DOCS

We've got a very rich set of docs planned for Aurelia. Unfortunately, we haven't quite finished them yet. However, for this early preview period, we've put together this document, containing examples of the most common tasks you might want to perform. If you have questions, we hope that you will join us on our gitter channel (https://gitter.im/aurelia/discuss).

> **Note:** Looking for this guide in another language? Have a look in our documentation repo (https://github.com/aurelia/documentation).

Browser Support

Aurelia was originally designed for Evergreen Browsers. This includes Chrome, Firefox, IE11 and Safari 8. However, we have identified how to support IE9 and above. To make this work, you need to add an additional polyfill for MutationObservers. This can be achieved by a jspm install of

`github:polymer/mutationobservers`. Then wrap the call to `aurelia-bootstrapper` as follows:

```
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
  // Loads WeakMap polyfill needed by MutationObservers
  System.import('core-js').then( function() {
    // Imports MutationObserver polyfill
    System.import('polymer/mutationobservers').then( function() {
      // Ensures start of Aurelia when all required IE9 dependencies are loaded
      System.import('aurelia-bootstrapper');
    })
  });
</script>
```

> **Note:** WeakMap is not required by Aurelia itself but it is used by the MutationObserver polyfill.

Startup & Configuration

Most platforms have a "main" or entry point for code execution. Aurelia is no different. If you've read the Get Started (http://aurelia.io/get-started.html) page, then you've seen the `aurelia-app` attribute. Simply place this on an HTML element

and Aurelia's bootstrapper will load an *app.js* and *app.html*, databind them together and inject them into the DOM element on which you placed that attribute.

Often times you want to configure the framework or run some code prior to displaying anything to the user though. So chances are, as your project progresses, you will migrate towards needing some startup configuration. In order to do this, you can provide a value for the `aurelia-app` attribute that points to a configuration module. This module should export a single function named `configure`. Aurelia invokes your `configure` function, passing it the Aurelia object which you can then use to configure the framework yourself and decide what, when, and where to display your UI. Here's an example configuration file:

```
import {LogManager} from 'aurelia-framework';
import {ConsoleAppender} from 'aurelia-logging-console';

LogManager.addAppender(new ConsoleAppender());
LogManager.setLevel(LogManager.logLevel.debug);

export function configure(aurelia) {
  aurelia.use
    .defaultBindingLanguage()
    .defaultResources()
    .history()
    .router()
    .eventAggregator()
    .plugin('./path/to/plugin');

  aurelia.start().then(a => a.setRoot('app', document.body));
}
```

With the exception of the custom plugin, this code is essentially what `aurelia-app` normally does for you. When you switch to the configuration file approach, you need to configure these things yourself, but you can also install custom plugins, set up the dependency injection container with some services, and install global resources to be used in view templates.

If you want to switch to the configuration file approach for startup, you can actually write a simple file that sums up all the standard options which we spelled out above. Here's what that would look like:

```
export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .developmentLogging();

  aurelia.start().then(a => a.setRoot());
}
```

Logging

Aurelia has a simple logging abstraction that the framework itself uses. By default it is a no-op. The configuration above shows how to install an appender which will take the log data and output it the console. You can also see how to set the log level. Values for the `logLevel` enumeration include: `none`, `error`, `warn`, `info` and `debug`.

You can easily create your own appenders. Simply implement a class that matches the appender interface. The best way to see how to do this is to look at our own console log appender's source (https://github.com/aurelia/logging-console/blob/master/src/index.js).

Plugins

A *plugin* is only a module with an exported `configure` function. During startup Aurelia will load all plugin modules and call their `configure` functions, passing to

them the Aurelia instance so that they can configure the framework appropriately. Plugins can optionally return a `Promise` from their `configure` function in order to perform asynchronous configuration tasks. When writing a plugin, be sure to explicitly supply all metadata, including a View Strategy for Custom Elements.

In order to do configuration on your plugin from within the app you can specify a function or object as the second argument to the `configure` function. Your plugin's install function can then use that after your installation work is done. The consumer of your plugin might write code like this:

```
aurelia.use.plugin('./path/to/plugin', config => { /* configuration work */ });
```

> **Note:** Do not rely on naming conventions inside plugins. You do not know how the consumer of your plugin will change Aurelia's conventions. 3rd party plugins should be explicit in order to ensure that they function correctly in different contexts.

Promises

By default, Aurelia uses ES6 native Promises or a polyfill. However, you can replace this with the excellent Bluebird (https://github.com/petkaantonov/bluebird) Promise library. Simply include it in your page before you reference the other scripts. It will

provide its own standards-compliant Promise implementation which is currently faster than native and has better debugging support. Additionally, when used in combination with the Babel transpiler, you can use coroutines (http://babeljs.io/docs/usage/transformers/other/bluebird-coroutines/) for improved async code.

The Aurelia Object

Since both a custom configuration module and plugins do their work by interacting with the Aurelia object, we provide a brief explanation of that API in code below:

```
export class Aurelia {
  loader:Loader; //the module loader
  container:Container; //the app-level dependency injection container
  use:Plugins; //the plugins api (see above)

  withInstance(type, instance):Aurelia; //DI helper method (pass through to containe
  withSingleton(type, implementation):Aurelia; //DI helper method (pass through to c
  globalizeResources(...resourcePaths):Aurelia; //module ids of resources relative t
  renameGlobalResource(resourcePath, newName); //renames a globally available resour

  start():Promise; //starts the framework, causing plugins to be installed and resou
  setRoot(root, applicationHost):Promise; //set your "root" or "app" view-model and
}
```

Views and View Models

In Aurelia, user interface elements are composed of *view* and *view-model* pairs. The *view* is written with HTML and is rendered into the DOM. The *view-model* is written with JavaScript and provides data and behavior to the *view*. The templating engine and/or DI are responsible for creating these pairs and enforcing a predictable lifecycle for the process. Once instantiated, Aurelia's powerful *databinding* links the two pieces together allowing changes in your data to be reflected in the *view* and vice versa. This Separation of Concerns is great for developer/designer collaboration, maintainability, architectural flexibility, and even source control.

Dependency Injection (DI)

View-models and other interface elements, such as Custom Elements and Custom Attributes, are created as classes which are instantiated by the framework using a dependency injection container. Code written in this style is easy to modularize and test. Rather than creating large classes, you can break things down into small objects that collaborate to achieve a goal. The DI can then put the pieces together for you at runtime.

In order to leverage DI you, simply decorate your class to tell the framework what it should pass to its constructor. Here's an example of a view-model that depends on Aurelia's HttpClient.

```
import {inject} from 'aurelia-framework';
import {HttpClient} from 'aurelia-http-client';

@inject(HttpClient)
export class CustomerDetail{
    constructor(http){
        this.http = http;
    }
}
```

With ES7 or TypeScript Decorators enabled, you just add the `inject` decorator, passing one argument per injected type. If you aren't using a language that supports Decorators, or just don't want to use them, you can also add a static property or method to your class named `inject`. This must return an array of injectable types. Here's the same example in CoffeeScript with CommonJS modules:

```
HttpClient = require('aurelia-http-client').HttpClient;

class Flickr
  constructor: (@http) ->
  @inject:[HttpClient]
```

If you are working with TypeScript, you can use the `--emitDecoratorMetadata` compiler flag along with Aurelia's `@autoinject` decorator to enable the framework to read the standard TS type information. As a result, there's no need to duplicate the types. Here's what that looks like:

```
import {autoinject} from 'aurelia-framework';
import {HttpClient} from 'aurelia-http-client';

@autoinject
export class CustomerDetail {
    constructor(public http:HttpClient) {
        this.http = http;
    }
}
```

> **Note:** There's an interesting detail of the way that TypeScript implements this compilation option. It actually works with any decorator. So, if you've got other decorators on your TS class, there's no need to include the `autoinject` decorator. The Type information will still be discoverable by Aurelia's dependency injection framework.

