# Component Composition

The essence of componentization is reusability and composability. It just makes us more efficient. So this section covers how to define components for later use. In CHTML, the idea is to define once and use everywhere.

We start with the question of where to place things or where components should live – especially important as we might need to work with a handful of them. Then we’ll see different ways to reuse components.

## Namespaces and Templates, Layout and Bundling

In CHTML, components intended for later use are defined together in a <template> element. They may, however, be defined individually in separate files on the server with a view to bundling them into a <template> element – whatever makes them live comfortably and easy to maintain.

### Namespaces and Templates

Components defined together in a <template> element are each assigned a unique namespace. Namespaces organize them in virtual categories and help us reference them and reuse them.

Namespaces are like file paths. They are assigned on the *chtml-ns* attribute.

<template>

<div chtml-role=”article” chtml-ns=”html/content/article”></div>

</template>

Here we’ve placed article under a category named content. There could be other types in this category. And we can have subcategories – to as much as organization requires.

<template>

<!-- Displays article -->

<div chtml-role=”article” chtml-ns=”html/content/article/readonly”></div>

<!-- Allows editing -->

<div chtml-role=”article” chtml-ns=”html/content/article/editable”></div>

</template>

Note that the way namespaces are used in CHTML demands that a namespace be of, at least, two parts. So a single word like *content* isn’t a valid namespace; something like *html/content* is.

Templates can be defined either as elements in the <head> section of a document or as external HTML files that can be linked to the document. For CHTML to automatically pick them up, the *chtml-bundle* type of <template> must be used.

<html>

<head>

<template is=”chtml-bundle”>

<div chtml-ns=”html/content/article”></div>

</template>

</head>

<body></body>

</html>

To link a bundle from the server, the *src* attribute is used. This attribute isn’t defined as part of the <template> element and is entirely a CHTML’s way of loading template content as the standard <link type=”import”> element no longer exists.

<!—file: bundle.html -->

<div chtml-ns=”html/content/article/readonly”></div>

<div chtml-ns=”html/content/article/editable”></div>

<!—app -->

<html>

<head>

<!—The *src* attribute demands that the <template> element be empty -->

<template is=”chtml-bundle” src=”/bundle.html”></template>

</head>

<body></body>

</html>

Now multiple bundles can be used – whether defined internally or externally. They will all work together in a cascaded manner, as covered in a subsequent section for bundle cascading. This opens the door to building rich applications with bundles from multiple sources.

### Layout and Bundling

The folder-based layout approach takes things from virtual to actual namespacing. Here, components are defined in stand-alone HTML files placed in a hierarchy of folders.

Here is the equivalent layout of the namespaced article components above:

content

|-- article

|-- readonly.html

|-- editable.html

Now, a component’s namespace naturally becomes the combination of its path and filename; and the *chtml-ns* attribute isn’t required anymore.

In a straightforward process called bundling, it is easy to put all our components back to a <template> file. This time, namespaces are automatically derived and assigned to each component.

## Imports and Recomposition, Inheritance and Cascading

CHTML makes it seamless to reuse a defined component either in whole or in part. This helps us build entire applications with much fewer components.

### Imports

CHTML imports are a way to place anything anywhere in a HTML document. The idea is to use a temporary element to import the needed element or component. This temporary element is the *chtml-import* element.

<chtml-import chtml-ns=”html/content/article/readonly”></chtml-import>

Imports use the *chtml-ns* attribute to reference their source element. The import element itself gets replaced by the imported element; so at best, an import element is only a placeholder.

Using imports, we can easily compose a larger component with smaller components. An article component, for example, can now simply import a user component as its author node.

<template is=”chtml-bundle”>

<div chtml-role=”user” chtml-ns=”html/badge/user“>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

<div chtml-role=”article” chtml-ns=”html/content/article/readonly“>

<div>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user”></chtml-import>

<div>

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

</div>

</div>

</div>

</template>

Actually, anything can be imported – anything tag-based – as long it is placed in a CHTML bundle and has a namespace. So although we like to refer to things as components, a more general name for them is module – CHTML module. A bundle is composed of modules.

