CODING FRAMEWORKS

Java in the Browser with TeaVM

Build Web apps using Java on both the front and back ends.

by Andrew Oliver

October 8, 2019

You may remember a time when Java ran in the browser, a time when "pure Java" UIs could be easily launched from a web page. Back then, you could develop a user interface and back end in the same, strongly typed language. Data classes and validation logic could be shared. Code quality tools worked for front-end and back-end code alike. Life was good.

In recent years, however, browser vendors have favored JavaScript and they have steadily degraded support for Java. In a world dominated by client-side web apps, Java is seemingly relegated to being a server-side-only technology. The benefits of a full-stack Java experience are lost. Development productivity nosedives as developers need to jump between different languages and toolsets. Errors are introduced when refactorings on the server side are missed on the client side and vice versa. The excellent Java toolsets for maintaining code quality then need to be reinvented on the client side. It feels like a big step backward.

A Solution

Fortunately there is a solution. TeaVM is an open source project that takes Java code and converts it to fast, compact JavaScript at build time. It provides wrappers for numerous web APIs so your code can fully interact with the browser, from DOM manipulation to getting the user's location. It includes a lightweight framework for web page templating called Flavour. Flavour also has built-in support for easy invocation of JAX-RS web services, including JSON binding.

TeaVM is conceptually related to Google Web Toolkit (GWT), as both products enable you to code in Java and produce browser-friendly JavaScript. However, TeaVM has advantages over GWT in several areas: flexibility, performance, and web-nativeness. TeaVM supports any JVM language, including Kotlin and Scala, whereas GWT is Java-only. TeaVM builds your code faster than GWT, and produces smaller JavaScript that performs better in the browser. Last, but perhaps most important, the TeaVM's Flavour library allows the app content and components to be built with HTML and CSS, which enables developers and designers to work together.

When you pair TeaVM on the front end with traditional Java services on the back end, you have a full Java stack once again. You can code in Java from top to bottom. Share classes between the UI and server. Refactor names in the server and the UI is refactored too. Write unit tests for all your business logic in JUnit. Measure test quality with PIT mutation testing. Check code quality with PMD, Checkstyle, and other utilities.

Getting Started

A TeaVM Flavour app is a standard Maven web project with a few additions. The main page of the application is in the usual place: webapp/index.html (with CSS in `webapp/css/app.css` by convention). However, the main page is usually short, pulling in HTML templates (page fragments) from the resources/templates folder. The HTML templates are linked one-to-one with Java view classes. The view classes (under `src/main/java`) provide business logic for the page. They contain the business logic, properties to be used in the templates, and bindings. More concretely, view classes can do anything from reacting to events, to communicating with a REST service, to changing the displayed template.

In this section I'll show you how to stand up a simple TeaVM application from scratch. Along the way I'll create HTML templates linked to Java-based business logic that reacts to user actions entirely in the browser.

The fastest way to get started is to use the Maven archetype to create a TeaVM Flavour project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.teavm.flavour \
    -DarchetypeArtifactId=teavm-flavour-application
    -DarchetypeVersion=0.2.0
```

If you use `com.mycompany` for the group and package and `flavour` for the artifact ID, you'll end up with these key files:

- `src/main/java/com/mycompany/Client.java`: This file contains the Java logic for the app, referred to as the *view*, and it includes a property (name) used in the template described in the next bullet point.
- `src/main/resources/templates/client.html`: This is the HTML template for the app. It includes an input field and HTML text, which are bound to the `name` field. If you update the input, the HTML changes. As you'll see later in the sections on routing and components, *the HTML template and the view are a fundamental pair used throughout Flavour*.
- `src/main/webapp/css/app.css`: The CSS for the app goes here.
- `src/main/webapp/index.html`: This is the wrapper HTML page for the application. There's not much in here—the real action happens in the templates.

Build the project using the `package` goal:

```
mvn clean package
```

The unpackaged web app files end up in `target/flavour-1.0-SNAPSHOT/`. You can open the index.html file in your browser and see your app working right away. No plugins or downloads are required. Your Java app is running directly in the browser.

If you want to deploy your app in Tomcat, use `target/flavour-1.0-SNAPSHOT.war`. Or you can use a symlink from `${TOMCAT_HOME}/webapps/flavour` to the `flavour-1.0-SNAPSHOT` folder. If you use this latter approach, you can access the app at `http://localhost:8080/flavour` (if Tomcat is configured to run on the default port, which is 8080).

Using Flavour to Show Lists

It is common to want to show a list of items in a UI. You can show short-to-medium length lists using the `<std:foreach>` Flavour element. Just like a regular Java `foreach` loop, it takes a collection to iterate over and a variable name for the collection element, scoped to the loop body.

The collection is read from the view class. If you have defined a list of songs with a getter `public List<Song> getSongs()`, you can make

a simple list as shown next. (All code mentioned in this article is available from the download area.) This code is found in the `list` project in `client.html`:

```
<ol>
  <std:foreach var="song" in="songs">
    <li>
      <html:text value="song.name"> by
      <strong><html:text value="song.artist"></stro
    </li>
  </std:foreach>
</ol>
```

The ordered list (`ol`) produced by Flavour has a list item (`li`) for every `Song` returned by `getSongs()`. Each item in the list includes the song name and artist. The `<html:text>` element evaluates the `value` expression and inserts it into the page. Flavour's expression language allows you to access JavaBeans-style properties (`getX/setX`) by name without explicitly invoking getters and setters. Thus, `song.name` is the easy way to write `song.getName()`.

Let's look at each of these features in more detail.

Other Standard Components

In addition to `std:foreach`, Flavour supports many more standard components to control the page contents:

- `std:if` enables you to add content to a page based on a boolean condition.
- `attr:xyz` lets you set the attribute `xyz` with an expression. For example, `attr:class` would let you set an element's class dynamically.

The standard components work closely with the Flavour Expression Language, discussed next.

Expression Language

Attributes in Flavour components are specified using an expression language (EL). The EL is Java-like, but it has some extra syntactic sugar and a few tweaks to work well inside an HTML page.

All the standard primitives are supported, including the following:

- strings (these use single quotes to work easily inside HTML)
- numbers (integer and floating-point)
- booleans (true and false)

Methods and properties from the view class can be used by name. Method invocations take parameters as in Java. Properties, however, can be referenced directly by name. As you saw earlier, you only need to specify `title`. Flavour handles invoking `getTitle()` for you.

Here are examples of several standard components showing how to use the EL (from `client.html` in `standard-components`).

```
<!-- Uses showHeading boolean property -->
<std:if condition="showHeading">
  <h1>Flavour Messenger</h1>
</std:if>

<!-- Enables a button when the message is entered -
<!-- message.empty is EL shorthand for message.isEmp
<button html:enabled="!message.empty">Check Spellin
```

Reacting to Events

A UI isn't complete without the ability to respond to user interaction. Flavour enables you to invoke Java methods when DOM events occur by using the `event` attribute. Among the commonly used options are the following:

- `event:click`: Handles a click event
- `event:change`: Handles a change in the value of a component

Suppose that in the template, you want to invoke a `send` method when a button is clicked. Simply add the `event:click` attribute on the button:

```
<button event:click="send()">Send</button>
```

Binding

Binding lets you link a property in the view class to a display component. There are several kinds of binding. Some cause the UI to be updated as the properties in the view class change. Some cause properties in the view class to be updated as users manipulate UI components. Bidirectional bindings combine the two, keeping the view class properties and the UI in sync when changes happen on either side. Here are some of the most common:

- `html:text` outputs HTML in the template based on view class properties.
- `html:value` updates an input component based on changes in the view class properties.
- `html:change` invokes view class methods when an input value is changed.
- `html:bidir-value` is the most powerful; it combines `html:value` and `html:change` to keep a UI input field and a view class property in sync, no matter which one changes.

From the previous file:

```
<form>
<!-- Reads/writes from/to the message property on va
  <input type="text" html:bidir-value="message">
</form>
```

Routing

Single-page applications (SPAs) have multiple screens that are switched in-browser, without requiring requests to the server. Flavour provides full support for SPAs via its routing feature. Routing involves several pieces:

- Route interface: Defines the screens and their URLs
- Route implementation: Instantiates the screens on demand
- Screen HTML templates: Contain the layout and components, one per screen
- View classes: Provide data and event handlers for the templates, one per template

Suppose you have a screen that lists restaurants and then a detail page for each restaurant. You could create a template for the restaurant listing and another for the restaurant details. Each template would have a view page. The route interface (in ApplicationRoute.java in my sample code) looks like this:

```
@PathSet
interface ApplicationRoute extends Route {
  @Path("/restaurants")
  public void restaurantList();

  @Path("/restaurant/{id}")
```

```
  public void restaurantDetails(@PathParameter("id"
}
```

Note how the `restaurantDetails` page has a parameter for the restaurant ID.

The `Route` implementation has the logic to switch between pages. By convention, the `Route` is implemented in the `Client.java` main page:

```
@BindTemplate("templates/client.html")
public class Client extends ApplicationTemplate
  implements ApplicationRoute {
    RestaurantSource source = new RestaurantSource(

    public static void main(String[] args) {
        Client client = new Client();
        new RouteBinder()
            .withDefault(ApplicationRoute.class,
                         route -> route.restaurantLi
            .add(client)
            .update();
        client.bind("application-content");
    }

    @Override
    public void restaurantList() {
        setView(new RestaurantListView(source));
    }

    @Override
    public void restaurantDetails(int id) {
        setView(new RestaurantDetailsView(source, id
    }
}
```

The restaurant listing page (`restaurant-list.html` in `routing`) uses `std:foreach` to show the list of restaurants, each with a link to the corresponding restaurant detail page.

```
<div>
  <h1>Restaurants</h1>
  <ol>
    <std:foreach var="restaurant" in="restaurants">
      <li>
        <button type="button"
                event:click="showRestaurant(restaura
          <html:text value="restaurant.name"/>
        </button>
      </li>
    </std:foreach>
  </ol>
</div>
```

The restaurant detail page (`restaurant-detail.html`) shows information about one restaurant. `<html:text>` is used to show fields from the restaurant object.

```
<div>
  <h1><html:text value="restaurant.name"/></h1>
  <h3>In business for
    <html:text value="restaurant.yearsInBusiness"/>
  <p></p>
  <p><button type="button" event:click="showList()">
  Return to the restaurant list</button></p>
</div>
```

RESTful APIs

Most web applications communicate with a server. Common uses are to invoke remote services, store data, and get updates.

Flavour provides an easy way to access web services. The `RESTClient` class can construct clients for JSON-based web services declared as JAX-RS services. Given the popularity of JSON and JAX-RS, there is a good chance your web services are eligible. Simply include your JAX-RS interface in your UI module's POM file and you are then ready to instantiate it in Flavour with one line of code:

```
YourService service =
    RESTClient.factory(YourService.class)
              .createResource("api");
```

Continuing with the restaurant example, let's say your service had a method to get the list of restaurants:

```
List getRestaurants();
```

You could invoke it this way:

```
List restaurants = service.getRestaurants();
```

That's it! Now you have the list of restaurants client-side to use in your view, as shown in the previous section.

Small REST changes have been announced for an upcoming TeaVM version 0.6 release. To make your code work with 0.6, you need only add marker annotations in two places:

- `@Resource(org.teavm.flavour.rest.Resource)` annotation on your REST service interfaces
- `@JsonPersistable (org.teavm.flavour.json.JsonPersistable)` annotation on any custom class passed to or from your service method. (you don't need to do this for standard classes like `String` and `Long`)

Custom Components

As you design your pages and view classes, you'll likely discover there is a chunk of HTML and code that you want to reuse. Two common cases I see regularly are

- A portion of a page needs to be repeated on two different pages
- A page has repeated sections, for example, either a list or table

The first part of defining a component is very similar to defining a normal page in a Flavour app: You have an HTML template bound to a view class. The templates (by convention located in `src/main/components`) contain the HTML for the component. The view class contains the business logic and properties to be used in the template (like before), but adds bindings. Bindings are unique to components—they specify how attributes and other values are bound to the template. Attributes are used to customize the component.

Let's see how you create a restaurant component that could be reused in a list or on multiple pages.

First, define the template in `components/restaurant.html`:

```
<div style="background-color: #99e">
  <div>
    <h1><html:text value="restaurant.name"/></h1>
  </div>
  <div>
    <h3><html:text value="restaurant.yearsInBusines
     years in business!</h3>
```

```
  </div>
</div>
```

Next, define the view class in `src/main/java/com/app/component/Restaurant.java`. Note that the restaurant to be shown in the component is bound to a `restaurant` attribute.

