# Meta-Programming Page Content

Extending the If Component Cookbook Using Select With a List

## Meta-Programming Page Content

It is likely that you have some cross-cutting concerns across your pages, specific features you would like to "mix in" to your pages without getting tied into knots by inheritance. This is one of those areas where Tapestry shines.

This specific example is adapted from a real client requirement: the client was concerned about other sites wrapping his content in a frameset and making the site content appear to be theirs. Not all pages (in some cases, that would be an advantage) but specific pages in the application. For those pages, the following behaviors were required:

- Set the X-Frame-Options response header to "DENY"
- Include JavaScript to "pop" the page out of a frame, if in one

Again, this *could* be done by having a specific base-class that included a `beginRender()` method, but the meta-programming approach is nearly as easy and much more flexible.

### Component Meta-Data

In Tapestry, every component (and remember, pages are components) has *meta data*: an extra set of key/value pairs stored in the component's ComponentResources.

By hooking into the component class transformation pipeline, we can change an annotation into meta-data that can be accessed by a filter.

### Defining the Annotation

**ForbidFraming.java**

```
package com.fnord.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Marker annotation for pages that should not allow framing.
 */
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface ForbidFraming {

}
```

This annotation presence is all that's needed; there aren't any additional attributes to configure it.

### Converting the Annotation into Meta-Data

This is in three parts:

- Define the meta-data key, and define a constant for that key
- Set a default meta-data value for the key
- Set a different value for the key when the annotation is present

Our key is just "forbid-framing", with values "true" and "false". The default is "false".

#### Defining the Constant

**FnordSymbols.java**

```
package com.fnord;

import org.apache.tapestry5.services.BaseURLSource;

import com.fnord.annotations.ForbidFraming;

public class FnordSymbols {

  /**
   * Meta-data key; when true, MarkupRendererFilter will inject some extra
   * content into the response to enforce that the content may not be framed
   * (i.e., "stolen").
   *
   * @see ForbidFraming
   */
  public static final String FORBID_FRAMING = "forbid-framing";

}
```

## Setting the Meta-Data Default

Next, we'll create a module just for the logic directly related to framing. In the module, we'll define the default value for the meta-data.

**ForbidFramingModule.class**

```
package com.fnord.services.forbidframing;

import org.apache.tapestry5.ioc.MappedConfiguration;
import org.apache.tapestry5.ioc.annotations.Contribute;
import org.apache.tapestry5.ioc.services.FactoryDefaults;
import org.apache.tapestry5.ioc.services.SymbolProvider;

import com.fnord.FnordSymbols;

public class ForbidFramingModule {

  @Contribute(SymbolProvider.class)
  @FactoryDefaults
  public static void setupForbidFramingDefault(
      MappedConfiguration<String, String> configuration) {
    configuration.add(FnordSymbols.FORBID_FRAMING, "false");
  }
}
```

## Mapping the Annotation

Most of the work has already been done for us: we just have to make a contribution to the MetaWorker service, which is already plugged into the component class transformation pipeline. MetaWorker spots the annotations we define and uses a second object, a MetaDataExtractor we provide, to convert the annotation into a meta-data value.

**ForbidFramingModule.java (partial)**

```
  @Contribute(MetaWorker.class)
  public static void mapAnnotationsToMetaDataValue(
      MappedConfiguration<Class, MetaDataExtractor> configuration) {
    configuration
        .add(ForbidFraming.class, new FixedExtractor<ForbidFraming>(
            FnordSymbols.FORBID_FRAMING));
  }
```

If the ForbidFraming annotation had attributes, we would have provided an implementation of MetaDataExtractor that examined those attributes to set the meta-data value. Since it has no attributes, the FixedExtractor class can be used. The argument is the meta-data key, and the default value is "true".

## Plugging Into Page Rendering

The work we ultimately want to do occurs when rendering a page. Tapestry defines a pipeline for that overall process. The point of a pipeline is that we can add filters to it. We'll add a filter that checks for the meta-data key and adds the response header and JavaScript.

The service is MarkupRenderer, which (being a pipeline service), takes a configuration of filters (in this case, MarkupRendererFilter.

We contribute into the pipeline; the order is important: since the filter will need to write JavaScript, it must be added *after* the built-in filter that provides the JavaScriptSupport environmental object.

**ForbidFramingModule.java (partial)**

```
@Contribute(MarkupRenderer.class)
public static void addFilter(
    OrderedConfiguration<MarkupRendererFilter> configuration) {
  configuration.addInstance("ForbidFraming", ForbidFramingFilter.class,
      "after:JavascriptSupport");
}
```

How do you know what filters are built-in and where to add your own? The right starting point is the JavaDoc for the method of TapestryModule that contributes the base set: contributeMarkupRenderer()

## Implementing the Filter

Everything comes together in the filter:

**ForbidFramingFilter.java**

```
package com.fnord.services.forbidframing;

import org.apache.tapestry5.MarkupWriter;
import org.apache.tapestry5.ioc.annotations.Inject;
import org.apache.tapestry5.services.MarkupRenderer;
import org.apache.tapestry5.services.MarkupRendererFilter;
import org.apache.tapestry5.services.MetaDataLocator;
import org.apache.tapestry5.services.RequestGlobals;
import org.apache.tapestry5.services.Response;
import org.apache.tapestry5.services.javascript.InitializationPriority;
import org.apache.tapestry5.services.javascript.JavaScriptSupport;

import com.fnord.FnordSymbols;

public class ForbidFramingFilter implements MarkupRendererFilter {

  @Inject
  private RequestGlobals requestGlobals;

  @Inject
  private MetaDataLocator metaDataLocator;

  @Inject
  private Response response;

  @Inject
  private JavaScriptSupport jsSupport;

  public void renderMarkup(MarkupWriter writer, MarkupRenderer renderer) {

    String pageName = requestGlobals.getActivePageName();

    boolean forbidFraming = metaDataLocator.findMeta(
        FnordSymbols.FORBID_FRAMING, pageName, boolean.class);

    if (forbidFraming) {
      response.setHeader("X-Frame-Options", "DENY");

      jsSupport.addScript(InitializationPriority.IMMEDIATE,
          "Fnord.popOutOfFrame();");

    }

    renderer.renderMarkup(writer);

  }

}
```

There's a bit going on in this short piece of code. The heart of the code is the MetaDataLocator service; given a meta-data key and a page name, it can not only extract the value, but then coerce it to a desired type, all in one go.

How do we know which page is being rendered? Before Tapestry 5.2 that was a small challenge, but 5.2 adds a method to RequestGlobals for this exact purpose.

Both Request and JavaScriptSupport are per-thread/per-request services. You don't see that here, because that's part of the service definition, and invisible to the consumer code, as here.

Of course, it is vitally important that the filter re-invoke `markup()` on the next renderer in the pipeline (you can see that as the last line of the method).

This code makes one assumption: that the fnord application's Layout component added fnord.js to every page. That's necessary for the JavaScript that's added:

**fnord.js (partial)**

```
Fnord = {
  popOutOfFrame : function() {
    if (top != self)
      top.location.replace(location);
  }
}
```

## Conclusion

That's it: with the above code, simply adding the @ForbidFraming annotation to a page will add the response header and associated JavaScript; no inheritance hassles. This basic pattern can be applied to a wide range of cross-cutting concerns, such as security, transaction management, logging, or virtually any other kind of situation that would normally be solved with inheritance or ugly boilerplate code.

The code in this example was designed for Tapestry version 5.2 and later.