Sapphire Framework

Programmer’s Reference

Version

[Architecture 1](#_Toc378167128)

[Introduction 2](#_Toc378167129)

[Goals 2](#_Toc378167130)

[Terminology 2](#_Toc378167131)

[Structure of an Application 4](#_Toc378167132)

[Best Practices 5](#_Toc378167133)

[Separate Design and Code 5](#_Toc378167134)

[Model/View/Controller 6](#_Toc378167135)

[Server API 7](#_Toc378167136)

[Introduction 8](#_Toc378167137)

[Routing 9](#_Toc378167138)

[Application Routing 9](#_Toc378167139)

[Static file Routing 9](#_Toc378167140)

[Services Routing 9](#_Toc378167141)

[Application Class 11](#_Toc378167142)

[initialize 11](#_Toc378167143)

[addState 12](#_Toc378167144)

[addVariable 12](#_Toc378167145)

[addUrl 12](#_Toc378167146)

[addConfig 12](#_Toc378167147)

[setBody 13](#_Toc378167148)

[addTemplates 13](#_Toc378167149)

[addFileReplacement 13](#_Toc378167150)

[addStringReplacement 13](#_Toc378167151)

[addJS 14](#_Toc378167152)

[addCSS 14](#_Toc378167153)

[addPage 14](#_Toc378167154)

[addDialog 15](#_Toc378167155)

[setTitle 15](#_Toc378167156)

[addMetadata 15](#_Toc378167157)

[setFavicon 15](#_Toc378167158)

[getHTML 16](#_Toc378167159)

[Example 17](#_Toc378167160)

[Services 18](#_Toc378167161)

[Standard Responses 18](#_Toc378167162)

[Directory Organization 19](#_Toc378167163)

[Client API 20](#_Toc378167164)

[Introduction 21](#_Toc378167165)

[Object 22](#_Toc378167166)

[function 22](#_Toc378167167)

Section

1

# Architecture

## Introduction

Sapphire is specifically designed for the creation of Single Page Applications (SPA). These applications have a number of special considerations over more traditional web applications. These include hot loading of parts of the application when they are needed, construction of the application from multiple sources, AJAX service function to deliver data and perform functions and a front end API that ties it all together.

Sapphire is built in node.js and makes use of a couple of libraries. These include MooTools for class creation and Q for promises.

The framework is object oriented, on both the server side and the client side. The client side framework uses an Model/View/Controller (MVC) paradigm. The server is broken into three major areas of functionality.

1. Building the application to send to the client
2. Writing AJAX service functions
3. Delivering assets such as images, JavaScript and CSS files.

### Goals

This framework was created with a number of goals in mind, these are list below:

|  |  |
| --- | --- |
| Separation of Skills | There should be a clean divide between application code and the visual design. Also, the JavaScript coding should be easily separable from the servers-de code as staff is frequently divided by these different skills. |
| Locality | Code is organized so that files that support functionality are located close together. |
| Others |  |

### Terminology

The following terms describe concepts within the framework.

*Page* A block of html and corresponding client-side programming logic within which application features are presented. All pages occupy the same visual space within an application, and can be switched in and out according to application logic. The client-side framework has methods to manage pages. Pages are hot-loaded when first displayed.

*Dialog* Like a page, it is a block of html and programming within which application features are presented. Dialogs appear on top of pages, and can be stacked one on top of another. The client-side framework has methods to manage dialogs. Dialogs are hot-loaded when first displayed.

*Templates* These are reusable blocks of html, typically used as partials.

*Modules* A feature that is potentially reusable. For example, a common header used between different applications might be written as a module.

*Pruning* Pruning is the process of removing a page from the DOM when it is not in use. This is usually needed for pages that contain flash objects, since in many browsers, the flash object will reload when added back to the DOM.

*Cache Busting* Assuring that changed assets will not be in the user’s browser cache.

## Structure of an Application

There are two ways to refer to the structure of the application. There is how it exists on the server, and how it exists once it has been delivered to the browser. On the server is a framework to assemble the various pieces of an application and deliver them to the browser. On the front end is a set of libraries that are used to implement the application from the assembled pieces.

Assembling the application is done using the Application object in the node.js server code. This is not to be confused with the Application object running on the client. Use the Application object to specify the various pieces of the application such as the body HTML, templates, pages, dialogs, JavaScript and CSS. Once the specification of the application is complete, Application will construct the HTML to be sent to the server.

## Best Practices

### Separate Design and Code

It is a good idea to separate design and engineering efforts. Engineers should not be creating markup and style sheets, and designers should never have to modify JavaScript. Mixing JavaScript directly with design will make both tasks much harder.

