

# Angular

Wael Kdouh

Email: Wael.Kdouh@Microsoft.com

Twitter: @waelkdouh

Senior Consultant

Premier Developer



# Agenda

- Set Up The Environment
- Planning The Application Architecture
- Modules
- Components
  - Templates
  - Classes
- Data Binding
- Pipes
- Directives
- Services

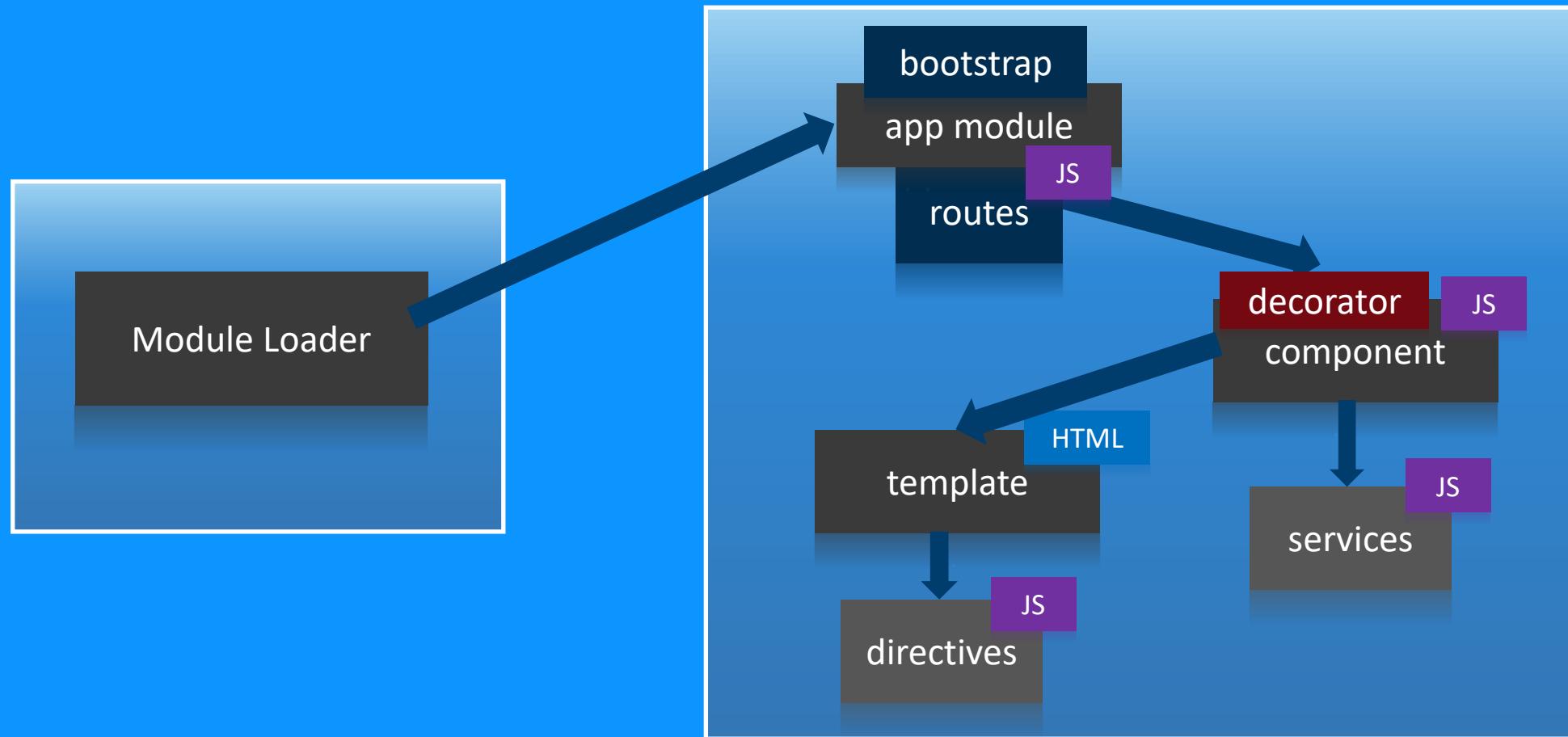
# Agenda

- Reactive Programming
- Routing
- Forms
- Lazy Loading Modules
- State Management With The Redux Pattern
- AOT Compilation
- CLI Environment Options
- Debugging
- Testing

# Agenda (Optional)

- Containerization
- Progressive Web Apps (PWA)
- Angular Libraries

# Angular - The Big Picture



# Let Us Talk About Angular Versions

- Angular implements semantic versioning
- The version of Angular doesn't matter anymore beyond bug fixing purposes. The framework should simply be called Angular

# Set Up the Environment

- Pick the language
  - JavaScript - ES5 (supported in most browsers)
  - JavaScript - ES6, now called ES2015 (must be transpiled)
  - TypeScript – superset of JavaScript (must be transpiled)
- Pick an editor
  - Visual Studio
  - Visual Studio Code
  - and more...

