JavaScript Modules

JavaScript Modules are a mechanism for bringing modern concepts of variable scope and dependency management to JavaScript. *Starting with version 5.4*, Tapestry uses RequireJS modules internally, and provides support for using RequireJS modules in your own Tapestry application.

Related Articles

- JavaScript Modules
- TypeScript
- Client-Side JavaScript
- CoffeeScript
- JavaScript FAQ
- Ajax and Zones
- Legacy JavaScript
- Component Cheat Sheet
- Assets

The Need for Modules

As web applications have evolved, the use of JavaScript in the client has expanded almost exponentially. This has caused all kinds of growing pains, since the original design of the web browser, and the initial design of JavaScript, was never intended for this level of complexity. Unlike Java, JavaScript has no native concept of a "package" or "namespace" and has the undesirable tendency to make everything a global.

In the earliest days, client-side JavaScript was constructed as libraries that would define simple functions and variables:

```
function onclickHelp(event) {
  if (helpModal === undefined) {
    helpModal = ...
  }
  document.getElementById("modalContainer") ...
}
$("#helpButton").click(onClickHelp);
```

What's not apparent here is that function `onclickHelp()` is actually attached to the global window object. Further, the variable `helpModal` is also not local, it too gets defined on the window object. If you start to mix and match JavaScript from multiple sources, perhaps various kinds of third-party UI widgets, you start to run the risk of name collisions.

One approach to solving these kinds of problems is a *hygienic function wrapper*. The concept here is to define a function and immediately execute it. The functions and variables defined inside the function are private to that function.

```
(function() {
  var helpModal = null;

  function onClickHelp(event) { ... }

  $("#helpButton").click(onClickHelp);
})();
```

This is an improvement in so far as it assists with name collisions. The variables and functions can only be referenced by name from inside the wrapper.

However, if you are building a library of code to reuse across your application (or building a library to share between applications) then something is still missing: a way to expose just the function you want from inside you wrapper to the outside world.

The old-school route is to choose a hopefully unique prefix, building a cumbersome name (perhaps `myapp_onClickHelp`), and attach that to the global window object. But that just makes your code that much uglier, and leaves you open to problems if not all members of your development team understand the rules and prefixes.

Enter the Asynchronous Module Definition. The AMD is pragmatic way to avoid globals, and adds a number of bells and whistles that can themselves be quite important.

Tapestry uses the RequireJS library as the client-side implementation of AMD. It supplements this on the server-side with Tapestry services for even more flexibility.

Under AMD, JavaScript is broken up into *modules*.

- Modules have a unique name, such as `t5/core/dom` or `app/tree-viewer`.
- A module has a constructor function that *exports* a value.
- A module defines *dependencies* on any number of other modules.
- The export of each dependency is provided as a parameter to the constructor function.

Here's an example from Tapestry itself:

Module t5/core/confirm-click

```
(function() {
  define(["jquery", "./events", "./dom", "bootstrap/modal"], function($, events, dom) {
    var runDialog;
    runDialog = function(options) {
      ...
    };
    $("body").on("click", "[data-confirm-message]:not(.disabled)", function() {
      ...
    });
    dom.onDocument("click", "a[data-confirm-message]:not(.disabled)", function() {
      ...
    });
    return {
      runDialog: runDialog
    };
  });
}).call(this);
```

The `confirm-click` module is used to raise a modal confirmation dialog when certain buttons are clicked; it is loaded by the Confirm mixin.

This module depends on several other modules: `jquery`, `t5/core/events`, `t5/core/dom`, and `bootstrap/modal`. These other modules will have been loaded, and their constructor functions executed, before the `confirm-click` constructor function is executed. The export of each module is provided as a parameter in the order in which the dependencies are defined.

With AMD, the JavaScript libraries may be loaded in parallel by the browser (that's the *asynchronous* part of AMD); RequireJS manages the dependency graph and invokes each function just once, as soon as its dependencies are ready, as libraries are loaded. In some cases, a module may be loaded just for its side effects; such modules will be listed last in the dependency array, and will not have a corresponding parameter in the dependent module's constructor function. In `confirm-click`, the `bootstrap/modal` module is loaded for side-effects.

`confirm-click` defines a local function, `runDialog`. It performs some side-effects, attaching event handlers to the body and the document. The module's export is a JavaScript object containing a function that allows other modules to raise the modal dialog.

If a module truly exports only a single function and is unlikely to change, then it is acceptable to just return the function itself, not an object containing the function. However, returning an object makes it easier to expand the responsibilities of `confirm-click` in the future; perhaps to add a `dismissDialog` function.

Location of Modules

Modules are stored as a special kind of Tapestry asset. On the server, modules are stored on the class path under `META-INF/modules`. In a typical environment, that means the sources will be in `src/main/resources/META-INF/modules`.

Typically, your application will place it's modules directly in this folder. If you are writing a reusable library, you will put modules for that library into a subfolder to prevent naming conflicts. Tapestry's own modules are prefixed with `t5/core`.

