

The Prudence Manual

Version 2.0-dev4

Main text written by Tal Liron

July 16, 2013

Copyright 2009-2013 by Three Crickets LLC.

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

Contents

Tutorial	4
Installing Prudence	4
At a Glance	4
Your First Application	4
Generating HTML	5
A Manual Resource	5
Managing the URI-space	5
Routing Paradigms	5
routing.js	6
app.routes	6
Injecting Conversation Attributes	9
Resource Mapping	11
URI Capturing	13
Resource Dispatching	13
The Internal URI-space	15
Virtual Hosts	15
app.hosts	17
Architecture Tips	17
Under the Hood	20
app.preheat	21
Implementing Resources	21
Comparison Table	21
Manual Resources	21
Scriptlet Resources	22
Static Resources	22
Integrating Java	22
Caching	22
Introduction: Integrated Caching	22
Server-Side Caching	22
Client-Side Caching	22
Programming	22
Introduction: Scripturian	22
JavaScript	22
Other Languages	22
Execution Environments	22
Managing State	23
APIs	26
Working in a Cluster	28
Shared Globals	28
Task Farming	28
Accepting Uploads	28
Cookies	28
Filtering	28
How Routing Works	28
Injection	28
Built-in Filters	28

Configuring Applications	28
settings.js	28
routing.js	28
Scheduling Tasks (Cron)	28
Default Directories	28
Configuring the Component	29
/component/servers/	29
/component/clients/	29
/component/hosts/	29
/component/services/	29
/component/templates/	30
Debugging	30
Logging	30
Debug Page	30
Deployment	30
The Joys of Sincerity	30
Deployment Strategies	30
Directory Organization	30
Operating System Service	30
Monitoring	30
Security	30
Proxying	31
Deploying Clusters	31
Cache Backends	31
Utilities for Restlet	31
Utility Restlets	31
Client Data	31
Redirection	32
Fallback Routing	32
Resolver Selection	32
Web Filters	32
Upgrading from Prudence 1.1	32
Upgrading Applications	33
Upgrading the Component	33

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

Managing 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”) error.

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. All matched URIs belong to your URI-space.

The URI-space is mostly configured in the application’s `routing.js`. However, the resource implementations can add their own special handling. For example, in the above example we can make sure in code that “{id}” would always be a decimal integer, thus limiting the extent of the URI-space. More generally, Prudence supports “wildcard” URI templates, allowing you to delegate the parsing of the URI entirely to your resource code.

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:** 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) **URI capturing:** 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 under Resource Mapping. This use case does not belong to the URI/resource separation paradigm.)
 - (b) **Resource dispatching:** 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 to whether you should organize your code around *resource mapping* or *URI/resource separation*. Generally, URI/resource separation is preferred for larger applications because it allows you full control over your code organization. However, it does add an extra layer of configuration and is not as transparent as resource mapping. It may make sense to use both paradigms. Read on, and make sure you understand how to use all three routing techniques.

routing.js

Before discussing the routing techniques, let's look at how routing is configured.

By convention, routing is configured in your application's routing.js file (which is executed by its default.js file). The file should configure at least app.hosts and app.routes, and app.dispatchers and app.preheat 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. It will be explained in detail in the rest of this chapter.

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

app.routes = {
    '/*': [
        'manual',
        'scriptlet',
        {type: 'cacheControl', mediaTypes: {'image/png': 'farFuture', 'image/jpeg': 'farFuture'},
         {type: 'javaScriptUnifyMinify', next:
             {type: 'zuss', next: [
                 'static',
                 {type: 'static', root: sincerity.container.getLibrary()
             ]}
        ],
        '/sample1/': '@sample', // (dispatched)
        '/sample2/': '/sample/' // (captured)
    ]
}

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

app.routes

Routes are configured in your application's routing.js, in the “app.routes” dict.

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 [TODO]. You can access the string values of there variables in your resource via the conversation.locals API.
- 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 [TODO] 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.

The possible values of the “app.routes” dict are *route type* 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 (note that the class names have the first character of the type capitalized). The values set in the 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.

Routing

dispatch 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 handle 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.

capture 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.

redirect 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.

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

hidden Use the “!” or “hidden” string values to hide a URI. 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. Furthermore, *you cannot hide URI templates*, only exact URI matches.

resource 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.

Mapping

static Use the “static” type to create a static resource handler. By default uses the application’s “/resources/” subdirectory for its “root”. Note that if you include it in a “chain” with “manual” and/or “scriptlet”, it should be the last entry in the chain. Handled by the Prudence.Routing.Static class.