# Set Up the Environment

1. Install NodeJS/NPM
2. Install the Angular CLI
3. Install VS Code – Make sure to select the context menu option to open with VS Code
4. Install the Git Client
5. Setup up your application

# Set Up the Environment



The Node.js homepage features a prominent green gradient bar at the top. On the left side of this bar is the Node.js logo, which includes a white 'node' wordmark above a green hexagon icon with the letters 'js'. To the right of the logo, the text 'Interactive' is written in a large, white, sans-serif font, with 'NORTH AMERICA' in smaller capital letters below it. Underneath that, 'Vancouver, Canada' and 'October 4-6, 2017' are also displayed in white. To the right of the green bar, there is a large black rectangular area containing white text: 'Join us for Node.js Interactive happening in Vancouver, Canada October 4 - 6, 2017'. The background of the entire page is a photograph of a city skyline at sunset or sunrise, with warm orange and yellow light reflecting off the buildings and water.

Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world.

[Download for Windows \(x64\)](#)

[v6.10.3 LTS](#)  
Recommended For Most Users

[v8.0.0 Current](#)  
Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)      [Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [LTS schedule](#).

# Set Up the Environment

## Download Visual Studio Code

Free and open source. Integrated Git, debugging and extensions.



↓ Windows

Windows 7, 8, 10

.zip



↓ .deb

Debian, Ubuntu

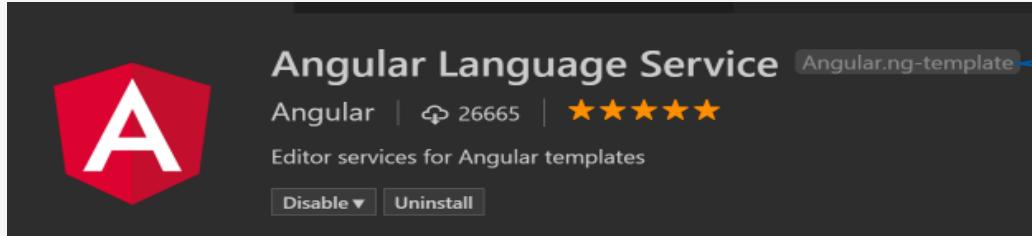
.tar.gz | 32 bit versions



↓ Mac

macOS 10.9+

By downloading and using Visual Studio Code, you agree to the [license terms](#) and [privacy statement](#).



**Tip**  
Recommended  
Exensiontion



# Install Git Client

[About](#)

[Documentation](#)

[Blog](#)

[Downloads](#)

GUI Clients

Logos

[Community](#)

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

## Downloads



[Mac OS X](#)



[Windows](#)



[Linux](#)



[Solaris](#)

Older releases are available and the [Git source repository](#) is on GitHub.

### GUI Clients

Git comes with built-in GUI tools ([git-gui](#), [gitk](#)), but there are several third-party tools for users looking for a platform-specific experience.

[View GUI Clients →](#)

### Git via Git

If you already have Git installed, you can get the latest development version via Git itself:

```
git clone https://github.com/git/git
```

You can also always browse the current contents of the git repository using the [web interface](#).

Latest source Release

**2.14.2**

[Release Notes \(2017-09-22\)](#)

[Downloads for Windows](#)



### Logos

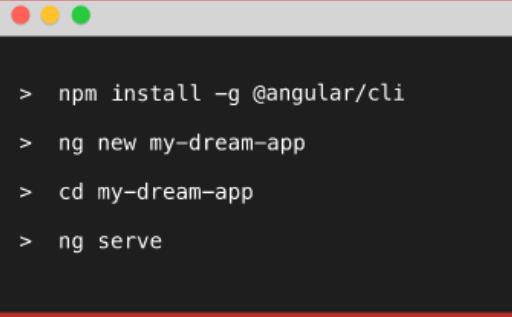
Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

[View Logos →](#)

# Set Up the Environment

Secure | <https://cli.angular.io>

 ANGULAR CLI DOCUMENTATION GITHUB



```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

## Angular CLI

A command line interface for Angular

[GET STARTED](#)

### ng new

The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows our best practices!

### ng generate

Generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these.

### ng serve

Easily test your app locally while developing.

# Set Up the Environment



Angular Console

Download   About   Features

## The Power of the Angular CLI. The Convenience of an App.

Spend less time looking up command line arguments, and more time shipping incredible products.

[Download Angular Console](#)



# Angular Console



# Lab 1

Set Up the Project

# Angular Must Have Tools

## 7 Angular Tools That You Should Consider

# VS Code Must Have Extensions

[VS Code Must have Extensions](#) – Part 1

[VS Code Must have Extensions](#) – Part 2

# Run Angular CLI Repos Directly In Your Browser

[StackBlitz](#)



# Modules

- They each have their own way
  - Angular 1 Modules (defined using JavaScript)
  - TypeScript Modules (defined in TypeScript)
  - ES2015 Modules (a file is a module)
- Angular uses ES2015 Modules to load code modules into JavaScript
- Angular itself contains Modules called `@NgModule` which consolidates components, directives and pipes into cohesive blocks of functionality... Modules can also add services

# Modules

Why do we need a new module system at all? Why can't we just use ES2015/TypeScript modules?

# ES 2015 Modules

book.ts

