LEARNING aurelia

#aurelia

Table of Contents

About

Chapter 1: Getting started with aurelia

Remarks

Aurelia is a modular modern front-end Javascript framework for building browser, mobile and desktop applications built on open web standards. Rather than being an all inclusive framework, Aurelia adopts a feature-per-module approach to its architecture. Any piece of Aurelia is replaceable with a third-party dependency (or removed altogether).

Unlike competing frameworks such as Angular, Aurelia's core philosophy is to abide by official web standards and DOM API's, this ensures more bare-metal performance as it sits closer to native API's and has very little abstraction. Your views are HTML, your view-models are Javascript (or TypeScript) and Aurelia's component model is based on W3C Web Components HTML Templates and ShadowDOM, so your code mirrors that of a real emerging web standard.

Even though Aurelia is a fully-featured single page application framework, the learning curve is a lot lower than existing alternatives like Angular and Ember. Its templating syntax is intuitive and shares similarities with previous frameworks and libraries you might have worked with.

A developer or company might opt to use Aurelia over other solutions because standards matter to them and writing abstract framework-specific boilerplate code does not feel right to you and you're looking for a framework that prefers to let native DOM API's do the work for you.

Examples

Hello World: Getting started with aurelia-cli

This example will show you how to quickly get a hello world Aurelia application up and running using the Aurelia CLI.

Prerequisites

The Aurelia CLI is a Node.js based application, so make sure you install it first before proceeding. You will need Node.js 4.4.7 or later.

You will also need a Git client installed. Some great options include: Github Desktop, Sourcetree and Git SCM.

Installing the CLI

Provided you installed Node.js and Npm correctly, open up a Command Prompt/PowerShell or Bash terminal and install the CLI globally using the following command:

```
npm install aurelia-cli -g
```

Before proceeding, run `au -v` to make sure that the Aurelia CLI successfully installed. You should see a version number displayed.

Creating your first Aurelia application

Now you have the CLI installed, to create a new project run the following command and following the informative on screen prompts:

```
au new
```

You'll get a choice of different formats and loaders, to keep things simple just select the defaults. As you become more familiar with the CLI, you can configure these options to match your needs.

Running your Aurelia application

To run your Aurelia application, from the same folder run: `au run` - you should now see a fully-functioning hello world application when you open up your application in a web browser. By default, the CLI dev server will be available at `http://localhost:9000`

Conclusion

You have just successfully created a "hello world" Aurelia application using the CLI.

Read Getting started with aurelia online: https://riptutorial.com/aurelia/topic/1063/getting-started-with-aurelia

Chapter 2: Aurelia CLI

Examples

Adding Bootstrap To A CLI Application

A commonly used CSS/Javascript library is Bootstrap. To install it into your Aurelia CLI driven application first you need to install it using Npm.

```
npm install bootstrap --save
```

Because Bootstrap has a hard dependency on jQuery, we need to make sure we also have jQuery installed:

```
npm install jquery --save
```

Now in your preferred IDE/code editor open up the following file in your project directory: `aurelia_project/aurelia.json` and scroll down to the `build.bundles` section towards the end of the file. We will want to add Bootstrap to one of the bundles. For this example, we will be adding both jQuery and Bootstrap into the vendor bundle.

```
"jquery",
{
  "name": "bootstrap",
  "path": "../node_modules/bootstrap/dist",
  "main": "js/bootstrap.min",
  "deps": ["jquery"],
  "exports": "$",
  "resources": [
    "css/bootstrap.css"
  ]
}
```

This will make Bootstrap accessible in your application and importable via `import 'bootstrap'` in your code (this is the `name` property defined above). Notice the reference to `jquery` in the `"deps": []` section of our bootstrap definition. We also specify that we want to bundle Bootstrap's CSS in our main `vendor-bundle.js` file using the `"resources":[]` property

Last and not least, to use our newly added Bootstrap dependency, we want to first import the Bootstrap library inside of our `main.js` file by putting the following at the beginning of the file.

```
import 'bootstrap';
```

We want to include the Bootstrap CSS from a place where it will be globally accessible to the whole application, so inside of `app.html` put the following between the `<template></template>` tags.

```
<require from="bootstrap/css/bootstrap.css"></require>
```

Adding Lodash To A CLI Application

A commonly used utility library is Lodash. To install it into your Aurelia CLI driven application first you need to install it using Npm.

```
npm install lodash --save
```

Now in your preferred IDE/code editor open up the following file in your project directory: aurelia_project/aurelia.json and scroll down to the build.bundles section towards the end of the file. We will want to add Lodash to one of the bundles. For this example, we will be adding Lodash into the vendor bundle.

Because Lodash does follow the CommonJS conventions and exports one concatenated file we can easily register it by adding the following object ot the dependencies array:

```
{
  "name": "lodash",
  "path": "../node_modules/lodash/lodash.min"
}
```

> Note that we omited the .js file suffix as those will be automatically appended when resolving the dependency.

