each application deployable on its own.

Use for [application.sharedGlobals \(page ??\)](#) (and possibly [application.distributedGlobals \(page ??\)](#))—similar concerns apply to it) only when you explicitly need a bridge *between* applications. Examples:

1. To save resources. For example, if an application detects that a database connection has already been opened by another application in the Prudence instance, and stored in `application.sharedGlobals`, then it could use that connection rather than create a new one. This would only work, of course, if a few applications share the same database, which is common in many deployments.
2. To send messages between applications. This would be necessary if operations in one application could affect another. For example, you could place a task queue in `application.sharedGlobals`, where applications could queue required operations. A thread in another application would consume these and act accordingly. Of course, you will have to plan for asynchronous behavior, and especially allow for failure. What happens if the consumer application is down? It may make more sense in these cases to use a persistent storage, such as a database, for the queue.

Generally, if you find yourself having to rely on `application.sharedGlobals`, ask yourself if your code would be better off encapsulated as a single application. Remember that Prudence has powerful URL routing, support for virtual hosting, etc., letting you easily have one application work in several sites simultaneously.

Note for Clojure flavor: *All* Clojure vars are VM-wide globals equivalent in scope to `executable.globals`. You usually work with namespaces that Prudence creates on the fly, so they do not persist beyond the execution. However, if you explicitly define a namespace, then you can use it as a place for shared state. It will then be up to you to make sure that your namespace doesn't collide with that of another application installed in the Prudence instance. Though this approach might seem to break our rule of thumb here, of preferring `application.globals` to `application.sharedGlobals`, it is more idiomatic to Clojure and Lisps generally.

`application.sharedGlobals` vs. `executable.globals`

`executable.globals` (page ??) are in practice identical to `application.sharedGlobals` (page ??). The latter is simply reserved for Prudence applications. If you are running non-Prudence Scripturian code on the same VM, and need to share state with Prudence, then `executable.globals` are available for you.

Concurrency

Though `application.globals` (page ??), `application.sharedGlobals` (page ??), `application.distributedGlobals` (page ??) and `executable.globals` (page ??) are all thread safe, it's important to understand how to use them properly.

Note for Clojure flavor: Though Clojure goes a long way towards simplifying concurrent programming, it does not solve the problem of concurrent access to global state. You still need to read this section!

For example, this code (Python flavor) is broken:

```
def get_connection():
    data_source = application.globals['myapp.data.source']
    if data_source is None:
        data_source = data_source_factory.create()
        application.globals['myapp.data.source'] = data_source
    return data_source.get_connection()
```

The problem is that in the short interval between comparing the value in the “if” statement and setting the global value in the “then” statement, another thread may have already set the value. Thus, the “`data_source`” instance you are referring to in the current thread would be different from the “`myapp.data.source`” global used by other threads. The value is not truly shared! In some cases, this would only result in a few extra, unnecessary resources being created. But in some cases, when you rely on the uniqueness of the global, this can lead to subtle bugs.

This may seem like a very rare occurrence to you: another thread would have to set the value *exactly* between our comparison and our set. If your application has many concurrent users, and your machine has many CPU cores, it can actually happen quite frequently. And, even if rare, your application has a chance of breaking if *just two users use it at the same time*. This is not a problem you can gloss over, even for simple applications.

Use this code instead:

```
def get_connection():
    data_source = application.globals['myapp.data.source']
    if data_source is None:
        data_source = data_source_factory.create()
        data_source = application.getGlobal('myapp.data.source',
                                           data_source)
    return data_source.get_connection()
```

The `getGlobal` call is an atomic compare-and-set operation. It guarantees that the returned value is the unique one.

Optimizing for Performance You may have noticed, in the code above, that if another thread had already set the global value, then our created data source would be discarded. If data source creation is heavy and slow, then this could affect our performance. The only way to guarantee that this would not happen would be to make the entire operation atomic, by synchronizing it with a lock:

Here's an example:

```

def get_connection():
    lock = application.getGlobal('myapp.data.source.lock', RLock())
    lock.acquire()
    try:
        data_source = application.globals['myapp.data.source']
        if data_source is None:
            data_source = data_source_factory.create()
            application.globals['myapp.data.source'] = data_source
        return data_source.get_connection()
    finally:
        lock.release()

```

Note that we have to store our RLock as a unique global, too.

Not only is the code above complicated, but synchronization has its own performance penalties, which *might* make this apparent optimization actually perform worse. It's definitely not a good idea to blindly apply this technique: attempt it only if you are experiencing a problem with resource use or performance, and then make sure that you're not making things worse with synchronization.

Here's a final version of our `get_connection` function that lets you control whether to lock access. This can help you more easily compare which technique works better for your application:

```

def get_connection(lock_access=False):
    if lock_access:
        lock = application.getGlobal('myapp.data.source.lock', RLock())
        lock.acquire()

    try:
        data_source = application.globals['myapp.data.source']
        if data_source is None:
            data_source = data_source_factory.create()
            if lock_access:
                application.globals['myapp.data.source'] =
                    data_source
            else:
                data_source = application.getGlobal('myapp.data.
                    source', data_source)
        return data_source.get_connection()
    finally:
        if lock_access:
            lock.release()

```

Complicated, isn't it? Unfortunately, complicated code and fine-tuning is the price you must pay in order to support concurrent access, which is the key to Prudence's scalability.

But, don't be discouraged. The standard protocol for using Prudence's globals will likely be good enough for the vast majority of your state-sharing needs.

APIs

Prudence provides you with an especially rich set of APIs.

The core APIs required for using Prudence are multilingual, in that they are implemented via standard JVM classes that can be called from all supported programming languages: JavaScript, Python, Ruby, PHP, Lua, Groovy and Clojure. Indeed, the entire JVM standard APIs can be access in this manner, in addition to any JVM library installed in your container (under `"/libraries/jars/"`).

Most of these languages additionally have a rich standard API of their own which you can use, as well as an ecology of libraries. JavaScript, however, stands out for having a very meager standard API. To fill in this gap, Sincerity comes with a useful set of JavaScript Libraries, which you are free to use. Some of these are written pure JavaScript, offering new and useful functionality, while others provide JavaScript-friendly wrappers over standard JVM libraries.

Furthermore, Prudence comes with JavaScript-friendly wrappers over the core Prudence APIs. Future versions of Prudence may provide similar friendly wrappers for the other supported languages (please contribute!). Until then, there's nothing that these wrappers can do that you can't do with the core APIs.

Using the Documentation

The Prudence team has spent a great amount of time on meticulously documenting the APIs. Please send us a bug report if you find a mistake, or think that the documentation can use some clarification!

For the sake of coherence all these APIs are documented together online in their JavaScript format. This includes both the multilingual as well as the JavaScript-specific APIs. For the multilingual APIs, just make sure to call the APIs using the appropriate syntax for the programming language you are using. For example, here is the same API call in all supported languages:

```
JavaScript: conversation.redirectSeeOther('http://newsite.org/')
Python:    conversation.redirectSeeOther('http://newsite.org/')
Ruby:      $conversation.redirect__see__other 'http://newsite.org/'
PHP:       $conversation->redirectSeeOther('http://newsite.org/');
Lua:       conversation:redirectSeeOther('http://newsite.org/')
Groovy:    conversation.redirectSeeOther('http://newsite.org/')
Clojure:   (... conversation redirectSeeOther "http://newsite.org/")
```

The APIs are not fully documented here, but rather summarized to give you a global view of what's available, with links to the full documentation available online. The documentation also lets you view the complete JavaScript source code.

You may be further interested in Prudence's low-level API, which is also fully documented online. As a final resort, sometimes the best documentation is the source code itself.

A few more language-specific notes:

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

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

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

Lua You will need to use the get- and set- notation to access attributes. For example, you must use "conversation:getEntity()" to access conversation.entity.

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

Prudence APIs

These core APIs are implemented by the JVM and can be used by any support programming language. The APIs consist of three namespaces that are defined as global variables.

application

document

conversation

Scripturian API

executable

JavaScript Libraries

The APIs are only available for JavaScript running within Scripturian.

Sincerity JavaScript Library

- /sincerity/calendar/: Sincerity.Calendar
- /sincerity/classes/: Sincerity.Classes
- /sincerity/cryptography/: Sincerity.Cryptography
- /sincerity/files/: Sincerity.Files
- /sincerity/iterators/: Sincerity.Iterators
- /sincerity/json/: Sincerity.JSON
- /sincerity/jvm/: Sincerity.JVM
- /sincerity/localization/: Sincerity.Localization
- /sincerity/lucene/: Sincerity.Lucene
- /sincerity/mail/: Sincerity.Mail
- /sincerity/objects/: Sincerity.Objects
- /sincerity/rhino/: Sincerity.Rhino
- /sincerity/templates/: Sincerity.Templates
- /sincerity/xml/: Sincerity.XML

Prudence JavaScript Library

- /prudence/blocks/: Prudence.Blocks
- /prudence/lazy/: Prudence.Lazy
- /prudence/logging/: Prudence.Logging
- /prudence/resources/: Prudence.Resources
- /prudence/tasks/: Prudence.Tasks

Libraries for Bootstrap and Configuration

- /sincerity/annotations/: Sincerity.Annotations
- /sincerity/container/: Sincerity.Container
- /prudence/routing/: Sincerity.Routing
- /prudence/lazy/: Sincerity.Lazy

Diligence

Execution Environments

Bootstrap

Straightforward beginning-to-end script
Except for initialization tasks

Manual Resources and Handlers

Scriptlet Resources

Execute Resource

Cron Tasks

Two options!

Debugging

TODO

Logging

`application.logger`

`application.getSubLogger`

Configuring Logging

`/configuration/logging/`

See Sincerity Manual

Debug Page

Live Execution

See the [on-the-fly scriptlet resources \(page 23\)](#).

FAQ

Please also refer to the FAQs for Sincerity and Scripturian.

Technology

How is REST different from RPC?

Well, it's not *necessarily* better. One important advantage is that it works inside the already-existing, already-deployed infrastructure of the World Wide Web, meaning that you can immediately benefit from a wide range of optimized functionality. We discuss this in greater detail in [The Case for REST \(page 81\)](#).

But we don't advocate REST for every project. See [this discussion \(page 86\)](#) for one reason why RPC may be more appropriate. Choose the right tool for the job!

How is Prudence different from Node.js?

Both are server-side platforms for creating network servers using JavaScript. Both have evolved to support package managers: Sincerity for Prudence, npm for Node.js.

But that's pretty much where the similarities end.

Purpose and Architecture Prudence is a platform for REST services, such as web pages and RESTful APIs. Node.js is a platform for asynchronous services, such as streaming video and audio servers. These are *very* different use cases.

First, note that *both* use non-blocking servers at the low level. So they're *both* asynchronous in *that* particular respect.

Actually, because Prudence uses Jetty as its driver, you have the option to change it to a *blocking* server, which may behave better in some controlled workloads. See [configuring servers \(page 74\)](#). Generally, Jetty is designed from the ground-up for web services, and behaves smartly and exceptionally well under high volumes.

On top of the server Prudence uses a multi-threaded RESTful application environment with Restlet handling the many intricacies of HTTP. Why multi-threaded? Because generating HTML is logically a *single-event* procedure: at the moment you get the client's request, you generate the HTML content and send it immediately. Thus, via a managed, configurable thread-pool, Prudence can leverage multi-core CPUs as well as highly-concurrent database backends to serve several several user requests simultaneously.

Node.js could not be more different: it is by design *single*-threaded and *event-driven*: requests are *never* handed simultaneously, and only a single CPU core would ever be used by the HTTP server itself.

Seems odd? Actually, this “raw” architecture makes a lot of sense for streaming applications: as opposed to HTML pages, streams are always *multi*-event procedures, each event generating a “chunk” of the stream that saturates the socket with data. There is no advantage to using more than one thread if a single thread is already taking up all the bandwidth. In fact, thread synchronization could introduce overhead that would slow the server down. Really, your only variable in terms of scalability is the size of the chunks: you'll want them smaller under high load in order to degrade performance fairly among clients. That said, other libraries you might use from Node.js *can and do* use threads: this is useful for CPU-bound workloads like CPU-intensive video encoding.

Node.js is great for its intended use case. It's vastly easier to write event handlers in JavaScript than in C/C++, and JavaScript also makes it easy to bootstrap the server.

If you like Node.js but JavaScript is not your favorite dynamic language, similarly excellent asynchronous platforms are available: check out Tornado and Twisted for Python, and EventMachine for Ruby.

You're Doing It Wrong It's very odd that Node.js has become a popular platform for the *non*-streaming, data-driven web: the Express framework, for example, provides some minimal RESTful functionality on top of Node.js. But event-driven servers are not designed for REST, and are in fact quite a bad fit: consider that in a single-threaded runtime, if a single event handler hangs, the whole server will hang. Node.js deals with the problem by in effect offloading it to external libraries written in C++: its database drivers, for example, handle queries in their own connection thread pools, and push events to Node.js when data is available. But in your JavaScript event handlers, you have to take extra care not to do any time-consuming work: a delay you cause would affect *all* operations waiting their turn on the single-threaded event loop. Thus, a scalable Node.js application must have event handlers with no risky “side effects,” while external services must be called upon in C++.

And even if you did a good job in JavaScript, your implementation will not in *any way* be more scalable for this use case than by using a thread pool: users still need to wait for their requests to complete, and databases still have to return results before you can complete those requests. There's nothing in an event-driven model that changes these essential facts. (It's worth repeating: both Prudence and Node.js use non-blocking I/O servers: Node.js being event-driven has nothing to do with that.)

You can add some parallelism (and make use of more CPU cores) by running multiple Node.js processes behind a load balancer, but the problems quickly multiply: each process loads its own version of the database driver, with its own connection thread pool, which can't be shared with the other Node.js processes. In Prudence, by contrast, the same pool is trivially shared by all requests.

It should be clear that Node.js is simply the wrong tool for the job. So, why is Node.js so misused? One reason is that its raw architecture is attractively simple: multi-threaded programming is hard to get right, single-threaded easy. JavaScript, too, is attractive as a language that many programmers already know. So, despite being a problematic web platform, it's one in which you can build web services quickly and with little fuss, and sometimes that's more important than scalability or even robustness, especially if the goal is to create in-house services. But another reason for Node.js' popularity is more worrying: ignorance. People who should know better heard that Node.js is “fast” because it's “asynchronous” and think that would lead to faster page load-times for web browsers and the ability to handle more page hits. That's a very wrong conclusion. You can do great REST in Node.js, but would have to work against the platform's limitations for the scenario.

We believe that Prudence is a much more sensible choice for the RESTful web. Beyond the basic architecture, also consider Prudence's features aimed specifically at scalability, such as [integrated caching](#) (page 38), full control of [conditional HTTP](#) (page 22), and [clusters](#) (page 77) for scaling horizontally in the “cloud.” Check out our [Scaling Tips](#) article, too, which is useful even if you don't choose Prudence.