To do this, engineers and designers create a contract about the id’s of the nodes the JavaScript will need to manipulate, and the JavaScript engineers need never touch the page templates or the css. For instance the following is a test page template.

<div id=”test-page”>

<h1>Test Page</h1>

<div id=”test-page-name”></div>

<img id=”test-page-image” />

<div id=”test-page-message”>Here is a list of stuff</div>

<div id=”test-page-container”></div>

</div>

In this example, the designer and the JavaScript engineer will have agreed to the names of various nodes, for example “test-page-name” and “test-page-image”. Presumably when this page is displayed, the JavaScript code will fill these nodes with relevant content.

Often, when a node needs to be repeated multiple times for a list of items, the JavaScript coder will simply construct the relevant nodes in code and then add them to the DOM for each repeated item. This, however, makes it more difficult for a designer to control the presentation of these items.

As an alternative, the designer should be able to create an html block that represents the repeated item, which he is free to create and maintain as part of the designing process, without interaction with the engineer. This is called a template. Add the CSS class “template” to all templates so that the client framework can manage them.

The engineer and designer can agree on the name of this node, and the engineer can create JavaScript to use the template for each item in the list. As with everything else, nodes within this template will have agreed upon names so that the engineer can update the information for each item. The JavaScript library has functions that facilitate this behavior.

For out example, the template node might look like this:

<div id=”list-item-template” class=”list-item-template template”>

<div id=”list-item-img” />

<div id=”list-item-description”></div>

</div>

And when it is time to add items to the list would do the following:

var container = $('#test-page-container');

items.each(function (item)

{

var template =Sapphire.templates.get(.clone(true, true);

template.find('#list-item-img').attr('src', item.url);

template.find('#list-item-descrption').html(item.description);

container.append(template);

}, this);

The section on the client API will explain the objects and methods used in this example more fully.

### Model/View/Controller

The model/view/controller (MVC) paradigm works well with single page applications, and is recommended. In traditional web applications, the MVC lives on the server, and is responsible for routing control from page to page based on user actions. In a single page application all the application logic lives on the client, so the MVC is moved there. The server code becomes much simpler.

**Model**

Sends and receives events to the application server . Maintains application state

**Controller**

Manages the flow of control based upon user actions and application state

**View**

Presents the view of the current application state, based on commands from the controller

AJAX Services

HTML/CSS

A model is responsible for saving and retrieving a type of data, a controller manages the application flow, and a view updates and monitors the user interface. A view has knowledge of the html layout, a model understands the server-side interface and a controller knows about both a model and a view.

Communication from a view to a controller and from a model to a controller should be done via events and callback functions, not via direct calls into a controller.

Section

2

# Server API

## Introduction

Configuration files

## Routing

The Sapphire server does things other than just routing control to your application. It also serves static files, routes services, and manages cookies and sessions.

### Application Routing

Applications can have sub-applications, for example, the application horizon could have two sub-applications, manage and operate, accessed through the urls “/horizon/manage” and “horizon/operate”. All applications live under the apps directory in the Sapphire root directory. “apps/horizon/operate/operate.js” would be the entry point for the “horizon/operate” application. All application files export a single function called buildApplication which takes three parameters, the request, the response and a callback. The callback should be invoked with the HTML to send to the browser. This is usually built using the Application call documented later in this chapter.

Before the buildApplication function is called, the request object is loaded with a session object and a cookies object. Any changes to the session object will be saved when the application is finished. This is also true of the cookies object. Cookies can be added to the cookies object under the request object, but should be added to the cookies object in the response object.

### Static file Routing

There are four types of static files.

* Global static files. These are the files available to all applications. They are in the /public/assets directory off the sapphire root.
* Local static files. These are files specific to an application or module. They are located under the assets folder for the module or application.
* Page files. These are the page template files; they will be served from pages directory under the application or module directory.
* Dialog files. These are the dialog template files; they will be served from the dialogs directory under the application or module directory.

### Services Routing

Service URLs take the following form:

<app>/services/<service>/[...objects]/<method>

These are the pieces of the URL

app The name of the application that implemented the service

service The name of the service being called

objects a nested list of objects, for example, "building/resources"

method the specific service method being called

#### Example

/horizon/services/system/health

To find the service code, the service router looks for a directory named services in the application directory, and within that directory tries to load <service>.js. Each service must export a function or object that corresponds to the first part of the service path. The router will then attempt to drill down into this service object to find the objects specified. For instance, if your service was named ‘account’, and you had an object named 'settings' and a method named 'set', it would look for the presence of 'settings' within the service object. The last part of the route is assumed to be the method name. The router will verify that this is a function, and then call it, passing the request, the response and the post data.