Now we can use Lodash by simply importing it, using the ES6 import syntax:

```
import _ from 'lodash';
```

Installing The Aurelia-I18N Plugin

In order to get the official I18N Plugin into your CLI Project we need to install it by following the next steps.

First you want to install the plugin via npm:

```
npm install aurelia-i18n
```

Since Aurelia-I18N is backed by i18next, you should install it and a backend plugin of your choice. As an example we're going to leverage the i18next-xhr-backend:

```
npm install i18next i18next-xhr-backend
```

After that we need to tell our CLI App about the new dependencies. To do so we're going to open the file *aurelia_project/aurelia.json* and scroll down to section named *dependencies*. In there add the following three entries:

```
{
  "name": "i18next",
  "path": "../node_modules/i18next/dist/umd",
  "main": "i18next"
},
{
  "name": "aurelia-i18n",
  "path": "../node_modules/aurelia-i18n/dist/amd",
  "main": "aurelia-i18n"
},
{
  "name": "i18next-xhr-backend",
  "path": "../node_modules/i18next-xhr-backend/dist/umd",
  "main": "i18nextXHRBackend"
}
```

Great, now following the official Aurelia-I18N Guide we create a folder in the root of your app named *locales*.

> You have to put the folder into the root (on same level as *src*) as this is the hosted root of your app

In there add subfolders for each language you'd like to support, eg *en* and *de* for English and German language.

Inside of each of those folders create a file named *translation.json* with your translation keys and values. Follow the official guide for detailed info.

Last but not least it's time to wire up the plugin inside your app. Therefore go to your *src/main.js* file and configure it as follows.

```
/**********************************************/
/          add the necessary imports          /
/**********************************************/
import environment from './environment';
import Backend from 'i18next-xhr-backend';

//Configure Bluebird Promises.
//Note: You may want to use environment-specific configuration.
Promise.config({
  warnings: {
    wForgottenReturn: false
  }
});

export function configure(aurelia) {
  aurelia.use
    .standardConfiguration()
    .feature('resources');

  if (environment.debug) {
    aurelia.use.developmentLogging();
  }

  if (environment.testing) {
    aurelia.use.plugin('aurelia-testing');
  }

  /**********************************************/
  /             register the plugin             /
  /**********************************************/
  aurelia.use.plugin('aurelia-i18n', (instance) => {
    // register backend plugin
    instance.i18next.use(Backend);
    // adapt options to your needs (see http://i18next.com/docs/options/)
    // make sure to return the promise of the setup method, in order to guarantee proper
loading
    return instance.setup({
      backend: {                                  // <-- configure backend settings
        loadPath: './locales/{{lng}}/{{ns}}.json', // <-- XHR settings for where to get the
files from
      },
      lng : 'de',
      attributes : ['t','i18n'],
      fallbackLng : 'en',
      debug : false
    });
  });

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

Adding Aurelia Configuration to a CLI Application

Adding the aurelia-configuration to a cli application sometimes produces a build error. This is caused by a missing dependency so we simply add the dependency to the build bundle.

Try the following:

```
npm install deep-extend --save
npm install aurelia-configuration --save
```

Now add the following to the aurelia.json file in the dependencies array:

```
{
    "name": "deep-extend",
    "path": "../node_modules/deep-extend/lib/deep-extend"
  },
  {
    "name": "aurelia-configuration",
    "path": "../node_modules/aurelia-configuration/dist/amd",
    "main": "index"
  }
```

Read Aurelia CLI online: https://riptutorial.com/aurelia/topic/1916/aurelia-cli

Chapter 3: Aurelia CLI Explained

Examples

Setting Up Environment for Aurelia-cli Explained

- **OS:** Mac OS X 10.11 (Should work on Windows / Linux since we are using Vagrant)
    - Vagrant 1.8.4 Installed
- **Directory Structure on Host OS (Mac OS):**
    - /path/to/project
        - /provision
            - /packages
                - Note: If you use different vesions, be sure to update Variables at top of provision.sh script below.
                - atom.x86_64.rpm (Download: Atom)
                - node-v6.4.0-linux-x64.tar.xz (Download: Node)
        - /vagrant
            - Vagrantfile (File contents below)
            - provision.sh (File contents below)
- **Starting up the Virtual Machine** ($ == Terminal prompt)
    - In Mac OS Terminal
    - `$cd /path/to/project/vagrant`
    - `$vagrant up`
        - Downloads CentOS 7 vagrant box, runs provision script
        - Launches VM window outside of your Mac OS terminal
    - When all done, log into VM using gui
        - User: vagrant
        - PW: vagrant
    - Launch X Windows:
        - `$startx` (Starts a Gnome UI)
- **Setting up the VM**
    - Launch a Terminal window (Applications Drop Down Menu / Utilities)
    - Set up sudo to run "npm"
        - Get path to npm: `$which npm` (/opt/node-v6.4.0-linux-x64/bin)
        - Add to visudo "secure_path"
            - `$sudo visudo` (Requires basic knowledge of Vi/Vim)
            - Type "i" to go into Insert mode
            - Use arrow keys to navigate to "secure_path" line
            - Append ":/opt/node-v6.4.0-linux-x64/bin" to "secure_path"
            - Type "esc" to exit Insert mode
            - Type ":wq" to write changes and quit Vi
    - Update npm, fixes critical bugs in aurelia cli
        - `$npm install npm -g`
    - Install aurelia cli globally
        - `$npm install aurelia-cli -g`

- **Starting/Stopping the VM**
  - In Mac OS Terminal
  - `$cd /path/to/project/vagrant` (If you're not already in the directory with the Vagrantfile)
  - `$vagrant halt` (Keeps all files, closes GUI, shuts down VM)
  - `$vagrant up` (Restart VM, login via GUI when it's ready)
  - `$vagrant destroy` (Will need to download everything again, your local files will be deleted. Use this to start from scratch.)

Now you're ready to start building your website using Aurelia CLI!

Vagrantfile

