The Inner Workings of the HTML5 Silicon Designer Client

A Technical Whitepaper by Dorian Smiley and Max Dunn

Overview

Our Silicon Designer product is perhaps the most powerful online editing system ever created. This whitepaper explains how we built the HTML5 web client and the architectural approaches we took to attain goals such as:

1. We want to create any number of custom implementations of Silicon Designer, allowing for customization of the core feature set while still allowing us to leverage a common code base and preserve functionality.
2. We want to provide complete customization of the view that can be handled by the design team (our client's or ours) without a robust knowledge of JavaScript.
3. We want the programming to be manageable by a team rather than an individual, with multiple developers collaborating on different areas of the code without colliding with each other.
4. We want automated testing to maintain the integrity of the code base as development occurs, avoiding human testing to the extent possible.

The Silicon Designer SDK

The SDK encapsulates core functionality including services, serialization, event management, data binding, and document view components. It also provides access to global application resources such as the event bus, services, model, and serializers. It is minified and included in the core Angular application. In production the SDK is typically served from a CDN, allowing for a single shared stable code base that is deployed for all application implementations.

Test-driven design

The core SDK uses Karma and Jasmine to enable both automated unit testing and automated end-to-end testing. Every object in the SDK has a corresponding unit test, including advanced implementations that require testing of asynchronous operations. End-to-end tests are also developed for view components to test UI interactions in isolation before being deployed in a broader application, further enhancing the integrity of the code.

Developed using prototypal inheritance enabling polymorphism

All objects in the SDK implement prototypal inheritance in a standardized manner. An object utility exposes functions to both extend and mix in objects with proven cross-browser stability. The ability to both extend and mix in objects ensures DRY code, and enables objects to incorporate any required behavior already exposed through existing objects. The fact that prototypal inheritance is enforced for every single object in the SDK allows for the override of any behavior at a fine grained level, while still preserving access to the behavior of the base class.

In the following example a new object is declared. Immediately following are the declaration of protected instance variables, then the call to the base class constructor, followed by the declaration of accessor methods. Immediately following the declaration of the object's constructor is the call to the core SDK utility responsible for extension. This order of operation is critical to preserving base class functionality while allowing for sub class overrides.

```
SpiSdk.AsyncOperationModel = function(){
    var _asyncOperationCount = 0;//Number
    var _asyncOperationComplete = true;//Boolean

    SpiSdk.Subject.prototype.constructor.call(this);

    this.addProperties({
        asyncOperationCount: {
            get: function () {
                return _asyncOperationCount;
            },
            set: function (val) {
                _asyncOperationCount = val;
                this.Notify(val, "asyncOperationCount");
            }
        },
        asyncOperationComplete: {
            get: function () {
                return this.asyncOperationCount <= 0;
            },
            set: function (val) {
                _asyncOperationComplete = val;
                this.Notify(val, "asyncOperationComplete");
            }
        }
    });
}
/************* Inherit from Subject for data binding *************/
SpiSdk.ObjectUtils.extend(SpiSdk.Subject, SpiSdk.AsyncOperationModel);
```

Example of object mix in using our SDK

```
SpiSdk.ObjectUtils.mixin(SpiSdk.AbstractEventDispatcher, SpiSdk.ArrayList, this);
```

In this example the ArrayList class mixes in the functionality of the core SDK event dispatcher. This enables event dispatching and listening on plain javascript objects.

Browser-specific override mechanism

In engineering our browser-specific override architecture we took full advantage of the fact that every object implements prototypal inheritance. We engineered a series of fixer objects that are mapped to specific browsers and versions using feature testing. The architecture allows for complete override on an object, or just specific methods. This enables all SDK components to continue to function without any knowledge that the underlying implementation has been changed. This is immensely important when separating the concerns of dev teams, freeing the front end team from the challenges of fixing browser bugs within the SDK, and allowing the SDK team to test specific features in isolation, and implement fixes without ever having to test in an implementing application.