```
export class Book {  
}
```

library.ts

```
import { Book } from  
'./book'
```



Demo

ES2015 Modules

# Angular Modules

- `@NgModule` determines what tags are compiled and what dependencies should be injected
- Angular needs to know what components define valid tags and where dependencies are coming from

# What Does An Angular Module look Like?

@NgModule annotation is what actually defines the module

We can import other modules by listing them in the imports array

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 ...
5
6 @NgModule({
7   declarations: [AppComponent, MyComboboxComponent,
8                 CollapsibleDirective, CustomCurrencyPipe],
9   imports: [BrowserModule],
10  providers: [UserService, LessonsService]
11 })
12 export class ExampleModule {
13
14 }
```

We can list the components, directives and pipes that are part of the module in the declarations array

We can list the services that are part of the module in the providers array

# Angular Modules

- This declarative grouping is useful if for nothing else for organizing our view of the application and documenting which functionality is closely related
- But Angular Modules are more than just documentation

# Angular Modules

- An Angular module allows Angular to define a context for compiling templates
- For example when Angular is parsing HTML templates, it's looking for a certain list of components, directives and pipes
- Each HTML tag is compared to that list to see if a component should be applied on top of it, and the same goes for each attribute. The problem is: how does Angular know which components, directives and pipes should it be looking for while parsing the HTML?

# Angular Modules

- This is where Angular modules come in. They provide that exact information in a single place

# Angular Modules

The root module has the conventional name of AppModule

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 ...
5
6 @NgModule({
7   declarations: [AppComponent, MyComboboxComponent,
8                 CollapsibleDirective, CustomCurrencyPipe],
9   imports: [BrowserModule],
10  providers: [UserService, LessonsService],
11  bootstrap: [AppComponent]
12})
13export class AppModule {
```

The bootstrap property is used, providing a list of components that should be used as bootstrap entry points for the application. There is usually only one element in this array: the root component of the application

The root module in the case of web applications imports the BrowserModule, which for example provides Browser specific renderers, and installs core directives like ngIf, ngFor, etc.

# Angular Modules

We can see that this module can quickly grow to contain large arrays as the application grows

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 ...
5
6 @NgModule({
7   declarations: [AppComponent, MyComboboxComponent,
8                 CollapsibleDirective, CustomCurrencyPipe],
9   imports: [BrowserModule],
10  providers: [UserService, LessonsService],
11  bootstrap: [AppComponent]
12 })
13 export class AppModule {
14
15
16 }
17
```

# Angular Modules and Visibility

```
import {Home} from "./home.component";  
  
@NgModule({  
    declarations: [Home]  
})  
export class HomeModule {  
  
}
```

Define a separate module with only one component called Home

```
import {HomeModule} from "./home.module";  
  
@NgModule({  
    declarations: [...],  
    imports: [BrowserModule, HomeModule],  
    providers: [UserService, LessonsService],  
    bootstrap: [AppComponent]  
})  
export class AppModule {  
  
}
```

Let's now try to use in our root module, by importing it

# Angular Modules and Visibility

```
import {Home} from "./home.component";  
  
@NgModule({  
    declarations: [Home]  
})  
export class HomeModule {  
}
```

```
import {HomeModule} from "./home.module";  
  
@NgModule({  
    declarations: [...],  
    imports: [BrowserModule, HomeModule],  
    providers: [UserService, LessonsService],  
    bootstrap: [AppComponent]  
})  
export class AppModule {  
}
```

You might be surprised to find out that this does not work. Even if you use the <home></home> component in your template, nothing will get rendered

# Angular Modules and Visibility

```
import {Home} from "./home.component";

@NgModule({
  declarations: [Home]
})
export class HomeModule {
```

```
import {HomeModule} from "./home.module";

@NgModule({
  declarations: [...],
  imports: [BrowserModule, HomeModule],
  providers: [UserService, LessonsService],
  bootstrap: [AppComponent]
})
export class AppModule {
```

Adding Home to the declarations of HomeModule does not automatically make the component visible to any other modules that might be importing it  
This is because the Home Component might be just an implementation detail of the module that we don't want to make publicly available

# Angular Modules and Visibility

```
@NgModule({  
    declarations: [Home],  
    exports: [Home]  
})  
export class HomeModule {  
}
```

To make it publicly available, we need to export it

```
import {HomeModule} from "./home.module";  
  
@NgModule({  
    declarations: [...],  
    imports: [BrowserModule, HomeModule],  
    providers: [UserService, LessonsService],  
    bootstrap: [AppComponent]  
})  
export class AppModule {  
}
```

With this, the Home component would now be correctly rendered in any template that uses the home HTML tag

# Angular Modules and Visibility

```
@NgModule({  
    declarations: [Home],  
    exports: [Home]  
})  
export class HomeModule {  
}
```

Notice that we could also have only exported it without adding it to declarations. This would happen in the case where the component is not used internally inside the module