When explicitly declaring dependencies, it's important to know that they don't have to be just constructor types. They can also be instances of `resolvers`. For example, have a look at this:

```
import {Lazy, inject} from 'aurelia-framework';
import {HttpClient} from 'aurelia-http-client';

@inject(Lazy.of(HttpClient))
export class CustomerDetail{
    constructor(getHTTP){
        this.getHTTP = getHTTP;
    }
}
```

The `Lazy` resolver doesn't actually provide an instance of `HttpClient`. Instead, it provides a function which, when called, will return you an instance of HttpClient. There are several different resolvers out-of-the-box and you can create your own by authoring a class that inherits from `Resolver`. Here's a list of what we provide for you:

- `Lazy` - Injects a function for lazily evaluating the dependency.
  - ex. `Lazy.of(HttpClient)`
- `All` - Injects an array of all services registered with the provided key.
  - ex. `All.of(Plugin)`
- `Optional` - Injects an instance of a class only if it already exists in the container; null otherwise.
  - ex. `Optional.of(LoggedInUser)`

In addition to these resolvers, you can also use `Registration` decorators to specify

the default registration or lifetime for an instance. By default, the DI container assumes that everything is a singleton instance; one instance for the app. However, you can use a registration decorator to change this. Here's an example:

```
import {transient, inject} from 'aurelia-framework';
import {HttpClient} from 'aurelia-http-client';

@transient()
@inject(HttpClient)
export class CustomerDetail{
    constructor(http){
        this.http = http;
    }
}
```

Now, each time the DI container is asked for an instance of `CustomerDetail` the container will return a new instance, rather than a singleton. `singleton` and `transient` registrations are provided out-of-the-box, but you can create your own by writing a class that implements the following method `register(container, key, fn)`. Then, simply add an instance of it to a class with the `registration` decorator.

If you can't or don't want to use decorators, don't worry. We have a fallback mechanism. Simply provide a static `decorators` property or method and then use our chainable `Decorators` helper. The helper has methods for all our decorators, so it's easy for you to use in any language. Here's how the above example could be

written in CoffeeScript:

```
HttpClient = require('aurelia-http-client').HttpClient;
Decorators = require('aurelia-framework').Decorators;

class CustomerDetail
  constructor: (@http) ->
  @decorators:Decorators.transient().inject(HttpClient);
```

Parent View Models

By default a View-model's access is limited to injected objects as well as children of the class. Sometimes it may be desirable to refer to objects and methods on a parent View-model, which can be achieved by storing the parent during the *bind* method of the view lifecycle:

```
class ChildViewModel {
  bind(bindingContext) {
    this.$parent = bindingContext;
  }
}
```

Templating

Aurelia's templating engine is responsible for loading your views and their required

resources, compiling your HTML for optimal performance and rendering your UI to the screen. To create a view, all you need to do is author an HTML file with an `HTMLTemplate` inside. Here's a simple view:

```
<template>
    <div>Hello World!</div>
</template>
```

Everything inside the `template` tag will be managed by Aurelia. However, since Aurelia uses HTMLImport technology to load views, you can also include links, and they will be properly loaded, including relative resource resolution semantics. In other words, you can do this:

```
<link rel="stylesheet" type="text/css" href="./hello.css">

<template>
    <div class="hello">Hello World!</div>
</template>
```

This enables you to dynamically load per-view style sheets and even Web Components on the fly.

Any time you require an Aurelia-specific resource, such as an Aurelia *Custom Element*, *Custom Attribute* or *Value Converter*, you should use a `require` element inside your view instead. Here's an example:

```
<template>
  <require from='./nav-bar'></require>

  <nav-bar router.bind="router"></nav-bar>

  <div class="page-host">
    <router-view></router-view>
  </div>
</template>
```

In this case `nav-bar` is an Aurelia *Custom Element* which we've required for use. Using Aurelia's `require` element causes the framework's resource pipeline to process the imported item, which has the following advantages:

- Deduping - The resource is downloaded once in the app. Even if other views require the same element, it will not be downloaded again.
- One-time Compilation - Templates for Custom Elements required this way are compiled once for the entire application.
- Local Scope - The required resource is only visible inside the view that requires it, reducing the likelihood of name conflicts and improving maintainability and understandability by eliminating globals.
- Renaming - Resources can be renamed during require if two 3rd party resources with the same or similar name need to be used in the same view.
    - ex. `<require from="./nav-bar" as="foo-bar"></require>` - Now instead

of using a `nav-bar` element you can use a `foo-bar` element. (This is based on ES6 import syntax where renaming is considered a replacement for using an Alias because it strictly renames the type.)

- Packages - The require can point to a module with multiple resources which will all be imported into the same view.
- Extensibility - You can define new types of resources which, when required in this way, can execute custom loading (async one-time) and registration (once per-view). This is a declarative, extensible resource loading pipeline.
- ES6 - Code is loaded by the ES6 loader rather than the HTMLImport mechanism, enabling all the features and extensibility of your loader. This design choice fully unifies all app resource loading, whether through JavaScript or HTML.

In your view you will often leverage the different types of resources mentioned above as well as databinding.

> **Note:** You may be concerned about the tediousness of having to import things into each view. Remember, during the bootstrapping phase you can configure Aurelia with global resources to be available in every view. Just use `aurelia.globalizeResources(...resourcePaths)`.

Aurelia polyfills browsers that don't support templates. However, a few features of

templates can't be polyfilled and require workarounds. In particular this occurs when adding `<template>` elements inside `<select>` and `<table>` elements. The following can't be done in a browser that doesn't natively support templates:

```
<table>
  <template repeat.for="customer of customers">
    <tr>
      <td>${customer.fullName}</td>
    </tr>
  </template>
</table>
```

In order to repeat over the `<tr>` elements, simply add the `repeat` on the `<tr>` itself:

```
<table>
  <tr repeat.for="customer of customers">
    <td>${customer.fullName}</td>
  </tr>
</table>
```

Databinding

Databinding allows you to link the state and behavior in a JavaScript object to an HTML view. When this link is established, any changes in linked properties can be synced in one or both directions. Changes in the JavaScript object can be reflected

in the view and changes in the view can be reflected in the JavaScript object. To establish this link, you will leverage "binding commands" in your HTML. Binding commands are clearly identifiable via their use of the "." as a kind of binding operator. Whenever an HTML attribute contains a ".", the compiler will pass the attribute name and value off to the binding language for interpretation. The result is one or more binding expressions that are capable of establishing the linkage when the view is created.

You can extend the system with your own binding commands, but Aurelia provides a collection to cover the most common use cases.

bind, one-way, two-way & one-time

The most common binding command is `.bind`. This will cause the property to be bound using a "one-way" binding for all attributes, except form element values, which are bound with a "two-way" binding.

What does this mean though?

One-way binding means that changes flow from your JavaScript view-models into the view, not from the view into the view-model. Two-way binding means that changes flow in both directions. `.bind` attempts to use a sensible default by assuming that if you are binding to a form element's value property then you probably wish the changes made in the form to flow into your view-model. For everything else it uses one-way binding, especially since, in many cases, two-way

binding to non-form elements would be nonsensical. Here's a small binding example using `.bind`:

```
<input type="text" value.bind="firstName">
<a href.bind="url">Aurelia</a>
```

In the above example, the `input` will have its `value` bound to the `firstName` property on the view-model. Changes in the `firstName` property will update the `input.value` and changes in the `input.value` will update the `firstName` property. On the other hand, the `a` tag will have its `href` bound to the `url` property on the view-model. Only changes in the `url` property will flow into the `href` of the `a` tag, not the other way.

You can always be explicit and use `.one-way` or `.two-way` in place of `.bind` though. A common case where this is required is with Web Components that function as input-type controls. So, you can imagine doing something like this:

```
<markdown-editor value.two-way="markdown"></markdown-editor>
```

In order to optimize performance and minimize CPU and memory usage, you can alternatively leverage the `.one-time` binding command to flow data from the view-model into the view "one time". This will happen during the initial binding phase, after which, no synchronization will occur.

delegate, trigger & call

Binding commands don't only connect properties and attributes, but can be used to trigger behavior. For example, if you want to invoke a method on the view-model when a button is clicked, you would use the `trigger` command like this:

```
<button click.trigger="sayHello()">Say Hello</button>
```

When the button is clicked, the `sayHello` method on the view-model will be invoked. That said, adding event handlers to every single element like this isn't very efficient, so often times you will want to use event delegation. To do that, use the `.delegate` command. Here's the same example but with event delegation instead:

```
<button click.delegate="sayHello()">Say Hello</button>
```

The `$event` property can be passed as an argument to a delegate/trigger function call if you need to access the event object.

```
<button click.delegate="sayHello($event)">Say Hello</button>
```

> **Note:** If you aren't familiar with event delegation, it's a technique that uses the bubbling nature of DOM events. When using `.delegate`, a single event handler is attached to the document, rather than on each element. When the element's event is fired, it bubbles up the DOM until it reaches the document, where it is handled. This is a more memory efficient way of handling events and it's recommended to use this as your default mechanism.

