
{ }
AURELIA
FOR REAL WORLD WEB APPLICATIONS
DWAYNE CHARRINGTON

Aurelia For Real World Web Applications

Using the Aurelia Javascript framework to build real world applications.

Dwayne Charrington

This book is for sale at http://leanpub.com/aurelia-for-real-world-applications

This version was published on 2018-12-03

Tweet This Book!

Please help Dwayne Charrington by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Aurelia For Real World Applications by @AbolitionOf

The suggested hashtag for this book is #rwaurelia.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#rwaurelia

This book is dedicated to my beautiful wife Marie and our son Daedalus. Thank you for putting up with my long hours and nights when I come to bed late. A special thank you to my employer Fathom for allowing me to work with Aurelia on a daily basis.

Also, a shout out to various members of the Aurelia core team, Rob Eisenberg (the man behind Aurelia), Jeremy Danyow, Patrick Walters and others who have helped answer questions and also the community.

Contents

1. The Fundamentals of Aurelia

Before we journey into Aurelia land and start building things, we need to talk about a few basic fundamentals that are key to almost every Aurelia application you will write. The theory before the practical so-to-speak.

Thankfully Rob Eisenberg and his carefully assembled team had a vision to abide by official specifications where possible, which has resulted in a framework that is easy to learn. The key things you need to understand are just basic modern Javascript and HTML. However, much like any other framework or library there are conventions that you need to learn.

The first thing you need to be aware of is Aurelia works on the premise of both views and view-models. If you come from a MVVM/MVC/MVwhatever background, then you will already be familiar with the concept of a view and view-model.

Most of what you end up doing in Aurelia will at the very least in most cases consist of a view-model (a Javascript class). Sometimes you might have a view, sometimes you might only have a view and no view-model. There are no strict rules how things must be done, but there are certain constraints in how you do things, for your own sanity mostly.

Structure is important, so Aurelia's modular component-first approach is something you will really enjoy if you come from a framework or library that has no structure or promotes a really convoluted way of working.

The CLI (which you'll be using shortly) assumes your code will live inside of a directory called `src` you can change this, but we aren't going to change it as `src` makes perfect sense, it's just considered a common best practice when developing a SPA with Aurelia.

Terminology

Throughout the book you will see terms such as; view, view-model, dependency injection and more. This section aims to explain what these terms are in a broad overview.

Later on in further chapters, you will learn more in greater detail about each of these concepts.

View
: A view is a HTML template. All content is contained within opening and closing `<template>` tags. By default custom elements and routed components unless otherwise configured are assumed by Aurelia to have an accompanying view.

View-model
: Most of what you do in building an Aurelia application, I would say about 95% is based on the premise of view-model classes. A view-model is merely a Javascript class, which we learned about in chapter two.

Dependency Injection
: This is what underpins Aurelia as a whole. Whenever you import, inject and export something, behind the scenes Aurelia has a dependency injection container which is responsible for ensuring things are imported where and how they should be. You will learn about dependency injection later on.

Custom Attribute
: A custom attribute is comprised of a view-model Javascript class which allows you to decorate existing elements (including custom elements). A custom attribute does not have an associated view.

Custom Element
: Pretty similar to a custom attribute, a custom element is comprised of a view-model Javascript class and unless otherwise configured, a matching view template of the same filename with a `.html` file extension.

The component life-cycle

In Aurelia each component goes through an event life-cycle where a series of callback functions are called at different times depending what is happening. This applies to not only Aurelia's internal classes, but any component that you create such as a custom element or even just a generic view-model goes through this process.

constructor(...injectables)
: The class constructor for your view-model. This is where you tell your view-model about any injected dependencies. The injected dependencies are parameters on your constructor function. Technically this is not an Aurelia method, a constructor is a native method for every class.

created(owningView: View, myView: View)
: This method is called immediately after both the view and view-model has been created. We have direct access to the view instance via the view parameter passed through to this function.

bind(bindingContext: Object, overrideContext: Object)
: This method is called after the data-binding engine has bound to the view. The binding context supplied to this function is the instance that the view is data-bound to.

unbind()
: The inverse of bind. When the data-binding engine unbinds the view, this method is called.

attached()
: In standard Javascript you are probably familiar with `DOMContentLoaded` or if you use jQuery, then `$(document).ready()` which is a callback to say the DOM is now ready for you to interact with. This method is called when the view has been attached to the DOM. Inside of this method is where you will mutate DOM elements, attach DOM events and instantiate jQuery plugins (which we will get too later on).

detached()
: This is the inverse of the `attached()` method. When the view is detached from the DOM, this

callback hook is called. Perfect for cleaning up any events you might have registered in the DOM and de-registering jQuery plugins.