On this general note, a bundle could look like:

<template is=”chtml-bundle”>

<!—components -->

<div chtml-role=”user” chtml-ns=”html/badge/user“>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

<!—Media: images, videos, etc -->

<img src=”/assets/img/brand/logo.png” chtml-ns=”img/brand/logo” />

<!— Media with data-URLs. -->

<!-- Good for preloading resources -->

<img src=”data:image/png,%89PNG%0D%0A…” chtml-ns=”img/brand/logo2” />

<!—SVG, possibly SVG icons -->

<svg chtml-ns=”svg/brand/logo/24x24” >

<path />

</svg>

<!—Stylesheets -->

<style type=”text/css” chtml-ns=”css/badge” >

/\* rules \*/

</style>

</template>

Now we can import them as needed:

<html>

<head>

<!—

If we had created the bundle as a html file, we link to it here

<template is=”chtml-bundle” src=”/path/to/bundle.html”></template>

-->

</head>

<body>

<!—Place an image here -->

<chtml-import chtml-ns=”img/brand/logo2”></chtml-import>

</body>

</html>

#### Imports Ondemand

By default, imports are resolved as they land in the DOM. This makes sense most of the times. At other times, we may want imports to resolve on-demand – just at the time they are accessed in an application. This is achieved with the *ondemand* Boolean attribute.

When implemented as a node in a component, for example, imports of this type are provisional and get resolved on first access to the node.

<div chtml-role=”article”>

<chtml-import ondemand chtml-role=”article-author” chtml-ns=” html/badge/user”></chtml-import>

</div>

#### Shadow Imports

It is possible to import components directly into an element’s shadow DOM. This form of import is called shadow import. Shadow imports are qualified with the *shadow* Boolean attribute. An import’s shadow host is its immediate parent.

The import element itself gets replaced and is never part of the Shadow DOM nor the light DOM. Imported components are never visible anywhere in the DOM but live hidden in the host element’s shadow DOM.

<div id=”host”>

<chtml-import shadow chtml-ns=” html/badge/user”></chtml-import>

</div>

We could even send some CSS into the shadow DOM for the component.

<div id=”host”>

<chtml-import shadow chtml-ns=” html/badge/user”></chtml-import>

<chtml-import shadow chtml-ns=”css/badge”></chtml-import>

</div>

### Recomposition

Import elements can do more than just place a component. They can be empowered to recompose the component being imported. We do this by predefining contents on the import element with a view to having these on the component they import. The import element is replaced as usual, but this time, with a richly composed component.

For example, if we wanted an incoming component to have an ID, we would set this on the import element.

<chtml-import id=”some-id” chtml-ns=”html/badge/user”></chtml-import>

We could import the same component in another place with a different ID.

<chtml-import id=”some-other-id” chtml-ns=”html/badge/user”></chtml-import>

Recomposition helps us fine-tune imported components to fit perfectly into every new use-case. The component below, for example, could be recomposed on every import we make of it.

<template is=”chtml-bundle”>

<div chtml-role=”user” chtml-ns=”html/badge/user“ style=”color:black”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</template>

#### Attributes and Params Recomposition

Certain attributes can be predefined on an import element with a view to having them on the imported component. For some types of attribute, values are replaced on the component; for others, values are merged.

**Classes and Inline CSS Are Merged.** Classes predefined on an import element are merged with any existing class-list on the component. Also, any inline CSS styles are appended to the component’s style attribute, and CSS cascading takes effect, with styles from the import element taking priority.

<div chtml-role=”article”>

<chtml-import class=”class1 class2” style=”color:blue” chtml-ns=”components/badge/user”></chtml-import>

</div>

The final composition gives us:

<div chtml-role=”article”>

<div chtml-role=”user” class=”class1 class2” style=”color:black; color:blue” chtml-ns=”components/badge/user“>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</div>

**Roles Are Merged.** Any ARIA roles (in the *role* attribute) and CHTML roles (in the *chtml-role* attribute) predefined on an import element are merged with existing component roles. This adapts the imported component for additional roles to play.

Below, we’re importing the *user* component to be an *article*’s *author* node. Notice that this works because roles are merged. (We’ve done this a number of times before now, but the details might not have been obvious.)