Technology and Ecosystem Prudence was designed specifically for the JVM, to provide you with access to its rich and high-quality ecosystem, to leverage its excellent concurrency libraries and monitoring/profiling capabilities, and to be portable and integrative. The JVM platform is very mature and reliable, as are many of the libraries that Prudence uses, such as Jetty, Restlet and Hazelcast. It's easier to connect C/C++ libraries to Node.js, but on the other hand it's easier to write and deploy extensions in Java/Scala/Groovy for Prudence.

And Prudence is not just JavaScript: it supports many dynamic languages running on top of the JVM—Python, Ruby, PHP, Lua, Groovy and Clojure—as well as their respective ecosystems. That said, it does give special love to JavaScript: Sincerity comes with a rich foundation library, offering essentials such as OOP and string interpolation, as well as friendly wrappers for powerful JVM services, such as concurrent collections and cryptography.

Node.js is JavaScript-centric, and though it has a much younger and narrower ecosystem, it is vibrant and quickly growing.

No Benchmarks for You In terms of sheer computational performance, Node.js has done well to leverage the “browser wars,” which have resulted in very performative JavaScript interpreters and JITs. However, it’s worth remembering that these engines are really optimized for web browser environments, not servers, and have very limited support for threading. Prudence can run JavaScript on your choice of either Nashorn or Rhino, which can both use the JVM’s excellent concurrency management. Nashorn is as of this writing still under development, to be released with JVM 8 in March 2014 (Prudence already supports it). It promises excellent performance, on par with Java code in some cases. Rhino is not as fast, but still performs well and is very mature.

But a comparative benchmark would make little sense. Node.js’ single-threaded model really *needs* blazing-fast language performance, as it directly affects its scalability. Prudence’s multi-threaded model and RESTful expectations mean that it’s rarely CPU-bound: for example, you spend orders of magnitude more time waiting for database backends to respond than for functions to be called. For the web page use case, smart architecture—and smart caching—are *far* more important for scalability than language engine performance.

Note, too, that all the performance-critical parts of Prudence are written in Java, just as they are written in C++ for Node.js.

Summary Choose the right tool for the job! Node.js is a great choice for streaming and streaming-like services, and Prudence—we hope you’ll agree—is a great choice for web services and RESTful APIs.

Performance and Scalability

How well does Prudence perform? How well does it scale?

First, recognize that there are two common uses for the term “scale.” REST is often referred to as an inherently scalable architecture, but that has more to do with project management than technical effectiveness. This difference is addressed in the [“The Case for REST”](#) (page 81).

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

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

2. Generating HTML Prudence implements what might be the most sophisticated [caching system](#) (page 38) of any web development framework. Caching is truly the key to scalable software. See [“Scaling Tips”](#) (page 89) for a comprehensive discussion of the role of caching.

3. Running code There may be a delay when starting up a specific language engine in Prudence for the first time in an application, as it loads and initializes itself. Then, there may be a delay when accessing a dynamic web page or resource for the first time, or after it has been changed, as it might require compilation. Once it’s up and running, though, your code performs and scale very well—as well as you’ve written it. You need to understand concurrency and make sure you make good choices to handle coordination between threads accessing the same data. If all is good, your code will actually perform better throughout the life of the application. The JVM learns and adapts as it runs, and performance can improve the more the application is used.

All language engines supported of Prudence are generally very fast. In some cases, the JVM language implementations are faster than their “native” equivalents. This is demonstrable for Python, Ruby and PHP. The reason is that the JVM, soon reaching version 8, is a very mature virtual machine, and incorporates decades-worth of optimizations for live production environments.

If you are performing CPU-intensive or time-sensitive tasks, then it’s best to profile these code segments precisely. Exact performance characteristics depend on the language and engine used. The Bechmarks Game can give you some comparisons of different language engines running high-computation programs. In any case, if you have a

piece of intensive code that really needs to perform well, it's probably best to write it in Java and access it from the your language. You can even write it in C or assembly, and have it linked to Java via JNI.

If you're not doing intensive computation, then don't worry too much about your language being "slow." It's been shown that for the vast majority of web applications, the performance of the web programming language is rarely the bottleneck. The deciding factors are the usually performance of the backend data-driving technologies and architectures.

I heard REST is very scalable. Is this true? Does this mean Prudence can support many millions of users?

Yes, if you know what you're doing. See ["The Case for REST" \(page 81\)](#) and ["Scaling Tips" \(page 89\)](#) for in-depth discussions.

The bottom line is that it's very easy to make your application scale poorly, whatever technology or architecture you use, and that Prudence, in embracing REST and the JVM, can more easily allow for best-practice scalable architectures than most other web platforms.

That's not very reassuring, but it's a fact of software and hardware architecture right now. Achieving massive scale is challenging.

Errors

How to avoid the "Adapter not available for language: xml" parsing exception for XML files?

The problem is that the XML header confuses Scripturian, Prudence's language parser, which considers the "<?" a possible scriptlet delimiter:

```
<?xml version='1.0' encoding='UTF-8'?>
```

The simple solution is to force Scripturian to use the "<%" for the page via an empty scriptlet, ignoring all "<?":

```
<% %><?xml version='1.0' encoding='UTF-8'?>
```

Licensing

The author is not a lawyer. This is not legal advice, but a personal, and possibly wrong interpretation. The wording of the license itself supersedes anything written here.

Does the LGPL mean I can't use Prudence unless my product is open sourced?

The GPL family of licenses restrict your ability to *redistribute* software, not to use it. You are free to use Prudence as you please within your organization, even if you're using it to serve public web sites—though with no warranty nor an implicit guarantee of support from the copyright holder, Three Crickets LLC.

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

And note that Prudence uses the *Lesser* GPL, which has even fewer restrictions on redistribution than the regular GPL. Essentially, as long as you do not alter Prudence in any way, you can include Prudence in any product, *even if it is not free*. With one exception: Prudence uses version 3 of the Lesser GPL, which requires your product to not restrict users' ownership of data via schemes such as DRM if Prudence is to be included in its distribution.

Even if your product does not qualify for including Prudence in it, you always have the option of distributing your product without Prudence, and instructing your customers to download and install Prudence on their own.

We understand that in some cases open sourcing your product is impossible, and passing the burden to the users is cumbersome. As a last resort, we offer you a commercial license as an alternative to the GPL. Please contact Three Crickets for details.

Three Crickets, the original developers of Prudence, are not trying to force you to purchase it. That is not our business model, and we furthermore find such trickery bad for building trusting relationships. Instead, we hope to encourage you to 1) pay Three Crickets for consultation, support and development services for Prudence, and to 2) consider releasing your own product as free software, thereby truly sharing your innovation with all of society.

Why the LGPL and not the GPL?

The Lesser GPL used to be called the “Library GPL,” and was originally drafted for glibc. It is meant for special cases in which the full GPL could limit the adoption of a product, which would be self-defeating. The assumption is that there are many alternatives with less restrictions on distribution.

In the case of the Linux project, the full GPL has done a wonderful job at convincing vendors to open source their code in order to ship their products with Linux inside. However, it doesn’t seem likely that they would do the same for Prudence. There are so many great web development platforms out there with fewer restrictions.

Note that the LGPL version 3 has a clause allowing you to “upgrade” Prudence to the full GPL for inclusion in your GPL-ed product. This is a terrific feature, and another reason to love this excellent license.

Part II

Advanced Manual

The Internal URI-space

Who says that you need HTTP, or any kind of networking, for REST? The principles themselves are applicable and suitable to in-memory communication, and represent an attractive architectural paradigm for APIs. Attractive especially because it can work *both* locally and over the wire.

The RIAP Pseudo-Protocol

In Prudence, internal REST is elegantly supported by specifying the “RIAP” pseudo-protocol in URIs. RIAP stands for “Restlet Internal Access Protocol”. There are two common formats for RIAP URIs:

- “riap://application/*”: This will route to the *current* application. The remainder will exactly match the relative URIs you’ve mapped in [app.routes](#) (page 8).
- “riap://component/{internal host}/*”:

Internal Requests

See also [external requests](#) (page 37).

Of course, RIAP URIs cannot be used by clients, only by requests initiated in your code.

If you’re using JavaScript, you

Prudence.Resources.request

Configuring the Internal URI-space

The “internal” host (page 17)

Implementing Internal Resources

document.internal

Optimizing using application/java MIME type

Avoid serialization (page 36).

String Interpolation

Template variables, delimited by curly brackets, can be used to interpolate strings for three use cases:

- Captured URI targets, via the [“capture” route type](#) (page 10)
- Redirection URI targets, via the [“redirect” route type](#) (page 10)
- [Cache keys](#) (page 38)

Prudence supports many built-in interpolation variables, extracted from the conversation attributes and summarized below. See also the related Restlet API documentation.

Request URIs

These variables are composed of a prefix and a suffix. The prefix specifies which URI you are referring to, while the suffix specifies the part of that URI. For example, the prefix “{r-}” can be combined with the suffix “{-i}” for “{ri}”, to specify the complete request URI.

Prefixes

- {r-}: actual URI (rference)
- {h-}: virtual host URI
- {o-}: the application’s root URI on the current virtual host
- {f-}: the referred URI (sent by some clients: usually means that the client clicked a hyperlink or was redirected here from elsewhere)

Suffixes

- {-i}: the complete URI (identifier)
- {-h}: the host identifier (protocol + authority)
- {-a}: the authority (for URLs, this is the host or IP address)
- {-p}: the path (everything after the authority)
- {-r}: the remaining part of the path after the base URI (see below)
- {-e}: a relative path from the URI to the application’s base URI (see below; note that this is a constructed value, not merely a string extracted from the URI)
- {-q}: the query (everything after the “?”)
- {-f}: the fragment (the tag after the “#”; note that web browsers handle fragments internally and *never* send them to the server, however fragments may exist in URIs sent *from* the server: see the “{R-}” variable mentioned below)

Base URIs

Every URI also has a “base” version of it: in the case of wildcard URI templates, it is the URI before the wildcard begins. Otherwise it is usually the application’s root URI on the virtual host. It is used in the “{-r}” and “{-e}” suffixes above.

To refer to the base URI directly, use the special “{-b-}” infix, to which you would still need to add one of the above suffixes. For example, “{rbi}” refers to the complete base URI of the actual URI.

Interpolating the Wildcard

This is a common use case for redirection, worth emphasizing. According to the rules above, “*” would be the “{rr}” variable. For example:

```
app.routes = {  
    ...  
    '/assets/*': '/files/media/{rr}'  
}
```

The above would capture a URI such as “/assets/images/logo.png” to “/files/media/images/logo.png”.

Request Attributes

- {p}: the protocol (“http,” “https,” “ftp,” etc.)
- {m}: the method (in HTTP, it would be “GET,” “POST,” “PUT,” “DELETE,” etc.)
- {d}: date (as a Unix timestamp)

Client Attributes

- {cia}: client iP address
- {ciua}: client upstream IP address (if the request reached us through an upstream load balancer)
- {cig}: client agent name (for example, and identifier for the browser)

Payload Attributes

All these refer to the payload (“entity”) sent by the client.

- {es}: entity size (in bytes)
- {emt}: entity media (MIME) type
- {ecs}: entity character set
- {el}: entity language
- {ee}: entity encoding
- {et}: entity tag (HTTP ETag)
- {eed}: entity expiration date
- {emd}: entity modification date

Response Attributes

[these are not supported in capturing or other forms of server-side redirection, because redirection happens before a response is actually generated.] [useful for cache patterns?]

These are all in uppercase to differentiate them from the request variables:

- {S}: the HTTP status code
- {SIA}: server IP address
- {SIP}: server port number
- {SIG}: server agent name
- {R-}: the redirection URI (see “Request URIs” above for a list of suffixes, which must also be in uppercase)

Additionally, all the entity attributes can be used in uppercase to correspond to the response entity. For example, “{ES}” for the response entity size, “{EMT}” for the response media type, etc.

conversation.locals

As we’ve seen in the [app.routes guide \(page 8\)](#), URI templates delimited by curly brackets can be used to parse incoming request URIs and extract the values into [conversation.locals \(page 45\)](#). For example, a “/person/{id}/” URI template will match the “/person/linus/” URI and extract “linus” into the “id” conversation.local.

But you can also do the opposite: interpolate the values that were extracted from the matched URI pattern. An example of redirection that both extracts and interpolates:

```
app.routes = {  
  ...  
  "/person/{id}"/": ">http://newsite.org/profile/?id={id}"  
}
```


Model-View-Controller (MVC)

Background

The model-view-controller (MVC) family of architectural patterns has had great influence over user-interface programming and even design. At its core is the idea that the “model” (the data) and the “view” (the user interface) should be decoupled and isolated. This essentially good idea allows each layer to be optimized and tested on its own. It also allows the secondary benefit of easier refactoring in the future, in case one of the layers needs to be replaced with a different technology, a not uncommon requirement.

Forms, Forms, Forms The problem is that you need an intermediary; and it could be a big problem. Consider that “classic” MVC, based around a thick controller layer, isn’t as popular as it used to be. There, the controller does *as much as possible*, under the assumption of predictability, in order to automate the production of large, repetitive user interfaces. It thus handles form validation, binding of form fields to database columns, and even form flows.

Forms, forms, as far as the eye can see. MVC was, and still is, the domain of “enterprise” user interfaces. It’s telling that the manipulation of the data model is called “business logic”: the use case for MVC really is big business for big businesses. At its best, MVC makes these repetitive stacks of forms easier to maintain in the long run. At its worst, programmers drown in an ocean of controller configuration files, fighting opaque layers of APIs that can only do what they were programmed to do, but not what is needed for a sensible UI.

Outside of the big business world, where automation reigns supreme, UI implementations use more flexible derivatives of MVC, such as Model-View-Presenter (MVP). The “presenter,” far from an opaque layer, is implemented directly by the programmer in code, often by inheriting classes that provide the basic predictable functionality. Depending on the variation you’re using, the “business logic” might even be in the presenter rather than the model. Still, MVP, like MVC, stems from the same anxiety about mixing model and view.

MVC and the Web These two do not seem an obvious match. The web is RESTful, such that the user interface (the “view”) is no different from data (the “model”): both are RESTful resources, implemented similarly. In other words, in REST *the model is the view*.

Well, that’s only really and entirely true for the “classic” web. Using JavaScript and other in-browser plugins, we get a “rich” web that acts no differently from desktop applications. The backend remains RESTful, essentially the “model,” with controllers/presenters as well as views implemented entirely client-side. You can do full-blown, conventional MVC with the “rich” web.

MVC, however, has found inroads into the *classic* web: there exist many frameworks that treat web pages as a pure “view,” an approach they go so far as to enforce by allowing you to embed only limited templating code into your HTML. Some of these frameworks even allow you to configure the form flow, which they then use to generate an opaque URI-space for you, and can even sabotage the browser “back” button to enforce that flow. (MVC automation at its finest! Or worst...)