Below is an example of a fixer for Chrome.

```
SpiSdk.ChromeFixer.run = function () {
    SpiSdk.DocumentView.prototype.scaleTo = function (scale) {
        // This is an ugly workaround for Chrome. If some scale transform is applied to the
        // div, and we change it to another one, Chrome blurs all text. To avoid this scale
        // recalc issue, we "reset" scale transform, redraw the element and apply a new one.
        var viewport = $(this.viewport.element);
        viewport.css({"transform" : '', visibility : 'hidden'});
        window.setTimeout(function () {
            viewport.css({"transform" : 'scale(' + scale + ')', visibility : 'visible'});
        }, 0);
    }
}
```

In this example the document view's scaleTo function is overridden to support a work around for Chrome's poor text rendering when a scale transform is applied. The call to SpiSdk.ChromeFixer.run is executed within the feature tests contained in the main application. This enables the SDK to change behavior of specific methods to support browser limitations without any need to refactor additional code.

Framework specific implementations for increased interoperability

While our SDK is framework agnostic, we do occasionally create framework specific objects to take advantage of features, while maintaining abstraction via a common interface. For example, we created an HTTP service implementation for Angular. The object's interface definitions allow interoperability withing the SDK even though it uses Angular's $http service to perform HTTP requests. Injection of the object is handled through a combination of configuration variables and a service factory.

Advanced Text conversion between InDesign and HTML5

Without a doubt the biggest single challenge involved in creating our SDK was mapping the advanced text rendering capabilities of InDesign to HTML5. Because SPI employs some of the best typesetting and InDesign specialists on the planet, we were able to push the limits of what is possible with HTML5, and confidently rule out specifically what is not. The attributes we do support result in identical display between InDesign and supported browsers, which is no small accomplishment.

Our conversion format preserves all features for round trip with InDesign, hence no formatting data is lost along the way. To achieve this we did the following:

1. Map all InDesign text rendering features to their CSS counterpart.
2. Custom text rendering engine which ensures proper alignment and wrapping of text.
3. Engineer specific workarounds for anything not directly supported by CSS.
4. Ensure all InDesign text rendering and style information is mapped to model object attributes.
5. Created a CSS utility class that accepts any SDK model object and returns the corresponding CSS styles for that object.
6. Serialize model objects for export back to InDesign so data is not lost and unsupported attributes pass through.

The Core Angular Application

Test-driven design

As with the SDK, our core Angular application uses the Karma and Jasmine testing frameworks to enable both automated unit testing, and automated end-to-end testing. Every object in the Angular application has a corresponding unit test.

Developed using prototypal inheritance enabling polymorphism

Again, as with the SDK, every object implement prototypal inheritance. Much like the SDK, this enables application level objects, like mediators, to have specific methods overridden while preserving base class functionality. The power of this approach will become obvious as we explain how core Angular application features are overridden.

Graceful enhancement

Using a combination of feature testing and media queries, our core angular application provides browser fixers and CSS overrides to support different display sizes and browsers.

Inversion of control

1. Dependency Injection

Injection of resources is accomplished using Angular JS. However, the main application controller is responsible for receiving all injections from Angular and passing along in object initialization functions. This is an important difference from most Angular applications, and enables nearly every application object to stay framework agnostic, while still taking advantage of the IOC container. The basic flow of dependencies is as follows:

The application context declares all Angular modules, and exposes a static start up method which is called as part of differed initialization of the Angular framework. The start up method then assigns the controller for the main application module passing all dependencies.

Below is an example of the declaration of the required Angular modules. Applications that extend the core Angular application can either concatenate this array, or overwrite it completely.

```
SpiRef.Context.modules = [‘ngResource’,
    ‘ngRoute’,’ui.jq’,
    ‘ui.event’,
    ‘spiRefApp.services’,
    ‘spiRefApp.model’,

    … full code omitted ….

    ‘spiRefApp.commandMapper’,
    ‘spiRefApp.controllerMap’];
```

The main application controller method (defined using ng-controller) is then called which results in the initialization of all injected resources. Below is an example of the controller method receiving all dependencies and performing initialization of injected resources:

```
function Controllers($scope,
                   services,
                   model,
                   replaceControlImageButtonMediator,
                   recordSetNavButtonMediator,
… full code omitted ….
         ){
         controllerMap.init($scope,
                   services,
                   model,
                   replaceControlImageButtonMediator,
                   recordSetNavButtonMediator,
… full code omitted ….
                   );
         };
```

The controller mapper's init method performs actual object initialization. Injecting the controller mapper as a dependency allows us to change the behavior of how resources are mapped while still preserving the basic wiring of the initialization process. This will become clearer when we examine applications that extend the core Angular application.

