Alloy XML Markup

Introduction

In Alloy, the XML markup abstracts the Titanium SDK UI components, so you do not need to code the creation and setup of these components using JavaScript and the Titanium SDK API. All view files must be placed in the `app/views` folder of your project with the '.xml' file extension. During code compilation, Alloy looks for these markup files in this specific location to transform them into Titanium code, which can be executed by Studio and the CLI.

The following code is an example of a view file:

app/views/index.xml

```
<Alloy>
    <Window class="container">
        <Label id="label" onClick="doClick">Hello, World</Label>
    </Window>
</Alloy>
```

The `Alloy` tag is the root element for the XML markup and is required in all views. The Window element defines an instance of the `Ti.UI.Wind` ow object and within that window instance is the Label element, which defines an instance of a `Ti.UI.Label` object. Almost all of the Alloy XML tags are the class names of the Titanium UI components without the preceding namespace. Exceptions to this rule are listed in the Element table below.

Within a controller, a UI component can be referenced if its ID attribute is defined. For instance, the Label component in the above example has its ID defined as 'label' and can be referenced in the controller using `$.label`.

If the top-level UI component does not have an ID defined, it can be referenced using the name of the view-controller prefixed with a dollar sign and period ('$.'). For instance, the Window element in the above example can be referenced in the controller using `$.index`.

The following code is how you would traditionally code the markup example using the Titanium SDK:

Resources/app.js

```
var win = Ti.UI.createWindow();
var label = Ti.UI.createLabel({
  text: 'Hello, World'
});
label.addEventListener('click', doClick);
win.add(label);
```

In the previous example, the `win.open` call and implementation of the `doClick` callback are missing. In Alloy, your JavaScript code and Titanium API calls need to be placed in the associated controller file to the view. For this example, the code would need be placed in `app/controllers/index.js`.

The following table lists the attributes for the UI components:

Attribute	Description
`id`	Identifies UI elements in the controller (prefixed with '$.') and style sheet (prefixed with '#'). IDs should be unique per view but are not global, so multiple views can have components with the same ID.
`class`	Applies additional styles (prefixed with '.' in the TSS file). Overwrites the element style but not the id style.
`autoStyle`	Enables the autostyle feature for dynamic styling when adding or removing classes. See Dynamic Styles: Autostyle for more details.
`formFactor`	Acts as a compiler directive for size-specific view components. Value can either be `handheld` or `tablet`. See Conditional code for more details.
`if`	Use a custom query to apply additional styles to the element. See Conditional code and Alloy Styles and Themes: Custom Query Styles for more details.
`module`	Requires in a CommonJS module. Note that the XML element must be named after a `create<XMLElement>` method in the module. See Module Attribute for more details.
`ns`	Overrides the default `Titanium.UI` namespace. See Namespace for more details.
`platform`	Switches the namespace based on the platform and acts as a compiler directive for platform-specific view components. Values can be any combination of platforms. See Conditional code and Namespace for more details.
<properties>	Assigns values to UI object properties. See Property mapping for more details.
<events>	Assigns callbacks to UI object events. See Event handling for more details.

The following table lists the special XML elements besides the Titanium UI components:

Element	Description
`Alloy`	Root element for all view XML files. Required in all views.
`Collection`	Creates a singleton or instance of the specified collection. See the Collection element for more details.
`Model`	Creates a singleton or instance of the specified model. See the Model element for more details.
`Module`	Imports a module view inside this view. See the Module element for more details.
`Require`	Imports a widget or includes another view inside this view. See the Require element for more details.
`Widget`	Imports a widget inside this view. Same as the Require element with the type specified to 'widget'. See Importing widgets for more details.

`index.xml` is a special case that only accepts the following view components as direct children of the Alloy tag:

- Ti.UI.Window or <Window>
- Ti.UI.TabGroup or <TabGroup>
- Ti.UI.NavigationWindow or <NavigationWindow>
- Ti.UI.iOS.SplitWindow or <SplitWindow>

Other views do not have any format restrictions.

