Sapphire Framework

Programmer's Reference

Version 0.1

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, for example, hot loading of parts of the application when they are needed, construction of the application from multiple sources, AJAX service functions to perform backend actions and retrieve updated data, and a front end API that ties it all together.

Sapphire is built in node.js and makes use of a number of 3rd party modules, such as Q for promises, MooTools for more traditional class support, JQuery for DOM Manipulation and the whole thing is built on top of express, giving access to lots of useful middleware. However, routing in sapphire is done very differently than for a typical express application.

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

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

Sapphire is a robust framework and handles a number of high level functions right out of the box.

- Session management using redis, memcache or cookies. Applications can also define their own session storage mechanism.
- Hot loading of resources only when needed
- Cache busting
- Localization
- Minification of JavaScript and CSS.

These will all be discussed in more detail later.

Goals

This framework was created with a number of goals in mind:

Separation of Skills	There should be a clean divide between application code and the visual design. The framework is designed to allow visual and UX designers who can create HTML and CSS to be active, productive members of the team, freeing up programmers to concentrate on application logic, not pixel pushing. There is also an easy transition for some of this staff to participate in coding if that is their career trajectory. Also, the browser code should be easily separable from the server code as staff is frequently divided by these different skills.

Locality	Code is organized so that files that support functionality are located close together. This makes working on different aspects of an application easier, and helps reduce dependencies.
MVC on Client	Single page applications have the bulk of the user interaction in within the browser; the majority of the application code will be there. So, in Sapphire, that's where MVC is implemented. The paradigm of the back end is centered around its specific needs.

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. Typically, pages are sandwiched between a static header and footer. The browser framework has methods to manage pages. All the assets for a page are hot-loaded when first used.
Dialog	Like a page, a dialog is a block of html and JavaScript within which application features are presented. Dialogs are modal bits of user interface that temporarily take over all user interaction, for example to display a message, or prompt the user to login. The browser framework has methods to manage dialogs. All the assets for a dialog are hot-loaded when first used.
Panel	This is similar to a page, in that a region of the screen can be reserved to swap in and out different bits of functionality. However, panels can appear anywhere, typically inside of a page. The browser framework has methods to manage panels. Panels are hot-loaded when first used.
Templates	These are reusable blocks of html, typically used as partials.
Feature	A self-contained set of functionality that is potentially reusable. For example, a common header used between different applications might be written as a feature. Features allow you to create a large set of functionality where all the assets are local to the feature itself. Features are also a useful way to manage a complex application.
Pruning	Pruning is the process of removing a page, panel or dialog from the DOM when it is not in use. However, some pages should not be pruned, and this can be specified when describing the page. This is frequently 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.

3rd Party Libraries

MooTools

Mootools is a JavaScript library that works on both the client and server. MooTools is different from a number of libraries, such as jquery and underscore, that namespace all of its APIs, in that it extends native types as well as creating new types of its own.

For example, MooTools has a method to test the presence of an item in an array. So, rather than having something like `Mootools.arrayContains(a, v)`, the method is implemented in the Array prototype, and is directly available on the array itself.

```
['a', 'b', 'c'].contains('c'); // returns true
['a', 'b', 'c'].contains('d'); // returns false
```

However, the primary reason for using MooTools is to get access to its class facilities. It can not only create new classes, but it supports a number of other features such inheritance, class reopening, mixins and monkey patching.

Inheritance

To inherit from a class, use the member Extends when creating a new class. This has to appear as the first member in the new class. For example:

```
var MyClass = new Class({
    Extends : BaseClass
});
```

To access your parent's version of a method, call `this.parent()`;

Class Reopen

To reopen a class, use the implement method on the Class to be reopened. For example:

```
MyClass.implement({
    newMethod : function()
    {
        .    .    .
    }
});
```

Mixins

With Mootools you can create classes whose sole purpose is to have its methods merged into another class. In some languages this is known as composition, or mixins. To do this, use the member Implements when defining your class. It should follow Extends if you are also inheriting from a base class. For example:

```
Package('MahJongg', {
    Service : new Class({
        Extends : Sapphire.Eventer,
        Implements: [Sapphire.Services.AjaxService],

        .    .    .

    })
});
```

Monkey Patching

Monkey patching is changing the defined methods of an existing class at runtime. This does not create a new class; it changes an existing class by overriding its existing methods to do something new. To do this, use Class.refactor. For example:

```
var Cat = new Class({
    energy: 0,
        eat: function(){
            this.energy++;
    }
});

Class.refactor(Cat, {
    eat: function(){
        this.previous(); //energy++!
        alert("this cat has " + this.energy + " energy");
    }
});
```

Notice that to access the original implementation of a method, use `this.previous()`.

