Principles

Related Articles

- Introduction
- Tapestry for JSF Users
- Tapestry Tutorial
- Principles
- Getting Started

Principle 1 – Static Structure, Dynamic Behavior

The concept of "Dynamic Behavior" should be pretty obvious when you are building a web application; things should look different for different users/situations. But what does it mean that Tapestry has "Static Structure?" Static structure implies that when you build a page in Tapestry you are going to define all of the types of components that are used within that page. Under no circumstance during the rendering or event processing of the page will you be able to dynamically create a new type of component and place that into the component tree.

At first glance, this seems quite limiting ... other frameworks allow new elements to be created on the fly; it's also a common feature of desktop GUIs such as Swing. But static structure turns out to be not so limiting after all. You *can* create new elements (you're actually re-rendering existing components with different properties). And you have plenty of options for getting dynamic behavior out of your static structure; from the simple conditional and looping components to the more advanced implementations of Tapestry's BeanEditor or Grid components, Tapestry gives you control over what renders and when, and even where it appears on the page. And starting in Tapestry 5.3 you can even use the Dynamic component, which renders whatever is in an external template file.

Why did Tapestry choose static structure as a core principle? It's really a matter of meeting the requirements of agility and scalability.

Agility

Tapestry is designed to be an agile working environment; "code less, deliver more". To support you writing less code Tapestry does a lot of work on your POJO pages and components when first loading them. It also uses shared instances of page and component classes (shared across multiple threads and requests). Having dynamically modifiable structure would imply that each request has its own instance and, further, that the entire structure would need to be serialized between requests so that it can be restored to handle later requests.

Tapestry also makes you more agile by speeding up the development cycle with Live Class Reloading. Tapestry monitors the file system for changes to Java page classes, component classes, service implementation classes, HTML templates and component property files, and it hot-swaps the changes into the running application without requiring a restart *or losing session data*. This provides a very short code-save-view cycle that no other framework can touch.

Scalability

When building large scale systems it is important to consider how your resources are going to be used on each deployed server, and how that information is going to be shared between servers. Static structure means that page instances do not need to be stored inside the HttpSession and simple browsing users do not require extra system resources. This lean use of the HttpSession is key to Tapestry's very high scalability, especially in a clustered configuration. Again, linking an instance of a page to a particular client would require vastly more server-side resources than having a single shared page instance.

Principle 2 – Adaptive API

A key feature of Tapestry 5 is its adaptive API.

In traditional Java frameworks (including Struts, JSF and even the now-ancient Tapestry 4) user code is expected to conform to the framework. You create classes that extend from framework-provided base classes, or implement framework-provided interfaces.

This works well until you upgrade to the next release of the framework: with the new features of the upgrade, you will more often than not experience breaks in backwards compatibility. Interfaces or base classes will have changed and your existing code will need to be changed to match.

In Tapestry 5, the framework adapts to your code. You have control over the names of the methods, the parameters they take, and the value that is returned. This is driven by annotations, which tell Tapestry under what circumstances your methods are to be invoked.

For example, you may have a login form and have a method that gets invoked when the form is submitted:

```
public class Login
{
  @Persist
  @Property
  private String userId;

  @Property
  private String password;

  @Component
  private Form form;

  @Inject
  private LoginAuthenticator authenticator;

  void onValidateFromForm()
  {
    if (! authenticator.isValidLogin(userId, password))
    {
      form.recordError("Invalid user name or password.");
    }
  }

  Object onSuccessFromForm()
  {
    return PostLogin.class;
  }
}
```

This short snippet demonstrates a bit about how Tapestry operates. Pages and services within the application are injected with the @Inject annotation. The method names, `onValidateFromForm()` and `onSuccessFromForm()`, inform Tapestry about when each method is to be invoked. This naming convention identifies the event that is handled, ("validate" and "success") and the id of the component from which the event is triggered (the "form" component).

The "validate" event is triggered to perform cross-field validations, and the "success" event is only triggered when there are no validation errors. The `onSuccessFromForm()` method's return value directs Tapestry on what to do next: jump to another page within the application (here identified as the class for the page, but many other options exist). When there are exceptions, the page will be redisplayed to the user.

By contrast, in Tapestry 4 the Form component's listener parameter would be bound to the method to invoke, by name. Further, the listener method had to be public. The Tapestry 5 approach not only supports multiple listeners, but also provides an improved separation of view concerns (inside the page's HTML template) and logic concerns, inside the Java class.

In many cases, additional information about the event is available and can be passed into the method simply by adding parameters to the method. Again, Tapestry will adapt to your parameters, in whatever order you supply them.

Tapestry also saves you needless effort: the @Property annotation marks a field as readable and writable; Tapestry will provide the accessor methods automatically.

Finally, Tapestry 5 explicitly separates actions (requests that change things) and rendering (requests that render pages) into two separate requests. Performing an action, such as clicking an action link or submitting a form, results in a client-side redirect to the new page. This is the "Post/Redirect/Get" pattern (alternatively "Post-Then-Redirect", or "Redirect After Post"). This helps ensure that URLs in the browser are book-markable ... but also requires that a bit more information be stored in the session between requests (using the @Persist annotation).

Principle 3 – Differentiate Public vs. Internal APIs

An issue plaguing much ancient versions of Tapestry (4 and earlier) was the lack of a clear delineation between private, internal APIs and public, external APIs. The fact that your code would extend from base objects but that many of the methods on those base objects were "off limits" further confused the issue. This has been identified as a key factor in the "steep learning curve of Tapestry" meme.

Designed from a clean slate, Tapestry 5 is much more ruthless about what is internal vs. external.

First of all, anything inside the org.apache.tapestry5.internal package is internal. It is part of the implementation of Tapestry. It is the man behind the curtain. You should not ever need to directly use this code. It is a bad idea to do so, because internal code may change from one release to the next without concern for backwards compatibility.

If you ever find yourself forced to make use of internal APIs, please bring it up on the developer mailing list; this is how we know which services should be exposed as public, and fall under the backwards compatibility umbrella.

Principle 4 – Ensure Backwards Compatibility

Older versions of Tapestry were plagued by backwards compatibility problems with every major release. Tapestry 5 did not even attempt to be backwards compatible to Tapestry 4. Instead, it laid the ground work for true backwards compatibility going forwards.

Tapestry 5's API is based largely on naming conventions and annotations. Your components are just ordinary Java classes; you annotate fields to allow Tapestry to maintain their state or to allow Tapestry to inject resources, and you name (or annotate) methods to tell Tapestry under what circumstances a method should be invoked.

Tapestry will adapt to your classes. It will call your methods, passing in values via method parameters. Instead of the rigidness of a fixed interface to implement, Tapestry will simply adapt to your classes, using the hints provided by annotations and simple naming conventions.

Because of this, Tapestry 5 can change internally to a great degree without it affecting any of the application code you write. This has finally cracked the backwards compatibility nut, allowing you to have great assurance that you can upgrade to future releases of Tapestry without breaking your existing applications.

This is already evident in Tapestry 5.1, 5.2 and 5.3 where major new features and improvements have occurred, while remaining 100% backwards compatible to Tapestry 5.0 – as long as you've avoided the temptation to use internal APIs.