For examples, refer to the 'Alloy XML Markup' examples in the Titanium API Guides site. Most examples are in the components under the Titanium.UI section and also in Titanium.Android.Menu, Modules.Facebook.LoginButton, Modules.Map and Titanium.Media.VideoPlayer.

Collection element

The `Collection` XML element creates a singleton or instance of a collection. The `Collection` tag needs to be a child of the `Alloy` parent tag.

The collection singleton is available in the `Alloy.Collections` namespace for the controller code to access. To create a singleton, use the `Collection` tag in markup and assign the `src` attribute to the model file minus the '.js' extension. To access the collection from a controller, use the `Alloy.Collections` namespace and append the model filename (minus the '.js' extension) to the end of it.

For example, the code below creates a collection singleton based on a model called 'book.'

```
<Alloy>
    <Collection src="book" />
    <Window>
        <TableView id="table" />
    </Window>
</Alloy>
```

The code below demonstrates how to access this collection from a controller:

```
var library = Alloy.Collections.book;
library.fetch();
```

The Collection tag can also be used to create an instance of a collection that is only available to one controller. To create an instance of a collection, use the `Collection` tag in markup, assign the `src` attribute to the model file minus the '.js' extension, assign the `id` attribute to access the collection in the controller, and set the `instance` attribute to true. To access the instance in a controller, use the ID that was defined in the markup.

For example, the code below creates a collection instance based on a model called 'book.'

```
<Alloy>
    <Collection id="localLibrary" src="book" instance="true"/>
    <Window>
        <TableView id="table" />
    </Window>
</Alloy>
```

The code below demonstrates how to access this collection from a controller:

```
var library = $.localLibrary;
library.fetch();
```

Model element

The `Model` XML element creates a singleton or instance of a model. The `Model` tag needs to be a child of the `Alloy` parent tag.

The model singleton is available in the `Alloy.Models` namespace for the controller code to access. To create a singleton, use the `Model` tag in markup and assign the `src` attribute to the model file minus the '.js' extension. To access the model from a controller, use the `Alloy.Models` namespace and append the model filename (minus the '.js' extension) to the end of it.

For example, the code below creates a model singleton based on a model called 'book.'

```
<Alloy>
    <Model src="book" />
    <Window>
        <TableView id="table" />
    </Window>
</Alloy>
```

The code below demonstrates how to access this model from a controller:

```
var drama = Alloy.Models.book;
drama.set('title', 'Hamlet');
drama.set('author', 'William Shakespeare');
```

The `Model` tag can also be used to create an instance of a model that is only available to one controller. To create an instance of a model, use the `Model` tag in markup, assign the `src` attribute to the model file minus the '.js' extension, assign the `id` attribute to access the model in the controller, and set the `instance` attribute to true. To access the instance in a controller, use the ID that was defined in the markup.

For example, the code below creates a model instance based on a model called 'book.'

```
<Alloy>
    <Model id="myBook" src="book" instance="true"/>
    <Window>
        <TableView id="table" />
    </Window>
</Alloy>
```

The code below demonstrates how to access this model from a controller:

```
var drama = $.myBook;
drama.set('title', 'Hamlet');
drama.set('author', 'William Shakespeare');
```

Module attribute

You can require a CommonJS module in an Alloy view using the `module` attribute of an XML element. To use the module attribute:

1. Place the CommonJS module in your project's `app/lib` folder. This CommonJS module must expose a public method called `create<XXX>`, where <XXX> is used as the XML element in the Alloy view. This method also must return a Titanium UI object that can be added to the view.
2. Add the <XXX> element to the Alloy view and set the `module` attribute to the path (after the `app/lib` folder) and name of the CommonJS module minus the extension. Custom attributes of the element are passed to the public method.

