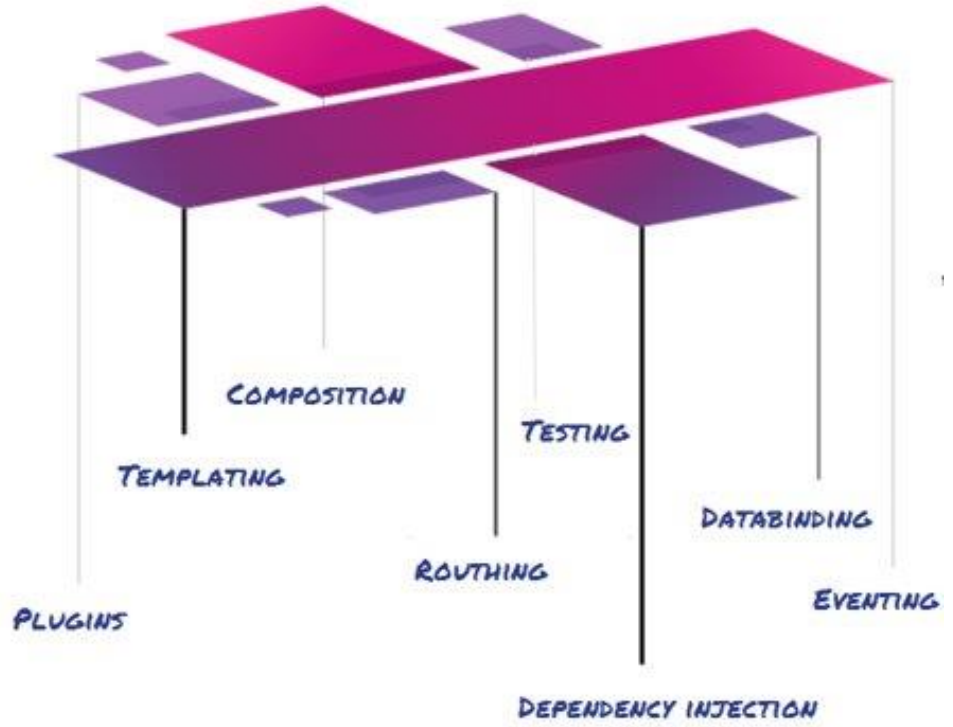
Aurelia

THE NEXT GENERATION FRAMEWORK

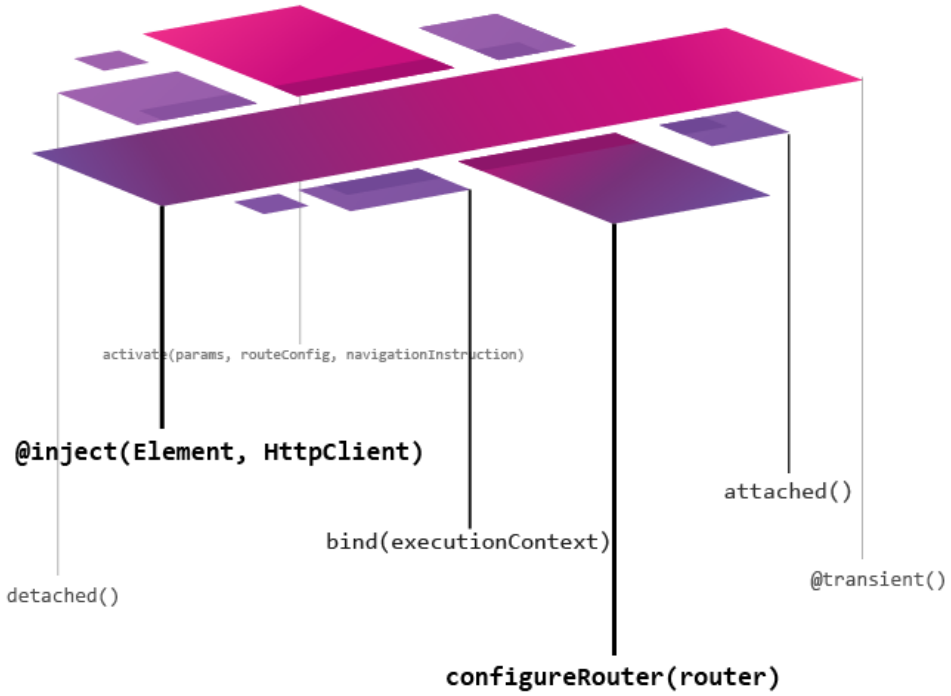
Aurelia?