The impetus for these brutally extreme measures is similar to the one with which we started: a desire to decouple the user interface from everything else. HTML is the realm of web designers, not programmers, and mixing the work of both professions into a single file presents project management challenges. However, there’s a productive distance between cleaning up HTML pages and full-blown MVC, which unfortunately not enough frameworks explore. And actually, not everything called “MVC” really is MVC.

So what are we left with in Prudence? As you’ll see, Prudence supports a straightforward MVP architecture while still adhering to RESTful principles. Read on, and consider if it would benefit your project to use it. Our recommendation is to use what works best for you and your development team.

Tutorial

Models

Implement this layer according to your preferences and as appropriate to the database technology you are using. For example, you may want to use object-oriented architecture. The model layer as a whole should live in your “/libraries/” subdirectory. For this example, let’s put it under “/libraries/models/”.

Do you want the “business logic” to live in the model layer? If so, your classes should be of a somewhat higher level of abstraction above the actual data structure. If you prefer the models to more directly represent the data, then you have the option of putting the “business logic” in your presenters instead.

For our example, let's implement a simple in-memory model, as `/libraries/models/user.js`:

```
var Models = Models || {}

/**
 * Retrieve a person model from the database.
 */
Models.getPerson = function(name) {
    var person = new Models.Person()
    person.setUsername(name)
    return person
}

Models.Person = function() {
    this.getUsername = function() {
        return this.username
    }

    this.setUsername = function(username) {
        this.username = username
    }

    this.getComments = function() {
        return this.comments
    }

    this.comments = new Models.Messages()
}

Models.Messages = function() {
    this.get = function() {
        return this.messages
    }

    this.add = function(message) {
        this.messages.append(message)
    }

    this.messages = []
}
```

Views

In Prudence, these are hidden scriptlet resources (page 23). For this example, let's put them under `/libraries/includes/views/`.

If you prefer to use templating languages for your views, Velocity and Succinct are supported (page 23). Your designers may also find it useful to use the supported HTML markup languages (page 23). Even if you prefer templating, you can still “drop down” to dynamic languages, such as JavaScript (server-side), when useful: Prudence allows you to easily mix and match scriptlets in different languages. If you do so, take special note of the nifty in-flow tag.

There are some who shudder at the thought of mixing dynamic languages and HTML. This likely comes from bad experience with poorly-designed PHP/JSP/ASP applications, where everything gets mixed together into the “view” file. If you're afraid of losing control, then you can simply make yourself a rule that only templating languages are allowed in scriptlets. It's purely a matter of project management discipline. We recommend, however, relaxing some of that extremism: for example, you can make the rule that no “business logic” should appear together with HTML, while still allowing some flexibility for

using server-side JavaScript, but *only* for UI-related work. Still unconvinced? We'll show you below how to use your favorite pure templating engine with Prudence (page 60).

The required data will be injected into the view by the presenter as an “object” POST payload, available via the `conversation.entity` API. We'll detail below how that happens. For now, here's our example view, “`/libraries/includes/views/user/comments.html`”:

```
<html>
<%
  var context = conversation.entity.object
  var person = context.person
  var comments = person.getComments().get()
%>
<body>
  <p>These are the comments for user: <%= person.getUsername() %></p>
  <table>
<% for (var c in comments) { %>
    <tr><td><%= comments[c] %></td></tr>
<% } %>
  </table>
  <p>You may add a comment here:</p>
  <form>
    <input name="comment" />
    <input type="submit" />
  </form>
</body>
</html>
```

Presenters

In Prudence, these are the resources that are actually exposed in the URI-space, while the views remain hidden. The presenter retrieves the appropriate view and presents it to the user.

You can use either manual or scriptlet resources as your presenters. However, manual resources (page 20) offer a bit more flexibility, so we will choose them for our example. Our example presenter is in “`/resources/user/comments.m.js`”:

```
document.require(
  '/models/user/',
  '/prudence/resources/',
  '/sincerity/objects/')

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

function handleGet(conversation) {
  var name = conversation.locals.get('name')
  var person = Models.getPerson(name)
  return getView('user/comments', {person: person})
}

function handlePost(conversation) {
  var name = conversation.locals.get('name')
  var comment = conversation.form.get('comment')
  var person = Models.getPerson(name)
  person.getComments().add(comment)
  return getView('user/comments', {person: person})
}
```

```

}

function getView(view, context) {
    var page = Prudence.Resources.request({
        uri: '/views/' + view + '/',
        internal: true,
        method: 'post',
        mediaType: 'text/*',
        payload: {
            type: 'object',
            value: context
        }
    })
    return Sincerity.Objects.exists(page) ? page : 404
}

```

To keep the example succinct, we're only making use of a single view in this presenter, though it should be clear that you can use any appropriate logic here to retrieve any view using `getView`.

`getView` is where the MVC “magic” happens, but as you can see it's really nothing more than an internal request. We're specifically using two special features of internal requests:

- We can request URIs that are hidden: in this case, anything under “/libraries/includes/”.
- We can transfer “POST” payloads directly using the “object” type.

We'll remind you also that internal requests are *fast*. They emphatically do not use HTTP, and “object”-type payloads are *not* serialized.

Here's our `routing.js` entry for the presenter:

```

app.routes = {
    ...
    '/user/{name}/comments/': '/user/comments!'
}

```

Note the use of capture-and-hide (page 13).

Voila. Test your new MVC application by pointing your web browser to “/user/Linus/comments/” under your application's base URL.

Implications for Caching

Caching of scriptlet resources (“views”) works as usual here, though you should take care to remember that the request hitting the scriptlet resource is the *internal* one, *not* the external one, which actually hits the presenter. If there are attributes of the external request that you want to use for the cache key pattern, then you must transfer them manually.

View Templates

One size does not fit all. Almost every web framework comes with its own solution to templating, with its own idiosyncratic syntax and set of features, manifesting its own templating philosophy. As you've probably picked up, Prudence's philosophy is that the programmer knows best: scriptlets should be able to do *anything*, and the programmer doesn't need to be “protected” from bad decisions via a dumbed-down, sandboxed templating domain language.

There are two common counter-arguments, which we don't think are very convincing.

The first is that the people designing the templates might not, in fact, know best: they might not be proficient programmers, but instead web designers who specialize in HTML/CSS coding and testing. They would be able to deal with a few inserted template codes, but not a full-blown programming language. The “real” programmers would be writing the controllers/presenters, and injecting values into the templates according to the web designers' needs. This argument carries less validity than it used to: proficient web designers these days need to know JavaScript, and if they can handle client-side JavaScript, they should be able to handle server-side JavaScript, too. Will they need to learn some new things? Yes, but learning a new templating language is no trivial task, either.

The second counter-argument is about discipline: even competent programmers might be tempted to make “shortcuts,” and insert “business logic” into what should be purely a “view.” This would short-circuit the MVC separation and create hard-to-manage “spaghetti” code. A restricted templating language could, then, enforce this discipline. This seems like a brutal solution: programmers get annoyed if their own platforms don’t trust them, and in any case can circumvent these restrictions by writing plugins that would then do what they need. But the real issue is that discipline should be handled as a social issue of project management, not by tools.

In any case, we won’t force our philosophy on you: Prudence has built in support for two templating engines (page 23), and it’s easy to plug in a wide range of alternative templating engines into Prudence. If you’re familiar and comfortable with a particular one, use it. We’ll guide you in this section.

There are many templating engines you can use. The best performing and most minimal solutions are pure JVM libraries: StringTemplate, Thymeleaf and Snippetory. However, the most popular ones use other languages, for example Jinja2 for Python. Below are examples per each type.

The technique we’ll show for using both types is the same: writing a custom dispatcher (page 18), so make sure you understand dispatching before you continue to read.

StringTemplate Example

For this example, we chose StringTemplate: it’s very minimal, and stringently espouses a philosophy entirely opposite to Prudence’s: proof that Prudence is not forcing you into a paradigm! We’ll of course use the Java/JVM port, though note that StringTemplate is also ported to other languages.

It’s available on Maven Central, so you can install it in your container using Sincerity:

```
sincerity attach maven-central : add org.antlr ST4 : install
```

Otherwise, you can also download the binary Jar from the StringTemplate site and place it in your container’s “/libraries/jars/” directory.

Here’s our application’s “/libraries/dispatchers/st.js”:

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

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

function handlePost(conversation) {
    if (!conversation.internal) {
        return 404
    }
    var id = String(conversation.locals.get('com.threecrickets.prudence.dispatcher.id'))
    if (id.endsWith('/')) {
        id = id.substring(0, id.length - 1)
    }
    var st = getDir().getInstanceOf(id)
    if (!Sincerity.Objects.exists(st)) {
        return 404
    }
    if (Sincerity.Objects.exists(conversation.entity)) {
        var context = conversation.entity.object
        if (Sincerity.Objects.exists(conversation.context)) {
            for (var key in context) {
                var value = context[key]
                if (Sincerity.Objects.isArray(value)) {
                    for (var v in value) {
                        st.add(key, value[v])
                    }
                }
            }
        }
    }
    else {
```

```

                                st.add(key, value)
                                }
                            }
                        }
                    }
                }
            }
        return st.render()
    }

function getDir() {
    var dir = new org.stringtemplate.v4.STRawGroupDir(application.root + '/libraries/view
    dir.delimiterStartChar = '$'
    dir.delimiterStopChar = '$'
    return dir
}

```

The StringTemplate API is very straightforward and this code should be easy to follow. Notes:

- We've allowed only internal requests through: we want to hide this dispatcher from the public URI space because it should only be accessed by our presenters.
- We're stripping trailing slashes from the ID because STRawGroupDir doesn't support them.
- We've switched to the older "\$" delimiters because the default "<" and ">" delimiters are awkward to use with HTML.
- STRawGroupDir does not pick up template file changes on-the-fly, so we're recreating it per request. Because it caches compiled templates, it would be more efficient to make it a global, but then you would have to find an alternative way for invalidating it for live application edits.

Now for our routing.js:

```

app.routes = {
    ...
    '/views/*': '@st:{rr}'
}

app.dispatchers = {
    ...
    st: {
        dispatcher: '/dispatchers/st/'
    }
}

```

See how we've interpolated the wildcard ([page 55](#)) into "{rr}": this means that a URI such as "/views/hello/" would translate to the ID "hello".

Let's create our template, "/libraries/views/user/comments.st":

```

<html>
<body>
    <p>These are the comments for user: $username$</p>
    <table>
        $comments:{ c | <tr><td>$c$</td></tr> }$
    </table>
</body>
</html>

```

Note the use of an anonymous template (a lambda). As an alternative, we can also use named templates, which we can group into reusable libraries. This is easy to do with group files. With that, here's an alternative definition of the above, saved as "/libraries/views/user.stg":

```

comments(username, comments) ::= <<
<html>
<body>
    <p>These are the comments for user: $username$</p>
    <table>
        $comments:row() $
    </table>
</body>
</html>
>>

row(content) ::= <<
<tr><td>$content$</td></tr>
>>

```

Note that STRawGroupDir treats these “.stg” files as if they were a directory with multiple files when you look up an ID, so the URI to access “comments” would be the same in both cases: “/views/users/comments/”.

Finally, our presenters would work the same as in the [MVC tutorial \(page 59\)](#). The only change would be to flatten out the contexts for StringTemplate to use:

```

var person = Models.getPerson(name)
...
return getView('comments', {
    username: person.getUsername(),
    comments: person.getComments().get()
})

```

That’s it!

Caching? Not with StringTemplate, unfortunately. Its brutal rejection of any kind of programming logic in templates means that you can’t create a plugin to support caching. Your only option would be to cache the views yourself in the presenter, or even use an [upstream cache \(page 94\)](#). In our next example, we’ll be using a more flexible engine that allows for more integration with Prudence features.

Jinja2 Example

Jinja2 is an embeddable engine that mimics the well-known template syntax of the Django framework. We’ll go over its basic integration into Prudence, and also show you how to write Jinja2 custom tags to easily take advantage of Prudence’s [caching mechanism \(page 38\)](#).

First we need to install Python and Jinja2 in our container:

```
sincerity add python : install : easy_install Jinja2==2.6 simplejson
```

We’re also installing the simplejson library, because Jython doesn’t come with the JSON support we’ll need (more on that later).

Note that Jinja2 version 2.7 doesn’t work in Jython (might be fixed for version 2.8), but version 2.6 does, so that’s what we use here.

For our dispatcher, we’ll do something a bit different from before: because we want to support caching of templates, we will want the actual template renderer as a [scriptlet resource \(page 23\)](#). The dispatcher, then, will simply delegate to that scriptlet resource. Another change is that we’ll be writing it all in Python, so we can call the Jinja2 API. Let’s start with “/libraries/dispatchers/jinja.py”:

```

import simplejson, urllib
from org.restlet.representation import ObjectRepresentation

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

```

```

def handle_post(conversation):
    if not conversation.internal:
        return 404
    id = conversation.locals['com.threecrickets.prudence.dispatcher.id']
    if id[-1] == '/':
        id = id[0:-1]
    id += '.html'
    context = {}
    if conversation.entity:
        if conversation.entity.mediaType.name == 'application/java':
            context = conversation.entity.object
        else:
            context = conversation.entity.text
            if context:
                context = simplejson.loads(context)
    payload = ObjectRepresentation({
        'context': context,
        'uri': str(conversation.reference),
        'base_uri': str(conversation.reference.baseRef)})
    resource = document.internal('/jinja-template/' + urllib.quote(id, '')) + '/', 'text/html'
    result = resource.post(payload)
    if not result:
        return 404
    return result.text

```

Notes:

- We’ve allowed only internal requests through: we want to hide this dispatcher from the public URI space because it should only be accessed by our presenters.
- Jinja2’s `FileSystemLoader` requires the full filename, so we’re stripping trailing slashes and adding “.html”.
- We’re forwarding a few useful attributes of the request: the original URI and the original base URI. We’ll show you later how to use those for our custom tags.
- We’re supporting “application/java” payloads (page 36), but also JSON payloads. Avoiding serialization is fine for Python-to-Python calls, but if we call from another programming language—say, JavaScript—the native structures are incompatible. Thus, we’re allowing the use of JSON as an interchange format. On the other side, when we send the payload to “/jinja-template/”, it’s fine to send a raw object, because it’s Python-to-Python.

Our scriptlet resource is “/resources/jinja-template.s.html”:

```

<%python
import urllib
from jinja2 import Environment, FileSystemLoader
from jinja2.exceptions import TemplateNotFound
from os import sep

id = urllib.unquote(conversation.locals['id'])
payload = conversation.entity.object
context = payload['context']

loader = application.globals['jinaj2.loader']
if not loader:
    loader = FileSystemLoader(application.root.path + sep + 'libraries' + sep + 'views')
    loader = application.getGlobal('jinja2.loaded', loader)
env = Environment(loader=loader)

try:

```



```

        template = env.get_template(id)
        print template.render(context)
except TemplateNotFound:
    conversation.statusCode = 404
%>

```

The Jinja2 API is very straightforward and this code should be easy to follow. Note that we’re caching the FileSystemLoader as an application.global: because it can pick up our changes on-the-fly and cache them, it’s eminently reusable.

Now for our routing.js:

```

app.routes = {
    ...
    '/views/*': '@jinja:{ rr }',
    '/jinja-template/{id}/*': '/jinja-template/!'
}

app.dispatchers = {
    ...
    jinja: {
        dispatcher: '/dispatchers/jinja/'
    }
}

```

See how we’ve interpolated the wildcard (page 55) into “{rr}”: this means that a URI such as “/views/hello/” would translate to the ID “hello/”.

Let’s create our template, “/libraries/views/user/comments.html”:

```

<html>
<body>
    <p>These are the comments for user: {{username}}</p>
    <table>
{% for comment in comments %}
        <tr><td>{{comment}}</td></tr>
{% endfor %}
    </table>
</body>
</html>

```

Finally, we need to make a small change to our presenter (page 59) in order to send JSON payloads (if we were writing it in Python, we could optimize by sending “object” payloads):

```

function getView(view, context) {
    var page = Prudence.Resources.request({
        uri: '/views/' + view + '/',
        internal: true,
        method: 'post',
        mediaType: 'text/*',
        payload: {
            type: 'json',
            value: context
        }
    })
    return Sincerity.Objects.exists(page) ? page : 404
}

```

That’s it!

Custom Tags It’s fairly easy to add custom tags to Jinja2. Let’s add some to support Prudence caching, as well as other useful Prudence values. Here’s “/libraries/jinja_extensions.py”:

```

from jinja2 import nodes
from jinja2.ext import Extension
from org.restlet.data import Reference

class Prudence(Extension):
    # a set of names that trigger the extension
    tags = set(['current_uri', 'application_uri', 'to_base', 'cache_duration', 'cache_tags'])

    def __init__(self, environment):
        super(Prudence, self).__init__(environment)

        # add the defaults to the environment
        environment.extend(
            prudence_document=None,
            prudence_uri=None,
            prudence_base_uri=None
        )

    def parse(self, parser):
        token = parser.stream.next()
        tag = token.value
        lineno = token.lineno

        if tag == 'current_uri':
            return _literal(self.environment.prudence_uri, lineno)

        elif tag == 'application_uri':
            return _literal(self.environment.prudence_base_uri, lineno)

        elif tag == 'to_base':
            base = Reference(self.environment.prudence_base_uri)
            reference = Reference(base, self.environment.prudence_uri)

            # reverse relative path to the base
            relative = base.getRelativeRef(reference).path

            return _literal(relative, lineno)

        elif tag == 'cache_duration':
            duration = parser.parse_expression().as_const()
            self.environment.prudence_document.cacheDuration = duration

        elif tag == 'cache_tags':
            tags = [parser.parse_expression().as_const()]
            while parser.stream.skip_if('comma'):
                tags.append(parser.parse_expression().as_const())

            cache_tags = self.environment.prudence_document.cacheTags
            for tag in tags:
                cache_tags.add(tag)

            return _literal('', lineno)

    def _print(text, lineno):
        return nodes.Output([nodes.TemplateData(text)]).set_lineno(lineno)

```

We'll then modify our `"/resources/jinja-template.s.html"` to use our extension, and set it up using the attributes forwarded from the dispatcher.:

```
env = Environment(loader=loader, extensions=['jinja_extensions.Prudence'])
env.prudence_document = document
env.prudence_uri = payload['uri']
env.prudence_base_uri = payload['base_uri']
```

Here's a simple template to test the extensions:

```
<html>
{% cache_duration 5000 %}
{% cache_tags 'tag1', 'tag2' %}
<body>
<p>This page is cached for 5 seconds.</p>
<p><b>current_uri</b>: {% current_uri %}</p>
<p><b>application_uri</b>: {% application_uri %}</p>
<p><b>to_base</b>: {% to_base %}</p>
</body>
</html>
```

RESTful Models

In our MVC tutorial above, we've implemented our models as classes (OOP). However, it may make sense to implement them as RESTful resources instead.

Doing so allows for powerful deployment flexibility: it would be possible to decouple the model layer entirely, over HTTP. For example, you could have "model servers" running at one data center, close to the database servers, while "presentation servers" run elsewhere, providing the direct responses to users. In this scenario, the presenters would be calling the models using secured HTTP requests, instead of function calls.

Even if you're not planning for such flexibility at the moment, it might still be a good idea to allow for it in the future. Until then, you could optimize by treating the model layer as an internal API (page 54), which makes it about as fast as function calls.

There are two potential downsides to a RESTful model layer. First, there's the added programming complexity: it's easier to create a class than a resource. Second, RESTful resources are limited to four verbs: though GET/POST/PUT/DELETE might be enough for most CRUD operations, it can prove harder to design a RESTful URI-space for complex "business logic."

A good compromise, if necessary, can be to still use HTTP to access models, just not RESTfully: use Remote Procedure Call (RPC) instead. We discuss this option in the URI-space architecture tips (page 86).

Tutorial

For simplicity, we'll use a mapped resource, `"/resources/models/person.m.js"`:

```
document.require(
    '/models/user/',
    '/sincerity/json')

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

function handleGet(conversation) {
    var name = conversation.locals.get('name')
    var person = Models.getPerson(name)
    var result = {
        username: person.getUsername(),
```

```

        comments: person.getComments().get()
    }
    return conversation.mediaTypeName == 'application/java' ? result : Sincerity.JSON.to(
}

function handlePost(conversation) {
    var name = conversation.locals.get('name')
    var payload = Sincerity.JSON.from(conversation.entity.text)
    if (payload.comment) {
        var person = Models.getPerson(name)
        person.getComments().add(comment)
        var result = {
            username: person.getUsername(),
            comments: person.getComments().get()
        }
        return conversation.mediaTypeName == 'application/java' ? result : Sincerity.
    }
    return 400
}

```

Note that we've optimized for internal requests (page 36).

We would then modify our presenter like so:

```

function handleGet(conversation) {
    var name = conversation.locals.get('name')
    var person = getModel('person/' + encodeURIComponent(name))
    return getView('comments', {person: person})
}

function handlePost(conversation) {
    var name = conversation.locals.get('name')
    var comment = conversation.form.get('comment')
    var person = postModel('person/' + encodeURIComponent(name), {comment: comment})
    return getView('comments', {person: person})
}

function getModel(model) {
    return Prudence.Resources.request({
        uri: '/models/' + model + '/',
        internal: true
    })
}

function postModel(model, payload) {
    return Prudence.Resources.request({
        uri: '/models/' + model + '/',
        internal: true,
        method: 'post',
        payload: {
            type: 'object',
            value: payload
        }
    })
}

```

We've assumed an internal request here, but it's easy to change it to an external request if the model layer runs elsewhere on the network.

Finally, here's our addition to routing.js, using capture-and-hide (page 13):

```
app.routes = {
    ...
    '/models/person/{name} /': '/models/person /!'
}
```

Background Tasks

TODO

crontab

TODO

startup

Task API

TODO

Filtering

In Prudence, “filters” are route types (page 10) used to add effects to resources. Though you can often code the same effect directly into a resource, filters are decoupled from resources, allowing you to reuse the filter, which is especially useful with mapping route types (page 9).

Furthermore, filters are the *only* way to add effects to static resources (page 24), which cannot themselves be programmed.

Because it’s easy to enable and disable filters just by editing routing.js, filters are often used to add debugging and testing effects.

Tutorial

Filters are implemented similarly to manual resources (page 20): as source code files (in any supported language) with either or both filter entry points: `handleBefore` and `handleAfter`. The former is called before all requests reach the next (downstream) route type, and the latter is called on the way back, after the downstream finishes its work.

Let’s start with configuring our filter, using the “filter” route type (page 10) in routing.js. We’ll put it in front of all our main mapping resources:

```
app.routes = {
    ...
    '/*': {
        type: 'filter ',
        library: '/my-filter /',
        next: [
            'manual ',
            'scriptlet ',
            'static '
        ]
    }
}
```

Now let’s create the actual filter in “/libraries/my-filter.js”. We’ll start with a trivial `handleAfter` implementation:

```
function handleAfter(conversation) {
    application.logger.info('We got a request for a resource at: ' + conversation.referen
}
```

Our filter doesn't do much, but it's easy to test that the code is being called by looking at the log. Of course you can do many other things here, as detailed in the examples below. Most of the conversation APIs are available to you, including the redirection APIs.

Note that while you need to restart your application for the changes in routing.js to take hold, you are free to edit my-filter.js and have the changes will be picked up on-the-fly.

handleAfter is a bit more sophisticated in that it also requires a return value:

```
function handleBefore(conversation) {
    application.logger.info('We got a request for a resource at: ' + conversation.reference)
    return 'continue'
}
```

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

- **“continue” or 0**: Continue to the “next” handler
- **“skip” or 1**: Skip the “next” route type and go immediately to our own handleAfter
- **“stop” or 2**: Stop *our* handling altogether: the same as “skip,” but handleAfter is *not* called (note that if a filter was installed *before* us, it would still be called)

Again, you may define both a handleBefore and a handleAfter in the same filter.

Examples

Changing the Request

It may be useful to change user requests for testing purposes. Specifically, we can affect content negotiation by changing the accepted formats declared by the user.

For example, let's say we want to disable compression for all resources, even if users declare that they are capable of handling it:

```
function handleBefore(conversation) {
    conversation.client.acceptedEncodings.clear()
    return 'continue'
}
```

Easy! Note that this would be *much* harder to do by changing the response: we would have to decompress all compressed responses.

Overriding the Response

Filters can be useful for overriding the response under certain conditions.

The following example filter always sets the response to a web page displaying “blocked!”, unless a special “admin” cookie (page 31) is used with a magic value. It can be used to make sure that certain resources are unavailable for users who are not administrators:

```
function handleAfter(conversation) {
    if (!isAuthorized(conversation)) {
        var content = '<html><body>' + conversation.reference + ' is blocked to you!</body></html>'
        conversation.setResponseText(content, 'text/html', 'en', 'UTF-8')
    }
}

function isAuthorized(conversation) {
    var cookie = conversation.getCookie('admin')
    return (null !== cookie) && (cookie.value === 'magic123')
}
```

Another, simpler trick, would be to redirect the response:

```
function handleAfter(conversation) {
    if (!isAuthorized(conversation)) {
        conversation.redirectSeeOther(conversation.base + '/blocked/')
    }
}
```

Note on *changing the response*: You might think that filters could be useful to affect the content of responses, for example to “filter out” inappropriate language in HTML pages. Actually, Prudence filters are *not* a good way to do this, because there’s no guarantee that response payloads returned from resources are textual, even if the content is text: they could very well be compressed (gzip) and also chunked. You would then need to decode, disassemble, make your changes, and then reassemble such responses, which is neither trivial nor efficient. *Content* filtering should best be handled at the level of the resource code itself, *before* the response payload is created.

Side Effects

Filters don’t have to change anything about the request or the response. They can be useful for gathering statistics or other debugging information.

In this example, we’ll gather statistic about agent self-identification: specifically web browser product names and operating systems (via the `conversation.client` API):

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

importClass(
    java.util.concurrent.ConcurrentHashMap,
    java.util.concurrent.atomic.AtomicInteger)

var logger = application.getSubLogger('statistics')

function handleBefore(conversation) {
    var agent = conversation.client.agentName
    var os = conversation.client.agentAttributes.get('osData')

    getCounter('agent', agent).incrementAndGet()
    getCounter('os', os).incrementAndGet()

    logger.info('Agent stats: ' + application.globals.get('counters.agent'))
    logger.info('OS stats: ' + application.globals.get('counters.os'))

    return 'continue'
}

function getCounter(section, name) {
    var counters = application.getGlobal('counters.' + section, new ConcurrentHashMap())
    var counter = counters.get(name)
    if (null == counter) {
        counter = new AtomicInteger()
        var existing = counters.putIfAbsent(name, counter)
        if (null != existing) {
            counter = existing
        }
    }
    return counter
}
```

Here we’re storing statistics in memory and sending them to the log, but for your uses you might prefer to store them in a database using atomic operations.

Built-in Filters

Prudence comes with a few built-in filters, each with its own route type (page 10). Many of them are useful specifically with static resources (page 24), and are discussed in that chapter. A few others are more generally useful, and are discussed here.

Inversion of Control (IoC) via Injection

You already know that you can configure parts of your application via application.global presets (page 44). Globals, of course, affect the entire application. However, you may sometimes need *local* configurations: the ability to a specific instance of a resources differently from others. That's where the "injector" route type (page 10) comes in.

Note that because IoC is most often used together with capturing and dispatching, there is a shortcut notation to apply to the "capture" and "dispatch" route types (page 16). However, you can also use injection independently. An example for routing.js:

```
app.routes = {
  ...
  '/user/{name}': {
    type: 'injector',
    locals: {
      deployment: 'production'
    },
    next: '@user'
  }
}
```

To access the injected value in your resource code, simply use the `conversation.locals` API:

```
var deployment = conversation.locals.get('deployment')
if (deployment == 'production') {
  ...
}
```

You can inject any kind of object using an injector, though keep in mind that the native types of JavaScript may not be easily accessible in other programming languages. For example, if you're injecting a dict or an array, it would not be automatically converted to, say, a Python dict or vector. However, primitive types such as strings and numbers would be OK for all supported languages.

Templates Note that injectors specially recognize Template instances and casts them before injecting. This allows you to interpolate conversation attributes (page 54) into strings:

```
app.routes = {
  ...
  '/user/{name}': {
    type: 'injector',
    locals: {
      protocol: new org.restlet.routing.Template('{p}'),
      deployment: 'production'
    },
    next: '@user'
  }
}
```

(The above example is not that useful: you can just as easily access the protocol using `conversation.request.protocol.name`.)

HTTP Authentication

You can implement simple HTTP authentication using the "httpAuthenticator" route type (page 10):


```

app.routes = {
    ...
    '/*': {
        type: 'httpAuthenticator ',
        realm: 'Authorized users only!',
        credentials: {
            moderator: 'moderatorpassword ',
            admin: 'adminpassword '
        },
        next: [
            'manual',
            'scriptlet '
        ]
    }
}

```

Note that the implementation relies on basic authentication (BA), which is unencrypted. It is thus strongly recommended that you use it only with [HTTPS \(page 74\)](#).

Configuring the Component

TODO

Order: applications, services, starts component, then runs initialization tasks

/component/servers/

See [configuring servers \(page 74\)](#).

/component/clients/

/component/hosts/

See [configuring hosts \(page 75\)](#).

/component/services/

Run *after* the component is configured but *before* it is started.

/component/services/prudence/caching

Configure the caching backend

/component/services/prudence/distributed

Load the Hazelcast configuration

/component/services/prudence/executor

Configures thread pools for task execution.

/component/services/log

Configures the component's log service, which is used for logging client requests. (By default web.log)

/component/services/prudence/singleton

Prudence assumes a single Restlet Component instance. If for some reason you have a more complex setup, you can configure Prudence's initialization here.

