Assets

In Tapestry, **Assets** are any kind of *static* content that may be downloaded to a client web browser, such as images, style sheets and JavaScript files.

Assets can be in one of three places within a Tapestry app:

1. In the **web application's context folder**, stored inside the web application WAR file in the usual JEE fashion. In a project following Maven's directory layout conventions, this would be src/main/webapp or a subdirectory of it (but *not* under src/main/webapp/WEB-INF).
2. For Tapestry 5.4 and later: under **`META-INF`**`, with JavaScript modules under` **`META-INF/modules`** `and other assets under` **`META-INF/assets`**`. This would be src/main/resources/META-INF/modules and src/main/resources/META-INF/assets if following Maven directory conventions.`
3. On the **classpath**, with your Java class files. *This is deprecated in Tapestry 5.4 and later (with a warning).* If following Maven directory conventions, this would correspond to a package-named subdirectory under src/main/resources/, such as src/main/resources/com/example/myapp/pages).

Related Articles

- Assets
- Layout Component
- CSS
- Request Processing
- Configuration
- Legacy JavaScript

Referencing Assets from Templates

For referencing assets from templates, two binding prefixes exist: "context:" and "asset:". The "context:" prefix matches assets in the web application's context folder, and the "asset:" prefix is for assets from the classpath.

src/main/webapp/com/example/myapp/images/tapestry_banner.gif

```
<img src="${context:images/tapestry_banner.gif}"/>
```

This is an example of using a *template expansion* inside an ordinary element (rather than a component).

If you don't provide either prefix, "asset:" is assumed.

Also note that in older code you may occasionally see ${asset:context:...}. That means the same thing as ${context:...}.

Assets in Component Classes

Assets are available to your code as instances of the Asset interface.

Components access assets via injection, using the @Inject annotation, which allows Assets to be injected into components as read-only properties. The path to the resource is specified using the Path annotation:

```
@Inject
@Path("context:images/tapestry_banner.gif")
private Asset banner;
```

Assets are located within *domains*; these domains are identified by the prefix on the @Path annotation's `value`.

META-INF/assets

Support for storing assets under `META-INF/assets` was added in Tapestry 5.4.

For security reasons (detailed below), it is best to have the assets that will be exposed to the client segregated from compiled Java classes. For that reason, classpath assets must be stored in `META-INF/assets` or a subfolder.

For an *application* asset, the assets can be stored directly in `META-INF/assets`.

For a *library* asset, Tapestry uses the library's name (from its LibraryMapping) (such as "core" for the Tapestry core library); The library name becomes a folder under `META-INF/assets`; for example, Tapestry stores its component-related assets under `META-INF/assets/core`.

Classpath Assets

If the prefix is omitted, the value will be interpreted as a path relative to the Java class file itself, within the "classpath:" domain. This is often used when creating component libraries, where the assets used by the components are packaged in the JAR with the components themselves.

Unlike elsewhere in Tapestry, *case matters*. This is because Tapestry is dependent on the Servlet API and the Java runtime to access the underlying files, and those APIs, unlike Tapestry, are case sensitive. Be aware that some *operating systems* (such as Windows) are case insensitive, which may mask errors that will be revealed at deployment (if the deployment operating system is case sensitive, such as Linux).

In Tapestry 5.3 and earlier, classpath assets are packaged in the same folder as the compiled Java class (as well as component templates and so forth). Relative assets are based on this location, the location of the component's .class file.

In Tapestry 5.4, this is supported (but will generate a runtime warning). Classpath resources are expected to be stored under `META-INF/assets`.

In Tapestry 5.5, support for classpath assets **not** under `META-INF/assets` may be removed.

Relative Assets

You can use relative paths with domains (if you omit the prefix):

```
@Inject
@Path("images/edit.png")
private Asset icon;
```

This represents a relative path from the default location for the asset. For Tapestry 5.4, this will resolve as either relative to the component's class file (the logic for Tapestry 5.3 and earlier), or relative to the correct folder within `META-INF/assets` (the logic for Tapestry 5.4 and later).

You may use the standard `./` and `../` prefixes to refer to the current folder, and containing folder, respectfully.

Since you must omit the asset domain prefix in order to specify a relative path, this only makes sense for components packaged in a library for reuse.

Symbols For Assets

Symbols inside the annotation value are expanded. This allows you to define a symbol and reference it as part of the path. For example, you could contribute a symbol named "skin.root" as "context:skins/basic" and then reference an asset from within it:

```
@Inject
@Path("${skin.root}/style.css")
private Asset style;
```


The use of the `${...}` syntax here is a *symbol expansion* (because it occurs in an annotation in Java code), rather than a *template expansion* (which occurs only in Tapestry template files).

An override of the skin.root symbol would affect all references to the named asset.

Localization of Assets

Main Article: Localization

Assets are localized; Tapestry will search for a variation of the file appropriate to the effective locale for the request. In the previous example, a German user of the application may see a file named `edit_de.png` (if such a file exists).

New Asset Domains

If you wish to create new domains for assets, for example to allow assets to be stored on the file system or in a database, you may define a new AssetFactory and contribute it to the AssetSource service configuration.

Asset Fingerprinting (Tapestry 5.3 and earlier)

Tapestry creates a new URL for assets (whether context or classpath). The URL is of the form /assets/**version**/**folder**/**path**.

- **version**: Application version number, defined by the `tapestry.application-version` symbol in your application module (normally AppModule.java). The default is a random hex number.
- **folder**: Identifies the library containing the asset, or "ctx" for a context asset, or "stack" (used when combining multiple JavaScript files into a single virtual asset).
- **path**: The path below the root package of the library to the specific asset file.