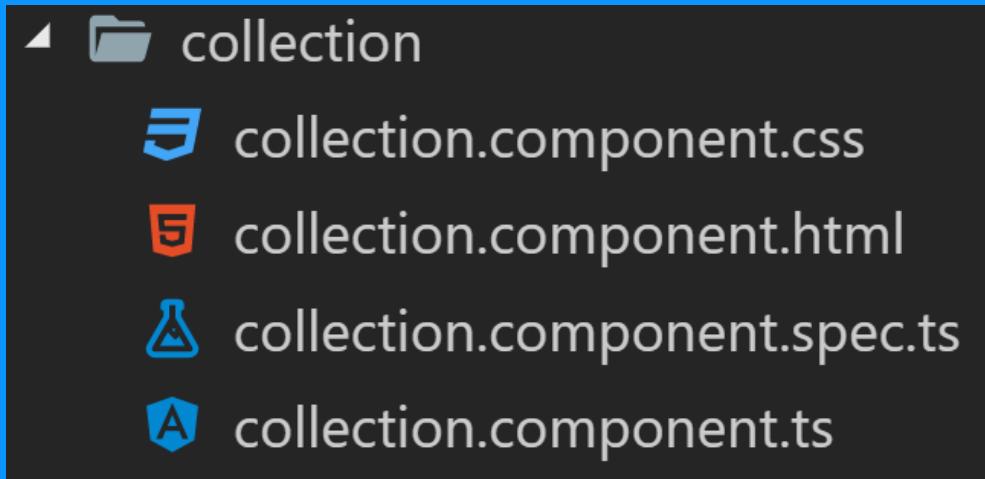
# Feature Modules

- A feature module is meant to extend the global application
- Create an NgModule for all distinct features in an application; for example, a library feature
- A feature module can expose or hide its implementation from other modules
- A feature module identifies distinct sets of related components that comprise the feature area

# Feature Modules

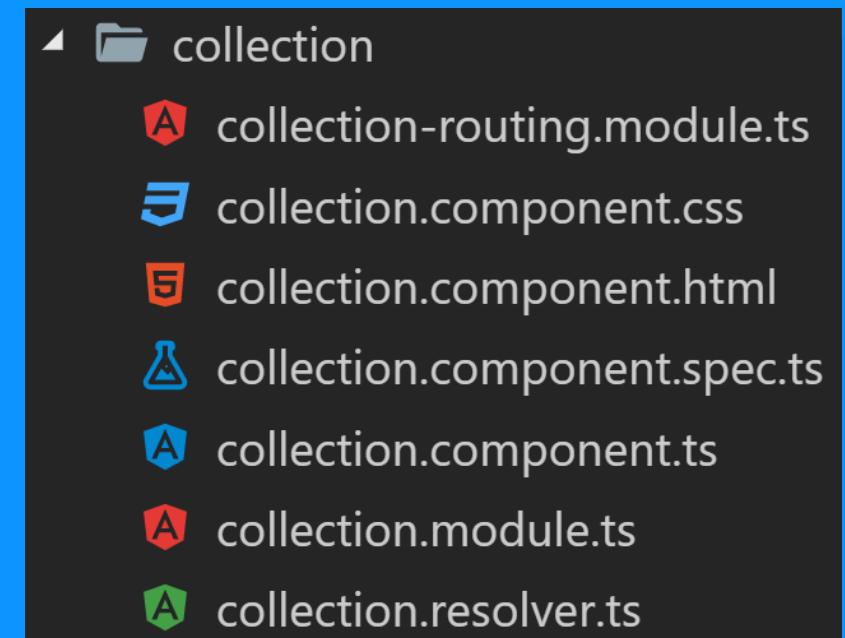
- A feature module can easily be routed to both eagerly and lazily

Collection created inside of the root Module



Collection component added to the  
app.module.ts (aka root module)

Collection created as a Feature Module



Collection component added to the  
collection.module.ts

# Feature Modules

- Each module needs to define separately what it can 'see' in its context
- If we try to run this we will run into the following error message:
  - Unhandled Promise rejection: Template parse errors: Can't bind to 'ngStyle' since it isn't a known property of 'h3'.

```
1 @NgModule({  
2   declarations: [Home],  
3   exports: [Home],  
4   providers: [LessonsService]  
5 })  
6 export class HomeModule {  
7 }  
8  
9
```

```
1 @Component({  
2   selector: 'home',  
3   template: `<h2>Home works !</h2>  
4             <h3 [ngStyle]="{{color:'red'}}">Lessons Service Id = {{lessonsService.id}}</h3>  
5 `)  
6 export class Home {  
7   ...  
8 }  
9
```

# Feature Modules

- But `ngStyle` was already imported into the application via `BrowserModule`, which includes `CommonModule` where `ngStyle` is defined. So why does this work in the application component but not in the Home component?