Q

Q is a promises library. It is used in a number of places on both the server and the client side code. The API for this module can be found here https://github.com/kriskowal/q/wiki/API-Reference.

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 the other 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. In the browser 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 on the server. 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, the Application object will construct the HTML to be sent to the server.

Philosophy

Separate Layout, Presentation 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 and sometimes classes of the DOM 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 some HTML needs to be repeated multiple times for a list of items, the JavaScript code will simply construct the relevant HTML 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.

Instead, in Sapphire, the designer will 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 example, the template node might look like this:

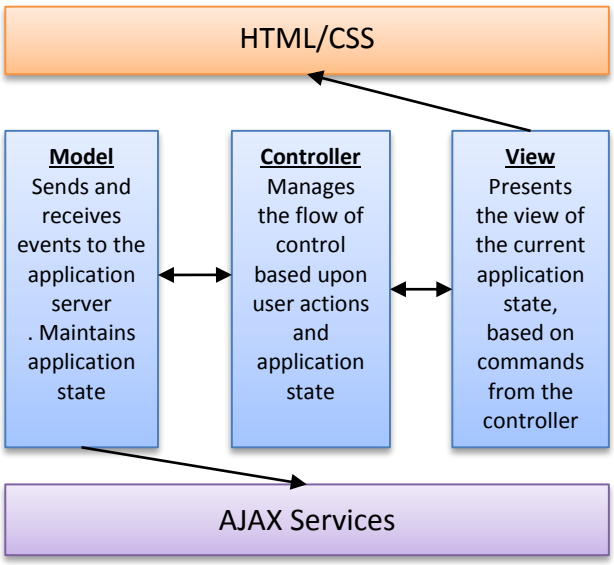
```
<div id="list-item" class="list-item-template template">
    <div id="list-item-img" />
    <div id="list-item-description"></div>
</div>
```

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('list-item');

    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.

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

Getting Started

Installing Sapphire

We're going to make a quick application to get up and running with Sapphire. Create a project directory to work in then execute the following commands at a command prompt within that directory.

```
npm install -g sapphire-express
npm install sapphire-express
npm install q
sapphire install
sapphire app test
test
node server
```

In your browser, hit the following url: localhost:8080. Wee, wasn't that fun? Let's look at these commands one at a time and see what we did.

```
npm install -g sapphire-express
```

This installs the sapphire-express framework globally. This will give access to the sapphire command line tool which we will use heavily when developing an application. This tool is used to create boilerplate for a number of sapphire features.

```
npm install sapphire-express
npm install q
```

These commands install sapphire and Q into your project node_modules directory.

```
sapphire install
```

This installs the server.js file that will be used to run sapphire applications. Although this file can be changed, it should not need to be.

```
sapphire app test
```

This creates a basic application.

```
test
```

When you executed the previous application, it created a batch file that configures the system variables that point to the configuration for the application. (need linux details here).

```
node server
```

This runs your application.

Using the Command Line Tool

Section 3

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 feature. They are located under the assets folder for the feature or application.
- Page files. These are the page template files; they will be served from pages directory under the application or feature directory.
- Dialog files. These are the dialog template files; they will be served from the dialogs directory under the application or feature 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.

addPanel

```
addPanel : function(setName, spec)
```

Call this method to add a loadable panel to the application. Panels are sub-parts of an application that are not pages or dialogs, but managed separately. For instance, a page may need many sub-parts, each one standing alone. Panels are specified using a data structure with the following members

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

Parameters:

setName	This is a name of a panel set where this panel will be used. This must be a valid JavaScript identifier.
spec	This is the specification for this panel

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 features—account and header—and Q for promises.

```
var application = require('application.js');
var account = require('../features/account/account.js');
var header = require('../features/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/features/header/dialogs/signup.html',
        javascript: [
            '/destination/assets/js/Views/Dialog.js',
            '/destination/assets/js/Controllers/Dialog.js',
            '/destination/features/header/assets/js/Views/Signup.js',
            '/destination/features/header/assets/js/Controllers/signup.js'
        ],
        css: [
            '/destination/assets/css/dialogs.css',
            `/destination/features/header/assets/css/signup.css'
        ]
    });


    var promise = account(req, res, builder)
        .then(header.bind(this, req, res))
        .then(function(builder)
        {
            app.getHTML(callback);
        }).done();
}
```

Feature Class

A feature is a set of build instructions where the required assets such as templates, js and css are under a single directory. All the instructions to build this part of the application use paths relative to the path of the feature's javascript file. Use the Feature class to create a feature.

