<C2HTML />

## A component System that’s as Simple and Universal as HTML Itself!

CHTML is a componentization framework for HTML. It is aptly named as it shifts the entire idea of components from JavaSheet to plain HTML. And this is an important change!

## We miss how plain old HTML made us feel!

# But why?

HTML, CSS, and JS are until the end of the internet

Building components in JS is too far from declarative.

If semantics is CHTML is based on, then it’s the right thing to ask for. Everyone is needs it.

Things go out of fashion… and the only thing the web is ever surviving with is its standards.

# And Here We Go…

## Web Standards are Your Best Bet!

We believe that attempts to \*\*replace\*\* standard technologies or methodologies are a false sense of civilization; everything is short-lived, and is eventually phased-out! So we did things diferently with CHTML!

Consider just a glimpse of how CHTML leverages web standards, technologies and methodologies:

\* CHTML's component/node model -> HTML's standard content models as defined in the specs; HTML semantic markup and the Web Accessibility technology. (CHTML supports these out of the box along with an additional role-based markup pattern that makes custom models possible.)

\* CHTML's JSEN Params -> The JavaScript syntax. (Since the objective with CHTML directives is to consume JavaScript-based API (data/state and methods), no other syntax does it better than the very language the of the API - standardized over the years.)

\* CHTML's JSEN Params -> CSS. (Params cascading -> CSS cascading; inline params -> inline CSS; sheet-based params -> internal stylesheets.)

\* CHTML's multiple bundle cascading -> CSS stylesheets.

\* CHTML's node recompositon -> Slottables in the Web Components technology.

# The Component Framework

The Component Accessibility Framework provides a unified way of exposing UI components and their nodes from piles of markup details. This helps both humans and applications think of, and work with, the UI in terms of components and nodes, not just of components. As you’ll see, this remove all guess work and opens up UI components for universal consumption regardless of their implementation details.

## The Component Model

A component is simply an element that is designed or designated for a specific role or functionality. Components may be designed to work with other key elements called nodes. These work as subroles within their owning component or scope. This scope-node relationship is simply the component model.

Component models are formed at markup level, either explicitly by using a role-based markup pattern or implicitly as based on standard HTML and ARIA semantic models.

### Explicit Models

Explicit component models are formed using the *chtml-role* attribute. A rolename represents a functional scope

<div chtml-role=”article”></div>

Certain elements can be associated with this component as nodes using the *chtml-role* attribute. This time, the rolename (or subrole) is prefixed with the scope name (or owning role).

<div chtml-role=”article”>

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

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

</div>

And nodes could be anywhere within the component’s subtree.

<div chtml-role=”article”>

<div>

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

<div>

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

</div>

</div>

</div>

Now we have a clear component model that we can always bank on regardless of a component’s implementation details.

article

|--- author

|--- content

#### Nested Components and Scope Boundaries

Components can be nested. So a node may constitute a component of its own, and establish its own scope and have its own nodes.

Below, the *article-author* node is also a component.

<div chtml-role=”article”>

<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-content”></div>

</div>

</div>

</div>

This produces the following component model.

article

|--- author (user)

| |--- avatar

| |--- name

|--- content

Where two identical scopes (components) are nested, nodes are associated with the scope that’s closest to them up the hierarchy.

<div chtml-role=”article anotherrole”>

<div>

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

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

<div chtml-role=”article-brief article”>

<div chtml-role=”article-date”></div>

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

<div chtml-role=”anotherrole-othernode”></div>

</div>

</div>

</div>

The nested articles produce the following component model.

Article (anotherrole)

|--- author

|--- content

|--- othernode

|--- brief (article)

|--- date

|--- description

#### Special Nodes

The element on which a functional scope is established is called the **root node**. While a component references its nodes by their individual role, it reserves an invisible reference to this root node with the special “el” role. The *el* role is thus reserved and cannot be used as a role name.

Nested components have a parent-child relationship. Now, for every child component, there exists an invisible reference to an inherent **parent node** on the special underscore character “\_”. The underscore character is thus reserved and cannot be used as a role name.