```
    Vagrant.configure("2") do |config|
        config.vm.box = "centos/7"
        config.vm.hostname = "dev"

        config.vm.provider "virtualbox" do |vb|
            vb.gui = true
            vb.cpus = "4"
            vb.memory = "3092"
        end

        # Networking
        config.vm.network "private_network", ip: "192.168.0.3"
        config.vm.network :forwarded_port, guest: 80, host: 80 # nginx
        config.vm.network :forwarded_port, guest: 9000, host: 9000 # au run
        config.vm.network :forwarded_port, guest: 3001, host:3001 # BrowserSynch

        # Shares
        config.vm.synced_folder "../provision", "/home/vagrant/provision"

        # Provision
        config.vm.provision "shell", path: "provision.sh"
    end
```

provision.sh

```
HOME=/home/vagrant
NODE=node-v6.4.0-linux-x64
EPEL=epel-release-latest-7.noarch.rpm
ATOM=atom.x86_64

echo "*************************************"
echo "Provisioning virtual machine..."
echo "*************************************"
sudo cd $HOME

echo "************************"
echo "Updating yum..."
echo "************************"
sudo yum clean all
sudo yum -y install deltarpm yum-utils
sudo yum -y update --exclude=kernel*

echo "************************"
echo "Updating yum, installing Dev Tools..."
echo "***********************"
sudo yum -y groupinstall "Base"
sudo yum -y groupinstall "GNOME Desktop"
sudo yum -y groupinstall "Development Tools"

echo "***********************"
echo "Installing tools..."
echo "***********************"
sudo yum install -y git tar gcc vim unzip wget curl tree nginx

if [ -d "/opt/$NODE" ]
then
  echo "**********************************"
  echo "Node already installed..."
  echo "**********************************"
else
  echo "**********************************"
  echo "Installing Node and update npm..."
  echo "**********************************"
  sudo cp /home/vagrant/provision/packages/$NODE.tar.xz /opt
  sudo tar -xpf /opt/$NODE.tar.xz -C /opt
  sudo echo "export PATH=\"$PATH:/opt/$NODE/bin\"" >> $HOME/.bash_profile
  sudo echo "export PATH=\"$PATH:/opt/$NODE/bin\"" >> /root/.bash_profile
  sudo source $HOME/.bash_profile
fi

echo "**********************************"
echo "Installing Atom..."
echo "**********************************"
sudo rpm -ivh /home/vagrant/provision/packages/$ATOM.rpm

echo "******************************"
echo "Installing EPEL..."
echo "******************************"
wget -P /etc/yum.repos.d http://dl.fedoraproject.org/pub/epel/$EPEL
sudo rpm -ivh /etc/yum.repos.d/$EPEL

echo "***********************"
echo "Git setup..."
echo "***********************"
git config --global user.email "your@email.com"
git config --global user.name "Your Name"
git config --global github.user "Username"

echo "***********************"
echo "Don't forget to:"
echo "sudo visudo"
echo "Append npm path to secure_path: /opt/$NODE/bin"
echo "sudo npm install npm -g"
echo "sudo npm install aurelia-cli -g"
echo "***********************"