The methods of the Feature class mirror methods in the Application class, but any relative paths specified in any of these methods will be modified to point to the feature directory.

The Feature class is implemented as a convenience, so that paths do not need to be duplicated multiple places, and so that the feature can be more easily relocated.

initialize

```
initialize : function(app, path)
```

This is the constructor for the feature. The path should be absolute rooted off the app directory, e.g.

```
var admin = new Feature(app, '/horizon/features/pages/admin/'
```

Parameters

app	The application object that this feature is a subset of.
path	The path to the feature root.

Example

The following code is an example of a feature.

```
var Q = require('q');
var Feature = require('feature').Feature;

module.exports = function(req, res, app)
{
    var admin = new Feature(app, '/horizon/features/pages/admin/');

    admin.addPage({
        name: 'admin',
        url: 'templates/admin.html',
        javascript: ['assets/js/Controllers/Admin.js', 'assets/js/Views/Admin.js'],
        css: []
    });

    admin.addPanel('admin', {
        name: 'users',
        url: 'panels/users.html',
        javascript: ['assets/js/Controllers/Users.js', 'assets/js/Views/Users.js'],
        css: []
```

```
    });

    admin.addDialog({
        name: 'edit-user',
        url: 'dialogs/edit-user.html',
        javascript: [
            '/horizon/assets/js/Views/Dialog.js', '/horizon/assets/js/Controllers/Dialog.js',
            'assets/js/Views/EditUser.js', 'assets/js/Controllers/EditUser.js'
        ],
        css: ['/horizon/assets/css/dialogs.css', 'assets/css/edit-user.css']
    })

    return Q(app);
}
```

Notice that all the files local to the feature are specified with relative paths, while those that are outside of the feature, such as shared JavaScript and CSS files are specified with an absolute path.

This example is implemented using promises, which allows the main application to include this feature as part of a promise chain.

```
    var promise = account(req, res, app)
        .then(header.bind(this, req, res))
        .then(admin.bind(this, req, res))
        .then(function(app)
        {
            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: js, css, images, pages, panels, dialogs
- <sub application directories>
- **features/** features go here
- **services/** services go here
- **node_modules/** needed packages go here
- **templates/** html template files

The directory structure for features is similar to applications

- **assets/** static assets for your application
- **templates/** html template files

Section 3

Client API

Packages

The Sapphire framework has a concept called packages. A package is a namespace for classes. By putting new classes into packages, it frees the global name space, and permits organizing classes by functional groupings. For example, the package `Horizon.Controllers` could hold all of the application's controller classes, and `Horizon.Views` could hold all the applications view classes. When this is the case, you can have classes with the same name but in different packages, such as `Horizon.Controllers.Admin` and `Horizon.Views.Admin`.

There are typically two top level packages, `Sapphire` which contains all the framework specific classes, such as `Sapphire.Application`, and a package that represents the application, for example, `Horizon`.

There are two global functions for using packages. `Package` and `Import`.

Package

```
function Package(name, members)
```

This function opens a package to add new members. Use this to defile new classes.

Parameters

name	the name of the package. Use dot notation to nest packages, for example, "Horizon.Views".
members	an object that contains the new members to add to this package.

Example

```
Package('Horizon.Controllers', {
    Admin : new Class({
        Extends : Sapphire.Controller,
    }
});
```

Import

```
function Import(name)
```

Call this function to get a reference to a package. This is handy if you need to reference a number of classes in the same package. Rather than using the full name, you can use the imported package.

Parameters

Name	The name of the package to be imported, using dot notation.

Example:

```
var views = Import('Horizon.Views');

.   .   .

this.ListView = new views.List();
```

Global Variables

The framework declares two global variables. One is `SAPPHIRE` which is used as a name space for a number of framework objects, such as the application object. The other variable is the namespace declared in the server-side Application class. The namespace is used to hold the application defined variables added while building the HTML.

Sapphire.Eventer Class

Use the Eventer class to listen for and fire events. This class is used internally by the Sapphire classes, such as Application. To add eventing to your own classes, you should extend them from this as a base class.

listen