#### Related Nodes

Sometimes a component may need to reference elements outside of its root as related nodes. References to related nodes are declared together in the *chtml-related* attribute. This attribute follows a CSS-like convention of key/value pairs, and pairs are separated by a semicolon.

<div chtml-role=”collapsible” chtml-related=”nodeName1:reference1; nodeName2:reference2”></div>

**References Are Node-To-Node Paths to Foreign Nodes.** A related node is actually a node contained in another component outside of the component’s scope. A reference is a node-to-node path to this foreign node, written in dot (.) or bracket [] notation.

Below, an app has multiple components. Now a collapsible component is referencing a button node in *app-bar* as its *control* node.

<body chtml-role=“app”>

<header chtml-role=”app-bar header”>

<div chtml-role=”header-button”></div>

</header>

<main chtml-role=”app-content collapsible” chtml-related=”control:\_.bar.button”></main>

</body>

Our path begins with a back reference to the parent component – app. From here, we could easily traverse down the component tree.

Since related nodes are usually out of scope, paths may always begin with a back reference.

In another scenario, we are moving two levels up the hierarchy to reference our button. We also played with the bracket notation.

<body chtml-role=“app”>

<header chtml-role=”app-bar header”>

<div chtml-role=”header-button”></div>

</header>

<main chtml-role=”app-main”>

<div>

<div chtml-role=”main-content collapsible” chtml-related=”control:\_.\_.bar[‘button’]”></div>

</div>

</main>

</body>

Notice the quoted key in our brackets! Unquoted keys are actually references of their own. Later, in the section for parameters and cascading, we will see other properties of a component that we can reference this way in our paths. These properties are usually not seen in markup (just as the parent reference isn’t a role name in markup.) We’ll also see how paths can be dynamic by using logical expressions.

**References Can Be Defined in a Params Sheet.** Inline references (related nodes listed in the *chtml-* *related* attribute), like inline CSS, can quickly begin to look congested. CHTML lets us define them in a *Data Block* type of script element where these can be listed more comfortably, just as with a style sheet. This script element is called JSEN Params Sheet (jsen-p for short) and is placed at the root of the component.

<div chtml-role=”main-content collapsible”>

<script type=”text/jsen-p”>

@related {

control: \_.\_.bar[‘button’];

}

</script>

</div>

Where references are declared both inline and in JSEN-P, the combined parameters are used in a cascaded manner, usually with inline parameters taking precedence on duplicate parameters. The cascaded nature of parameters is covered in the section for cascading.

### Implicit Models

Certain elements in HTML have been naturally designed to work together in a predefined manner. Usually, one element establishes the functional role or scope, while the others serve as rubroles or nodes. These natural relationships or implicit models are automatically recognized in CHTML and need not be explicitly designed.

Not commonly known, every element has a design note or specification that states the element’s **category** and dictates what and what can serve as its content – better known as the element’s **content model**. For example, the <html> element must only have the <head> and <body> tags as its direct children. Okay, that’s common knowledge, but really, that’s the specs dictating.

Content models are automatic component models in CHTML; here an element’s category, if defined, forms the scope, and its permissible elements, serve as nodes. Below are examples of these implicit models. The same rule holds for other models as defined in their standards specs.

The <html> element’s content model permits two elements that must also be direct children.

<html>

<head></head>

<body></body>

</html>

This gives us the component model:

html

|--- head

|--- body

The <head> element’s content model permits only elements that have been categorized a *Metadata* elements.

Below, the following *Metadata* elements can only be used once in the <head> element.

<head>

<title></title>

<base />

</head>

This gives us the following component model

head

|--- title

|--- base

Now certain elements are permitted to appear any number of times within their permissible scope. The <head> element, again, permits multiple <meta> elements.