For example, the following CommonJS module, called `foo.js`, exposes the `createFoo` method, which returns a Label object inside a View object. In the Alloy view, to include this object, add the `Foo` tag and set the module attribute to `foo`.

app/lib/foo.js

```
// XML attributes are passed to the function
exports.createFoo = function (args) {
    var viewArgs = {
        backgroundColor: args.color || 'white',
        width: 100,
        height: 100
    };
    var view = Ti.UI.createView(viewArgs);

    var labelArgs = {
        color: args.textColor || 'black',
        text: args.text || 'Foobar'
    };
    var label = Ti.UI.createLabel(labelArgs);
    view.add(label);

    // Return a UI object that can be added to a view
    return view;
};
```

app/views/index.xml

```
<Alloy>
    <Window backgroundColor="white">
        <!-- Requires in the lib/foo.js module and calls the createFoo method -->
        <Foo module="foo" color="blue" textColor="orange" text="Hello, World!"/>
    </Window>
</Alloy>
```

Module element

You can also include a view from a native module using the `Module` XML element. To use the Module tag:

1. Add the module to your project. For instructions on adding a module to your project, see Using Titanium Modules.
2. Add the `Module` tag in to an Alloy view as a child of a window or another parent object depending on the view object returned by the module.
3. Set the `module` attribute to the name of the module.
4. Set the `method` attribute to the name of the method that creates a view object. If this attribute is not specified, Alloy uses `createView`.
5. If the method invoked uses a simple JavaScript object as its only parameter, you can optionally pass in the parameters either inline or with the TSS file.

For example, to use the Paint Module, first download and add the module to your project. The Paint Module creates a Titanium View, which can be drawn on, using the `createPaintView` method. To use this view in the index view-controller, you need to add it as a child of a window (or similar parent view object). In the code below, the Module tag is used to add the Paint Module to the window and passes properties inline that are specific to the module. You may also pass any `Titanium.UI.View` properties to the module since it extends a Titanium View.

app/views/index.xml

```
<Alloy>
 <Window>
  <Module id="paint" module="ti.paint" method="createPaintView" eraseMode="false"
strokeWidth="1.0" strokeColor="red" strokeAlpha="100" />
  <Button onClick="eraseMe" bottom="0">Erase</Button>
 </Window>
</Alloy>
```

If you can call methods on the created object, then you can invoke those methods in the controller using the assigned ID from the view as a reference to the object. For example, the `PaintView` object created earlier has a method called `clear` that erases all content in the Titanium View. The view in the previous example contains a button with the `eraseMe` function bound to a click event, and since the module has an `id` defi ned, the controller can invoke the `clear` method:

app/controllers/index.js

```
function eraseMe() {
 $.paint.clear();
}

$.index.open();
```

Require element

The `Require` XML element has two uses: including external views and importing widgets into the current view.

Including views

Views may be included in other views using the `Require` element. Specify the `type` attribute as 'view' and the `src` attribute should be the view file minus the '.xml' extension, and assign a unique value to the `id` attribute to reference the UI objects in the controller code. If you omit the `type` attribute, Alloy assumes it is implicitly set to 'view'.

The example below creates a tab group in the main view file and includes two separate files for each tab.

Contents of the main view file (`index.xml`) that includes the rss and about views:

```
<Alloy>
    <TabGroup>
        <Tab id="leftTab">
            <Require type="view" src="rss" id="rssTab"/>
        </Tab>
        <Tab id="rightTab">
            <Require type="view" src="about" id="aboutTab"/>
        </Tab>
    </TabGroup>
</Alloy>
```

Contents of the rss view file (`rss.xml`):

```
<Alloy>
    <Window id="rssWindow">
        <WebView id="rssView" />
    </Window>
</Alloy>
```

Contents of the about view file (`about.xml`):

```
<Alloy>
    <Window id="aboutWindow">
        <WebView id="aboutView" />
    </Window>
</Alloy>
```

To use UI objects from the included views, the controller needs to reference the ID specified in the `Require` element and use the `getView` functi on with the ID of the object as the argument: `var object = $.requireId.getView('objectId')`. The code below demonstrates how to access the web view object from the about view, in the previous example code, to change the URL property.

```
var aboutView = $.aboutTab.getView('aboutView');
aboutView.url = 'http://www.google.com';
```

Importing widgets

Within a view in the regular Alloy project space (`app/views`), use the `<Widget>` tag to import the widget into the application. A `<Widget/>` ele ment is equivalent to a `<Require/>` element whose `type` attribute is set to "widget".