echo "*********************************"
echo "VM Provisioning Complete!"
echo "*********************************"
```

Aurelia CLI Basics Explained

This Example assumes you setup as explained in Example: **Setting Up Environment for aurelia-**

cli Explained of this document.

Creating a new project

- In main host os, open terminal (Mac OS in my case)
- `$cd /path/to/project/vagrant`
- `$vagrant up` (Launches VM GUI)
- Log into VM via UI
  - User: vagrant / PW: vagrant
  - `$startx`
  - Windows X starts Gnome session
  - Open a terminal in VM
- `$cd /home/vagrant`
- `$au new` (starts a series of prompts to setup project)
  - **Prompt:** Please enter a name for your new project below.
  - `$[aurelia-app]> MyProject` (Press Enter) (This will be the name of project main directory)
  - **Prompt:** Would you like to use the default setup or customize your choices?
  - `$[Default ESNext] > 3` (Press Enter) (Custom options)
  - **Prompt:** What transpiler would you like to use?
  - `$[Babel] >` (Press Enter) (Lets use Bable to transpile our ESNext to ES5)
  - **Prompt:** What css processor would you like to use?
  - `$[none] > 3` (If you want to use Sass for your css)
  - **Prompt** Would you like to configure unit testing?
  - `$[Yes] > 2` (Lets not setup testing for simplicity)
  - **Prompt** What is your default code editor?
  - `$[Visual Studio Code] > 2` (Atom, if you followed setup for this document)
  - **Outputs Setup Summary**
    - Project Configuration
    - **Name:** MyProject
    - **Platform:** Web
    - **Transpiler:** Babel
    - **CSS Processor:** Sass
    - **Unit Test Runner:** None
    - **Editor:** Atom
  - **Prompt** Would you like to create this project?
  - `$[Yes] > 1` (Press Enter)
  - **Prompt** Would you like to install the project dependencies?
  - `$[Yes] > 1` (Press Enter)

  Creates directory structure and downloads all the dependencies via npm.

You're now done and ready to start building!

Basic Project directory structure

- /home/vagrant/MyProject
    - /aurelia_project (Configuration files, Gulp Tasks, Generators)
    - /scripts (Builds publish here)
    - /src (This is where you will be wroking primarily)
        - resources (Put your images/fonts/icons here.)
        - app.js (The default View-Model)
        - app.html (The default View for an Aurelia App. The main entry point)
        - environment.js
        - main.js
    - favicon.ico
    - index.html (No need to edit this file)
    - package.json (npm install --save adds entries to this file)

Development cycle on the VM

- Open Atom, add project folder: /home/vagrant/MyProject
    - This is where you will be editing the html/js/scss files
- Open a Terminal:
    - `$au run --watch` (Builds app, starts BrowserSynch web server)
- Open web browser of choice in VM and go to: http://localhost:9000
    - As you edit your code and save files, you will see the updates live
- If you setup using Vagrantfile above, then you can use browser in Host OS (Mac OS) and go to: http://192.168.0.3:9000

Host your Aurelia Cli app on Nginx

Environment

This Example assumes you setup as explained in Example: **Setting Up Environment for Aurelia-cli Explained** of this document.

Summary of setup:

- On Mac OS X with Vagrant 1.8.4
- `$cd /path/to/project/vagrant`
- `$vagrant up`
- Log into VM via UI as User:vagrant / PW:vagrant
- `$cd /home/vagrant/MyProject` (Where you setup your Aurelia project as described in "Aurelia Cli Basics Explained" above.)
- EPEL repository should be setup

Basic Project directory structure

- /home/vagrant/MyProject
    - /aurelia_project

    - /scripts
    - /src
        - resources
        - app.js
        - app.html
        - environment.js
        - main.js
    - favicon.ico
    - index.html
    - package.json

Build your App

- `$au build` (Creates bundled files in projects /scripts directory)

Setting up Nginx

- `$sudo yum install nginx`
- Start the sever: `$sudo nginx`
- Edit nginx.conf file (See below)
    - Set root location to /var/www/html
    - Set resources location to /var/www/
- Reload the server after editing the file: `$sudo nginx -s reload`

Copy files from Aurelia project to server

```
$sudo cp /home/vagrant/MyProject/index.html /var/www/html/
$sudo cp -R /home/vagrant/MyProject/scripts /var/www/html/
$sudo cp mkdir /var/www/src/
$sudo cp -R /home/vagrant/MyProject/src/resources /var/www/src/
```

- Go to http://localhost on web browser in VM, you should see the default Aurelia app.
- If you setup as described above, port 80 is forwarded by VM, so you can go to a browser on your host OS and go to http://192.168.0.3:80 and see the site there as well.

  /etc/nginx/nginx.conf

  ```
  # For more information on configuration, see:
  #   * Official English Documentation: http://nginx.org/en/docs/
  #   * Official Russian Documentation: http://nginx.org/ru/docs/

  user nginx;
  worker_processes auto;
  error_log /var/log/nginx/error.log;
  pid /run/nginx.pid;

  # Load dynamic modules. See /usr/share/nginx/README.dynamic.
  include /usr/share/nginx/modules/*.conf;

  events {
      worker_connections 1024;
  }
  ```

```
http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile            on;
    tcp_nopush          on;
    tcp_nodelay         on;
    keepalive_timeout   65;
    types_hash_max_size 2048;

    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;

    # Load modular configuration files from the /etc/nginx/conf.d directory.
    # See http://nginx.org/en/docs/ngx_core_module.html#include
    # for more information.

    server {
        listen       80 default_server;
        listen       [::]:80 default_server;
        server_name  _;

        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;

    root /var/www/html/;

    location /src/resources {
        root /var/www/;
    }

        error_page 404 /404.html;
            location = /40x.html {
        }

        error_page 500 502 503 504 /50x.html;
            location = /50x.html {
        }
    }
}
```

Read Aurelia CLI Explained online: https://riptutorial.com/aurelia/topic/7040/aurelia-cli-explained

Chapter 4: Binding

Examples

Binding To File Inputs

```
export class MyViewModel {
    selectedFiles;
}
```

```
<template>
    <input type="file" files.bind="selectedFiles">
</template>
```

Binding To Select Elements

Strings Array

When selecting a value in the select dropdown and providing an array of strings, the selected value will be bound to the select element's value property as a string that we can display using string interpolation.

