<C2HTML />

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

# But why?

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

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

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…

## The Roles Model

* Role-Based: Expose the semantics of your markup using a straight-forward roles model.
* Declarative Binding: Describe behaviors as actions and reactions centrally using simple phrases.
* Architecture: Leverage namespacing and hierarchical layout plan for more organization.
* Reusability: Define once and reuse in whole or in part using Imports, Overrides and Inheritance.

# The Component Accessibility Framework

The Component Accessibility Framework provides a unified way of exposing UI components and their implementation 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. It is also a new beginning for universally accessible component technology.

This framework is made up of the Accessible Component Model and the Component Accessibility API.

## The Accessible Component Model

A component is simply an element that is designed or designated for a specific role or functionality. This role or functionality may involve or even require certain other key elements called nodes. Nodes work as subroles within their owning context or superrole. This gives us a component-node relationship, or simply, 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 roles.

#### Explicit Models

Elements may be designated for a role using the *data-role* attribute. Single-word role names represent a component.

<div data-role=”article”></div>

Elements may also be purpose-built for a role using a two-part tagname convention. These are called Custom Elements.

<my-article></my-article>

Built-in elements may also be customized using the *is* attribute. These are called Customized Built-in Elements.

<div is=”my-article”></div>

Custom Elements are governed by the Web Components Specifications.

In each case, the rolename of the elements is consistently “article”. Regarding the last two cases, the first part of a custom tagname or the *is* attribute is usually only a namespace and so does not contribute to the rolename.

All three conventions establish a functional context or component with a role name.

##### Nodes

Nodes may be added to an explicit model. This is also done explicitly using the data-role attribute. This time, the rolename is prefixed with the owning role or supperrole.

<div data-role=”article”>

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

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

</div>

The same convention holds for custom elements.

<my-article>

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

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

</my-article>

… and customized built-ins.

<div is=”my-article”>

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

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

</div>

Now, nodes could be anywhere within a component scope.

<div data-role=”article”>

<div>

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

<div>

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

</div>

</div>

</div>

In all four cases, we have the same component model that is clearly identifiable. This predictability over unpredictable markup details is the wisdom in the component model!

article

|--- author

|--- description

#### Implicit Models

Certain elements in HTML are designed to work together in a predefined manner. Usually, one element establishes the main role or context, while the others serve as rubroles or nodes. These semantic relationships or implicit models are automatically recognized in CHTML.

##### The Document Model

HTML’s document-level elements work together as an implicit document model.

<html>

<head></head>

<body></body>

</html>

The equivalent explicit model.

<html data-role=”document”>

<head data-role=”document-head”></head>

<body data-role=”document-body”></body>

</html>

###### The Document Metadata (Head) Model

###### The Document Content (Body) Model

The implicit document model extends to include any HTML5 landmark elements that may be associated with the document body. And as we play into HTML5’s semantic elements and relationships, we will be carrying along the ARIA alternatives to the models or parts of the models.

**The document main.** Only one main element is expected in an entire document, whether directly at the body-root level or nested within other elements.

<body>

<main></main>

</body>

The ARIA alternative to the main element.

<body data-role=”body”>

<div role=”main”></div>

</body>

The equivalent explicit model.

<body data-role=”body”>

<main data-role=”main”></main>

</body>

**Document ends.** The document header and footer are header and footer elements that are semantically associated with the body element as their closest [sectioning root](https://html.spec.whatwg.org/multipage/sections.html#sectioning-root) / [section content](https://www.w3.org/TR/html5/dom.html#sectioning-content-2).

##### Layout Models

##### Widget Models

This means that nodes can explicitly reference the body element as if its role were defined.

<html>

<head>

<title>…</title>

</head>

<body>

<div c-role=”body-main”></div>

</body>

</html>

Taking a step further, we can even use the <main> semantic element instead of the <div>. By virtue of its semantics, it is automatically recognized as a node of the body component.

<html>

<head>

<title>…</title>

</head>

<body>

<main></main>

</body>

</html>

<html>, <

The ARIA specification defines equivalent roles for declaring the same standard semantics explicitly on any element on a special *role* attribute. ARIA roles explicitly designate elements as components.