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 Implementing Resources]

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 Implementing Resources]

Combining

chain 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

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. [See Filtering]

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. 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 TODO

Handled by the Prudence.Routing.HttpAuthenticator class.

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.

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 “/libraries/web/scripts/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Routing.JavaScriptUnifyMinify class.

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 “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Routing.CssUnifyMinify class.

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 “/libraries/web/style/” subdirectories. “next” is any route type configuration, allowing for nesting. Handled by the Prudence.Routing.Zuss class.

Custom Route Types

If you know how to use the Restlet library, then 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, and also inherit from `Prudence.Routing.Restlet`.

Here's complete example in which we implement a route type that redirects using HTTP status code 303 ("see other"). (Note this can also be 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/': {type: 'see', uri: 'http://newsite.org/new/uri/'},
}
```

Injecting Conversation Attributes

As we've seen in the `app.routes` section above, URI template variables delimited by curly brackets can be used to match incoming requests and extract the values into `conversation.locals`. For example, a `"/person/{id}/"` URI template will match the `"/person/linus/"` URI and set the `"id"` `conversation.local` to `"linus"`.

Template variables can furthermore be used to *inject* values in three use cases:

- Captured URI targets
- Redirection URI targets
- Cache keys (see caching)

In each of these cases you can inject the same `conversation.locals` that were extracted from the matched URI pattern. For example, you can redirect from `"/person/{id}/"` to `"http://newsite.org/profile/?id={id}"`.

Prudence furthermore supports many built-in variables extracted from the conversation attributes. A list of them is available in the Restlet API documentation, but we'll summarize them here in detail.

Request URIs

The variables are composed of a prefix and a suffix. The prefix specifies which URI you are referring to, while the suffix specified which part of that URI you need. For example, the prefix `"{r-}"` can be combined with the suffix `"{-i}"` for `"{ri}"`, which is the complete actual URI.

Prefixes

- `{r-}`: actual URI (reference)
- `{h-}`: virtual host URI
- `{o-}`: the application's oot 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 (identitifer)
- {-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 patterns, 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-}” suffix, which also function as a prefix to which you need to add one of the above suffixed. For example, “{rbi}” refers to the complete base URI of the actual URI.

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)

Entity Attributes

All these refer to the data (“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.

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 the PHP, ASP and JSP platforms 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. For example:

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

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

Capturing URI Templates

Later on we’ll present URI capturing as an alternative paradigm to resource mapping. However, capturing also has a specific use case in conjunction with resource mapping, which we’ll present here.

Specifically, URI templates can be captured to mapped resources. 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}/”. Furthermore, you 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 can also configure capturing and hiding separately. The following is equivalent to the above:

```
app.routes = {
    ...
    '/user/profile/{userId} /': '/user/profile /',
    '/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 via the `conversation.locals` API:

```
<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 is used in both cases.

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 (see above). For example:

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

The “{m}” variable is injected with the request method name, for example “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 kinds of requests.

You cannot use the “!” capture-and-hide trick with dynamic capturing, because Prudence can only hide exact matches of URIs. Use explicit hiding instead:

```
app.routes = {
    ...
    '/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 URI capturing or resource dispatching, documented below.

URI Capturing

URI capturing, for the purpose of the URI/resource separation paradigm, only makes sense for scriptlet resources. For manual resources, use resource dispatching (explained below) instead.

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

The effect is that you can put your scriptlet resources under “/libraries/scriptlet-resources/”, using any directory structure that makes sense to you, and capture the public-facing URI in your routing.js.

For example, let’s assume that you have the following files in “/libraries/scriptlet-resources/”: “/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:

```
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.

Resource Dispatching

Resource dispatching, for the purpose of the URI/resource separation paradigm, only makes sense for manual resources. For scriptlet resources, use URI capturing (explained above) instead.

Configuring a dispatch is straightforward. In routing.js, use the “dispatch” route type configuration, or the “@” shortcut:

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

The IDs must each be unique in your application. You must furthermore configure your dispatchers. There is one dispatcher per programming language, and JavaScript is the default dispatcher. We can configure it like so:

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

The “/manual-resources/” value is the document name to be executed from your application’s “/libraries/” subdirectory. In our case, we must thus also have a “/libraries/manual-resources.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 manual-resources.js file does not refer to URIs, but instead to dispatch IDs, which you can dispatch as you see fit.

Inversion of Control

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

When defining the dispatch, you can also set conversation.locals to set values. You would need to use the long-form configuration for this:

```

app.routes = {
    ...
    '/user/{userId}/preferences': {type: 'dispatch', id: 'user', locals: {section: 'preferences'}},
    '/user/{userId}/profile': {type: 'dispatch', id: 'user', locals: {section: 'profile'}}
}

```

Note that both URI templates are dispatched to the exact same ID, but the “locals” dict used is different for each. In your resource implementation, you can allow for different behavior according to the value of the “section” conversation.local. This allows you to configure your resource in routing.js, rather from its implementation in resource.js. In other words, “control” is “inverted,” via value injection.

Other Programming Languages

Resource dispatching is also supported for Groovy, Python, Ruby, PHP 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/'
}

```

Custom Dispatching

Under the hood resource dispatching is handled by URI capturing. The URI is captured to a special manual resource called a “dispatcher” with an injected value specifying to which resource it should dispatch.

Prudence comes with a few dispatchers to 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.

However, you can also write your own dispatchers to handle more complex dispatching paradigms. An example configuration of overriding the default “javascript” dispatcher:

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

You would then need a “/libraries/dispatchers/mydispatcher.js” file under your application’s subdirectory. You would be able to access the dispatch ID as the “prudence.dispatcher.id” conversation.local.

Note that you do not need to override the default dispatchers, and can use any dispatcher name:

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

“customValue1” and “customValue2” would then be available to your dispatcher code as the “prudence.dispatcher.special.customValue1” and “prudence.dispatcher.special.customValue2” application.globals respectively (replace “special” in the name with your dispatcher name). Thus, you can configure your dispatcher in routing.js.

The Internal URI-space

TODO

Virtual Hosts

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

Advantages of using a single container

1. Possibly simpler deployment: a single base directory, and all configuration is done by JavaScript inside the container, so it can be fully dynamic.
2. Less memory use.
3. You can use application.sharedGlobals to share state between applications. (Note that if they are running in multiple containers, you can use application.distributedGlobals instead. Will also work on VMs running on separate machines in the cluster. But distributed globals have to be serializable, of course.)

Advantages of using multiple containers

1. Possibly simpler deployment: several base directories can mean separate code/distribution repositories, and configuration may happen at the reverse-proxy level with your web frontend.
2. Crashes/deadlocks/memory leaks in one VM won’t affect others.
3. You can restart the VM for one container without affecting others that are running.

There is no performance advantage in either scenario. Everything is designed around high-concurrency and threading, and threads are managed by the OS globally. (Well, there are some caveats to this: Linux can group threads per running process for purposes of prioritization, but this is only really done for desktop applications. But this 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 deployments, where CPU usage is often high.)

Here is documentation on managing routing in a single container. Note that the Prudence Administration app shows all of this routing information at runtime:

Servers

Under `/component/servers/` you can define as many servers as you want. Just add a new file and assign a port for it. There is actually routing done per server, too, so you can define which IP address you want the server to bind to, in case your machine as several IP addresses. On AWS, this could be useful because machines can indeed have both a public and “internal” IP address.

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

Routing

`server.address = [string]`
Defaults to null (which defaults to localhost), but you can filter on a specific IP address. “*” wildcards are supported.

(This is how you would add a TLS `https://` server, too, if you want it managed by Prudence. Though another option is to use a frontend, like Apache/nginx, with a reverse proxy. Which one is better would be another discussion.)

Hosts

Under `/component/hosts/` you can add as many virtual hosts as you want. A virtual host signifies the basic domain name and port assignment.

```
var host = new VirtualHost(component.context)
host.name = privatehost [used when attaching applications to the host]
component.hosts.add(component.defaultHost)
```

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

By default all of these are null, meaning that all incoming requests are routed, but you can filter on specific incoming requests. For example, you might want the virtual host to only accept a specific port or server IP address. “*” wildcards are supported for all of these.

Applications

In each application’s `routing.js` you have `app.hosts` by which you assign the application to a base URI on that host. By default we only use a single host, the default one, but you can assign each app to one or more hosts, with different base URIs under each. The app instance only runs **once** in such cases: all `app.globals`, etc., are single. It’s only the routing that changes according to the requests coming in from various servers and virtual hosts. If you’d like the application to behave differently for each host, you can do that in your code.


