

Legacy JavaScript



For Older Versions of Tapestry

This page describes JavaScript usage in Tapestry versions up through 5.3.x. For version 5.4 and later, see [Client-Side JavaScript](#).

JavaScript is a first-class concept in Tapestry, and sophisticated JavaScript support is provided right out of the box, including rich [Ajax and Zones](#), download optimization, client-side logging, and localization.

In production mode, by default, Tapestry will merge JavaScript libraries, add version numbering, and set a far-future expires header to encourage aggressive browser caching. Starting with version 5.3, Tapestry can also automatically minify (compress) JavaScript libraries when in [production mode](#).

In addition, as will be described in detail [below](#), Tapestry comes with the [Prototype](#) and [Scriptaculous](#) libraries, or you can easily swap in JQuery using a 3rd-party module.

Adding Custom JavaScript

When adding your own custom JavaScript or third-party libraries, just follow the strategies below to take advantage of Tapestry's JavaScript support mechanisms.

The recommended practice in Tapestry is to package up any significant amount of JavaScript as a static JavaScript library, a .js file that can be downloaded to the client. Keep your in-page JavaScript code to a minimum, just the few statements needed to initialize objects and reference methods in the JavaScript libraries.

Linking to your JavaScript libraries

Tapestry provides several ways to add a link to a JavaScript library within your page or component. Although you can use direct `<script type="text/javascript" src="xxx.js"></script>` approach, you should only use it for JavaScript that resides outside of your application. For JavaScript within your app, Tapestry provides *much* better ways to do the same thing. Most users choose the simplest, the `@Import` annotation approach.

Approach 1: @Import

Use the `@Import` annotation (or `@IncludeJavaScriptLibrary` in Tapestry 5.0 and 5.1) to include links to JavaScript (and CSS) files in your page or component. Tapestry ensures that each such file is only referenced once in your page.

[JumpStart Demo](#)

[JavaScript](#)

For Tapestry 5.2 and later

```
@Import(library={"context:js/jquery.js",
                "context:js/myeffects.js"})
public class MyComponent
{
    . . .
}
```

For Tapestry 5.0 and 5.1

```
@IncludeJavaScriptLibrary(value={"context:js/jquery.js",
                                "context:js/myeffects.js"})
public class MyComponent
{
    . . .
}
```

`@Import` may also be applied to individual methods, in which case the import operation only occurs when the method is invoked.

Note: When specifying a file to import, you'll often use the **context:** binding prefix to indicate that the file is stored in the web application context, and not on the classpath. Relative paths will be on the classpath, relative to the Java class. See [Component Parameters](#) for other binding prefix options.

Related Articles

- [Legacy JavaScript](#)
- [Ajax and Zones](#)
- [TypeScript](#)
- [JavaScript Modules](#)
- [Client-Side JavaScript](#)
- [CoffeeScript](#)
- [JavaScript FAQ](#)
- [Ajax Components FAQ](#)
- [Component Cheat Sheet](#)
- [Assets](#)

Adding the same JavaScript library multiple times does *not* create duplicate links. The subsequent ones are simply ignored. In this way, each component can add the libraries it needs, without worrying about conflicts with other components.

Approach 2: JavaScriptSupport

Alternatively, you can use [JavaScriptSupport](#) (for Tapestry 5.2 or later) or [RenderSupport](#) (for Tapestry 5.0 and 5.1) to include a JavaScript library in your page or component. JavaScriptSupport and RenderSupport are [environmental services](#) that include a number of methods that will be used by components, or by services that are called from components. For example:

The importJavaScriptLibrary method

The `importJavaScriptLibrary` method (or `addScriptLink` for Tapestry 5.0 and 5.1) adds a link to a JavaScript library. A component can inject such a script and pass one or more of assets to this method:

Tapestry 5.2 and later

```
@Inject @Path("context:/js/myeffects.js")
private Asset myEffects;

@Environmental
private JavaScriptSupport javascriptSupport;

void setupRender()
{
    javascriptSupport.importJavaScriptLibrary(myEffects);
}
```

Tapestry 5.1 and earlier

```
@Inject @Path("context:/js/myeffects.js")
private Asset myEffects;

@Environmental
private RenderSupport renderSupport;

void setupRender()
{
    renderSupport.addScriptLink(myEffects);
}
```

Tapestry will ensure that the necessary `<link>` elements are added to the *top* of the document (in the `<head>` element). With Tapestry 5.3 and later the new elements are inserted at the bottom of the `<head>` element; in versions before 5.3 they appear at the top of the `<head>` element).