/component/services/prudence/scheduler

Configure the cron scheduler (cron4j)

/component/services/prudence/status

Configures Restlet's status service to use Prudence's implementation.

/component/services/prudence/version

Provides access to Prudence and Restlet versions.

/component/templates/

Servers and Hosts

Restlet has excellent virtual host support, and there is a many-to-many relationship routing between almost servers, hosts and application. So, you can easily have a single Prudence container (running in a single JVM instance) managing several sites at once with several applications on several domains on several servers.

Configuring Servers

Define your servers under “/component/servers/”. At the simplest, you just add a new file and assign a port for it. A minimal server definition for HTTP:

```
var server = new Server(Protocol.HTTP, 8080)
server.name = 'myserver'
component.servers.add(server)
```

You can also bind a server to a specific IP address, in case your machine has more than one IP address:

```
server.address = [string]
```

There are also many configuration parameters for Jetty, the HTTP engine, which you can set in the server's context. Here we configure the size of the thread pool:

```
server.context.parameters.set('minThreads', '6')
server.context.parameters.set('maxThreads', '12')
```

Here we switch to a blocking server:

```
server.context.parameters.set('type', '1') // blocking NIO
```

See Jetty's `HttpServerHelper` as well as all inherited parameters from the parent classes.

Secure Servers (HTTPS)

If you are using a load balancer or another kind of proxy (page 79), it may make sense to handle secure connections there. But Prudence can also handle secure (HTTPS) connections itself. Here's an example configuration for “/component/servers/https.js”:

```
var server = new Server(Protocol.HTTPS, 443)
server.name = 'secure'
component.servers.add(server)

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

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

See the Restlet Jetty `HttpServerHelper` documentation for all security configuration parameters.

Security Keys The above configuration assumes the you have a Java KeyStore (JKS) file at “/path/prudence.jks” containing your security key. You can create a key using the “keytool” utility that is bundled with most JVMs. For example:

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

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

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

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

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

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

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

Handling HTTPS It’s sometimes necessary to support HTTPS specially in your implementation. One useful strategy is to create separate applications for HTTP and HTTPS, and then attach them to different virtual hosts (page 75), one for each protocol (the “resourceScheme” parameter). However, if the application behaves mostly the same for HTTP and HTTPS, but differs only in a few specific resources, it may be useful to check for HTTPS programmatically, via the conversation.reference.schemeProtocol API. For example:

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

Configuring Hosts

Define you virtual hosts under “/component/hosts/”. A minimal host definition:

```
var host = new VirtualHost(component.context)  
host.name = 'privatehost'  
component.hosts.add(host)
```

The “host.name” param exactly matches the string used in app.hosts (page 17) per each application.

A virtual host can route according to domain name, and incoming server IP address and port assignment:

```
host.resourceScheme = [string]  
host.resourceDomain = [string]  
host.resourcePort = [string]  
host.serverAddress = [string]  
host.serverPort = [string]  
host.hostScheme = [string]  
host.hostDomain = [string]  
host.hostPort = [string]
```

“*” wildcards are supported for all of these properties. Some notes:

- “resourceScheme”, “resourceDomain” and “resourcePort” refer to the actual incoming URI. Thus “resourcePort” would be meaningful *only if* URIs explicitly include a port number. To match a host to a specific server you would likely want to use “serverPort”.
- “hostScheme”, “hostDomain” and “hostPort” are for matching the “Host” HTTP header used by some proxies.

The Hungry Host

The “default” host that comes with the Prudence skeleton doesn’t configure any routing limitations, meaning that *all* incoming requests are routed. We call such a host “hungry.” Thus, if you want to use more than one host, you have to make sure that the hungry host is the *last* one, so that it acts as a fallback for when other hosts don’t match incoming requests. Otherwise, it would “eat” requests from other hosts. Because host files are executed in alphabetical order, a good way to ensure that the hungry host is last is to number your host filenames. For example:

- 1-public.js
- 2-private.js
- 3-default.js (this is the hungry host)

Deploying Multiple Sites

Using the virtual hosts and application model, Prudence can let you manage several sites using a single Prudence installation (a “container”). But is this always a good idea?

Advantages of Using a Single Container

1. Possibly simpler deployment: you are using a single base directory for the entire project, which might be easier for you. Because all configuration is done by JavaScript inside the container, it is very flexible.
2. Less memory use than running multiple JVMs.
3. Shared memory: you can use `application.sharedGlobals` to share state between applications.

Advantages of Using Multiple Containers

1. Possibly simpler deployment: several base directories can mean separate code/distribution repositories, which might be easier for you. Configuration will be at your web frontend (reverse proxy).
2. Robustness: crashes/deadlocks/memory leaks in one VM won’t affect others. With this in mind, it may even be worth having each *single application* running in its own JVM/container.
3. Run-time flexibility: you can restart the JVM for one container without affecting others that are running.

There is no performance advantage in either scenario. Everything in Prudence is designed around high-concurrency and threading, and generally threads are managed by the OS globally.

Well, there are caveats to that statement: Linux can group threads per running process for purposes of prioritization, but this is only really done for desktop applications. The feature could be possibly useful when running several containers, if you want to guarantee high thread priority to one of the containers over the others. Any of this would only effect *very* high concurrency and highly CPU-bound deployments.

Routing In-Depth

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

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

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

This abstract, flexible routing mechanism is one of Prudence’s most powerful features, but it’s important to understand these basic principles. A common misconception is that routing is based on the hierarchical structure of URIs, such that a child URI’s routing is somehow affected by its parent URI. While it’s possible to explicitly

design your routes hierarchically, routing is primarily to be understood in terms of the order of routers and filters along the way. A parent and child URI could thus use entirely different handlers.

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

Step 1: Servers Requests come in from servers. Prudence instances have at the minimum one server, but can have more than one. Each server listens at a particular HTTP port, and multiple servers may in turn be restricted to particular network interfaces on your machine. By default, Prudence has a single server that listens to HTTP requests on port 8080 coming in from all network interfaces.

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

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

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

Step 4: Applications Using `app.hosts`, you can configure which virtual hosts your application will be attached to, and the base URI for the application on each virtual host. An application can accept requests from several virtual hosts at once.

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

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

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

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

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

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

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

Working in a Cluster

TODO

Not only for horizontal scaling, but also for isolation.

Distributed Globals

Task Farming

Deploying

The Joys of Sincerity

Configuration-by-Script

Plugins

Deployment Strategies

Synchronization

Unison, rsync

Packaging

Maven Using your own repository (Nexus)

Debian/RPM

Version Control

Subversion

Git What to ignore

Directory Organization

Sincerity Standalone

Operating System Service

See Sincerity Manual

Monitoring

Security

SSL

Howto

HTTP Authentication

Locked-Down User

Service Plugin

Firewall

HTTP ports

Hazelcast ports

Cache backends

Database and other services

Proxying

Nginx

Apache

The “Host” Header

`conversation.request.hostRef`

Deploying Clusters

Loadbalancing

Perlbal

Security Concerns

Configuring Hazelcast

`/configuration/hazelcast/prudence/`
or
`/configuration/hazelcast.alt.conf`

Cache Backends

Utilities for Restlet

If you are a Restlet Java programmer, Prudence may still be of use to use. Prudence is also available as a small standalone Java library (a jar), and as such has several well-documented classes useful for any Java-written Restlet application. They’re all in the “`com.threecrickets.prudence.util`” package, and introduced below.

Utility Restlets

We wish these general-purpose utilities existed in the standard Restlet library!

- `CacheControlFilter`: A Filter that adds cache control directives to responses.
- `Injector`: A Filter that adds values to the request attributes before moving to the next restlet. It allows for a straightforward implementation of IoC (Inversion of Control).
- `StatusRestlet`: A restlet that always sets a specific status and does nothing else.

Client Data

These classes add no new functionality, but make working with some client data a bit easier.

- `CompressedStringRepresentation`: This is a `ByteArrayRepresentation` that can be constructed using text and an encoding, which it then compresses into bytes according the encoding. This is an alternative to using an `Encoder` filter, allowing you direct control over and access to the final representation.
- `ConversationCookie`: A modifiable extension of a regular `Cookie`. Tracks modifications, and upon calling `save()` stores them as a `CookieSetting`, likely in the `Response`. Also supports cookie deletion via `remove()`.
- `FormWithFiles`: A form that can parse `MediaType.MULTIPART_FORM_DATA` entities by accepting file uploads. Files will appear as parameters of type `FileParameter`.

Redirection

Restlet’s server-side redirection works by creating a new request. Unfortunately, this means that some information from the original request is lost. Prudence includes a set of classes that work together to preserve the original URI, which we here call the “captured” URI.

- `CapturingRedirector`: A `Redirector` that keeps track of the captured reference.
- `NormalizingRedirector`: A `Redirector` that normalizes relative paths. This may be unnecessary in future versions of Restlet. See Restlet issue 238.

Fallback Routing

“Fallback” is a powerful new routing paradigm introduced in Prudence that lets you attach multiple restlets to a single route.

- `Fallback`: A restlet that delegates `Restlet.handle(Request, Response)` to a series of targets in sequence, stopping at the first target that satisfies the condition of `wasHandled`. This is very useful for allowing multiple restlets a chance to handle a request, while “falling back” to subsequent restlets when those “fail.”
- `FallbackRouter`: A `Router` that takes care to bunch identical routes under `Fallback` restlets.

Resolver Selection

Restlet does not provide an easy way to use different template variable resolver instances. We’ve created new implementations of a few of the core classes that let you choose which resolver to use.

- `ResolvingTemplate`: A `Template` that allows control over which `Resolver` instances it will use.
- `ResolvingRedirector`: A `Redirector` that uses `ResolvingTemplate`.
- `ResolvingRouter`: A `Router` that uses `ResolvingTemplate` for all routes.

Web Filters

A set of filter classes for web technologies.

- `CssUnifyMinifyFilter`: A `Filter` that automatically unifies and/or compresses CSS source files, saving them as a single file. Unifying them allows clients to retrieve the CSS via one request rather than many. Compressing them makes their retrieval faster. Compression is done via `CSSMin`.
- `JavaScriptUnifyMinifyFilter`: A `Filter` that automatically unifies and/or compresses JavaScript source files, saving them as a single file. Unifying them allows clients to retrieve the JavaScript via one request rather than many. Compressing them makes their retrieval faster. Compression is done via John Reilly’s Java port of Douglas Crockford’s `JSMin`.
- `ZussFilter`: A `Filter` that automatically parses ZUSS code and renders CSS. Also supports minifying files, if the “.min.css” extension is used.
- `CustomEncoder`: A more localized alternative to using the `EncoderService`.

Upgrading from Prudence 1.1

Prudence 1.1 did not use Sincerity: instead, it was a self-contained container with everything in the box. This meant it could also not be modular, and instead supported several distributions (“flavors”) per supported programming language. Because of this, it allowed you to use any programming language for your bootstrapping code, and indeed the project maintained a separate set of bootstrapping code for all languages.

This was not only cumbersome in terms of documentation and maintenance, but it also made it hard to port applications between “flavors.”

With the move to Sincerity in Prudence 2.0, it was possible to make Prudence more minimal as well as more modular, as Sincerity handles the bootstrapping and installation of supported languages. Though Sincerity can

ostensibly run bootstrapping scripts in any Scripturian-supported language, it standardizes on JavaScript in order to maintain focus and portability. The bottom line is that if you used non-JavaScript flavors of Prudence 1.1, you will need to use JavaScript for your bootstrapping scripts, even if your application code (resources, scriptlets, tasks, etc.) is written in a different language.

To be 100% clear: *all “flavors” supported in Prudence 1.1 are still supported in Prudence 2.0*, and your application code will likely not even have to change. You *only* need (or rather, are recommended) to use JavaScript for bootstrapping.

Upgrading Applications

There are no significant API changes between Prudence 1.1 and Prudence 2.0. However, the bootstrapping and configuration has been completely overhauled. You will likely need to take a few minutes to rewrite your settings.js, routing.js, etc. Here is a step-by-step checklist:

1. Start with a new application based on the default template.
 - (a) Rename old application (add “-old”), for example: “myapp-old”
 - (b) Use the “prudence” tool to create a new application for your application name:

```
sincerity prudence create myapp
```
2. Copy over individual settings from settings.js, using the new manual (page 41) to find equivalences.
3. Redo your routing.js, using the new manual (page 73) to find equivalences. Prudence 2.0 uses a far more powerful and clearer routing configuration.
4. Rename “/resources/” files to add a “.m.” pre-extension (they are now called “manual resources”). Under Unix-like operation systems, you can rename the all files in the tree via a Perl expression using something like this:

```
find . -name "*.js" -exec rename -v 's /\.js$ /\.m.js /i' {} \;
```
5. Rename “/web/dynamic/” files to add a “.s.” pre-extension (they are now called “scriptlet resources”). Under Unix-like operation systems, you can rename the all files in the tree via a Perl expression using something like this:

```
find . -name "*.html" -exec rename -v 's /\.html$ /\.s.html /i' {} \;
```
6. Merge “/web/dynamic/” and “/web/static/” into “/resources/”.
7. Move “/web/fragments/” to “/libraries/includes”.

Upgrading the Component

TODO

Part III Articles

The Case for REST

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

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

Let’s call ours a “resource-oriented architecture.”

Resources

Our base unit is a “resource,” which, like an object in object-oriented architectures, encapsulates data with some functionality. However, we’ve learned from object-orientation that implementing arbitrary interfaces is a recipe for complexity: proxy generation, support for arbitrary types, marshaling, etc. All that often requires a middleware layer to handle the protocol. So, instead, we’ll keep it simple and define a limited, unified interface that would be just useful enough.

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

Identifiers

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

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

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

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

Delete

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

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

Read

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

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

Update

The problem with “update” is that it always references a certain version that we have “read” before. In some cases, though not all, we need some way to make sure that the data we expect to be there hasn’t changed since we’ve last “read” it. Let’s call this a “conditional update.” (In databases, this is called a “compare-and-set” atomic operation.)

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

We’ll need our resource-oriented architecture to support a general “conditional” operation feature.

Create

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

```
Create: /animal/cat/13/ -> Error, already exists  
Create: /animal/cat/14/ -> Error, already exists  
Create: /animal/cat/15/ -> Error, already exists
```

```
...
Create: /animal/cat/302041/ -> Success!
```

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

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

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

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

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

Aggregate Resources

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

