# The Fujion Framework

*Developer Guide*

Version 1.1

29 August 2018

**http://www.fujion.org**

## Table of Contents

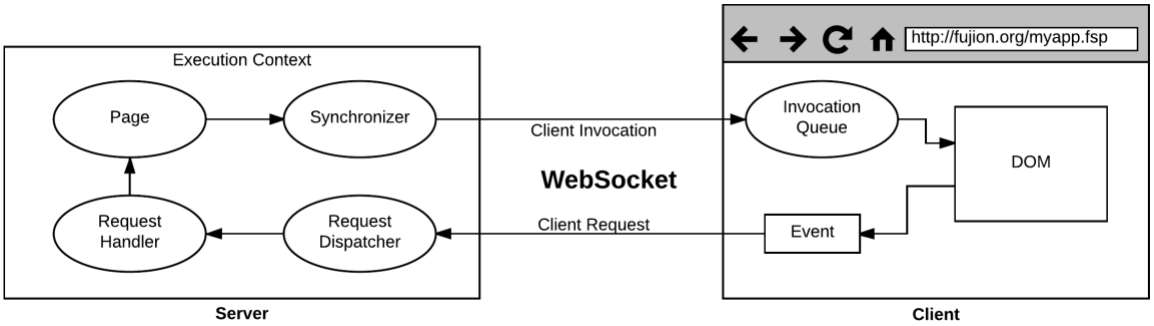
## Introduction

The **Fujion Framework**[*] is an open source, server-centric, single-page web framework that is built upon the Spring Framework and other open source technologies. At its core is a Java-based component model. Each server-side Java component has a client-side JavaScript counterpart, or widget. Through a synchronization engine that utilizes a WebSocket connection, changes to the state of each Java component will be reflected in its corresponding JavaScript widget. Similarly, events and state changes originating at the client are propagated to the server. The Fujion name was inspired by this effective fusion of client and server states.

[*] Pronounced like "*fusion*", the "*j*" emphasizing its Java underpinnings.

## Architecture

The Fujion Framework is a powerful, open-source, WebSocket-enabled web application framework that provides a rich user experience on par with thick client applications. Employing a programming paradigm more closely aligned with thick client development than traditional web application development, Fujion effectively blurs the separation between client- and server-side development by creating a unified model of web application development. Fujion allows the developer to create and modify the user interface programmatically or declaratively via XML resources known as Fujion Server Pages (FSPs). Fujion's synchronization engine transparently synchronizes changes to the server-side model of the user interface with the client and conveys events triggered in the client to event handlers on the server, all using a WebSocket connection. This architecture is depicted below:




While Fujion is at its core a server-centric framework (complex applications may be created without the need to write any JavaScript code), it seamlessly integrates with client-centric frameworks like React and AngularJS. It is also easily extended, supporting the creation of custom components.

The Fujion Framework uses a UI component model very similar to traditional thick client frameworks such as Windows Foundation Classes, .NET's Winforms Library or Delphi's Visual Component Library. As with these traditional models, Fujion UI components have parent-child relationships, event handlers, properties and methods that affect their presentation and behavior. A developer can create a Fujion-based UI in one or both of two ways: declaratively using the FSP markup language in a Fujion Server Page (i.e., web resources ending in a `.fsp` extension) or programmatically by directly manipulating the component model in code. Regardless of which method is chosen (and often it will be a combination of both), Fujion creates an in-memory model of the UI on the server. Once attached to a page, each server-side component has a companion widget on the client web browser. Changes to component state on the server are automatically reflected in the client through the synchronization engine. Events generated at the client (such as clicking on a button) are automatically propagated to event handlers on the server.

The FSP markup language syntax closely mirrors Fujion's underlying component model, with each tag having a corresponding component class. Consider the following sample FSP with some simple markup:

```
<page>
  <window name="myWindow">
    <label name="aLabel" label="Click the button please."/>
    <button name="aButton" label="Click Me"/>
  </window>
</page>
```

The **page** root tag is not required if you have only one top-level element (**window** in this example). In the above example, we have declared three Fujion UI components: a window, a label, and a button. The window is the parent component of the other two by virtue of its position in the XML hierarchy. All Fujion components have a common set of properties (represented as attributes in the FSP). For example, the **name** property is common to all Fujion components.

Returning to the above sample FSP, one can create an identical UI in code. The in-memory representation would be identical to that created by Fujion when it compiles and materializes the FSP declaration. Consider the following Java code snippet:

```
Window myWindow = new Window();
myWindow.setName("myWindow");
Label label = new Label("Click the button please.");
label.setName("aLabel");
label.setParent(myWindow);
Button button = new Button("Click Me");
button.setName("aButton");
button.setParent(myWindow);
```

This code would create the same in-memory model as the FSP above. Note that the Fujion Java class names typically (but not always) correspond to the FSP tags by the same name.

## Project Organization

The Fujion Framework is organized into several physically distinct jar files, each performing a related set of functions and each identified by a unique Maven artifact identifier. All framework artifacts share a common Maven group identifier of `org.fujion`. Below are the Java packages that comprise the Framework, organized under the identifier of the Maven artifact that contains them.

| **artifactId: fujion-common** | |
|---|---|
| **Java Package** | **Description** |
| **org.fujion.common** | **This is a collection of utility methods for manipulating strings, numbers, dates, etc. and of classes in support of commonly used data structures such as caches and registries.** |
| **artifactId: fujion-core** | |
| **Java Package** | **Description** |
| **org.fujion.ancillary** | **Classes that augment and support the component classes.** |
| **org.fujion.annotation** | **Classes that support code annotations.** |
| **org.fujion.client** | **Classes that coordinate interactions with the web client.** |
| **org.fujion.component** | **The base and implementation classes for Fujion components.** |
| **org.fujion.core** | **Miscellaneous classes in support of core capabilities.** |
| **org.fujion.dragdrop** | **Classes and interfaces in support of drag-and-drop functions.** |
| **org.fujion.event** | **Event classes and supporting infrastructure.** |
| **org.fujion.expression** | **Classes related to Expression Language (EL) support.** |
| **org.fujion.ipc** | **Support for inter-process communication.** |
| **org.fujion.logging** | **Bindings between client- and server-side logging.** |
| **org.fujion.model** | **Classes supporting model-based rendering.** |

| | |
|---|---|
| **org.fujion.page** | **Classes supporting the compilation and materialization of Fujion Server Pages.** |
| **org.fujion.script** | **Base classes for the integration of server-side scripting languages.** |
| **org.fujion.servlet** | **Servlet configuration and extensions.** |
| **org.fujion.spring** | **Support for Spring namespace extensions and custom scopes.** |
| **org.fujion.taglib** | **Tag library support.** |
| **org.fujion.theme** | **Theming support.** |
| **org.fujion.websocket** | **WebSocket connection and session support.** |

### artifactId: fujion-test

| Java Package | Description |
|---|---|
| **org.fujion.test** | **A unit testing framework with mock classes for simulating a running web application.** |

### artifactId: fujion-testharness

| Java Package | Description |
|---|---|
| **org.fujion.testharness** | **Sample web application for demonstrating and testing Fujion components.** |

### artifactId: fujion-sandbox

| Java Package | Description |
|---|---|
| **org.fujion.sandbox** | **Sandbox application for dynamic development and testing of FSP's.** |

### artifactId: fujion-script-clojure

| Java Package | Description |
|---|---|
| **org.fujion.script.clojure** | **Support for Clojure-based scripts.** |

### artifactId: fujion-script-groovy

| Java Package | Description |
|---|---|
| **org.fujion.script.groovy** | **Support for Groovy-based scripts.** |

| artifactId: fujion-script-jruby | |
|---|---|
| **Java Package** | **Description** |
| org.fujion.script.jruby | Support for Ruby-based scripts. |
| **artifactId: fujion-script-jython** | |
| **Java Package** | **Description** |
| org.fujion.script.jython | Support for Python-based scripts. |
| **artifactId: fujion-script-renjin** | |
| **Java Package** | **Description** |
| org.fujion.script.renjin | Support for R-based scripts. |
| **artifactId: fujion-codemirror** | |
| **Java Package** | **Description** |
| org.fujion.codemirror | Component wrapper for CodeMirror JS editor. |
| **artifactId: fujion-highcharts** | |
| **Java Package** | **Description** |
| org.fujion.highcharts | Component wrapper for HighCharts JS graphing library. |
| **artifactId: fujion-angular-core** | |
| **Java Package** | **Description** |
| org.fujion.angular | AngularJS integration support. |
| **artifactId: fujion-react-core** | |
| **Java Package** | **Description** |
| org.fujion.react | ReactJS integration support. |

## WebJars

Fujion can handle web resources, such as JavaScript and CSS files, and their dependencies in much the same way it handles Java resources. Following the WebJar specification, web resources may be packaged into Maven-enabled jar files. WebJars for many popular JavaScript libraries already exist and are published to Maven Central. Fujion uses this source for the third-party JavaScript libraries it requires. Fujion automatically detects WebJars on startup and configures SystemJS, the client JavaScript module loader used by Fujion, to be able to load these resources on demand. This greatly simplifies management of and access to web resources.

WebJars are available in three different flavors. Classic WebJars include information on how to configure the JavaScript module loader as a property within the embedded **`pom.xml`** file. NPM and Bower WebJars depend on an embedded **`package.json`** file to supply configuration information.

Occasionally, Fujion is unable to determine the correct SystemJS configuration for a WebJar (this is more common with NPM and Bower flavors). To circumvent this problem, Fujion will look for files called **`systemjs.config.json`** under the **`META-INF`** folder. These files contain JSON-formatted content that can be used to override incorrect SystemJS settings. Here is an actual **`systemjs.config.json`** file from the Fujion distribution as an example:

```
{
  "paths": {
    "balloon-css": "webjars/balloon-css/balloon.css",
    "jquery-ui-css": "webjars/jquery-ui/jquery-ui.css"
  }
}
```

Many of Fujion's artifacts are packaged as hybrid WebJars. These are jar files that contain both Java and JavaScript resources. Fujion's Maven build configuration supports the creation of classic and hybrid WebJars. A detailed description of how to leverage this capability to produce your own WebJars can be found in the section on custom component development.

## Fujion Server Pages

A Fujion Server Page (FSP) is simply an XML file containing FSP markup and ending with a **`.fsp`** extension. When the browser requests a FSP resource, Fujion will retrieve and compile the resource into a page definition. Compiled page definitions are cached, so compilation occurs only with the first request. From the page definition, Fujion may then materialize a component tree based on that definition. Finally, the component tree is attached to the main page, which invokes the page's synchronizer to reproduce the server-side component tree as a client-side widget tree. From that point, any changes to the server-side component tree are immediately reflected in the client.

A page definition is basically an in-memory representation of the FSP XML, with tags represented as a hierarchical arrangement of page elements. When a page definition is materialized, the materializer traverses the tree of page elements, creating the corresponding components, assigning attribute values to component properties (possibly evaluating embedded EL expressions), and establishing parent-child relationships.

While most FSP attributes correspond to properties on their respective component, some attributes may have special behaviors. For example, the **`controller`** attribute will bind its component to a backing controller, wiring fields and event handlers as directed by annotations on the controller class. The **`if`** and **`unless`** attributes control materialization of blocks of elements based on the evaluation of their associated expression. The **`impl`** attribute directs the materializer to use an alternate implementation when creating a component.