Example of the declaration of the 'controllerMap' resource mapper:

```
angular.module('spiRefApp.controllerMap', ['ngResource']).factory('controllerMap', ['model',
SpiRef.BaseController.newInstance ]);
```

The controller mapper's init function starts up the core SDK, passing required dependencies, maps mediators, and attaches the application model to the scope. Finally the resource mapper calls the init function of the application's main mediator which triggers the default start up behavior of loading a document instance. In the example below, the 'controllerMap' dependency is mapped to an instance of SpiRef.BaseController.

```
SpiRef.BaseController.prototype.init = function( $scope,
                                                 services,
                                                 model,
                                                 fontSelectorMediator,
                                                 rotateButtonMediator,
                                                 boldButtonMediator,
                                                 italicButtonMediator,
                                                 underlineButtonMediator,
                                                 horizontalAlignButtonMediator,
                                                 verticalAlignButtonMediator,
                                                 colorPickerMediator,
                                                 fontSizeMediator,
                                                 mainMediator,
                                                 commandMapper ){
    //add the model attribute to the scope so it's available for data binding
    $scope.model = model;
    //init global resources
    SpiSdk.init( $scope.model.config, $scope );
    //Map commands
    commandMapper.mapCommands();
    // IMPORTANT: init the mediators! Because construction is delegated to angular we have
    // to init the mediator afterward so the SDK is inited and dependencies can be fulfilled
    fontSelectorMediator.init($scope);
    rotateButtonMediator.init($scope);
    boldButtonMediator.init($scope);
    italicButtonMediator.init($scope);
    underlineButtonMediator.init($scope);
    horizontalAlignButtonMediator.init($scope);
    verticalAlignButtonMediator.init($scope);
    colorPickerMediator.init($scope);
    fontSizeMediator.init($scope);
    //map events, you must use angular.bind to preserve this inside the mediator instance
    $scope.onFontSelectionChange = angular.bind(fontSelectorMediator,
         fontSelectorMediator.onFontSelectionChangeHandler);
    $scope.onRotateBtnClick = angular.bind(rotateButtonMediator,
         rotateButtonMediator.onClickHandler);
    $scope.onBoldBtnClick = angular.bind(boldButtonMediator,
         boldButtonMediator.onClickHandler);
    $scope.onItalicBtnClick = angular.bind(italicButtonMediator,
         italicButtonMediator.onClickHandler);
    $scope.onUnderlineBtnClick = angular.bind(underlineButtonMediator,
         underlineButtonMediator.onClickHandler);
    $scope.onHorizontalAlignBtnClick = angular.bind(horizontalAlignButtonMediator,
         horizontalAlignButtonMediator.onClickHandler);
    $scope.onVerticalAlignBtnClick = angular.bind(verticalAlignButtonMediator,
         verticalAlignButtonMediator.onClickHandler);
    $scope.onColorPickerChange = angular.bind(colorPickerMediator,
         colorPickerMediator.onClickHandler);
    $scope.onFontSizeChange = angular.bind(fontSizeMediator, fontSizeMediator.onClickHandler);
    mainMediator.init($scope);
}
```

By confining dependency injection into a single start up process exposed through a static start up method, we are able to isolate Angular implementation details into the smallest amount of code possible, which greatly enhances our ability to become interoperable with other frameworks.

2. UI Event Management and Mediator Mapping

We used the Angular ui-event directive essentially to glue the view to application mediators. For example:

```
<label ui-event="{click : 'onBoldBtnClick($event)'}">
```

In this example, the above label's click event is mapped to the $scope's onBoldBtnClick handler which was defined in the start up process described above:

```
$scope.onBoldBtnClick = angular.bind(boldButtonMediator, boldButtonMediator.onClickHandler);
```

Whenever the label is clicked, the boldButtonMediator instance that was injected into the context will have its onClickHandler function called, passing the event object. This enables our design team to use predefined ui-event tags to implement features with the complete freedom to exclude those that will not be used, or completely reshape the view including altering element id values, css classes, etc., without breaking one ounce of functionality.