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 ...
5
6 @NgModule({
7   declarations: [AppComponent, MyComboboxComponent,
8                 CollapsibleDirective, CustomCurrencyPipe],
9   imports: [BrowserModule],
10  providers: [UserService, LessonsService]
11 })
12 export class ExampleModule {
13
14 }
15
```

HomeModule would be inserted here

# Feature Modules

- This is because HomeModule did not itself import CommonModule where the core ngStyle directive is defined
- Each module needs to define separately what it can ‘see’ in its context, so the solution here is to import the CommonModule

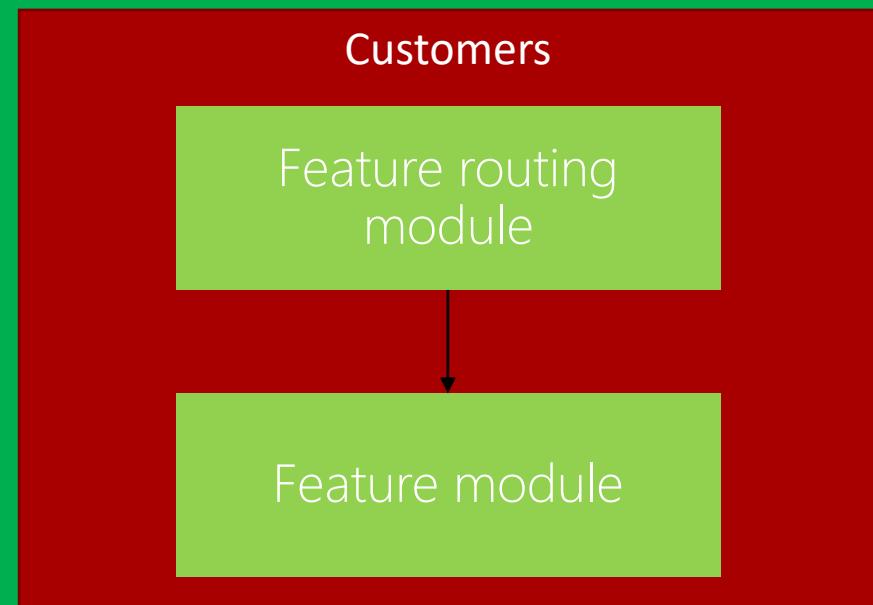
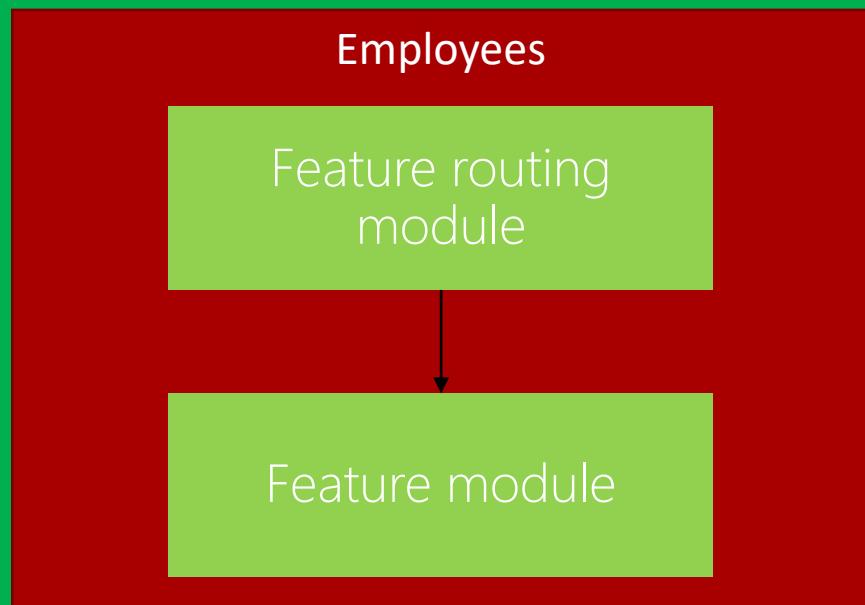
```
1 @NgModule({  
2   declarations: [Home],  
3   exports: [Home],  
4   providers: [LessonsService]  
5 })  
6 export class HomeModule {  
7 }  
8  
9
```