As with the annotation approach, adding the same asset multiple times does *not* create duplicate links.

The `setupRender` method (the name is specifically linked to a [render phase](#)) is the correct place to inform the JavaScriptSupport (or RenderSupport) service that the library is needed.

The addScript method

The `addScript` method is used when you need to add some JavaScript code directly to the page. This will be inserted at the *bottom of the document*, and will only be executed when the document has finished loading on the client (i.e., from the `window.onload` event handler).

JumpStart Demo

[Reusable JavaScript](#)

Tapestry 5.2 and later

```
void afterRender()
{
    javascriptSupport.addScript(
        "$('%s').observe('click', hideMe());",
        container.getClientId());
}
```

Tapestry 5.1 and earlier

```
void afterRender()  
{  
    javascriptSupport.addScript(String.format(  
        "${'%s'}.observe('click', hideMe());",  
        container.getClientId()));  
}
```

When calling the method, the format string can include standard substitutions (such as '%s') for arguments. This saves you the trouble of calling `String.format()` yourself. (For Tapestry 5.1 and earlier, you must call `String.format()` yourself.) In any case, the formatted JavaScript is added to the script block in the rendered output.

Injecting JavaScriptSupport

JavaScriptSupport (like RenderSupport before it) is an *environmental* object, so you will normally inject it via the [@Environmental](#) annotation:

For Tapestry 5.2 and later

```
@Environmental  
private JavaScriptSupport javascriptSupport;
```

For Tapestry 5.0 and 5.1

```
@Environmental  
private RenderSupport renderSupport;
```

The `@Environmental` annotation only works inside components, but occasionally you may want to inject JavaScriptSupport (or RenderSupport) into a service. Fortunately, a proxy has been set up to allow the use of `@Inject` instead:

For Tapestry 5.2 and later

```
@Inject  
private JavaScriptSupport javascriptSupport;
```

For Tapestry 5.0 and 5.1

```
@Inject  
private RenderSupport renderSupport;
```

... or, in a service implementation constructor:

For Tapestry 5.2 and later

```
public MyServiceImpl(JavascriptSupport support)  
{  
    . . .  
}
```

For Tapestry 5.0 and 5.1

```
public MyServiceImpl(RenderSupport support)  
{  
    . . .  
}
```

Inside a component, you should use `@Environmental`, to highlight the fact that `RenderSupport` (like most environmental objects) is only available during rendering, not during action requests.

Combining JavaScript libraries

Added in 5.1.0.2

In production mode, Tapestry automatically *combines* JavaScript libraries. A single request (for a *virtual asset*) will retrieve the combined content of all referenced JavaScript library files.

Note: starting with Tapestry 5.2, JavaScript libraries are only combined if they are part of a JavaScript Stack (see below).

This is a very useful feature, as it reduces the number of requests needed to present a page to the user. It can be disabled, however, by setting the `SymbolConstants.COMBINE_SCRIPTS` [configuration symbol](#) to false in your application's module class (normally `AppModule.java`). By default it is enabled when in production mode and disabled otherwise.

As elsewhere, if the client browser supports gzip compression, the combined JavaScript will be compressed.

Minifying JavaScript libraries

Added in 5.3

In production mode, Tapestry can automatically *minify* (intelligently compresses) JavaScript libraries (and CSS) when the application starts up. This can significantly decrease the size of static content that the browser needs to download.

Minification is accomplished using the `ResourceMinimizer` service. A YUI Compressor-based implementation is available, but this can be overridden.

IMPORTANT NOTE: The `tapestry-core` module only provides the empty infrastructure for supporting minification; the actual logic is supplied in the `tapestry-yuicompressor` module. To use it, you'll need to update your dependencies to include this module.

Maven pom.xml (partial)

```
<dependency>
  <groupId>org.apache.tapestry</groupId>
  <artifactId>tapestry-yuicompressor</artifactId>
  <version>${tapestry-release-version}</version>
</dependency>
```

Gradle would be similar, of course. If you aren't using something like Maven or Gradle, you'll have to download the jar and its dependency (`com.yahoo.platform.yui:yuicompressor`) yourself.