All of this works against DOM events in some way or another. Occasionally you may have an Aurelia Custom Attribute or Element that wants a reference to your function directly so that it can invoke it manually at a later time. To pass a function reference, use the `.call` binding (since the attribute will *call* it later):

```
<button touch.call="sayHello()">Say Hello</button>
```

Now the Custom Attribute `touch` will get a function that it can call to invoke your `sayHello()` code. Depending on the nature of the implementor, you may be able to receive data from the caller. This works the same as with trigger/delegate by providing an `$event` object.

string interpolation

Sometimes you need to bind properties directly into the content of the document or interleave them within an attribute value. For this, you can use the string interpolation syntax `${expression}`. String interpolation is a one-way binding, the output of which is converted to a string. Here's an example:

```
<span>${fullName}</span>
```

The `fullName` property will be interpolated directly into the span's content. You can also use this to handle css class bindings like so:

```
<div class="dot ${color} ${isHappy ? 'green' : 'red'}"></div>
```

In this snippet "dot" is a statically present class and "green" is present only if `isHappy` is true, otherwise the "red" class is present. Additionally, whatever the value of `color` is...that is added as a class.

> **Note:** You can use simple expressions inside your bindings. Don't try to do anything too fancy. You don't want code in your view. You only want to establish the linkage between the view and its view-model.

ref

In addition to commands and interpolation, the binding language recognizes the

use of a special attribute: `ref`. By using `ref` you can create a local name for an element which can then be referenced in another binding expression. It will also be set as a property on the view-model, so you can access it through code. Here's a neat example of using `ref`:

```
<input type="text" ref="name"> ${name.value}
```

You can also use `ref` as a binding command to get the view-model instance that backs an Aurelia Custom Element or Custom Attribute. By using this technique, you can connect different components to each other:

```
<producer producer.ref="producerVM"></producer>
<consumer input.bind="producerVM.output"></consumer>
```

`producer.ref="producerVM"` creates an alias to the view-model for the `producer` custom element which you can then use elsewhere to pass to another custom element or to use properties of the VM. Thus in the second line of the code above, `consumer` has a property called `input` that has been bound to the `output` property of the VM of the `producer`. There are a few ways to use `ref` to reference elements and view-models:

- `attribute-name.ref="someIdentifier"` - create a reference to a custom attribute's class instance

- `element-name.ref="someIdentifier"` - create a reference to a custom element's class instance
- `ref="someIdentifier"` - create a reference to the HTMLElement in the DOM

select elements

`value.bind` on an HTMLSelectElement has special behavior to support the element's single and multi-select modes as well as binding to objects.

A typical select element is rendered using a combination of `value.bind` and `repeat`, like this:

```
<select value.bind="favoriteColor">
    <option>Select A Color</option>
    <option repeat.for="color of colors" value.bind="color">${color}</option>
</select>
```

Sometimes you want to work with object instances rather than strings. Here's the markup for building a select element from a theoretical array of employee objects:

```
<select value.bind="employeeOfTheMonth">
  <option>Select An Employee</option>
  <option repeat.for="employee of employees" model.bind="employee">${employee.fullNa
</select>
```

The primary difference between this example and the previous example is we're storing the option values in a special property, `model`, instead of the option

element's `value` property which only accepts strings.

multi select elements

You can bind the select element's value to an array property in multi-select scenarios. Here's how you'd bind an array of strings, `favoriteColors`:

```
<select value.bind="favoriteColors" multiple>
    <option repeat.for="color of colors" value.bind="color">${color}</option>
</select>
```

This works with arrays of objects as well:

```
<select value.bind="favoriteEmployees" multiple>
  <option repeat.for="employee of employees" model.bind="employee">${employee.fullNa
</select>
```

radios

`checked.bind` on an HTMLInputElement has special behavior to support binding non-boolean values such as strings and objects.

A typical radio button group is rendered using a combination of `value.bind` and `repeat`, like this:

```
<label repeat.for="color of colors">
  <input type="radio" name="clrs" value.bind="color" checked.bind="$parent.favoriteC
  ${color}
</label>
```

Sometimes you want to work with object instances rather than strings. Here's the markup for building a radio button group from a theoretical array of employee objects:

```
<label repeat.for="employee of employees">
  <input type="radio" name="emps" model.bind="employee" checked.bind="$parent.employ
  ${employee.fullName}
</label>
```

The primary difference between this example and the previous example is we're storing the input values in a special property, `model`, instead of the input element's `value` property which only accepts strings.

You can also bind a radio group to a boolean property like this:

```
<label><input type="radio" name="tacos" model.bind="null" checked.bind="likesTacos"
<label><input type="radio" name="tacos" model.bind="true" checked.bind="likesTacos"
<label><input type="radio" name="tacos" model.bind="false" checked.bind="likesTacos"
```

checkboxes

To better support multi-select scenarios Aurelia enables binding an input element's

checked property to an array. Here's how you'd bind an array of strings, `favoriteColors`:

```
<label repeat.for="color of colors">
  <input type="checkbox" value.bind="color" checked.bind="$parent.favoriteColors" />
  ${color}
</label>
```

This works with arrays of objects as well:

```
<label repeat.for="employee of employees">
  <input type="checkbox" model.bind="employee" checked.bind="$parent.favoriteEmploye
  ${employee.fullName}
</label>
```

You can of course bind each checkboxes to it's boolean properties like this:

```
<li><label><input type="checkbox" checked.bind="wantsFudge" />Fudge</label></li>
<li><label><input type="checkbox" checked.bind="wantsSprinkles" />Sprinkles</label><
<li><label><input type="checkbox" checked.bind="wantsCherry" />Cherry</label></li>
```

innerHTML

You can bind an element's `innerHTML` property using the `innerhtml` attribute:

```
<div innerhtml.bind="htmlProperty"></div>
<div innerhtml="${htmlProperty}"></div>
```

Aurelia provides a simple html sanitization converter that can be used like this:

```
<div innerhtml.bind="htmlProperty | sanitizeHtml"></div>
<div innerhtml="${htmlProperty | sanitizeHtml}"></div>
```

You're encouraged to use a more complete html sanitizer such as sanitize-html (https://www.npmjs.com/package/sanitize-html). Here's how you would build a converter using this package:

```
jspm install npm:sanitize-html
```

```
import sanitizeHtml from 'sanitize-html';

export class MySanitizeHtmlValueConverter {
  toView(untrustedHtml) {
    return sanitizeHtml(untrustedHtml);
  }
}
```

> NOTE: Binding using the `innerhtml` attribute simply sets the element's `innerHTML` property. The markup does not pass through Aurelia's templating system. Binding expressions and require elements will not be evaluated. A solution for this scenario is being tracked in aurelia/templating#35 (https://github.com/aurelia/templating/issues/35).

textContent

You can bind an element's `textContent` property using the `textcontent` attribute:

```
<div textcontent.bind="stringProperty"></div>
<div textcontent="${stringProperty}"></div>
```

Two-way data-binding is supported with `contenteditable` elements:

```
<div textcontent.bind="stringProperty" contenteditable="true"></div>
```

style

You can bind a css string or object to an element's `style` attribute:

```
export class Foo {
  constructor() {
    this.styleString = 'color: red; background-color: blue';

    this.styleObject = {
      color: 'red',
      'background-color': 'blue'
    };
  }
}
```

```
<div style.bind="styleString"></div>
<div style.bind="styleObject"></div>
```

Use the `style` attribute's alias, `css` when doing string interpolation to ensure your application is compatible with Internet Explorer:

```
<!-- good: -->
<div css="width: ${width}px; height: ${height}px;"></div>

<!-- incompatible with Internet Explorer: -->
<div style="width: ${width}px; height: ${height}px;"></div>
```

Adaptive Binding

Aurelia has an adaptive binding system that chooses from a number of strategies when determining how to most efficiently observe changes. For more info on how this works checkout this post (http://blog.durandal.io/2015/04/03/aurelia-adaptive-

binding/). For the most part you don't need to think about these details however it does help to be aware of scenarios that lead to inefficient use of the binding system.

The #1 thing to be aware of is computed properties (properties with getter functions) are observed using dirty-checking. More efficient strategies such as Object.observe and property rewriting are not compatible with these types of properties.

In today's browser environment dirty-checking is a necessary evil. Very few browsers support Object.observe at the time of this writing. Aurelia's dirty-checking mechanism is similar to that used in Polymer (https://www.polymer-project.org/). It's very efficient and utilizes Aurelia's micro-task-queue to batch updates to the DOM.

A few bindings using dirty-checking will not cause performance problems in your application. Extensive use of dirty-checking may. Fortunately there's a way you can avoid dirty-checking simple computed properties. Consider the 'fullName' property in the example below:

```
export class Person {
  firstName = 'John';
  lastName = 'Doe';

  @computedFrom('firstName', 'lastName')
  get fullName(){
    return `${this.firstName} ${this.lastName}`;
  }
}
```

We've used the `@computedFrom` decorator to provide a hint to the Aurelia binding system. The binding system now knows to only check `fullName` for changes when `firstName` or `lastName` changes.

