Sapphire Framework

Programmer’s Reference

Version

[Architecture 1](#_Toc378083080)

[Introduction 2](#_Toc378083081)

[Goals 2](#_Toc378083082)

[Terminology 2](#_Toc378083083)

[Structure of an Application 4](#_Toc378083084)

[Best Practices 5](#_Toc378083085)

[Separate Design and Code 5](#_Toc378083086)

[Model/View/Controller 6](#_Toc378083087)

[Server API 7](#_Toc378083088)

[Routing 8](#_Toc378083089)

[Features 9](#_Toc378083090)

[Application Class 10](#_Toc378083091)

[initialize 10](#_Toc378083092)

[addState 11](#_Toc378083093)

[addVariable 11](#_Toc378083094)

[addUrl 11](#_Toc378083095)

[addConfig 11](#_Toc378083096)

[setBody 12](#_Toc378083097)

[addTemplates 12](#_Toc378083098)

[addFileReplacement 12](#_Toc378083099)

[addStringReplacement 12](#_Toc378083100)

[addJS 12](#_Toc378083101)

[addCSS 12](#_Toc378083102)

[addPage 12](#_Toc378083103)

[addDialog 13](#_Toc378083104)

[setTitle 13](#_Toc378083105)

[addMetadata 14](#_Toc378083106)

[setFavicon 14](#_Toc378083107)

[Session Class 15](#_Toc378083108)

[Examples 16](#_Toc378083109)

[Folder Organization 17](#_Toc378083110)

[Firing It Up 18](#_Toc378083111)

[Client API 19](#_Toc378083112)

[Introduction 20](#_Toc378083113)

[Object 21](#_Toc378083114)

[function 21](#_Toc378083115)

[Services 22](#_Toc378083116)

[Introduction 23](#_Toc378083117)

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.

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

## 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 these applications, and is the recommended paradigm. The model is responsible for tracking application state, the controller manages the flow of control between pages based on changes in the application state, and the view updates pages and captures user feedback to send back to the controller for action. The view has knowledge of the html layout, the model understands the back-end interface and the controller knows about both the model and the view.

**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

Client Services

HTML/CSS

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

Section

2

# Server API

## Routing

## Features

Lazy loading

Cache busting

## 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

### addStringReplacement

### addJS

### addCSS

### 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

## Session Class

## Examples

This section contains a few examples that demonstrate how this stuff is all tied together.

## Folder Organization

The folders for the platform are arranged as follows, the root is the location of the index.php file.

* **cfg/** is the location of server wide configuration files.
* **classes/** is the location of general classes used by games, platforms and extensions. For example the application class is here.
* **extensions/** is the location of extensions. Each extension is in a folder named after the extension.
* **games/** is the location of all games. Each game lives in its own folder named after the game.
  + **<game>/** The main game class lives here, as well as the platform specific game classes and usually the body.php file. Each game has its own folder structure:
    - **cfg/** is the location of game specific configuration files, for example the facebook api keys.
    - **classes/** is the location of any game specific php classes. An example would be the player class, which controls all interactions with the player.
    - **data/** is the location of any data files. For example, the list of what level is reached at what experience points.
    - **dialogs/** is the location of all the dialog templates.
    - **images/** is the location of all game specific images.
    - **js/** is the location of all the javascript files used by the game.
    - **pages/** is the location of all the page templates.
    - **services/** is the location of all Ajax services.
* **images/** is the location of any image files that will be shared across multiple games.
* **js/** contains the library javascript files included in all games
  + 3rdpart/ contains javascript files not written by fundly
  + app/ contains javascript files that are core to application management
  + lib/ contains other javascript libraries needed for proper game operation.
* **platforms/** is the location of all platforms. Each platform lives in a directory named after the platform.

## Firing It Up

Section

3

# Client API

## Introduction

There are three main objects within the framework, these are APPLICATION, UTILS and UI. The application object is responsible for page and dialog management, as well as containing a number of variables needed for basic game operation, for example the device and device-user id’s.

In addition there are a number of other classes, such as the Events class, available for applications to use.

The library uses mootools as its primary JavaScript library for dom manipulation. There are a number of methods in this library that assume the presence of mootools. Mootools was chosen because it is very compact and fits well on resource constrained devices.

## Object

### function

prototype

#### Parameters

name the name of the dialog to display.

#### Returns

Returns the name of the page.

# Services

## Introduction

The framework includes a mechanism for consistent handling of services. This is not a mandatory part of using the framework but is certainly a useful feature. In the root of the framework folder is a file service.php that implements this service framework. Calls to this php file will be routed to the correct php representing the service.

This file looks for one query string or post variable named cmd. The value of this will be separated into three pieces, separated by the pipe character, ‘|’. The first part of this string is the name of the game, the second is an application defined object or service, and the third is a specific method within that service.

The service handler will load the php file located in path “games/<gamename>/services/<object>.php”. In this case object represents the second part of the command string.

It will then construct the class “<object>Service”, where the object name has its first character upper case, for example, “PlayerService”. Then it will call the method “call” within this object passing it the third part of the command string and the $\_REQUEST global variable.