<div chtml-role=”article”>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user”></chtml-import>

</div>

The final composition gives us:

<div chtml-role=”article”>

<div chtml-role=”article-author user” chtml-ns=”html/badge/user“ style=”color:black”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</div>

**Cascaded Params Are Merged.** Both inline params (the *chtml-related* and *chtml-directives* attributes) and the Cascaded Params Sheet (JSEN-P) that may be predefined on an import element are composed into the imported component. So while a component would normally be defined with parameters for general use-case, we can compose new parameters into each import we make of it.

For **inline parameters**, parameters on the import element are appended to the component’s existing parameters or created on the component for the first time. When appended, cascading takes effect naturally and newer parameters get to override existing identical parameters, as governed by JSEN-P cascading rules.

This is what happens below.

<template is=”chtml-bundle”>

<div chtml-role=”user” chtml-ns=”html/badge/user“ chtml-directives=”name.el.append:data.first\_name”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</template>

<div chtml-role=”article”>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user” chtml-related=”extra\_node:\_.some\_node” chtml-directives=”name.el.append:data.first\_name.toUpperCase() !important”></chtml-import>

</div>

The final composition would give us:

<div chtml-role=”article”>

<div chtml-role=”user” chtml-ns=”html/badge/user“ chtml-related=”extra\_node:\_.some\_node” chtml-directives=”name.el.append:data.first\_name; name.el.append:data.first\_name.toUpperCase() !important”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</div>

For **sheet-based parameters**, an import element’s JSEN-P is either created on the component for the first time or its parameters are appended to the component’s existing JSEN-P. When appended, cascading takes effect naturally and newer parameters get to override existing identical parameters, as governed by JSEN-P cascading rules.

This is what happens below.

<template is=”chtml-bundle”>

<div chtml-role=”user” chtml-ns=”html/badge/user”>

<script type=”text/jsen-p”>

@bindings {

name.el.append:data.first\_name;

}

</script>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</template>

<div chtml-role=”article”>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user”>

<script type=”text/jsen-p”>

@related {

extra\_node:\_.some\_node;

}

@bindings {

name.el.append:data.first\_name.toUpperCase() !important;

}

</script>

</chtml-import>

</div>

The final composition would give us:

<div chtml-role=”article”>

<div chtml-role=”user” chtml-ns=”html/badge/user“>

<script type=”text/jsen-p”>

@related {

extra\_node:\_.some\_node;

}

@bindings {

name.el.append:data.first\_name;

name.el.append:data.first\_name.toUpperCase() !important;

}

</script>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name”></div>

</div>

</div>

**Other Attributes Are Set If Not Already Exists.** Other attributes found on an import element (except the *chtml-ns* attribute) are re-created on the imported component if not found on the element.

##### Managing Recomposition

As a general rule, everything an import element has gets stripped and composed into the component it imports. But this recomposition can be controlled. There are two ways.

**Using the Norecompose Attribute.** To object to recomposition on an element, the *norecompose* attribute can be used. If without a value, this attribute disables recomposition completely. Setting its value to \* achieves the same thing. A list of attribute names may, however, be set to specify the attributes that should be ignored during composition.

<template is=”chtml-bundle”>

<div chtml-role=”user” chtml-ns=”html/badge/user” chtml-directives=”el.prepend:data.names” style=”color:blue” norecompose=”chtml-directives @JSEN-P style”></div>

</template>

To ignore sheet-based parameters recomposition, the *@JSEN-P* keyword could be added to the *norecompose* list.

The norecomposition directive may also be set generally for all modules in a bundle.

<template is=”chtml-bundle” norecompose=”chtml-directives @JSEN-P style”>

<div chtml-role=”user” chtml-ns=”component/badge/user” chtml-directives=”el.prepend:data.names” style=”color:blue”></div>

</template>

With a bundle-level norecompose directive in place, modules will now be imported without having the listed attributes recomposed. A module-level norecompose directive could still be set to list additional attributes specific to the module.

**Using a Callback.** As we will see in the section for CHTML API, the entire recomposition process can be intercepted, or hooked-into, using a callback. This offers the ability to handle the composition of other types of attributes that would have been simply transferred by default.