It's also important to be mindful of how dirty-checking works. When a property is "dirty-checked" the binding system periodically checks whether the property's current value matches the previously observed value for the property. By default this check happens every 120 milliseconds. This means your property's getter function has the potential to be called quite often which means it should be as efficient as possible. You should also avoid unnecessarily returning new instances of objects or arrays. Consider the following view:

```
<template>
  <label for="search">Search Issues:</label>
  <input id="search" type="text" value.bind="searchText" />
  <ul>
    <li repeat.for="issue of filteredIssues">${issue.abstract}</li>
  </ul>
</template>
```

Naive view model implementation:

```
export class IssueSearch {
  searchText = '';

  constructor(allIssues) {
    this.allIssues = allIssues;
  }

  // this returns a new array instance on every call which will in-turn result in a
  get filteredIssues() {
    if (this.searchText === '')
      return [];
    return this.allIssues.filter(x => x.abstract.indexOf(this.searchText) !== -1);
  }
}
```

Improved view model implementation:

```
export class IssueSearch {
  filteredIssues = [];
  _searchText = '';

  constructor(allIssues) {
    this.allIssues = allIssues;
  }

  get searchText() {
    return this._searchText;
  }
  set searchText(newValue) {
    this._searchText = newValue;
    if (newValue === '') {
      this.filteredIssues = [];
    } else {
      this.filteredIssues = this.allIssues.filter(x => x.abstract.indexOf(this.searc
    }
  }
}
```

HTML Extensions

In addition to databinding, you also have the power of Aurelia's HTML extensions. There are two types:

- Custom Elements - Extend HTML with new tags! Your custom elements can have their own views (which use databinding and other html extensions) and

optionally leverage ShadowDOM (http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/) (even if the browser doesn't support it).
- Custom Attributes - Extend HTML with new attributes which can be added to existing or custom elements. These attributes add new behavior to the elements.

Naturally, all of this works seamlessly with databinding. Let's look at the set of Custom Elements and Attributes that Aurelia provides for you and which are available globally in every view.

show

The `show` Custom Attribute allows you to conditionally display an HTML element. If the value of show is `true` the element will be displayed, otherwise it will be hidden. This attribute does not add/remove the element from the DOM, but only changes its visibility. Here's an example:

```
<div show.bind="isSaving" class="spinner"></div>
```

When the `isSaving` property is true, the `div` will be visible, otherwise it will be hidden.

if

The `if` Custom Attribute allows you to conditionally add/remove an HTML element. If the value is true, the element will also be present in the DOM, otherwise it will

not.

```
<div if.bind="isSaving" class="spinner"></div>
```

This example looks similar to that of `show` above. The difference is that if the binding expression evaluates to false, the `div` will be removed from the DOM, rather than just hidden.

If you need to conditionally add/remove a group of elements and you cannot place the `if` attribute on a parent element, then you can wrap those elements in a template tag which has the `if` attribute. Here's what that would look like:

```
<template if.bind="hasErrors">
    <i class="icon error"></i>
    ${errorMessage}
</template>
```

repeat

The `repeat` Custom Attribute allows you to render a template multiple times, once for each item in an array. Here's an example that renders out a list of customer names:

```
<ul>
    <li repeat.for="customer of customers">${customer.fullName}</li>
</ul>
```

An important note about the repeat attribute is that it works in conjunction with the `.for` binding command. This binding command interprets a special syntax in the form "item of collection" where "item" is the local name you will use in the template and "collection" is a normal binding expression that evaluates to an array or map.

Speaking of Maps, here's how you would bind to an ES6 Map:

```
<ul>
  <li repeat.for="[id, customer] of customers">${id} ${customer.fullName}</li>
</ul>
```

If instead of iterating over a collection you would rather iterate a specified number of times, you can instead use the syntax "i of count" where "i" is the index of the iteration and "count" is a binding expression that evaluates to an integer.

```
<ul>
  <li repeat.for="i of rating">*</li>
</ul>
```

> **Note:**: Like the `if` attribute, you can also use a `template` tag to group a collection of elements that don't have a parent element.

Each item that is being repeated by the `repeat` attribute has several special

contextual values available for binding:

- `$parent` - At present, the main view model's properties and methods are not visible from within the repeated item. We hope to remedy this in an update soon. For the mean time, you can access that view-model with `$parent`.
- `$index` - The index of the item in the array.
- `$first` - True if the item is the first item in the array.
- `$last` - True if the item is the last item in the array.
- `$even` - True if the item has an even numbered index.
- `$odd` - True if the item has an odd numbered index.

compose

The `compose` Custom Element enables you to dynamically render UI into the DOM. Imagine you have a heterogeneous array of items, but each has a type property which tells you what it is. You can then do something like this:

```
<template repeat.for="item of items">
    <compose
      model.bind="item"
      view-model="widgets/${item.type}">
    </compose>
</template>
```

Now, depending on the *type* of the item, the `compose` element will load a different view-model (and view) and render it into the DOM. If the view-model has an

`activate` method, the `compose` element will call it and pass in the `model` as a parameter. The `activate` method can even return a `Promise` to cause the composition process to wait until after some async work is done before actually databinding and rendering into the DOM.

The `compose` element also has a `view` attribute which can be used in the same way as `view-model` if you don't wish to leverage the standard view/view-model convention. If you specify a `view` but no `view-model` then the view will be databound to the surrounding context.

```
<template repeat.for="item of items">
    <compose view="my-view.html"></compose>
</template>
```

If you would like to databind a particular object when using only `view` rather than a full fledged view model (perhaps as part of a repeat), you may do so by binding the `view-model` directly. Now you will be able to use properties of that object directly in your `view`:

```
<template>
    <div repeat.for="item of items">
      <compose view="my-view.html" view-model.bind="item">
    </div>
</template>
```

What if you want to determine the view dynamically based on data though? or runtime conditions? You can do that too by implementing a `getViewStrategy()` method on your view-model. It can return a relative path to the view or an instance of a `ViewStrategy` for custom view loading behavior. The nice part is that this method is executed after the `activate` callback, so you have access to the model data when determining the view.

global-behavior

This is not an HTML enhancement that you will use directly. Rather, it works in conjunction with a custom binding command to dynamically enable the use of jQuery plugins and similar APIs declaratively in HTML. Let's look at an example in order to help clarify the idea:

```
<div jquery.modal="show: true; keyboard.bind: allowKeyboard">...</div>
```

This sample is based on the Bootstrap modal widget (http://getbootstrap.com /javascript/#modals). In this case, the `modal` jQuery widget will be attached to the `div` and it will be configured with its `show` option set to `true` and its `keyboard` option set to the value of the `allowKeyboard` property on the view-model. When the containing view is unbound, the jQuery widget will be destroyed.

This capability combines the special `global-behavior` with custom syntax to enable these dynamic capabilities. The syntax you see here is based on the syntax of the

native `style` attribute which lists properties and values separated in the same fashion as above. Note that you can use binding commands such as `.bind` to pass data from your view-model directly to the plugin or `.call` to pass a callback function directly to the plugin.

Here's how it works:

When the binding system sees a binding command that it doesn't recognize, it dynamically interprets it. The attribute name is mapped to a global binding handler which interprets the binding command. The handler can use the values to create an options object which it can pass to the plugin. When the view is unbound, the handler can also cleanup after itself. In this case the jQuery handler knows the pattern for instantiating plugins and using the `destroy` method to cleanup.

> **Note:** The `global-behavior` has a handlers list you must configure. It is only configured with jQuery by default. You can turn all of this off, if you desire, but it makes it easy to take advantage of basic jQuery plugins without any work on your part.

Routing

There are many different application styles you could be called upon to create. From navigation apps, to dashboards, to MDI interfaces, Aurelia can handle them all. In many of these cases a key component of your architecture is a client-side router, capable of translating url changes into application state.

If you've read the getting started guide, you know that there are two parts to routing. First, there's the `Router` which lives in your view-model. It's configured with route information and controls navigation. Then, there's the `router-view` which lives in the view and is responsible for displaying whatever the router identifies as the current state.

Let's look at an example configuration.

```
export class App {
  configureRouter(config, router){
    this.router = router;

    config.title = 'Aurelia';
    config.map([
      { route: ['', 'home'],       name: 'home',       moduleId: './home/index' },
      { route: 'users',            name: 'users',      moduleId: './users/index',
      { route: 'users/:id/detail', name: 'userDetail', moduleId: './users/detail' },
      { route: 'files*path',       name: 'files',      moduleId: './files/index',
    ]);
  }
}
```

We begin by implementing the `configureRouter` method. We can optionally set a `title` property to be used in constructing the document's title, but the most important part is setting up the routes. The router's `map` method takes a simple JSON data structure representing your route table. The two most important properties are `route` (a string or array of strings), which defines the route pattern, and `moduleId`, which has the *relative* module Id path to your view-model. You can also set a `name` property, to be used to generate a link to the route later, a `title` property, to be used when generating the document's title, a `nav` property indicating whether or not the route should be included in the navigation model (it can also be a number indicating order) and an `href` property which you can use to bind to in the *navigation model*.

