# 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;

});

}