```
export class MyViewModel {
    animals = [];
    favouriteAnimal = null;

    constructor() {
        this.animals = [
            'Cat',
            'Dog',
            'Fish',
            'Rabbit',
            'Tiger',
            'Bat'
        ];
    }
}
```

```
<template>
    ${favouriteAnimal}
    <select name="animals" value.bind="favouriteAnimal">
        <option repeat.for="animal of animals">${animal}</option>
    </select>
</template>
```

Objects Array

Unlike the above example, when supplying an array of objects, when a value is selected in a dropdown, the model bound to that particular option is the supplied object.

```
export class MyViewModel {
    animals = [];
    favouriteAnimal = null;

    constructor() {
        this.animals = [
            {label: 'Cat', value: 99},
            {label: 'Dog', value: 493},
            {label: 'Fish', value: 934839200},
            {label: 'Rabbit', value: 8311},
            {label: 'Tiger', value: 22},
            {label: 'Bat', value: 3711}
        ];
    }
}
```

```
<template>
    <p>Favourite animal ID: ${favouriteAnimal.value}</p>
    <p>Favourite animal name: ${favouriteAnimal.label}</p>
    <select name="animals" value.bind="favouriteAnimal">
        <option repeat.for="animal of animals" model.bind="animal">${animal.label}</option>
    </select>
</template>
```

Binding To Checkboxes

Basic Checkboxes

```
export class MyViewModel {
    favoriteColors = [];
    colors = ['Red', 'Yellow', 'Pink', 'Green', 'Purple', 'Orange', 'Blue'];
}
```

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

  <p>Favourite colors:</p>
  <ul if.bind="favoriteColors">
    <li repeat.for="color of favoriteColors">${color}</li>
  </ul>
</template>
```

Checkboxes With Object Arrays

```
export class MyViewModel {
    people = [];
    selectedPeople = [];

    constructor() {
        this.people = [
            {name: 'John Michaels'},
            {name: 'Gary Stevens'},
            {name: 'Carrie Smitch'},
            {name: 'Jesus Wohau'}
        ];
    }
}
```

```
<template>
  <label repeat.for="person of people">
    <input type="checkbox" model.bind="person" checked.bind="selectedPeople" />
    ${person.name}
  </label>

  <p>Selected people:</p>
  <ul if.bind="selectedPeople">
    <li repeat.for="person of selectedPeople">${person.name}</li>
  </ul>
</template>
```

Checkbox with a Boolean

```
export class MyViewModel {
    agreeToTerms = false;
}
```

```
<template>
  <label><input type="radio" name="terms" model.bind="true" checked.bind="agreeToTerms"
/>Yes</label>
  <label><input type="radio" name="terms" model.bind="false" checked.bind="agreeToTerms"
/>No</label>
  <br><br>
  <strong>${agreeToTerms ? 'I agree' : 'I disagree'}</strong>
</template>
```

Binding To Radio Inputs

Basic Radios

```
export class MyViewModel {
    favoriteColor = null;
    colors = ['Red', 'Yellow', 'Pink', 'Green', 'Purple', 'Orange', 'Blue'];
}
```

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

Radios With Object Arrays

```
export class MyViewModel {
    people = [];
    selectedPerson = null;
    constructor() {
        this.people = [
            {name: 'John Michaels'},
            {name: 'Gary Stevens'},
            {name: 'Carrie Smitch'},
            {name: 'Jesus Wohau'}
        ];
    }
}
```

```
<template>
  <label repeat.for="person of people">
    <input type="radio" name="person" model.bind="person" checked.bind="selectedPerson" />
    ${person.name}
  </label>
</template>
```

Radios with a Boolean

```
export class MyViewModel {
    agreeToTerms = null;
}
```

```
<template>
  <label><input type="radio" name="terms" model.bind="null" checked.bind="agreeToTerms" />No
Answer</label>
  <label><input type="radio" name="terms" model.bind="true" checked.bind="agreeToTerms"
/>Yes</label>
  <label><input type="radio" name="terms" model.bind="false" checked.bind="agreeToTerms"
/>No</label>
</template>
```

Binding Styles

Binding to the browser native `style` attribute using Aurelia. If using string interpolation, you should use the `css` alias so styling works in Internet Explorer.

Style String

```
export class MyViewModel {
  constructor() {
    this.styleString = 'color: #F2D3D6; background-color: #333';
  }
}
```

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

Style Object

```
export class MyViewModel {
  constructor() {
    this.styles = {color: '#F2D3D6', 'background-color': '#333'};
  }
}
```

```
<template>
  <div style.bind="styles"></div>
</template>
```

String Interpolation

Very similar to string binding above, this allows you to use string interpolation to bind to styles instead. If any of the values change, they will be updated in the view accordingly.