3. Command Mapping

It may seem strange to other Angular developers that we would create a command map. After all, Angular can map events. However, our application architecture relies heavily on a central event bus that enables plain Javascript objects to broadcast application level events that need to be mediated, with greater efficiency and interoperability than can be offered by a framework such as Angular. In addition, other languages offer IOC containers that contain advanced command features, such as mapping command instances as singletons, that we absolutely wanted to take advantage of. For this reason, we found it beneficial to create our own command mapper that would work with the application's event bus, and expose some of the same advanced features we've grown fond of.

Below is an example of using our command mapper:

```
var commandMap = SpiRef.CommandMapper.getInstance();
commandMap.addCommand(SpiRef.AppEvent.START_UP, SpiRef.StartUpCommand);
```

The addCommand function also exposes two optional parameters: functionName and useSingleton. If functionName is not passed, the command object's execute method is used. If useSingleton is true, a single instance of the command will handle all events of the supplied event type.

In addition, the command map is injected into the application. This enables us to easily override how events are processed by commands, and change the concrete command implementation responsible for handling a specific event. Below is an example of a command map implementation that would be injected:

```
SpiRef.BaseCommandMap = function () {

}

//Do not call until SDK is inited
SpiRef.BaseCommandMap.prototype.mapCommands = function () {
    var commandMap = SpiRef.CommandMapper.getInstance();
    commandMap.addCommand(SpiRef.AppEvent.START_UP, SpiRef.StartUpCommand);

    … full code omitted ….
}

SpiRef.BaseCommandMap.newInstance = function(){
    return new SpiRef.BaseCommandMap();
}
```

4. Custom Directives and Factories

We heavily leverage Angular directives to support custom view components that require a large degree of set up and configuration. However, our method of defining directives deviates slightly from the norm. We use a directive factory to map a directive implementation to the specific tag or attribute. This layer of abstraction makes it easier to preserve application start up logic, and takes advantage of polymorphism to change behavior. For example:

```
angular.module('spiRefApp.directives', [])
    .directive('spidocview', function($document, $compile, $http ) {
        return {
            restrict: 'E',
            controller: SpiRef.DirectiveFactory.getInstance().getDirectiveController,
            link: SpiRef.DirectiveFactory.getInstance().getDirectiveLink,
            transclude: true,
            template:'<div id="docView" ng-transclude></div>'//THIS IS TEMP: real html here
        };
    });
```

In this example, the spidocview directive is declared. Any time an instance of the spidocview tag is encountered, the DirectiveFactory's getDirectiveController and getDirectiveLink functions are called. Order of execution in those functions is important, and each function fulfills specific dependencies for the directive instance. The call the getDirectiveController receives the $document, $compile, and $element dependencies. The call to getDirectiveLink receives the $scope, $element, $attrs, and $controller dependencies. All our directives share a common base class, SpiRef.SpiBaseDirective, that stores these dependencies as instance variables. These dependencies are then used by different directive implementation to create their view. Below is an example of our document view directive:

```
SpiRef.SpiDocViewDirective = function() {
    SpiRef.SpiBaseDirective.prototype.constructor.call(this);
}
/************* Inherit from base directive *************/
SpiSdk.ObjectUtils.extend(SpiRef.SpiBaseDirective, SpiRef.SpiDocViewDirective);

SpiRef.SpiDocViewDirective.prototype.init = function(){
    this.addEventListeners();
    var docViewDiv = this.element.find('#docView');
    $("#errorModal").dialog({ autoOpen: false });
    var docView = new SpiSdk.DocumentView();
    docView.model = this.scope.model;
    docView.element = docViewDiv[0];
    docView.setUpView();
}
```

The call to the directive's init function is handled by the SpiBaseDirective, and is called once all dependencies are fulfilled. Of course any implementation could override the behavior of the SpiBaseDirective base class. The init method first adds any required event listeners, then selects the element with the id of docView. This element is passed to the SpiSdk.DocumentView component instance, along with other required dependencies, and the component's setUpView function is called.

Again, because we use prototypal inheritance throughout our application architecture, we can easily override specific methods of the SpiRef.SpiDocViewDirective. Further, because concrete directive implementations are mapped via a factory, we can easily change which implementation is used without changing our core application start up process or wiring. This offers immense time savings when needing to alter the behavior of a single directive method, or remap a directive implementation entirely.