Asset Fingerprinting (Tapestry 5.4 and later)

Tapestry 5.4 changes how Asset URLs are constructed. The version number is now a *content fingerprint*, a hash of the actual content of the asset.

Assets get a far-future expires header. It is no longer necessary or desirable to change the application version number.

During development or production, if an asset is changed in any way, it will have a new content fingerprint and will appear, to the browser, to be an entirely new immutable resource.

CSS Link Rewriting

It is frequently the case that CSS files will include links to other files, such as background images, using the `url()` value syntax. Under 5.4, the URL for the CSS file and the targeted file would be broken, due to the inclusions of the CSS file's content hash fingerprint. To fix this, Tapestry parses CSS files, locates the `url()` directives, and rewrites the URLs to be absolute (including the targeted file's content hash fingerprint).

Performance Notes

Assets are expected to be entirely static (not changing while the application is deployed). This allows Tapestry to perform some important performance optimizations.

Tapestry GZIP compresses the content of all assets – if the asset is compressible, the client supports it, and you don't explicitly disable it.

Further, the asset will get a *far future expires header*, which will encourage the client browser to cache the asset.

For Talestry 5.3 and earlier, you should have an explicit application version number for any production application. Client browsers will aggressively cache downloaded assets; they will usually not even send a request to see if the asset has changed once the asset is downloaded the first time. Because of this it is *very important* that each new deployment of your application has a new version number, to force existing clients to re-download all assets.

Asset Security

This applies to how Tapestry 5.3 and earlier manage classpath assets; Tapestry 5.4 introduces another system which doesn't have this issue.

Because Tapestry directly exposes files on the classpath to the clients, some thought has gone into ensuring that malicious clients are not able to download assets that should not be visible to them.

First off all, there's a package limitation: classpath assets are only visible if there's a LibraryMapping for them, and the library mapping substitutes for the initial folders on the classpath. Since the most secure assets, things like `hibernate.cfg.xml` are located in the unnamed package, they are always off limits.

But what about other files on the classpath? Imagine this scenario:

- Your Login page exposes a classpath asset, `icon.png`.
- A malicious client copies the URL, `/assets/1.0.0/app/pages/icon.png (`which would indicate that the Login page is actually inside a library, which is unlikely. More likely, icon.png is a context asset and the malicious user guessed the path for Login.class by looking at the Tapestry source code.) and changes the file name to `Login.class`.
- The client decompiles the class file and spots your secret emergency password: goodbye security! (Never create such back doors, of course!)

Fortunately, this can't happen. Files with extension ".class" are secured; they must be accompanied in the URL with a query parameter that is the MD5 hash of the file's contents. If the query parameter is absent, or doesn't match the actual file's content, the request is rejected.

When your code exposes an Asset that is secured, Tapestry generates a URL that automatically includes MD5 hash query parameter. The malicious user is locked out of access to the files. (The only way they could generate the MD5 hash is if they somehow already have the files, in which case they don't need to download them again anyway.)

By default, Tapestry secures file extensions ".class', ".tml" and ".properties". The list can be extended by contributing to the ResourceDigestGenerator service:

AppModule.java (partial)

```
public static void contributeResourceDigestGenerator(Configuration<String> configuration)
{
    configuration.add("xyz");
}
```

Starting in Tapestry 5.4, there is a move to ensure that all assets are stored under `META-INF/assets`, rather than on the general classpath.

In Tapestry 5.5 and later, assets on the general classpath may not be supported at all.

Minimizing Assets

Since version 5.3, Tapestry provides a service, ResourceMinimizer, which will help to minimize all your static resources (principally CSS and JavaScript files).

Minimization takes place before GZip compression. When aggregating JavaScript for a JavaScriptStack, the minimization is on the aggregated asset, not the individual assets being aggregated.

By default, this service does nothing. You should include a the tapestry-yuicompressor library (for Tapestry 5.3) or tapestry-webresources (for Tapestry 5.4), which makes it possible to minimize CSS and JavaScript files.

For Tapestry 5.3: pom.xml (partial)

```
<dependency>
    <groupId>org.apache.tapestry</groupId>
    <artifactId>tapestry-yuicompressor</artifactId>
    <version>5.3.1</version>
</dependency>
```

For Tapestry 5.4: pom.xml (partial)

```
<dependency>
    <groupId>org.apache.tapestry</groupId>
    <artifactId>tapestry-webresources</artifactId>
    <version>5.4</version>
</dependency>
```

By adding this dependency, all your JavaScript and CSS files will be minimized when Production Mode is true. You can force the minimization of these files, by changing the value of the constant SymbolConstants.MINIFICATION_ENABLED in your module class (usually AppModule.java):

AppModule.java (partial)

```
@Contribute(SymbolProvider.class)
@ApplicationDefaults
public static void contributeApplicationDefaults(MappedConfiguration<String, String> configuration)
{
    configuration.add(SymbolConstants.MINIFICATION_ENABLED, "true");
}
```

If you want to add your own minimizer for particular types of assets, you can contribute to the ResourceMinimizer service. The service configuration maps the MIME-TYPE of your resource to an implementation of the ResourceMinimizer interface.

AppModule.java (partial)

```
@Contribute(ResourceMinimizer.class)
@Primary
public static void contributeMinimizers(MappedConfiguration<String, ResourceMinimizer> configuration)
{
    configuration.addInstance("text/coffeescript", CoffeeScriptMinimizer.class);
}
```