# Component Parameters

**Component parameters** are the primary means for a component instance and its container to communicate with each other. Parameters are used to *configure* component instances.

In the following example, `page` is a parameter of the `pagelink` component. The page parameter tells the pagelink component which page to go to when the user clicks on the rendered hyperlink:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <t:pagelink page="Index">Go Home</t:pagelink>
</html>
```

A component may have any number of parameters. Each parameter has a specific name, a specific Java type (which may be a primitive value), and may be *optional* or *required*.

Within a component class, parameters are declared by using the @Parameter annotation on a private field, as we'll see below.

### Related Articles

- Component Parameters
- Component Templates
- Component Classes
- Templating and Markup FAQ
- Page And Component Classes FAQ
- Supporting Informal Parameters
- Default Parameter
- Enum Parameter Recipe
- Component Cheat Sheet

## Parameter Bindings

In Tapestry, a parameter is not a slot into which data is pushed: it is a *connection* between a field of the component (marked with the @Parameter annotation) and a property or resource of the component's container. (Components can be nested, so the container can be either the page or another component.)

The connection between a component and a property (or resource) of its container is called a *binding*. The binding is two-way: the component can read the bound property by reading its parameter field. Likewise, a component that updates its parameter field will update the bound property.

This is important in a lot of cases; for example a TextField component can read *and update* the property bound to its value parameter. It reads the value when rendering, but updates the value when the form is submitted.

The component listed below is a looping component; it renders its body a number of times, defined by its `start` and `end` parameters (which set the boundaries of the loop). The component can update a `result` parameter bound to a property of its container; it will automatically count up or down depending on whether `start` or `end` is larger.

**Contents**

```
package org.example.app.components;

import org.apache.tapestry5.annotations.AfterRender;
import org.apache.tapestry5.annotations.Parameter;
import org.apache.tapestry5.annotations.SetupRender;

public class Count
{
    @Parameter (value="1")
    private int start;

    @Parameter(required = true)
    private int end;

    @Parameter
    private int result;

    private boolean increment;

    @SetupRender
    void initializeValues()
    {
        result = start;
        increment = start < end;
    }

    @AfterRender
    boolean next()
    {
        if (increment)
        {
            int newResult = value + 1;

            if (newResult <= end)
            {
                result = newResult;
                return false;
            }
        }
        else
        {
            int newResult= value - 1;
            if (newResult>= end)
            {
                result = newResult;
                return false;
            }
        }
        return true;
    }
}
```

The name of the parameter is the same as field name (except with leading "_" and "$" characters, if any, removed). Here, the parameter names are "start", "end" and "result".

The component above can be referenced in another component or page template, and its parameters *bound*:

```
<html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <p> Merry Christmas: <t:count end="3"> Ho! </t:count>
    </p>
</html>
```

The end attribute is used to *bind* the end parameter of the Count component. Here, it is being bound to the string value "3", which is automatically coerced by Tapestry into the int value, 3.

Any number of parameters may be bound this way.

Component parameters may also be bound using the @Component annotation inside the component class. (Where conflicts occur, the parameters bound using the Component annotation will take precedence over parameter bindings in the template.)

# Binding Expressions

The value inside the template, "3" in the previous example, is a *binding expression*.

By placing a prefix in front of the value, you can change how Tapestry interprets the remainder of the expression (the part after the colon):

| Prefix | Description |
|---|---|
| asset: | The relative path to an asset file (which must exist) |
| block: | The id of a block within the template |
| component: | The id of another component within the same template |
| context: | Context asset: path from context root |
| literal: | A literal string |
| nullfieldstrategy: | Used to locate a pre-defined NullFieldStrategy |
| message: | Retrieves a string from the component's message catalog |
| prop: | A property expression to read or update |
| symbol: | Used to read one of your symbols |
| translate: | The name of a configured translator |
| validate: | A *validator specification* used to create some number of field validators |
| var: | Allows a render variable of the component to be read or updated |

Most of these binding prefixes allow parameters to be bound to read-only values; for instance a parameter bound to "message:some-key" will see the message for "some-key" from its container's message catalog in the field. If the component tries to update the parameter (by setting the value of the field), a runtime exception will be thrown to indicate that the value is read-only.

Only prop: and var: binding prefixes are updateable (but you must *not* use the ${..} syntax here; see the warning below).

Each parameter has a default prefix, defined by the component, that is used when the prefix is not provided. The most common are "literal:" and "prop:".

A *special prefix*, "inherit:", is used to support Inherited Parameter Bindings.

## Render Variables: Bindings

Components can have any number of *render variables*. Render variables are named values with no specific type (they are ultimately stored in a Map). Render variables are useful for holding simple values, such as loop indices, that need to be passed from one component to another.

For example, the following template code:

```
<ul>
    <li t:type="loop" source="1..10" value="index">${index}</li>
</ul>
```

and the following Java code:

```
@Property
private int index;
```

... could be rewritten as just:

```
<ul>
    <li t:type="loop" source="1..10" value="var:index">${var:index}</li>
</ul>
```

In other words, you don't have to define a property in the Java code. The disadvantage is that render variables don't work with the property expression syntax, so you can pass around a render variable's *value* but you can't reference any of the value's properties.

Render variables are automatically cleared when a component finishes rendering.

Render variable names are case insensitive.

### Property: Bindings

Main Article: Property Expressions

The "prop:" binding prefix indicates a property expression binding.

Property expressions are used to link a parameter of a component to a property of its container. Property expressions can navigate a series of properties and/or invoke methods, as well as several other useful patterns.

The default binding prefix in most cases is "prop:", which is why it is usually omitted.

### Validate: Bindings

Main Article: Forms and Validation

The "validate:" binding prefix is highly specialized. It allows a short string to be used to create and configure the objects that perform input validation for form control components, such as TextField and Checkbox.

The string is a comma-separated list of *validator types*. These are short aliases for objects that perform the validation. In many cases, the validation is configurable in some way: for example, a validator that enforces a minimum string length needs to know what that minimum string length is. Such values are specified after an equals sign.

For example: `validate:required,minLength=5` would presumably enforce that a field requires a value, and with at least five characters.

### Translate: Bindings

The "translate:" binding prefix is also related to input validation. It is the name of a configured Translator, responsible for converting between server-side and client-side representations of data (for instance, between client-side strings and server-side numeric values).

The list of available translators is configured by the TranslatorSource service.

### Asset: Bindings

Main Article: Assets

Assets bindings are used to specify Component Parameters, static content served by Tapestry. By default, assets are located relative to the component class in your packaged application or module. This can be overridden by prefixing the path with "context:", in which case, the path is a context path from the root of the web application context. Because accessing context assets is relatively common, a separate "context:" binding prefix for that purpose exists (described below).

### Context: Bindings

Main Article: Assets

Context bindings are like asset bindings, but the path is *always* relative to the root of the web application context. This is intended for use inside templates, i.e.:

```
<img src="${context:images/icon.png}"/>
```

Tapestry will adjust the URL of the image so that it is processed by Tapestry, not the servlet container. It will gain a URL that includes the application's version number, it will have a far-future expires header, and (if the client supports it) its content will be compressed before being sent to the client.

# @Parameter annotation

### Required Parameters

Parameters that are required **must** be bound. A runtime exception occurs if a component has unbound required parameters.

```
public class Component{

  @Parameter(required = true)
  private String parameter;

}
```

Sometimes a parameter is marked as required, but may still be omitted if the underlying value is provided by some other means. This is the case, for example, with the Select component's value parameter, which may have its underlying value set by contributing a ValueEncoderSource. Be sure to read the component's parameter documentation carefully. Required simply enables checks that the parameter is bound, it does not mean that you must supply the binding in the template (or @Component annotation).

### Optional Parameters

Parameters are optional unless they are marked as required.

You may set a default value for optional parameters using the `value` element of the @Parameter annotation. In the Count component above, the start parameter has a default value of 1. That value is used unless the start parameter is bound, in which case, the bound value supersedes the default.

### Parameter Binding Defaults

The @Parameter annotation's `value` element can be used to specify a *binding expression* that will be the default binding for the parameter if otherwise left unbound. Typically, this is the name of a property that that will compute the value on the fly.