If you are using the optional `tapestry-web-resources` module (that's a server-side module, not an AMD module), then you can write your modules as CoffeeScript files (or TypeScript, starting in Tapestry 5.5); Tapestry will take care of compiling them to JavaScript as necessary.

The service ModuleManager is the central piece of server-side support for modules. It supports *overriding* of existing modules by contributing overriding module definitions. This can be useful to monkey patch an existing module supplied with Tapestry, or as part of a third-party library.

Loading Modules from Tapestry Code

Often, you will have a Tapestry page or component that defines client-side behavior; such a component will need to load a module.

The simplest approach is to use the Import annotation:

```
@Import(module = "t5/core/confirm-click")
public class Confirm
{
  ...
}
```

The `module` attribute may either a single module name, or a list of module names.

In many cases, you not only want to require the module, but invoke a function exported by the module. In that case you must use the JavaScriptSupport environmental.

```
@Environmental
JavaScriptSupport javaScriptSupport;

...

javaScriptSupport.require("my-module").with(clientId, actionUrl);

...

javaScriptSupport.require("my-module").invoke("setup").with(clientId, actionUrl);
```

In the first example, `my-module` exports a single function of two parameters. In the second example, `my-module` exports an object and the `setup` key is the function that is invoked.

Development Mode

In development mode, Tapestry will write details into the client-side console.

This lists modules *explicitly* loaded (for initialization), but does not include modules loaded only as dependencies. You can see more details about what was actually loaded using *view source*; RequireJS adds `<script>` tags to the document to load libraries and modules.

Libraries versus Modules

Tapestry still supports JavaScript libraries. When the page is loading, all libraries are loaded before any modules.

Libraries are loaded sequentially, so if you can avoid using libraries, so much the better in terms of page load time.

Libraries work in both normal page rendering, and Ajax partial page updates. Even in partial page updates, the libraries will be loaded sequentially before modules are loaded or exported functions invoked.

Aggregating Modules

An important part of performance for production applications is JavaScript aggregation.

In development mode, you want your modules and other assets to load individually. For both CSS and JavaScript, smaller files that align with corresponding server-side files makes it much easier to debug problems.

Unlike assets, modules can't be fingerprinted, so on each page load, the client browser must ask the server for the module's contents frequently (typically getting a 304 Not Modified response).

This is acceptable in development mode, but quite undesirable in production.

By default, Tapestry sets a max age of 60 (seconds) on modules, so you won't see module requests on every page load. This is configurable and you may want a much higher value in production. If you are rapidly iterating on the source of a module, you may need to force the browser to reload after clearing local cache. Chrome has an option to disable the client-side cache when its developer tools are open.

With JavaScript aggregation, the module can be included in the single virtual JavaScript library that represents a JavaScript stack. This significantly cuts down on both the number of requests from the client to the server, and the overall number of bytes transferred.

Adding a module to the stack is not the same as `require`-ing it. In fact, you must still use `JavaScriptSupport.require()` regardless.

What adding a module to a stack accomplishes is that the module's code is downloaded in the first, initial JavaScript download; the download of the stack's virtual library. When (and if) the module is required as a dependency, the code will already be present in the browser and ready to execute.

Tapestry **does not** attempt to do dependency analysis; that is left as a manual exercise. Typically, if you aggregate a module, your should look at its dependencies, and aggregate those as well. Failure to do so will cause unwanted requests back to the Tapestry server for the dependency modules, even though the aggregated module's code is present.

Because Tapestry is open, it is possible to contribute modules even into the **core** JavaScript stack. This is done using your application's module:

```
@Contribute(JavaScriptStack.class)
@Core
public static void addAppModules(OrderedConfiguration<StackExtension> configuration) {
    configuration.add("tree-viewer", StackExtension.module("tree-viewer"));
            configuration.add("app-utils", StackExtension.module("app-utils"));
}
```

To break this down:

- @Contribute indicates we are contributing to a JavaScriptStack service
- Since there are (or at least, could be) multiple services that implement JavaScriptStack, we provide the @Core annotation to indicate which one we are contributing to (this is a marker annotation, which exists for this exact purpose)
- It is possible to contribute libraries, CSS files, other stacks, and modules; here we are contributing modules
- Each contribution has a unique id and a StackExtension value

The core stack includes several libraries and modules; the exact configuration is subject to a number of factors (such as whether Prototype or jQuery is being used as the underlying framework). That being said, this is the *current* list of modules aggregated into the core stack:

- jquery
- underscore
- t5/core/
    - alert
    - ajax
    - bootstrap
    - console
    - dom
    - events
    - exception-frame
    - fields
    - pageinit
    - messages
    - util
    - validation

The optimum configuration is always a balancing act between including too little and including too much. Generally speaking, including too much is less costly than including too little. It is up to you to analyze the requests coming into your application and determine what modules should be aggregated.