#### Node-To-Node Recomposition

Nodes in a component can be replaced on each import we make of it. We do this by creating the replacement nodes on the import element ahead of the incoming component. On arrival, the imported component gets its matched nodes replaced by these replacement nodes.

This is what happens below. We import the user component into the article component, but with a replacement node (*user-name*) that features special styling.

<div chtml-role=”article”>

<div>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user”>

<div chtml-role=”user-name” style=”text-transform:uppercase; font-weight:bold”></div>

</chtml-import>

<div>

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

</div>

</div>

</div>

The final composition would give us:

<div chtml-role=”article”>

<div>

<div chtml-role=”article-author user” chtml-ns=”html/badge/user“ style=”color:black”>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-name” style=”text-transform:uppercase; font-weight:bold”></div>

</div>

<div>

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

</div>

</div>

</div>

Attributes and params recomposition also happens on each node replacement. Before they are finally replaced, attributes and parameters from component nodes are composed into replacement nodes. So replacement nodes inherit properties of the component’s original nodes.

Now a few notes apply:

* Replacement nodes must be at the root of the import element if they must be discovered.
* Replacement nodes must define a role that corresponds to an original node in source component. But imports could also have root-level elements without a role. These get copied to the root of the imported component.

##### Recursive Recomposition

When we import a component and replace its node, we perform one level of import. But we can use replacement nodes to perform another level of import – a replacement node also being an import on its own.

For example, as we import an article component into a page, we could predefine a replacement node for its *author* node, but this time, the replacement node will be an import on its own, referencing a special type of user component.

This is what happens below. In the second level of import, we’re importing a new type of user component as the article’s *author* node. This user component splits a user’s name as first name, and last name (*fname* and *lname*).

<html>

<head>…</head>

<body>

<chtml-import chtml-role=”body-main” chtml-ns=”html/content/article/readonly”>

<chtml-import chtml-role=”article-author” chtml-ns=”html/badge/user2”></chtml-import>

</chtml-import>

</body>

</html>

The final composition would give us:

<html>

<head>…</head>

<body>

<div chtml-role=”body-main article” chtml-ns=”html/content/article/readonly”>

<div chtml-role=”article-author user” chtml-ns=”html/badge/user2“>

<div chtml-role=”user-avatar”></div>

<div chtml-role=”user-fname”></div>

<div chtml-role=”user-lname”></div>

</div>

</div>

</body>

</html>

We could go even deeper to a third level, and a fourth, and as far as composition requires!

### Inheritance

In CHTML, any piece of code is considered a piece of knowledge. Inheritance provides extra opportunity to avoid costly duplication of knowledge. It’s a type of code reuse that is inherent – by virtue of where components live; it requires no extra steps to work.

#### Namespace-Based Inheritance

The namespace-based component layout in CHTML is no mere naming convention or code organization. It brings with itself the wonderful concept of inheritance.

In CHTML, components in a namespace are believed to be built off their supernamespace. So, the component at *html/content/article/readonly/dark-mode* is believed to be built off the component at *html/content/article/readonly*, which in turn, is believed to be built off the root component at *html/content/article.* Now instead of having to repeat common semantics down the namespace, CHTML makes components inherit them!

**Components Are Implicitly Composed.** Semantics from the component at a supernamespace are, by default, composed into components at each level of a namespace path.

By default, only attributes and params composition is performed, which means that nodes are not inherited; components retain their unique structural semantics.

Here is how this could look.

<template is=”chtml-bundle”>

<!—Standard, readonly article -->

<div chtml-role=”article” chtml-ns=”html/content/article/readonly” chtml-directives=”title.el.append:data.title; content.el.append:data.content”>

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

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

</div>

<!—Dark-mode, readonly article -->

<div chtml-ns=”html/content/article/readonly/dark-mode”>

<div style=”color:white; background-color:black”>

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

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

</div>

</div>

</template>

Notice that in our dark-mode edition of an article component, it was not necessary to redefine the role *article*. The bindings were also inherited.

Inheritance takes effect at the time a component is being used.

Taking things a little further, a node-to-node recomposition is also possible, which means that structural semantics can be inherited. The *chtml-import* element comes into play here, this time as a component definition on its own. Here is how this looks.