> **Note:** Any properties that you leave off will be conventionally determined by the framework based on what you have provided.

So, what options do you have for the route pattern?

- static routes
  - ie 'home' - Matches the string exactly.
- parameterized routes
  - ie 'users/:id/detail' - Matches the string and then parses an `id` parameter. Your view-model's `activate` callback will be called with an

object that has an `id` property set to the value that was extracted from the url.

- wildcard routes
  - ie 'files*path' - Matches the string and then anything that follows it. Your view-model's `activate` callback will be called with an object that has a `path` property set to the wildcard's value.

All routes with a truthy `nav` property are assembled into a `navigation` array. This makes it really easy to use databinding to generate a menu structure. Another important property for binding is the `isNavigating` property. Here's some simple markup that shows what you might pair with the view-model shown above:

```
<template>
  <ul>
    <li class="loader" if.bind="router.isNavigating">
      <i class="fa fa-spinner fa-spin fa-2x"></i>
    </li>
    <li repeat.for="item of router.navigation">
      <a href.bind="item.href">${item.title}</a>
    </li>
  </ul>

  <router-view></router-view>
</template>
```

The Screen Activation Lifecycle

Whenever the router processes a navigation, it enforces a strict lifecycle on the view-models that it is navigating to and from. There are four stages in the lifecycle. You can opt-in to any of them by implementing the appropriate method on your view-model's class. Here's a list of the lifecycle callbacks:

- `canActivate(params, routeConfig, navigationInstruction)` - Implement this hook if you want to control whether or not your view-model *can be navigated to*. Return a boolean value, a promise for a boolean value, or a navigation command.
- `activate(params, routeConfig, navigationInstruction)` - Implement this hook if you want to perform custom logic just before your view-model is displayed. You can optionally return a promise to tell the router to wait to bind and attach the view until after you finish your work.
- `canDeactivate()` - Implement this hook if you want to control whether or not the router *can navigate away* from your view-model when moving to a new route. Return a boolean value, a promise for a boolean value, or a navigation command.
- `deactivate()` - Implement this hook if you want to perform custom logic when your view-model is being navigated away from. You can optionally return a promise to tell the router to wait until after your finish your work.

The `params` object will have a property for each parameter of the route that was parsed, as well as a property for each query string value. `routeConfig` will be the

original route configuration object that you set up. `routeConfig` will also have a new `navModel` property, which can be used to change the document title from data loaded by your view-model. For example:

```
activate(params, routeConfig) {
  this.userService.getUser(params.id)
    .then(user => {
      routeConfig.navModel.setTitle(user.name);
    });
}
```

> **Note:** A *Navigation Command* is any object with a `navigate(router)` method. When one is encountered, the navigation will be cancelled and control will be passed to the navigation command. One navigation command is provided out of the box: `Redirect`.

Child Routers

If you haven't read the "Get Started" guide, we recommend that you do that now and pay special attention to the section titled "Bonus: Leveraging Child Routers".

Whenever you set up a route to map to a view-model, that view-model can actually contain its own router...and when you set up routes with that...those view-models

can have their own routers...and so on. The route patterns are relative to the parent router and the module and view ids are relative to the view-model itself. This allows you to easily encapsulate features or child applications as well as handle complex hierarchical state.

A child router is just a router like any other. So, everything we've discussed above applies. To add a child router, just implement the `configureRouter` method again. The screen activation lifecycle discussed above applies to child routers as well. Each phase of the lifecycle is run against the entire router hierarchy before moving on to the next phase. The activate hooks run from top to bottom and the deactivate hooks run from bottom to top.

Conventional Routing

As with everything in Aurelia, we have strong support for conventions. So, you can actually choose to dynamically route rather than pre-configuring all your routes up front. Here's how you configure a router to do that:

```
export class App {
  configureRouter(config){
    config.mapUnknownRoutes(instruction => {
      //check instruction.fragment
      //set instruction.config.moduleId
    });
  }
}
```

All you have to do is set the `config.moduleId` property and you are good to go. You can also return a promise from `mapUnknownRoutes` in order to asynchronously determine the destination.

> **Note:** Though not necessarily related to conventional routing, you may sometimes have a need to asynchronously configure your router. For example, you may need to call a web service to get user permissions before setting up routes. To do this, return a promise from `configureRouter`.

Customizing the Navigation Pipeline

The router pipeline is composed out of separate steps that run in succession. Each of these steps has the ability to modify what happens during routing, or stop the

routing altogether. The pipeline also contains a few extensibility points where you can add your own steps. These are `authorize` and `modelbind`. `authorize` happens before `modelbind`. These extensions are called route filters.

The sample below shows how you can add authorization to your application:

```
import {Redirect} from 'aurelia-router';

export class App {
  configureRouter(config) {
    config.title = 'Aurelia';
    config.addPipelineStep('authorize', AuthorizeStep); // Add a route filter to the
    config.map([
      { route: ['welcome'],    name: 'welcome',       moduleId: 'welcome',      nav:
      { route: 'flickr',       name: 'flickr',        moduleId: 'flickr',       nav:
      { route: 'child-router', name: 'childRouter',   moduleId: 'child-router',  nav
      { route: '',              redirect: 'welcome' }
    ]);
  }
}

class AuthorizeStep {
  run(routingContext, next) {
    // Check if the route has an "auth" key
    // The reason for using `nextInstructions` is because
    // this includes child routes.
    if (routingContext.nextInstructions.some(i => i.config.auth)) {
      var isLoggedIn = /* insert magic here */false;
      if (!isLoggedIn) {
        return next.cancel(new Redirect('login'));
      }
    }

    return next();
  }
}
```

These extensibility points are in and of themselves small pipelines, and multiple steps can be added to each of them. For instance, if in addition to the `AuthorizeStep` above (which would just check that a user is logged in), you could add an `IsAdminStep` to the `authorize` extensibility point. They would then run in succession.

It's also possible to create your own named filters by simply passing a different name into `addPipelineStep`. This can be used like in the example below:

```
config.addPipelineStep('myname', MyFirstStep); // Transparently creates the pipeline
config.addPipelineStep('myname', MySecondStep); // Adds another step to it.
config.addPipelineStep('modelbind', 'myname'); // Makes the entire `myname` pipeline
```

Configuring PushState

If you'd prefer to get rid of the `#` (hashes) in your URLs, then you're going to have to enable `pushState` in your app. Good thing Aurelia supports that! You will also have to do some work on the server side to ensure it works properly. Let's start with the Aurelia side of the equation.

First you need to tell Aurelia in the `router config` that you want to use `pushState` like so:

```
export class App {
  configureRouter(config) {
    config.title = 'Aurelia';
    config.options.pushState = true; // <-- this is all you need here
    config.map([
      { route: ['welcome'],    name: 'welcome',     moduleId: 'welcome',      nav: t
      { route: 'flickr',       name: 'flickr',      moduleId: 'flickr',       nav: t
      { route: 'child-router', name: 'childRouter', moduleId: 'child-router', nav: t
      { route: '',             redirect: 'welcome' }
    ]);
  }
}
```

You will also want to add a base tag (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base) to the head of your html document. This is important, so don't leave it off.

Next, the server side needs to be configured to send back the same `index.html` file regardless of the request being made because all the routing is done client side. So, if you're using the `gulp watch` task with `browsersync` as per the navigation sample, then you can modify your setup like so:

From the console in the root of your project, run the following:

```
npm install --save connect-history-api-fallback
```

This will download and install the middleware plugin you need for this. Then open

up your *build/tasks* folder and locate the *serve* task. Open that and put this somewhere near the top with the other require statements:

```
var historyApiFallback = require('connect-history-api-fallback')
```

Lower down you can modify the `serve` task to use the new `middleware`:

```
gulp.task('serve', ['build'], function(done) {
  browserSync({
    open: false,
    port: 9000,
    server: {
      baseDir: ['.'],
      middleware: [historyApiFallback(), function (req, res, next) { // it's the firs
        res.setHeader('Access-Control-Allow-Origin', '*');
        next();
      }]
    }
  }, done);
});
```

Now your node server should behave itself and let Aurelia deal with the routing.

