Modular UI

Modular UI is the practice of dividing the UI into reusable, right-sized components. It is the modern wisdom that just makes interface development easier, faster, and more manageable going forward.

## Both the Methodology & Tooling

It calls for a bit of engineering - the discipline and tooling - to get the most of a modular approach to UI development. Modular UI captures the big picture of the UI’s three major languages in three simple methodologies and tools - Object-Oriented HTML (OOHTML), Class-Compositional CSS (CSS), OOHTML-API.

## What’s In for You and Your UI?

### A Universal and Forever UI Language!

We wanted not just a system of reusable components but components you can use across technology stacks; UI blue-prints you can take to any language of choice without transpiling or ever changing a code. Our deliberate choice of HTML and JSON for a UI language is your bet on relevance and universality. Just however the future comes, your bet on Modular-UI will always pay off!

### Loose Coupling

With Modular UI as a DOM Abstraction Layer, you will never again get functional code entangled with the DOM. The DOM now gets out of the way for better web app development as all we ever really wanted is a way to manipulate the UI without assuming or enforcing a DOM structure. Now you can simply continue evolving your UI without any back-and-forth implications.

### Granular Composition and Selective Evolution

We pushed beyond a declarative UI framework and introduced Separation Over Interpolation (SoI) –separating declarations from markup – unlike most frameworks. Now these different concerns can evolve independently, to any scale, without one getting in the way of the other. And you can simply narrow maintenance down to as little as a directive without digging through clutter!

### New Opportunities for Low-code Tooling

# Architectural Concepts - OOHTML

A component isa functionally-standalone UI block that can be reused*.* It anchors on a DOM element – the component’s root element; and may involve one or more other elements – nodes.

### **Components are role-based.**

Roles establish a component’s purpose within a given context. Fortunately, HTML has different ways of expressing semantics – via attributes and tagnames, resolving to hybrid and dedicated components in OOHTML respectively.

*1.a: Hybrid components*

Hybrid components have their role or roles defined in an attribute. We can base our design off the role attribute that already serves this purpose for assistive technologies, or the is attribute that draws the parallel in the Custom Elements API, or some other attribute that fits our semantics. We only need to be consistent with one convention.

<div role=”author">...</div>

Rolenames can be dash-separated, as allowed or required by the attribute type.

<div role=”namespace-author">...</div>

Using an attribute type that accepts multiple values, a component can play multiple roles.

<div role=”namespace-author namespace-anotherrole">...</div>

*1.b: Dedicated components*

The semantics of dedicated components come from their tagname. They are purpose-built for this role. It becomes needles to (re)define their role in an attribute.

Autonomous Elements in the Custom Elements API are dedicated. Tagnames are always dash-separated.

<namespace-author>...</namespace-author>

Certain *native* HTML elements are also *dedicated* by nature. For example, the head, title, and body elements are absolute in meaning within a document. These elements can be used without explicitly (re)defining their role, although it wouldn’t hurt either.

The code samples below are equally semantic in Modular-UI.

(a)

<html>

<head>

<title>…</title>

</head>

<body></body>

</html>

(b)

<html role=”document”>

<head role=”head”>

<title role=”title”>…</title>

</head>

<body role=”body”></body>

</html>

When using the role attribute for our Hybrid versions, then how a *native dedicated* element implies a certain role is guided by the same principles that govern ARIA implicit roles such that an implied role must also validate within its context. For example, a <header> element in the ARIA specification can only imply the role banner when not found within an <article> or <section> element.

// The header below has an implied banner role in OOHTML

<body>

<header></header>

</body>

// The header below does not imply a banner role in OOHTML, unless explicitly defined

<body>

<article>

<header></header>

</article>

</body>

### **Components can define nodes within their root.**

Nodes are strategically-placed members of a component. A simple class-naming convention does the association: a classname combining the super role and the sub role, separated by a hyphen.