Aurelia is a modern, front-end JavaScript framework designed for building browser, mobile, and desktop applications, all open source and built on open web standards.



Architecture

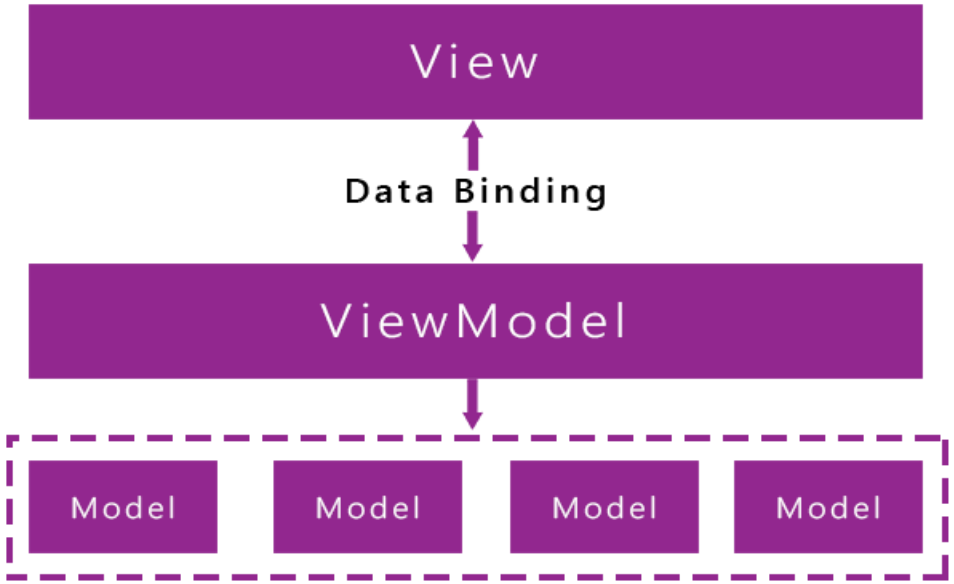
The architecture is **not monolithic**, it is actually a collection of more functionally oriented modules that help us build simple web applications. The Aurelia feature-oriented modules include plugins, templating, composition, routing, testing, dependency injection, data binding, and eventing.