```
listen : function(name, callback)
```

Call this method to register a listener for an event.

Parameters

name	the name of the event to listen for.
callback	the function to call when the event is fired.

Returns

This function returns a unique identifier that can be used to remove the event listener.

fire

```
fire : function(name, …)
```

Call this method to fire an event. All registered event handlers will be called in the order they were added.

Parameters

name	The name of the event to fire
…	Any additional parameters are passed to the event handlers.

remove

```
remove : function(name, id)
```

Call this function to remove a previously registered event handler.

Parameters

name	the name of the event
id	the value returned from listen

Sapphire.Application Class

The application class on the client is different from the Application class on the server. Whereas the server Application class revolves around constructing the HTML for the application, the one on the client is used to manage the application space. There is a single instance of the application class created by the framework, `SAPPHIRE.application`.

Some of its primary functions are the management of pages, dialogs and panels.

Pages

Pages appear in a fixed area of the application specifically reserved for them. The application object assumes there is a DOM node with the Id of 'pages' and will use this node to display the active page.

An individual page is a specification of the html that will occupy the pages area of the application, along with all the assets necessary to render the page and perform the necessary functions it represents.

Dialogs

Dialogs are similar to pages, and they are specified the same way. However, dialogs are designed to be modal, and to be displayed over the rest of the application. Also, more than one dialog can be displayed at any given time, whereas pages can only have one active at a time.

Panels

Panels are sort of like pages, except that you can create multiple panel sets, each set being displayed in a different region of the application. Panels are specified in the same way as pages and dialogs, and like them are hot loaded.

In addition to pages and dialogs, the Application class also acts as a registry for global model, views and controllers. And perhaps most importantly, it controls the application startup flow and the hot-loading flow.

Startup Flow

Two key events are fired during application startup, 'start' and 'ready.'

start	this is fired before any pages or dialogs are displayed, but after the DOM is fully loaded. This method can be used to delay full application startup until some prerequisite action has taken place, for example, an intercessory ad. When the start event is fired, a callback function is passed to the event listeners. The 'ready' event will not be fired until every listener to 'start' has called this method.

ready	This is fired when the start events have all finished. This is the signal that everything is ready to go, and pages can now be displayed, and normal operation can proceed.

Hot Loading Flow

There are two key stages of hot loading, the load itself and the showing of the page, dialog or panel that was hot loaded. The following events are fired during hot loading.

load	This is fired when all the resources for a hot-loaded object have been loaded, this includes the HTML, the CSS and the JavaScript. The HTML for the page is in the DOM when this event is fired.
show	This is fired every time the hot-loaded object is displayed using an application showXXX method.
firstShow	This is only fired the first time the object is shown.

showPage

```
showPage : function(name, …)
```

Call this method to hide the current page and show a new one. If the current page has not been marked as dontPrune, then it will be removed from the DOM. Before the page has been removed, any hide events will be fired. The show events will be fired once the page has been added back into the DOM.

Parameters

name	The name of the page
…	Any arguments passed after name will be passed to any show listeners for this page

showDialog

```
showDialog : function(name, …)
```

Call this method to show a dialog. Any dialogs already shown will remain shown. Show events will be fired once the dialog has been added back into the DOM. Dialogs are modal elements, and must be completed before the application can proceed.

Parameters

name	the name of the dialog
…	Any arguments passed after name will be passed to any show listeners for this dialog

Returns

This returns a Q promise. A deferred is passed as the first parameter to the show listeners, and they can use this deferred to fulfill the promise with the result of the

dialog. For example, if the dialog is intended to solicit the click of a yes or no button, then when one of the buttons is clicked, the deferred's resolve method can be called with an indication of which button was clicked. This will fulfill the promise and its then method will be called.

