Sample Code

Implementing a Hybrid TV App with TVMLKit

Display content options with document view controllers and fetch and populate content with TVMLKit JS.

Download

tvOS 13.0+ | Xcode 15.2+

Overview

This hybrid app consists of a server built with TVMLKit JS and a client built with TVMLKit. The app uses native TVMLKit Swift code to take content from the server and build the user interface on screen, and uses TVMLKit JS to provide the requested data.

Traditionally, hybrid tvOS apps depend on TVMLKit JS to initialize view controllers, push view controllers to the navigation stack, and handle events, such as a selection or play event. This app demonstrates the stateless JavaScript design pattern, in which native TVMLKit, instead of TVMLKit JS, takes on these tasks.

Because the TV content resides in JavaScript in the server, the app uses `TVDocumentViewController` objects to allow TVMLKit to communicate with TVMLKit JS. TVMLKit JS is responsible for fetching data and populating the native UI. Building your app with the stateless model is useful if you want to use the functionalities provided by Swift or if you are more familiar with native development.

Configure the Sample Code Project

The sample relies on a running server containing JavaScript files, image assets, and templates that specify the app's content. The app takes the content from the server and builds the user interface

on screen. Before running the app, start the server as follows:

1. In Finder, navigate to this project's folder.
2. In Terminal, enter `cd`, followed by a space.
3. Drag the Server folder from inside the Finder window into your Terminal window. Execute the `cd` command in terminal to change the directory to the Server folder.
4. In Terminal, enter `ruby -run -ehttpd . -p9001` to start the server.
5. Build and run the app.

Create and Display a Document View Controller

XML files provide the structure of a TVMLKit app. The files contain TVML elements such as buttons and lockups, and lay out the elements on screen. In this app, the templates folder in the server directory contains two XML files: Index.xml, and ProductSingle.xml. These files contain the structure of the two primary views used in this app. Index.xml is responsible for providing the structure of the initial page, which consists of a grid of items to select. When the user selects an item from the main page, the ProductSingle.xml file displays the content for the selected item.

At launch time, create a `TVApplicationController` object in the `application(_:didFinishLaunching:)` method. That object provides the infrastructure your app needs to display content. You then use the app controller's delegate method to specify the initial page's content by creating a new `TVDocumentViewController` object.

The `TVDocumentViewController`'s initializer accepts a context dictionary, which provides TVMLKit JS with a route to the new document's data. It then pushes the `TVDocumentViewController` onto the app controller's navigation stack. TVMLKit apps use this navigation controller to manage navigation between pages. When this method returns, TVMLKit builds the app's UI from the Index.xml file. It then queries your server for the data needed to fill that structure.

```
func appController(_ appController: TVApplicationController, didFinishLaunching opti
    // Specify the context for the initial document request.
    let contextDictionary = ["url": "templates/Index.xml"]
    // This URL is sent over to application.js and is part of the request in App.onL
    let documentController = TVDocumentViewController(context: contextDictionary, fo
    documentController.delegate = self
    // Push it onto the navigation stack to start the loading of the document.
    appController.navigationController.pushViewController(documentController, animat
}
```

For more information about how to build the structure of your app, see TVML.

Fetch and Populate TV Content

The structure of the interface is empty until the TVMLKit JS provides the data for the TV content. When you create a `TVDocumentViewController` object in `appController(_:didFinish Launching:)`, TVMLKit determines that it needs data, which causes TVMLKit JS to trigger the `onDocumentRequest` event in `application.js`. This event creates a new `DocumentLoader` object, which grabs data from a `documentProcessor` based on the URL of the request. The URL of the request used is the context used to construct the `TVDocumentViewController` object (templates/Index.xml).

```
App.onDocumentRequest = (request, response) => {
    if (request.requestType == "document") {
        // Setup a document load to update the link.
        new DocumentLoader(gHostController.getRemoteURL(request.url)).load((document
            let processor = documentProcessor[request.url];
            if (processor != null) {
                processor(request, document);
            }
            response.document = document;
            response.close(errorStr ? { 'errorStr' : errorStr } : null);
        });
    }
    else {
        response.close();
    }
};
```