```
1 import {CommonModule} from "@angular/common";  
2  
3 @NgModule({  
4   imports: [CommonModule],  
5   declarations: [Home],  
6   exports: [Home],  
7   providers: [LessonsService]  
8 })  
9 export class HomeModule {  
10   ...  
11 }  
12
```

# Architectural Tip #1 – Use Feature Modules

- Avoid adding all your components into the app.module (aka root module)
- Instead distribute your components into feature modules as this would allow different team members to be responsible for different features

# Architectural Tip #1 – Use Feature Modules



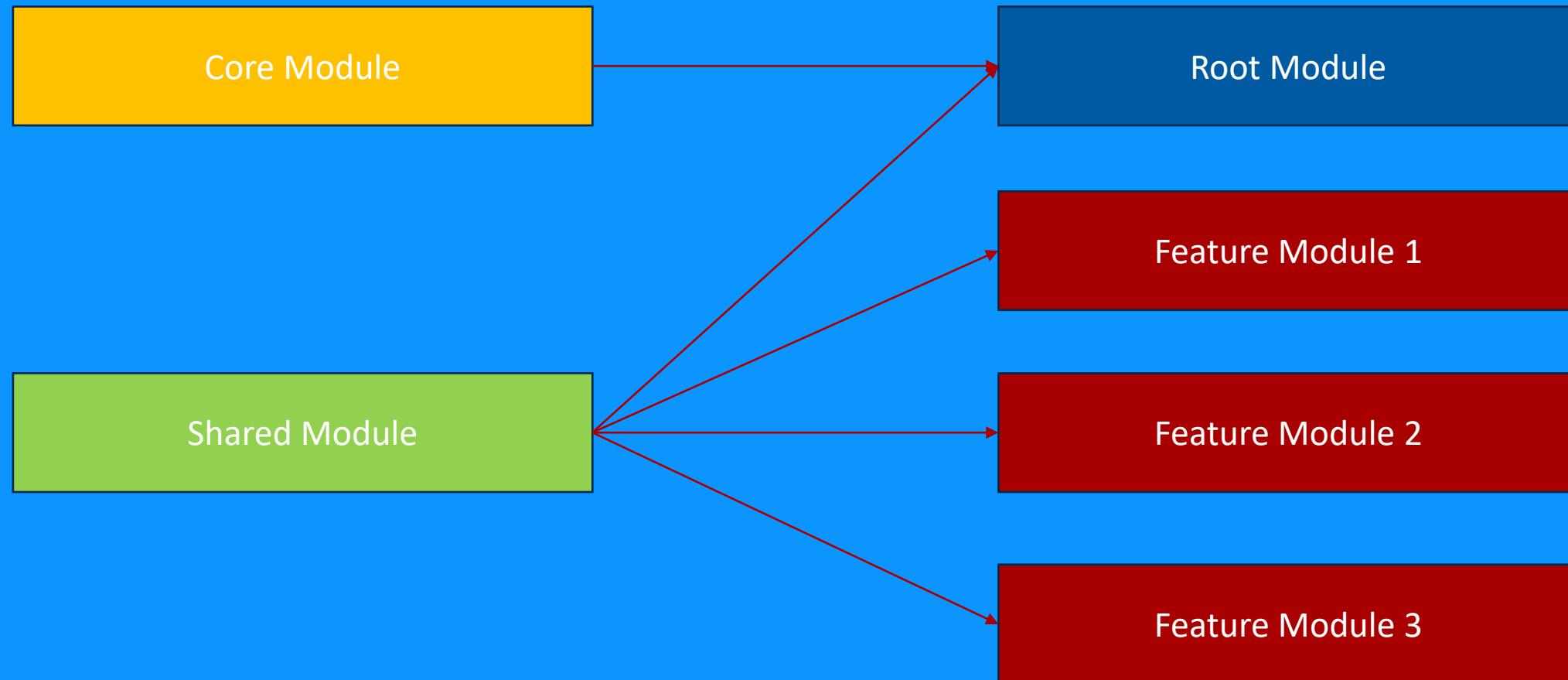
# Core Modules

- Core folder should include singleton services shared across the whole application
- There are exceptions to this rule where a singleton service is used by one feature. For example in our application the DataService could be included in the “collection” feature folder

# Shared Modules

- Shared folder should include reusable components, pipes, and directives. For example RatingComponent

# Importing Core and Shared Modules



# Architectural Tip #2 – Use Core Modules

- Allows you to reuse services across different modules instead of having to inject it to each feature module

# Architectural Tip #3 – Use Shared Modules

- Allows you to reuse components/directives/pipes across different modules instead of having to import them inside each feature module

# Application Architecture

```
▲ ANGULARWORKSHOP
  ▲ src
    ▲ app
      ▷ about
      ▷ book-detail
      ▷ collection
      ▷ guards
      ▷ new-book
      ▷ newsletter
      ▷ pipes
      ▷ rating
      ▷ services
      ▷ tabs
      A app-routing.module.ts
      E app.component.css
      S app.component.html
      L app.component.spec.ts
      A app.component.ts
      A app.module.ts
      TS ibook.ts
      TS inavlink.ts
```

For simplicity, the app we will be building places all the components in the root module