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

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

```
app.hosts = {
    'default': '/publicbaseuri/',
    'privatehost': '/privatebaseuri'
}
```

app.hosts

TODO

Architecture Tips

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.

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.

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. [SO?]

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 will 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 entanglements.

Note: 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 not make much sense to apply in 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 robust and powerful support for JSON-RPC, XML-RPC and ExtDirect, 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, an an RPC-based solution may prove a better match for your project.

It’s also perfectly possible to allow for both REST and RPC. 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.

Note that 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

TODO

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. Until it is available, consider adopting a resource documentation standard for your project.

```
/*
 * 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}/
 * Aspects: /service/{id:decimal}/status/
 * Verbs: GET, POST, PUT, DELETE
 * Media types: application/json, application/xml, text/plain (as JSON)
 * Query params:
 *   indent: decimal — if non-zero will return a human-readable indented version
 *                   of the representation with lines indented by the integer value
 *   clean: boolean — if "true" or "yes" or "1" will force removal of an existing
 *                  service during a PUT operation on an existing service
 *
 * Representation as application/json:
 * {
 *   "id": number,
 *   "name": string (the service name),
 *   "status": string:"active"|"inactive"
 * }
 *
 * POST/PUT payload as 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/
* Aspect of: /service/{id:decimal}/
* Verbs: GET, POST, PUT, DELETE
* Media types: text/plain
*
* Representation as text/plain:
*   "active"|"inactive"
*/

```

Under the Hood

In this final section, we'll describe in detail how routing works in Prudence. It can be considered optional reading for advanced developers.

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.

Servers are configured in “/component/servers/”.

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.

Virtual hosts are configured in “/component/hosts/”.

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.

app.preheat

Implementing Resources

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.*	n/a
<i>Programming Languages</i>	Determined by filename extension	Determined by scriptlet tags (multiple languages possible per resource)	n/a
<i>Content Negotiation</i>	Manually determined in handleInit; multiple MIME types possible	Single MIME type determined by filename extension; multiple encodings automatically supported and cached	Single MIME type determined by filename extension; multiple encodings 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

Manual Resources

Mapping vs. Dispatching

handleGetInfo

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

Scriptlet Resources

Must be mapped. In the future may be dispatched.

Scriptlets

Working with Different Programming Languages

Controlling the Format

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

Server-Side Caching

Client-Side Caching

Scriptlet Plugins

Static Resources

Must be mapped.

Controlling the Format

Client-Side Caching

CSS and JavaScript

ZUSS

Integrating Java

Resources

Other Restlets

Caching

Introduction: Integrated Caching

Server-Side Caching

Client-Side Caching

Content Negotiation

Programming

Introduction: Scripturian

JavaScript

Other Languages

Execution Environments

Bootstrap

Straightforward beginning-to-end script

Except for initialization tasks

Manual Resources and Handlers

Textual Resources

Cron Tasks

Two options!

Managing 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.

`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` (page ??) 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 ??), `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 ??).

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

The rule of thumb is to always prefer to use `application.globals` (page ??). 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 name space, 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 more 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, Groovy and Clojure. Indeed, the entire JVM standard APIs can be accessed 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/');
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`.

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

Working in a Cluster

TODO

Shared Globals

Task Farming

Accepting Uploads

TODO

Cookies

TODO

Filtering

TODO

How Routing Works

Injection

Built-in Filters

Configuring Applications

settings.js

app.settings

app.settings.description

app.settings.errors

app.settings.code

app.settings.uploads

app.settings.mediaTypes

app.globals

routing.js

app.hosts

app.routes

app.errors

app.dispatchers

Scheduling Tasks (Cron)

Default Directories

/resources/
/libraries/

/libraries/
/uploads/

Configuring the Component

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

/component/servers/

/component/clients/

/component/hosts/

See virtual hosts in managing URI space.

/component/services/

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

caching

Configure the caching backend

distributed

Load the Hazelcast configuration

executor

Configures thread pools for task execution.

log

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

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.

scheduler

Configure the cron scheduler (cron4j)

status

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

version

Provides access to Prudence and Restlet versions.

[/component/templates/](#)

Debugging

Logging

`application.logger`

`application.getSubLogger`

Configuring Logging

[/configuration/logging/](#)

See Sincerity Manual

Debug Page

Deployment

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

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.

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 Prudence Manual to find equivalences.
3. Copy over individual settings from routing.js, using the new Prudence Manual to find equivalences. Prudence 2.0 has 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/scriptlet-resources”.

Upgrading the Component