If you're using a .NET server side framework such as ASP.NET MVC then config is as follows:

- Create a Controller and call it ApplicationController or what ever you want to call it. It should look something like this:

```
public class ApplicationController : Controller {
  public ActionResult Index() {
    return View();
  }
}
```

- Create an "index.cshtml" view in your Views folder.
- Setup your routing configuration like this:

```
context.MapRoute(
  name: "AureliaRouting",
  url: "{*.}",
  defaults: new { controller = "Application", action = "Index" }
);
```

Note that with the above you will be forced to use a Razor view file. If you want to use a regular HTML file, there are different ways to do it. This SO article might help you (http://stackoverflow.com/questions/20871938/render-html-file-in-asp-net-mvc-view).

If you are using Nancy FX (http://nancyfx.org/), then the config is just as simple. Locate your `IndexModule.cs` or whatever you called it and make sure it looks something like this and all will be well:

```
public class IndexModule : NancyModule {
  public IndexModule()     {
    this.Get["/robots.txt"] = p => this.Response.AsFile("robots.txt");
    this.Get["/sitemap.xml"] = p => this.Response.AsFile("sitemap.xml");
    this.Get["/"] = x => this.View["index"];
    this.Get["/{path*}"] = x => this.View["index"];
  }
}
```

Similar techniques can be used in other server environments - you just need to make sure that whatever server you're using, it needs to send back the same `index.html` regardless of the request being made. All server side frameworks should be able to achieve this. Aurelia will figure out which page to load based on its own route data.

Reusing an existing VM

Sometimes you might want to use the same VM for multiple routes. By default Aurelia will see those routes as aliases to the same VM and thus only perform the build and attach process as well as the complete life-cycle once. This might not be exactly what you are looking for. Take the following router example:

```
export class App {
  configureRouter(config) {
    config.title = 'Aurelia';
    config.map([
      { route: 'product/a',    moduleId: './product',     nav: true },
      { route: 'product/b',    moduleId: './product',     nav: true },
    ]);
  }
}
```

Since the VM's life-cycle is called only once you may have problems to recognize that the user switched the route from `Product A` to `Product B`.

To work around this issue implement the method `determineActivationStrategy` in your VM and return hints for the router about what you'd like to happen. E.g in order to force a rebuild of the VM implement it like this:

```
import {activationStrategy} from 'aurelia-router';

export class YourViewModel {
  determineActivationStrategy(){
    return activationStrategy.replace;
  }
}
```

If you just want to force a refresh of the life-cycle (useful with `<compose>` bindings) you may do something like the following:

```
import {activationStrategy} from 'aurelia-router';

export class YourViewModel {
  determineActivationStrategy(){
    return activationStrategy.invokeLifecycle;
  }
}
```

> **Note:** Keep in mind that by forcing refreshes, Aurelia has to rebuild the complete VM. As for performance reasons a simple observer on the `router.currentInstruction` might be sufficient for scenarios where you'd simply like to exchange some data.

Rendering multiple ViewPorts

Sometimes you need to render content in more than one area of the page. Aurelia's router lets you specify multiple `router-view`s to be activated by a single route. First, add named `router-view` elements to your view: `markup <template> <div class="page-host"> <router-view name="left"></router-view> </div> <div class="page-host"> <router-view name="right"></router-view> </div> </template>` Then in your route config, specify which modules should be activated

for each named `router-view`. `javascript configureRouter(config){ config.map([{ route: 'edit', viewPorts: { left: { moduleId: './editor' }, right: { moduleId: './preview' } } }]); }` If you don't name a `router-view`, it will be available under the name `'default'`.

Generating route URLs

If you need a to create a URL that matches an existing route, the router can generate one for you.

```
router.generate('userDetail', { id: 123 });
```

The first parameter is the route name, as specified in the route config. The second parameter is an object with route parameters to fill in to the route template. Any properties of the object that don't map to route parameters will be automatically appended to the query string.

If you want to navigate to the generated URL, use `router.navigateToRoute('userDetail', { id: 123 })`.

If you simply want to render an anchor in your view, you can use the `route-href` custom attribute.

```
<a route-href="route: userDetail; params.bind: { id: user.id }">${user.name}</a>
```

Extending HTML

Aurelia has a powerful and extensible HTML template compiler. The compiler itself is just an algorithm for interacting with these extensions. Out of the box, Aurelia provides two extensions: *Custom Elements* and *Custom Attributes*.

These extensions are not visible to the compiler by default. There are three main ways to plug them in:

- Use the `require` element to require an extension in a view. The `from` attribute specifies the relative path to the extension's module. The extension will be locally defined.
- Use the Aurelia object during your bootstrapping phase to call `.globalizeResources(...resourcePaths)` to register extensions with global visibility in your application.
- Install a plugin that registers extensions with global visibility in your application.

> **Note:** A recommended practice for your own apps is to place all your app-specific extensions, value converters, etc. into a *resources* folder. Then create an *index.js* file that turns them all into an internal plugin. Finally, install that plugin during your app's bootstrapping phase. This will keep your resources located in a known location, along with their registration code. It will also keep your configuration file clean and simple.

All extensions can opt into the view lifecycle by implementing any of the following hooks:

- `bind(bindingContext)` - Invoked when the databinding engine binds the view. The binding context is the instance that the view is databound to.
- `unbind()` - Invoked when the databinding engine unbinds the view.
- `attached()` - Invoked when the view that contains the extension is attached to the DOM.
- `detached()` - Invoked when the view that contains the extension is detached from the DOM.

> **Note:** If you choose to implement the `bind` callback, the initial binding of your extension will flow a little differently. Usually, if you have callbacks for your extension's bindable properties, these are each individually called during the bind phase. However, if you add the `bind` callback, they will not be called during initialization. Rather, the `bind` callback will be called once all properties have their initial bound values set. This is an important and useful characteristic, particularly for complex extensions which may not want to "act" until they have all evaluated values available.

Custom Attributes

Custom Attributes extend HTML with functionality by adding new behavior to existing HTML elements. Common uses for Custom Attributes include:

- Wrapping jQuery and similar plugins (when the `global-behavior` is insufficient).
- Shortcuts for common style, class or attribute bindings.
- Just about anything that needs to change an existing HTML element or even a Custom Element which you cannot directly alter.

Custom Attributes tend to represent cross-cutting concerns. For example you might create a custom tooltip attribute that you can then attach to any element. This is a better idea than building tooltip functionality directly into every custom element you create.

Let's look at one of Aurelia's own Custom Attribute implementations: `show`. Here's how it is used:

```
<div show.bind="isSaving" class="spinner"></div>
```

The `show` attribute will conditionally apply a css class to an element based on the falseness of its value. (The css class, when applied, hides the element.) Here's the implementation:

```
import {inject, customAttribute} from 'aurelia-framework';

@customAttribute('show')
@inject(Element)
export class Show {
  constructor(element) {
    this.element = element;
  }

  valueChanged(newValue){
    if (newValue) {
      this.element.classList.remove('aurelia-hide');
    } else {
      this.element.classList.add('aurelia-hide');
    }
  }
}
```

The first thing to note is that Custom Attributes are classes and follow the same patterns we've already seen. Notice that the decorators play an important role in defining the attribute. Here's what they are doing:

- `@customAttribute('show')` - Indicates that this class is a custom attribute so the HTML compiler knows how this class "plugs in". It will be recognized by the compiler any time it sees an attribute named "show".
- `inject` - This is part of the dependency injection system; the same as you've seen before. Custom Attributes can have the Element they are placed on

injected into the constructor. That is what is happening here. All you have to do use use the browser's `Element` type to indicate that.

There are a few other interesting things that happen too.

- Attributes in html have a value. So, your Custom Attribute class will have a `value` property that is kept in sync with the HTML. If you implement a `valueChanged` method, it will be invoked any time the attribute's value changes. The first argument will be the new value and the second will be the old value.

What about conventions?

If your class's export name matches the pattern {SomeName}CustomAttribute, then you don't need to include the `@customAttribute` decorator at all. The attribute name will be inferred from the export name by stripping off "CustomAttribute" and lowercasing and hyphenating the remaining part of the name. ie. some-name

These conventions mean that we can actually define our `show` attribute like this:

```
export class ShowCustomAttribute {
  static inject = [Element]; //showing non-decorator method here for variety

  constructor(element) {
    this.element = element;
  }

  valueChanged(newValue){
    if (newValue) {
      this.element.classList.remove('aurelia-hide');
    } else {
      this.element.classList.add('aurelia-hide');
    }
  }
}
```

> **Note:** So, why doesn't Aurelia itself leverage these conventions internally? Any time you are creating a 3rd party library, it's best to be explicit. You don't know whether or not developers consuming your library will have changed Aurelia's conventions, thus breaking your library. In order to prevent this, always be explicit by using decorators. Inside your own apps though, you can use the conventions all you want to simplify development.