5. Interoperability of view components

We support Angular view components, jQuery component, web components, or 100% custom components. This is due to the fact that there is no dependency between the view and the surrounding application. The link between the two is accomplished using a combination of the Angular ui-event directive and custom directives, both of which use inversion of control which decouples the view from the application. This is an enormous benefit for our design team as it can have complete independence in choosing components that deliver on the best possible use case, and eliminates concerns over integration with the application framework.

The Results of this Architecture

While there are many benefits we have realized in the adoption of this architecture, the broadest benefits can be summarized as follows:

Extremely reusable, portable, and customizable code

It's extremely easy to create any number of custom applications without the need to duplicate code. At a minimum a custom application includes the following:

sd.html – The main application HTML page.

spisdk.min.js – The minified core SDK.

spirefapp.min.js – The minified core Angular application.

Model.js – Contains configuration settings and overrides the dependency injection for the model in the core Angular application. For example:

```
Model = function(){
    var model = SpiSdk.ModelLocator.getInstance();
    model.config.parserCode = "1.0";
    model.config.eventDispatcherCode = "abstract";
    model.config.context = 'printing';

    … full code omitted ….

    return model;
}
//override the model resource injected into the core angular application
angular.module('spiRefApp.model', ['ngResource']).factory('model', [ Model ]);
```

app.js – Sets up error handling, sets the main application module reference, and starts up the application. For example:

```
/* App Module */
//init the app with the required dependencies to fullfill all DI
var spiApp;
angular.element(document).ready(function() {
    spiApp = angular.module('spiRefApp', SpiRef.Context.modules);
    // By default, AngularJS will catch errors and log them to the Console. We want to
    // keep that behavior; however, we want to intercept it so that we can also log the
    // errors to the server for later analysis.
    spiApp.provider(
        "$exceptionHandler",
        {
            $get: function( errorLogService ) {

                return( errorLogService );

            }
        }
    );
    angular.bootstrap(document, ['spiRefApp']);
    SpiRef.Context.module = spiApp;
    spiApp.run(SpiRef.Context.startUp);
});
```

In this example, the core Angular application context is configured to point to the main application module (spiApp), and the app is inited passing the core Angular application's start up method.

While the view still has to be customized by the design team, nothing further is required to configure the application. Custom directives will be executed, mediators will be mapped to events, and the application will render documents just as it did in the core Angular application.

Ability for design teams to work independently of programming teams

Because the design team uses predefined directives to implement application features, they are free to develop the view independently of the programming team. All UI controls such as text styling buttons, font selection menus, etc., are mapped to application behavior using Angular's ui-event directive. This enables the design team to swap out components, adapt CSS, and completely reshape the HTML.

Ability to split programming teams into core SDK, core Angular, and custom implementation teams

Because our core SDK is so loosely coupled to the core Angular application, our SDK team can work independently without worrying about how the components they create are consumed. Test-driven design ensures that SDK components are stable and ready for use by the Angular team upon release. Our core Angular team can focus on the implementation of new SDK features, and is free to create new directives and mediators that fulfill SDK component dependencies, and trigger application behaviors. And because the core Angular team handles the details of wiring SDK components into the surrounding application, the custom implementation team can focus on creating mediator and directive overrides to support custom behaviors, and varying event object formats.

Rapid production workflows

Because we can divide our programming teams into separate disciplines, and completely separate them from the design team, we are able to focus resources on tasks best suited to their unique skill set. We are also able to manage the teams using different roadmaps, increasing our overall agility. Team sizes can also scale independently of each other which allows us to staff around the application layer that requires the most resources based on the project's road map.

Reliable code checked against automated tests throughout the product lifecycle

Because our IDE is integrated with our testing framework, developers can monitor tests as they make changes. In addition our testing suites are constantly refined to provide fine grained visibility for our QA engineers. Whether creating a patch release, or developing advanced features for the latest sprint, test-driven design is fully integrated into our development process.

Increased stability of deployed applications

A major net benefit of being able to deploy both our SDK and core Angular application as minified JS files is the fact that applications using these resources do not need to duplicate code. For example, a custom application can fully leverage our core Angular application's start up process without duplicating the code in any way. If a bug is discovered in that start up process, the core Angular application is refactored to correct the issue, and the minified library is redeployed to our CDN in the next release, automatically correcting the issue in any custom application. This greatly enhances the overall stability of production applications because there is no need to dig back into every legacy application to correct bugs that may have propagated out as the result of duplicate code.