```
@Parameter(value="defaultMessage") // or, equivalently, @Parameter("defaultMessage")
private String message;

@Parameter(required=true)
private int maxLength;

public String getDefaultMessage(){
        return String.format("Maximum field length is %d.", maxLength);
}
```

As elsewhere, you may use a prefix on the value. A common prefix to use is the "message:" prefix, to access a localized message.

### Parameter Caching

Reading a parameter value can be marginally expensive (because of type coercion). Therefore, it makes sense to cache the parameter value, at least while the component is actively rendering itself.

In rare cases, it is desirable to defeat the caching; this can be done by setting the cache() attribute of the @Parameter annotation to false.

## Don't use the ${...} syntax!

Main Article: Expansions

You generally should *not* use the Template Expansion syntax, ${...}, within component parameter bindings. Doing so results in the property inside the braces being converted to an (immutable) string, and will therefore result in a runtime exception if your component needs to update the value (whenever the default or explicit binding prefix is `prop:` or `var:`, since such component parameters are *two-way* bindings).

**This is right**

```
<t:textfield t:id="color" value="color"/>
```

**This is wrong**

```
<t:textfield t:id="color" value="${color}"/>
```

The general rule is, only use the ${...} syntax in non-Tapestry-controlled locations in your template, such as in attributes of ordinary HTML elements and in plain-text areas of your template.

**This is right**

```
<img src="${context:images/banner.png}"/>
```

**This is wrong**

```
<img src="context:images/banner.png"/>
```

# Informal Parameters

Main Article: Supporting Informal Parameters

Many components support *informal parameters*, additional parameters beyond the formally defined parameters. Informal parameters will be rendered into the output as additional attributes on the tag rendered by the component. Generally speaking, components that have a 1:1 relationship with a particular HTML tag (such as <TextField> and <input> will support informal parameters.

Only components whose class is annotated with @SupportsInformalParameters will support informal parameters. Tapestry silently drops informal parameters that are specified for components that do not have this annotation.

Informal parameters are often used to set the CSS class of an element, or to specify client-side event handlers.

The default binding prefix for informal parameters depends on *where* the parameter binding is specified. If the parameter is bound inside a Java class, within the @Component annotation, then the default binding prefix is "prop:". If the parameter is bound inside the component template, then the default binding prefix is "literal:". This reflects the fact that a parameter specified in the Java class, using the annotation, is most likely a computed value, whereas a value in the template should simply be copied, as is, into the result HTML stream.

Informal parameters (if supported) are always rendered into the output *unless* they are bound to a property whose value is null. If the bound property is null then the parameter will *not* be present at all in the rendered output.

If your component should render informal parameters, just inject the ComponentResources for your component and invoke the `renderInformalParameters()` method. See Supporting Informal Parameters for an example of how to do this.

# Parameters Are Bi-Directional

Parameters are not simply variables; each parameter represents a connection, or *binding*, between a component and a property of its container. When using the prop: binding prefix, the component can force changes *into* a property of its container, just by assigning a value to its own instance variable.

```
<t:layout xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <p> Countdown:
        <t:count start="5" end="1" result="index">
          ${index} ...
        </t:count>
    </p>
</t:layout>
```

Because the Count component updates its result parameter (the `result` field), the index property of the containing component is updated. Inside the Count's body, we output the current value of the index property, using the expansion `${index}`. The resulting output will look something like:

```
  <p> Countdown: 5 ... 4 ... 3 ... 2 ... 1 ... </p>
```

(Though the whitespace will be quite different.)

The relevant part is that components can read fixed values, or *live* properties of their container, and can *change* properties of their container as well.

# Inherited Parameter Bindings

A special prefix, "inherit:" is used to identify the name of a parameter of the containing component. If the parameter is bound in the containing component, then it will be bound to the same value in the embedded component.

If the parameter is not bound in the containing component, then it will not be bound in the embedded component (and so, the embedded component may use a default binding).

Inherited bindings are useful for complex components; they are often used when an inner component has a default value for a parameter, and the outer component wants to make it possible to override that default.

**Index.tml**

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
  <body>
    <div t:type="layout" t:menuTitle="literal:The Title">
      ...
    </div>
  </body>
</html>
```

**Layout.tml**

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">

        <div t:type="title" t:title="inherit:menuTitle"></div>

        <t:body />

</t:container>
```

**Title.java**

```
package org.example.app.components;

import org.apache.tapestry5.annotations.Parameter;

public class Title {

        @Parameter
        private String title;

}
```

# Computed Parameter Binding Defaults

In *rare* cases, you may want to compute the binding to be used as a parameter default. In this case, you will provide a *default binding method*, a method that takes no parameters. The returned value is used to bind the parameter. The return value may be a Binding instance, or it may be a simple value (which is more often the case).

The method name is "default" plus the capitalized name of the parameter.

Using this approach, the previous example may be rewritten as:

```
@Parameter
private String message;

@Parameter(required=true)
private int maxLength;

@Inject
private ComponentResources resources;

@Inject
private BindingSource bindingSource;

Binding defaultMessage()
{
  return bindingSource.newBinding("default value", resources, "basicMessage");
}

public String getBasicMessage()
{
  return String.format("Maximum field length is %d.", maxLength);
}
```

In this example, a property expression, "basicMessage", is used to access the message dynamically.

Alternately, the previous example may be written even more succinctly as:

```
@Parameter
private String message;

@Parameter(required=true)
private int maxLength;

@Inject
private ComponentResources resources;

String defaultMessage()
{
  return String.format("Maximum field length is %d.", maxLength);
}
```

This form is more like using the "literal:" binding prefix, except that the literal value is computed by the defaultMessage() method.

Obviously, this is a lot more work than simply specifying a default value as part of the @Parameter annotation. In the few real cases where this is approach is used, the default binding method will usually deduce a proper binding, typically in terms of the component's id. For example, the TextField component will deduce a value parameter that binds to a property of its container with the same name.

A default binding method will *only* be invoked if the @Parameter annotation does not provide a default value.

# Unbound Parameters

If a parameter is not bound (and is optional), then the value may be read or *updated* at any time.

Updates to unbound parameters cause no side effects. In the first example, the value parameter of the Count component is not bound, and this is perfectly valid.

Note: updates to such fields are temporary; when the component *finishes rendering*, the field will revert to its default value.

# Parameter Type Coercion

Main Article: Parameter Type Coercion

Tapestry includes a mechanism for coercing types automatically. Most often, this is used to convert literal strings into appropriate values, but in many cases, more complex conversions will occur. This mechanism is used for component parameters, such as when an outer component passes a literal string to an inner component that is expecting an integer.

You can easily contribute new coercions for your own purposes.

# Parameter Names

By default, Tapestry converts from the field name to the parameter name, by stripping off leading "$" and "_" characters.

This can be overridden using the name() attribute of the @Parameter annotation.

# Determining if Bound

In rare cases, you may want to take different behaviors based on whether a parameter is bound or not. This can be accomplished by querying the component's resources, which can be injected into the component using the @Inject annotation:

```
public class MyComponent
{
  @Parameter
  private int myParam;

  @Inject
  private ComponentResources componentResources;

  @BeginRender
  void setup()
  {
      if (componentResources.isBound("myParam"))
      {
        . . .
      }
  }
}
```

The above sketch illustrates the approach. Because the parameter type is a primitive type, int, it is hard to distinguish between no binding, and binding explicitly to the value 0.

The @Inject annotation will inject the ComponentResources for the component. These resources are the linkage between the Java class you provide, and the infrastructure Tapestry builds around your class. In any case, once the resources are injected, they can be queried.

# Publishing Parameters

Often when creating new components from existing components, you want to expose some of the functionality of the embedded component, in the form of exposing parameters of the embedded components as parameters of the outer component.

In Tapestry 5.0, you would define a parameter of the outer component, and use the "inherit:" binding prefix to connect the inner component's parameter to the outer component's parameter. This is somewhat clumsy, as it involves creating an otherwise unused field just for the parameter; in practice it also leads to duplication of the documentation of the parameter.

In Tapestry 5.1 and later, you may use the publishParameters attribute of the @Component annotation. List one or more parameters separated by commas: those parameters of the inner/embedded component become parameters of the outer component. You should **not** define a parameter field in the outer component.

**ContainerComponent.tml**

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
<t:pageLink t:id="link">Page Link</t:pageLink>
</t:container>
```

**ContainerComponent.java**

```
public class ContainerComponent{
    @Component(id="link", publishParameters="page")
    private PageLink link;
}
```

**Index.tml**

```
<t:ContainerComponent t:id="Container" t:page="About" />
```

There are still cases where you want to use the "inherit:" binding prefix. For example, if you have several components that need to share a parameter, then you must do it the Tapestry 5.0 way: a true parameter on the outer component, and "inherit:" bindings on the embedded components. You can follow a similar pattern to rename a parameter in the outer component.

Property Expressions | User Guide | Parameter Type Coercion