# Directives and Bindings

While the CHTML Component API helps us work with components programmatically, it also allows us express those same operations declaratively as simple statements (or directives) right within component markup.

This section discusses how to declare directives in CHTML without writing application code, and how reactivity works automatically.

## Directives

Directives are executable statements declared right at markup level. They are composed in the *chtml-directives* attribute.

<body chtml-role=”app” chtml-directives=”el.append:‘Hello World!’”></body>

A directive looks like a programmatic expression but follows a key/value format like CSS rules. This syntax is called JSEN Parameters (JSEN-P) – derived from the JavaScript Expression Notation (JSEN).

In the declaration above, *el.append* is the directive, and the expression *‘Hello World!’* is the *argument*. Although the directive and its argument look separated, they really do end up as JSEN expressions. So *el.append:‘Hello World!’* later becomes *el.append(‘Hello World!’).* (As a general rule, joining a directive and its arguments should produce a valid JSEN expression.)

Now multiple directives are semicolon-separated. And a trailing semicolon is allowed.

<body chtml-role=”app” chtml-directives=”el.prepend:‘Hello ’; el.append:’World!’;”></body>

Paths are used to address nodes – written in dot (.) or bracket [] notation, and usually with *el* as the endpoint.

<body chtml-role=”app” chtml-directives=”content.el.append:‘Hello World!’; el.append:‘Thanks for visiting!’”>

<div chtml-role=”app-content”></div>

</body>

With CHTML’s choice of JSEN, we stand to enjoy writing dynamic expressions the way they naturally lend themselves to be written. This is good news as there is no concept of string interpolations and some funny template language that would falsely attempt to replicate the same thing.

### Arguments

Arguments are pure JSEN expressions. They are normally enclosed in parentheses. But this is optional for single-argument directives.

<body chtml-role=”app” chtml-directives=”el.prepend:’Hello World!’; content.el.append:(‘Thanks ‘, ‘for visiting!’)”></body>

Here are more JSEN expressions. And the JSEN docs contain even richer examples.

// Math expression.

el.append:‘Summing 2 and 3 gives us ‘ + (2 + 3);

// (Deep) references to properties of the component.

el.append:‘The color of this text is ‘ + el.style.color;

// Method calls.

el.append:‘My name is ‘ + el.tagName.toLowerCase();

// Assertions, Comparison and Conditionals.

el.append:el.hasAttribute(‘is‘) || el.tagName.indexOf(‘-‘) > -1 ? ‘I am a custom element.’ : ‘I am a standard element.’;

// Array and Object constructs.

el.animate:[{color:’red’}, {color:’blue’},];

// ES6 Function constructs.

content.el.addEventListener:(‘click’, (e) => {e.target.animate([{color:’red’}, {color:’blue’},])});

// ES6 Function constructs for a loop.

content.el.children.forEach:((child, index) => {child.append(index)});

### Conditionals

Directives can be conditional expressions. This is possible in a number of ways that are sure to end up as valid JSEN expressions.

**Using Dynamic Node Paths.** By nature, JSEN path expressions like *el.append* and *content.el.append* can be rephrased as dynamic expressions using the bracket notation.

For example, to dynamically decide between append and prepend based on the presence of a node, we could say:

// If the content node exists, append, otherwise, prepend.

el[content ? ‘append’ : ‘prepend’]:’Thanks for visiting!’;

// This ends up as a valid JSEN Call expression

el[content ? ‘append’ : ‘prepend’](’Thanks for visiting!’);

**Using Assertions.** Just before terminating a directive in a JSEN Call expression like *el.append(),* we could make certain assertions that should either validate or fail the entire expression. These expressions use the logical AND (&&) and OR (||) operators.

For example, if we wanted to fail an append operation on the absence of a node, we could say:

// This validates only if the content node is available.

// Fails if not

content && el.append:’Thanks for visiting!’;

// This ends up as a valid JSEN Assertion expression

content && el.append(’Thanks for visiting!’);

// Conversely, this fails if the content node is available.

// Validates if not.

content || el.append:’Thanks for visiting!’;

**Using Ternary Operators.** Parenthesized expressions can form the basis of a Call expression. So we can place a conditional expression in here.

For example, to dynamically choose the node on which to finally execute a directive, we could say:

// Append to the content node, if available.

// Otherwise, to the root element.

(content ? content.el : el).append:’Thanks for visiting!’;

// This ends up as a valid JSEN Call expression

(content ? content.el : el).append(’Thanks for visiting!’);

### Chaining

Directives can be written as chained operations based on the return type of each operation.

Below, we are taking the first node of some component to the end of the child list. In the same statement, we’re adding some content to this node.

// el.appendChild() will return the appended childNode,

// and the .append() method that comes next will be called on this childNode.

el.appendChild:(firstNode.el).append(‘I used to be the first node!’);

Notice that although el.appendChild() is a single-argument directive, parenthesis for its argument has to be added. As it is, parentheses are required for arguments of the base directive for chaining to be syntactically correct.

Chaining especially comes into play when working with methods that return JavaScript Promises. Here is how we could use some imaginary animate method (anim()) that returns a Promise.

