Component Events

Component events are Tapestry's way of conveying a user's interactions with the web page, such as clicking links and submitting forms, to designated methods in your page and component classes. When a component event is triggered, Tapestry calls the event handler method you've provided, if any, in the containing component's class.

Let's look at a simple example. Here's a portion of the template for a page (let's call it "Review") that lists documents and lets a user click to edit any one of them.

Related Articles

- Page Life Cycle
- Component Rendering
- Component Events
- Page Navigation
- Request Processing
- Component Events FAQ

Review.tml (partial)

```
<p> Select document to edit: </p>
<t:loop source="documents" value="document">
    <div>
        <t:actionlink" t:id="edit" context="document.id"> ${document.name} </t:actionlink>
    </div>
</t:loop>
```

Notice that Review.tml contains an ActionLink component in a loop. For each rendering within the loop, the ActionLink component creates a component event request URL, with the event type set to "action". In this case, each URL might look like:

```
http://localhost:8080/review.edit/3
```

This URL identifies the **page** that contains the component ("review"), the **Component id** of the component within the page ("edit"), and the **context** value ("3", the "id" property of the document). *Additional context values, if any, are appended to the path.* (The URL may also contain the **event name**, unless, as here, it is "action".)

There's no direct mapping from URL to a piece of code. Instead, when the user clicks on the link, the ActionLink component triggers events. And then Tapestry ensures that the correct bits of code (your event handler method, see below) get invoked for those events.

This demonstrates a critical difference between Tapestry and a more traditional, action oriented framework. The URL doesn't say what happens when the link is clicked, it identifies *which component is responsible* when the link is clicked.

Often, a navigation request (originating with the user) will spawn a number of flow-of-control requests. For example, a form component may trigger an action event, which will then trigger notification events to announce when the form submission is about to be processed, and whether it was successful or not, and those event could be further handled by the page component.

Event Handler Methods

When a component event occurs, Tapestry invokes any event handler methods that you have identified for that event. You can identify your event handler methods via a naming convention (see Method Naming Convention below), or via the @OnEvent annotation.

```
@OnEvent(component="edit")
void editDocument(int docId)
{
    this.selectedId = docId;
    // do something with the document here
}
```

Tapestry does two things here:

- Because of the annotation, it identifies method editDocument() as the method to invoke whenever the component whose ID is "edit" triggers an event.
- Because there is a method parameter, when the link is clicked the context value of the request is converted from a string to an integer and passed in as the method's value parameter.

Added in 5.3

Starting in release 5.3, Tapestry will throw an exception if the component identified for the event handler method doesn't exist in the containing component's template. This helps prevent typos.

In the above example, the editDocument() method will be invoked when any event occurs in in the "edit" component (and has at least one context value).

For some components, more than one type of event can occur, in which case you will want to be more specific:

```
@OnEvent(value="action", component="edit")
void editDocument(int docId)
{
    this.selectedId = docId;
    // do something with the document here
}
```

For the OnEvent annotation, the *value* attribute identifies the name of the event to match. We specified "action" because the ActionLink component triggers the "action" event, as noted in the Component Events section of its javadocs.

Alternatively, we can use the EventLink component, in which case the name of the event is determined by us – either through the "event" parameter or the element's ID:

An EventLink that emits the "delete" event

```
<t:eventlink event="delete" context="document.id"> ${document.name} </t:eventlink>
```

which is equivalent to:

An EventLink that emits the "delete" event

```
<a t:type="eventlink" t:id="delete" context="document.id"> ${document.name} </a>
```

Note that if you omit the `component` part of the OnEvent annotation, then you'll receive notifications from *all* contained components, possibly including nested components (due to event bubbling).

You should usually specify exactly which component(s) you wish to receive events from. Using @OnEvent on a method and not specifying a specific component ID means that the method will be invoked for events from *any* component.

To support testing, it's a common practice to give event handler methods *package-private* visibility, as in the examples on this page, although technically they may have any visibility (even private).

A single event handler method may receive notifications from many different components.

As elsewhere, the comparison of event type and component ID is case-insensitive.