```
▲ ANGULARWORKSHOP
  ▲ src
    ▲ app
      ▷ about
      ▷ collection
        ▷ book-detail
        ▷ guards
        ▷ new-book
        ▷ pipes
        ▷ rating
        ▷ services
          A collection-routing.module.ts
          E collection.component.css
          S collection.component.html
          L collection.component.spec.ts
          A collection.component.ts
          A collection.module.ts
          G collection.resolver.ts
      ▷ newsletter
      ▷ tabs
      A app-routing.module.ts
      E app.component.css
      S app.component.html
      L app.component.spec.ts
      A app.component.ts
      A app.module.ts
      TS ibook.ts
      TS inavlink.ts
```

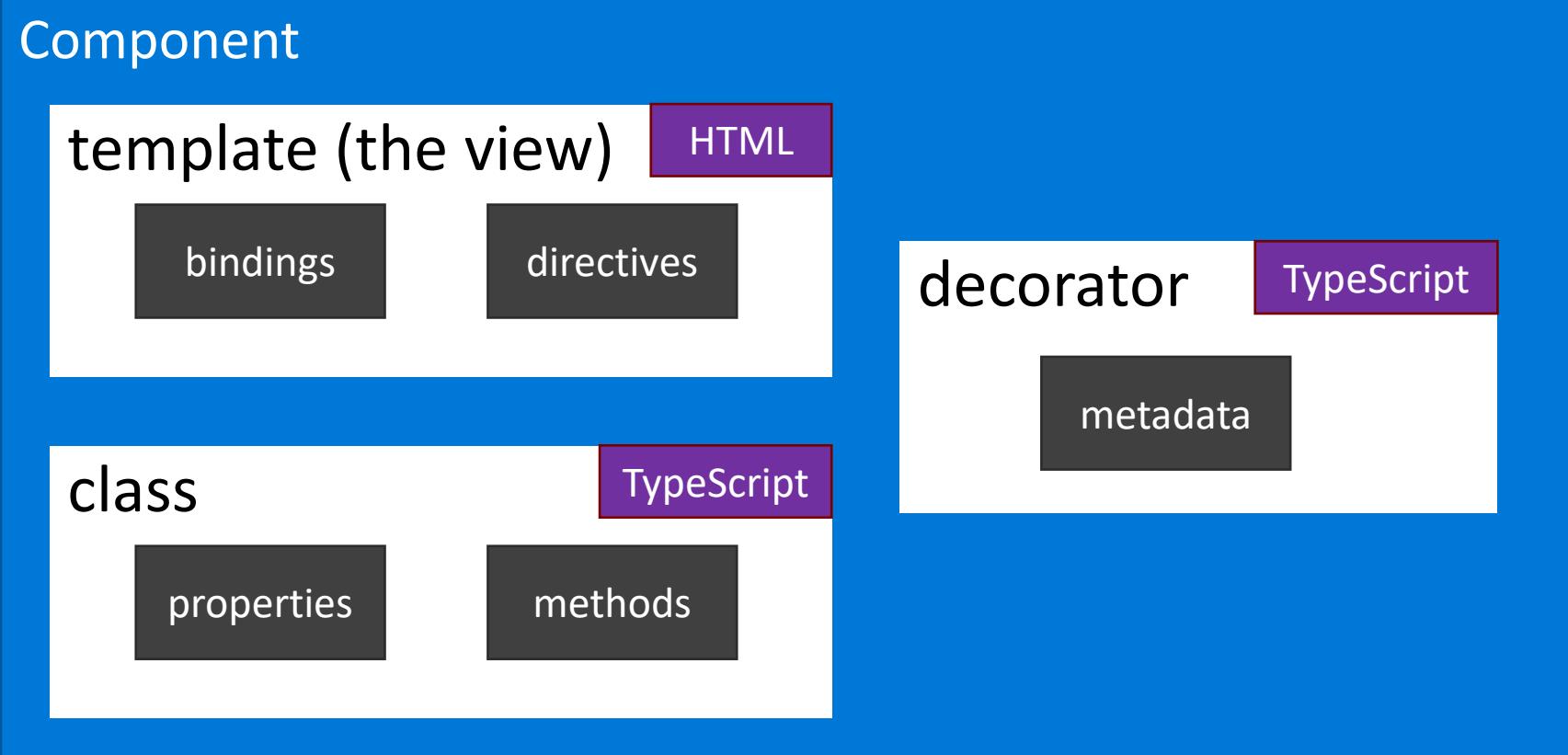
For production applications utilize core/shared/feature modules



# Components

- Represent the visual parts of the UI
- Contain templates
- Attach metadata using decorators

# Components



# Components

```
@Component({  
  templateUrl: 'app/home/about.component.html'  
})  
export class AboutComponent {  
  public pageTitle = "About";  
}
```

Decorator for the class

Binding Expression

```
<div class="panel panel-primary">  
  <div class="panel-heading">  
    {{pageTitle}}  
  </div>  
  <div class="panel-body">  
  </div>  
</div>
```

# Components

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  templateUrl: 'app/home/about.component.html',
  styleUrls: ['app/home/about.component.css']
})
export class AboutComponent {
  public pageTitle = "About";
}
```

Include on the page  
using a directive

Directive name

Optional stylesheets

<body>  
 <my-app>Loading...</my-app>  
</body>

# Components

- A component can only belong to one Angular Module
  - When you make a component part of a module you impose on it a set of rules when it is compiled
  - Having a component without belonging to a NgModule is meaningless as the compiler can't compile it
  - Having a component be part of more than one module is also weird as you are saying that depending on which module you choose, the rules for compiling are different. And when you dynamically load such a component it would be ambiguous which set of compilation rules you wanted

# Components

- But what if reusing the same component across different Angular Modules is required?
  - As a workaround you would create a new Angular Module for the component and add the new module to imports: [...] of the other modules where you want to use it

# Templates

- HTML
- Directives
- Bindings

# Templates

Inside a component

```
@Component({  
  selector: 'my-app',  
  template: '<h1>My First Angular App</h1>'  
})  
export class AppComponent {}
```

# Templates

Inside a component using multiline

```
@Component({  
  selector: 'my-app',  
  template: `<h1>My App</h1>  
            <div>  
              <p>Isn't is pretty?</p>  
            </div>`  
})  
export class AppComponent {}
```

# Templates

Containing a directive to another component

```
@Component({  
  selector: 'my-app',  
  template: `<h1>My App</h1>  
            <my-message></my-message>`  
})  
export class AppComponent {}
```

# Templates

External and referenced from a component

```
@Component({  
  selector: 'my-app',  
  templateUrl: 'demo.component.html'  
})  
export class AppComponent {}
```

# Module Loader

Index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>ng-workshop</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <script type="text/javascript" src="https://unpkg.com/@angular/core@6.1.0/bundles/core.umd.js">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

Main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

App.component.html

```
<div class="container">
  <md-toolbar color="warn">
    <span>{{title}}</span>
    <span class="fill-remaining-space"></span>
    
  </md-toolbar>
  <my-tabs></my-tabs>
</div>
```

App.component.ts

```
import { Component } from '@angular/core';
import './rxjs-operators';
import { DataService } from './data.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [DataService]
})
export class AppComponent {
  title = 'Material Library App';
}
```