## Application Class

Use the Application class to specify all of the pieces of your application and assemble them into the HTML that will be delivered to the browser. The overall structure of the HTML document is defined in a file named master.html. This file is a simple outline of the document with placeholders for where various parts of the application will be located. This is the entire file.

<html>

<head>

{favicon}

{title}

{metadata}

{masterCSS}

</head>

<body class="{states}">

{body}

{masterJS}

{javaScript}

</body>

</html>

There are a number of keywords inside curly braces in this file, and these are used to mark where specific application pieces will be located. These are.

|  |  |
| --- | --- |
| favicon | The metadata tag to specify the favicon goes here. |
| title | The title tag goes here. |
| metadata | All the specified metadata tags go here |
| masterCSS | The list of added CSS files go here |
| states | States are CSS classes that define the initial state of the application. They go here. |
| body | The body is the HTML that specifies the overall chrome for the application. This is stuff outside of the pages. It goes here. |
| masterJS | All the specified JavaScript files go here |
| javaScript | The generated JavaScript goes here. |

### initialize

initialize : function(ns)

This is the constructor for the application object. It is automatically called when you create the application object.

#### Parameters

ns This is the namespace for the application variables.

### addState

addState : function(state)

Call this method to add a state class to the body tag of the HTML file. States are used to setup initial presentation states for the application. For example, a ‘login’ state could configure the application header to display the login information, rather than a login button.

#### Parameters

state The name of the state

### addVariable

addVariable : function(name, value)

Call this function to add a variable to the application. Variables will be added to the name space specified when the Application was constructed.

#### Parameters

name The name of the variable.

value The value of the variable. This should be the native type, not a JSON string.

### addUrl

addUrl : function(name, value)

Call this method to add a URL to the list of managed URLs. URLs will be available in <namespace>.urls.<name>

#### Parameters

name The name of the url

value The actual URL itself

### addConfig

addConfig : function(name, value)

Call this method to add a configuration variable. Configuration variables will appear before the JavaScript files.

#### Parameters

name The name of the variable

value The value of the variable

### setBody

setBody : function(file)

Call this method to set the body file for the application. The body defines the overall chrome of the application. It should contain an element with the id “pages” which specifies where the in the document pages are placed. It should also contain an element named “dialogs” for the dialogs.

#### Parameters

file The path to the file, relative to the root of the application.

### addTemplates

addTemplates : function(file)

Call this method to add templates to the HTML file. The HTML file specified will be added to the output immediately after the body.

#### Parameters

file The path to the file, relative to the root of the application.

### addFileReplacement

addFileReplacement : function(name, file)

Call this method to add a file replacement. In addition to the curly braced keywords in the master.html file, user defined replacements can exist in either this file or in the body file. This function will replace the replacement with the passed name, with the contents of the file specified.

#### Parameters

name The name of the replacement

file The location of the file, relative to the sapphire root.

### addStringReplacement

addStringReplacement : function(name, value)

Call this method to add a string replacement. In addition to the curly braced keywords in the master.html file, user defined replacements can exist in either this file or in the body file. This function will replace the replacement with the passed name, with the passed value.

#### Parameters

name The name of the replacement

value The replacement string

### addJS

addJS : function(files, dontBust)

Call this method to add JavaScript files to the application.

#### Parameters

files The array of JavaScript files. These should be relative to either the apps root, or /public for global files.

dontBust Set this value to true if the files should not be cache busted. Usually JavaScript files are cached busted. When the file is cache busted, the md5 hash of the contents will be added to the filename to force the file to be retrieved from the server whenever it has changed.

### addCSS

addCSS : function(files, dontBust)

Call this method to add CSS files to the application.

#### Parameters

files The array of CSS files. These should be relative to either the apps root, or /public for global files.

dontBust Set this value to true if the files should not be cache busted. Usually CSS files are cached busted. When the file is cache busted, the md5 hash of the contents will be added to the filename to force the file to be retrieved from the server whenever it has changed.

### addPage

addPage : function(spec)

Call this method to add a page to the application. Pages will be loaded on demand. The spec is an object with a number of fields specifying the details for this page.

|  |  |
| --- | --- |
| name | This is the name of the page as it will be referenced in the client. |
| url | This is the path to the page template file. This is an HTML file. |
| javascript | This is an array of JavaScript files that will be loaded the first time a page is shown. |
| css | This is an array of css files that will be loaded the first time a page is shown. |
| dontPrune | Set this to true if the page should not be pruned. When a page is not pruned it will remain in the DOM even when it is not shown. |

#### Parameters

spec The specification for the page.