<head>

<title></title>

<meta name=”” content=”” />

<meta name=”” content=”” />

<meta name=”” content=”” />

<base />

</head>

This gives us the following component model. (Notice that the *meta* node becomes a list.)

head

|--- title

|--- meta (3)

|--- base

#### Nested Scopes Create a Boundary

Where two identical scopes are nested, nodes are associated with the scope that’s closest to them up the hierarchy. The <body> element and the <blockquote> element are a good example of nested identical scopes.

The <body> element is categorized as *Sectioning Root* which permits elements categorized as *Flow Content*. The <blockquote> element is one of those *Flow Content* elements. (Most elements are categorized as *Flow Content* – h1, div, p, etc.) Now the <blockquote> element is also a *Sectioning Root* that could have its own *Flow Content*. So below, every element is a node to the body’s *Sectioning Root* scope except those within blockquote’s *Sectioning Root* scope.

<body>

<div>

<h1></h1>

<div>

<p></p>

</div>

<blockquote>

<h1></h1>

<div></div>

</blockquote>

<div></div>

</div>

</body>

This gives us the following component model.

body

|--- div (3)

| [0]

| : (this model is omitted for simplicity)

| [1]

| |--- p (1)

| [2]

|--- h1 (1)

|--- blockquote (1)

| [0]

| |--- h1 (1)

| |--- div (1)

|--- p (1)

Another form of the *Sectioning Root* category is *Sectioning Content*. Elements in this category are <article>, <aside>, <nav>, and <section>. Notice how the <header> element, being a *Flow* Content element, is semantically associated below. Also note that there cannot be more than one within its permissible scope.

<body>

<div>

<header></header>

<article>

<header></header>

<div></div>

</article>

</div>

</body>

This gives us the following component model.

body

|--- div (1)

| [0]

| : (this model is omitted for simplicity)

|--- header

|--- article (1)

[0]

|--- header

|--- div (1)

#### ARIA Roles Are Automatically Recognized

The implicit and explicit roles defined in the ARIA specification are automatically recognized in CHTML. Elements that have an explicit or implicit ARIA role can be accessed by both their rolename and their tagname.

<body>

<div>

<div role=”header”></div>

<article>

<div role=”header”></div>

<div></div>

</article>

</div>

</body>

This gives us the following component model.

body

|--- div (2)

| [0]

| : (this model is omitted for simplicity)

| [1]

|--- header

|--- article (1)

[0]

|--- div (2)

|--- header

The <section> element has the implicit ARIA role *region*. So both the names *section* and *region* will point to the <section> element.

<body>

<section>

</section>

<section>

</section>

</body>

This gives us the following component model.

body

|--- section/region (2)

The <ul> element has the implicit ARIA role *list*. The <li> element has the implicit ARIA role *listitem*. Here’s how it looks.

<body>

<ul>

<li></li>

<li></li>

</ul>

</body>

This gives us the following component model. Also notice that the <li> element do not reflect in the <body> as their permissible scope ends with the <ul>.

body

|--- ul/list (1)

[0]

|--- li/listitem (2)

## The Component API

The CHTML API is the official API for CHTML. It provides a complete suite of functions for working with the various aspects of CHTML. This API can be implemented in any language and platform. But the specification and examples in this guide are of JavaScript implementation.

### The Component Instance

The Chtml instance is used for translating a component’s Conceptual Model to an Object Model for programmatic use. In its basic form, a Chtml instance lets us access a CHTML component model as properties and objects.

import Chtml from ‘@onephrase/chtml’;

If Chtml has been loaded via a script tag, it will be available in the global “OnePhrase” object.

const Chtml = OnePhrase.Chtml;

A component instance is created via the Chtml constructor. The constructor takes a DOM element as it’s the component’s root element, and optionally accepts a params object that provides additional parameters for the instance.

Syntax:

const component = new Chtml(el[, params]);

This creates an object that maps its properties to the underlying component’s nodes.

Examples:

// Lets create a component on the DOM documentElement itself

// and access its nodes.

const doc = new Chtml(document.documentElement);

let head = doc.get(‘head’);

let body = doc.get(‘body’);

// But we can better access nodes as properties using a proxied version of the component instance.