Minification can be disabled by setting the [`tapestry.enable-minification`](#) configuration symbol to false in your application's module class (usually `AppModule.java`). By default it is enabled when in production mode and disabled otherwise.

Please test your applications well: the YUI Compressor code can be somewhat finicky about the application server and JDK version.

Client-side Logging

Deprecated since 5.3

In versions prior to 5.3, Tapestry used a modified version of the Blackbird JavaScript console. The `Tapestry` object includes three functions: `debug`, `warn` and `error`.

Each of these functions take a message and an optional pattern; if the pattern is provided, the message is interpolated on the pattern. The final message is displayed in the Blackbird console, which will make itself visible automatically.

In production mode, debug messages will be filtered out (they will not be visible until the user presses F2 to display the console, and then clicks the grayed out icon for debug messages). In development mode, debug messages are not filtered out.

Example usage:

```
Tapestry.debug("Field id is #{id}, value is #{value}", field);

Tapestry.error("Server is not available.");
```

With Tapestry 5.3 and later the Blackbird console has been removed; just use the standard console logging features (e.g. `console.log()`) built into modern browsers.

Handling Slow Page Loads

If your page loads slowly (typically, because of scripts loaded from external sites), you may see a race condition where the user can click on a link before an event handler for that link has been wired up.

The client-side function `Tapestry.waitForPage()` can be used in an element's onclick handler to force a wait for the page to fully load. In this race condition, the screen will dim and a message will appear advising the user to wait a moment; once the page is fully loaded, this modal dialog will be removed.

The correct usage is:

```
<a href="..." onclick="javascript:Tapestry.waitForPage(event);"> ... </a>
```

The constant `MarkupConstants.WAIT_FOR_PAGE` contains the part of this snippet inside the quotes.

The Standard Tapestry Library

Tapestry's client-side support, the standard Tapestry library, consists of `tapestry.js`, which has dependencies on Prototype and on Scriptaculous Effects. `tapestry.js`, along with its dependencies. The `tapestry.js` library is automatically added to the page when your code adds any other JavaScript or JavaScript library.

Tapestry Namespace

Tapestry defines a number of object and classes inside the Tapestry namespace.

It also adds a handful of methods to the Form class, and to Form elements. These are mostly related to input validation and determining element visibility.

The Tapestry Object `$T()`

Deprecated since 5.2 (no replacement)

The standard library adds a new function, `$T()`. This function is used much like Prototype's `$()`, except that instead of returning a DOM object, it returns a hash (an initially empty JavaScript object) that is associated with the DOM object. This hash is known as *the Tapestry object*.

You may pass in an object id (as a string) or an object reference. The Tapestry Object is created on first invocation. Note: you'll see it as a property name `_tapestry` on the DOM object (which may be useful when debugging).

When Tapestry adds information to a DOM object, it does so in the Tapestry object. This helps avoid name conflicts, and groups all Tapestry-added properties into one place which is much easier to debug.

For example, you might store a value for an element in one place:

```
$T(myid).fadeDuration = .5;
```

Then use it somewhere else:

```
new Effect.Fade($(myId), { duration: $T(myid).fadeDuration });
```

Ajax Components and Mixins

Tapestry provides easy-to-use support for *Ajax*, the technique of using JavaScript to dynamically updating parts of a web page with content from the server without redrawing the whole page. See [Ajax and Zones](#) for details.

Built-in Libraries

Tapestry 5.4 and earlier come with the [Prototype](#) and [Scriptaculous](#) libraries ... no extra download is required. Tapestry will automatically link into your pages the `prototype.js`, `scriptaculous.js`, and `effects.js` libraries, as well as the Tapestry library, `tapestry.js` (which largely consists of support for form input validation). Starting with Tapestry 5.3, [Underscore](#) is also included.

Prototype and Scriptaculous Versions

Tapestry included only Prototype and Scriptaculous in versions prior to Tapestry 5.4. See [Supported Environments and Versions](#) for a matrix of prototype and scriptaculous versions supported by Tapestry.