### addDialog

addDialog : function(spec)

Call this method to add a dialog to the application. Dialogs are conceptually very similar to pages, but function differently. Dialogs will be loaded on demand. The spec is an object with a number of fields specifying the details for this dialog.

|  |  |
| --- | --- |
| name | This is the name of the dialog as it will be referenced in the client. |
| url | This is the path to the dialog template file. This is an HTML file. |
| javascript | This is an array of JavaScript files that will be loaded the first time a dialog is shown. |
| Css | This is an array of css files that will be loaded the first time a dialog is shown. |

#### Parameters

spec The specification for the dialog.

### setTitle

setTitle : function(title)

Call this method to set the title for the HTML document.

#### Parameters

title The title

### addMetadata

addMetadata : function(name, content)

Call this method to add metadata tags to the output HTML.

#### Parameters

name The name of the metadata

content The content of the metadata

### setFavicon

setFavicon : function(url)

Call this method to set a favicon metadata tag to the HTML.

#### Parameters

url The url of the favicon

### getHTML

getHTML : function(callback)

Call this method to get the HTML for the application. Because this process may have to wait for files to be loaded and processed, this is an asynchronous operation. The callback will be invoked with the contents of the HTML when this is complete. The HTML is ready to be sent to the browser.

#### Parameters

callback The function to call with the resulting HTML.

## Example

Here is example code for an application called destination/home. This application uses two modules—account and header--and Q for promises.

var application = require('application.js');

var account = require('../modules/account/account.js');

var header = require('../modules/header/header.js');

exports.buildApplication = function(req, res, callback)

{

var session = req.session.get();

var app = new application.SpaBuilder('DESTINATION');

app.setTitle('home');

app.setBody('apps/destination/home/templates/body.html');

app.addCSS([

'/destination/assets/css/common.css',

'/destination/assets/css/fonts.css'

]);

app.addState('no-dialog');

app.addJS([

'/assets/js/lib/templates.js',

'/assets/js/lib/ajax-service.js',

'/assets/js/lib/translate.js',

'/destination/assets/js/service.js',

'/destination/assets/js/Controllers/Canvas.js',

'/destination/assets/js/Views/Canvas.js'

]);

app.addDialog({

name: 'signup',

url: '/destination/modules/header/dialogs/signup.html',

javascript: [

'/destination/assets/js/Views/Dialog.js',

'/destination/assets/js/Controllers/Dialog.js',

'/destination/modules/header/assets/js/Views/Signup.js',

'/destination/modules/header/assets/js/Controllers/signup.js'

],

css: [

'/destination/assets/css/dialogs.css',

‘/destination/modules/header/assets/css/signup.css'

]

});

var promise = account(req, res, builder)

.then(header.bind(this, req, res))

.then(function(builder)

{

app.getHTML(callback);

}).done();

}

## Services

Unlike building an application, which relies on a callback to provide the result, service functions must return a Q promise that will be fulfilled when the service function is complete. This makes it easier for the service router to capture errors and return a properly formatted response. Server responses are assumed to be in JSON.

Here is an example services, this would be called by posting to the url:

/sample/services/user/login

exports.login = function(req, res)

{

var user = new User();

var session = req.session.get();

var email = req.body.email;

var password = req.body.password;

return user.login(email, password)

.then(function(user)

{

if (user === false)

return {success: false, message : 'invalid login'};

else

{

res.cookies.set('identity',

encryptCookie(user.user.emailAddress, user.user.password));

session.user = user.user;

return {success: true, result: user.getIdentity(), identity: user.getIdentity()};

}

}.bind(this));

}

### Standard Responses

Sapphire applications are written with a standard response format in mind. Responses are in JSON, with the top level items being

success Will be true or false if the function succeeded. Results that return an empty set should be considered successful. Only problems with the request like missing data, or database errors, should be considered failures.

result This is the result of the service call. It can be any data type.

In addition to these two standard members, others can be added to the top level of the response. These can be intercepted in the client to look for global level changes, like the user identity changing.

## Directory Organization

The directory structure for sapphire is organized like this.

* **node\_modules/** contains all the required packages as well as the sapphire specific middleware and classes.
* **node\_module/config** contains the environment configuration files.
* **public/assets** the standard assets available for all sapphire applications.
* **apps/**applications go here.

Each application also has a directory structure

* **assets/** static assets for your application
* <sub application directories>
* **modules/** modules go here
* **services/** services go here
* **node\_modules/** needed packages go here
* **pages/** html templates for page
* **dialogs/** html templates for dialogs
* **templates/** htmltemplate files

The directory structure for modules is similar to applications

* **assets/** static assets for your application
* **pages/** html templates for page
* **dialogs/** html templates for dialogs
* **templates/** htmltemplate files

Section

3

# Client API

## Introduction

## Object

### function

prototype

#### Parameters

name the name of the dialog to display.

#### Returns

Returns the name of the page.