iTunes Extras/iTunes LP Development: TuneKit Programming Guide v1.0

11-18-2009

Contents

TuneKit Reference

The TuneKit controller classes are designed to streamline the creation of Cocktail booklets that follow basic design patterns. Using the TuneKit controllers, authors can use JSON to instantiate most of the controllers' properties and rely on an automated spatial keyboard and Apple Remote focus navigation across interactive portions of the booklet.

This document gives an introduction to the TuneKit framework. For more comprehensive details, refer to the TuneKit API documentation.

TKController

The TKController class is the base class for all controllers, as such all of its properties apply to other controller types described below. A basic controller is instantiated as follows:

```
var controller = new TKController({
  id: 'home',
  actions : [
    { selector: '.play', action: bookletController.playFeature }
  ],
  navigatesTo : [
    { selector: '.chapters', controller: chaptersController },
    { selector: '.extras', controller: extrasController }
  ]
});
```

In this example, we create a controller which will automatically load the element with id "home" and wires up actions and navigation anchors to its content.

View

Controllers manage an HTML fragment referred to as the view. The view is loaded dynamically by first looking for an element in the existing DOM tree with the given id property, and if it cannot be found, loading the content of the HTML page in the booklet's views/ directory with the filename of the id. If this also fails, an empty element with that id is created.

Outlets

For any element that you would like to get a reference to, you can define an outlet. Using the outlets property, simply define an array of objects each specifying a name and a selector to target the element. Then, you'll simply have a reference of the name provided on your controller, without having to write any code to get a pointer to it. So if you define an outlet with name : myOutlet, you'll be able to just use the .myOutlet property on your controller object to get to it.

Actions

Actions simply tie an HTML element specified by a CSS selector to a JavaScript callback. Use the actions property to assign an array of such actions, where each action is an object with a selector property specifying the CSS selector to use to target the element, and an action property for the JavaScript function to call. The value of action can be a reference to a function defined elsewhere or a direct inline function definition. It's important to use controller actions instead of wiring click events directly in order to abstract the actual user interaction required depending on the platform. Then TuneKit will handle the user interaction for you. This is particularly important on Apple TV, where the lack of pointer means the target for events like click are unclear.

Navigation

Instead of manually interacting with the TKNavigationController, you can simply identify HTML elements that move to a new controller with the navigatesTo property. Simply pass an array of objects defining both a selector and a controller,

which should be another TKController instance, or one of its subclasses. TuneKit will then automatically set up user interaction that navigates between controllers when the user activates the element given in the selector.

Also, you can define an element that acts as a back button, thus popping the controller from the navigation stack. Simply use the backButton property and set it to whatever CSS selector matches the element you wish to use as a trigger to go back.

Custom Transitions

As the user navigates from controller to controller, the TKNavigationController will use a fade transition by default, but you can provide a custom transition. Use the becomesActiveTransition and becomesInactiveTransition properties. Both these properties expect an object supporting the following:

- properties: an array of CSS properties that are transitioned during the transition. REQUIRED.
- base: an associative array of basic properties to apply before the transition starts. Even indexes are CSS properties and odd indexes are values. OPTIONAL.
- from: an array of CSS values for the initial state of the transition, each index mapping the property at the same index in properties. OPTIONAL.
- to: an array of CSS values for the end state of the transition, each index mapping the property at the same index in properties. REQUIRED.
- duration: a float duration for the transition in seconds. OPTIONAL, defaults to 0.5.
- delay: a float duration for the delay to apply before the transition starts in seconds. OPTIONAL, defaults to 0.

Callbacks

Controllers have a variety of functions that you can override and subclass to monitor their life cycle and various interactions:

- viewDidLoad: the controller's view was loaded for the first time, although it may not have been appended to the DOM tree just yet.
- viewWillAppear: the controller's view will appear on screen, for instance as it's pushed as the topController of the navigation stack.
- viewWillDisappear: the controller's view will disappear from screen, for instance when the user navigates away from it.
- viewDidAppear: the controller's view has appeared on screen, for instance as it's pushed as the topController of the navigation stack and the transition has completed.
- viewDidDisappear: the controller's view has disappeared from screen, for instance when the user navigates away from it and the transition has completed.

TKNavigationController

Traditionally, you will use a single TKNavigationController instance to manage the navigation between the various controllers. On top of the TKController properties defined above, TKNavigationController offers two extra properties. The rootController property lets you define the controller to use as the first controller displayed on screen, while the delegate property lets you define an optional supporting object that will get delegate method calls to track as controllers are pushed onto the stack. There are two optional delegate methods for a TKNavigationController:

- navigationControllerWillShowController(navigationController, controller): notifies that a controller will appear, either because it's being pushed, or it was the back controller when the top controller got popped off the stack. It can be interesting to implement this delegate method to customize the transitions used on the controller going off-screen and the one coming on-screen.

- navigationControllerDidShowController(navigationController, controller): notifies that a controllers did appear, after a push or pop transition has completed.

Note that you can access the top controller at all times using the topController property.