<template is=”chtml-bundle”>

<!—Dark-mode, readonly article, with a different type of *content* node -->

<chtml-import chtml-ns=”html/content/article/readonly/dark-mode/framed”>

<div chtml-role=”article-content” style=”border-color:white“></div>

</chtml-import>

</template>

Now so much has been saved!

**Namespaces Can Be Used Ahead of Implementation.** By virtue of inheritance, we would be safe to find a component deep in a namespace that is yet to be implemented, as long as fallbacks exist up the namespace hierarchy.

Right below, we’re importing an animated type of article that we’re yet to create. Maybe soon; maybe someday! But we can, at the moment, make do with the closest implement that lives up the namespace hierarchy.

<!—file: index.html -->

<html>

<head>…</head>

<body>

<chtml-import chtml-role=”body-main” chtml-ns=”html/content/article/readonly/dark-mode/animated”></chtml-import>

</body>

</html>

Now this really allows us to progressively build features into an app while using a real layout plan from the start.

#### Cross-Bundle Inheritance

When multiple bundles are defined on a document, they are all used in a cascaded manner; in the same way multiple CSS stylesheets work. This helps us reuse code across bundles from multiple sources.

Now this is how it works: when a namespace is accessed for import, all bundles are queried for the requested component and matches are gathered and composed. Components from latter bundles thus inherit components from earlier bundles. Obviously, the order of these bundles matter.

In the example below, let us assume that the first bundle was linked from a third-party. The second bundle is a local bundle that inherits a component from the first bundle and applies dark-mode styling.

<template is=”chtml-bundle”>

<!—Standard, readonly article -->

<div chtml-role=”article” chtml-ns=”html/content/article/readonly” chtml-directives=”title.el.append:data.title; content.el.append:data.content”>

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

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

</div>

<img src=”/assets/img/ui/banner.png” chtml-ns=”ui/banner” />

</template>

<template is=”chtml-bundle”>

<!—Dark-mode, readonly article -->

<chtml-import chtml-ns=”html/content/article/readonly/dark-mode” style=”color:white; background-color:black”></chtml-import>

<img src=”/assets/img/brand/logo.png” chtml-ns=”img/brand/logo” />

</template>

The two bundled images above do not live in the same namespace. So they are imported independently of each other.

But how does this work with the namespace-based inheritance? The answer lies in a wonderful type of algorithm.

#### The Inheritance Matrix

With namespace-based inheritance working from left to right and cross-bundle inheritance working top-down, a matrix is formed. On this matrix, inheritance begins from the left and moves level-by-level towards the right, but with top-down inheritance being applied at each level. This becomes clear in an example.

The two bundles below each have a pair of components. Each pair constitutes namespace-based, left-right inheritance. The two bundles themselves constitute cross-bundle, top-down inheritance. Now we have four components and three have a CSS color each. Let’s see what happens as we try to import them.

<template is=”chtml-bundle”>

<div style=”color:blue” chtml-ns=”html/base1/base2”></div>

<div style=”color:red” chtml-ns=”html/base1/base2/red”></div>

</template>

<template is=”chtml-bundle”>

<div style=”color:yellow” chtml-ns=”html/base1/base2”></div>

<div chtml-ns=”html/base1/base2/red”></div>

</template>

If we tried importing *html/base1/base2*, all bundles will first be asked for *html/base1* to see if there is anything for the following level to inherit. The first bundled is asked first, then the second. Since no component can be found, we come empty to *html/base1/base2*. Again the first bundle is asked first, and we find a blue component, then the second bundle, and we find a yellow component. They are composed to produce a yellow component, which is what we would expect.

<div style=”color:blue; color:yellow” chtml-ns=”html/base1/base2”></div>

If we tried importing *html/base1/base2/red*, inheritance produces the same result above by the time we get to *components/base1/base2*. So we get into *html/base1/base2/red* with two colors as inheritance. Here again, the first bundle is asked first, and we find a red component, then the second bundle, and we find the final component but without a color. They are all composed to produce a red component, which is what we would expect.