Whether implicitly or explicitly defined, the semantics of these components are governed by standard specifications.

## The Component Accessibility API

The Component Accessibility API is a translation of a component’s structural model into an Object Model for programmatic use.

# Structural Concepts

## The Custom Roles Model

CHMTL is based on semantic markup.

### A Component

A component in CHTML is simply an HTML element that is marked as a standalone block of functionality. This functionality is indicated on an element’s *role* attribute.

<div c-role=”article”></div>

#### Nodes

A component may implement special elements called nodes. Nodes are associated to a component on their *role* attribute. They become properties of the component.

<div c-role=”author”>

<div c-role=”article-author”></div>

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

</div>

Nodes could be anywhere within a component, but the component model can remain clearly identifiable. Nodes remain direct properties of the component.

<div c-role=”author”>

<div>

<div c-role=”article-author”></div>

<div>

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

</div>

</div>

</div>

#### The Root Node

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

### Nested Components

Components can be nested. So a node may constitute a component of its own, and start a new roles model.

<div c-role=”article”>

<div>

<div c-role=”article-author user”>

<div c-role=”user-avatar”></div>

<div c-role=”user-name”></div>

</div>

<div>

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

</div>

</div>

</div>

#### The Parent Node

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 in the *related* attribute. This attribute follows a CSS-like convention of name:value pairs.

<div c-role=”collapsible” c-related=”roleName1:reference1; roleName2:reference2”></div>

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 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 element.

<div c-role=“app”>

<header c-role=”app-bar header”>

<div c-role=”header-button”></div>

</header>

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

</div>

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.

<div c-role=“app”>

<header c-role=”app-bar header”>

<div c-role=”header-button”></div>

</header>

<main c-role=”app-main”>

<div>

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

</div>

</main>

</div>

Notice the quoted key in our brackets! Unquoted keys are actually references of their own. Later, 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.)

## The Semantic Roles Model

While a naming convention helps us identify elements as components and nodes, it may not be necessary for certain elements with special meaning in HTML. The semantics of these elements are automatically recognized in CHTML. So they can be used without explicitly (re)defining their role, although it wouldn’t hurt either. This applies to semantic native elements and all custom elements.

#### Semantic Elements

Semantic elements like the head, title, and body are absolute in meaning within a document. Their role-based tag name establishes a component context in CHTML. So the two documents below mean the same thing in CHTML.

(a)

<html>

<head>

<title>…</title>

</head>

<body></body>

</html>

(b)

<html c-role=”document”>

<head c-role=”document-head”>

<title c-role=”document-title”>…</title>

</head>

<body c-role=”document-body body”></body>

</html>

This means that nodes can explicitly reference the body element as if its role were defined.

<html>

<head>

<title>…</title>

</head>

<body>

<div c-role=”body-main”></div>

</body>

</html>

Taking a step further, we can even use the <main> semantic element instead of the <div>. By virtue of its semantics, it is automatically recognized as a node of the body component.

<html>

<head>

<title>…</title>

</head>

<body>

<main></main>

</body>

</html>

#### Custom elements

Custom elements are purpose-built elements. They are named on their tag for certain functionality. Their role-based tag name establishes a component context in CHTML. So the two elements below make the same component in CHTML.

(a)

<my-app></my-app>

(b)

<my-app c-role=”app”></my-app>

Notice that with their required two-part name, only the last part works as role in CHTML, as the first is usually only a namespace.

Now, nodes, if any, must be explicitly defined, this time, based on the context’s last name-part.

<my-app>

<div c-role=”app-bar”></div>

</my-app>

Nested custom elements can also work as nested components.

<my-app>

<my-header c-role=”app-bar”>

<div c-role=”header-button”></div>

</my-header>

</my-app>

An explicit role can, however, be used instead of the implicit.

<my-app>

<my-header c-role=”app-bar foo”>

<div c-role=”foo-button”></div>

</my-header>

</my-app>

## List Types

In HTML, lists elements have a fixed relationship with their direct children. Here, all root-level elements are associated with the list container as items. This semantics translates as-is to a list component in CHTML. Here, list containers are automatically recognized as list components, and every root-level element is automatically taken as node.