TKTabController

A TKTabController allows to bind a series of elements to a series of controllers that will be shown one at a time as the user selects tabs. The two key properties for a tab controller are:

- tabs: a CSS selector that matches the elements used as tabs
- controllers: an array of TKController instances

There should be as many tabs as there are controllers or the behavior is undefined. When a tab is selected, it receives a DOM focus event and uses the transitions defined on the target controller to show it, and remove the previous controller. You can find out what the current controller is using the selectedController and selectedIndex properties. As tabs are selected, the following delegate methods will be triggered on the object passed as the delegate property:

- tabControllerWillShowController(tabController, controller): notifies that a controller will appear.
- tabControllerDidShowController(tabController, controller): notifies that a controllers did appear.

TKPageSliderController

A TKPageSliderController combines and controls a TKSlidingView and a TKPageControl, always keeping the two in sync. There are two properties to provide the data to either of those views:

- slidingViewData: the data for the managed TKSlidingView.
- pageControlData: the data for the managed TKPageControl.

The TKPageSliderController will set itself as the delegate to both views and you should attempt to monitor delegate method calls on either objects. Instead, you'll be notified of a change in focus or selection using the methods pageWasFocused(index) and pageWasSelected(index), both providing the index of the element of interest.

You can optionally provide a pointer to elements that should be used as triggers to move one page back or one page further using the previousPageButton and nextPageButton properties. Both expect a CSS selector and the correct scripting logic will be automatically provided to connect with the sliding view.

Image faders (hover buttons)

TuneKit provides a very simple way to have buttons that react to hover (as the pointer moves over the element) and to any specific focus navigation that is applied by the system. The technique also has the benefit of being flicker-free (no flashing on first use) since it does not require the images to be preloaded. While it is not mandatory, we encourage you to use this technique.

An image fader is a container element with two children. The container is marked with the CSS class "image-fader". The children of the container should be the "off" and "on" states for the hover button respectively. Here is an example that provides a "Home" button within a view.

```
<div class="home image-fader">
  <img src="images/home.png">
  <img src="images/homeOver.png">
</div>
```

When the image fader is not active (when it doesn't have focus and the pointer is not over the element) the first child will be visible. As the element gets focus or is hovered, the first child will fade out and the second child will fade in.

The following is the CSS that provides the functionality:

```
.image-fader {
  position: absolute;
  font-size: 0;
 cursor:pointer;
}

.image-fader > img {
  -webkit-transition: opacity 0.25s;
}

.image-fader > img:nth-of-type(2) {
  position: absolute;
  top: 0;
  left: 0;
  opacity: 0;
}

.image-fader:hover,
.image-fader.tk-highlighted > img:nth-of-type(2) {
  opacity: 1;
}
```

Preloading image assets

While TuneKit provides a technique for "hover" buttons that avoids the need for preloading images, in some cases it is unavoidable (for example, when the image content is dynamically loaded from script). Every TuneKit controller has an optional `preloads` attribute that takes a list of URLs to load as the view is processed for the first time. While this does not guarantee that the image will be available when you need to display it (especially on non-desktop platforms) it is a good compromise between using performance and appearance.

```
var extrasController = new TKController({
   ...
   preloads: [
     'images/extras/backgroundA.jpg',
     'images/extras/backgroundB.jpg'
   ];
   ...
});
```

Background Audio

When TuneKit initializes it looks in the `appData` object for an attribute called `audioLoop` which specifies if any background music should be played.

The `audioLoop` attribute is an object with two parameters:

- `src`: a URL that points to an audio file. The supported formats for audio are AAC and MP3. REQUIRED
- `loop`: a boolean value that indicates whether or not the audio should repeat infinitely. This also controls whether the music restarts after the user has played media (such as the movie or listening to music). The default value is false. OPTIONAL.

An example of playing background audio is shown below.

```
audioLoop : { src : "audio/background.m4a", loop : false }
```

Navigation on the Apple TV

In general, any interactive element you register with any of the properties from TKController or any of its subclass automatically registers as a keyboard-navigable element. On top of such elements, any HTML `<a>` elements will be registered as well. Any time an element is reached using the automated keyboard navigation, it's given the tk-highlighted CSS class, on top of any other CSS classes it may already have. It also receives a highlight DOM event. When an element loses highlight, the custom CSS class is removed and the unhighlight DOM event is triggered.

In order to find the nearest element to navigate to in a given direction, the built-in navigation mechanism queries the CSS metrics of all the navigable elements in real-time. As such, it is important that all elements that can be given highlight have available CSS metrics and those metrics are accurate (that is, the element is not sized larger than the area it requires to display). Note that container elements that have only absolutely-positioned children will not get a size, so make sure to provide an explicit size on those elements using CSS. It is easy using the Web Inspector to check that your navigable elements have a size, you can simply hover the HTML element in the Elements tab and see the bounding box painted as an overlay on the webpage.

Note that there are some simple ways to customize the metrics for a controller. You can override the customMetricsForElement(element) method on your controller to provide custom metrics for a given element instead of those that would be naturally queried from CSS.

You can also explicitly exclude elements from the list of navigable elements by applying the tk-inactive CSS class to them.

By default, the top-most element in the controller is highlighted. You can use the highlightedElement property to specify a different one with a CSS selector.

Finally, you might want to completely override the navigation system to provide your own custom navigation. This can be done easily by overriding the preferredElementToHighlightInDirection(currentElement, direction) method on your controller.