```
@BindTemplate("components/restaurant.html")
@BindElement(name = "restaurant")
public class RestaurantComponent extends AbstractWi
    private Supplier<Restaurant> restaurantSupplier

    public RestaurantComponent(Slot slot) {
      super(slot);
    }

    @BindAttribute(name = "restaurant")
    public void setNameSupplier(
                    Supplier<Restaurant> supplier)
        this.restaurantSupplier = supplier;
    }

    public Restaurant getRestaurant() {
        return restaurantSupplier.get();
    }
}
```

The component also must be registered before use by adding the view class name in

```
META-INF/flavour/component-packages/com.mycompany
```

If your components are in a different package, change `com.mycompany` to match.

With these steps complete, the restaurant component can be used in other templates. Let's redo the restaurant listing page from above to show each restaurant on the listing page. I will place the component inside a `std:foreach` loop, creating one instance of the component for each restaurant (as in `restaurant-list.html`):

```
<?use comp:com.mycompany?>
<div>
  <h1>Restaurants</h1>
  <ol>
    <std:foreach var="restaurant" in="restaurants">
      <li>
        <comp:restaurant restaurant="restaurant"/>
      </li>
    </std:foreach>
  </ol>
</div>
```

Note the `use` processing instruction at the start of the template. It tells Flavour you want to use a component, and it specifies the prefix used later in the file. Here, the prefix is declared as `comp`. Later when it is time to use the component, the prefix reappears in `comp:restaurant-detail`.

Use Location API

You can use the geolocation API to ask users for their location. Add the following code to the sample app's view class (here, `Client.java` in `geolocation`):

```
public void locate() {
    if (Navigator.isGeolocationAvailable()) {
```

```
        Navigator.getGeolocation().getCurrentPositio
            (Position pos) -> {
                final Coordinates coords = pos.getCo
                location = "Lat/Lon: " + coords.getLa
                        + "/" + coords.getLongitude
                Templates.update();
               },
            (PositionError positionError) -> {
               switch (positionError.getCode()) {
                   case PositionError.PERMISSION_DEN
                     location = "The user blocked lo
                     break;
                   case PositionError.POSITION_UNAVA
                     location = "Location could not
                     break;
                   case PositionError.TIMEOUT:
                     location = "A timeout occurred
                             + " to determine your lo
               }
               Templates.update();
            });
    } else {
          location = "This browser doesn't support
          Templates.update();
    }
}

public String getLocation() {
    return location;
}
```

Let's add a button to trigger geolocation lookup and a separate text field to show the results:

```
<div>
  <div>
    <button event:click="locate()">Locate</button>
  </div>

  <div>
    Your location is: <html:text value="location"/>
  </div>
</div>
```

Because the geolocation happens asynchronously, use `Templates.update()` to trigger a display refresh. `Templates.update()` is inexpensive, so don't worry about calling it wherever you need to.

Conclusion

That's it! You are now ready to build your own Java web apps with TeaVM. You can create a restaurant listing website, a new social network, or a breakthrough online game.

For more ideas and information or if you have questions beyond the scope of this article, I recommend you visit the TeaVM website. It has detailed documentation, examples, a forum, an issue list, and more.

Alexey Andreev is the author of TeaVM and Flavour. I am thankful to him for reviewing a draft of this article.

Andrew Oliver

Andy Oliver has been coding in, speaking on, and writing about Java for more than 20 years. He was a speaker at the O'Reilly Conference on Enterprise Java. He has built UIs with a variety of Java toolkits, including AWT, Swing, JavaFX, and Codename One. He currently is working on several TeaVM/Flavour-based projects.

Share this Page