<ul>

<li></li>

<li></li>

</ul>

Now, all nodes play the same role, but each node is uniquely identified by its numeric position in the list. So while nodes in a regular component are alphabetically accessed, nodes in a list component are numerically accessed, as we will now demonstrate.

Below, an app has a menu component in app-bar. Now a popup component is referencing a menu item node as its button element.

<div c-role=“app”>

<header c-role=”app-bar header”>

<ul c-role=”header-menu”>

<li>View Profile Picture</li>

<li>Edit Info</li>

</ul>

</header>

<main c-role=”app-content”>

<div c-role=”popup” c-related=”button:\_.\_.bar.menu[0]”>

<img src=”/users/1/pic” alt=”Oxford’s Profile Picture” />

</div>

</main>

</div>

# Definition System

## Definition, Namespaces and Layout

A component may be assigned a namespace to place it in a virtual category. This results in better code organization.

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

<div c-ns=”content/article”></div>

Here we’ve categorized article as a type of content component. There could be other types in this category. And we can have subtypes – to as much as organization requires.

// Displays article

<div c-ns=”content/article/readonly”></div>

// Allows editing

<div c-ns=”content/article/editable”></div>

Interestingly, we can go from virtual namespacing to actual folder-based layout. The equivalent layout of the above namespaces would look like:

content

|-- article

|-- readonly.html

|-- editable.html

This meets the challenge as to where to place things or where components should live, especially when we have a handful of them to work with.

Now with a hieratical layout establishing a clear mental relationship between components, we can do without explicitly assigning an ns attribute.

In a straightforward process called bundling, it is easy to put all our components back to one file – a portable file that can be exported or otherwise imported by a web browser. This time, namespaces are assigned to each component from their actual layout path.

## Imports, Overrides and Inheritance

Reusability is the essence of componentization. The idea is to define once and use everywhere. CHTML offers different ways to reuse components.

### Imports

Imports are a way to place components anywhere in a HTML document. The idea is to use a temporary component to import the needed component. This temporary component is the import element.

<c-import c-ns=”content/article/readonly”></c-import>

Imports use an ns attribute to reference the source component. The import component itself gets replaced by the actual component; so at best, it is only a placeholder component.

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

<!— file: badge/user.html -->

<div c-role=”user”>

<div c-role=”user-avatar”></div>

<div c-role=”user-name”></div>

</div>

<!— file: content/article/readonly.html -->

<div c-role=”article”>

<div>

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

<div>

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

</div>

</div>

</div>

### Overrides

Imports can do more than just place a component. They can also override parts of the component they’re importing. Since we talk about components in terms of nodes, overriding is a node-to-node replacement concept.

Now as we import the user component into the article component, we can override the user’s name node with one that features special styling.

<!— file: content/article/readonly.html -->

<div c-role=”article”>

<div>

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

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

</c-import>

<div>

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

</div>

</div>

</div>

So imports can possibly have their own nodes; after all, they’re almost a component of their own! But they do not need to establish a context for these nodes, as their nodes are actually heading out to a different context. Thus it was not necessary to define an extra user role as in c-role=”article-author user”. Meanwhile, the c-role=”user-name” was appropriate for the replacement node in view of its target context.

Now a few notes apply:

* An import’s nodes are called replacement nodes. The nodes they replace are called slots.
* 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 a slot node in source component. Hence imports must not have a root-level element without a valid role.
* Attributes of the import element are reassigned to the source component’s root element. Values of the role attribute are merged.
* For nodes, attributes of slot nodes are reassigned to replacement nodes. Values of the role attribute are merged.

Something that looks like CHTML’s overrides is the slot concept in the Web Components specification. In fact, that inspired this! And it is such a wonderful way for both component authors and consumers to maintain each their ratio of control.

#### Deep Overrides

Taking things a little, it is possible for an import’s replacement node to be an import on its own. So if we were to use our article component in a page, for example, we could decide to use a totally different type of user component.

<!— file: index.html -->

<html>

<head>…</head>

<body>

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

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

</c-import>

</body>

</html>

Awesome yet with deep overrides? We could go even deeper as far as overriding requires!