### Anatomy of a FSP

A FSP consists of one or more tags arranged in a hierarchical fashion. If there is more than one top level tag, the document must be enclosed in opening and closing **`fsp`** tags; otherwise the **`fsp`** tag is optional. Each tag can have zero or more attribute/value pairs. An attribute value can consist of the actual attribute value, or can contain embedded EL expressions that get evaluated at the time a page is materialized. An EL expression must be preceded by the **`${`** prefix and terminated with a **`}`**. See the section on EL expressions for more information. Some tags accept text content (i.e., non-markup text between the opening and closing tag). Fujion also recognizes special constructs called processing instructions. A processing instruction starts with **`<?`** and ends with **`?>`**, with one or more space-delimited key/value pairs in between. While they may resemble a tag declaration, they are not. For more information, see the section on processing instructions.

### Using XML Namespaces

Your FSP may contain Fujion XML namespace declarations on any tag (but typically these are placed on the top-level tag). Adding these declarations serves two purposes. First, they allow a schema-aware editor to perform validation and content-assistance on the FSP structure. Second, when namespace-prefixed attributes are used in a FSP, the namespace declaration for the prefix must be

declared. Here is an example of a FSP that declares a complete set of namespaces on a **`fsp`** tag:

```
<fsp xmlns="http://www.fujion.org/schema/fsp"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:attr="http://www.fujion.org/schema/fsp/attr"
                xmlns:on="http://www.fujion.org/schema/fsp/on"
                xsi:schemaLocation="
                        http://www.fujion.org/schema/fsp
                        http://www.fujion.org/schema/fsp.xsd">

                <div attr:myattribute="my attribute value">
                        <button on:click="self.setLabel('clicked!')" />
                </div>

</fsp>
```

Fujion recognizes the following namespaces:

| URI | Description |
|---|---|
| http://www.fujion.org/schema/fsp | **This is the schema namespace for a FSP. It is typically declared as the default namespace with a matching `schemaLocation` declaration (as in the above example).** |
| http://www.fujion.org/schema/fsp/attr | **Used for setting values on the component's attribute map.** |
| http://www.fujion.org/schema/fsp/on | **Used for attaching event handlers.** |

## How a FSP is Materialized

The materialization of a FSP in the web browser is a multistep process that starts with the browser requesting a web resource ending in a **`.fsp`** extension. On the server, this request is intercepted by a special resource transformer. The FSP resource transformer does several things. First, it creates an uninitialized page component (a Java class instance). Next it creates a minimal HTML document from a template (the bootstrap template). The bootstrap template is a skeletal HTML document with placeholders for various initialization parameters. The FSP resource transformer generates an HTML document using this bootstrap template, replacing placeholders with their corresponding parameter values. Among these parameter values is the id of the newly created page component. It is this minimal HTML document that is returned to the browser. Once loaded by the browser, JavaScript code is executed that passes the initialization parameters to bootstrap code. Among the operations the bootstrap code performs is the creation of a WebSocket connection back to the server. The browser then sends an initialization request to the server via the WebSocket connection. This request includes the id of the page being initialized. When the server receives a client request via the WebSocket, the request dispatcher selects a handler for that request based on its type. In this case, the initialization request handler receives the request, identifies the page being initialized, and completes the page initialization with information provided in the request. At this point the page component is fully initialized, but it has no children.

Next, the initialization handler, using the URL of the originally requested FSP web resource, compiles the FSP into a page definition (page definitions are cached, so compilation occurs only once per FSP). The page definition's materializer is then invoked. The materializer creates the server-side component tree that corresponds to the FSP markup. The component tree is then attached to the page. This triggers the page's synchronizer to translate the server's Java component tree into a corresponding JavaScript widget tree on the browser. This process occurs via a synchronization payload that is sent to the browser via the WebSocket connection. The synchronization payload consists of a series of JavaScript method invocations (client invocations) that direct the creation of the widget tree. At this point, the FSP is fully materialized and synchronized. Any further changes to the component tree will be communicated to the browser via the synchronizer. Events generated at the browser will be delivered to their corresponding server-side listeners via event-type client requests.

Once the WebSocket connection has been established, all communication from server to browser occurs via client invocations – parameterized method invocations on JavaScript objects (typically the widgets themselves, but not always). Communication from browser to server occurs via client requests. Each client request has a specific type that the request dispatcher uses to determine which request handler will process the request on the server. Fujion defines only a small number of client request types, with the event request type being the most frequently used by far. Implementers may define custom request types along with the Java handlers that will process them. That said, the event client request is extremely flexible and will accommodate most communication needs.

## Processing Instructions

Processing instructions are used to trigger specialized operations during FSP compilation or materialization. A processing instruction looks like this:

```
<?name attr1=val1 attr2=val2?>
```

Fujion recognizes the following processing instructions:

### Attribute Processing Instruction

This processing instruction has the form:

```
<?attribute name="{name}" value="{value}"?>
```

This processing instruction sets an attribute value on the component produced by the enclosing tag. For example:

```
<div>
  <?attribute name="myattr" value="myvalue" ?>
</div>
```

This assigns an attribute named **myattr** with the value **myvalue** in the attribute map of the enclosing **div** component.

### Import Processing Instruction

This processing instruction has the form:

```
<?import src="{fsp url}" prefix="{prefix}"?>
```

This processing instruction imports another FSP into the current one, adding all of its top-level elements as children to the enclosing tag. This differs from the **import** tag in that the latter creates its own component and the import operation occurs at materialization time. The **import** processing instruction does not create its own component and the import occurs at parsing time.

```
<div>
<?import src="mypage.fsp"?>
</div>
```

### Tag Library Processing Instruction

This processing instruction has the form:

```
<?taglib uri="{uri}" prefix="{prefix}"?>
```

This processing instruction associates a tag library (identified by its unique URI), with a prefix that can be used in EL expressions to reference functions exported by the tag library. The scope of this association is that of the enclosing tag and all of its

descendants. If the declaration occurs outside the top-level tag, the scope is the entire page. For example:

```
<?taglib uri="http://www.fujion.org/tld/test" prefix="mytaglib"?>

<div>${mytaglib.getLabel('test.message')}</div>
```

This invokes a function named **`getLabel`** on the specified tag library.

For a tag library to be referenced in this manner, it must be available on the application's class path. The URI serves only as a unique identifier; it is not used to download the resource. On startup, Fujion will scan the class path for tag library resources and index each by its unique URI.

## Expression Language (EL) Expressions

Expression Language (EL) is a simple scripting language originally developed for use in Java Server Pages, but now much more widely adopted. Within a FSP, EL expressions may occur inside an attribute value or inside a tag's text content. They are easily recognized by the **`${`** and **`}`** opening and closing delimiters, respectively. Fujion uses Spring Framework's EL parser, SpEL. A complete description of EL is beyond the scope of this document, but the aforementioned reference describes Spring's implementation in great detail. However, some important features merit mention here.

### Referencing Spring Beans

To reference a Spring bean in an EL, prefix its name with an **`@`** character. SpEL will automatically resolve the reference from the Spring application context. For example,

```
<div>${@fujion_ELEvaluator.class.name}</div>
```

would look up a Spring bean named **`fujion_ELEvaluator`** and return the name of its Java class.

### Internationalization

Fujion defines a Spring bean named **`message`** (its alias **`msg`** may also be used for greater brevity) that may be used to reference text that has been externalized in **`messages.properties`** files (see discussion on internationalization support). This bean is referenced like any other Spring bean, but is special in that any part of the reference that follows the bean name and a period delimiter is interpreted as an identifier for an externalized message value. For example,

```
<label label="${@msg.mylabel}"/>
```

would look up an externalized message named **`mylabel`**.

### Variable References

SpEL will resolve variable references relative to the context in which they are encountered. When resolving a variable reference, the following context elements are consulted in order:

1. Active tag library prefixes
2. The argument map passed to the materializer
3. The component's attribute map
4. The component's namespace
5. All ancestor component attribute maps

The special variable **`self`** refers to the component itself.

## Data Binding

Data binding is a convenient way to bind properties on a model object to component properties on a FSP. There are three binding flavors: **`read`**, **`write`**, and **`dual`**. For a **`read`** binding, whenever a property on the model object changes, the bound property on a component is automatically changed. For a **`write`** binding, whenever a component property changes, the bound model property is automatically changed. A **`dual`** binding exhibits both **`read`** and **`write`** behaviors; i.e., the bound properties are kept in sync.

Establishing a binding requires a binder – an object that implements the **`IBinder`** interface. The **`IBinder`** interface defines a set of methods for establishing each of the three types of bindings. Each method takes, at a minimum, the name of the model property that is to be bound, and returns a binding instance – an object that implements one or both of the **`IBinding`** sub-interfaces, **`IReadBinding`** and/or **`IWriteBinding`**. Both of the **`IBinding`** sub-interfaces include a method for invoking an update operation and one for initializing the binding with information about the component whose property is being bound. Fujion provides a binder implementation, **`GenericBinder`**, that will serve most needs.

Consider the following FSP page:

```
<fsp xmlns="http://www.fujion.org/schema/fsp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:attr="http://www.fujion.org/schema/fsp/attr">

  <toolbar attr:binder="${new org.fujion.model.GenericBinder(new MyModel())}">
    <textbox value="${binder.dual('color')}" synchronized="true"/>
    <label label="${binder.read('color')}"
      style="${binder.read('color', 'background-color: %s')}"/>
    <radiogroup>
      <radiobutton label="green"
        checked="${binder.dual('color', binder.lambda('green','equals'), binder.choice('green', binder.NOVALUE))}"/>
      <radiobutton label="red"
        checked="${binder.dual('color', binder.lambda('red','equals'), binder.choice('red', binder.NOVALUE))}"/>
    </radiogroup>
    <combobox readonly="true">
      <comboitem label="green"
        selected="${binder.dual('color', binder.lambda('green','equals'), binder.choice('green', binder.NOVALUE))}"/>
      <comboitem label="red"
        selected="${binder.dual('color', binder.lambda('red','equals'), binder.choice('red', binder.NOVALUE))}"/>
    </combobox>
  </toolbar>

</fsp>
```

and model object:

```
public class MyModel extends org.fujion.model.ObservableModel {

    private String color = "green";

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
        propertyChanged("color");
    }

}
```

**MyModel** extends the **ObservableModel** class, a convenience class that extends the **Observable** class, providing a **propertyChanged** method to signal that a property value has changed. It has a single property, **color**, with getter and setter methods. In the FSP page above, we create a **GenericBinder** instance, and pass it an instance of our model object. Because the model object is an **Observable**, the binder will automatically be notified when the it signals a property change. In the FSP, we bind several component properties to the **color** property of the model object:

| Component Property | Expression |
|---|---|
| textbox.value | binder.dual('color') |
| label.label | binder.read('color') |
| label.style | binder.read('color', 'background-color: %s') |
| radiobutton.selected<br>comboitem.selected | binder.dual('color', binder.lambda('green','equals'), binder.choice('green', binder.NOVALUE)) |
| radiobutton.selected<br>comboitem.selected | binder.dual('color', binder.lambda('red','equals'), binder.choice('red', binder.NOVALUE)) |

The **textbox.value** property is assigned a simple **dual** binding. Whenever the text box value changes, the model's **color** property will be changed to match. Conversely, whenever the model's **color** property changes, the text box value will be changed to match.

The **label.label** property is assigned a simple **read** binding. It will sync with any change to the model's **color** property. Because there is no way for the user to change the **label.label** property directly, a **read** binding makes sense.

A binding can include a conversion function. For a **read** binding, the conversion function takes the property value from the model object and converts it to another form before passing it to the bound component property. For a **write** binding, the conversion function converts a component property value before passing it to the bound model property. Not surprisingly, a **dual** binding may have two such conversion functions.