Options Attributes

You may be wondering what to do if you want to create a Custom Attribute with multiple properties, since attributes usually map to a single value. It's actually quite simple. Just create several `bindable` properties:

```
import {customAttribute, bindable} from 'aurelia-framework';

@customAttribute('my-attribute')
export class MyAttribute {
  @bindable foo;
  @bindable bar;
}
```

This creates a Custom Attribute named `my-behavior` with two properties `foo` and `bar`. Each of these properties are available directly on the class. Each can have optional change callbacks, `fooChanged` and `barChanged` respectively. However, they are configured in HTML a bit different. Here's how that would be done:

```
<div my-attribute="foo: some literal value; bar.bind: some.expression"></div>
```

Notice that we don't use a binding command on the attribute itself. Instead, we can use them on each individual property inside the attribute's value. You can use literals as well as the standard binding commands.

> **Note:** You don't use `delegate` or `trigger` commands inside an options attribute. Those are always attached to the element itself, since they work directly with native DOM events. However, you can use `call`.

If you aren't using ES7 property initializers, you can put the `@bindable` decorator directly on the class. Just be sure to provide the property name like this `@bindable('propertyName')`. To specify more details for a bindable property, you should pass an options object instead like this:

```
@bindable({
  name:'myProperty', //name of the property on the class
  attribute:'my-property', //name of the attribute in HTML
  changeHandler:'myPropertyChanged', //name of the method to invoke when the property
  defaultBindingMode: bindingMode.oneWay, //default binding mode used with the .bind
  defaultValue: undefined //default value of the property, if not bound or set in HTML
})
```

The defaults and conventions are shown above. So, you would only need to specify these options if you need to deviate. R

> **Note:** There is also a special `@dynamicOptions` decorator. This allows a custom attribute to have a dynamic set of properties which are all mapped from the options attribute syntax into the class at runtime. Don't declare `bindable` properties. Simply add a single `@dynamicOptions` decorator and anything the consumer lists in the options attribute syntax will be mapped.
>
> **Note:** Remember that all decorators are available on the `Decorators` helper and can be specified with a static `decorators` property or method if you prefer (or if you are using a language that doesn't support decorators). See the CoffeeScript examples above for details.

Template Controllers

Custom Attributes can indicate that they are a Template Controller with the `@templateController` decorator. This indicates that they convert DOM into an inert HTML template. The custom attribute class can then decide when and where (or how many times) to instantiate the template in the DOM. Examples of this are the `if` and `repeat` attributes. Simply place one of these on a DOM node and it becomes a template, controlled by the Custom Attribute class.

Let's take a look at the implementation of the `if` Custom Attribute to see how one of these is put together. Here's the full source code:

```
import {BoundViewFactory, ViewSlot, customAttribute, templateController, inject} fro

@customAttribute('if')
@templateController
@inject(BoundViewFactory, ViewSlot)
export class If {
  constructor(viewFactory, viewSlot){
    this.viewFactory = viewFactory;
    this.viewSlot = viewSlot;
    this.showing = false;
  }

  valueChanged(newValue){
    if (!newValue) {
      if(this.view){
        this.viewSlot.remove(this.view);
        this.view.unbind();
      }

      this.showing = false;
      return;
    }

    if(!this.view){
      this.view = this.viewFactory.create();
    }

    if (!this.showing) {
      this.showing = true;
```

```
      if(!this.view.bound){
        this.view.bind();
      }

      this.viewSlot.add(this.view);
    }
  }
}
```

Before we dig into the unique aspects, let me remind you of what you see here that is similar. First, we have a simple class with decorators. It also has a single `value` property by default which can be observed by adding a `valueChanged` callback.

Ok, what's different? Take a look at the constructor. We have two unique items being injected: `BoundViewFactory` and `ViewSlot`.

The `BoundViewFactory` is capable of generating instances of the the HTML template that the attribute is attached to. No need to worry about compiling, etc. That's taken care of for you. Why is it called "Bound" View Factory though? Well, it's already referencing the parent binding context. It's "bound" in a sense. So, if you call its `create` method it will instantiate a new View from the template which will be bound to that context. This is what you want with an `if` attribute. It's not what you want with a `repeat` attribute. In that case, each time you call `create` you want a view bound to a particular array item. To achieve this, simply pass any object you want the view to be bound against into the `create` method.

The `ViewSlot` represents the slot or location within the DOM that the template was extracted from. This is usually the location that you want to add View instances to.

> **Note**: Unlike previous attributes, a template controller works more directly with the *primitives* of the framework. Views, ViewFactories and ViewSlots are all low level parts of the templating engine.

Take a close look at the `valueChanged` callback. Here you can see where the `if` attribute is creating the view and adding it to the slot, based on the truthiness of the value. There are a few important details of this:

- The attribute always calls `bind` on the View *before* adding it to the ViewSlot. This ensures that all internal bindings are initially evaluated outside of the live DOM. This is important for performance.
- Similarly, always call `unbind` *after* removing the View from the DOM.
- After the View is initially created, the `if` attribute does not throw it away even when the value becomes false. It caches the instance. Aurelia can re-use Views and even re-target them at different binding contexts. Again, this is important for performance, since it eliminates needless re-creation of Views.

Custom Elements

Custom Elements add new tags to your HTML markup. Each Custom Element can have its own view template which can be rendered into the Light DOM or the Shadow DOM. Custom Elements can also have any number of properties which they surface as attributes in HTML for databinding support and which they can databind to inside their view template.

Why don't we create a simple custom element so that we can see how that works? We'll make an element that says hello to someone, called `say-hello`. Here's how we want to be able to use it when we're done:

```
<template>
    <require from="./say-hello"></require>

    <input type="text" ref="name">
    <say-hello to.bind="name.value"></say-hello>
</template>
```

So, how do we build this? Well, we're going to start with a class, just like we did with the Custom Attribute. Here's what it looks like:

say-hello.js

```
import {customElement, bindable} from 'aurelia-framework';

@customElement('say-hello')
export class SayHello {
  @bindable to;

  speak(){
    alert(`Hello ${this.to}!`);
  }
}
```

If you read the section on Custom Attributes, then you know what this does. There's some conventions too, which means we can do this if we want:

say-hello.js (with conventions)

```
import {bindable} from 'aurelia-framework';

export class SayHelloCustomElement {
  @bindable to;

  speak(){
    alert(`Hello ${this.to}!`);
  }
}
```

Be default, Custom Elements have a view. Here's the view for ours:

say-hello.html

```
<template>
    <button click.trigger="speak()">Say Hello To ${to}</button>
</template>
```

As you can see, we've got access to our class's properties and methods. It's important to note that you don't need to declare `@bindable` properties for every property you want to bind to in your template. You only need to declare it for properties you want to exist as attributes on your custom element.

That's really all there is to it. You follow the same view-model/view naming conventions and all the same patterns for custom elements. There are a few unique decorators for custom elements you may also need:

- `@syncChildren(property, changeHandler, selector)` - Creates an array property on your class that has its items automatically synchronized based on a query selector against its view.
- `@skipContentProcessing` - Tells the compiler not to process the content of your custom element. It is expected that you will do custom processing yourself.
- `@useView(path)` - Specifies a different view to use.
- `@noView` - Indicates that this custom element does not have a view and that the author intends for the element to handle its own rendering internally.

Template Parts

Template part replacement in custom elements allows a custom element to specify certain parts of its view which can be replaced with alternate markup at runtime on a per-instance basis.

If you are using a custom element you can mark any part of it’s view as `replaceable`. Then the consumer of your element can specify a template in the element's content indicating the part they want it to replace in the element’s view. Use `part="someName"` to identify a part of the template that is replaceable. If it’s not a template for a template controller (repeat or if) then you also need the `replaceable` attribute on the part. Finally, when the consumer wants to replace that part, they add `replace-part="someName"` on a template inside the elements' content to provide the alternate version.

Here's an example that shows how to make the template inside of a repeater replaceable without affecting the `li` container. It also shows how to create the custom element so that the runtime binding context where the custom element is used can be reached by the replaced template.

example.js

```
export class Example {
  constructor(){
    this.items = [1,2,3,4,5];
  }

  bind(context){
    this.$parent = context;
  }
}
```

example.html

```
<template>
  <ul>
    <li class="foo" repeat.for="item of items">
      <template replaceable part="item-template">
        Original: ${item}
      </template>
    </li>
  <ul>
</template>
```

welcome.js

```
export class Welcome{
  heading = 'Welcome to the Aurelia Navigation App!';
  firstName = 'John';
  lastName = 'Doe';

  get fullName(){
    return `${this.firstName} ${this.lastName}`;
  }

  welcome(){
    alert(`Welcome, ${this.fullName}!`);
  }
}
```

welcome.html

```
<template>
  <require from="./demo"></require>

  <demo>
    <template replace-part="item-template">
      Replacement: ${item} ${$parent.$parent.fullName} <button click.delegate="$pare
    </template>
  </demo>
</template>
```

Eventing

Eventing is a powerful tool when you need decoupled components of your application to talk to one another. Aurelia supports both standard DOM events as well as more application-specific events via the `EventAggregator`.