*2.a: in a hybrid component, unnamespaced*

<div role="author">

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

</div>

// With multiple roles

<div role="author button">

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

<div class=”button-text”>…</div>

</div>

*2.b: in a dedicated component*

<certain-author>

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

</certain-author>

*2.c: in a dedicated component having an implied role*

// The <body> element implies a “body” role as a descendant of document

<body>

// The <header> element implies a “banner” role as a non-descendant of article or section

<header class=”body-banner”>

<div class=”banner-display”></div>

</header>

<main class=”body-main”></main>

</body>

### **Components can reference any element as node using CSS selectors.**

This comes into play where certain nodes do not live within the component root in the DOM, or where a component is being dynamically composed by a script. A special attribute – the selectors attribute – is used to map nodenames to DOM queries.

#### Syntax

<body selectors=”nodeName:query”>…</body>

For hybrid components that can naturally play more than one role, unique selectors attributes can be defined for each role using the rolename as the attribute’s namespace. On the other hand, a selectors attribute without the namespace will apply to all roles.

And different query schemes are allowed.

*3.a: CSS selectors*

<div class=”author-wrapper”>

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

<div role=”author” author:selectors=”avatar:`.author-avatar`; name:`.author-name`”></div>

</div>

*Complex CSS selectors are supported*

<div role=”author” author:selectors=”avatar:`.author-wrapper .author-avatar`; name:`.author-wrapper .author-name`”></div>

*3.b: Resolvable queries – selectors that are resolved relative the root element.*

<div class=”author-wrapper”>

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

<div role=”author” author:selectors=”avatar:{siblings:`.author-avatar`}; name:{siblings:`.author-name`}”></div>

</div>

*Dependencies are supported*

<div role=”author” author:selectors=”wrapper:{closest:`div`}; avatar:{wrapper.children:`.author-avatar`}; name:{wrapper.children:`.author-name`}”></div>

*Multiple resolutions are supported*

<div role=”author” author:selectors=”wrapper:{closest:`div`}; everything:{wrapper.children:`.author-avatar`; siblings:`.author-name`}”></div>

### **Components can be nested.**

This is where a component’s node is also defined as a component of its own – to make a child component.

<div role="article">

<div class=”some-div”>

<div class=”article-thumbnail”>…</div>

<div class=”article-title”>…</div>

</div>

<div class=”article-author” role="author">

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

</div>

</div>

Nesting respects scope boundaries such that sub roles are isolated in scope from their super roles. In other words, nodes are scoped to their closest associable role in hierarchy.

<div role="article card" id=”parent-component”>

<div class=”some-div”>

// These two DIVs belong to #parent-component

<div class=”article-thumbnail”>…</div>

<div class=”article-title”>…</div>

</div>

<div role="article" id=”child-component”>

// These two DIVs belong to #child-component

<div class=”article-thumbnail”>…</div>

<div class=”article-title”>…</div>

// This button belongs to #parent-component as a card node

<button class=”card-cta”></button>

</div>

</div>

# The Modular UI CSS

Building and maintaining CSS for modular UI comes with its unique challenges for development teams. Usually, the challenge lies in how they tend to draw the parallel between modularity in CSS and modularity in HTML. Many think it is best to adopt a component-scoped CSS approach in order to literally correspond with a component-based HTML.

With the new Shadow DOM technology, yes this can now be enjoyed. But in the real world, it turns out to be very impractical, if not impossible, as CSS inherently does not provide for such scoping. Here is where the class-composition approach comes in!

With the class-compositional approach, single-purpose classes are seen as CSS’ own unit of reusability – the CSS components. These classes are defined once and reused as desired in composing the HTML. This is the type of modularity in CSS that Modular-UI adopts. Now, structural classes like article, article-header, etc are dedicated to structure and consumed by the Modular-UI API, while classes like bold, faint, etc are used to compose presentation and behavior into components.