// Details shortly.

const \_doc = doc.proxy();

let head = \_doc.head;

let body = \_doc.body;

// Here’s how we could write to the body node

body.innerText = ‘Hello beautiful world!’;

Notes:

1. Proxied instances obtained from the proxy() method give us the benefit of accessing nodes as properties while actually forwarding each access to the instance’s get() method. In this mode however, instance methods would need to be prefixed with the $ character to prevent the proxy from forwarding the method name as node name.

let body = \_doc.body;

let body = \_doc.$get(‘body’);

You can always tell whether or not an instance was proxied. Chtml proxies are created using the Js utilites from @onephrase/commons which allows us to work with the proxied object in other ways. If you have Commons installed...

import {Js} from ‘@onephrase/commons’;

// Test if an instance is a proxy

if (Js.isProxy(\_doc)) {

// true

}

// Get the original instance object

let doc = Js.getProxyTarget(\_doc);

1. By default, an instance’s underlying root element is the native DOM element. But we could decide to have these DOM elements wrapped in a DOM abstraction object like jQuery. This is done via params.nodeCallback parameter. The params.nodeCallback should be a function that receives these DOM elements, as they are created, and returns their abstraction.

const doc = new Chtml(document, {nodeCallback: el => $(el));

To do this globally for all instances created from the Chtml class, the global Chtml.params object is used.

Chtml.params.nodeCallback = el => $(el);

With jQuery objects now returned, we can now work with DOM elements with a more interesting syntax.

let \_doc = doc.proxy();

\_doc.body.html(‘Hello from the other side!’);

And we can go on to extend jQuery with some custom methods.

Most examples in this documentation will use the jQuery DOM manipulation API. Note that this is just for demonstration purposes as CHTML does not ship with jQuery nor does it even require it.

1. Nodes are lazy-loaded. So the DOM is accessed once for each node. The node is stored for subsequent access.

#### The Chtml.from() Method

In addition to creating instances from the Chtml constructor, the Chtml.from() static method may be used. This method accepts the same augments as with the constructor, but also agrees to accept the root element as a CSS selector or even a HTML markup.

Syntax:

const component = Chtml.from(input[, params]);

Examples:

// Create an instance from a DOM object as usual

const doc = Chtml.from(document.documentElement);

// Create an instance from a selector

const body = Chtml.from(‘body’);

const component = Chtml.from(‘#some-element’);

// Create an instance from markup;

// the from() method will first automatically resolve this to an element.

let markup = ‘<div chtml-role=”comp”><span chtml-role=”node”></span></div>’;

const component = Chtml.from(markup);

### The Component Tree and Drilldown

With a mental model of a component’s nodes and its nested sub components, we can easily traverse the component tree. Here is how we could access an article’s author component from the article component itself.

<div chtml-role=”article” id=”article”>

<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>

const article = Chtml.from(‘#article’).proxy();

let articleAuthor = Chtml.from(article.author).proxy();

let authorName = articleAuthor.name;

Well this could be much simpler. Chtml implements a params.drilldown property that provides a way to seamlessly access deep nodes in a component tree. With this feature on, all nodes are returned as a component instance instead of the default node type.

const article = Chtml.from(‘#article’, {drilldown: true}).proxy();

let authorName = article.author.name;

The authorName variable is now rather holding a Chtml instance, not a DOM element. This makes it seamless to continue the drilldown. Using a *.el* at any point drops the chaining and returns the node’s underlying element.

const article = Chtml.from(‘#article’, {drilldown: true}).proxy();

let author = article.author.el;

let authorNameElement = articleAuthor.name.el;

// With an object like jQuery as endpoint…

Chtml.from(‘#article’, {drilldown: true, nodeCallback: el => $(el)}).proxy();

article.author.el.html(‘John Doe’);

Notes:

1. Drilldown always returns a Chtml instance for every node, whether or not the node has been defined as a component in HTML. This makes everything more predictable when accessing a component tree. The special *el* key is used to access the node’s underlying element.
2. A component’s params object is transferred from component to sub components as they are created. So we don’t have to worry about passing params to deep nodes.
3. Drilldown wisely employs the Chtml.from() method to resolve a node.