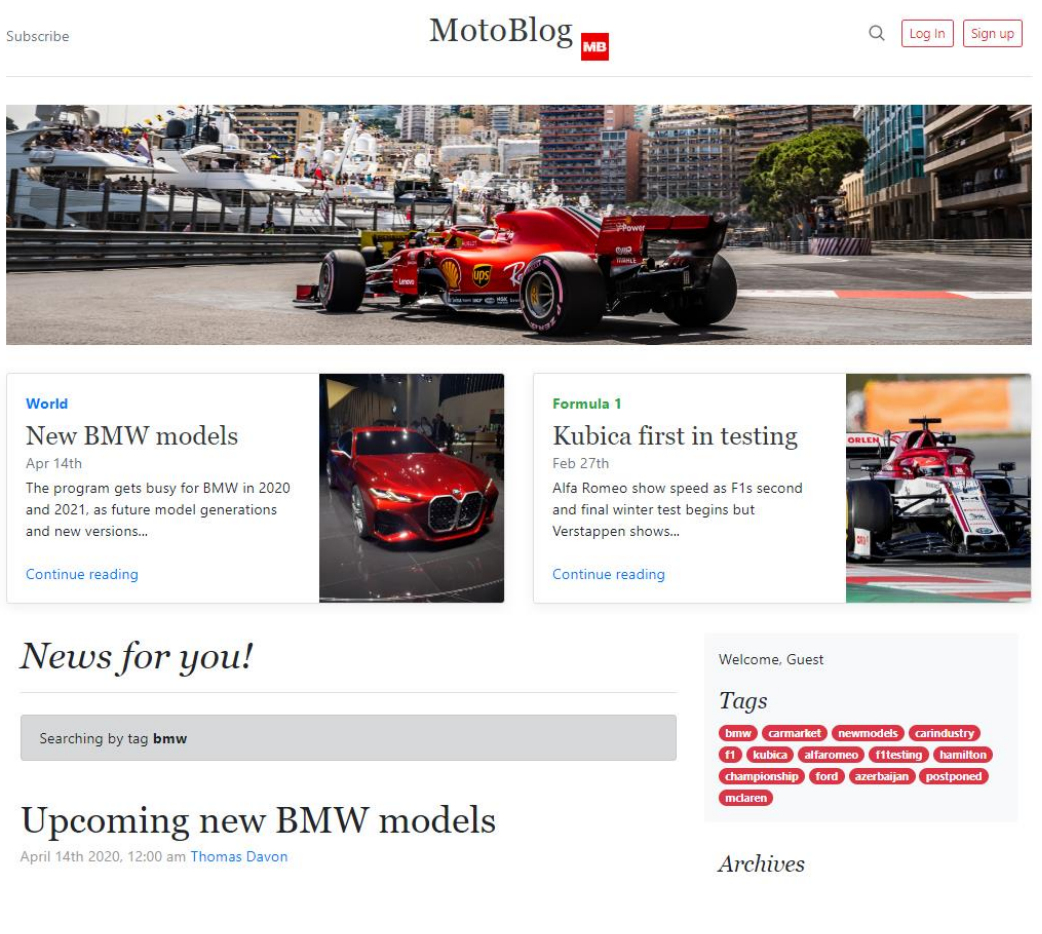
MVVM Architecture

Aurelia follows a **MVVM** software architectural pattern.

- The **View**
- The **ViewModel**
- The **Model**



Project as a proof of concept



Components

- In Aurelia, user interface components 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 **ES Next** and provides data and behavior to the view.

test.html (view)

```
<template>
  <h1>${message}</h1>
</template>
```

test.js (view-model)

```
export class Test {
  constructor() {
    this.message = 'Hello world';
  }
}
```