Modular-UI is keen on separation of concern, and thankfully holds a minimal class footprint for structure while leaving you with more classspace to compose presentational and behavioral classes – the same wisdom underscored by methodologies like OOCSS.

## Framework

Many class-compositional CSS frameworks are facilitating the paradigm shift.

# The Modular-UI API

It is possible to represent UI components as data objects for scripting languages like JavaScript, in what we can call a Modular-UI Object Model (MUIOM). An object in the MUIOM is a mirror of a UI component whose structural parts (nodes and child components) are exposed as properties. These objects stack up to produce a component view of the UI – the Component Tree, much like the DOM tree, that can be traversed and manipulated programmatically.

A mental model. An object model. JS Syntax

The Modular UI API can be implemented in any language and platform. But the specification and examples in this guide are of JavaScript implementation.

## Instantiation and Usage

MUIOM objects are created from the UICOM constructor. This constructor accepts a root element (either a DOM element, a string markup, or path to a markup file), then, an optional namespace parameter and an optional params object.

### Syntax

var component = new View(elem[, namespace[, params]]);

The namespace parameter is used to optionally associate the component instance to just one role in the case of hybrid components that support multiple roles. Where omitted, the component instance will operate on all defined roles.

With the params parameter, we can change a few things about the component instance. We’ll be demonstrating these as we go. Meanwhile, the API documentation has the implementation details.

Now, properties of the component instance are mapped to the underlying component’s nodes. The element property (as in component.element) is reserved to the component’s root element.

A node instance is not created until first access. This is some lazy-loading which significantly increases performance gains.

By default, nodes are made of jQuery instances.

#### Example

var article = new View(articleElement);

// To set the article title…

article.title.html(‘This is article title’);

// To access the articles’s author and then set the author’s name…

var authorView = new View(article.author);

authorView.name.html(‘Jack’);

### Selectors

The params object can be used to pass-in the component’s nodes via its selectors property. This is the JavaScript object version of the selectors attribute. But, while the selectors attribute does this definition in markup, the params.selectors parameter supplies this at runtime, with values overriding similar values from the selectors attribute, thus providing extra flexibility.

The params.selectors property offers all the possibilities of the selectors attribute.

<div class=”author-wrapper”>

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

<div id=”author-component” role=”author”></div>

</div>

var params = {selectors:{avatar:’.author-avatar‘, name:’.author-name‘,},};

var author = new View($(‘author-component’), ‘author’, params);

author.name.html(‘Jack’);

### Drilldown

It is possible to drilldown the component tree to any level seamlessly – without manually creating new View instances on nodes. We simply set the params.drilldown parameter to true, and a View instance is automatically returned for each node, instead of a jQuery object.

Since nodes are now View instances, a node’s element is then accessed via the element property.

var params = {drilldown:true};

var article = new View(articleElement, ‘article’, params);

article.title.element.html(‘This is article title’);

article.author.firstname.element.css(‘color’, ‘red’);

### Parser Options

Parser options are passed via the options.interpreterOpts property. These options affect how the compiler interprets and parses string conventions. For example to change the default quote character from the backtick to the single quote character, you use the quote property:

var params = {drilldown:true, interpreterOpts:{quote:“’”},};

var author = new View(authorElement, ‘author’, params);

// Notice we can now use the single quote instead of backticks

<div role=”author” author:selectors=”wrapper:{closest:’some-id’}”></div>

// Access now

author.wrapper.element.css(‘color’, ’red’);

### Selector Literals

Values in the params object can be referenced in selector expressions. Certain arbitrary values can be set in the params object for this purpose.

var params = {drilldown:true, someID:“#some-id”,};

var author = new View(authorElement, ‘author’, params);

// Now we’ll use the someID parameter below, which equates to “#some-id” as defined above

<div role=”author” author:selectors=”wrapper:{closest:someID}”></div>

// Access now

author.wrapper.element.css(‘color’, ’red’);

## Components Live in Folders

With composition being at the core of Modular-UI, components can be broken down into individual folders; each containing the component’s html file and certain optional definition files. This lets us reuse/import components just by folder path.