### Reactivity and Bindings

Applications can be built with CHTML as View layer. This is even more seamless with the concept of reactivity and bindings.

#### Reactivity

Reactivity is one beautiful thing about the component object. The Chtml class is built off the Observable class from @onephrase/observable. This brings us all the reactivity we’ll need with components and everything else that’s possible with Observables.

Being an Observable instance, we can observe when nodes land for the first time on the component object and when they exit the object – either by an explicit delete operation or via a direct removal from the DOM.

// Remember that nodes are lazy-loaded.

// So the *author* node won’t be loaded on the component until the first time it is accessed.

// First, let’s construct an observer for this event…

article.$observe(‘author’, author => {

console.log(author.el);

});

// Let’s access the node for the first time.

// Our observer should be called.

article.author.el.html(‘John Doe’);

// Let’s explicitly delete the node from the object.

// Our observer should be called and its author parameter should be undefined.

delete article.author;

// Re-accessing the node should recreate the node from the DOM and call our observer.

article.author.el.html(‘New Name’);

// Let’s directly remove the node from the DOM.

// Our observer should be called and its author parameter should be undefined.

article.author.el.remove();

// Re-accessing the node should now return undefined.

article.author.el.html(‘New Name’); // Reference error

#### Binding

It is possible to introduce arbitrary properties into the component object. A component’s properties must not all be nodes; application data can be set on a component on any property. But we will be careful with our choice of property name to avoid accidentally unsetting a node.

So we could actually do the following:

// Trying to avoid conflicting with a node…

article.authorNameVal = ‘John Doe’;

article.author.el.html(article.authorNameVal);

But we will be safer to use a property name that won’t interfere with node names. The “$” character should be our best choice. (The “$” character will now be reserved from being used as a node name.)

// Safe from conflicts…

article.$ = ‘John Doe’;

article.author.el.html(article.$);

Suppose we had more than one data value to set on the component. This would normally mean setting multiple arbitrary properties on the component; but that would be polluting the component’s property namespace. A better approach would be to make the “$” property an object.

// Set multiple values out of the component’s property namespace…

article.$ = {};

article.$.author = ‘John Doe’;

article.$.description = ‘Article description’;

article.author.el.html(article.$.author);

article.description.el.html(article.$.description);

We can leverage reactivity and make the above operations dynamic. So we observe the “$” data property.

// Render data dynamically

article.$observe(‘$’, $ => {

article.author.el.html($.author);

article.description.el.html($.description);

});

// We could as well observe the data values as path

article.$observe(‘$.author’, author => article.author.el.html(author));

article.$observe(‘$.description’, description => article.author.el.html(description));

// Bring the data anytime

article.$ = {author:’John Doe’, description:’Article description’,};

// Update the data anytime

article.$ = {author:’Mark Spencer’, description:’Updated article description’,};

At this point, we have bound operations to the “$” data property. But we still have more to explore with bindings.

##### The Chtml.bind() Instance Method

This method is just another way to set the component’s data property. It accepts the data component to bind and sets it to the “$” property automatically.

article.bind({author:’John Doe’, description:’Article description’,});

To change the default data property from the “$” character to something else, the params.dataqKey is used.

// Set this globally

Chtml.params.dataKey = “data”;

// To change this per instance

var article = new Chtml(el, {dataKey:’data’});

Calls to *article.bind()* will now set the given data object to the component’s ‘data’ property. So it serves to hide implementation details and constitutes a more standard way to add data to a component.

##### Binding Observables

If you noticed, binding plain data objects required that we replace the entire plain object to update. This is because updating properties of the plain object “$” would not bubble up to notify the *article* object of a change. But this is easy to fix by making the data object an Observable.

// Bring in the Observable class

import Observable from ‘@onephrase/observable’;

// Set the observable base for data.