A conversion function may be a format string or a lambda function that takes a single parameter and returns the converted value. Because Spring EL does not currently support lambda syntax, the **GenericBinder** class provides a convenience function, **lambda**, that takes an object instance, the name of the function to invoke on that instance, and optional function parameters (called "curried" parameters). The **GenericBinder** also provides a conversion function, **choice**, which returns one of the specified parameters using the value being converted as a zero-based index. The value must be an enum, boolean or integer type, or a string value that can be converted to one of those types.

Returning to the above example, the **label.style** property is also assigned a **read** binding. It differs in that it also specifies a format string. The format string will be applied to the model's **color** property, converting it to a CSS style expression, before passing it to the **label.style** property.

The **radiobutton.selected** and **comboitem.selected** properties are assigned a more complex **dual** binding. These bindings specify two conversion functions: the first for the **read** binding and the second for the **write** binding. The **read** binding conversion functions use the **GenericBinder lambda** method to invoke the **equals** method on a string literal, returning the result of the comparison to the bound model property value. The **write** binding conversion functions use the **GenericBinder choice** method. Because the bound component properties (the **selected** property) are boolean values, the **choice** method will return the first choice if the value is true or the second choice if it is false. Note that the second choice is the constant **binder.NOVALUE**. This is a special value that, when returned by a conversion function, will result in no change to the bound property.

## Execution Context

Fujion uses WebSockets for bidirectional communication between the client and the server. There is one WebSocket connection per Fujion page. Most server-side WebSocket implementations use thread pooling to service communication requests. This means that one cannot predict what thread may service a given client request. How, then, does a server thread know which Fujion page a given client request is targeting? Fujion handles this by way of execution contexts. An execution context is a thread local variable that is set when a client request is received by the WebSocket handler and is passed to its request dispatcher. It contains information about the WebSocket session, the associated page, the servlet context, and the client request being serviced. Since every client request contains the id of the requesting page, the request dispatcher is able to create an execution context for that page. It is only after the request dispatcher has initialized the execution context that it hands off the client request to the request handler that is registered to service that request type. Thus, any logic that is executed during the servicing of a client request (i.e., in a request processing thread) has access to this execution context (through the **`ExecutionContext`** class). Once the request handler returns, the dispatcher clears the execution context and that thread becomes available to service other requests potentially on other WebSocket connections.

### Background Threads and Execution Context

Often, to avoid blocking user interaction, a lengthy operation must be performed in the background using a worker thread. However, worker threads present some challenges.

First, how can a worker thread access the execution context for a page? Because the execution context is not thread-inheritable, a worker thread will not have access to the execution context by default. However, if the thread knows the id of the page it wants to target, the **`ExecutionContext`** class has a static method, **`invoke`**, that takes a page id and an object that implements **`Runnable`**. This method will ensure that the **`Runnable`** is executed (synchronously) within the execution context of the specified page.

Second, since changes to a page's state should only take place on the request processing thread, how does a worker thread ensure that any changes occur only on that thread? This is best accomplished by having the worker thread post an event to the page's event queue. Posted (queued) events are guaranteed to be processed in a request processing thread.

## Controllers

In Fujion, a controller is a Java class that has logic that can respond to events and manipulate components for all or part of a FSP. For a controller to work, it must first be wired to the FSP that it is to control. This may be done by specifying a controller attribute on the target component or, as with many Fujion tasks, can be done programmatically by calling the target component's **`wireController`** method. The value of a controller attribute can be the name of the controller's Java class or an EL expression that resolves to an instance of the controller. In the case of the former, the named class must have a no-argument constructor. A controller instance will be created from this class when the FSP is materialized. For the latter, if the scope of the referenced bean is **`prototype`**, a new controller instance will be created each time the FSP is materialized, just as it is with the class name. However, if the scope is instead **`singleton`**, the same controller instance will be used each time the FSP is materialized. This may be a desired behavior, especially for a stateless controller (e.g., one that has only event handlers and no direct references to components). For a stateful controller (e.g., one that has field references that are annotated to be auto-wired), this will almost certainly be problematic.

Here are examples of the two approaches to attaching controllers:

```
<span controller="org.fujion.testharness.MenusController">

...

</span>
```

```
<span controller="${@myControllerBean}">

...

</span>
```

Finally, a controller may wish to be notified once it has been fully initialized. By implementing the **`IAutoWired`** interface, a controller will be notified via the callback method **`afterInititalized`** once the controller has been fully wired.

### Controller Annotations

Fujion uses annotations to direct how a controller is to be wired. There are two annotations for this purpose, one for wiring a controller's fields and one for wiring its event handlers.

The **`@WiredComponent`** annotation does as its name suggests – it wires a component instance to the field that it annotates. In its simplest form, the annotation may be specified with no parameters, using only default values. When specified, the parameters are:

@WiredComponent org.fujion.annotation.WiredComponent

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| value | String | Name of the annotated field is used. | The name property of the component instance to be injected into the field. A "/"-delimited namespace path may also be used. |
| overwrite | boolean | false | If false and the annotated field already has an assigned value, it is considered a wiring failure. If true, the existing value is overwritten. |
| onFailure | OnFailure | EXCEPTION | Indicates the action to be take if wiring of the field fails. Possible values are:<br>• IGNORE – ignore the failure.<br>• EXCEPTION – throw an exception.<br>• LOG – log the failure as a warning. |

As Fujion wires a controller to its target component, it first scans the controller's class and its super classes for **`@WiredController`** annotations. For each such annotation it encounters, it searches the namespace of the target component for a component whose **`name`** property matches the annotation's **`value`** argument and whose class is assignment-compatible with that of the annotated field. A wiring failure occurs if a matching component cannot be found, or if the annotated field already has an assigned value and the **`overwrite`** parameter is set to false. The action to be taken in the event of a wiring failure is determined by the **`onFailure`** parameter. The default action is to throw an exception. Note that fields of any scope (including private) may be annotated.

While the **`@WiredController`** annotation is a field-level annotation, the **`@EventHandler`** annotation is a method-level annotation. After Fujion has processed all **`@WiredController`** annotations, it scans for methods annotated with **`@EventHandler`**. Any method (of any scope) that takes no parameter or a single parameter that is assignment-compatible to Fujion's **`Event`** class may be annotated with **`@EventHandler`**. Using the annotation's parameter values, the annotated method is wrapped with an event listener that is registered to the specified target component(s) and event type(s). The **`@EventHandler`** annotation has one required and three optional parameters:

@EventHandler org.fujion.annotation.WiredComponent

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| value | String or String[] | none | The event type(s) to be handled. If the method is to be registered as a handler for more than one event type, the array syntax must be used. |
| target | String or String[] | self | The target component(s) whose events will be handled. A target may be specified as:<br>• The special value "self" represents the component to which the controller is wired.<br>• The value of the name property of the target component.<br>• The namespace path identifying target component.<br>• A value prefixed with an "@" character indicates that the value of the class field with that name should be used. The name may optionally be followed by a property name (e.g., "grid.rows"). |
| syncToClient | boolean | true | Determines whether the client should be notified of this handler. Normally, the default value of true should always be used. If it is known that a widget always sends an event of the desired type regardless of whether a server-side event handler is known to exist or if the widget is known to never produce a certain type of event, setting this to true can avoid unnecessary communication with the client. |
| onFailure | OnFailure | EXCEPTION | Indicates the action to be take if wiring of the event handler fails. Possible values are:<br>• IGNORE – ignore the failure.<br>• EXCEPTION – throw an exception.<br>• LOG – log the failure as a warning. |

A wiring failure can be the result of an invalid method signature, an unresolvable component name, or an unassigned field value. A single method can have multiple `@EventHandler` annotations if desired.

Here is an example of using controller annotations in their various forms:

```
// Wire field to component with same name, throwing an exception on failure.
@WiredComponent
private Grid grid;

// Wire field to component named cboxOtherName, throwing an exception on failure.
@WiredComponent("cboxOtherName")
private Combobox myCombobox;

// Wire field to component named myPopupBox, ignoring on failure.
@WiredComponent(value = "myPopupBox", onFailure = OnFailure.IGNORE)
private Popupbox popupbox;

// Wire handler for change events on component named anEditBox, logging a warning on failure.
@EventHandler(value = "change", target = "anEditBox", onFailure = OnFailure.LOG)
private void changeHandler() {
}

// Wire handler for click and dblclick events on the component to which this controller is being wired.
@EventHandler("dblclick")
@EventHandler("click")
private void testHandler(ClickEvent event) {
}

// Wire handler for myevent and anotherevent events on the component referenced in the field popupbox.
@EventHandler(value = {"myevent", "anotherevent"}, target = "@popupbox")
private void myEventHandler(Event event) {
}

// Wire handler for change events on the component referenced by the rows property of the value in the field grid.
@EventHandler(value = "change", target = "@grid.rows")
private void rowsChangeHandler(Event event) {
}
```

## Components: BaseComponent

All Fujion components share a common base class, **`BaseComponent`**, which provides certain behaviors common to all components.

### Lifecycle

When a component is first created, it has no corresponding client-side widget and no id. It isn't until a component is attached to a page that a client-side widget is created. It is also at this point that a component receives its unique id (a read-only property of every component). It is this id that links the component with its widget counterpart. It is also the id that is assigned to the top-level HTML element for the widget and, therefore, can be used as an element selector. Once a component is attached to a page, it can never be attached to a different page. The **`destroy`** method will remove a component's widget from the client (and all of its children as well). A destroyed component can no longer be re-attached to its page. Any method that attempts to manipulate the widget of a destroyed component will throw an exception.

| Method | Description |
|---|---|
| destroy | Detaches the component and destroys it and all of its children. |
| destroyChildren | Detaches and destroys all children of the component. |

### Containment

Containment refers to the parent-child relationship of components within a component tree. Fujion constrains these relationships by virtue of **`@Component`** class annotations. The **`@Component`** annotation defines the type (and number) of children a component may have as well as the type of parent. Some components aren't picky. For example, a **`Div`** component can have just about any type of parent or child. On the other hand, a **`Listitem`** component may only have a **`Listbox`** as a parent and may have no children. Violating a containment constraint produces an exception. This may occur during parsing of a FSP, or during execution of logic that manipulates the component tree.

The following is a summary of methods that relate to containment and are common to all components:

| Method | Description |
|---|---|
| addChild | Adds a child component. |
| addChildren | Adds multiple child components. |
| detach | Detaches a component from its parent. Same as setting the parent to null. |
| getAncestor | Searches up the component tree for an ancestor of a specified type. |
| getChild | Searches immediate children for one of a specified type. |
| getChildCount | Returns the number of children. |
| getChildren | Returns an immutable list of children. |
| getFirstChild | Returns the first child. |
| getIndex | Returns the position of this component relative to its siblings. |
| getLastChild | Returns the last child. |
| getNextSibling | Returns the next sibling. |

| | |
|---|---|
| **getPage** | **Returns the page to which this component belongs.** |
| **getParent** | **Return the parent component.** |
| **getPreviousSibling** | **Returns the previous sibling.** |
| **getRoot** | **Returns the root of the component tree.** |
| **isContainer** | **Returns true if the component may accept children.** |
| **setIndex** | **Sets the position of this component relative to its siblings.** |
| **setParent** | **Sets the parent of this component.** |
| **swapChildren** | **Swaps the position of two children.** |

### Events

There are a number of methods that relate to events. The section on events describes these in more detail.

| Method | Description |
|---|---|
| **addEventForward** | **Adds a forward directive.** |
| **addEventListener** | **Adds an event listener.** |
| **fireEvent** | **Fires an event to this component.** |
| **hasEventListener** | **Returns true if this component has a listener for the specified event type.** |
| **notifyAncestors** | **Fires an event to ancestors of this component.** |
| **notifyDescendants** | **Fires an event to descendants of this component.** |
| **removeEventForward** | **Removes a forward directive.** |
| **removeEventListener** | **Removes an event listener.** |

### Data Linkage

Every component has two ways of linking to arbitrary data: an attribute map and a data property.

| Method | Description |
|---|---|
| **findAttribute** | **Finds the named attribute by searching up the component tree.** |
| **getAttribute** | **Returns the value of an attribute.** |
| **getAttributes** | **Returns the attribute map.** |
| **getData** | **Returns the data object associated with this component.** |
| **hasAttribute** | **Returns true if the specified attribute exists.** |
| **removeAttribute** | **Removes an attribute.** |
| **setAttribute** | **Sets an attribute.** |
| **setData** | **Sets the data object associated with this component.** |

### Namespace

Every component has a **`name`** property that can be used to identify the component. The **`name`** property is used when wiring a controller's fields and event handlers, for example. But what happens when two different component instances in the same component tree have the same name? Fujion allows for this, but in a very constrained way through the concept of namespaces. A namespace is a defined boundary within which all component names must be unique. So how is a namespace declared? Some component types are namespaces by virtue of the fact that they implement the marker interface **`INamespace`**. The **`Window`** and **`Namespace`** components are examples of this. In addition, any component in a FSP

may be declared a namespace by setting its **`namespace`** attribute to true. (For a component that is a namespace by default, you could set its namespace attribute to false to override.) Finally, a component tree's root component will always act as a namespace boundary whether or not it is declared so explicitly.