To create component instances from paths, the path to component is passed instead of an element instance.

var article = new View(‘path/to/article’, […]);

Component paths are always relative to a common base. But a base path isn’t necessary in a JavaScript environment as component paths would be resolved against an in-memory source. Implementations dealing with a real file system would want to accept such a common base path via some form of config. This way, component paths can remain consistent whatever the environment.

### The view.html file

In each component folder, the component’s html file must be named view.html.

Here are two components. One will be imported twice into the other – one article having two authors.

// author/view.html

<div role="author">

<div class=”author-avatar”>…</div>

<div class=”author-name”>…</div>

</div>

// article/view.html

// Notice how we define that the author component be imported

// Also remember that the value of the article:selectors attribute can be moved into a JSON file

<div id=”article-element” role="article" article:selectors=”author:{importBefore:`author`}; coauthor:{importBefore:`author`}”>

<div class=”some-div”>

<div class=”article-title”>Untitled</div>

<div class=”article-content”>…</div>

</div>

</div>

// Access the tree now

// Recall that nodes are lazy-loaded. So the author and coauthor components aren’t imported until they are first-accessed.

var article = new View($(‘#article-element’));

article.author.name.element.html(‘Jack’);

article.coauthor.name.element.html(‘Sawya’);

// Result

<div id=”article-element” role="article" article:selectors=”author:{importBefore:`author`}; coauthor:{importBefore:`author`}”>

<div role="author" class=”article-coauthor”>

<div class=”author-avatar”>…</div>

<div class=”author-name”>Sawya</div>

</div>

<div role="author" class=”article-author”>

<div class=”author-avatar”>…</div>

<div class=”author-name”>Jack</div>

</div>

<div class=”some-div”>

<div class=”article-title”>Untitled</div>

<div class=”article-content”>…</div>

</div>

</div>

### The selectors.json file

A component’s selectors definition can be moved into the component’s folder to a selectors.json file, but this is optional. To define selectors for a specific role in a hybrid component, the rolename should prefix the filename, as in article.selectors.json.

// article/selectors.json

{

“author”: {

“importBefore”:”`author`”

},

“coauthor”: {

“importBefore”:”`author`”

}

}

As with the params.selectors property, a selectors.json file comes with all the possibilities of the selectors attribute. And besides keeping our markup clean, the selectors.json idea gives us that Separation of Concern at its best.

### Folder Inheritance

Components can be laid out in a hierarchy of folders, where applicable. This architecture helps to naturally establish the relationship between main components and derived components. But besides the parent-child relationship, inheritance is also possible.

#### Full Inheritance

A component in a subfolder can inherit an entire file definition from its parent folder. This makes it easy to derive new components from existing ones without repeating code unnecessarily. This raises the reusability bar and ease of maintenance.

Here are two types of article components that use the code samples above to demonstrate inheritance – one is basic, featuring just the article title and content; the other is advanced, featuring also the author and coauthor. We could simply derive the authored one from the basic with a folder structure like article/ and article/authored/ respectively. All we have to do is define the selectors.json file in the subfolder. The view.html file and any other definition files that may be in the parent folder will be automatically inherited.

##### A: the article folder

// article/view.html

<div id=”article-element” role="article">

<div class=”some-div”>

<div class=”article-title”>Untitled</div>

<div class=”article-content”>…</div>

</div>

</div>

##### B: the article/authored folder

// article/authored/selectors.json

{

“author”: {

“importBefore”:”`author`”

},

“coauthor”: {

“importBefore”:”`author`”

}

}

#### Partial Inheritance

It is possible to extend or replace a part of an inherited JSON file definition. This only applies to the file’s root-level properties.

To demonstrate, the article/authored folder above may further be extended into an article that features a different type of component for just its coauthor while inheriting its author importation as-is.