```
/animal/cats/
```

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

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

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

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

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

Formats

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

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

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

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

Shared State

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

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

Summary of Features

So, what do we need?

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

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

Transactions... Not!

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

Let's Do It!

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

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

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

The Punchline

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

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

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

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

It's All About Infrastructure

The most important lesson to take from this exercise is the importance of infrastructure, something easily forgotten when planning architecture in ideal, abstract terms. This is why, I believe, Roy Fielding named Chapter 5 of his 2000 dissertation “Representational State Transfer (REST)” rather than, say, “resource-oriented architecture,” as we have here. Fielding, one of the authors of the HTTP protocol, was intimately familiar with its deployment challenges, and the name “REST” is intended to point out the key characteristic of its infrastructure: HTTP and similar protocols are designed for transferring lightly annotated data representations, nothing more. “Resources” are merely logical encapsulations of these representations, depending on a contract between client and server. The infrastructure does not, in itself, do anything in particular to maintain, say, a sensible hierarchy of addresses, arbitrary atomicity of CRUD operations, etc. That’s up to your implementation. But, representational state transfer—REST—is the mundane, underlying magic that makes it all possible.

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

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

Does REST Scale?

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

Consider that there are two uses of the word “scalable”:

First, it’s **the ability to respond to a growing number of user requests without degradation in response time**, by “simply” adding hardware (horizontal scaling) or replacing it with more powerful hardware (vertical scaling). This is the aspect of scalability that engineers care about. The simple answer is that REST can help, but it doesn’t stand out. SOAP, for example, can also do it pretty well. REST aficionados sometimes point out that REST is “stateless,” or “session-less,” both characteristics that would definitely help scale. But, this is misleading. Protocols might be stateless, but architectures built on top of them don’t have to be. For example, we’ve specifically talked about sessions here, and many web frameworks manage sessions via cookies. On the other hand, you can easily make poorly scalable REST. The bottom line is that there’s nothing in REST that guarantees scalability in *this* respect. Indeed, engineers coming to REST due to this false lure end up wondering what the big deal is. We wrote a whole article for [Scaling Tips \(page 89\)](#), which is indeed not specifically about REST.

The second use of “scalability” comes from the realm of enterprise and project management. It’s **the ability of your project to grow in complexity without degradation in your ability to manage it**. And that’s REST’s beauty—you already have the infrastructure, which is the hardest thing to scale in a project. You don’t need to deploy client stacks. You don’t need to create and update proxy objects for five different programming languages used in your enterprise. You don’t need to deploy incompatible middleware by three different vendors and spend weeks trying to force them to play well together. Why would engineers care about REST? Precisely because they don’t have to: they can focus on application engineering, rather than get bogged down by infrastructure management.

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

Prudence

Convinced? The best way to understand REST is to experiment with it. You’ve come to the right place. Start with the [tutorial \(page 6\)](#), and feel free to skip around the documentation and try things out for yourself. 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!

URI-space Architecture

REST does not standardize URI-spaces, and indeed has little to say about URI design. However, it does *imply* a preference for certain architectural principles. We go over much of the impetus in [The Case for REST article \(page 81\)](#), and suggest you start there.

It's a good idea to think very carefully about your URI-space. A RESTful URI-space can help you define well-encapsulated RESTful resources. Below are some topics to consider.

Nouns vs. Verbs

It's useful to think of URIs as syntactic *nouns*, a grammatical counterpart to HTTP's *verbs*. In other words, make sure that you do not include verbs in your URIs. Examples:

- Good: `"/service/{id}/status/"`
- Bad: `"/service/{id}/start/"`, `"/service/{id}/stop/"`

What is wrong with verbs in URIs?

One potential problem is clarity. Which HTTP verb should be used on a verb URI? Do you need to POST, PUT or DELETE to `"/service/{id}/stop/"` in order to stop the service? Of course, you can support all and document this, but it won't be immediately obvious to the user.

A second potential problem is that you need to keep increasing the size of your URI-space the more actions of this sort you want to support. This means more files, more classes, and generally more code. Handling these operations *inside* a single resource would just mean a simple "if" or "switch" statement and an extra method.

A third, more serious potential problem is idempotency. The idempotent verbs PUT and DELETE may be optimized by the HTTP infrastructure (for example, a smart load balancer) such that requests arrive more than once: this is allowed by the very definition of idempotency. However, your operations *may* not be semantically idempotent. For example, if a "stop" is sent to an already-stopped service, it may return an "already stopped" 500 error. In this case, if the infrastructure allows for two "stop" commands to come through, then the user may get an error even though the operation succeeded for the first "stop." There's an easy way around this: simply allow *only* POST, the non-idempotent verb, for all such operations. The infrastructure would never allow more than request to come through per POST. However, if you enforce the use of POST, you will lose the ability of the infrastructure to optimize for non-idempotency. POST is the least scalable HTTP verb.

The bottom line is that if you standardize on only using nouns for your URIs, you will avoid many of these semantic pitfalls.

Beware of gerunds! A URI such as `"/service/{id}/stopping/"` is technically a noun, but allows for some verb-related problems to creep in.

Do You Really Need REST?

In the above section, it was suggested that you prefer nouns to verbs. However, this preference may seem for constraining for your application. Your application may be very command-oriented, such that you will end up with a very small set of "noun" URIs that need to support a vast amount of commands.

REST shines because it is based on a tiny set of very tightly defined verbs: GET, POST, PUT, DELETE. The entire infrastructure is highly optimized around them: load balancers, caches, browsers, gateways, etc., all should know how best to handle each of these for maximum scalability and reliability. But, it's entirely possible that your needs cannot be easily satisfied by just four verbs.

And that's OK. REST is not always the best solution for APIs.

Instead, take a look at RPC (Remote Procedure Call) mechanisms. The Diligence framework, based on Prudence, provides easy and robust support for both JSON-RPC and XML-RPC in its RPC Service as well as Ext Direct in its Sencha Integration, allowing you to hook a JavaScript function on the server directly to a URI. In terms of HTTP, these protocols all use HTTP POST, and do not leverage the HTTP infrastructure as well as a more fully RESTful API. But, one size does not fit all, and an RPC-based solution may prove a better match for your project.

It's also perfectly possible to use *both* REST and RPC in your project. Use each approach where it is most appropriate.

Hierarchies

It's entirely a matter of convention that the use of “/” in URIs implies hierarchy. Historically, the convention was likely imported from filesystem paths, where a name before a “/” signifies a directory rather than a file.

This convention is useful because it's very familiar to users, but additionally it implies semantic properties that can add clarity and power to your resource design. There are two possible semantic principles you may consider:

1. A descendant resource *belongs to* its ancestor, such that resources have cascading relationships in the hierarchy. This implies two rules:
 - (a) Operations on a resource *may* affect descendants. This rule is most obvious when applied to the DELETE verb: for example, if you delete “/user/{id}/”, then it is expected that the resources at “/user/{id}/profile/” and “/user/{id}/preferences/” also be deleted. A PUT, too, would also affect the descendant resources.
 - (b) Operations on a resource *should not* affect ancestors. In other words, a descendant's state is isolated from its ancestors. For example, if I send a POST to “/user/{id}/profile/”, the representation at “/user/{id}/” should remain unaltered.
2. A descendant resource *belongs to* its ancestor and also represents *an aspect of* its ancestor, such that operations on a resource can be fine-tuned to particular aspects of it. This implies three rules:
 - (a) Descendant representations *are included* in ancestor representations. For example, a GET on “/service/{id}/” would include information about the status that you would see if you GET on “/service/{id}/status/”. The latter URI makes it easier for the client to direct operations at the status aspect.
 - (b) Operations on a resource *may* affect descendants. See above.
 - (c) Operations on a resource *will* affect ancestors. This is the *opposite* of the above: the descendant's state is *not isolated* from its ancestors. For example, a POST to “/service/{id}/status/” would surely also affect “/service/{id}/”, which includes the status.

You can see from the difference between rule 1.b and 2.c. that it's important to carefully define the *nature* of your hierarchical relationships. Unlike filesystem directory hierarchies, in a URI-space there is no single standard or interpretation of what of a hierarchy means.

Formats

A format should not be considered “an aspect” in the sense used in principle 2. For example, “/service/{id}/html/” would not be a good way to support an HTML format for “/service/{id}/”. The reason is that you would be allowing for more than one URI for the same encapsulated resource, creating confusion for users. For example, it's not immediately clear what would happen if they DELETE “/service/{id}/html/”. Would that just remove the ability to represent the service as HTML? Or delete the service itself?

Supporting multiple formats is best handled with content negotiation, within the REST architecture. If further formatting is required, URI query parameters can be used. For example: “/service/{id}/?indent=2” might return a JSON representation with 2-space indentation.

Plural vs. Singular

You'll see RESTful implementations that use either convention. The advantage of using the singular form is that you have less addresses, and what some people would call a more elegant scheme:

```
/animal/cat/12/ -> Just one cat
/animal/cat/    -> All cats
```

Why add another URL format when a single one is enough to do the work? One reason is that you can help the client avoid potential errors. For example, the client probably uses a variable to hold the ID of the cat and then constructs the URL dynamically. But, what if the client forgets to check for empty IDs? It might then construct a URL in the form “/animal/cat/” which would then successfully access *all* cats. This can cause unintended consequences and be difficult to debug. If, however, we used this scheme:

```
/animal/cat/12/ -> Just one cat
/animal/cats/ -> All cats
```

...then the form “/animal/cat/” would route to our singular cat resource, which would indeed not find the “empty” cat and return the expected, debuggable 404 error.

From this example, we can extract a good rule of thumb: *clearly separate URI templates by usage*, so that mistakes cannot happen. More URI types means more debuggability.

Documenting Your URI-Space

If you create a programming language API, you will surely want to document it in a human language. You will want to define the acceptable types and usages of function arguments, describe return values, possible raises exceptions, add implementation and performance notes, etc. Many programming languages include tools for embedding such documentation as comments in the source code, and generating a reference manual from it.

Consider that documenting your URI-space is just as important. A tool to generate such documentation for you is being considered for a future version of Prudence (not available in 2.0). Until we have it, consider adopting a resource documentation standard for your project. Here’s a suggestion:

```
/*
 * This resource represents a service running on the server. Servers have unique
 * IDs defined by integers. A service can be either active or inactive.
 *
 * Use POST to change the name or status of an existing service. You may
 * not use it change the ID of an existing service. PUT will create a new
 * service, and DELETE will stop and remove it.
 *
 * Implementation note: if you PUT a service with an ID that already exists, then
 * it will only stop and restart the service rather than removing/recreate it,
 * which would be too resource intensive. Use DELETE if you absolutely need the
 * service to be removed first, or set the "clean" query param to "true" to
 * force removal.
 *
 * @URI /service/{id:decimal}/
 * @Aspect /service/{id:decimal}/status/
 * @Verb GET, POST, PUT, DELETE
 * @MediaType application/json, application/xml, text/plain = as JSON
 * @Query {decimal} indent If non-zero will return a human-readable indented version
 * of the representation with lines indented by the integer value
 * @Query {boolean} clean If "true" or "yes" or "1" wil force removal of an existing
 * service during a PUT operation on an existing service
 *
 * @Representation {application/json}
 * {
 *   "id": number,
 *   "name": string (the service name),
 *   "status": string:"active"|"inactive"
 * }
 *
 * @Payload {application/json}
 * {
 *   "name": ...
 *   "status": ...
 * }
 */