To import a widget:

1. Copy the widget to the `app/widgets` folder. The widget must be contained within its own folder.
2. Add the `<Widget>` tag in the XML markup and specify its `src` attribute as the folder name of the widget.
3. Update the `dependencies` object in the `config.json` file by adding a key/value pair with the name of the widget as the key and the version number as the value.

You can optionally add the `id` and `name` attributes to the `Require` element:

- The `id` attribute allows you to reference the widget in the controller code. You can use this reference to call methods exported by the widget.
- The `name` attribute allows you to import a specific view-controller in the widget rather than the default one (`widget.xml/widget.js`). Specify the name of the view-controller minus the extension.

For example, to import the `mywidget` widget in to a project, copy `mywidget` to the `app/widgets` folder.

```
app
 config.json
 controllers
    index.js
 views
    index.xml
 widgets
     mywidget
         controllers
            foo.js
            widget.js
         views
            foo.xml
            widget.xml
         widget.json
```

Then, add the `<Widget>` tag in the XML markup. Specify the `src` attribute as `mywidget`. Additionally, define the `id` and `name` attributes. Since the `name` attribute is defined, the `foo` view-controller is used instead of the `widget` view-controller.

app/views/index.xml

```
<Alloy>
    <Window>
        <Widget src="mywidget" id="foo" name="foo" />
    </Window>
</Alloy>
```

Since the `id` attribute is defined, the widget can be accessed from the controller.

app/controllers/index.js

```
$.index.open();
$.foo.myMethod();
```

Finally, update the dependencies object in the `config.json` file by adding a key/value pair with the `mywidget` as the key and the `1.0` as the value:

```
...
    "dependencies": {
        "mywidget":"1.0"
    }
```

Passing arguments

You can add any custom attributes to the markup to initialize a widget or controller. For example, consider the following mark-up:

apps/views/index.xml

```
<Require id="foobar" src="foo" customTitle="Hello" customImage="hello.png"/>
```

This is equivalent to the following JavaScript:

apps/controllers/index.js

```
var foobar = Alloy.createController('foo', {
    id: 'foobar',
    customTitle: 'Hello',
    customImage: 'images/hello.png' // Filesystem: app/assets/images/hello.png
});
```

In the required view's controller, the custom properties can be referenced using the `$.args` variable, for example:

apps/controllers/foo.js

```
var title = $.args.customTitle || 'Foobar';
var image = $.args.customImage || 'default.png';
```

See Alloy Controllers: Passing Arguments for more details.

Binding events

To bind a callback to an event in a required view using the on attribute as detailed in Event Handling below, add an event listener for the UI component to trigger the event. For example, suppose you want to require a view that only contains a button. In the parent view, you require the button view and assign a callback to the click event:

Parent View:

```
<Require id="fooButton" src="button" onClick="doClick" />
```

Button View:

```
<Alloy>
    <Button id="button">Click Me!</Button>
</Alloy>
```

The `doClick` method is defined in the parent's controller.

In the controller of the required view, you need to define an event listener that triggers the event for the parent view to receive:

```
$.button.addEventListener('click', function(e) {
    $.trigger('click', e);
});
```

When the button is clicked in the parent view, the controller code in the required view fires a click event, which is caught by the parent view and executes the `doClick` method.

Adding children views

If your Require element is a parent view, you can add children elements to it. These children elements are passed to the parent controller as an array called `$.args.children`. Use this array to access the children views to add them to the parent.