The boundary of a namespace extends to all components within the subtree rooted below the namespaced component. The exception to this is nested namespaces. A nested namespace is isolated from its enclosing namespace. So, more precisely stated, any component may have a name as long as that name is unique within its immediately enclosing namespace. Whenever a named component is attached to a parent, Fujion validates that its name does not violate this constraint, throwing an exception if it does.

The following methods pertain to namespaces:

| Method | Description |
|---|---|
| **findByName** | **Looks up a component by name within this component's namespace.** |
| **getName** | **Returns the name of this component. Must be unique within its namespace.** |
| **getNamespace** | **Returns the component that serves as the namespace for this component.** |
| **isNamespace** | **Returns true if this component declares a namespace.** |
| **setName** | **Sets the name for this component. Must be unique within its namespace.** |

The **`findByName`** method looks up a component by its name. The name may be qualified by a "/"-delimited path specification. The look up always starts relative to the component whose **`findByName`** method was invoked. Consider the component tree that would be created by the following FSP:

```
<window name="myouter">
  <label name="mycomp" />
  <div name="myinner" namespace="true">
    <button name="mycomp" />
    <span name="refcomp">
      <toolbar name="mycomp2" />
    </span>
  </div>
</window>
```

Here we have an example of a nested namespace. The outermost namespace is defined by the **`window`** component (which is a namespace by default). The innermost namespace is the **`div`** component, which is explicitly declared a namespace. There are two components with the same name (**`mycomp`**), but in different namespaces. The **`span`** component (named, **`refcomp`**) will serve as our reference component (i.e., the component whose **`findByName`** method is invoked). Now consider the different path inputs below:

| Path | Finds | Description |
|---|---|---|

| | | |
|---|---|---|
| **mycomp** | **button** | **Searches in the immediately enclosing namespace (declared by the div component).** |
| **^/mycomp** | **label** | **Searches one level up from the immediately enclosing namespace.** |
| **mycomp2** | **toolbar** | **Searches in the immediately enclosing namespace (declared by the div component).** |
| **^/mycomp2** | **<nothing>** | **Searches one level up from the immediately enclosing namespace.** |
| **^/myinner/mycomp2** | **toolbar** | **Searches for "myinner" one level up from the immediately enclosing namespace, then for "mycomp2" in the namespace declared by "myinner".** |

## Widget

Every server-side Java component has a corresponding client-side JavaScript widget with which it communicates. A detailed description of widget structure and function may be found in the section on writing custom components. The following component methods are used to interact with the client-side widget:

| Method | Description |
|---|---|
| **invoke** | **Invokes a method on the client-side widget, with optional arguments.** |
| **invokeIfAttached** | **Same as invoke, but only if the component is attached to a page.** |

## Controllers

For a detailed description of controllers, see the separate discussion on this topic. While the `WiredComponentScanner` and `EventHandlerScanner` annotation scanning classes have static methods for wiring controllers, it is recommended using the component's `wireController` method to do this. This has the advantage that the component is able to track the controllers that have been wired to it.

| Method | Description |
|---|---|
| **getControllers** | **Returns an immutable list of controllers that have been wired to this component, in the order they were wired.** |
| **getController** | **Returns the last controller wired, if any.** |
| **wireController** | **Wires a controller to the component.** |

## Components: BaseUIComponent

The class **`BaseUIComponent`** class is a specialization of **`BaseComponent`** that includes behaviors unique to visual components.

### CSS Classes

Several methods allow CSS classes to be added, removed, or replaced. CSS classes are specified in the same manner as HTML, but with a couple of special extensions. First, a class name preceded by a "**-**" character means that class will be removed. This is especially useful when setting class values in a FSP where you might want to remove an existing class. Second, Fujion supports the concept of class groups. Classes belonging to the same group are mutually exclusive – adding one will remove any existing classes belonging to the same group. Groups are specified as a prefix to the class name, delimited by a colon. A group name may be any arbitrary value. Fujion uses two group names by convention. The group name **`flavor`** is used for the mutually exclusive Bootstrap classes that have suffixes like **`-danger`**, **`-success`**, or **`-default`**. The group name **`size`** is similarly used for the Bootstrap classes that have suffixes like **`-sm`**, **`-xs`**, or **`-lg`**. So, for example, the CSS class string

```
flavor:btn-danger -myoldclass mynewclass
```

would result in the addition of the **`btn-danger`** class (removing any existing classes in the **`flavor`** group), removal of the **`myoldclass`**, and the addition of **`mynewclass`**.

Methods relating to CSS classes include:

| Method | Description |
|---|---|
| addClass | Adds one or more classes. Existing classes are left as is except where group names or "-" prefixes result in class removal, as noted above. |
| getClasses | Returns the current CSS classes for the component. |
| removeClass | Removes one or more classes. |
| setClasses | Replaces all existing CSS classes with those specified. |
| toggleClass | Toggles between one of two classes, depending on a conditional value. |

### CSS Styles

CSS styles may be added or removed individually, or replaced entirely. In addition, several commonly used styles have corresponding properties that serve as shortcuts. In such cases, setting the style via its shortcut property or setting it via one of the general style methods has the same effect.

| Method | Description |
|---|---|
| addStyle | Adds a single style, replacing any existing value for that style. |
| addStyles | Adds multiple styles, specified in the same format as the HTML style attribute. |
| getCss | Returns the inline CSS associated with this component. |
| getFlex | A shortcut for retrieving the flex style setting. |
| getHeight | A shortcut for retrieving the height style setting. |
| getStyle | Returns the value of the named style. |

| | |
|---|---|
| **getStyles** | **Returns a space-delimited list of all current styles.** |
| **getWidth** | **A shortcut for retrieving the width style setting.** |
| **removeStyle** | **Removes the setting for a single style.** |
| **setCss** | **Sets the inline CSS associated with this component.** |
| **setFlex** | **A shortcut for setting the flex style.** |
| **setHeight** | **A shortcut for setting the height style.** |
| **setStyles** | **Replaces all existing styles. Specified in the same format as the HTML style attribute.** |
| **setWidth** | **A shortcut for setting the width style.** |

## Visibility

Component visibility may be set using CSS classes or styles. However, in general, you should use the **`visible`** property to control visibility. This is the only reliable way to track a component's visibility. There are several methods related to component visibility:

| Method | Description |
|---|---|
| **getFirstVisibleChild** | **Returns the first visible child.** |
| **hide** | **A shortcut for setting the visible property to false.** |
| **isVisible** | **Returns true if the component is visible.** |
| **setVisible** | **Sets the value of the visible property.** |
| **show** | **A shortcut for setting the visible property to true.** |

## Drag and Drop

Fujion provides event-based drag-and-drop support, based on the concept of drag ids and drop ids. A target component will accept a dropped component only if one its drop ids matches one of the dragged component's drag ids. Drop ids are set for target components using the **`dropid`** property while those for dragged components are set using the **`dragid`** property. Both properties can accept multiple ids by separating them with a space. An id of **`*`** will match any other id. When a dragged component is successfully dropped on a target component, the target component receives a **`DropEvent`** event. The drop event's **`getDraggable`** method can be used to retrieve a reference to the dragged component.

| Method | Description |
|---|---|
| **getDragid** | **Returns the drag id(s).** |
| **getDropid** | **Returns the drop id(s).** |
| **setDragid** | **Sets the drag id(s).** |
| **setDropid** | **Sets the drop id(s).** |

## Focus

Although all **`BaseUIComponent`** subclasses have methods pertaining to input focus, not all can receive focus. For those that cannot, these methods will have no affect.

| Method | Description |
|---|---|
| **focus** | **A shortcut for setting the focus to true.** |

| | |
|---|---|
| **getTabindex** | **Determines the tab order for receiving focus.** |
| **setFocus** | **Sets or removes focus.** |
| **setTabindex** | **Sets the tab order for receiving focus.** |

## Masks

Masks are semi-transparent overlays that prevent user interaction with a component and its children. A mask may have a caption and an associated context menu.

| Method | Description |
|---|---|
| **addMask** | **Adds a mask overlay to the component with optional caption and context menu.** |
| **removeMask** | **Removes any existing mask overlay.** |

## Keyboard Shortcuts

The Fujion Framework provides native support for keyboard shortcuts. It does this by allowing a UI component to specify which key combinations it wishes to intercept (via the **`keycapture`** property, using spaces to separate multiple entries). When one of the specified key combinations is detected, a **`KeycaptureEvent`** is fired. By providing a handler to process this event, the program can execute special logic when the shortcut key combination is typed.

To specify a key combination to intercept, you first specify zero or more key modifiers:

- ^ The control key
- @ The alt key
- ~ The meta key (command key on Macs, Windows key on PCs)
- $ The shift key

The key modifiers are followed by the key code. A key code may be specified in one of two ways. The first is to express the key code as its numeric value by preceding the numeric value with a **`#`**. The second way is to use the key's mnemonic identifier (case insensitive), with or without the **`VK_`** prefix. A reference of key code values and their mnemonic identifiers can be found by inspecting the **`KeyCode`** enumeration. The following are examples of valid key capture values:

| Value | Description |
|---|---|
| **^A** | **The control and "A" keys were depressed.** |
| **^#65** | **Same as ^A.** |
| **F1** | **The first function key was depressed.** |
| **VK_F1** | **Same as F1.** |
| **~0** | **The meta and "0" keys were depressed.** |
| **@J** | **The alt and "J" keys were depressed.** |
| **$C** | **The shift and "C" keys were depressed.** |
| **^$F3** | **The control, shift, and third function keys were depressed.** |

# Events

Fujion relies heavily on events. Events are the primary mechanism by which the client communicates with the server. Events can be used to defer execution of a piece of programming logic. Events can be used to decouple an action from its trigger. Events can be used to communicate between worker threads and the primary request handling thread. All Fujion events descend from the **`org.fujion.event.Event`** class and most have specializations of that class. Like their JavaScript counterparts, events target a component and are delivered to handlers registered on that component. Events targeting a widget on the client will be propagated to their server-side counterpart if it has any registered listeners for that event type. Events may also be propagated by programmatically firing them at a target component. The mechanism for delivering them to their handlers is the same.