/*
 * This resource represents the status of a service.
 */
```



```

* DELETE on this resource is identical to PUT or POST with "inactive".
* PUT and POST are handled identically.
*
* @URI /service/{id:decimal}/status/
* @AspectOf /service/{id:decimal}/
* @Verb GET, POST, PUT, DELETE
* @MediaType text/plain
*
* @Representation {text/plain}
* "active"|"inactive"
*/

```

Scaling Tips

Scalability is the ability to respond to a growing number of user requests without degradation in response time. Two variables influence it: 1) your total number of threads and 2) the time it takes each thread to process a request. Increasing the number of threads seems straightforward: you can keep adding more machines behind load balancers. However, the two variables are tied, as there are diminishing returns and even reversals: beyond a certain point, time per request can actually grow longer as you add threads and machines.

Let's ignore the first variable here, because the challenge of getting more machines is mostly financial. It's the second that you can do something about as an engineer.

If you want your application to handle many concurrent users, then you're fighting this fact: a request will get queued in the best case or discarded in the worst case if there is no thread available to serve it. Your challenge is to make sure that a thread is always available. And it's not easy, as you'll find out as you read through this article. Minimizing the time per request becomes an architectural challenge that encompasses the entire structure of your application

Performance Does Not Equal Scalability

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

Get it? Performance does not equal scalability.

This is an important mantra for two reasons:

1. Performant Can Mean Less Scalable

Optimizing for performance can adversely affect your scalability. The reason is contextual: when you optimize for performance, you often work in an isolated context, specifically so you can accurately measure response times and fine-tune them. For example, making sure that a specific SQL query is fast would involve just running that query. A full-blown experiment involving millions of users doing various operations on your application would make it very hard to accurately measure and optimize the query. Unfortunately, by working in an isolated context you cannot easily see how your efforts would affect other parts of an application. To do so would require a lot of experience and imagination. To continue our example, in order to optimize your one SQL query you might create an index. That index might need to be synchronized with many servers in your cluster. And that synchronization overhead, in turn, could seriously affect your ability to scale. Congratulations! You've made one query run fast in a situation that never happens in real life, and you've brought your web site to a halt.

One way to try to get around this is to fake scale. Tools such as JMeter, Siege and ApacheBench can create "load." They also create unfounded confidence in engineers. If you simulate 10,000 users bombarding a single web page, then you're, as before, working in an isolated context. All you've done is add concurrency to your performance optimization measurements. Your application pathways might work optimally in these situations, but this might very well be due to the fact that the system is not doing anything else. Add those "other" operations in, and you might get worse site capacity than you did before "optimizing."

2. Wasted Effort

Even if you don't adversely affect your scalability through optimizing for performance, you might be making no gains, either. No harm done? Well, plenty of harm, maybe. Optimizing for performance might waste a lot of

development time and money. This effort would be better spent on work that could actually help scalability.

And, perhaps more seriously, it demonstrates a fundamental misunderstanding of the problem field. If you don't know what your problems are, you'll never be able to solve them.

Pitfalls

Study the problem field carefully. Understand the challenges and potential pitfalls. You don't have to apply every single scalability strategy up-front, but at least make sure you're not making a fatal mistake, such as binding yourself strongly to a technology or product with poor scalability. A bad decision can mean that when you need to scale up in the future, no amount of money and engineering effort would be able to save you before you lose customers and tarnish your brand.

Moreover, be very careful of blindly applying "successful" strategies used and recommended by others to your product. What worked for them might not work for you. In fact, there's a chance that their strategy doesn't even work for them, and they just think it did because of a combination of seemingly unrelated factors. The realm of web scalability is still young, full of guesswork, intuition and magical thinking. Even the experts are often making it up as they're going along.

Generally, be very suspicious of products or technologies being touted as "faster" than others. *"Fast" doesn't say anything about the ability to scale.* Is a certain database engine "fast"? That's important for certain applications, no doubt. But maybe the database is missing important clustering features, such that it would be a poor choice for scalable applications. Does a certain programming language execute faster than another? That's great if you're doing video compression, but speed of execution might not have a noticeable effect on scalability. Web applications mostly do I/O, not computation. The same web application might have very similar performance characteristics whether it's written in C++ or PHP.

Moreover, if the faster language is difficult to work with, has poor debugging tools, limited integration with web technologies, then it would slow down your work and your ability to scale.

Speed of execution can actually help scalability in its financial aspect: If your application servers are constantly at maximum CPU load, then a faster execution platform would let you cram more web threads into each server. This could help you reduce costs. For example, see Facebook's HipHop: they saved millions by translating their PHP code to C. Because Prudence is built on the fast JVM platform, you're in good hands in this respect. Note, however, that there's a potential pitfall to high performance: more threads per machine would also mean more RAM requirements per machine, which also costs money. Crunch the numbers and make sure that you're actually saving money by increasing performance. Once again, performance does not equal scalability.

That last point about programming languages is worth some elaboration. Beyond how well your chosen technologies perform, it's important to evaluate them in terms to how easy they are to manage. Large web sites are large projects, involving large teams of people and large amounts of money. That's difficult enough to coordinate. You want the technology to present you with as few extra managerial challenges as possible.

Beware especially of languages and platforms described as "agile," as if they somehow embody the spirit of the popular Agile Manifesto. Often, "agile" seems to emphasize the following features: forgiveness for syntax slips, light or no type checking, automatic memory management and automatic concurrency—all features that seem to speed up development, but could just as well be used for sloppy, error-prone, hard-to-debug, and hard-to-fix code, slowing down development in the long run. If you're reading this article, then your goal is likely not to create a quick demo, but a stable application with a long, evolving life span.

Ignore the buzzwords ("productivity", "fast"), and instead make sure you're choosing technology that you can control, instead of technology that will control you.

We discuss this topic some more in ["The Case for Rest" \(page 81\)](#). By building on the existing web infrastructure, Prudence can make large Internet projects easier to manage.

Analysis

Be especially careful of applying a solution before you know if you even have a problem.

How to identify your scalability bottlenecks? You can create simulations and measurements of scalability rather than performance. You need to model actual user behavior patterns, allow for a diversity of such behaviors to happen concurrently, and replicate this diversity on a massive scale.

Creating such a simulation is a difficult and expensive, as is monitoring and interpreting the results and identifying potential bottlenecks. This is the main reason for the lack of good data and good judgment about how to

scale. Most of what we know comes from tweaking real live web sites, which either comes at the expense of user experience, or allows for very limited experimentation. Your best bet is to hire a team who's already been through this before.

Optimizing for Scalability

In summary, your architectural objective is to increase concurrency, not necessarily performance. Optimizing for concurrency means breaking up tasks into as many pieces as possible, and possibly even breaking requests into smaller pieces. We'll cover numerous strategies here, from frontend to backend. Meanwhile, feel free to frame these inspirational slogans on your wall:

Requests are hot potatoes: Pass them on!

And:

It's better to have many short requests than one long one.

Caching

Retrieving from a cache can be orders of magnitude faster than dynamically processing a request. It's your most powerful tool for increasing concurrency.

Caching, however, is only effective if there's something in the cache. It's pointless to cache fragments that appear only to one user on only one page that they won't return to. On the other hand, there may very well be fragments on the page that will recur often. If you design your page carefully to allow for fragmentation, you will reap the benefits of fine-grained caching. Remember, though, that the outermost fragment's expiration defines the expiration of the included fragments. It's thus good practice to define no caching on the page itself, and only to cache fragments.

In your plan for fine-grained caching with Prudence, take special care to isolate those fragments that cannot be cached, and cache everything around them.

Make sure to change the cache key (page ??) to fit the lowest common denominator: you want as many possible requests to use the already-cached data, rather than generating new data. Note that, by default, Prudence includes the request URI in the cache key. Fragments, though, may very well appear identically in many different URIs. You would thus not want the URI as part of their cache key.

Cache aggressively, but also take cache validation seriously. Make good use of Prudence's cache tags (page ??) to allow you to invalidate portions of the cache that should be updated as data changes. Note, though, that every time you invalidate you will lose caching benefits. If possible, make sure that your cache tags don't cover too many pages. Invalidate only those entries that really need to be invalidated.

(It's sad that many popular web sites do cache validation so poorly. Users have come to expect that sometimes they see wrong, outdated data on a page, sometimes mixed with up-to-date data. The problem is usually solved within minutes, or after a few browser refreshes, but please do strive for a better user experience in your web site!)

If you're using a deferred task handler (page 94), you might want to invalidate tagged cache entries when tasks are done. Consider creating a special internal API that lets the task handler call back to your application to do this.

How long should you cache? As long as the user can bear! In a perfect world, of limitless computing resources, all pages would always be generated freshly per request. In a great many cases, however, there is no harm at all if users see some data that's a few hours or a few days old.

Note that even very small cache durations can make a big difference in application stability. Consider it the maximum throttle for load. For example, a huge sudden peak of user load, or even a denial-of-service (DOS) attack, might overrun your thread pool. However, a cache duration of just 1 second would mean that your page would never be generated more than once every second. You are instantly protected against a destructive scenario.

Cache Warming

Caches work best when they are "warm," meaning that they are full of data ready to be retrieved.

A "cold" cache is not only useless, but it can also lead indirectly to a serious problem. If your site has been optimized for a warm cache, starting from cold could significantly strain your performance, as your application

servers struggle to generate all pages and fragments from scratch. Users would be getting slow response times until the cache is significantly warm. Worse, your system could crash under the sudden extra load.

There are two strategies to deal with cold caches. The first is to allow your cache to be persistent, so that if you restart the cache system it retains the same warmth it had before. This happens automatically with database-backed caches (page 93). The second strategy is to deliberately warm up the cache in preparation for user requests.

Consider creating a special external process or processes to do so. Here are some tips:

1. Consider mechanisms to make sure that your warmer does not overload your system or take too much bandwidth from actual users. The best warmers are adaptive, changing their load according to what the servers can handle. Otherwise, consider shutting down your site for a certain amount of time until the cache is sufficiently warm.
2. If the scope is very large, you will have to pick and choose which pages to warm up. In Prudence, this is supported via `app.preheat` (page 19). You would want to choose only the most popular pages, in which case you might need a system to record and measure popularity. For example, for a blog, it's not enough just to warm up, say, the last two weeks of blog posts, because a blog post from a year ago might be very popular at the moment. Effective warming would require you to find out how many times certain blog posts were hit in the past two weeks. It might make sense to embed this auditing ability into the cache backend itself.

Pre-Filling the Cache

If there are thousands of ways in which users can organize a data view, and each of these views is particular to one user, then it may make little sense to cache them individually, because individual schemes would hardly ever be re-used. You'll just be filling up the cache with useless entries.

Take a closer look, though:

1. It may be that of the thousands of organization schemes only a few are commonly used, so it's worth caching the output of just those.
2. It could be that these schemes are similar enough to each other that you could generate them all in one operation, and save them each separately in the cache. Even if cache entries will barely be used, if they're cheap to create, it still might be worth creating them.

This leads us to an important point:

Prudence is a "frontend" platform, in that it does not specify which data backend, if at all, you should use. Its cache, however, is general purpose, and you can store in it anything that you can encode as a string.

Let's take as a pre-filling example a tree data structure in which branches can be visually opened and closed. Additionally, according to user permissions different parts of the tree may be hidden. Sounds too complicated to cache all the view combinations? Well, consider that you can trigger, upon any change to the tree data structure, a function that loops through all the different iterations of the tree recursively and saves a view of each of them to the cache. The cache keys can be something like "branch1+.branch2-.branch3+", with "+" signifying "-" whether the branch is visually open or closed. You can use similar +'s and -'s for permissions, and create views per permission combinations. Later, when users with specific permissions request different views of the tree, no problem: all possibilities were already pre-filled. You might end up having to generate and cache thousands of views at once, but the difference between generating one view and generating thousands of views may be quite small, because the majority of that duration is spent communicating with the database backend.

If generating thousands of views takes too long for the duration of a single request, another option is to generate them on a separate thread. Even if it takes a few minutes to generate all the many, many tree views combinations, it might be OK in your application for views to be a few minutes out-of-date. Consider that the scalability benefits can be very significant: you generate views only *once* for the entire system, while millions of concurrent users do a simple retrieval from the cache.

Caching the Data Backend

Pre-filling the cache can take you very far. It is, however, quite complicated to implement, and can be ineffective if data changes too frequently or if the cache has to constantly be updated. Also, it's hard to scale the pre-filling to *millions* of fragments.

If we go back to our tree example above, the problem was that it was too costly to fetch the entire tree from the database. But what if we cache the tree itself? In that case, it would be very quick to generate any view of the tree on-demand. Instead of caching the view, we'd be caching the data, and achieving the same scalability gains.

Easy, right? So why not cache *all* our data structures? The reason is that it's very difficult to do this correctly beyond trivial examples. Data structures tend to have complex interrelationships (one-to-many, many-to-many, foreign keys, recursive tree structures, graphs, etc.) such that a change in data at one point of the structure may alter various others in particular ways. For example, consider a calendar database, and that you're caching individual days with all their events. Weekly calendar views are then generated on the fly (and quickly) for users according to what kinds of events they want to see in their personal calendars. What happens if a user adds a recurring event that happens every Monday? You'll need to make sure that all Mondays currently cached would be invalidated, which might mean tagging all these as "monday" using Prudence's cache tags. This requires a specific caching strategy for a specific application.

By all means, cache your data structures if you can't easily cache your output, but be aware of the challenge!

Cache Backends

Your cache backend can become a bottleneck to scalability if 1) it can't handle the amount of data you are storing, or 2) it can't respond quickly enough to cache fetching.

Before you start worrying about this, consider that it's a rare problem to have. Even if you are caching millions of pages and fragments, a simple relational-database-backed cache, such as Prudence's `SqlCache` implementations, could handle this just fine. A key/value table is the most trivial workload for relational databases, and it's also easy to shard (page 97). Relational database are usually very good at caching these tables in their memory and responding optimally to read requests. Prudence even lets you chain caches together to create tiers: an in-process memory cache in front of a SQL cache would ensure that many requests don't even reach the SQL backend.

High concurrency can also be handled very well by this solution. Despite any limits to the number of concurrent connections you can maintain to the database, each request is handled very quickly, and it would require *very* high loads to saturate. The math is straightforward: with a 10ms average retrieval time (very pessimistic!) and a maximum of 10 concurrent database connections (again, pessimistic!) you can handle 1,000 cache hits per second. A real environment would likely provide results orders of magnitude better.

The nice thing about this solution is that it uses the infrastructure you already have: the database.

But, what if you need to handle *millions* of cache hits per second? First, let us congratulate you for your global popularity. Second, there is a simple solution: distributed memory caches. Prudence comes with Hazelcast and support for memcached, which both offer much better scalability than database backends. Because the cache is in memory, you lose the ability to easily persist your cache and keep it warm: restarting your cache nodes will effectively reset them. There are workarounds—for example, parts of the cache can be persisted to a second database-backed cache tier—but this is a significant feature to lose.

Actually, Hazelcast offers fail-safe, live backups. While it's not quite as permanent as a database, it might be good enough for your needs. And memcached has various plugins that allow for real database persistence, though using them would require you to deal with the scalability challenges of database backends (page 98).

You'll see many web frameworks out there that support a distributed memory cache (usually memcached) and recommend you use it ("it's fast!" they claim, except that it can be slower per request than optimized databases, and that anyway performance does not equal scalability). We'd urge you to consider that advice carefully: keeping your cache warm is a challenge made much easier if you can store it in a persistent backend, and database backends can take you very far in scale without adding a new infrastructure to your deployment. It's good to know, though, that Prudence's support for Hazelcast and memcached is there to help you in case you reach the popularity levels of LiveJournal, Facebook, YouTube, Twitter, etc.

Client-Side Caching

Modern web browsers support client-side caching, a feature meant to improve the user experience and save bandwidth costs. A site that makes good use of client-side caching will appear to work fast for users, and will also help to increase your site's popularity index with search engines.

Optimizing the user experience is not the topic of this article: our job here is to make sure your site doesn't degrade its performance as load increases. However, client-side caching can indirectly help you scale by reducing the number of hits you have to take in order for your application to work.

Actually, doing a poor job with client-side caching can help you scale: users will hate your site and stop using it—voilà, less hits you have to deal with. OK, that was a joke!

Generally, Prudence handles client-side caching automatically. If you cache a page, then headers will be set to ask the client to cache for the same length of time. By default, conditional mode is used: every time the client tries to view a page, it will make a request to make sure that nothing has changed since their last request to the page. In case nothing has changed, no content is returned.

You can also turn on “offline caching” mode, in which the client will avoid even that quick request. Why not enable offline caching by default? Because it involves some risk: if you ask to cache a page for one week, but then find out that you have a mistake in your application, then users will not see any fix you publish until their local cache expires, which can take up to a week! It’s important that you understand the implications before using this mode. See the [dynamicWebClientCachingMode application setting \(page ??\)](#).

It’s generally safer to apply offline caching to your static resources, such as graphics and other resources. A general custom is to ask the client to cache these “forever” (10 years), and then, if you need to update a file, you simply create a new one with a new URL, and have all your HTML refer to the new version. Because clients cache according to URL, their cached for the old version will simply not be ignored. See [CacheControlFilter \(page ??\)](#). There, you’ll also see some more tricks Prudence offers you to help optimize the user experience, such as unifying/minimizing client-side JavaScript and CSS.

Upstream Caching

If you need to quickly scale a web site that has not been designed for caching, a band-aid is available: upstream caches, such as Varnish, NCache and even Squid. For archaic reasons, these are called “reverse proxy” caches, but they really work more like filters. According to attributes in the user request (URL, cookies, etc.), they decide whether to fetch and send a cached version of the response, or to allow the request to continue to your application servers.

The crucial use case is archaic, too. If you’re using an old web framework in which you cannot implement caching logic yourself, or cannot plug in to a good cache backend, then these upstream caches can do it for you.

They are problematic in two ways:

1. Decoupling caching logic from your application means losing many features. For example, invalidating portions of the cache is difficult if not impossible. It’s because of upstream caching, indeed, that so many web sites do a poor job at showing up-to-date information.
2. Filtering actually implements a kind of partitioning, but one that is vertical rather than horizontal. In horizontal partitioning, a “switch” decides to send requests to one cluster of servers or another. Within each cluster, you can control capacity and scale. But in vertical partitioning, the “filter” handles requests internally. Not only is the “filter” more complex and vulnerable than a “switch” as a frontend connector to the world, but you’ve also complicated your ability to control the capacity of the caching layer. It’s embedded inside your frontend, rather than being another cluster of servers. We’ll delve into [backend partitioning \(page 96\)](#) below.

Unfortunately, there is a use case relevant for newer web frameworks, too: if you’ve designed your application poorly, and you have many requests that could take a long time to complete, then your thread pools could get saturated when many users are concurrently making those requests. When saturated, you cannot handle even the super-quick cache requests. An upstream cache band-aid could, at least, keep serving its cached pages, even though your application servers are at full capacity. This creates an illusion of scalability: some users will see your web site behaving fine, while others will see it hanging.

The real solution would be to re-factor your application so that it does not have long requests, guaranteeing that you’re never too saturated to handle tiny requests. Below are tips on how to do this.

Dealing with Lengthy Requests

One size does not fit all: you will want to use different strategies to deal with different kinds of tasks.

Deferrable Tasks

Deferrable tasks are tasks that can be resolved later, without impeding on the user’s ability to continue using the application.

If the deferrable task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service. Prudence's tasks (page ??) implementation is a great solution for deferrable tasks, as it lets you run tasks on other threads or even distribute them in a Hazelcast cluster.

Deferring tasks does present a challenge to the user experience: What do you do if the task fails and the user needs to know about it? One solution can be to send a warning email or other kind of message to the user. Another solution could be to have your client constantly poll in the background (via "AJAX") to see if there are any error messages, which in turn might require you to keep a queue of such error messages per user.

Before you decide on deferring a task, think carefully of the user experience: for example, users might be constantly refreshing a web page waiting to see the results of their operation. Perhaps the task you thought you can defer should actually be considered necessary (see below)?

Necessary Tasks

Necessary tasks are tasks that must be resolved before the user can continue using the application.

If the necessary task is deterministically fast, you can do all processing in the request itself. If not, you should queue the task on a handling service and return a "please wait" page to the user.

It would be nice to add a progress bar or some other kind of estimation of how long it would take for the task to be done, with a maximum duration set after which the task should be considered to have failed. The client would poll until the task status is marked "done," after which they would be redirected back to the application flow. Each polling request sent by the client could likely be processed very quickly, so this strategy effectively breaks the task into many small requests ("It's better to have many short requests than one long one").

Prudence's tasks (page ??) implementation is a good start for creating such a mechanism: however, it would be up to you to create a "please wait" mechanism, as well as a way to track the tasks' progress and deal with failure.

Implementing such a handling service is not trivial. It adds a new component to your architecture, one that also has to be made to scale. One can also argue that it adversely affects user experience by adding overhead, delaying the time it takes for the task to complete. The bottom line, though, is you're vastly increasing concurrency and your ability to scale. And, you're improving the user experience in one respect: they would get a feedback on what's going on rather than having their browsers spin, waiting for their requests to complete.

File Uploads

These are potentially very long requests that you cannot break into smaller tasks, because they depend entirely on the client. As such, they present a unique challenge to scalability.

Fortunately, Prudence handles client requests via non-blocking I/O, meaning that large file uploads will not hold on to a single thread for the duration of the upload. See conversation form (page ??).

Unfortunately, many concurrent uploads will still saturate your threads. If your application relies on frequent file uploads, you are advised to handle such requests on separate Prudence instances, so that uploads won't stop your application from handling other web requests. You may also consider using a third-party service specializing in file storage and web uploads.

Asynchronous Request Processing

Having the client poll until a task is completed lets you break up a task into multiple requests and increase concurrency. Another strategy is to break an *individual request* into pieces. While you're processing the request and preparing the response, you can free the web thread to handle other requests. When you're ready to deliver content, you raise a signal, and the next available web thread takes care of sending your response to the client. You can continue doing this indefinitely until the response is complete. From the client's perspective it's a single request: a web browser, for example, would spin until the request was completed.

You might be adding some extra time overhead for the thread-switching on your end, but the benefits for scalability are obvious: you are increasing concurrency by shortening the time you are holding on to web threads.

For web services that deliver heavy content, such as images, video, audio, it's absolutely necessary. Without it, a single user could tie up a thread for minutes, if not hours. You would still get degraded performance if you have more concurrent users than you have threads, but at least degradation will be shared among users. Without asynchronous processing, each user would tie up one thread, and when that finite resource is used up, more users won't be able to access your service.

Even for lightweight content such as HTML web pages, asynchronous processing can be a good tactic for increasing concurrency. For example, if you need to fetch data from a backend with non-deterministic response time, it's best to free the web thread until you actually have content available for the response.

It's not a good idea to do this for every page. While it's better to have many short requests instead of one long one, it's obviously better to have one short request rather than many short ones. Which web requests are good candidates for asynchronous processing?

1. Requests for which processing is made of independent operations. (They'll likely be required to work in sequence, but if they can be processed in parallel, even better!)
2. Requests that must access backend services with non-deterministic response times.

And, even for #2, if the service can take a *very* long time to respond, consider that it might be better to queue the task on a task handler and give proper feedback to the user.

And so, after this lengthy discussion, it turns out that there aren't that many places where asynchronous processing can help you scale. Caching is far more useful.

As of version 1.1, Prudence has limited support for asynchronous processing, via [conversation.defer \(page ??\)](#). Better support is planned for a future version.

Backend Partitioning

You can keep adding more nodes behind a load balancer insofar as each request does not have to access shared state. Useful web applications, however, are likely data-driven, requiring considerable state.

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

The backend is one place where high-performance hardware can help. Ten expensive, powerful machines might be equal in total power to forty cheap machines, but they require a quarter of the synchronization overhead, giving you more elbow room to scale up. Fewer nodes means better scalability.

But CPUs can only take you so far.

Partitioning is as useful to backend scaling as caching is to web request scaling. Rather than having one big cluster of identical nodes, you would have several smaller, independent clusters. This lets you add nodes to each cluster without spreading synchronization overhead everywhere. The more partitions you can create, the better you'll be able to scale.

Partitioning can happen in various components of your application, such as application servers, the caching system, task queues, etc. However, it is most effective, and most complicated to implement, for databases. Our discussion will thus focus on relational (SQL) databases. Other systems would likely require simpler subsets of these strategies.

Reads vs. Writes

This simple partitioning scheme greatly reduces synchronization overhead. Read-only servers will never send data to the writable servers. Also, knowing that they don't have to handle writes means you can optimize their configurations for aggressive caching.

(In fact, some database synchronization systems will only let you create this kind of cluster, providing you with one "master" writable node and several read-only "slaves." They force you to partition!)

Another nice thing about read/write partitioning is that you can easily add it to all the other strategies. Any cluster can thus be divided into two.

Of course, for web services that are heavily balanced towards writes, this is not an effective strategy. For example, if you are implementing an auditing service that is constantly being bombarded by incoming data, but is only queried once in a while, then an extra read-only node won't help you scale.

Note that one feature you lose is the ability to have a transaction in which a write *might* happen, because a transaction cannot contain both a read-only node and a write-only node. If you must have atomicity, you will have to do your transaction on the writable cluster, or have two transactions: one to lookup and see if you need to change the data, and the second to perform the change—while first checking again that data didn't change since the previous transaction. Too much of this obviously lessens the effectiveness of read/write partitioning.

By Feature

The most obvious and effective partitioning scheme is by feature. Your site might offer different kinds of services that are functionally independent of each other, even though they are displayed to users as united. Behind the

scenes, each feature uses a different set of tables. The rule of thumb is trivial: if you can put the tables in separate databases, then you can put these databases in separate clusters.

One concern in feature-based partitioning is that there are a few tables that still need to be shared. For example, even though the features are separate, they all depend on user settings that are stored in one table.

The good news is that it can be cheap to synchronize just this one table between all clusters. Especially if this table doesn't change often—how often do you get new users signing up for your service?—then synchronization overhead will be minimal.

If your database system doesn't let you synchronize individual tables, then you can do it in your code by writing to all clusters at the same time.

Partitioning by feature is terrific in that it lets you partition other parts of the stack, too. For example, you can also use a different set of web servers for each feature.

Also consider that some features might be candidates for using a “NoSQL” database (page 98). Choose the best backend per feature.

By Section

Another kind of partitioning is sometimes called “sharding.” It involves splitting up tables into sections that can be placed in different databases. Some databases support sharding as part of their synchronization strategy, but you can also implement it in your code. The great thing about sharding is that it lets you create as many shards (and clusters) as you want. It's the key to the truly large scale.

Unfortunately, like partitioning by feature, sharding is not always possible. You need to also shard all related tables, so that queries can be self-contained within each shard. It's thus most appropriate for one-to-many data hierarchies. For example, if your application is a blog that supports comments, then you put some blogs and their comments on one shard, and others in another shard. However, if, say, you have a feature where blog posts can refer to other arbitrary blog posts, then querying for those would have to cross shard boundaries.

The best way to see where sharding is possible is to draw a diagram of your table relationships. Places in the diagram which look like individual trees—trunks spreading out into branches and twigs—are good candidates for sharding.

How to decide which data goes in which shard?

Sometimes the best strategy is arbitrary. For example, put all the even-numbered IDs in one shard, and the odd-numbered ones in another. This allows for straightforward growth because you can just switch it to division by three if you want three shards.

Another strategy might seem obvious: If you're running a site which shows different sets of data to different users, then why not implement it as essentially separate sites? For example, a social networking site strictly organized around individual cities could have separate database clusters per city.

A “region” can be geographical, but also topical. For example, a site hosting dance-related discussion forums might have one cluster for ballet and one for tango. A “region” can also refer to user types. For example, your social networking site could be partitioned according to age groups.

The only limitation is queries. You can still let users access profiles in other regions, but cross-regional relational queries won't be possible. Depending on what your application does, this could be a reasonable solution.

A great side-benefit to geographical partitioning is that you can host your servers at data centers within the geographical location, leading to better user experiences. Regional partitioning is useful even for “NoSQL” databases.

Coding Tips for Partitioning

If you organize your code well, it would be very easy to implement partitioning. You simply assign different database operations to use different connection pools. If it's by feature, then you can hard code it for those features. If it's sharding, then you add a switch before each operation telling it which connection pool to use.

For example:

```
def get_blogger_profile(user_id):
    connection = blogger_pool.get_connection()
    ...
    connection.close()

def get_blog_post_and_comments(blog_post_id):
    shard_id = object.id % 3
    connection = blog_pools[shard_id].get_connection()
```

```
...
connection.close()
```

Unfortunately, some programming practices make such an effective, clean organization difficult.

Some developers prefer to use ORMs (object-relational mappers) rather than access the database directly. Many ORMs do not easily allow for partitioning, either because they support only a single database connection pool, or because they don't allow your objects to be easily shared between connections.

For example, your logic might require you to retrieve an “object” from the database, and only then decide if you need to alter it or not. If you're doing read/write partitioning, then you obviously want to read from the read partition. Some ORMs, though, have the object tied so strongly to an internal connection object that you can't trivially read it from one connection and save it into another. You'd either have to read the object initially from the write partition, minimizing the usefulness of read/write partitioning, or re-read it from the write partition when you realize you need to alter it, causing unnecessary overhead. (Note that you'll need to do this anyway if you need the write to happen in a transaction.)

Object oriented design is also problematic in a more general sense. The first principle of object orientation is “encapsulation,” putting your code and data structure in one place: the class. This might make sense for business logic, but, for the purposes of re-factoring your data backend for partitioning or other strategies, you really don't want the data access code to be spread out among dozens of classes in your application. You want it all in one place, preferably even one source code file. It would let you plug in a whole new data backend strategy by replacing this source code file. For data-driven web development, you are better off not being too object oriented.

Even more generally speaking, organizing code together by mechanism or technology, rather than by “object” encapsulation, will let you apply all kinds of re-factorizations more easily, especially if you manage to decouple your application's data structures from any library-specific data structures.

Data Backends

Relational (SQL) databases such as MySQL were, for decades, the backbone of the web. They were originally developed as minimal alternatives to enterprise database servers such as Oracle Database and IBM's DB2. Their modest feature set allowed for better performance, smaller footprints, and low investment costs—perfect for web applications. The free software LAMP stack (Linux, Apache, MySQL and PHP) *was* the web.

Relational databases require a lot of synchronization overhead for clusters, limiting their scalability. Though partitioning can take you far, using a “NoSQL” database could take you even further.

Graph Databases

If your relational data structure contains arbitrary-depth relationships or many “generic” relationships forced into a relational model, then consider using a graph database instead. Not only will traversing your data be faster, but also the database structure will allow for more efficient performance. The implications for scalability can be dramatic.

Social networking applications are often used as examples of graph structures, but there are many others: forums with threaded and cross-referenced discussions, semantic knowledge bases, warehouse and parts management, music “genomes,” user-tagged media sharing sites, and many science and engineering applications.

Though fast, querying a complex graph can be difficult to prototype. Fortunately, the Gremlin and SPARQL languages do for graphs what SQL does for relational databases. Your query becomes coherent and portable.

A popular graph database is Neo4j, and it's especially easy to use with Prudence. Because it's JVM-based, you can access it internally from Prudence. It also has embedded bindings for many of Prudence's supported languages, and supports a network REST interface which you can easily access via Prudence's `document.external`.

Document Databases

If your data is composed mostly of “documents”—self-contained records with few relationships to other documents—then consider a document database.

Document databases allow for straightforward distribution and very fine-grained replication, requiring considerably less overhead than relational and graph databases. Document databases are as scalable as data storage gets: variants are used by all the super-massive Internet services.

The cost of this scalability is the loss of your ability to do relational queries of your data. Instead, you'll be using distributed map/reduce, or rely on an indexing service. These are powerful tools, but they do not match relational queries in sheer flexibility of complex queries. Implementing something as simple as a many-to-many

connection, the bread-and-butter of relational databases, is non-trivial in document databases. Where document databases shine is at listing, sorting and searching through very large catalogs of documents.

Candidate applications include online retail, blogs, wikis, archives, newspapers, contact lists, calendars, photo galleries, dating profiles. . . The list is actually quite long, making document databases very attractive for many products. But, it's important to always be aware of their limitations: for example, merely adding social networking capabilities to a dating site would require complex relations that might be better handled with a graph database. The combination of a document database with a graph database might, in fact, be enough to remove any benefit a relational database could bring.

A popular document database is MongoDB. Though document-based, it has a few basic relational features that might be just good enough for your needs. Another is CouchDB, which is a truly distributed database. With CouchDB it's trivial to replicate and synchronize data with clients' desktops or mobile devices, and to distribute it to partners. It also supports a REST interface which you can easily access via Prudence's `document.external`. And, both MongoDB and CouchDB use JavaScript extensively, making it natural to use with Prudence's JavaScript flavor.

We've started a project to better integrate Prudence JavaScript with MongoDB. It is included in the "Savory JavaScript" edition.

Column Databases

These can be considered as subsets of document databases. The "document," in this case, is required to have an especially simple, one-dimensional structure.

This requirement allows optimization for a truly massive scale.

Column databases occupy the "cloud" market niche: they allow companies like Google and Amazon to offer cheap database storage and services for third parties. See Google's Datastore (based on Bigtable) and Amazon's SimpleDB (based on Dynamo; actually, Dynamo is a "key/value" database, which is even more opaque than a column database).

Though you can run your own column database via open source projects like Cassandra (originally developed by/for Facebook), HBase and Redis, the document databases mentioned above offer richer document structures and more features. Consider column databases only if you need truly massive scale, or if you want to make use of the cheap storage offered by "cloud" vendors.

Best of All Worlds

Of course, consider that it's very possible to use both SQL and "NoSQL" (graph, document, column) databases together for different parts of your application. See [backend partitioning \(page 96\)](#).