In the example below, you have the index view which is using the `Require` element to include another view called `info`. The required view is a

yellow box with a brown border. Its controller adds the label view element passed in as the first element of the `$.args.children` array.

app/views/info.xml

```
<Alloy>
    <View backgroundColor="yellow" borderWidth="0.5" borderColor="brown"/>
</Alloy>
```

controllers/info.js

```
// add children if there are any
_.each($.args.children || [], function(child) {
    $.info.add(child);
});

$.info.height = Ti.UI.SIZE;
```

app/views/index.xml

```
<Alloy>
    <Window class="container">
        <Require src="info">
            <Label>I am an info box.</Label>
        </Require>
    </Window>
</Alloy>
```

Namespace

By default, all UI components specified in the views are prefixed with `Titanium.UI` for convenience. However, to use a component not part of the `Titanium.UI` namespace, use the `ns` attribute. For example, to use the `Titanium.UI.iOS.BlurView`, do:

```
<BlurView ns="Ti.UI.iOS" id="blurView"/>
```

For UI objects that belong to a specific platform, such as the navigation window. Use the `platform` attribute to use the object, for example:

```
<SplitWindow platform="ios"/>
```

⚠ If you used `<NavigationWindow platform="ios"/>` prior to Titanium 8.0.0, you will need to specify the platform type.

Many of the Titanium view proxies not part of the `Titanium.UI` namespace do not require that the `ns` attribute be explicitly set. The following elements are implicitly mapped to a namespace if one is not defined:

Element	Namespace
Menu	Ti.Android
MenuItem	Ti.Android

Annotation	Ti.Map
VideoPlayer	Ti.Media
MusicPlayer	Ti.Media
SearchView	Ti.UI.Android
AdView	Ti.UI.iOS
CoverFlowView	Ti.UI.iOS
NavigationWindow	Ti.UI
TabbedBar	Ti.UI.iOS
DocumentViewer	Ti.UI.iOS
Popover	Ti.UI.iPad
SplitWindow	Ti.UI.iOS
StatusBar	Ti.UI.iOS

Additionally, use the alias 'Ti' for 'Titanium.'

Conditional code

Add the `platform`, `formFactor` and `if` attributes to apply XML elements based on conditionals.

- To specify a platform-specific element, use the `platform` attribute and assign it a platform, such as, `android`, `ios`, `mobileweb`, or `windows`.
  Comma separate the values to logically OR the values together, for example, `platform='ios,android'` indicates both Android and iOS.
  Prepend the value with an exclamation point (`!`) to negate the value, for example, `platform='!ios'` indicates all platforms except iOS.
- To specify a device-size-specific element, use the `formFactor` attribute and assign it a device size–either `handheld` or `tablet`.
- To use a custom query, assign the `if` attribute to a conditional statement in the `Alloy.Globals` namespace. This conditional statement must return a boolean value. You may only assign **one** query to the `if` attribute.
- The application can also pass custom Boolean properties with the `Alloy.createController()` method, which can be accessed by the XML. Assign the `if` attribute to the name of the property prefixed with the `$.args` namespace, for example, `$.args.someProperty`.

You can use all the attributes in combination.

In the example below, different Annotation objects are displayed in the view based on the platform and device size.

```
<Alloy>
    <Window>
        <Module id="mapview" module="ti.map" method="createView">
            <Annotation title="Cupertino" platform='ios' formFactor='tablet'
latitude='37.3231' longitude='-122.0311'/>
            <Annotation title="Redwood City" platform='ios' formFactor='handheld'
latitude='37.4853' longitude='-122.2353'/>
            <Annotation title="Mountain View" platform='android' latitude='37.3861'
longitude='-122.0828'/>
            <Annotation title="Palo Alto" platform='android,ios,mobileweb'
latitude='37.4419' longitude='-122.1419'/>
            <Annotation title="San Francisco" platform='mobileweb' latitude='37.7750'
longitude='-122.4183'/>
        </View>
    </Window>
</Alloy>
```

You can also create subfolders, named as the platform, in the `views` directory as another way to create

platform-specific views. Refer to Alloy Concepts: Platform-Specific Resources.

Property mapping

Each Titanium UI object property is defined as an attribute in the XML markup and TSS file if it accepts a string, boolean, number or Titanium SDK constant, such as `TI.UI.SIZE` or `Ti.UI.TEXT_ALIGNMENT_CENTER`. Setting properties in the XML markup overrides the settings in the TSS file. Node text can also be used to define the Label text and Button title properties.