// This will fade the element out and remove it afterward

el.anim:([{opacity:1}, {opacity:0},], {duration:600}).then(() => el.remove());

#### Gotcha!

In some cases, additional care would need to be taken with parentheses to differentiate between argument-level chaining and directive-level chaining. In the case below, we intend to display the value of either of two attributes; so we construct the argument as a conditional expression.

// Since there is only one argument, we don’t need parentheses. This works as expected

el.append:condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’);

Suppose we want the finally-chosen attribute value to be upper-cased. Then we will need to group the conditional construct as one expression and call the .toUpperCase() method on the final result of the conditional expression. But notice in the first code below how our intended argument-level method call has turned out be directive-level chaining.

// PROBLEM: the parentheses for the conditional expression has constituted argument-list parentheses and .toUpperCase() will now be called on the return value of .append()

el.append:(condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase();

// The entire expression above will end up as… (directive-level chaining)

el.append(condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase();

// SOLUTION: To avoid the problem above, this complex argument has to be properly enclosed in explicit argument-list parentheses, even though this would have been optional

el.append:((condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase());

// The entire expression above will end up as…

el.append((condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase());

To disable directive-level chaining, explicitly set the params.chainableDirectives to false.

// Disable params.chainableDirectives at instantiation time

var component = Chtml.from(el, {chainableDirectives:false});

// Disable this globally

Chtml.params.chainableDirectives = false;

// Now the expression below is properly understood to be a single-argument directive that will automatically be qualified with argument-list parentheses

el.append:(condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase();

// So this later ends up as…

el.append((condition ? el.getAttribute(‘attribute-a’) : el.getAttribute(‘attribute-b’)).toUpperCase());

### Cascading

**Directives Are Cascaded Parameters.** In their key/value form, directives look like CSS rules. Interestingly, they really do work like CSS; specifically in the cascaded nature of CSS. Cascading answers cases where certain directives will need to, or happen to, be declared more than once.

<div chtml-role=”app-content” chtml-directives=”el.append:‘Hello World!’; el.append:‘Hello World! Thanks for visiting!’”>

</div>

Cascading rules are covered in the section for cascading.

**Directives Can Be Written in a Params Sheet.** Inline directives (directives written in the *chtml-directives* attribute), like inline CSS, can quickly begin to look congested. CHTML lets us declare directives in a *Data Block* type of script element where things can sit more comfortably, just as with a style sheet. This script element is called Cascaded Params Sheet (JSEN-P) and is placed at the root of the component.

<body chtml-role=”app”>

<script type=”text/jsen-p”>

@bindings {

el.append:‘Thanks for visiting!’;

content.el.append:‘Hello World!’;

}

</script>

<div chtml-role=”app-content”></div>

</body>

Where directives are written both inline and in JSEN-P, the combined parameters are used in a cascaded manner, usually with inline parameters taking precedence as governed by JSEN-P cascading rules.

## Bindings

In CHTML, every directive is a binding expression. Under the hood, after initially executing a directive, CHTML begins to observe all the variables (or references) in the expression with a view to re-executing the directive in the event of a change. See how the directive below is bound behind the scene.

<!—This directive displays content from a data component -->

<div chtml-directives=”el.html:$.fname + ‘ ‘ + $.lname;”></div>

// The directive is initially executed as… something like…

el.html($.fname + ‘ ‘ + $.lname);

// Next, the references made in the expression are obtained and observed

component.$observe([‘$.fname’, ‘$.lname’, ‘el.html’/\*although this is not a regular reference\*/], (changes) => {

// The directive is re-executed as… something like…

var returnValue = el.html(changes[0] + ‘ ‘ + changes[1]);

// Returning false will prevent other directives bound to this same change from being executed

// so we avoid that.

// If returnValue is a Promise, this Promise will be added to the list of Promises that is seen in the application via event.promises.

if (returnValue !== false) {

return returnValue;

}

});

With this understanding of how things work, we can declaratively implement our earlier Author Display component. Notice how the three parts of the component are each implemented. First, the directive that fades out and fades in the author element on *headsup=”publishing”* and *headsup=”published”* respectively. (This directive is using an imaginary *anim()* method that returns a Promise while playing an animation.) Next, the directive that updates the DOM. Then, the directive that updates the State component (Author).

<div chtml-role=”article” id=”article” chtml-directives=”author.el.anim:($.author.headsup === ‘publishing’ ? [{opacity:1}, {opacity:0}] : [{opacity:0}, {opacity:1}], {duration:600}); author.el.html:$.author.fname + ‘ ‘ + $.author.lname; el.on:(‘dblclick’, () => $.author.$next()); ”>

<div>

<div chtml-role=”article-author user”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

<div>

<div chtml-role=”article-description”></div>

</div>

</div>

</div>

In summary, functionalities are built in the application, made available in the CHTML component as plugged-in state components, and instructed as directives at markup.

As seen, directives are simply what they are “directives” – a declarative way to make the UI dynamic in the context of an application. Obviously, they belong in the Presentation layer not the Application layer, should we see things from an architectural perspective.

Additionally, the loosely-coupled nature of the UI and the application offers us a wonderful way to build apps. These layers of the app can be built in isolation and progressively on a common Actions/Reactions API contract.