The `documentProcessor`'s `data` variable contains a dictionary of URLs mapped to JSON content. This JSON content includes an artwork URL, background URL, URL path, and other metadata. The app uses this JSON to fill in the UI structure from Index.xml with presentable content, including the images and titles associated with each lockup. To insert the data into the TVML template, the processor retrieves the initial template's top level stack template, and sets its `dataItem` to the JSON data. This inserts the JSON into the template and maps the JSON content to the right variables.

```
let documentProcessor = {
    "templates/Index.xml": (request, document) => {
        let data = {
            movies : [
                {
                    artworkURL : gHostController.getRemoteURL('resources/images/squa
                    backgroundURL : gHostController.getRemoteURL('resources/images/p
```

```
                    logoURL : gHostController.getRemoteURL('resources/images/product
                    title : "Movie Title",
                    theme : "dark",
                    url : "templates/ProductSingle.xml"
                },
                ...
            ]
        };
        // Set on the template element.
        let templateElement = document.getElementsByTagName('stackTemplate').item(0)
        templateElement.dataItem = data;
    },
    "templates/ProductSingle.xml": (request, document) => {
        let extraInfo = request;
        // Set data on the template element.
        let templateElement = document.getElementsByTagName('stackTemplate').item(0)
        templateElement.dataItem = extraInfo;
    }
}
```

For example, the processor passes its movies into the section's binding attribute, populating four sections with the four movie elements. The sample fills in the image binding with the `{artwork URL}` from the data, as well as the `{title}` binding's text content.

```
<section binding="items:{movies};">
    <prototypes>
        <lockup useBrowser="false">
            <img binding="@src:{artworkURL};" contentsMode="aspectFill" width="410"
            <title binding="textContent:{title};" class="hidden_text"/>
        </lockup>
    </prototypes>
</section>
```

Handle Selection Events

The app triggers a selection event when the user selects a lockup. Because this app follows the stateless JavaScript model, you handle that selection event natively in the `documentView Controller(_:handleEvent:with:)` method.

When you trigger the `select` event on a lockup with the `useBrowser` attribute set to `true`, the app creates and pushes a `TVBrowserViewController` to the top of the navigation stack. When

you trigger the `select` event on a lockup with the `useBrowser` attribute set to `false`, the app pushes a standard document conforming to the ProductSingle.xml template to the top of the navigation stack. After the app pushes the appropriate view, it will set the `handled` variable to `true` and return it. This signals that the app natively handled the event, and that it needs no more work on the JavaScript end.

```
if event == .select {
    if let useBrowser = element.attributes!["useBrowser"], useBrowser == "true" {
        // Handle the select event that might lead to loading documents in a browser
        let superParent: TVViewElement? = element.parent?.parent?.name == "shelf" ?
        if let shelfElement = superParent, let browserController = TVBrowserViewCont
            browserController.dataSource = self
            appController.navigationController.pushViewController(browserController,
            handled = true
        }
    } else {
        let documentController = TVDocumentViewController(context: element.elementDa
        documentController.delegate = self
        appController.navigationController.pushViewController(documentController, ar
        handled = true
    }
}
```

When TVMLKit determines that there are unpopulated views that need data, TVMLKit triggers the `onDocumentRequest` event to request data for that document. The app references a `documentProcessor` again, except this time it presents data that corresponds to the ProductSingle.xml template instead of the Index.xml template. This is because the URL from the lockup points to templates/ProductSingle.xml. The new data populated into the ProductSingle template presents a full screen view with the selected image.

To learn more about the different `TVDocumentViewController` events, see TVDocumentViewController.Event.

See Also

JavaScript Environment

~~`class TVApplicationController`~~

An object that bridges the UI, navigation stack, storage, and event handling from JavaScript.

Deprecated

~~class TVApplicationControllerContext~~

Launch information provided to the TV application controller.

Deprecated