Method Naming Convention

As an alternative to the use of annotations, you may name your event handling methods following a certain convention, and Tapestry will find and invoke your methods just as if they were annotated.

This style of event handler methods start with the prefix "on", followed by the name of the event. You may then continue by adding "From" and a capitalized component id (remember that Tapestry is case insensitive about event names and component IDs). So, for example, a method named onValidateFromSave() will be invoked whenever a "Validate" event is triggered by a component whose component ID is "save".

The previous example may be rewritten as:

```
void onActionFromEdit(int docId)
{
    this.selectedId = docId;
    // do something with the document here
}
```

Many people prefer the naming convention approach, reserving the annotation just for situations that don't otherwise fit.

Method Return Values

Main Article: Page Navigation

For page navigation events (originating in components such as EventLink, ActionLink and Form), the value returned from an event handler method determines how Tapestry will render a response.

- **Null**: For no value, or null, the current page (the page containing the component) will render the response.
- **Page**: For the name of a page, or a page class or page instance, a render request URL will be constructed and sent to the client as a redirect to that page.

- **URL**: For a `java.net.URL`, a redirect will be sent to the client. (In Tapestry 5.3.x and earlier, this only works for non-Ajax requests.)
- **Zone body**: In the case of an Ajax request to update a zone, the component event handler will return the new zone body, typically via an injected component or block.
- **HttpError**: For an HttpError, an error response is sent to the client.
- **Link**: For a Link, a redirect is sent to the client.
- **Stream**: For a StreamResponse, a stream of data is sent to the client
- **boolean:** *true* prevents the event from bubbling up further; *false* lets it bubble up. See Event Bubbling, below.

See Page Navigation for more details.

Multiple Method Matches

In some cases, there may be multiple event handler methods matching a single event. In that case, Tapestry invokes them in the following order:

- Base class methods before sub-class methods.
- Matching methods within a class in alphabetical order.
- For a single method name with multiple overrides, by number of parameters, descending.

Of course, ordinarily would you *not* want to create more than one method to handle an event.

When a sub-class overrides an event handler method of a base class, the event handler method is only invoked once, along with any other base class methods. The subclass can change the *implementation* of the base class method via an override, but can't change the *timing* of when that method is invoked. See issue TAP5-51.

Event Context

The context values (the context parameter to the EventLink or ActionLink component) can be any object. However, only a simple conversion to string occurs. (This is in contrast to Tapestry 4, which had an elaborate type mechanism with the odd name "DataSqueezer".)

Again, whatever your value is (string, number, date), it is converted into a plain string. This results in a more readable URL.

If you have multiple context values (by binding a list or array of objects to the *context* parameter of the EventLink or ActionLink), then each one, in order, will be added to the URL.

When an event handler method is invoked, the strings are converted back into values, or even objects. A ValueEncoder is used to convert between client-side strings and server-side objects. The ValueEncoderSource service provides the necessary value encoders.

As shown in the example above, most of the parameters passed to the event handler method are derived from the values provided in the event context. Each successive method parameter matches against a value provided in the event context (the context parameter of the ActionLink component; though many components have a similar context parameter).

In many cases it is helpful to have direct access to the context (for example, to adapt to cases where there are a variable number of context values). The context values may be passed to an event handler method as parameter of the following types:

- EventContext
- Object[]
- List<Object>

The latter two should be avoided, they may be removed in a future release. In all of these cases, the EventContext parameter acts as a freebie; it doesn't match against a context value as it represents *all* context values.

```
Object onActionFromEdit(EventContext context)
{
    if (context.getCount() > 0) {
        this.selectedId = context.get(0);
        // do something with the document here
    } else {
        alertManager.warn("Please select a document.");
        return null;
    }
}
```

Accessing Request Query Parameters

A parameter may be annotated with the @RequestParameter annotation; this allows query parameters (?name1=value1&name2=value2, etc) to be extracted from the request, converted to the correct type, and passed to the method. Again, this doesn't count against the event context values.

See the example in the Link Components FAQ.

