Sample Code

Binding JSON data to TVML documents

Create full-fledged TVML documents by using data binding and queries on simplified TVML files.

Download

tvOS 12.0+ | Xcode 14.2+

Overview

This sample shows how to combine JavaScript, JSON data, and TVML elements to create compact TVML documents. You start by creating a local server on your computer to store all of the JSON data and TVML documents. The app reads the JSON data and applies it to the TVML document, and then displays movie posters onscreen. The titles display with different colors based on the amount of rental time remaining.

Configure the sample code project

Before running the app, you need to set up a local server:

1. In Finder, choose DataBindings > DataBindings in the sample project directory.
2. In Terminal, type `cd` followed by a space.
3. Drag the DataBindings folder from the Finder window into the Terminal window, and press Return. This changes the directory to that folder.
4. In Terminal, type `ruby -run -ehttpd . -p9001` and press Return to run the server.
5. Build and run the app in Xcode.

After testing the sample app in Apple TV simulator, you can close the local server by pressing Control-C in Terminal. Closing the Terminal window also stops the server.

Set up the TVML document

The TVML document uses the `prototypes` element to create a reusable collection of elements. Use the `prototype` attribute to uniquely identify the elements. The app creates two `lockup` elements that contain movie poster information.

Rather than creating a separate `lockup` in the TVML document for each JSON object, the sample uses a separate JSON file that contains the data that the `lockup` elements use. This creates smaller TVML files that are easier to read and understand. Reusing `lockup` elements also improves the loading speed when using large data sets.

```
<prototypes>
    <lockup prototype="beach">
        <img binding="@src:{url};" width="200" height="300"/>
        <placeholder tag="title2" />
```

Each `lockup` contains a set of rules, or queries, that change the color of the movie poster's title based on the JSON data. `specialize` elements contain queries that interact with JSON data and act as `if-then` statements. When the JSON data matches the query, the elements inside the `specialize` element replace the `placeholder` element with the matching `tag`. In the app, a movie poster's title appears in one of two colors, depending on the amount of time remaining for that object's availability. For example, movie titles appear in a default color when a user rents them; and then the titles change to an alternate color when the user has fewer than 24 hours remaining in the rental period.

```
<rules>
    <specialize state="({hoursRemaining}-less-than:24)">
        <title tag="title2" class="expiresSoon" binding="textContent:{title};" />"
    </specialize>
    <specialize state="({hoursRemaining}-greater-than-equal:24)">
        <title tag="title2" class="expiresLater" binding="textContent:{title};" />"
    </specialize>
</rules>
```

Retrieve the JSON data

Use an `XMLHttpRequest` to retrieve the JSON information from your server. This creates two arrays that hold JSON objects. Each object has the following properties: `type`, `ID`, `url`, `title`, and `hoursRemaining`. The `hoursRemaining` property contains the data that the queries in the previous section use.

```
{
    "beaches": [
        {"type": "beach", "ID": "00", "url": "Images/Beach_Movie_HiRes/Beach_Movie_2
        {"type": "beach", "ID": "02", "url": "Images/Beach_Movie_HiRes/Beach_Movie_2
        {"type": "beach", "ID": "01", "url": "Images/Beach_Movie_HiRes/Beach_Movie_2
    ],
    "cars": [
        {"type": "cars", "ID": "03", "url": "Images/Car_Movie_HiRes/Car_Movie_250x37
        {"type": "cars", "ID": "04", "url": "Images/Car_Movie_HiRes/Car_Movie_250x37
    ]
}
```

Bind the JSON data

You need to add a type and an identifier to each JSON object because creating new `DataItem` objects requires both a type and a unique identifier for each object. The `type` groups like items together and identifies the correct element to populate the `section` element. The `ID` identifies each object and needs to be unique within each type.

Parse the JSON data into a separate variable. Retrieve the desired `section` element and create a new data item for the section. This sample uses the `lockup` element to populate two `shelf` elements. Inside each shelf is a `section` element that contains the binding variable.

```
// Parse the JSON information.
var results = JSON.parse(information);

// Find the shelf elements.
let shelves = template.getElementsByTagName('shelf');

if (shelves.length > 0) {
    for (var i = 0; i < shelves.length; i++) {
        let shelf = shelves.item(i);
        let section = shelf.getElementsByTagName("section").item(0);
```

Create a new item to contain the JSON information. A mapping function creates a new `DataItem` object with the `type` and `ID` from a JSON object. Populate the object with the remaining JSON data. Return the object to map the information to your array of new items.

```
let newItems = results.cars.map((result) => {
    let objectItem = new DataItem(result.type, result.ID);
    objectItem.url = baseURL + result.url;
    objectItem.title = result.title;
    objectItem.hoursRemaining = result.hoursRemaining;
    return objectItem;
});
section.dataItem.setPropertyPath("images", newItems);
```

Use the `setPropertyPath` method to bind the new data items with the `section` data item.

```
section.dataItem.setPropertyPath("images", newItems);
```

See Also

Data Storage and Retrieval

`XMLHttpRequest`

An object used to retrieve data from a URL.

`DataItem`

An object used to create observable objects from JSON objects for data binding.

`Storage`

An object used to store key-value-pair information.

`DataSource`

An interface that allows the system to detect and respond to changes in your data.

`LoadIndexesRequest`

A request created when the `loadindexes` event is triggered.