```
SAPPIRE.showDalog('yesno', 'Do you really want to take a nap')
    .then(function(which)
    {
        if (which == 'yes') alert('ZZZZZZZZzzzzzzzzzzzz');
    });
```

hideDialog

hideDialog : function(name)

Call this method to hide a dialog. This method must be called to dismiss a dialog, it will not be called automatically.

Parameters

name	the name of the dialog to hide.

showPanel

```
showPage : function(set, name, …)
```

Call this method to hide the current panel in the given set and show a new one. Before the panel has been removed from the DOM, any hide events will be fired. The show events will be fired once the page has been added back into the DOM.

Parameters

set	The name of the panel set, defined when the panels were added in the server code
name	The name of the panel
…	Any arguments passed after name will be passed to any show listeners for this panel

setPanelContainer

```
setPanelContainer : function(set, selector)
```

Call this method to set the region of the application that will receive the panels of the given panel set.

Parameters

set	The name of the panel set, defined when the panels were added in the server code
selector	A JQuery selector for the region to display the panels.

listenPageEvent

```
listenPageEvent : function(event, which, callback)
```

Call this method to listen for a page specific event.

Parameters

event	The name of the event being listened for, for example, 'show.'
which	Which page you are listening to. If this string is empty, it will listen to all pages.
callback	The function to call when the event is fired.

listenDialogEvent

```
listenDialogEvent : function(event, which, callback)
```

Call this method to listen for a dialog specific event.

Parameters

event	The name of the event being listened for, for example, 'show.'
which	Which dialog you are listening to. If this string is empty, it will listen to all dialogs.
callback	The function to call when the event is fired.

listenPanelEvent

```
listenPanelEvent : function(event, set, which, callback)
```

Call this method to listen for a dialog specific event.

Parameters

event	The name of the event being listened for, for example, 'show.'
set	The name of the panel set, defined when the panels were added in the server code
which	Which panel you are listening to. If this string is empty, it will listen to all panels for the set.
callback	The function to call when the event is fired.

registerController

```
registerController : function(name, controller)
```

Call this method to register a global controller. When a controller is registered, other controllers can find it to call its methods.

Parameters

name	The name this controller should be indexed on.
controller	The constructed controller object. Controllers generally inherit from Sapphire.Controller.

registerView

```
registerView : function(name, view)
```

Call this method to register a global view. When a view is registered it is available to controllers which can use the view to update application presentation.

Parameters

name	The name this view should be indexed on.
view	The constructed view object. Views generally inherit from Sapphire.View.

registerModel

```
registerModel : function(name, model)
```

Call this method to register a global model. When a model is registered it is available to controllers which can use the model to call service functions and access data state.

Parameters

name	The name this model should be indexed on.
model	The constructed model object. Models generally inherit from Sapphire.Model.

getController

```
getController : function(name)
```

Call this method to get a previously registered controller.

Parameters

name	The name of the desired controller. This is the name passed to registerController.

Returns

The controller instance.

getView

```
getView : function(name)
```

Call this method to get a previously registered view.

Parameters

name — The name of the desired view. This is the name passed to registerView.

Returns

The view instance.

getModel

```
getModel : function(name)
```

Call this method to get a previously registered model.

Parameters

name — The name of the desired model. This is the name passed to registerModel.

Returns

The model instance.

Sapphire.Templates Class

Use the templates class to manage templates, which are DOM nodes that can be reused, mostly useful for partials. These nodes are removed from the DOM at start up and when HTML is hot loaded for pages, dialogs and panels. To specify the HTML for a template, add the class "template" to the HTML element that is the template. The id of that element will be used to reference that template in your JavaScript. To use templates, your application needs to add the JavaScript file "/assets/js/lib/templates.js".

There is one instance of the templates class, `SAPPHIRE.templates`.

get

```
get : function(which)
```

Call this function to get a copy of your template. The following code shows an example of using templates.

```
draw : function(users)
{
    var container = $('#user-list');
    container.empty();

    users.each(function(user)
    {
        var template = SAPPHIRE.templates.get('user-item');

        template.find('#username').html(user.identity.name);
        template.find('#edit-user-button').click(this.fire.bind(this, 'editUser', user));

        container.append(template);
    }, this);
}
```

Parameters

which — The id of the DOM node that is your template.

Returns

This function returns a copy of the template as a DOM node that is not yet attached to the document.

Sapphire.History Class

The history class manages the web client url to support both deep linking and the back button. It uses a URL hash. Each page switch generates a new URL, and hitting the back button will automatically switch to the previous page. If using the history functionality, the only parameter that should be passed to SAPPHIRE.application.showPage should be an object that represents a pseudo query string. Only numbers and strings should be in this object. The query string will be appended to the hash part of the url. An example url hash might be "#profile?user=10009". This would be the result of calling this code.