create.html

src > posts > create.html > template > form > div.form-group > label

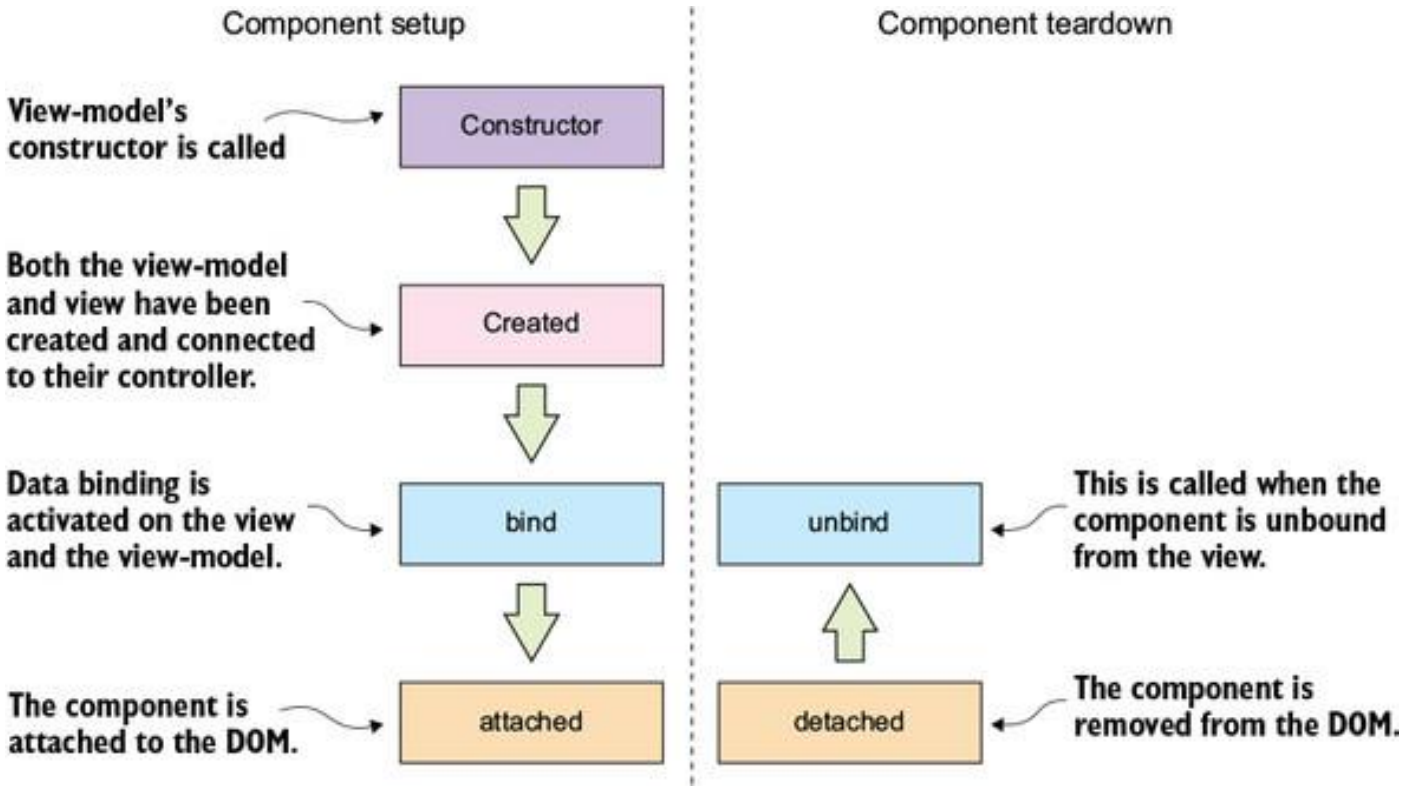
```
<template>
  <h1>Create Post</h1>
  <form submit.delegate="createPost()">
    <div class="form-group" style="margin-top: 20px;">
      <label for="title">Title</label>
      <input type="text" class="form-control" placeholder="Your Post Title" value.
      bind="post.title" />
    </div>
    <div class="form-group">
      <label for="body">Body</label>
      <textarea class="form-control" rows="10" value.bind="post.body"></textarea>
    </div>

    <div class="form-check" repeat.for="tag of allTags">
      <input class class="form-check-input" type="checkbox" value.bind="tag" checked.
      bind="post.tags" />
      <label class="form-check-label">
        ${ tag }
      </label>
    </div>

    <div class="form-group" style="margin-top: 20px;">
      <input type="text" value.bind="newTag" />
      <a href="" click.delegate="addTag()">add new tag</a>
    </div>

    <hr />

    <button type="submit" if.bind="currentUser" class="btn btn-danger">Create Post</
    button>

    <div if.bind="!currentUser">

      <div class="alert alert-danger">
        You have to be logged in to be able to create posts!
      </div>

    </div>

  </form>
</template>
```

create.js

src > posts > create.js > Create > constructor

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

import {
  Router
} from 'aurelia-router';

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

import {
  PostService
} from '../common/services/post-service';

@inject(EventAggregator, Router, PostService)

export class Create {
  constructor(EventAggregator, Router, PostService) {
    this.router = Router;
    this.postService = PostService;
    this.ea = EventAggregator;
  }

  attached() {
    this.post = {
      title: '',
      body: '',
      tags: []
    };
    this.postService.allTags().then(data => {
      this.allTags = data.tags;
    }).catch(error => {
      console.log(error);
    })
  }

  addTag() {
    this.allTags.push(this.newTag);
    this.post.tags.push(this.newTag);
    this.newTag = "";
  }

  createPost() {
    this.postService.create(this.post).then(data => {
      this.ea.publish('post-updated', Date());
```

Component lifecycle



constructor()

- Constructor method is used for initializing an object created with a class. This method is called first. If you don't specify this method, the default constructor will be used.