// Initial properties for the Observable instance are optional

article.$ = new Observable({author:’John Doe’});

// Update properties.

// Operations bound to ‘$.author’ will be re-executed.

article.$.author = ‘Mark Spencer’;

// Set new properties.

// Operations bound to ‘$.description’ will be executed for the first time.

article.$.description = ‘Article description’;

// Nest Observables as needed to make deep properties reactive.

article.$.author = new Observable({fname:‘John’, lname:’Doe’});

// Update… and anything bound to “$”, “$.author”, or “$.author.fname” will be called.

article.$.author.fname = ‘Mark’;

As a general good practice, application state, and all functionality over state, are not built on CHTML components, but off the component as standalone observable components that plug-in to the CHTML component.

##### Controlling State

In the examples above, we have directly mutated the state of our data object. But a proper way to do this is to encapsulate the mutation logic within the Observable object itself and expose mutations as methods. This way, we will be sure states are mutated in a standard way by the different parts of the application.

Below, we create a dedicated *Author* data component, with state control.

class Author extends Observable {

/\*\*

\* Here we initialize the instance with author names

\*/

constructor() {

super();

this.authors = [{fname:’John’, lname:’Doe’}, {fname:’Mark’, lname:’Spencer’}];

}

/\*\*

\* This method publishes a new author name on each call

\*/

next() {

var nextAuthor = this.authors.shift();

if (nextAuthor) {

// The set method must be used to set state

this.set(‘fname’, nextAuthor.fname);

this.set(‘lname’, nextAuthor.lname);

// Or in one set() operation

//this.set(nextAuthor);

}

}

}

Next, the binding that updates the DOM. And the binding that updates the data component.

// We let state-change in the data component trigger update on the DOM.

// We’ll be showing the full name

article.$observe([‘$.author.fname’, ‘$.author.lname’], authorNameArray => article.author.el.html(authorNameArray.join(‘ ‘)));

// We let event in the DOM trigger update on the data component.

// We’ll be seeing a new author on double-clicking the article element

article.el.on(‘dblclick’, () => article.$.author.$next());

Now, we plug in the Author data component and make our first click.

// Add an *Author* instance to the *article* component

article.$.author = new Author();

At this point, a control-flow pattern has emerged; let’s call this *Actions and Reactions*! Here, state components are controlled via actions (method calls), DOM components are controlled via reactions (state bindings).

In the Actions / Reactions control-flow pattern, one party (the State component) is designed to be totally agnostic of the other party (DOM components or other observers). So whether or not DOM Components have been bound, the State component remains functional and independent. In other words, it does not need to know who is triggering its methods (actions) and who is listening to states (observers, bindings). Meanwhile, any part of the application can *act on,* and *react to,* a State component.

##### Synchronizing Actions with Reactions

Although State components by design need not know about observers or bindings, they might still sometimes need a feedback from the observers bound to a state. This feedback comes very useful when the State component’s next action needs to synchronize with these observers. Consider the case below.

In the next() method of our Author component above, we simply published authors in succession on the *fname* and *lname* states. But we could make things more interesting by announcing the beginning and end of this publishing event. We will capture the feedback from observers of this announcement to determine how we go about publishing the new author details. Let’s call this project Author Display.

/\*\*

\* This method publishes a new author name on each call.

\*

\* It announces the start and end of each author-change with the “headsup” state.

\*/

next() {

var nextAuthor = this.authors.shift();

if (nextAuthor) {

// Announce the intention to publish new author details

var announcementFeedback = this.set(‘headsup’, ‘publishing’);

// Did any observer ask to prevent this action?

if (announcementFeedback.defaultPrevented) {

return;

}

// Did any observer return a promise?