Really when it comes down to a real world application, you will find yourself working with only a few of the above mentioned callback life-cycle hook methods. However, it is convenient to know when each function is called and what order they are called in.

Worth mentioning...

Each of the above callbacks is optional. If you do not define one or more of these, your application will work as intended. Only implement what meets your needs, but don't feel like you have to have them. As a rule of thumb if you implement `bind` you will also need to implement `unbind`. The same thing applies for `attached` and `detached` however, it isn't required.

In the chapter on routing, you will also learn about the screen activation life-cycle which works similarly to the component life-cycle, however only strictly applies to view-models instantiated by the router.

Conventions

By default Aurelia makes some assumptions about how things should work, unless you tell it otherwise. These conventions allow you to write code with out needing to configure anything upfront.

Some developers are not a fan of magic in frameworks and rather explicitly defining everything upfront, which is why Aurelia gives you the choice of both. While component libraries like Vue also technically have conventions, you need to implement a lot more out-of-the-box than you do in Aurelia.

In my experience working with Aurelia, the conventions are usually what you want. Only specific use-cases have required changing some aspect of one or more of Aurelia's default conventions.

View-model/View Pairs

In many cases, Aurelia makes the assumption when you have a view-model, you also have a view of the same name, just with a `.html` file extension. The situations in which Aurelia assumes you have a matching view are:

- Routed view-models: When defining a route that points to a view-model, by default Aurelia will also expect a view file of the same name. If you route to `src/welcome.js`, Aurelia will look for a file `src/welcome.html` in the same location as the view-model.

- Custom elements: When creating a custom element, it is assumed just like routed view-models, a view template of the same name and `.html` extension exists.
- Dynamic composition with `<compose>`: When dynamically composing a view-model, if you do not specify a view and the view-model being composed does not specify a particular view or no view at all, a matching `.html` file will attempt to be loaded.

If for whatever reason your view-model has no view (on purpose) using the `@noView` class decorator you can tell Aurelia that it shouldn't try looking for a view template.

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

@noView()
export class MyViewModel {

}
```

Overriding view/view-model conventions

If you wanted to specify a view be loaded from somewhere different or an entirely different name, there are two ways you can do this.

Using `@useView(viewString)`:

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

@useView('somelocation/my-view.html')
export class MyViewModel {

}
```

Using `getViewStrategy()`:

```
export class MyViewModel {
    // This method is fired upon view-model instantiation
    getViewStrategy() {
        return 'somelocation/my-view.html';
    }
}
```

Custom Attributes/Custom Elements/Value Converters

In Aurelia you will be working with custom attributes, elements and value converters from time to time. A naming convention employed by Aurelia means when you create your resources, you can tell Aurelia whether they are a particular resource type either using a naming convention on the view-model or through the use of a decorator.

MyResourceCustomAttribute
: Using the default naming convention, the `CustomAttribute` part of your view-model class is stripped out and the part before it becomes the name of the custom attribute, lowercased and hyphen separated. The example provided will create an attribute called `my-resource`.

MyResourceCustomElement
: Using the default naming convention, the `CustomElement` part of your view-model class is stripped out and the part before it becomes the name of the custom element. The example provided will create an element called `<my-resource></my-resource>`.

MyResourceValueConverter
: Using the default naming convention, the `ValueConverter` part of your view-model class is stripped out and the part before it becomes the name of the value converter. The example provided will create a value converter called `myResource`.

If you're not a fan of the naming convention, you can also use a decorator provided to decorate your view-model as a particular type of resource.

- `@customAttribute('my-resource')`
- `@customElement('my-resource')`
- `@valueConverter('my-resource')`

As discussed previously, decorators are used to decorate your view-model class. So put this above the class name and make sure you aren't also doubling-up and using both approaches, either use the naming convention or decorator.

Later on in their respective chapters, you will be creating custom attributes, elements and value converters.