// article/authored/advanced/selectors.json

{

“author”: {

“importBefore”:”`author`”

},

“coauthor”: {

“importBefore”:”`author/faded`”

}

}

### Composition Roadmap

Earlier we learnt that selectors definition can reference values in the params object passed-in at instantiation time. We can, in the same way, dynamically pass-in the path for importing subcomponents.

In the code below, notice the childComponentUri reference in the import directive and in the params object.

// article/authored/advanced/selectors.json

{

“author”: {

“importBefore”:”childComponentUri”

},

“coauthor”: {

“importBefore”:”childComponentUri”

}

}

var params = {childComponentUri: ‘author/advanced’}

var article = new Vieinstance(‘article/authored’, ‘article’, params);

But a sub-path can be passed more fluently by simply appending it to the base path using double forward slashes as the separator. Here is the convention: path/to/base//path/to/child. This path gets automatically broken apart and the descendant path gets automatically set in the instance params object as childComponentUri.

var article = new View(‘article/authored//author/advanced’, ‘article’, params);

If we wanted to pass-in the author’s next level of import, we could.

var article = new View(‘article/authored//author/advanced//avatar/round’, ‘article’, params);

Now we have a roadmap for the components down the line.

#### Repeating Patterns

There will be cases of repeating patterns along a composition roadmap. A menu tree, for example, is structurally made of repeating parent-child relationships, often in the <ul> - <li> combo. If we had the <ul> and <li> as separate components, here is what the composition roadmap could look like.

ul//li//ul//li//ul//li

Here the ul component imports li components; and each li component imports ul components, and the import chain repeats continuously down the line.

But couldn’t we just automate the repetition? Yes! We could simply express a continuous import pattern with trailing double forward slashes: ul//li//. Now each time it seems we’ve hit the end of the road, the chain repeats from the beginning.

##### Midway Repeats

A smart roadmap could even be smarter. Repetitions do not always have to start all the way from the beginning. A repeating pattern might only be desired somewhere along the full composition roadmap.

Here is the menu tree again, but this time, having root-level items different from sublevel items. Double pipe characters are used to mark the starting point for repetition.

// Repetition should start after the first two levels

headermenu//headermenuitem||ul//li//

Now ul//li// is repeated as many times as needed to create the submenu.

### Module Reassembly and Exporting

A component can be reassembled from its different little files into a single code block or module. Then multiple modules can be generated from the file system into a portable export file for use by other technologies. This is automatically made possible in a static export() function. This function allows us to optionally select the components to export, and to optionally specify in what format to create the export: HTML or JSON. The default format is HTML.

var export = View.export([componentsPaths, [format]]);

A module is made from its view.html file (or its parent’s, when inherited), and its path is automatically set in a component-uri attribute so as to be uniquely identified. Its selectors.json and rendering.json files are automatically serialized into the selectors and rendering attributes respectively, all with inheritance applied.

With the HTML format, modules are wrapped in a <template> tag, which can easily be imported into modern browsers using the HTMLImports API.

// export.html

<template>

<div role=”article” selectors=”…” rendering=”…” component-uri=”article”>

<div class=”article-title”></div>

<div class=”article-content”></div>

</div>

<div role=”author” selectors=”…” rendering=”…” component-uri=”author”>

<div class=”author-name”></div>

<div class=”author-avatar”></div>

</div>

</template>

With the JSON format, modules serialized to JSON strings and are listed in an array. JSONs can as well be easily imported by modern browsers as an ES6 module.

// export.json

[

“<div role=\”article\” selectors=\”…\” rendering=\”…\” component-uri=\”article\”>

<div class=\”article-title\”></div>

<div class=\”article-content\”></div>

</div>”,

“<div role=\”author\” selectors=\”…\” rendering=\”…\” component-uri=\”author\”>

<div class=\”author-name\”></div>

<div class=\”author-avatar\”></div>

</div>”

]

### Components are Renderable

The component API features a render method that brings some more magic to the component object. This method is used to manipulate or write application data to the component following some rendering instructions called directives.