Note: *for Internet Explorer compatibility, we use the alias `css` to bind styles. This ensures that string interpolation works in Internet Explorer.*

```
export class MyViewModel {
    color = 'red';
    height = 350;
    width = 350;
}
```

```
<template>
    <div css="width: ${width}px; height: ${height}px; color:${color}">My Text</div>
</template>
```

Conditionally Show & Hide a HTML Element

When using `show.bind` the element remains in the page and is either hidden or visible through the use of `display:none` or `display:block` behind the scenes.

```
export class MyViewModel {
    isVisible = false;
}
```

```
<template>
    <div show.bind="isVisible"><strong>I can be both hidden or visible, but not at the same
time</strong></div>
</template>
```

Conditionally Add or Remove a HTML Element

Unlike `show.bind` when using `if.bind` the element will be removed from the page if the supplied binding value is `false` or added into the page if the value is `true`.

```
export class MyViewModel {
    isVisible = false;
}
```

```
<template>
```

```
    <div if.bind="isVisible"><strong>If you can read this, I am still here.</strong></div>
</template>
```

Grouped Elements

Sometimes there might be a situation where you want to add or remove a whole group of elements at once. For this, we can use a `<template>` element to show or hide additional elements without the need for a placeholder element like a DIV.

```
export class MyViewModel {
    hasErrors = false;
    errorMessage = '';
}
```

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

Read Binding online: https://riptutorial.com/aurelia/topic/3497/binding

Chapter 5: Custom Elements

Introduction

A Custom Element in Aurelia is used to extend the basic set of HTML elements by feature-enriched, reusable components. A Custom Element normally exists out of two files, a View-Model based on Javasciprt, and a corresponding view written in HTML. Both files compose the HTML element which can then be used throughout the application like every other HTML element.

Examples

Creating A Custom Element With Bindable Properties

Creating a Custom Element with bindable properties is a snap. If you want to create an element that accepts one or more values which the plugin can use, the `@bindable` decorator and syntax is what you are looking for.

Below, we are creating a custom element that accepts an array of fruits and displays them.

Example: *my-element.js*

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

const validFruits = [
    'Apple',
    'Banana',
    'Orange',
    'Pineapple',
    'Grapes',
    'Peach',
    'Plum',
    'Dates',
    'Strawberries'
];

export class FruitCustomElement {
    @bindable fruits = [];

    fruitsChanged(newVal, oldVal) {
        if (newVal) {
            newVal.filter(fruit => {
                return validFruits.indexOf(fruit) >= 0;
            });
        }
    }
}
```

```
<template>
    <ul if.bind="fruits">
        <li repeat.for="fruit of fruits">${fruit}</li>
    </ul>
```

```
</template>
```

Using it:

```
export class MyViewModel {
    myFruits = [];

    attached() {
        this.myFruits = ['Apple', 'Banana', 'Orange', 'Pineapple', 'Grapes', 'Peach'];
    }
}
```

```
<template>
    <require from="./my-element"></require>

    <fruit fruits.bind="myFruits"></fruit>
</template>
```

HTML Only Custom Element With Bindable Parameters

In the following, we are creating an example of an Aurelia Custom Element which will allow you to display Youtube videos via their video ID.

An Aurelia Custom Element can be defined in two different ways: the first one is by creating a viewmodel and accompanying view, the second one is by just creating a basic HTML file and using the `bindable` property on the `<template>` tag of the view itself.

For our example below, an HTML-only Custom Element makes sense as we are just making it easy to use Youtube embed code in our application.

Example: *youtube.html*

```
<template bindable="videoId">
<iframe width="560" height="315" src="https://www.youtube.com/embed/${videoId}"
frameborder="0" allowfullscreen></iframe>
</template>
```

The filename for an HTML-only Custom Element is what will be used as the tag name in our HTML. Therefor, ensure that you don't call it something generic like `header.html`, `footer.html` or any other name that is a native HTML element already.

Using it:

```
<template>
    <require from="./youtube.html"></require>

    <youtube video-id="C9GTEsNf_GU"></youtube>
</template>
```

Creating A Custom Element Based On Naming Conventions

A basic custom element is created in Aurelia based on naming conventions, by simply adding the suffix `CustomElement` to the name of a class. This suffix will automatically be stripped out by Aurelia. The remaining part of the class name will be lowercased and separated using a hyphen and can then be used as the element name.

Example: ***my-element.js***

```
export class SuperCoolCustomElement {

}
```

```
<template>
    <h1>I am a custom element</h1>
</template>
```

Using it:

To use the newly defined custom element, at first, the tag name needs to be retrieved from the class name. The `CustomElement` suffix will be stripped out and the remaining part (`SuperCool`) will lowercased and separated by hyphen on each capital letter. Hence, `SuperCoolCustomElement` becomes `super-cool` and forms the basis of our element.