## Client-side Events

An event triggered on a client-side widget will be forwarded to the corresponding server-side component if that component has at least one registered listener for that event type, or the widget is specifically configured to always send the event. When an event is forwarded, it is serialized into a client request of type **`event`** and transmitted over the WebSocket connection. When the server receives the request, it passes it to the request handler that is registered to handle events. The handler deserializes the event and, based on its type, attempts to match it to an event class that is mapped to that type (via the **`@EventType`** class annotation). If no such a class is found, the **`Event`** class itself is used. The handler creates an instance of the event class and, directed by **`@EventParameter`** field annotations, injects parameter values into the instance from those found in the request. Among those injected values is that of the target component, which is necessary to successfully deliver the event to its registered handlers. Finally, the request handler fires the event to the target component's event listeners.

## Server-side Events

Events originating from the server are created programmatically. Once an application creates an event, it may fire that event to its target component by directly invoking the target's **`fireEvent`** method, or by using one of the utility methods in the **`EventUtil`** class.

## Sending vs Posting Events

An event that is fired via a send operation (the default) is immediately delivered to its listeners before control is returned to the caller. An alternative is to post the event. A posted event is placed on the current page's event queue. Control returns to the caller immediately. Posted events are delivered when the active request handler finishes processing the current request. This means that posted events are only delivered by the request handling thread. So, it is safe for worker threads to post events, but not necessarily to send them (depending on what their listeners do). Note that if a worker thread posts an event in this way, it is possible that the

event might not get processed for a long time, depending on the activity of the page's request handler. In other words, if a worker thread posts an event and there is a long period where no requests are sent by the client, there is no opportunity for the page's event queue to be processed. To avoid this scenario, Fujion will automatically send a ping request to the client when an event is queued by a worker thread. The response to the ping request will initiate a client request cycle on the server which, on completion, will cause the page's event queue to be processed.

### Registering Event Listeners

An event listener may be declared in a number of ways. Regardless of how it is declared, the resulting event listener is registered via one of the target component's **`addEventListener`** methods. Every event listener must implement the **`IEventListener`** interface, which declares a single **`onEvent`** method for delivering the event.

#### Registering a Listener Programmatically

The simplest way to register an event listener is to create an object instance that implements the **`IEventListener`** interface, providing the event handling logic in the implementation of the **`onEvent`** method, and then registering it using the target component's **`addEventListener`** method. Java 8 lambda expressions provide a particularly concise syntax for this. For example,

```
menuPopup.addEventListener("open", event -> {
   onOpen(event);
});
```

An event listener added programmatically may later be removed by calling the **`removeEventListener`** method.

#### Registering a Listener Using Annotations

Perhaps the most common way of registering an event listener is via the **`@EventHandler`** method annotation. When an instance of a class containing such an annotation is wired to a component (see the section on controllers), an event listener that invokes the annotated method is created and automatically registered to the specified event(s).

#### Registering a Listener Declaratively

Finally, a listener may be registered declaratively in a FSP using the **`on:XXXX`** attribute syntax, where **`XXXX`** is the name of the event. The value of the attribute may be raw Groovy script or an EL expression that resolves to either a script component or an object implementing the **`IEventListener`** interface. For example,

```
<button on:click="self.addStyle('color','green');" />
```

Because this method uses a namespaced attribute, you must declare the namespace prefix in your FSP. This is typically declared on the top-level tag, but can done on

anywhere in the tag hierarchy at or above where it is referenced. See the section on Fujion Server Pages to see how to declare namespaces.

### Forwarding Events

An event may be forwarded to a different target, possibly as a different event type. Fujion forwards an event by creating a listener for that event whose **`onEvent`** logic dispatches the event (or a different event) to its forwarding target. If the original event and the forwarded event types are the same, the original event is forwarded to the target. If they are different, Fujion creates a special **`ForwardedEvent`** type that wraps the original.

#### Forwarding an Event Programmatically

The **`BaseComponent`** class has the complementary methods **`addEventForward`** and **`removeEventForward`** for adding and removing event forwarding directives.

#### Forwarding an Event Declaratively

A forward directive may be registered declaratively in a FSP using the **`forward`** attribute. For example,

```
<listitem forward="dblclick=btnAdd.click" />
```

Here, the list item's double-click event will be forwarded as a click event to the component named **`btnAdd`**. The target component may be specified using namespace path syntax, as in this example. If a target component is not specified, it is assumed to be the same component to which the forward directive is registered. Multiple forward directives may be specified by separating them with a space.

## Internationalization

Fujion leverages Spring Framework's message source paradigm to provide support for multiple languages and dialects. This paradigm uses specially named property files to store locale-specific text indexed by unique identifiers. On startup, Fujion scans for files matching the pattern **`messages*.properties`** in the **`WEB-INF`** folder or anywhere on the class path. The suffix on the filename (i.e., the part that matches the **`*`** wildcard), identifies the locale of the file's contents. Each component of the locale is delimited by an underscore character. For example, a file named **`messages_en_US.properties`** would contain text entries written in the U.S. dialect of English. When retrieving locale-specific text given its unique identifier, Fujion will search for a corresponding property file whose suffix most closely matches the user's current locale. This search goes from specific to general, so a property file with no suffix would be matched if all other attempts fail.

The following is an example of such a property file:

```
example.message1=This is a test.
example.message2=This contains a reference: ${example.message1}
example.message3=This contains a recursive reference: ${example.message2}
example.message4=This contains a line continuation, \
an escaped character: \n and a replaceable parameter: {0}.
```

The identifier (the part to the left of the "**=**") must uniquely identify the associated text, so using namespaces is essential practice to avoid naming collisions. Note that you may include references to other entries using the **`${<text id>}`** syntax. Fujion safeguards against circular references that might arise. You may continue long lines by ending a line with a backslash character, escape special characters, and include placeholders for replaceable parameters.

Having created externalized text in this way, how then do you reference it? In a FSP, one can use an EL expression (see that section) and the special bean named **`message`** (or its shorter alias, **`msg`**). For example,

```
<label label="${@msg.example.message1}" />
```

would return the locale-specific value of the text identified by **`example.message1`**.

To retrieve externalized text programmatically, you may use the static method **`StrUtil.getLabel`**. For example,

```
StrUtil.getLabel("example.message4", "myparameter");
```

## Themes

Fujion makes heavy use of Bootstrap for styling its components. Many alternatives to Bootstrap's default theme are packaged as WebJars that can be included as Maven dependencies in your web application. Fujion automatically includes the WebJar containing the default Bootstrap theme and sets it as the default theme. You may override this behavior by providing an alternate style sheet (this is true of any style sheet, not just Bootstrap ones). Fujion employs URL rewriting to substitute one theme element for another. For each theme that you create, you must define the rules for rewriting URL's so that your theme resources are loaded instead of those of the default theme.

To create your own theme, you should declare a bean of type **`org.fujion.theme.Theme`** in your Spring configuration file. The theme registry will automatically find and register beans of this type. You must specify a theme name and one or more rules for mapping a source resource path to the equivalent path within your theme.

The following is an excerpt of a Spring configuration file that defines a theme for one of the publicly available Bootstrap theme alternatives:

```
<bean class="org.fujion.theme.Theme">
  <constructor-arg value="cosmo" />
    <property name="mappings">
      <map>
        <entry key="/webjars/bootstrap/**" value="/webjars/bootswatch-$0/$1" />
      </map>
    </property>
</bean>
```

The theme bean takes a single constructor argument, the theme name. This name uniquely identifies the theme. We also set the value for a property called **`mappings`**, which is a map of URL translation rules. You may potentially have multiple entries for this property. For each entry, the key is a URL pattern against which a requested resource will be matched. The pattern may be a glob style pattern (shown here) or a regular expression (which must begin and end with a **`^`** and **`$`**, respectively). If the pattern matches a requested resource, it is converted to its new form using the formatting template specified as the value of the map entry. The value may have replaceable parameters, recognized as a number preceded by a **`$`**. The value **`$0`** will be replaced by the theme's name. All other replaceable parameters represent the part of the path that was matched by a wildcard. So, **`$1`** would represent the first wildcard match, **`$2`** the second, and so on. In the example above, the following match and translation would occur:

/webjars/bootstrap/css/bootstrap.css → /webjars/bootswatch-cosmo/css/bootstrap.css

The highlighted portion represents the fragment that was matched by the **`**`** wildcard and thus would be represented by the **`$1`** parameter.

If theme bean with the same name as an existing theme is found, its URL mappings will be merged with the original. This allows you to contribute URL mappings to an existing theme. So, for example, if you are using a stylesheet that you want to be part of an existing theme, you may declare a bean with the name of the theme you want to use and provide one or more URL mappings to substitute your default stylesheet with the themed version.

So, how does Fujion know which alternate theme to use? It uses a theme resolver that first looks for a theme name specified as a query parameter in the original request URL. For example:

**`http://acme.org/index.fsp?theme=cosmo`**

Failing that, it looks for a theme name stored as a session cookie under the name specified by the constant **`CookieThemeResolver.DEFAULT_COOKIE_NAME`**. If a theme is found in either location, that theme's URL mappings will be applied to all resource requests.

The default theme name, if no other is specified, is **`default`**. If you have any resources that are to be associated with a theme, you must define a theme bean for the **`default`** theme. This looks like any other theme bean declaration except that the formatting template (the value portion of the map entry) is an empty string. The theme bean declaration for the default bootstrap theme looks like this:

```
<bean class="org.fujion.theme.Theme">
  <constructor-arg value="default" />
    <property name="mappings">
      <map>
        <entry key="/webjars/bootstrap/**" value="" />
      </map>
    </property>
</bean>
```

The purpose of a default declaration like this is identify to the theme manager those resources that should not be cached by the browser. Because browsers like to cache web resources once they are retrieved, the theme manager must tell the browser which resources it should not cache. Otherwise, a theme change would have no effect for those resources already cached and residing on the client.

# Spring Framework Integration and Extensions

The Fujion Framework relies heavily on the Spring Framework for managing component lifecycle, scope and interdependencies. This section describes some of the more important uses.

## Spring Configuration Files

The Fujion Framework will automatically locate Spring configuration files and process them. For this to work properly, it does impose some constraints on how you name these files and where they are located. First, all Spring configuration files must be located in the **`META-INF`** folder of the jar file. This is the same location where the jar's manifest file is located. Second, configuration files must end in **`-spring.xml`** in order to be discovered by Spring. How you prefix these file names is up to you, but we recommend that you use a mnemonic identifier that is not likely to be used elsewhere. Naming collisions for configuration files are only problematic in certain development environments, but should generally be avoided when possible. For example, the configuration file for a project might be named **`myproject-spring.xml`** and would be placed in the **`META-INF`** folder of that project's jar file.