```
import {
  PLATFORM
} from "aurelia-framework";

import {
  inject
} from 'aurelia-framework';

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


import {
  PostService
} from './common/services/post-service';

import {
  AuthService
} from './common/services/auth-service';


require('bootstrap/dist/css/bootstrap.min.css');
require('./assets/styles/blog.css');

@inject(EventAggregator, AuthService, PostService)
export class App {

  constructor(EventAggregator, AuthService, PostService) {
    this.ea = EventAggregator;
    this.postService = PostService;
    this.authService = AuthService;

  }
```

attached()

- Attached method is invoked once the component is attached to the DOM.

```
attached() {
  this.currentUser = this.authService.currentUser;

  this.subscription = this.ea.subscribe('user', user => {
    this.currentUser = this.authService.currentUser;
  })

  this.updateSidebar();

  this.postSubscription = this.ea.subscribe('post-updated', updatedAt => {
    this.updateSidebar();
  })
}
```

detached()

- This method is opposite to **attached**. It is invoked when the component is removed from the DOM.

```
detached() {
  this.subscription.dispose();
  this.postSubscription.dispose();
}
```

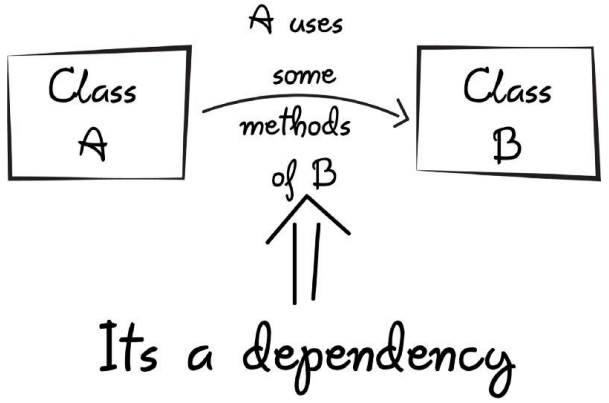
And the others:

- **created(owningView, myView)** – This is called once the view and view-model are created and connected to the controller. This method takes two arguments. The first one is the view where the component is declared **(owningView)**. The second one is the component view **(myView)**.
- **bind(bindingContext, overrideContext)** – At this point of time, the binding has started. The first argument represents the binding context of the component. The second one is **overrideContext**.
- **unbind()** – The last lifecycle method is **unbind**. It is called when the component is unbound.

Dependency injection

When building applications, it's often necessary to take a "divide and conquer" approach.

- @inject(type1, type2, ...)




```
@inject(EventAggregator, AuthService, PostService)
export class App {

  constructor(EventAggregator, AuthService, PostService) {
    this.ea = EventAggregator;
    this.postService = PostService;
    this.authService = AuthService;

  }
```

Aurelia-CLI generator

- Executing **"au generate <resource> <destination-subfolder>"**
- The line below generates a completely basic component consisting of:

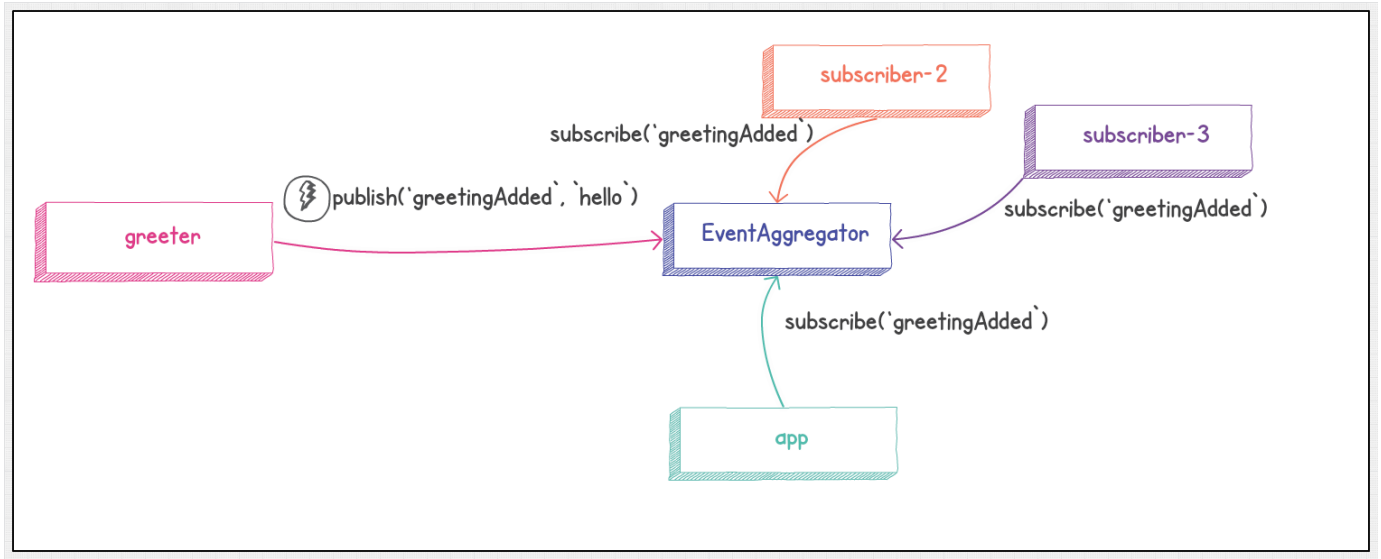
  - test.html (view)

  - test.js (view-model)
- Both in folder named posts.