```
<template>
    <require from="./my-element"></require>

    <super-cool></super-cool>
</template>
```

Worth noting is that our example above is a bit contrived. We could have just created a HTML-only Custom Element and not create a viewmodel at all. However, the viewmodel approach approach allows you to provide bindable properties to make your element configurable (as shown in other examples).

Creating A Custom Element Using the @customElement Decorator

In most examples, the class naming convention is used to define an Aurelia Custom Element. However, Aurelia also provides a decorator that can be used to decorate a class. The class is again treated as a custom element by Aurelia then.

The value supplied to the decorator becomes the name of the custom HTML element.

Example: *my-element.js*

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

@customElement('robots')
export class MyClass {

}
```

```
<template>
    <h1>I am a robots custom element</h1>
</template>
```

Using it:

```
<template>
    <require from="./my-element"></require>

    <robots></robots>
</template>
```

Javascript Only Custom Element

A Custom Element consisting of Javascript only includes the corresponding HTML view within the `@inlineView` decorator of Aurelia.

The following example takes two bindable paramters, a prename and a surename, and display both within a predefined sentence.

Example: *my-element.js*

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

@customElement('greeter')
@inlineView(`
  <template>
    <b>Hello, \${prename} \${surename}.</b>
  </template>
`)
export class Greeter {
  @bindable prename;
  @bindable surename;
}
```

Using it:

```
<template>
  <require from="./greeter"></require>
  <greeter prename="Michael" surename="Mueller"></greeter>
</template>
```

This will output "**Hello, Michael Mueller.**" in the browser's window.

Read Custom Elements online: https://riptutorial.com/aurelia/topic/2019/custom-elements

Chapter 6: Dependency Injection

Remarks

If injecting more than one class, the order you put them in the @inject() statement does not matter. However, the order they appear in the @inject() statement must match the order of the parameters in the constructor.

Examples

Get and Display Username by Id

```
import {User} from 'backend/user'; // import custom class
import {inject} from 'aurelia-framework'; // allows us to inject

@inject(User) // inject custom class
export class ProfileView {
  constructor(user) { // use instance of custom class as a parameter to the constructor
    this.user = user; // save instance as a an instance variable
    this.username = '';
  }

  activate(params) {
    // call function from custom class, then save the result as another instance variable
    return this.user.getUsernameById(param.user_id)
      .then(user => this.username = user.username);
  }
}
```

Read Dependency Injection online: https://riptutorial.com/aurelia/topic/7879/dependency-injection

Chapter 7: Templating

Examples

Creating A Basic Template

In Aurelia all HTML templates are defined inside of opening and closing `<template></template>` tags. All of your HTML and Aurelia specific logic goes inside of these template tags and cannot exist outside of them.

```
<template>

</template>
```

Working With Loops Using "repeat.for"

Looping over an iterable defined inside of your viewmodel or passed through as a bindable (if a Custom Attribute or Custom Element) can be done like so.

An array of string values

```
export class MyViewModel {
    myIterable = ['String 1', 'String 2', 'String 3', 'String 4'];
}
```

```
<template>
    <div repeat.for="item of myIterable">${item}</div>
</template>
```

An array of objects

```
export class MyViewModel {
    myIterable = [
        {name: "John Citizen", age: 42},
        {name: "Maxwell Smart", age: 75},
        {name: "Gary TwoShoes", age: 51}
    ];
}
```

```
<template>
    <div repeat.for="item of myIterable">
        <strong>Name:</strong> ${item.name}<br>
        <strong>Age:</strong> ${item.age}
    </div>
</template>
```

A Map

```
export class MyViewModel {
    myIterable = null;

    constructor() {
        this.myIterable = new Map();
        this.myIterable.set(0, 'My Value');
        this.myIterable.set(1, 'Something Different');
        this.myIterable.set(2, 'Another String #32837');
    }
}
```

```
<template>
    <div repeat.for="[id, item] of myIterable">
        ${id}<br>
        ${item}
    </div>
</template>
```

A numeric loop

```
<template>
    <div repeat.for="i of 10">${i}</div>
</template>
```

Read Templating online: https://riptutorial.com/aurelia/topic/2162/templating

Chapter 8: Value Converters

Remarks

This section provides an overview of Value Converters in Aurelia. It should detail not only how to create a value converter, but also why you might want to use them and many examples of basic tasks accomplished through the use of a Value Converter.

Value converters can be chained and used alongside other binding features in Aurelia such as Binding Behaviors.

Examples

Creating A Basic Value Converter

While Value Converters can be comprised of either a `toView` or `fromView` method, in the below example we will be creating a basic Value Converter which just uses the `toView` method which accepts the value being sent to the view as the first argument.

to-uppercase.js

```
export class ToUppercaseValueConverter {
    toView(value) {
        return value.toUpperCase();
    }
}
```

Using it:

```
export class MyViewModel {
    stringVal = 'this is my test string';
}
```

```
<template>
    <require from="./to-uppercase"></require>

    <h1 textContent.bind="stringVal | toUppercase"></h1>
</template>
```

The text value of our heading one element should be `THIS IS MY TEST STRING` this is because the `toView` method which accepts the value from the view and specifies that the view should get our new value which are are using `String.prototype.toUpperCase()`

The class name is in this case `ToUppercaseValueConverter`, where `ValueConverter` tells aurelia what it is (There is also a method with Anotations, but I didn't find an example on the internet). So the `ValueConverter` is necessary in the class name, but by calling the converter, this isn't necessary anymore, therefor you need to call the converter only with `toUppercase`in the html template.

Creating A Bi-directional Value Converter

A bi-directional value converter utilizes two methods in your Value Converter class: `toView` and `fromView` these methods are aptly named to signify which direction the data is flowing.

In our example we will be creating a prepend Value Converter which will make sure that an amount entered in our app has a dollar sign infront of it.

prepend.js

```
export class PrependValueConverter {

    /**
     * This is the value being sent back to the view
     *
     */
    toView(value) {
        return `$${value}`;
    }

    /**
     * Validate the user entered value
     * and round it to two decimal places
     *
     */
    fromView(value) {
        return parseFloat(value.toString().replace('$', '')).toFixed(2);
    }
}
```

Using it:

```
export class MyViewModel {
    amount = '';
}
```

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

    <h1>Current amount: ${amount}</h1>

    <label>Enter amount:</label>
    <input type="text" id="amount" value.bind="amount | prepend & debounce:500">
</template>
```

Worth noting is that we are using a binding behaviour to limit the rate of which our value is updated or it will be updated every time you type and not be the intended behaviour.

Chaining Value Converters

A Value Converter can be used alongside other value converters and you can infinitely chain them using the `|` pipe separator.

```
${myString | toUppercase | removeCharacters:'&,%,-,+' | limitTo:25}
```

The above theoretical example firstly applies `toUppercase` which capitalizes our string. Then it applies the `removeCharacters` Value Converter which allows us to remove specific characters from our string and lastly we limit the length of the string to 25 characters using `limitTo`.

Note: the above Value Converters do not actually exist. They are purely for example purposes only.

Read Value Converters online: https://riptutorial.com/aurelia/topic/1917/value-converters

Chapter 9: Working with

Examples

Compose with View only

Presumably the simplest way to use `compose` is with a View only. This allows you to include HTML templates without the need to declare a ViewModel with bindable properties for each of them, making it easier to reuse smaller pieces of HTML.

The BindingContext (ViewModel) of the View will be set to that of the parent ViewModel.

Usage:

src/app.html

```
<template>
  <compose view="./greeter.html"></compose>
</template>
```

src/greeter.html

```
<template>
    <h1>Hello, ${name}!</h1>
</template>
```

src/app.ts

```
export class App {
  /* This property is directly available to the child view
     because it does not have its own ViewModel */
  name = "Rob";
}
```

Compose with View, ViewModel and Model

Using `compose` with a View, ViewModel and Model is an easy way to reuse and combine different Views and ViewModels.

Given the following View and ViewModel (applies to each alternative below):

src/greeter.html

```
<template>
    <h1>Hello, ${name}!</h1>
</template>
```

src/greeter.ts

```
export class Greeter {
    name: string;

    /* The object that is bound to the model property of the compose element,
       will be passed into the ViewModel's activate method
    */
    activate(model) {
        this.name = model.name;
    }
}
```

1 - Basic usage, inline model:

src/app.html

```
<template>
  <compose view="./greeter.html" view-model="./greeter" model="{name: 'Rob'}"></compose>
</template>
```

src/app.ts

```
export class App {
}
```

2 - Basic usage, databound model:

src/app.html

```
<template>
  <compose view="./greeter.html" view-model="./greeter" model.bind="model"></compose>
</template>
```

src/app.ts

```
export class App {
  model = {
    name: Rob"
  };
}
```

3 - Updating the DOM when model properties change:

The only drawback of the second approach is that the `model`'s properties are not observed, so any changes them will not propagate to the DOM.

One way to overcome this is by making sure the `model` property itself changes whenever any of its properties change. This will cause the `compose` element to be re-compiled:

src/app.html

```
<template>
  <compose view="./greeter.html" view-model="./greeter" model.bind="model"></compose>
  <input type="text" value.two-way="name">
</template>
```

src/app.ts

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

export class App {
  name: string;

  @computedFrom("name")
  get model() {
    return {
      name: this.name
    };
  }

  /* Using computedFrom prevents "dirty checking" and is optional,
     but highly recommended for performance reasons.

     Simply pass an array with names of all properties you wish to "observe".
     Expressions / nested properties are not supported.
  */
}
```

Read Working with online: https://riptutorial.com/aurelia/topic/7282/working-with--compose-

Credits

S. No	Chapters	Contributors
1	Getting started with aurelia	Community, Dwayne Charrington, UTAlan
2	Aurelia CLI	Daryl Cober, Dwayne Charrington, kettch, zewa666
3	Aurelia CLI Explained	Jeremy Danyow, mauricio777, RamenChef
4	Binding	Dwayne Charrington
5	Custom Elements	Dwayne Charrington, Marc Scheib
6	Dependency Injection	UTAlan
7	Templating	Dwayne Charrington
8	Value Converters	Dwayne Charrington, EagleT, Eliran Malka
9	Working with	Fred Kleuver