In versions before 5.4, Tapestry used a modified version of the main Scriptaculous library, `scriptaculous.js`, with the library's default [autoloading](#) behavior turned off. This lets Tapestry and Tapestry components control which Scriptaculous scripts are loaded, rather than having *all* of them loaded unnecessarily.

If you need access to other Scriptaculous libraries, you can provide them as follows:

```
@Inject @Path("${tapestry.scriptaculous}/dragdrop.js")
private Asset dragDropLibrary;

@Environmental
private JavaScriptSupport javascriptSupport;

void setupRender()
{
    javascriptSupport.addScriptLink(dragDropLibrary);
}
```

The Asset is injected, using the [tapestry.scriptaculous](#) configuration symbol to reference the location of the Scriptaculous library.

Even though the `dragdrop.js` library is stored inside a JAR file, Tapestry ensures that it can be accessed from the client web browser. A Tapestry URL within the virtual folder `/assets` is created; the file will be given a version number (the application version number if not specified more specifically) and will be sent to the browser with a far-future expires header (to encourage the browser to cache the file aggressively).

JavaScript Stacks

Added in 5.2

Tapestry allows you to define groups of related JavaScript libraries and stylesheets as "stacks". The built-in "core" stack is used to define the core JavaScript libraries needed by Tapestry (currently, this includes Prototype and Scriptaculous, as well as Tapestry-specific libraries). Other component libraries may define additional stacks for related sets of resources, for example, to bundle together some portion of the ExtJS or YUI libraries.

A [JavaScriptStack](#) can be thought of as a generalization of Tapestry 5.1's ClientInfrastructure, which exists now to define the "core" JavaScript stack.

JavaScript assets of a stack may (when enabled) be exposed to the client as a single URL (identifying the stack by name). The individual assets are combined into a single virtual asset, which is then streamed to the client.

To group several static resources together in a single stack, you must create a new implementation of the JavaScriptStack interface. This interface has four methods:

Alternatives to Prototype

Tapestry 5.4 includes the ability to switch between Prototype and JQuery. For Tapestry 5.3 and earlier, you also have some options::

- [Tapestry5-Jquery module](#) – Using JQuery *instead of* Prototype
- [Tapestry5HowToIntegrateJQuery](#) – Using JQuery *in addition to* Prototype
- [TAPS-1364](#) – lists some starting points for ExtJS integration

- **getStylesheets** : This method will return a list of stylesheet files (StylesheetLink-type object) associated to this stack
- **getJavaScriptLibraries** : This method will return a list of javascript files (Asset-type object) associated to this stack
- **getStacks** : It is also possible to make a stack dependant of other stacks. All the stacks defined in this method will be loaded before the current stack.
- **getInitialization** : this method makes it possible to call a JavaScript initialization for the stack. Tapestry will automatically add this initialization to the page that imports the stacks.

myStack.java

```
public class myStack implements JavaScriptStack {

    private final AssetSource assetSource;

    public myStack (final AssetSource assetSource)
    {
        this.assetSource = assetSource;
    }

    public String getInitialization()
    {
        return null;
    }

    public List<Asset> getJavaScriptLibraries()
    {
        List<Asset> ret = new ArrayList<Asset>();

        ret.add(assetSource.getContextAsset("static/js/jquery.js", null));

        ret.add(assetSource.getContextAsset("static/js/jquery.ui.core.js", null));

        return ret;
    }

    public List<StylesheetLink> getStylesheets()
    {
        List<StylesheetLink> ret = new ArrayList<StylesheetLink>();

        ret.add(new StylesheetLink(assetSource.getContextAsset("static/css/style.css", null)));

        return ret;
    }

    public List<String> getStacks()
    {
        return Collections.emptyList();
    }

}
```

When your new Stack is created, you have to define it in your AppModule.

AppModule.java (partial)

```
@Contribute(JavaScriptStackSource.class)
public static void addMyStack (MappedConfiguration<String, JavaScriptStack> configuration)
{
    configuration.addInstance("MyNewStack", myStack.class);
}
```

You can now use it in your pages and components, by using the @Import annotation or the JavaScriptSupport service :

With @Import

```
@Import(stack="MyNewStack")
public class myPage {
}
```

With JavaScriptSupport

```
@Inject
private JavaScriptSupport js;

@SetupRender
public void importStack(){
    js.importStack("MyNewStack");
}
```