// That would mean asking this publishing event to hold for a time.

var returnedPromise = announcementFeedback.promises;

if (returnedPromise) {

returnedPromise.then(() => {

this.set(‘fname’, nextAuthor.fname);

this.set(‘lname’, nextAuthor.lname);

this.set(‘headsup’, ‘published’);

});

} else {

this.set(‘fname’, nextAuthor.fname);

this.set(‘lname’, nextAuthor.lname);

this.set(‘headsup’, ‘published’);

}

}

}

As seen, the next() method has chosen to honor observer feedbacks. Now we can create bindings that really do return a Promise. In the binding below, we implement a fade-out /fade-in animation as a new author gets published. We do this by returning a Promise on hearing the “publishing” announcement, while playing the fade-out animation. The Promise is resolved at the end of the animation and the next() method notices this and publishes the details. Finally, the fade-in animation plays on hearing that the details have been “published”.

// Bind the fading-out and fading-in to the “headsup” state

article.$observe(’$.headsup’, state => {

if (state === ‘publishing’) {

return new Promise((resolve, reject) => {

var animation = article.author.el.animate([{opacity:1, opacity:0}], {duration:600});

animation.onfinish = resolve;

});

} else if (state === ‘published’) {

article.author.el.animate([{opacity:0, opacity:1}], {duration:600});

}

});

// As before, the binding that updates the DOM

article.$observe([‘$.author.fname’, ‘$.author.lname’], authorNameArray => article.author.el.html(authorNameArray.join(‘ ‘)));

// As before, the binding that updates state

article.el.on(‘dblclick’, () => article.$.author.$next());

// Plug in the Author, and let double-clicking begin

article.$.author = new Author();

### Optimizing DOM updates

As seen, CHTML does not intercept operations that update the DOM. But it makes room for implementing DOM updates that are performant. A common technique is to batch DOM-mutation operations and execute them differently from DOM-read operations while keeping everything in sync with the browser’s “animation frame” (the window.requestAnimationFrame() function). That way, we would be avoid unnecessary DOM thrashing.

This is an optional optimization strategy and is covered outside the scope of CHTML. But below would be a contrived implementation of the el.html() method we have been using.

html(content) {

// We wrap the actual operation

// in a callback from an imaginary batch() function.

batch(() => {

this.innerHtml = content;

// Or if this were a custom jQuery method, we would say

//this.get(0).innerHtml = content;

});

}

# 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.

# 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.

# Misc

## The Composition API

Chtml offers certain methods for working with modules and bundles. We will demonstrate these methods against the simple template below.

<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“>

<div>

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

<div>

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

</div>

</div>

</div>

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

</template>

### The Chtml.import() Static Method

This method simply retrieves a module by namespace from across loaded bundles.

var articleEl = Chtml.import(‘html/content/article’);

The result is the HTML element when found, or undefined, when not found. At this point, all applicable recomposition would have been made.

Below, importing the *html/content/article/readonly* module will apply inheritance-based composition from its *html/content/article* super namespace. And the element returned remains the <chtml-import> element.

var readonlyArticleEl = Chtml.import(‘html/content/article/readonly’);

### The Chtml.from() Static Method

This method also works like the Chtml.import() method, but this time, used to retrieve elements from the within the document body, not bundles. Below, we retrieve a namespaced element from within the document body.

<body>

<chtml-import ondemand chtml-ns=”html/content/article/editable“></chtml-import>

</body>

var editableArticleEl = Chtml.from(‘html/content/article/editable’);

The result is an HTML element that’s fully resolved to the *html/content/article* module. At this point, the temporary <chtml-import> element will be automatically replaced by the imported article module.

### The Chtml.ready() Static Method

This method is used to keep code execution in sync with the document’s “ready” state. It accepts a callback to be run when the DOM announces readiness – an event that indicates that the document tree has been initialized and safe to access.

Chtml.ready(() => {

// Put code here

});

This method also extends to wait for CHTML bundles to load. This is useful when running code that relies on external CHTML bundles. As it is, bundles have to be loaded in order to access their modules. In fact, a warning is issued on attempting to import modules while bundles are still loading.

Chtml.ready(() => {

var remoteModule = Chtml.import(‘html/content/article/readonly’);

});

To prevent waiting for bundles, pass false as the second argument to this method.