```
PS C:\Users\tomek\source\repos\Aurelia.js-PoC> au generate component test posts
Local aurelia-cli v1.3.0
Created test in the 'src\posts' folder
PS C:\Users\tomek\source\repos\Aurelia.js-PoC>
```

Event aggregator

- Event aggregator should be used when your events need to be attached to more listeners or when you need to observe some functionality of your app and wait for the data update.
- Aurelia event aggregator has three methods. The publish method will fire off events and can be used by multiple subscribers. For subscribing to an event, we can use the subscribe method. And finally, we can use the dispose method to detach the subscribers.



publish()

```
29      login() {
30        this.error = null;
31        this.authService.login(this.name).then(data => {
32          this.ea.publish('user', data.name);
33          this.router.navigateToRoute('home');
34        }).catch(error => {
35          console.log(error.message);
36          this.error = error.message;
37        });
38      }
39    }
```

subscribe()

```
37    attached() {
38      this.currentUser = this.authService.currentUser;
39
40      this.subscription = this.ea.subscribe('user', user => {
41        this.currentUser = this.authService.currentUser;
42      })
43
44      this.updateSidebar();
45
46      this.postSubscription = this.ea.subscribe('post-updated', updatedAt => {
47        this.updateSidebar();
48      })
49    }
```

dispose()

```
121     detached() {
122         this.subscription.dispose();
123         this.postSubscription.dispose();
124     }
```

Routing

One of the most important Aurelia components.

Router in view

```
<div class="col-md-8 blog-main">

  <router-view></router-view>

  <hr />

  <a class="btn btn-outline-danger" style="margin-bottom: 30px;" href="#">MAIN PAGE</a>

</div>
```

Router in view-model

```
configureRouter(config, router) {
  config.title = 'MotoBlog';
  config.map([{
      route: '',
      name: 'home',
      moduleId: PLATFORM.moduleName('posts/index'),
      title: 'All Posts'
    },

    {
      route: 'login',
      name: 'login',
      moduleId: PLATFORM.moduleName('auth/login'),
      title: 'Log In'
    },

    {
      route: 'signup',
      name: 'signup',
      moduleId: PLATFORM.moduleName('auth/signup'),
      title: 'Sign Up'
    },

    {
      route: 'create-post',
      name: 'create-post',
      moduleId: PLATFORM.moduleName('posts/create'),
      title: 'Create Post'
    },

    {
      route: 'post/:slug',
      name: 'post-view',
      moduleId: PLATFORM.moduleName('posts/view'),
      title: 'View Post'
    },
```

Using routes

```
<aside class="col-md-4 blog-sidebar">
  <div class="p-3 mb-3 bg-light rounded">

    <span style="display: block; margin-bottom: 20px;">Welcome, ${ currentUser || 'Guest' }</span>


    <h4 class="font-italic">Tags</h4>
    <a route-href="route: tag-view; params.bind: { tag }" repeat.for="tag of tags">
      <span class="badge badge-pill badge-danger">${ tag }</span>
    </a>
  </div>