### Inheritance

A namespace-based component layout represents a hierarchy of categories and sub categories. Subnamespaces are, in concept, built from off their supernamespace. In fact, by default, a subnamespace implicitly inherits everything from its supernamespace. In other words, everything in a supernamespace naturally applies to a subnamespace unless an override occurs.

By virtue of such inheritance, we would be safe to find a component deep in a subnamespace that is yet to be implemented, as long as fallbacks exist up the hierarchy. Let’s try that in the last import we made, this time, using a sub-article namespace that we’re yet to create.

<!—file: index.html -->

<html>

<head>…</head>

<body>

<c-import c-role=”body-main” c-ns=”content/article/readonly/layout1/dark”></c-import>

</body>

</html>

If we call things like this shadow namespaces, then they’re cool as they allow us to progressively build features into an app while using a real layout plan from the start.

Now combining the power of imports and inheritance, we can more easily build components off a supercomponent. Earlier, we used the import element to place a component for display. This time, we’re using the import element to redefine a supercomponent into a new subcomponent. So we’re creating the …layout1 sub-article component from off its supercomponent with important changes.

<!—file: content/article/readonly/layout1.html -->

<c-import>

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

</c-import>

# Behavioural Description

## Bindings, Reactions and Actions

A core concept in HTML is dynamic functionality. HTML5 is especially charged with application-related vibes like its data- attribute. CSS3 brought certain magic in new functions like calc(), var(), and tons of others.

So HTML kicks – on the element level; CHTML should kick too – this time, on the component level! Now we will be able to designate component-wide functionalities on the component’s root element; remember this is where the role of a component is established.

CHTML’s convention on functionality is tightly based on the web’s familiar declarative languages.

* In the spirit of decency, CHTML keeps its attribute footprint small on elements, and adopts a CSS-like syntax of stating things in key/value pairs within a single attribute. And like everyone else, we like CSS’s function expressions too.
* Another familiar syntax is JSON. JSON helps us represent values in the format they should be – number, string, logical true and false, collection of values (array), pairs of values (object).

Now, with CSS-like and JSON-like familiarity, we can solve a component’s logical problems declaratively.

The directives available in selectors are based on the query methods of the node API in use. The examples above are based on the default node API which is similar to jQuery’s. But details are just a few lines away.

### Roadmaps

Namespace paths can be used to tell more than just where a component lives.

# CHTML API

CHTML defines an API for translating a component’s Conceptual Model to an Object Model for programmatic use. This API can be implemented in any language and platform. But the specification and examples in this guide are of JavaScript implementation.

In its basic form, a CHTML component provides programmatic access to a component element and its nodes.

A component is created via the CHTML constructor. The constructor takes a DOM element as its 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.

(The el property is uniquely mapped to the root element and thus should not be used for a node role.)

Examples:

// Lets create a component on the DOM document itself

// and access its semantic nodes.

const doc = new Chtml(document);

// The document title

let title = doc.title;

// Write to the document body

doc.body.innerHtml = ‘Hello beautiful world!’;

Notes:

1. Nodes are lazy-loaded. So the DOM is accessed once for each node. The node is stored for subsequent access. This results in quite some performance gains especially where certain nodes were never accessed.
2. By default. The actual object returned from the Chtml constructor is a Proxy instance. A proxy gives us the benefit of accessing nodes as properties while forwarding each access to the component’s get() method. If Proxies raise browser support issues or performance issues on the browsers you target, this feature can easily be turned off, via the param.proxied property.

const doc = new Chtml(document, {proxied: false});

We must now use the components get method to access its nodes.

let title = doc.get(‘title’);

Conversely, with proxied instances, component methods are prefixed with the $ character to prevent the proxy from forwarding the method name as a node key.

let title = doc.title;

let title = doc.$get(‘title’);

You can always tell whether or not a component instance was proxied using the Chtml.proxied() static method.

if (Chtml.proxied(doc)) {

//

}

1. By default, the component root and node elements are DOM elements. But we could decide to have these DOM elements wrapped in a DOM abstraction object like jQuery. We can set a callback globally that receives these DOM elements, as they are created to return their abstraction.