<div style=”color:blue; color:yellow; color:red” chtml-ns=”html/base1/base2/red”></div>

### Cascading

The level of reusability and composition in CHTML requires being able to manage (accept or reject) duplicate parameters. This is especially important as multiple bundles from different vendors may now be connected to the same document. Happily, the concept of cascading enables us to do so.

Cascading applies to inline parameters in the *chtml-related* and *chtml-directives* attributes and those in the Cascaded Params Sheet (JSEN-P). The way these parameters are processed in CHTML is governed by the following rules; they’re for the most part, like CSS cascading rules.

#### JSEN-P Cascading Rules

**Unique and Duplicate Declarations.** Declarations of the same key and value are duplicates. Each declaration overrides a previous declaration.

The following directives, for example, are duplicates. The last one is what takes effect.

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

Declarations of the same key but with variations in value, are considered unique and are all honoured. This is what happens below.

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

Note that declarations for a component’s related nodes are later streamlined to one per node name. So, below, even though the declarations are considered unique according to JSEN-P cascading, only the last one is used.

<main chtml-role=”article” chtml-related=”comments:\_.comments; comments:\_.\_.comments”></main>

**The !important keyword.** Just as with CSS, the !important keyword can be used at the end of a declaration. It marks the declaration as of high priority and asks to replace other declarations of the same key, regardless of their value, no matter whether they live before or after it.

The first directive below takes precedence.

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

Now where two declarations of the same key both have the !important keyword, they are treated as duplicates; the latter overrides the former. This is what happens below.

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

**The !fallback keyword.** For further control, the !fallback keyword can also be used at the end of a declaration. It marks the declaration as of lower priority and asks to be used ONLY where no declarations of the same key can be found anywhere before or after it.

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

This is especially useful where newer declarations do not want to take advantage of their position and want to relinquish their rights first to any previous declarations.

Now where two declarations of the same key both have the !fallback keyword, they a treated as duplicates; the latter overrides the former. This is what happens below.

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

#### Inline and Sheet-Based Declarations

Where declarations are made both inline and in a JSEN-P on the same component, the combined declarations are used, but inline declarations are considered newer for the JSEN-P Cascading Rules.

Below, *inline* wins for related nodes, JSEN-P wins for bindings.

<main chtml-role=”article” chtml-related=”comments:\_.comments;” chtml-directives=”el.append:(‘Thanks ‘, ‘for visiting!’)”>

<script type=”text/jsen-p”>

@related {

comments:\_.\_.comments;

}

@bindings {

el.append:‘Hello World!’ !important;

}

</script>

</main>

## List Rendering

It is intuitive to make a list in CHTML! Lists can be made either programmatically or declaratively.

### Using the Chtml.populate() Instance Method

This method is used to populate a list component programmatically. It takes a data component to populate and a namespace for importing children into the list component. Now for each item in the data component, a child element is imported and appended to the list component.

Here are the list and list-item elements.

<html>

<head>

<template is=”chtml-bundle”>

<ul chtml-ns=”html/list “></ul>

<li chtml-ns=”html/listitem“></li>

</template>

</head>

<body>

<!—import the list here -->

<chtml-import id=”list-component” chtml-ns=”html/list”></chtml-import>

</body>

</html>

The import element above should be automatically resolved. So the body should have a <ul> element by now. Now, we’ll populate this element with data. An import will be made for each data item.

Chtml.from(‘#list-component’).populate([‘item-1’, ‘item-2’], ‘html/listitem’);

This should produce the following:

<ul id=”list-component” chtml-ns=”html/list”>

<li chtml-ns=”html/listitem”></li>

<li chtml-ns=”html/listitem”></li>

</ul>

An object would as well produce the same result:

Chtml.from(‘#list-component’).populate({key1:‘item-1’, key2:‘item-2’}, ‘html/listitem’);

To manually set data on list items as they are loaded, provide a third argument as a callback function. This function will receive the just-imported list item and the data item.

Chtml.from(‘#list-component’).populate([‘item-1’, ‘item-2’], ‘html/listitem’, (listItem, dataItem, itemKey) => {

listItem.innerHtml = dataItem;

});

Now we should have something like the following:

<ul id=”list-component” chtml-ns=”html/list”>

<li chtml-ns=”html/listitem”>item-1</li>

<li chtml-ns=”html/listitem”>item-2</li>

</ul>

To achieve this declaratively, we could simply declare the bindings on the list-item elements themselves or on the import element that imports them.