## Spring Namespace Extensions

The Fujion Framework provides two Spring namespace extensions that simplify scanning Java classes for certain Fujion annotations. The first is for discovering **`@Component`** class annotations and the second for **`@EventType`** class annotations. Both have similar syntax in that you may specify either a class or a package name to be scanned. In addition, package names (but not class names) may contain wildcard characters. Consider the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:ffc="http://www.fujion.org/schema/component"
            xmlns:ffe="http://www.fujion.org/schema/eventtype"
            xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.fujion.org/schema/component
    http://www.fujion.org/schema/component-extensions.xsd
    http://www.fujion.org/schema/eventtype
    http://www.fujion.org/schema/eventtype-extensions.xsd">

            <ffc:component-scan package="org.acme.*" />

            <ffe:eventtype-scan class="org.acme.MyNewEvent" />

</beans>
```

In the above example, we are requesting a scan for **`@Component`** annotations for all packages that begin with "**`org.acme.`**" and a scan for **`@EventType`** annotations for the class "**`org.acme.MyNewEvent`**".

### Spring Page Scope

The Spring Framework has the notion of scopes, declared as the **`scope`** attribute on a bean definition. The most commonly used scopes are **`singleton`** (the default), where each reference to the bean returns the same instance, and **`prototype`**, where each reference to the bean returns a new instance. Spring defines additional scopes (**`request`**, **`session`**, and **`globalSession`**) that pertain to traditional web applications. While those scopes may be used with Fujion, they are of little utility since Fujion predominantly uses WebSockets for client-server communication rather than HTTP requests. Fujion, therefore, defines a new scope called **`page`**. The **`page`** scope is limited to the current execution context. In other words, a bean with this scope exists only within the execution context in which it was created. Repeated references to that bean from within the same execution context will return the same instance. References to that bean from another execution context will return a different instance. Finally, beans created in the **`page`** scope are destroyed when the execution context is destroyed. Note that, like all the web-specific scopes, you must create a scoped proxy for this to work properly. Consider the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
                http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop.xsd">

        <bean id="myPageScopedBean" class="org.acme.MyBean" scope="page">
                <aop:scoped-proxy />
        </bean>

        <bean id="mySingletonScopedBean" class="org.acme.MyOtherBean">
                <property name="myProperty" ref="myPageScopedBean"/>
        </bean>

</beans>
```

In the above example, we declare a page-scoped bean named **`myPageScopedBean`**. Note the **`aop:scoped-proxy`** declaration within the bean's definition. This is essential for the scope to behave as expected. The proxy intercepts method calls to the proxied bean and redirects them to the correct bean instance for the active execution context. In this way, a bean that references the page-scoped bean (the **`mySingletonScopedBean`** in this example) actually gets a reference to the proxy bean which transparently handles resolving the page scope when a method gets invoked. Under most circumstances, the referencing bean has no awareness that it is working with a proxied bean.

## Property Files

Fujion uses Spring Framework's support for loading and exposing properties from external files. On startup, Fujion will always look for property files located in a jar file's **`META-INF`** folder matching the naming pattern **`*-default.properties`**. This allows you to 1) make explicit the properties that your code recognizes, and 2) to supply default values for those properties. After loading all default properties, Fujion will then search the class path for property files matching the pattern **`fujion*.properties`**. (*Note*: This location and naming pattern may be overridden. See the section on configuration to see how this is done.) Because these properties are loaded after the default ones, any property whose name matches one that has already been assigned a default value will be overwritten by the new value.

*Note*: It is strongly recommended that you carefully namespace any properties that you define. This will avoid possible naming collisions. For example, all Fujion-specific properties begin with **`org.fujion`**.

Once defined, properties may be referenced in a Spring configuration file using the standard Spring syntax: **`${<property name>}`**. By default, Spring is configured to throw an exception if a referenced property has not been defined.

# Configuration

## Configuration Properties

See the previous discussion on property files for information on how Fujion locates property files. Property values may also be specified as system properties (how this is done depends on your web server implementation).

The following properties affect various aspects of Fujion's operation:

| Property | Default Value | Description |
|---|---|---|
| org.fujion.location .properties | **classpath:fujion*.properties** | **The path for locating property files. This may only be overridden with a system property.** |
| org.fujion.logging .debug | **NONE** | **Sets the client-side logging behavior for debug level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.logging .error | **BOTH** | **Sets the client-side logging behavior for error level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.logging .fatal | **BOTH** | **Sets the client-side logging behavior for fatal level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.logging .info | **NONE** | **Sets the client-side logging behavior for info level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.logging .trace | **NONE** | **Sets the client-side logging behavior for trace level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.logging .warn | **NONE** | **Sets the client-side logging behavior for warn level. Possible values are: NONE, CLIENT, SERVER, BOTH** |
| org.fujion.websocket .asyncSendTimeout | **0** | **Maximum time, in milliseconds, to wait for a send operation to succeed.** |
| org.fujion.websocket .maxSessionIdleTimeout | **0** | **Maximum idle time, in milliseconds, after which the socket session will be closed.** |
| org.fujion.websocket .maxBinaryMessageBufferSize | **32768** | **The maximum size, in bytes, for the binary message buffer.** |
| org.fujion.websocket .maxTextMessageBufferSize | **32768** | **The maximum size, in bytes, for the text message buffer.** |
| org.fujion.websocket .keepaliveInterval | **0** | **The keep-alive interval, in milliseconds. The client will transmit a ping request when no transmission has occurred during this interval. A value of 0 inactivates this feature.** |

## Web Application Configuration

The **`web.xml`** file found in the **`WEB-INF`** folder of the web application controls various behaviors of the web application. Fujion requires some additional entries to be added to the default version of this file:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <async-supported>true</async-supported>
  <load-on-startup>1</load-on-startup>

  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:/META-INF/fujion-dispatcher-servlet.xml
      classpath*:/META-INF/*-spring.xml
    </param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>

<context-param>
  <param-name>fujion.debug</param-name>
  <param-value>true</param-value>
</context-param>
```

The **`fujion.debug`** parameter is optional. Set it to true to facilitate debugging. The default value is false and should always be used in a production setting. This parameter may also be set as a system property. Doing so overrides any setting specified in the **`web.xml`** file. Enabling debugging affects a number of behaviors:

- Disables minification of web resources.
- Prevents clearing the web page when a socket connection is interrupted.
- Enables more detailed logging on the client console.

While some web applications recognize a **`<welcome-file-list>`** section in the **`web.xml`** file for specifying a default home page, Fujion simply looks for a file with the name **`index`** (any extension) at the root of the web application and loads that resource as the default home page. If more than one file named **`index`** is present, Fujion will pick the first one it encounters.

## AngularJS Integration

Fujion includes full support for AngularJS version 4 in the form of a container component that handles bootstrapping of the AngularJS component and exposes its public methods for remote invocation. AngularJS support is not included as part of the core Fujion library. To include it, specify the **`fujion-angular-core`** artifact id among your Maven dependencies. The typical AngularJS component manages its own bootstrapping and hardcodes the DOM anchor point (the **`selector`**, in AngularJS parlance) as part of the component's metadata. A consequence of this design is that a top-level AngularJS component can only be a singleton instance. Fujion solves this problem by taking over the task of bootstrapping the AngularJS component. Rather than hardcoding the DOM anchor point, Fujion's AngularJS bootstrap code determines the anchor point dynamically, setting it to the container widget.

The Fujion distribution includes a sample AngularJS component, an implementation of the classic Pomodoro timer app, packaged as a WebJar. The FSP page that displays the Pomodoro timer app looks like this:

```
<page>
  <angular src="fujion-angular-pomodoro" />
</page>
```

The **`angular`** tag represents the AngularJS container component, the Java class **`AngularComponent`**. Its **`src`** attribute specifies the name of the SystemJS module that contains the top-level AngularJS component. Since the Pomodoro timer app is packaged as a classic WebJar, we can examine its embedded **`pom.xml`** file to see the SystemJS settings for our SystemJS module. The relevant section looks like this:

```
<properties>
  <systemjs>
    'map': {'fujion-angular-pomodoro': 'pomodoro.js'}
  </systemjs>
  ⋮
</properties>
```

Here we see the module name, **`fujion-angular-pomodoro`** and the path to the main JavaScript resource, **`pomodoro.js`**. The latter is transpiled from the TypeScript resource, **`pomodoro.ts`**. Examining this file, we notice the absence of any bootstrap code (Fujion will handle this for us) and of a **`selector`** in the **`@Component`** decorator (the bootstrap code will set this to the AngularJS container) and the presence of an export directive:

```
export { PomodoroComponent as AngularComponent };
```

The Fujion AngularJS bootstrapper will examine the loaded module for one of the following exports:

- The top-level component class (i.e., the one to be bootstrapped), exported as **`AngularComponent`**. You may optionally also export an **`ngModule`** object to supplement or override the defaults.
- The application module (i.e., the class decorated with **`@NgModule`**) exported as **`AngularModule`**. The component to be bootstrapped must be specified in the **`@NgModule`** decoration.
- An **`ngModule`** object only. As with the previous option, the component to be bootstrapped must be specified in the **`@NgModule`** decoration.

This is essentially all that is required to host an AngularJS component in Fujion's AngularJS container. But what if we want to invoke a method on our hosted AngularJS component? The AngularJS container Java class, **`AngularComponent`**, provides a method called **`ngInvoke`**, which can be used to invoke any published method on your AngularJS component. This is similar to the **`BaseComponent.invoke`** method except that it invokes a method on the hosted AngularJS component, not on container itself. **`ngInvoke`** can even be called before the hosted AngularJS component is fully bootstrapped. This is important since the bootstrapping process is asynchronous. If the hosted component is not fully initialized, calls to **`ngInvoke`** will be queued until initialization is complete.

## React Integration

Fujion includes full support for React version 15 in the form of a container component that acts as an anchor point for the top-level React component. The React container handles the initialization and rendering of the top-level React component and exposes its public methods for remote invocation.

React support is not included as part of the core Fujion library. To include it, specify the **`fujion-react-core`** artifact id among your Maven dependencies.

The Fujion distribution includes a sample React component, an implementation of the classic Pomodoro timer app, packaged as a WebJar. The FSP page that displays the Pomodoro timer app looks like this:

```
<page>
  <react src="fujion-react-pomodoro" />
</page>
```

The **`react`** tag represents the React container component, the Java class **`ReactComponent`**. Its **`src`** attribute specifies the name of the SystemJS module that contains the top-level React component. Since the Pomodoro timer app is packaged as a classic WebJar, we can examine its embedded **`pom.xml`** file to see the SystemJS settings for our SystemJS module. The relevant section looks like this:

```
<properties>
  <systemjs>
    'map': {'fujion-react-pomodoro': 'pomodoro.js'}
  </systemjs>
  ⋮
</properties>
```

Here we see the module name, **`fujion-react-pomodoro`** and the path to the main JavaScript resource, **`pomodoro.js`**. The latter is transpiled from the ES6 resource, **`pomodoro.es6.js`**. Examining this file, we notice the absence of code that instantiates and attaches our top-level component, **`PomodoroComponent`** (Fujion will handle this for us) and the presence of an export directive:

```
export { PomodoroComponent as ReactComponent };
```

The Fujion React container requires the top-level component class to be exported as **`ReactComponent`**.

This is essentially all that is required to host a React component in Fujion's React container. But what if we want to invoke a method on our hosted React component?

The React container Java class, **`ReactComponent`**, provides a method called **`rxInvoke`**, which can be used to invoke any published method on your React component. This is similar to the **`BaseComponent.invoke`** method except that it invokes a method on the hosted React component, not on container itself. **`rxInvoke`** can even be called before the hosted React component is fully initialized. This is important since the initialization process is asynchronous. If the hosted component is not fully initialized, calls to **`rxInvoke`** will be queued until initialization is complete.

# Developer Tools

Fujion provides several tools for the software developer.

## Maven Archetypes

A Maven archetype is a template for creating a Maven project. You can think of it as a wizard for creating a new project. Fujion includes archetypes for creating the following project types:

- Fujion Server Page
- Web Application
- AngularJS Component
- React Component
- Custom Fujion Component

From the command line, you can use the Maven **`archetype:generate`** goal to create a project using one of these archetypes. Most IDE's provide a more integrated means to invoke Maven archetypes, but underneath they do the same thing. When you run this goal, Maven will prompt you for any required property values, then create the project in the current working directory. The required properties common to all of the above archetypes are:

| Property | Default Value | Description |
|---|---|---|
| groupId | **none** | **The group id for the new Maven project.** |
| artifactId | **none** | **The artifact id for the new Maven project.** |
| version | **1.0.0-SNAPSHOT** | **The version for the new Maven project.** |
| displayName | **none** | **The display name for the new Maven project.** |
| package | **same as groupId** | **The package name for your Java code.** |
| fujionVersion | **same as archetype version** | **The required Fujion version.** |

## FSP Project Archetype

To invoke the FSP project archetype from the command line:

```
mvn archetype:generate -DarchetypeGroupId=org.fujion -DarchetypeArtifactId=fujion-archetype-fsp
```

The FSP project archetype has one additional required property:

| Property | Default Value | Description |
|---|---|---|
| projectName | **none** | **The name for the FSP project (no spaces, use camel case).** |

## Web Application Archetype

To invoke the web app archetype from the command line:

```
mvn archetype:generate -DarchetypeGroupId=org.fujion -DarchetypeArtifactId=fujion-archetype-webapp
```

This archetype has no additional required properties.

### AngularJS Component Archetype

To invoke the AngularJS component archetype from the command line:

```
mvn archetype:generate -DarchetypeGroupId=org.fujion -DarchetypeArtifactId=fujion-archetype-angular
```

This archetype has no additional required properties.

### React Component Archetype

To invoke the React component archetype from the command line:

```
mvn archetype:generate -DarchetypeGroupId=org.fujion -DarchetypeArtifactId=fujion-archetype-react
```

This archetype has no additional required properties.

### Custom Component Archetype

To invoke the custom component archetype from the command line:

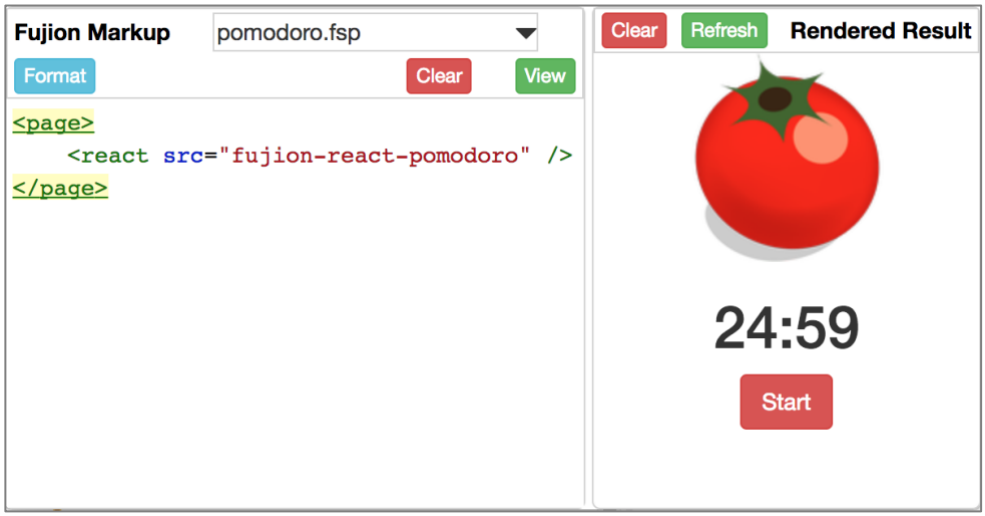
```
mvn archetype:generate -DarchetypeGroupId=org.fujion -DarchetypeArtifactId=fujion-archetype-component
```

The custom component archetype has one additional required property:

| Property | Default Value | Description |
|---|---|---|
| className | **none** | **The name for the widget class (no spaces, use camel case).** |

## Fujion Sandbox

The Fujion Sandbox allows you to enter Fujion markup and view the rendered result. You can also import an existing FSP. This can be very useful during the design phase of a FSP. This tool is available when running the Fujion test harness by loading the FSP **`sandbox.fsp`**:



```
http://localhost:8080/fujion-testharness/sandbox.fsp
```

## Writing Custom Components

Fujion is designed to be extensible. If the existing suite of components does not meet your needs, you may create your own. There are two possible approaches to this: implementation overrides and de novo development.

### Implementation Overrides

Use an implementation override if an existing component comes close to meeting your needs but just needs to be tweaked. This is the simplest way to write a custom component, but also the most limited. In this approach, you subclass the component whose behavior you want to extend. To use the extended version, you specify its class name in the **`impl`** attribute as in this example:

```
<button impl="org.acme.MyButtonSubclass" />
```

This approach has a several limitations. First, the implementation class must be a subclass of the original component class. Second, you may not add additional attributes or remove existing ones. Third, you may not change the containment constraints of the original.

### De Novo Development

With this approach, you create a custom component from scratch. You may choose to extend one of the abstract base component classes and build all of the functionality you need, or extend an existing working component to augment its capabilities. Fujion provides a Maven archetype for creating a custom component project (see the section on developer tools). This archetype will prompt you for a few required inputs and then create a basic Maven project with all of the necessary elements for producing a working custom component.

Let's use the custom component archetype to create a project that will turn the JQuery UI accordion widget (http://api.jqueryui.com/accordion/) into a first class Fujion component. The accordion widget consists of horizontally stacked labeled headers. When a header is clicked, it expands to reveal its associated panel. We will model the accordion as two separate components. The **`AccordionView`** component will act as a simple container for **`AccordionPane`** components. The **`AccordionPane`** component will wrap the header and its associated panel element. The **`AccordionPane`** component has a label, a selection state, and can contain any number of children of any type. If you look at Fujion's **`TabView`** component, it is modeled in much the same way. The complete **`AccordionView`** project may be downloaded from http://fujion.org/assets/1.1/examples/accordion.zip.

First, let's use a Fujion Maven archetype to create our initial project. We will do this from the command line, although your favorite Java IDE probably has a more streamlined way of doing this. From the command line, switch to the folder where

you want your new project to be created and run the Maven goal as shown below (ignoring the line breaks that are provided for clarity):

```
mvn archetype:generate
    -DarchetypeGroupId=org.fujion
    -DarchetypeArtifactId=fujion-archetype-component
```

The archetype plugin will prompt you for the required inputs. Respond to each as follows:

| Property | Value | Description |
|---|---|---|
| groupId | **org.acme** | **This will become the group id of your new Maven artifact.** |
| artifactId | **acme-accordion** | **This will become the artifact id of your new Maven artifact. It will also become the SystemJS package name for your new widget.** |
| package | **org.acme** | **This will become the package name for any Java code. It should default to the value you entered for the group id.** |
| className | **AccordionView** | **This will be the class name for the Java component and its JavaScript widget. Always use camel case.** |
| displayName | **Accordion View** | **This will be the descriptive name for your new Maven artifact.** |

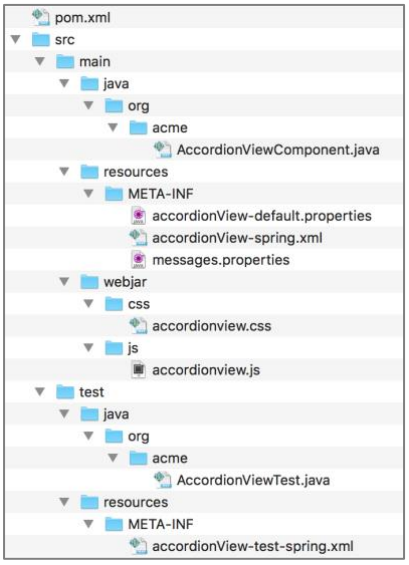
Confirm these choices when prompted. You should now have a Maven project in a folder called **`acme-accordion`**. Switch to that folder and confirm that it will build by entering the command:

```
mvn clean install
```

It should build without errors. At this point you have a working custom component that doesn't do much. Now let's look at the different parts of the project. You may want to import the project into your favorite IDE to facilitate viewing and editing.

Your project structure should look like the graphic to the right. You can see how our input of **`AccordionView`** as a class name was used to create the various file names with different case styles. You can also see that the package name **`org.acme`** contributed to the directory structure. You may delete the following files as we will not be using them for this project:

```
accordionView-default.properties
messages.properties
```



Now let's look at the **`AccordionView`** Java class that was created for us:

```
@Component(tag = "accordionview", widgetModule = "acme-accordion",
   widgetClass = "AccordionView", parentTag = "*")
public class AccordionView extends BaseUIComponent {

   private static final Log log = LogFactory.getLog(AccordionView.class);

   public AccordionView() {
   }

   @Override
   protected void _initProps(Map<String, Object> props) {
      super._initProps(props);
      props.put("wclazz", "accordionview");
   }

}
```

We can again see how some of our inputs were incorporated into the project. The **`@Component`** annotation tells Fujion much of what it needs to know about the component. Fujion will use the information in this annotation to build a component definition. The component definition plays a key role in parsing and materializing FSPs. It is also used to enforce the containment constraints imposed on the component. Here are the valid parameters for this annotation:

@Component org.fujion.annotation.Component

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| childTag | @ChildTag | none | This specifies what child tag(s) are allowed for this component and, optionally, their cardinality. No entry means no children are allowed. An asterisk means any child tag is allowed. See the description of the @ChildTag annotation for more information. |
| content | ContentHandling | ERROR | Specifies how to handle text content (i.e., non-markup text between opening and closing tags). |
| description | String | none | A text description of the component. |
| factoryClass | Class | ComponentFactory | The factory used to create an instance of the component. You will probably never need to modify this. |
| parentTag | String | none | This specifies the allowable tag(s) that may serve as a parent for this component. No entry means no parent is allowed. An asterisk means any parent is allowed. |
| tag | String | none | The name of the component's tag as it will appear in a FSP. This must be unique. |
| widgetClass | String | none | This is the name of the widget's implementing JavaScript class. |
| widgetModule | String | fujion-widget | This is the name of the JavaScript module that will be used by SystemJS to load the module on demand. The default value is reserved for use by Fujion, so you should always specify a different value. |

For components that can have children, the **`childTag`** parameter is itself an annotation type:

@ChildTag org.fujion.annotation.Component.ChildTag

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| value | String | none | The child tag. An asterisk means any child tag is allowed. |
| minimum | integer | 0 | The minimum allowable number of occurrences for this tag. By default, there is no minimum. |
| maximum | integer | <max integer value> | The maximum allowable number of occurrences for this tag. By default, there is no maximum. |

Since our **`@Component`** annotation has no **`childTag`** parameter, it will not allow any children. We are going to change this now. Each **`AccordionView`** component may have zero or more **`AccordionPane`** components. Let's tweak the **`@Component`** annotation to allow this:

```
@Component(tag = "accordionview", widgetModule = "acme-accordion",
                widgetClass = "AccordionView",
                parentTag = "*", childTag = @ChildTag("accordionpane"))
```

The rest of the Java class is pretty straightforward, with the possible exception of the code that is setting the **`wclazz`** property. This is a property of every widget class and specifies the fixed CSS class that will be assigned to the widget's top-level HTML element. While Fujion provides a default value for this, we want one that reflects our ownership of this widget. And, while we're on the topic, Fujion adopts the React Framework's notion of properties and states: a widget's property value, once set, is considered immutable. A widget's state, on the other hand, may change at any time.

Now let's create a skeletal Java class for the accordion panes. Create a new class called **`AccordionPane`** that looks like this:

```
@Component(tag = "accordionpane", widgetModule = "acme-accordion", widgetClass = "AccordionPane",
   parentTag = "accordionview", childTag = @ChildTag("*"))
public class AccordionPane extends BaseUIComponent {

   private static final Log log = LogFactory.getLog(AccordionPane.class);

   public AccordionPane() {
   }

   @Override
   protected void _initProps(Map<String, Object> props) {
      super._initProps(props);
      props.put("wclazz", "accordionpane");
   }

}
```

The **`@Component`** annotation for this class imposes the constraint that the only valid parent for an **`AccordionPane`** is an **`AccordionView`**, but an **`AccordionPane`** can accept any type of child. Now let's add a label property to **`AccordionPane`**:

```
private String label;

@PropertyGetter("label")
public String getLabel() {
   return label;
}

@PropertySetter("label")
public void setLabel(String label) {
   propertyChange("label", this.label, this.label = label, true);
}
```

A couple of things will stand out here. First, the property's setter method does more than just set the field value. That is because the **`label`** property is a synchronized property. That is, its value needs to be synchronized with the client widget. The **`propertyChange`** method does this for you (but only if the property value actually changed).

Second, the property's getter and setter methods are annotated. Like the class annotation **`@Component`**, these method annotations become part of the component definition that is used in the parsing and materialization of FSPs. They map the attribute name in the FSP markup to the methods that read and write the property named by the attribute. Valid parameters for these annotations are:

@PropertyGetter org.fujion.annotation.Component.PropertyGetter

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| value | String | none | The attribute mapped to this property getter. |
| hide | boolean | false | If true, hides any mapping for this method established in a superclass. |
| description | String | none | A text description of the property. |

@PropertySetter org.fujion.annotation.Component.PropertySetter

| Parameter | Datatype | Default Value | Description |
|---|---|---|---|
| value | String | none | The attribute mapped to this property setter. |
| hide | boolean | false | If true, hides any mapping for this method established in a superclass. |
| defer | boolean | false | If true, the setting of this property value is deferred until materialization is complete. |
| description | String | none | A text description of the property. |

Following this same pattern, let's add a **`selected`** property to **`AccordionPane`**:

```
private boolean selected;

@PropertyGetter("selected")
public boolean isSelected() {
   return selected;
}

@PropertySetter("selected")
public void setSelected(boolean selected) {
   _setSelected(selected, true, true);
}

protected void _setSelected(boolean selected, boolean notifyParent, boolean notifyClient) {
   if (propertyChange("selected", this.selected, this.selected = selected, notifyClient)) {
      if (notifyParent && getParent() != null) {
         ((AccordionView) getParent()).setSelectedPane(selected ? this : null);
      }
   }
}
```

The **`selected`** property is a little more complicated, because its state change can be triggered in several ways: by invoking the **`setSelected`** method, by invoking the parent **`AccordionView`**'s **`setSelectedPane`** method, or from the client when an accordion header is clicked. To deal with this added complexity, we have protected **`_setSelected`** method that will notify the parent component and the companion widget of the state change, as directed. If we don't do this, we risk creating an infinite update loop.

At this point, our code won't compile because we have yet to add the **`setSelectedPane`** method to **`AccordionView`**. We do that now:

```
private AccordionPane selectedPane;

public AccordionPane getSelectedPane() {
   return selectedPane;
}

public void setSelectedPane(AccordionPane selectedPane) {
   validateIsChild(selectedPane);

   if (this.selectedPane != null) {
      this.selectedPane._setSelected(false, false, true);
   }

   this.selectedPane = selectedPane;

   if (selectedPane != null) {
      selectedPane._setSelected(true, false, true);
   }
}
```

Note that we don't annotate the getter and setter methods. That is because there is no corresponding FSP attribute for the **`selectedPane`** property. We also don't have any client synchronization logic, since what we are doing is modifying the **`selected`** property on the accordion pane, which is already synchronized. So **`setSelectedPane`** first makes sure the pane we want to select belongs to the **`AccordionView`** component (throwing an exception if it doesn't). Then it deselects the previously selected pane before selecting the new pane.

So far, we've focused on the server side of custom component development. While there are a few more things to be done here, let's move to the client side to implement the functionality we need.

When we created our project, the Maven archetype created a skeletal SystemJS module in the file **`accordionview.js`** (comments removed for brevity):

```
define('acme-accordion', [
        'fujion-core',
        'fujion-widget',
        'acme-accordion-css'
        ],

function(fujion, Widget) {

        var AccordionView = Widget.UIWidget.extend({

                init: function() {
                        this._super();
                },

                render$: function() {
                        return $('<label>Accordion View</label>');
                }


        });

        return Widget.addon('org.acme', 'AccordionView', AccordionView);
});
```

The SystemJS **`define`** function creates a module call **`acme-accordion`** and imports three dependent modules, Fujion's core and widget modules and our CSS file (which we won't be using). In the body of the module, we create a JavaScript class, **`AccordionView`**, by extending an existing widget class, **`UIWidget`**, which is the companion to **`BaseUIComponent`** on the server. This new class overrides the **`init`** method (which just calls its **`_super`** method for now, so doesn't really do anything) and implements the **`render$`** method (methods and variables ending in **`$`** indicate that they return or contain a JQuery object). The **`render$`** method produces a JQuery object that is HTML representation of the widget. Finally, we register the new class and return a reference to the module itself.

Since we also need a widget for the **`AccordionPane`** class, let's add a minimal implementation for it right before the last return statement:

```
var AccordionPane = Widget.UIWidget.extend({

  init: function() {
    this._super();
  },

  render$: function() {
    var dom =
      '<span>'
    +   '<span id="${id}-lbl"/>'
    +   '<div id="${id}-cnt"/>'
    + '</span>';

    return $(this.resolveEL(dom));
  }

});

Widget.addon('org.acme', 'AccordionPane', AccordionPane);
```

The **`render$`** function creates two HTML elements wrapped by a span. Note the embedded placeholders. The **`resolveEL`** method call resolves these placeholders before rendering the HTML. We are basically adding id's, consisting of the widget's id with a suffix, to sub-elements of our widget. Note that we don't add an id to the top-level element. The rendering logic will do this for us. The JQuery object returned by this function is stored in the **`widget$`** field.

Next, let's add support for the **`label`** and **`selected`** properties to the **`AccordionPane`** widget class:

```
  label: function(v) {
    this.sub$('lbl').text(v);
  },

  selected: function(v) {
    this._parent.updateSelected();
  }
```

To understand what is happening here, you need to know how Fujion synchronizes property values from server to the client. Remember that the **`setLabel`** method on the **`AccordionPane`** Java class includes logic that synchronizes the state change with the client (ultimately via the **`sync`** method of **`BaseComponent`**). This logic invokes a method called **`updateState`** on the client widget, which does several things. First it compares the new state value with the current value that is stored in a state map. If the value has changed, it updates the state map and then looks for a method on the widget with the same name as the state. It invokes that method, passing the new value and the old value as arguments. That method is responsible for performing any necessary operations related to that state change. Because state values are stored in a map, they may be re-applied in the event that a re-rendering of the widget is necessary.

So, when our **`label`** property is updated on the server, the **`label`** function gets invoked on the corresponding widget. The **`label`** function sets the text content of the sub-element whose id has the **`lbl`** suffix. Similarly, the **`selected`** widget function gets invoked when the **`selected`** property changes on the server. That function in turn invokes a function on the parent widget (which we have yet to implement).

Let's complete the implementation of the **`AccordionPane`** widget overriding the **`anchor$`** function:

```
anchor$: function() {
  return this.sub$('cnt');
}
```

The **`anchor$`** function returns the JQuery element to which child widgets are to be attached. By default, this is the same as the **`widget$`** field. We need to override this behavior to make sure children are attached to the panel portion of the accordion pane widget.

At this point, the implementation of the **`AccordionPane`** widget is complete. Now let's finish the **`AccordionView`** widget. The JQuery UI accordion widget activates (expands) an accordion panel based on the index of that panel. Let's create a couple of functions to help with this:

```
updateSelected: function() {
  this.widget$.accordion('option', 'active', this.getSelected());
},

getSelected: function() {
  var selected = false;

  this.forEachChild(function(child, index) {
    if (child.getState('selected')) {
      selected = index;
      return false;
    }
  });

  return selected;
}
```

**`getSelected`** searches the child widgets for one with a selected state of true, returning its index. If one isn't found, it returns **`false`** (which the accordion widget interprets as no active panel). **`updateSelected`** uses that value to set the active panel.

Next, we implement the rendering logic:

```
render$: function() {
  return $('<div/>');
},

afterRender: function() {
  var self = this;
  this._super();

  this.widget$.accordion({
    activate: _activate,
    active: false,
    collapsible: true,
    create: _create,
    header: '> .accordionpane > :first-child'
  });

  function _create() {
    setTimeout(function() {
      self.updateSelected();
    });
  }

  function _activate(event, ui) {
    _open(ui.oldHeader, false);
    _open(ui.newHeader, true);
  }

  function _open(header$, selected) {
    var pane = header$ ? header$.fujion$widget() : null;

    if (pane) {
      pane.setState('selected', selected);
      pane.trigger('change', {value: selected});
    }
  }
},

onAddChild: function(child) {
  this.rerender();
}
```

First, we replace the **`render$`** implementation with one that simply returns a div element. Next, we override the **`afterRender`** function to initialize the accordion widget. The **`afterRender`** function is called after the widget and all of its children have been fully rendered, so it is a good place to do this. The initialization code also establishes a handler for panel activation events. This handler sets the associated pane's selection status and fires a **`change`** event with the selected state of the pane. Finally, we provide an implementation of the **`onAddChild`** function that forces a re-render. This is necessary because the accordion widget needs to be re-initialized when a new accordion panel is added.

This completes the implementation of the client-side code. Now we need to put the finishing touches on the server-side component classes.

The **`AccordionPane`** Java class needs to respond to the **`change`** event triggered by the client when the selection state of its widget changes. We do this by adding an event handler:

```
@EventHandler(value = "change")
private void _onChange(ChangeEvent event) {
   _setSelected(defaultify(event.getValue(Boolean.class), true), true, false);
   event = new ChangeEvent(this.getParent(), event.getData(), this);
   EventUtil.send(event);
}
```

The handler updates the selection state and also fires a **`change`** event to the parent component. Finally, we need to override the **`bringToFront`** method to set the selection state to true.

```
@Override
public void bringToFront() {
   setSelected(true);
   super.bringToFront();
}
```

```
@Override
public void afterAddChild(BaseComponent child) {
   if (((AccordionPane) child).isSelected()) {
      setSelectedPane((AccordionPane) child);
   }
}

@Override
public void afterRemoveChild(BaseComponent child) {
   if (child == selectedPane) {
      selectedPane = null;
   }
}
```

Finally, returning to the **`AccordionView`** component, we need to add logic to update the selected pane when a child is added or removed:

Our custom component is now complete. It should now build without error. To test it, include it as a dependency in the test harness and load the following FSP into the sandbox:

```
<accordionview>
  <accordionpane name="pane1" label="pane 1" selected="true">
    <button label="test"
          onClick="self.findByName('pane2').setSelected(true);"/>
  </accordionpane>
  <accordionpane name="pane2" label="pane 2"/>
</accordionview>
```

## Writing Custom Request Handlers

Fujion's WebSocket connection can be leveraged for situations where there is a specialized need to communicate from the client to the server. However, you may not access the WebSocket directly. Rather, you must use Fujion's API for this purpose, the **`fujion.ws.sendData`** method. This method requires you to specify a handler type in addition to the data payload. The handler type is a unique name that you assign. When the server's event dispatcher receives the request, it uses this name to identify the appropriate handler for the payload. You must create the Java class that will serve as the handler for your new type and register it with the event dispatcher.

A request handler must implement the **`IRequestHandler`** interface, which declares the following two methods:

| Method | Description |
|---|---|
| **getRequestType** | **This returns the name of the request type.** |
| **handleRequest** | **This receives the client request for processing.** |

Finally, your request handler must be instantiated and registered with the request dispatcher. The simplest way to do this is to declare it in a Spring configuration file, as in this example:

```
<bean class="org.acme.MyRequestHandler" />
```

When this bean is instantiated, Fujion will recognize it as a request handler by virtue of its interface implementation and will register it automatically.