  <a class="btn btn-danger btn-lg btn-block" if.bind="currentUser" route-href="create-post"> &#10133; NEW
    POST</a>

  <div class="p-3">
    <h4 class="font-italic">Archives</h4>
    <ol class="list-unstyled mb-0">
      <li repeat.for="archive of archives">
        <a route-href="route: archive-view; params.bind: { archive }">
          ${ archive }
        </a>
      </li>
    </ol>
  </div>
```

Using routes

```
{
  route: 'tag/:tag',
  name: 'tag-view',
  moduleId: PLATFORM.moduleName('posts/tag-view'),
  title: 'View Posts by Tag'
},

{
  route: 'archive/:archive',
  name: 'archive-view',
  moduleId: PLATFORM.moduleName('posts/archive-view'),
  title: 'View Posts by Archive'
},
```

Data binding

test.html (view)

```
<template>
  <h1>${message}</h1>
</template>
```

test.js (view-model)

```
export class Test {
  constructor() {
    this.message = 'Hello world';
  }
}
```

Two-way binding

- The true magic happens, when we find out that we can bind our data both ways. For example in the view part of my **login** component, I've got input, that was simply binded to the view-model variable **name**. It lets us save the name we wrote in the form and use it further along as a welcome message or as a new post author.

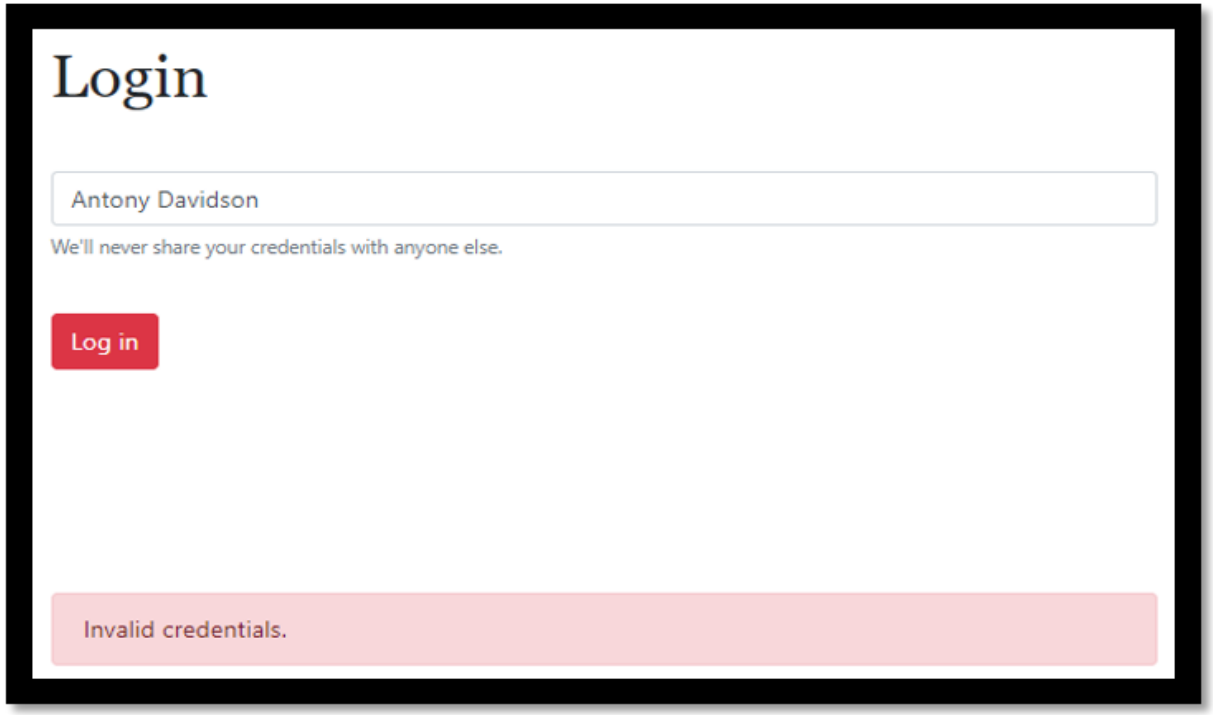
```
<div class="alert alert-danger" if.bind="error">
  ${ error }
</div>
```