<template is=”chtml-bundle”>

<li chtml-ns=”html/listitem“ chtml-directives=”el.html:$”></li>

</template>

Now, as a list item lands in the list component, its directives are automatically executed. It would like the implementation below:

Chtml.from(‘#list-component’).populate([‘item-1’, ‘item-2’], ‘html/listitem’, (listItem, dataItem) => {

Chtml.from(listItem).bind(dataItem);

});

### Using an Extended Namespace

The list above can be all declaratively-rendered! CHTML supports using a two-part namespace to import a list component. The first part of the namespace serves as the actual import namespace, while the second is stripped off for subsequently importing children. Two forward slashes separate these namespaces.

<body>

<!—import the list here -->

<chtml-import id=”list-component” chtml-ns=”html/list//html/listitem”></chtml-import>

</body>

Now we can simply set the data to the list component and have things automatically populated; every component loaded with a two-part namespace is automatically handled this way.

var list = Chtml.from(‘#list-component’).proxy();

list.$bind([‘item-1’, ‘item-2’]);

// Or put transparently…

//list.$ = [‘item-1’, ‘item-2’];

### Updating a List

List population is implement in CHTML as a binding contract between the DOM and data components. So more list items are imported as needed to match the number of data items received.

Below, we render two extra items to our list. Since two list-item elements already exists in the list component, they will simply be re-rendered. Only two new imports will now be made.

list.$ = [‘item-1’, ‘item-2’, ‘item-3’, ‘item-4’];

But what happens if we reduced the list data to a single item after having rendered many? The number of list-item elements will also be reduced! And setting an empty array will as well empty the list component. Remember, this is a binding contract!

list.$ = [‘item-1’,];

Now, our component should be:

<ul id=”list-component” chtml-ns=”html/list”>

<li chtml-ns=”html/listitem”>item-1</li>

</ul>

Really, instead of replacing the entire list and making the component render from the beginning, we can selectively add items. This time, the data component will need to be an Observable from the start.

var listData = new Observable([‘item-1’, ‘item-2’,]);

list.$ = listData;

listData[2] = ‘item-3’;

// We can directly call the observed array’s prototype methods…

// Add a fourth item

listData.push(‘item-4’);

// Empty the array

listData.splice(0);

// Add a new first item

listData.push(‘new item-1’);

And list rendering is index-aware! Items can be added to the data component on any key at any time, yet imported list-item elements will be correctly placed on the right index.

listData[10] = ‘item-10’;

listData[8] = ‘item-8’;

listData.unshift(‘newest item-1’);

The last item “newest item-1“ should still come first, and “item-8” should still be placed before “item-10”:

<ul id=”list-component” chtml-ns=”html/list”>

<li chtml-ns=”html/listitem”>newest item-1</li>

<li chtml-ns=”html/listitem”>new item-1</li>

<li chtml-ns=”html/listitem”>item-8</li>

<li chtml-ns=”html/listitem”>item-10</li>

</ul>

### Implementing Item Namespaces

When populating a list, items are retrieved, as seen, using the sub-namespace given. But individual items are retrieved with a namespace that is uniquely derived for the item. CHTML appends the item’s key to the base sub-namespace to import an item. So in the population we made above, the first item was imported using the namespace *html/listitem/0,* and the second item was imported using the namespace *html/listitem/1,* and so on*.*

The same pattern holds true for object-type list data. So for an object like {key1:‘item-1’, key2:‘item-2’}, the first item will be imported as *html/listitem/key1,* and the second, *html/listitem/key2.*

These derived namespaces can optionally be implemented where items really need to be unique. The base namespace, *html/listitem* in our case, remains the source of every item imported.

Directives work independently of each other and are bound to the specific context objects they might reference. So for directives that reference component nodes, only those elements of the DOM are touched as variables change.

This fits right with dynamic application data. Applications consuming a component can add data to the component which automatically becomes available to directives. As this data changes in the application, the directives using them are reapplied. We’ll learn more shortly.