Tapestry5HowToCreateADispatcher

When Tapestry receives a request for a resource (page, etc) a lot happens. Before reading this it would be prudent to first read about how requests are processed on the main page:

http://tapestry.apache.org/request-processing.html

For dispatching and handling requests, Tapestry employs a chain of command implemented as the MasterDispatcher service. The framework uses Tapestry IoC to configure itself, and therefore has a TapestryModule (just like your apps have an "AppModule"). Buried in this module is a method for configuring the MasterDispatcher service: TapestryModule#contributeMasterDispatcher. I strongly recommend looking at the source of this file and especially at the contributions to the MasterDispatcher, as that is how I learned what I am about to share.

There are many reasons to intercept requests. Tapestry installs several dispatchers to intercept certain kinds of requests, in a certain order. The kind of configuration used by the MasterDispatcher (OrderedConfiguration) allows us to insert a dispatcher anywhere in the chain, using the *before:Id* or *after:Id* syntax (again, read the source and API for more information on these). I had a personal interest in this because I work with applications that, like many, require a login and an access control mechanism. Logging in is easy, its implementing the access controls that embodies the real work. In the past I've worked with frameworks that have a roughly similar paradigm of pages, and the only way to implement access controls without coding logic into each controller (or page) class was to implement it in a common base class and subclass that. Well, times have changed. I don't like the fact that my pages know about access controls, it's none of their business. Tapestry 5 provides the infrastructure for a completely transparent access controller, and now we'll create a simple one to demonstrate the usefulness of both the configuration mechanism and the request processing pipeline in Tapestry 5.

When you look at the source of the contributeMasterDispatcher method, you'll notice several dispatchers get installed. The ids of them are:

- RootPath (RootPathDispatcher) - Responsible for rendering the Start page when the app root is requested.
- Asset (AssetDispatcher) - Handles serving assets from the classpath, context path, etc.
- PageRender (PageRenderDispatcher) - Responsible for resolving and rendering pages, as the name suggests.
- ComponentEvent (ComponentEventDispatcher) - Responsible for resolving event handlers for page and component actions.

These dispatchers are added and executed in this order. While it's certainly possible you might want to restrict other types of requests, I'd wager that most of the time developers need to protect pages, or entire sections of pages, from being accessed by unauthorized users.

Now that you have a decent understanding of what dispatchers are and how they work, you should realize that implementing a completely unobtrusive access control system is as easy as implementing a Dispatcher. That part I assure you, is easy. The internal architecture of your system may very will be quite complex, but hooking it into Tapestry 5 is cake, and this is how you do it in a nutshell:

1) Implement a Dispatcher

This is a wonderfully brief interface with one method to implement: dispatch. As expected, looking at the api/source for this interface, as well as some of the implementations will be helpful. For implementing an access controller, its essentially as easy as returning false or throwing an exception. Here's summary of the possible outcomes of this method:

- Return true. This will tell Tapestry that the request has been handled and no other dispatcher in the chain need be consulted. If you return true from your access controller dispatcher, you're likely to get a blank response (and a blank page). This is because "handling" the request also means returning a response, like a page. So for an access controller, you won't do this.
- Return false. This will tell tapestry that the request was not handled and the next dispatcher in the chain should be consulted. If your access controller determines that the user is allowed to access the page, return false.
- Throw an exception. This will short circuit the execution process altogether, as any unhandled exception will. I believe this is the best way to deal with access violations. Of course this raises other questions to be dealt with in other articles, such as what page should be displayed, how to redirect to a login page, etc.

This is how the dispatch method works, so let's implement it. In reality you'll want to use an interface to specify the contract of the access controller, but for the sake of brevity, will skip that.

```
import java.io.IOException;

import org.apache.tapestry5.services.Dispatcher;
import org.apache.tapestry5.services.Request;
import org.apache.tapestry5.services.Response;

public class AccessController implements Dispatcher {
        public boolean dispatch(Request request, Response response) throws IOException {
                boolean canAccess = false;

                /*
                 * Access control logic goes here. If the user is allowed to access the
                 * resource, canAccess should be set to true.
                 */

                if(!canAccess) {
                        /*
                         * This is an unauthorized request, so throw an exception. We'll need
                         * more grace than this, such as a customized exception page and/or
                         * redirection to a login page...
                         */
                        throw new RuntimeException("Access violation!");
                }

                return false;
        }
}
```

Now that you've got a bridge to your access control logic that will fit in to Tapestry's request processing pipeline, how do you hook it in? Contribute it of course!

2) Build or Bind Your Dispatcher

(All of these methods must be in your AppModule.) Before contributing you must first instantiate, or have Tapestry IoC instantiate your dispatcher implementation. How you do this will depend on how exactly you configure your dispatcher. Since we have a very dumb one that uses no dedicated interface (you do not want to bind to the Dispatcher interface), and it currently needs no initialization, we can use a binder. Add this to your bind method:

```
        public static void bind(ServiceBinder binder) {
                binder.bind(AccessController.class).withId("AccessController");
        }
```

3) Contribute Your Dispatcher to the MasterDispatcher

Now we can inject our object using the id "AccessController" into virtually anything. What we need to do now is contribute it to the MasterDispatcher, so we'll inject it into the appropriate contribution method in our AppModule. Before we do that, remember that the MasterDispatcher is a chain of command, and that this chain is structured such that dispatchers are accessed in a specific order. This is good since we need to have our access controller positioned before certain dispatchers, like the page renderer, so we can effectively secure pages. So how do we do this? First you will know from the api (or source) that the MasterDispatcher uses an OrderedConfiguration, and that such configurations allow constraints. These constraints allow you to add a new object before or after another, using the object's id and a simple syntax. You can look at the code for the syntax, but we know that we want our dispatcher to run before page requests are serviced. Page requests are handled by the dispatcher whose id is "PageRender", so here is the contribution we need in our AppModule:

```
    public void contributeMasterDispatcher(OrderedConfiguration<Dispatcher> configuration,
                @InjectService("AccessController") Dispatcher accessController) {
                configuration.add("AccessController", accessController, "before:PageRender");
        }
```

Conclusion

This is the nitty-gritty of writing a dispatcher, and how an access control system can be transparently implemented as one. If you are serious about creating an access control system this way (and it seems to be a perfectly suitable way), then you are left with a few things to figure out:

1. How to initialize your access controller. It seems the best way to do this is use a builder method in your AppModule, and provide the access list in the constructor. This list might be pulled from a database (tapestry-hibernate), or some other resource.

2. What to do when access is denied, and how to do it. There are 2 types of access violations: when a user is not logged in, and when a user is logged in but does not hold sufficient access privileges. The latter is easier, simply present a page telling the user they are not allowed to access what they requested. The former is a bit tricker since you'll want to redirect them to a login page. The access controller would have to know where to redirect to, and in both cases a redirect must be performed.

For now I leave these to you. Any suggestions, corrections, or other furtherings on this document are most welcome. If you'd like to chat with me about this, you can find me in #tapestry on irc.freenode.net (chrislewis).