Syntax

component.render(data);

### Data

Data is any object or array.

var data1 = {firstname: “Oxford”, lastname:“Harrison”, id:”012345”};

var data2 = {names: [“Ewan”, “Harry”], bio:”About User”, location:{country:”USA”, city:”New York”}};

The UICOM’s render method draws a parallel with the DOM API rendering ability.

### Directive Bindings

To render data, a component must define an object that contains rendering instructions – each a pair of a directive expression and an argument expression.

Directive expressions query and act on the component object while argument expressions query the data object or are simply a value.

Syntax:

<body directives=”directiveExpression:argumentExpression”>…</body>

For hybrid components that potentially play more than one role, unique directives attributes can be defined for each role using the rolename as the attribute’s namespace. A directives attribute without the namespace will apply to all roles.

// Write to the root element’s content using the html directive.

<div role="author" author:directives=”html:firstname”></div>

// Render data

component.render(data1);

// Result

<div role="author" author:directives=”html:firstname”>Oxford</div>

#### Directives

Directives are actions that can be called on a DOM node. Simply put, they’re DOM methods – as defined in the DOM API. These methods are called with their respective bound arguments resolved and passed in.

The code above employed the html directive. But that’s just one. Here are a few more. Their respective signatures are documented separately under the jQuery API.

attr(), css(), append(), prepend(), etc.

Custom directives can be achieved by simply adding new methods to the jQuery API. It’s a smart way of increasing rendering power.

#### Arguments

An argument (the right part of a directive binding) is any value that is of the type accepted by the method. An argument can be any literal string (enclosed in quotes), a reference to a value in the data object (unquoted string), a mathematical expression that may contain either or both, or a more complex value type like the array and object (using the JavaScript Array and Object notations respectively).

An equivalent to the JavaScript spread notation for argument list is also supported. This is useful where a list of arguments has to be passed to the method call. An object spread notation is a regular array notation preceded by three dots.

See the Type Notation appendix for details.

// To write to the element’s title attribute using argument spread

<div role="author" author:directives=”attr:…[`title`, id]”></div>

#### Drilldown Notations

Drilldown notations can be used in both the directive expression and the argument expression to query deep in the component object tree and data object tree respectively. The dot notation and the bracket notation can be used interchangeably. But keys in brackets must be quoted, except in the case of numeric keys and literals.

The “this” keyword in a Drilldown notation can be used to mean the root object, but this is not required.

##### Dot Notation Examples

// Write to the root element’s content using the this.html directive.

<div role="author" author:directives=”this.html:firstname”></div>

// 1-level and 2-level dot notations

<div role=”article” article:directives=”body.html:`Content…`; author.bio.html:bio”>

<div class=”article-header”>…</div>

<div class=”article-body”>…</div>

<div class=”article-author” role="author">

<div class=”author-avatar fa”>…</div>

<div class=”author-name”>…</div>

<div class=”author-bio”>…</div>

</div>

</div>

// Render data

component.render(data2);

The dot notation can be used in the argument expression to query deep in the data object tree.

// Write to the root element’s content using the this.firstname argument expression.

<div role="author" author:directives=”html:this.firstname”></div>

// Write to the root element’s content using the location.country (or this.location.country) argument expression.

<div role="author" author:directives=”html:location.country”></div>

// Render data

component.render(data2);

##### Bracket Notation Examples

// 1-level bracket notation

<div role=”article” article:directives=”body.html:`Content…`; author[`name`].html:names[0]”>

<div class=”article-header”>…</div>

<div class=”article-body”>…</div>

<div class=”article-author” role="author">

<div class=”author-avatar fa”>…</div>

<div class=”author-name”>…</div>

</div>

</div>

// Render data

component.render(data2);

##### Bracket Literals

Unquoted keys in brackets are special literals or variables that are used to pre-insert a data value to produce a new expression during parsing.