```
SAPPHIRE.application.showPage('profile', {user: 10009});
```

When a page is replayed with the back button, this same function will be called by the history class. To use the history class, your application needs to add the JavaScript file "/assets/js/lib/history.js".

There is only one instance of this class, `SAPPHIRE.history`. It has no callable methods.

Sapphire.Services.AjaxService Class

There are two kinds of services, socket based, and AJAX based. The service handler classes are written as mixins, so the application can create its own services object that includes the desired functionality. The general rule is that there is only a single service instance, which can be hooked by anybody who wants to examine the service result for global changes. For example, we have the following code from a model object that looks for any changes in the login state, and fires an event if found.

```
HORIZON.service.listen('ajaxResponse', this.onServiceResponse.bind(this));

.   .   .

onServiceResponse : function(response)
{
    if (response.identity != undefined)
    {
        this.identity = response.identity;
        HORIZON.identity = this.identity;
        this.fire('identityChange', response.identity);
    }
}
```

The following example shows how to integrate this mixin class with an application defined service class. Notice that the service class must extend the Eventer class, as the AjaxService class fires events.

```
Package('Horizon', {
    Service : new Class({
        Extends : Sapphire.Eventer,
        Implements: [Sapphire.Services.AjaxService],

        initialize : function()
        {
            this.parent();
            this.initializeAjaxService()
        }
    })
});

HORIZON.service = new Horizon.Service();
```

call

```
call : function(url, data, method, type)
```

Call this method to send a service request to the server-side code. Calls to this method are generally handled through a model class.

Parameters

url	The url of the service to call. This must follow the conventions documented earlier in this document.
data	This is an object with all the data that should be sent to the service. It should be a single level, with simple types. It can contain arrays of simple types.
method	This is the HTTP method, one of GET, POST, PUT or DELETE.
type	Specifies the type of return data expected, or the manner of its return. The default is json.

Returns

Returns a Q promise that will be resolved when the service completes.

Sapphire.Services.SocketService Class

This is the implementation of the socket services. The sapphire framework can be configured to listen for socket events, this is the client side API to call those socket services.

setupSocketServer

```
setupSocketserver : function(server)
```

Call this method is your socket class' initializer to set up the server that should be used for socket messages.

Parameters

server	the url of the server to use for socket messages

message

```
message : function(path, data)
```

Call this method to send a socket message to the server. The socket message will be passed a callback, which is expected to return a single result. This method returns a promise that will be resolved with that result.

Parameters

path	The message path. This is expected to follow this format <application>/<object>/<message>
data	The data to be passed to the message handler

Returns

A promise that will be fulfilled with the message result

socketListen

```
socketListen : function(what, callback)
```

Call this method to listen for an asynchronous message from the server. There is a limitation of one listener per message. This limitation will be removed.

Parameters

what	a string representing the message being sent
callback	the function to call when the message is received.

Translation

There are two inputs to the translation service, one is the list of strings that need to be translated, and the other is a list of global lookup strings that can be used in translated text. To specify this data, the translation class looks for two global variables, `translations` and `lookups`.

The `translations` variable is an object where the name of each element is the source string in English, and the value is the translated text.

The `lookups` variable is an object that contains the global lookups for translations. Lookups are strings that appear in curly braces in a translation, for example, "`please place the {thing} on the {surface}`". Lookups are frequently specific to the individual strings being translated, but sometimes they represent some sort of global variable, such as user name, "`Hello {name}`". The name of each element in this object is the name of the lookup and the value is the string to use to replace it.

There are two ways that strings can be translated, one is in the html source, and the other is in code. In the HTML source there will frequently be a number of strings such as labels and menu items that need to be translated. To translate these strings add the class 'translate' to the element.

The other way is to translate in code. To do so, use the _T function.

Another thing to note is that strings to be translated should never be constructed piecemeal. All translated strings should form a complete sentence. This is because the grammar of a language will not necessarily follow the same rules as English.

To use the translation facility your application needs to add the JavaScript file "/assets/js/lib/translate.js".

_T

```
function _T(text, replacements)
```

Call this function to get the translation for a string.

Parameters

text	the full text to be translated
replacements	an object containing the values of any replacements in the string that are not in the global replacement list.

Returns

The translated string

Example

```
$('#message').html(_T('please put the {object} on the floor', {object: 'football'}));
```

marklar query string

To facilitate translation verification there is a query string parameter that can be added that will affect the way translations are performed. This parameter is called marklar and can take these values.

"sub"	all replacement text will be replaced with the string "marklar." Use this to test for global replacements that are missing
"raw"	all strings that are translated will be replaced with the string "marklar." Use this to find untranslated strings.