```
<input type="text" class="form-control" style="margin-top: 40px;" placeholder="Your Name" value.bind="name" />
```

if.bind

```
<div class="alert alert-danger" if.bind="error">
  ${ error }
</div>
```

- We write **if.bind = "variable"** in the element attributes and it checks in the viewmodel if the variable is binded. If it's not, then the element of the view won't be displayed, but when it is – then we can see it on screen.



Value converter

- Most commonly we'll be creating value converters that translate model data to a format suitable for the view. But sometimes we'll need to convert data from the view to a format expected by the view-model, typically when using two-way binding with input elements.

```
import moment from 'moment';

export class DateFormatValueConverter {
  toView(value) {
    return moment(value).format('MMMM Do YYYY, h:mm a');
  }

  fromView(value) {
    //
  }
}
```

Custom elements

- The simplest way to create an Aurelia custom element is to create an Aurelia view template in an HTML file and then require it in to another Aurelia view template. HTML only custom elements are a highly useful strategy for dealing with functionality that has no need for ViewModel logic but is likely to be reused.
- It is even possible to create bindable properties for an HTML only custom element by putting a comma separated list of property names on the **bindable** attribute of the template element.

Custom elements

```
<template bindable="error, title, posts">
  <require from="./blog-post"></require>

  <h1 class="pb-3 mb-4 font-italic border-bottom">
    News for you!
  </h1>

  <div class="alert alert-danger" if.bind="error">
    ${ error }
  </div>

  <div class="alert alert-dark" style=" margin-bottom: 40px;">
    <slot> </slot>
  </div>

  <div repeat.for="post of posts">
    <blog-post post.bind="post"></blog-post>
    <a route-href="route: post-view; params.bind: { slug: post.slug }">View Post &raquo;</a>
    <hr />
  </div>

</template>
```

Repeaters

- There comes a time in most applications where you will want to loop through an array of data on the front-end. In Aurelia we have the **repeat.for** attribute which allows us to use Aurelia's repeater functionality in our view templates.

```
<template bindable="error, title, posts">
  <require from="./blog-post"></require>

  <h1 class="pb-3 mb-4 font-italic border-bottom">
    News for you!
  </h1>

  <div class="alert alert-danger" if.bind="error">
    ${ error }
  </div>

  <div class="alert alert-dark" style=" margin-bottom: 40px;">
    <slot> </slot>
  </div>

  <div repeat.for="post of posts">
    <blog-post post.bind="post"></blog-post>
    <a route-href="route: post-view; params.bind: { slug: post.slug }">View Post &raquo;</a>
    <hr />
  </div>

</template>
```

Slot API

- Most of the standard HTML elements allow for content inside them. Custom Elements would be of limited use if we couldn't put content inside them. Thus, we need a way to take this content and place it inside our custom element's template. The Shadow DOM spec provides the `slot` processing instruction for doing this.

Slot API

```
<template bindable="error, title, posts">
  <require from="./blog-post"></require>

  <h1 class="pb-3 mb-4 font-italic border-bottom">
    News for you!
  </h1>

  <div class="alert alert-danger" if.bind="error">
    ${ error }
  </div>

  <div class="alert alert-dark" style=" margin-bottom: 40px;">
    <slot> </slot>
  </div>

  <div repeat.for="post of posts">
    <blog-post post.bind="post"></blog-post>
    <a route-href="route: post-view; params.bind: { slug: post.slug }">View Post &raquo;</a>
    <hr />
  </div>

</template>
```

Slot API

```
<template>
  <require from="../resources/elements/post-list.html"></require>

  <post-list error.bind="error" posts.bind="posts">
    Viewing all posts
  </post-list>
</template>
```

```
<template>
  <require from="../resources/elements/post-list.html"></require>
  <post-list error.bind="error" posts.bind="posts">
    Searching by tag <strong>${tag}</strong>
  </post-list>
</template>
```

```
<template>
  <require from="../resources/elements/post-list.html"></require>
  <post-list error.bind="error" posts.bind="posts">
    Searching by archive of <strong>${archive}</strong>
  </post-list>
</template>
```

Thank you for your attention!