Chtml.ready(callback, /\*waitForBundles\*/false);

## The Bundler Utility

This is a little server-side utility that automatically bundles markup-based files into a single file that can be linked-to as an external CHTML bundle.

**Basic Usage**

import Bundler from ‘@onephrase/chtml/src/Bundler.js’;

const bundler = new Bundler(entry);

Above, *entry* is the base directory for finding files. Files are scanned recursively from here into subdirectories and are bundled each with a namespace that reflects their location.

To get the bundled contents as string, call the output() method.

var bundleContent = bundler.output();

To save the contents to a file, provide the destination file to the output() method; this can be an absolute path or a relative path – relative to the bundle *entry* given initially.

bundler.output(outputFile);

Let’s demonstrate this with the sample file structure below.

/root

|--- project-files

| |--- components

| | |--- component1.html

| |--- page1.html

| |--- page2.html

|--- public-folder

Now we bundle the files in the “project-files” directory into “public-folder”.

const bundler = new Bundler(‘/root/project-files/’);

bundler.output(‘/root/public-folder/bundle.html’);

This file can now be loaded as a chtml-bundle.

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

On checking the bundle, you will notice that the namespace of each module is prefixed with the extension name of their original file. Here’s how that could look:

<div chtml-ns=”html/components/component1”></div>

<div chtml-ns=”html/page1”></div>

<div chtml-ns=”html/page2”></div>

### Bundling From Multiple Entries

To create multiple bundles from multiple entries, use the static Bundler.multiple() method. This method accept a list of entries as an object and returns an object with each bundle name mapped to their respective bundle content.

var bundles = Bundler.multiple({

images: ‘path/to/images/’,

template: ‘path/to/templates/’,

});

console.log(bundles.images);

To save all bundles to a common directory, provide an output file path as second argument to Bundler.multiple().

Bundler.multiple({

images: ‘path/to/images/’,

templates: ‘path/to/templates/’,

}, ‘/path/to/public-folder/[name].bundle.html’);

Notice the *[name]* placeholder in the destination filename. For each of the bundles, this placeholder is replaced with the unique bundle name. So, while all bundles are saved to “/path/to/public-folder/”, each bundle is saved with a unique filename. (If a placeholder is not in the given file path, something bad happens – each bundle is saved to the same file and previous bundles are overwritten!)

We can now load each bundle.

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

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

### Bundling Assets

While HTML modules are created by reading the file’s contents, assets, like images, are handled differently. They are still allowed to remain on the server, but this time, copied to the output directory where the regular bundles are located. An element that points to this new location is automatically generated in the bundle. This is illustrated below.

This is the normal file structure. It now contains an image.

/root

|--- project-files

| |--- components

| | |--- component1.html

| |--- images

| | |--- image1.png

| |--- page1.html

| |--- page2.html

|--- public-folder

Let’s bundle the files in the “project-files” directory.

const bundler = new Bundler(‘/root/project-files/’);

bundler.output(‘/root/public-folder/bundle.html’);

Now the image at /root/project-files/images/image1.png will now be copied to /root/public-folder/images/image1.png and an image element pointing to this new location is added to the bundle.

This is the new file structure:

/root

|--- project-files

| |--- components

| | |--- component1.html

| |--- images

| | |--- image1.png

| |--- page1.html

| |--- page2.html

|--- public-folder

|--- images

| |--- image1.png

|--- bundle.html

And our bundle.html should look like this:

<div chtml-ns=”html/components/component1”></div>

<img chtml-ns=”png/images/image1” src=”images/image1.png” />

<div chtml-ns=”html/page1”></div>

<div chtml-ns=”html/page2”></div>

In another way, it is possible to bundle small images (or other media) in data-URL format. The Bundler just needs to know the at what file size to use the data-URL format. Set the static Bundler.maxDataURLsize to a size measured in bytes.

Bundler.maxDataURLsize = 1024;

Media files lower than this size will now be bundled in data-URL format. This can greatly reduce the number of HTTP requests the browser has to make.