For example, the following code defines multiple `Ti.UI.Label` properties and defines the Label text property as node text:

```
<Label borderWidth="1" borderColor="red" color="red" width="Ti.UI.FILL">Hello,
World!</Label>
```

Refer to the Titanium API Guides for the properties of each UI object.

Proxy properties

For properties that are assigned Titanium proxies, such as Views or Buttons, these properties can be declared in markup. Create a child tag under the Titanium UI object tag, using the name of the property with the first character capitalized. Then, declare your Titanium proxy inline with the child property tag. For example, the following code declares a `rightNavButton` for a Window:

```
<Alloy>
    <Window>
        <RightNavButton>
            <Button title="Back" onClick="closeWindow" />
        </RightNavButton>
    </Window>
</Alloy>
```

Currently, the following Titanium proxies and properties implemented using this syntax are:

Titanium Proxy Object / Alloy tag	Proxy Property	Child Alloy Tag	Since
Titanium.Android.MenuItem / <MenuItem>	actionView	<ActionView>	Alloy 1.6.0
Titanium.UI.iPad.Popover / <PopOver>	contentView	<ContentView>	Alloy 1.4.0
Titanium.UI.Label / <Label>	attributedString	<AttributedString>	Alloy 1.7.6
Titanium.UI.ListSection / <ListSection>	footerView	<FooterView>	Alloy 1.3.0
	headerView	<HeaderView>	
Titanium.UI.ListView / <ListView>	footerView	<FooterView>	Alloy 1.3.0
	headerView	<HeaderView>	
	pullView	<PullView>	
	searchView	<SearchBar> or <SearchView platform="android">	
Titanium.UI.OptionDialog / <OptionDialog>	androidView	<AndroidView> or <View>	Alloy 1.5.0
Titanium.UI.TableView / <TableView>	footerView	<FooterView>	Alloy 1.1.0
	headerPullView	<HeaderPullView>	
	headerView	<HeaderView>	
	search	<Search>	
Titanium.UI.TableViewSection / <TableViewSection>	headerView	<HeaderView>	
Titanium.UI.TextArea / <TextArea>	attributedString	<AttributedString>	Alloy 1.7.6
	keyboardToolbar	<KeyboardToolbar>	Alloy 1.5.0

Titanium.UI.TextField / <TextField>	attributedHintText	<AttributeHintText>	Alloy 1.7.6
	attributedString	<AttributedString>	Alloy 1.7.6
	keyboardToolbar	<KeyboardToolbar>	Alloy 1.3.0
	leftButton	<LeftButton>	Alloy 1.3.0
	rightButton	<RightButton>	Alloy 1.3.0
Titanium.UI.Window / <Window>	leftNavButton	<LeftNavButton>	Alloy 1.6.0
	rightNavButton	<RightNavButton>	
	titleControl	<TitleControl>	
	toolbar	<WindowToolbar>	

Android ActionBar

You can set ActionBar properties in the `ActionBar` element to modify the application's action bar. Add the `ActionBar` element as a child of either a `Window` or `TabGroup`, then set `ActionBar` attributes in the XML or TSS file. To add action items to the action bar, add the `Menu` elemen t as a child of either a `Window` or `TabGroup`, then add `MenuItem` elements as children of the `Menu` element. Set `MenuItem` attributes in either the XML or TSS file.