Method Matching

An event handler method will only be invoked *if the context contains at least as many values as the method has parameters*. Methods with too many parameters will be silently skipped.

Tapestry will silently skip over a method if there are insufficient values in the context to satisfy the number of parameters requested.

EventContext parameters, and parameters annotated with @RequestParameter, do not count against this limit.

Method Ordering

When multiple methods match within the same class, Tapestry will invoke them in ascending alphabetical order. When there are multiple overrides of the same method name, Tapestry invokes them in descending order by number of parameters. In general, these situations don't happen ... in most cases, only a single method is required to handle a specific event form a specific component.

An event handler method may return the value `true` to indicate that the event has been handled; this immediately stops the search for additional methods in the same class (or in base classes) or in containing components.

Event Bubbling

The event will bubble up the component hierarchy, first to the containing component, then *that* component's containing component or page, and so on, until it is *aborted* by an event handler method returning *true* or a non-null value.

Returning a boolean value from an event handler method is special. Returning *true* will abort the event with no result; use this when the event is fully handled without a return value and no further event handlers (in the same component, or in containing components) should be invoked.

Returning *false* is the same as returning null; event processing will continue to look for more event handlers, in the same component or its parent.

When an event bubbles up from a component to its container, the origin of the event is changed to be the component. For example, a Form component inside a BeanEditForm component may trigger a success event. The page containing the BeanEditForm may listen for that event, but it will be from the BeanEditForm component (which makes sense, because the id of the Form inside the BeanEditForm is part of the BeanEditForm's implementation, not its public interface).

If you want to handle events that have bubbled up from nested component, you'll soon find that you don't have easy access to the component ID of the firing component. In practical terms this means that you'll want to trigger custom events for the events triggered by those nested components (see Triggering Events, below), and use that custom event name in your event handler method.

Event Method Exceptions

Event methods are allowed to throw any exception (not just runtime exceptions). If an event method does throw an exception, Tapestry will catch the thrown exception and ultimately display the exception report page.

In other words, there's no need to do this:

```
void onActionFromRunQuery()
{
  try
  {
    dao.executeQuery();
  }
  catch (JDBCException ex)
  {
    throw new RuntimeException(ex);
  }
}
```

Instead, you may simply say:

```
void onActionFromRunQuery() throws JDBCException
{
  dao.executeQuery();
}
```

Your event handler method may even declare that it "throws Exception" if that is more convenient.

Intercepting Event Exceptions

When an event handler method throws an exception (checked or runtime), Tapestry gives the component and its containing page a chance to handle the exception, before continuing on to report the exception.

JumpStart Demo

Handling A Bad Context

Tapestry triggers a new event, of type "exception", passing the thrown exception as the context. In fact, the exception is wrapped inside a ComponentEventException, from which you may extract the event type and context.

Thus:

```
  Object onException(Throwable cause)
  {
    message = cause.getMessage();

    return this;
  }
```

The return value of the exception event handler *replaces* the return value of original event handler method. For the typical case (an exception thrown by an "activate" or "action" event), this will be a navigational response such as a page instance or page name.

This can be handy for handling cases where the data in the URL is incorrectly formatted.

In the above example, the navigational response is the page itself.

If there is no exception event handler, or the exception event handler returns null (or is void), then the exception will be passed to the RequestExceptionHandler service, which (in the default configuration) will render the exception page.

Triggering Events

JumpStart Demo

AJAX Components CRUD

If you want your own component to trigger events, just call the `triggerEvent` method of ComponentResources from within your component class.

For example, the following triggers an "updateAll" event. A containing component can then respond to it, if desired, with an "onUpdateAll()" method in its own component class.

Your component class (partial)

```
@Inject
ComponentResources componentResources;
 ...
private void timeToUpdate() {
    boolean wasHandled = componentResources.triggerEvent("updateAll", null, null);
    if (wasHandled) {
        ...
    }
}
```

The third parameter to triggerEvent is a ComponentEventCallback, which you'll only need to implement if you want to get the return value of the handler method. The return value of triggerEvent() says if the event was handled or not.