DOM Events

DOM events should be used when UI-specific messages need to be sent. They should not be used for application-specific messages. Aurelia doesn't add any functionality beyond the DOM for UI events (yet). Any Custom Attribute or Element can have its associated `Element` injected into its constructor. You can then use the `Element` to trigger events. To learn more about creating and triggering custom DOM events, please read this article (https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events).

The Event Aggregator

If you need loosely coupled application-events, you want to use the `EventAggregator`. Its streamlined pub/sub interface makes it ideal for a wide range of messaging scenarios.

The Event Aggregator can publish events to a message channel or it can publish strongly-typed messages. Let's look at publishing to channels first:

```
import {EventAggregator} from 'aurelia-event-aggregator';

export class APublisher{
    static inject = [EventAggregator];
    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
    }

    publish(){
        var payload = {}; //any object
        this.eventAggregator.publish('channel name here', payload);
    }
}
```

We begin by having the DI provide us with the singleton Event Aggregator. Next we call its `publish` method, passing it the message channel name and the data payload to send on that channel. Here's how a subscriber would set themself up to receive this:

```
import {EventAggregator} from 'aurelia-event-aggregator';

export class ASubscriber{
    static inject = [EventAggregator];
    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
    }

    subscribe(){
        this.eventAggregator.subscribe('channel name here', payload => {
            //do something with the payload here
        });
    }
}
```

As you can see, they use the same channel name, but provide a callback, which will be invoked for every message sent on the channel.

Alternatively, you can publish and subscribe to strongly-typed messages. Here's an example publisher:

```
export class SomeMessage{ }
```

```
import {EventAggregator} from 'aurelia-event-aggregator';
import {SomeMessage} from './some-message';

export class APublisher{
    static inject = [EventAggregator];
    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
    }

    publish(){
        this.eventAggregator.publish(new SomeMessage());
    }
}
```

In this case, we publish an instance of a particular message type. Here's a sample subscriber:

```
import {EventAggregator} from 'aurelia-event-aggregator';
import {SomeMessage} from './some-message';

export class ASubscriber{
    static inject = [EventAggregator];
    constructor(eventAggregator){
        this.eventAggregator = eventAggregator;
    }

    subscribe(){
        this.eventAggregator.subscribe(SomeMessage, message => {
            //do something with the message here
        });
    }
}
```

The subscriber will be called any time an instance of `SomeMessage` is published. Subscription is polymorphic, so if a subclass of SomeMessage is published, this subscriber will be notified as well.

> **Note:** All forms of the `subscribe` method return a *dispose function*. You can call this function to dispose of the subscription and discontinue receiving messages. A good place to dispose is either in a view-model's `deactivate` callback if it is managed by a router, or in its `detached` callback if it is any other view-model.

HTTP Client

As a convenience, Aurelia includes a basic `HttpClient` to provide a comfortable interface to the browser's `XMLHttpRequest` object. `HttpClient` is not included in the modules that Aurelia's bootstrapper installs, since it's completely optional and many apps may choose to use a different strategy for data retrieval. So, if you want to use it, first you must install it with the following command:

```
jspm install aurelia-http-client
```

Then you can use it like this:

```
import {HttpClient} from 'aurelia-http-client';

export class WebAPI {
    static inject = [HttpClient];
    constructor(http){
        this.http = http;
    }

    getAllContacts(){
        return this.http.get('url goes here');
    }
}
```

The `HttpClient` has the following implementation:

```
export class HttpClient {
  configure(fn){
    var builder = new RequestBuilder(this);
    fn(builder);
    this.requestTransformers = builder.transformers;
    return this;
  }

  createRequest(url){
    let builder = new RequestBuilder(this);

    if(url) {
      builder.withUrl(url);
    }

    return builder;
  }

  delete(url){
    return this.createRequest(url).asDelete().send();
  }

  get(url){
    return this.createRequest(url).asGet().send();
  }

  head(url){
    return this.createRequest(url).asHead().send();
  }
```

```
  jsonp(url, callbackParameterName='jsoncallback'){
    return this.createRequest(url).asJsonp(callbackParameterName).send();
  }

  options(url){
    return this.createRequest(url).asOptions().send();
  }

  put(url, content){
    return this.createRequest(url).asPut().withContent(content).send();
  }

  patch(url, content){
    return this.createRequest(url).asPatch().withContent(content).send();
  }

  post(url, content){
    return this.createRequest(url).asPost().withContent(content).send();
  }
}
```

As you can see, it provides convenience methods for all the standard verbs as well as `jsonp`. Each of these methods sends an `HttpRequestMessage` except `jsonp` which sends a `JSONPRequestMessage`. The result of sending a message is a `Promise` for an `HttpResponseMessage`.

The `HttpResponseMessage` has the following properties:

- `response` - Returns the raw content sent from the server.

- `responseType` - The expected response type.
- `content` - Formats the raw `response` content based on the `responseType` and returns it.
- `headers` - Returns a `Headers` object with the parsed header data.
- `statusCode` - The server's response status code.
- `statusText` - The server's textual status message.
- `isSuccess` - Indicates whether or not the status code falls within the success range.
- `reviver` - A function used to transform the raw `response` content.
- `requestMessage` - A reference to the original request message.

> **Note:** By default, the `HttpClient` assumes you are expecting a JSON responseType.

Interceptors

It is possible to hook into requests and responses with interceptors.

```
class RequestInterceptor {
  request(message) {
    // do something with the message
    return message;
  }

  requestError(error) {
    throw error; // or return a (Http/Jsonp)RequestMessage to recover from the error
  }
}

class ResponseInterceptor {
  response(message) {
    // do something with the message
    return message;
  }

  responseError(error) {
    throw error; // or return an HttpResponseMessage to recover from the error
  }
}

var client = new HttpClient();
  .configure(x => {
    x.withInterceptor(new RequestInterceptor());
    x.withInterceptor(new ResponseInterceptor());
  });i
```

> **Note:** It is important to realise that all interceptors used with a client form a chain. The return value of an intercept method is passed on as the arugment to the next. Interceptors are called in the order they were added.

There are two other apis that are worth noting. You can use `configure` to access a fluent api for configuring all requests sent by the client. You can also use `createRequest` to custom configure individual requests. Here's an example of configuration:

```
var client = new HttpClient()
  .configure(x => {
    x.withBaseUrl('http://aurelia.io');
    x.withHeader('Authorization', 'bearer 123');
  });

client.get('some/cool/path');
```

In this case, all requests from the client will have the baseUrl of 'http://aurelia.io (http://aurelia.io/)' and will have the specified Authorization header. The same API is available via the request builder. So, you can accomplish the same thing on an individual request like this:

```
var client = new HttpClient();

client.createRequest('some/cool/path')
  .asGet()
  .withBaseUrl('http://aurelia.io')
  .withHeader('Authorization', 'bearer 123')
  .send();
```

The fluent API has the following chainable methods: `asDelete()`, `asGet()`, `asHead()`, `asOptions()`, `asPatch()`, `asPost()`, `asPut()`, `asJsonp()`, `withUrl()`, `withBaseUrl()`, `withContent()`, `withParams()`, `withResponseType()`, `withTimeout()`, `withHeader()`, `withCredentials()`, `withReviver()`, `withReplacer()`, `withProgressCallback()`, and `withCallbackParameterName()`.

Customization

View and View-Model Conventions

How are views and view-models linked? Our simple convention is based on module id. If you've got a view-model with id (essentially path) './foo/bar/baz' then that will map to `./foo/bar/baz.js` and `./foo/bar/baz.html` by default. Suppose you want to follow a different convention though. What if all your view-models live in a `view-models` folder and you want their views to live in a `views` folder? How would

you do that? In order to do this, you want to change the behavior of the Conventional View Strategy. Here's how you do it:

```
import {ConventionalViewStrategy} from 'aurelia-framework';

ConventionalViewStrategy.convertModuleIdToViewUrl = function(moduleId){
  return moduleId.replace('view-models', 'views') + '.html';
}
```

You should execute this code as part of your bootstrapping configuration logic so that it takes effect before any Custom Elements are loaded. This will affect *everything* including custom elements. So, if you need or want those to act differently, you will need to account for that in your implementation of `convertModuleIdToViewUrl`.

> **Note:** This is an example of why 3rd party plugin authors should not rely on conventions. Developers may change these conventions in order to fit the needs of their own app.