Below, an author’s name is shown in either of two nodes depending on the author’s gender. Notice how the gender type is injected into the directive expression to form a new directive after parsing.

// The user object

var data4 = {name:”Jack”, gender:”male”};

var data5 = {name:”Gloria”, gender:”female”};

<div role=”article” article:directives=”author[gender].html:name”>

<div class=”article-author” role="author">

<div class=”author-male”>…</div>

<div class=”author-female”>…</div>

</div>

</div>

// Render data

// component.render(data3);

component.render(data4);

#### Inline Calls

Data objects may, in addition to properties, also define certain methods that can be called at runtime to return the value to render. These methods are used with a straight-forward syntax within the argument expression.

Below we write a return value to the element’s title attribute using an object-type argument. This works with the assumption that the getFull() method is defined on the location object.

// Write the user’s full location

<div role="author" author:directives=”attr:…[`title`, location.getFull()]”></div>

Inline calls can also pass-in arguments. Arguments to these methods are resolved from the data object.

// Write the user’s full location, comma-separated

<div role="author" author:directives=”attr:…[`title`, location.join(`, `)]”></div>

Inline calls are chainable.

// Write the user’s full location, comma-separated, upper-cased

<div role="author" author:directives=”attr:…[`title`, location.join(`, `).toUpperCase()]”></div>

#### Closure Constructs

Closures are flow control constructs that establish a new context within which to execute certain directives. Data object methods can define a callback parameter. The execution of a callback and the parameters available to the directives within are determined by a calling function.

Below, we use the JavaScript Array’s forEach() function to iterate and render the names list in data3. Notice the syntax for this function’s callback parameter.

var data3 = {names: [“Ewan”, “Harry”], id:”012347”, location:{country:”USA”, city:”New York”}};

// Iterate on the names array and append each in content

<div role="author" author:directives=”names.forEach(…(name)): {append:name}”></div>

// Iterate on the names array and write the first, then the rest in self duplicates

<div role="author" author:directives=”names.each(…(index, name)): {append:name}”></div>

component.render(data3);

#### Conditional Constructs

These are flow control constructs that establish a condition that must evaluate to true in order to render certain directives. A condition can be any expression.

// Show the avatar icon only if firstname evaluates to true

<div role="author" author:directives=”firstname:{avatar.addClass:fa-user; name.html:firstname + ` ` + lastname}”></div>

// Show the avatar icon only if the user’s id falls within the first 1000

<div role="author" author:directives=”id < 1000:{avatar.addClass:fa-user; name.html:firstname + ` ` + lastname}”></div>

### Relationship with Other Methodologies and Tools

How does Modular-UI relate with the DOM? Simple: Just as an element is the DOM’s smallest unit of composition, a UI component is the UI’s smallest unit of composition. And just as a DOM element can render application data using methods like el.innerHtml, el.setAttribute(), etc, a UI component also features a render() method that writes application data into itself.

What about the many patterns that exist for coding CSS and marking up CSS classes, e.g OOCSS, SMACSS, SUITCSS, Atomic, and BEM? Scope is a major difference. Modular UI is not just about the CSS architecture and CSS-classes convention. Thinking really far, it also provides the API that opens up new possibilities with the UI.

Next, frameworks like Vue and Angular. Now Modular UI is about, not the behavior of UI components, but the architectural and presentational components upon which behaviors can be built. But it provides a render() method as this functionality feels more natural in a lower, foundational level API than that of behavioral components.

Modular UI also draws a measure of parallel in thinking with existing technologies like ARIA and Web Components, and is designed to integrate with these and other technologies happily. You can simply deploy Modular UI into any technology of your choosing. For example, you will, in fact, need Modular UI to build better Web Components, if that is your technology of choice.

Type Notations

* Array notation
* Object notation
* Literals/Identifier
* String
  + Backticks are the default quotes.
* Arguments spread
* Closure argument

Attributes and Namespacing