ChtmlCore.nodeCallback = (node) => {

return $(node);

};

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

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

And we can go on to extend jQuery with some custom methods. We could as well decide on any other DOM abstraction library to use in our ChtmlCore.nodeCallback function. And that’s how easy it is to extend CHTML.

### The Chtml.from() Helper Method

In addition to creating component 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 component namespace, 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);

// Create an instance from a selector

const body = Chtml.from(‘body’);

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

// Create an instance from a namespace

const articles = Chtml.from(‘path/to/article’);

// Create an instance from markup, the from() method will automatically resolve it to an element.

const markup = ‘<div c-role=”comp”><span c-role=”node”></span></div>’;

const component = Chtml.from(markup);

Notes:

1. Where the given root element resolves to a Chtml list type, the from() method automatically creates a ChtmlList instance instead of a Chtml instance.
2. Inheritance is automatically resolved where the given component extends another component.

// Markup a component that extends another component.

const markup = ‘<div c-role=”comp” c-extends=”path/to/super/component”><span c-role=”node”></span></div>’;

const component = Chtml.from(markup);

1. Where the given root element resolves to a Chtml import type, it is automatically resolved to its final source component.

// Markup an import component

const import = ‘<c-import c-namespace=”path/to/super/component”><span c-role=”node”></span></c-import>’;

const component = Chtml.from(import);

### Nested Components and Drilldown

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

<div c-role=”article” id=”article”>

<div>

<div c-role=”article-author user”>

<div c-role=”user-avatar”></div>

<div c-role=”user-name”></div>

</div>

<div>

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

</div>

</div>

</div>

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

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

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});

let authorName = article.author.name;

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

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

let author = article.author.el;

let authorName = article.author.name.el;

Notes:

1. Drilldown always returns a component instance for every node, whether or not the node has been defined as a component in HTML. This makes everything more predictive when accessing a component tree.
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. If, for example, we initiated a proxied instance at the beginning, all nodes will be proxied, and vice versa.

Then some context-specific parameters must also change at each level of the chain. For example, the roadmap path we may have started with gets sliced down the road.

1. More is involved in figuring out a sub component. Inheritance must be resolved for nodes that extend other components. Nodes of type import must be resolved to their final source components. List types must translate to a ChtmlList instance. Thankfully, these and more are handled by the Chtml.from() method and drilldown wisely employs this method to resolve a node.

## Bindings and Reactivity

Here the collapsible component implements its control node outside of its root but a CSS selector provides a link to it. This is a computed node; a dynamically-adopted node.

Notice that the CSS selector itself is quoted. This allows us to differentiate between plain CSS selectors and other query schemes.

(1) Relative query schemes. These schemes specify a query method for finding the element. They are resolved relative to the component.

<div class=“wrapper”>

<div class=”bar”></div>

<div c-role=”collapsible” c-selectors=”control:sibling(‘.bar’)”></div>

</div>

(2) Relative query schemes with levels of dependency. These schemes specify a chain of dependencies for finding the element. Dependency chains are resolved relative to the component. Both the dot (.) and bracket ([]) notations are supported.

<div class=“wrapper”>

<div class=”bar”>

<div class=”icon”></div>

</div>

<div c-role=”collapsible” c-selectors=”control:sibling(‘.bar’); icon:control.find(‘.icon’)”></div>

</div>

Here the query that resolves the icon node is based on the availability of the control node. These queries can be in any order as long as a dependency is resolvable.

If we wanted only the icon as a computed node, we could find it without implementing the .bar as a control node. Our selector would look like:

icon:sibling(‘.bar’).find(‘.icon’)

If we wanted to reference a node in another component, we could do that in two separate queries. The first query would implement the foreign component as a local node, while the second query would be a dependency chain that’s based off this node. Notice how the author component now implements a quotes node by referencing the description node of its parent component.

<div c-role=”article”>

<div>

<div c-role=”article-author user” c-selectors=”article:closest(‘[c-role=article]’); quotes:article.description”>

<div c-role=”user-avatar”></div>

<div c-role=”user-name”></div>

</div>

<div>

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

</div>

</div>

</div>