app/views/index.xml

```
<Alloy>
    <Window title="My App">
        <ActionBar id="actionbar" platform="android" title="Home Screen"
onHomeIconItemSelected="showInfo" />
        <Menu>
            <MenuItem id="editItem" title="Edit" onClick="editInfo" />
            <MenuItem id="viewItem" title="View" onClick="viewInfo" />
        </Menu>
        <Label id="label">Use the ActionBar to Perform an Action.</Label>
    </Window>
</Alloy>
```

app/styles/index.tss

```
"MenuItem": {
    showAsAction: Ti.Android.SHOW_AS_ACTION_ALWAYS
},
"#item1": {
    icon: Ti.Android.R.drawable.ic_menu_edit
},
"#item2": {
    icon: Ti.Android.R.drawable.ic_menu_view
},
"#actionbar": {
    icon: "/appicon.png",
}
```

iOS navigation button shorthand

When specifying either the `LeftNavButton` or `RightNavButton` element with a `Window` or iPad `Popover` object, you do not need to create a separate `Button` object inside these elements in the XML file. Instead, you can define the `Button` attributes with the `LeftNavButton` and `Ri ghtNavButton` elements. Note that you cannot use node text to define the button title. It must be specified as the `title` attribute. For example:

app/views/index.xml

```
<Alloy>
    <NavigationWindow>
        <Window>
            <LeftNavButton title="Back" onClick="goBack" />
            <Label>I am iOS!</Label>
        </Window>
    </NavigationWindow>
</Alloy>
```

iOS systemButton shorthand

When specifying the `systemButton` attribute for a Button object, you do not need to use the `Ti.UI.iOS.SystemButton` namespace. For example, the following markup creates the iOS camera button:

```
<Button systemButton="CAMERA"/>
<!-- Instead of -->
<Button systemButton="Titanium.UI.iOS.SystemButton.CAMERA"/>
```

TextField keyboard shorthands

When specifying the keyboardType attribute or returnKeyType attribute for a TextField object, you do not need to use the `Titanium.UI.KEYBOARD_` or `Titanium.UI.RETURNKEY_` namespace, respectively. If you are using these shorthands in the TSS file, the shorthand must be specified as strings, so place them in quotes. For example:

```
<TextField id="txt" keyboardType="DECIMAL_PAD" returnKeyType="DONE"/>

"#txt": {
  keyboardType: "DECIMAL_PAD",
  returnKeyType: "DONE"
}


<!-- Instead of -->

<TextField id="txt" keyboardType="Titanium.UI.KEYBOARD_DECIMAL_PAD"
returnKeyType="Titanium.UI.RETURNKEY_DONE"/>

"#txt": {
  keyboardType: Titanium.UI.KEYBOARD_DECIMAL_PAD,
  returnKeyType: "Titanium.UI.RETURNKEY_DONE"
}
```

Event handling

In Alloy, events may be added in the views using a special attribute. Capitalize the first character of the event name and prefix it with 'on,' so the `Ti.UI.Button` object events `click`, `dblclick` and `swipe` events will become the attributes: `onClick`, `onDblclick`, and `onSwipe`, respectively. These attributes can be used to assign callbacks from the corresponding controller. For example, the view code below binds the button `click` event to the `confirmCB` callback using the `onClick` attribute. The `confirmCB` callback needs to be defined in the associated controller of the view.

```
<Alloy>
    <Window>
        <Button id="confirmButton" onClick="confirmCB">OK</Button>
    </Window>
</Alloy>
```

Refer to the Titanium API Guides for the events of each UI object.

Data binding

If you have a collection of model data that needs to be automatically updated to a view as it changes, you need to use data binding techniques to synchronize the model to a view. See Alloy Data Binding for more details.

Nonstandard syntax

Some Titanium view elements use special syntax. Refer to the 'Alloy XML Markup' examples in the Titanium API Guides site for the following view objects:

- AlertDialog or Ti.UI.AlertDialog
- ButtonBar or Ti.UI.ButtonBar
- CoverFlowView or Ti.UI.iOS.CoverFlowView
- DashboardView or Ti.UI.iOS.DashboardView
- ListView or Ti.UI.ListView
- Map or Ti.Map
- Menu or Ti.Android.Menu
- OptionDialog or Ti.UI.OptionDialog
- Picker or Ti.UI.Picker
- SplitWindow or Ti.UI.iOS.SplitWindow
- TabbedBar or Ti.UI.iOS.TabbedBar
- Toolbar or Ti.UI.Toolbar