App.module.ts

```
import { NewBookComponent } from './new-book/new-book.component';

@NgModule({
  declarations: [
    AppComponent,
    AboutComponent,
    TabsComponent,
    CollectionComponent,
    RatingCategoryPipe,
    RatingComponent,
    BookDetailComponent,
    NewBookComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    MaterialModule,
    BrowserAnimationsModule,
    routing
  ],
  entryComponents: [
    BookDetailComponent,
    NewBookComponent
  ],
  providers: [BookDetailGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Demo

Angular Modules

# Lab 2

Template Containing a Component

# Life without Data Binding

- JQuery for everything
- Register callbacks for DOM events
- Code to find elements in the DOM
- Code to manipulate the DOM
- Code to inject data into the DOM

# Data Binding

- Angular's change detection is based on unidirectional data flow
- With the unidirectional data flow you always know where a change to a particular UI element or a component property came from

# Data Binding

- These are advantages of Angular's unidirectional data flow compared to Angular 1:
  - Easier widget integration
  - No more \$apply
  - No more repeated digest cycles
  - No more watchers
  - No more performance issues with digest cycle and watcher limits

# Data Binding



# Interpolation

`{{2 + 2}}`

Expression

`<h1>{{pageTitle}}</h1>`

`{{amount > 0 ? 'Positive' : 'Negative'}}`

`{{calculateResult()}}`

`<span class="{{getClass()}}>`

  Hello

`</span>`

```
export class AppComponent {  
  pageTitle = "My App";  
  amount = 100;  
  calculateResult() { . . . };  
  getClass() { . . . };  
}
```

# One Way (Property) Binding

```
<img [src] = 'item.path' />
```

```
<div [hidden] = 'item.isHidden'>Hello</div>
```

```
<span [innerText] = 'person.name'></span>
```

```
<div [style.color] = 'isOk' ? 'green' : 'red'></div>
```

# Similarities between Interpolation and Property Binding

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{ fullName }}</h1>
  `
})
export class AppComponent {
  fullName: string = 'Martin Luther King Jr';
}
```

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1 [innerHTML]='fullName'></h1>
  `
})
export class AppComponent {
  fullName: string = 'Martin Luther King Jr';
}
```

# Similarities between Interpolation and Property Binding

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{citedExample}}</h1>
    <img src='{{imagePath}}'>
  </div>`
})
export class AppComponent {
  citedExample: string = 'Interpolation';
  imagePath: string = 'https://angular.io/assets/images/logos/angular/logo-nav@2x.pn
}
```

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<div>
    <h1 [innerHTML]='citedExample'></h1>
    <img [src]='imagePath'>
  </div>`
})
export class AppComponent {
  citedExample: string = 'Interpolation';
  imagePath: string = 'https://angular.io/assets/images/logos/angular/logo-nav@2x.pn
}
```

# Differences between Interpolation and Property Binding

- Interpolation is a special syntax that Angular converts into property binding
- It's a convenient alternative to property binding.

# Differences between Interpolation and Property Binding

- Let's make the difference clear with an example: In the example below when we need to concatenate strings we have to use interpolation instead property binding

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{citedExample}}</h1>
    <img src=' https://angular.io/{{imagePath}}' />
  </div>`
})
export class AppComponent {
  citedExample: string = 'Interpolation';
  imagePath: string = 'assets/images/logos/angular/logo-nav@2x.png';
}
```

# Differences between Interpolation and Property Binding

- To set an element property to a non-string data value, you must use property binding. In the example below, we are disabling a button by binding to the Boolean property `isEnabled`.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<div>
    <button [disabled]='isEnabled'>Try Me</button>
    </div>`
})
export class AppComponent {
  isEnabled: boolean = true;
}
```

# Differences between Interpolation and Property Binding

- If we decide to use interpolation instead of property binding, the button will always be disabled irrespective of isDisabled class property value is true or false.property isDisabled.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<div>
    <button disabled='{{isDisabled}}'>Try Me</button>
    </div>`
})
export class AppComponent {
  isDisabled: boolean = true;
}
```

# Interpolation and Property Binding Security

- Looking at both data binding technique from the angle of security, both interpolation and property binding protect us from malicious content, thanks to angular data binding that sanitizes malicious content before displaying it on the browser

# Interpolation and Property Binding Security

- Output: Hello <script>alert("You are just been Hacked or not");</script>

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<div>{{hacked}}</div>'
})
export class AppComponent {
  hacked: string = 'Hello <script>alert("You are just been Hacked or not");</script>'
}
```

# Event Binding

```
<button (click)='saveItem()'>Save</button>  
<input (blur)='saveItem()' />
```



<https://developer.mozilla.org/en-US/docs/Web/Events>

# Two Way Binding

```
<input [(ngModel)]="item.quantity" />
<select [(ngModel)]="customer.level">
  <option value="gold">Gold</option>
  <option value="silver">Silver</option>
</select>
```

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [FormsModule]
})
export class AppModule { }
```



# Two Way Binding

Any directive that is used inside a component template should be part of the same Angular Module

- So where does [(ngModel)] come from?
  - It comes from FormsModule since [(ngModel)] is usually used when building data entry forms

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [FormsModule]
})
export class AppModule { }
```

Demo

Data Binding

Lab 3

Data Binding

# Pipes

- Transform bound properties before display
  - Built-in pipes
    - date
    - number, decimal, percent, currency
    - json, slice
    - Etc.
  - Custom Pipes

# Pipes

```
<p>Score: {{score | percent}}</p>
```

```
{{ startDate | date:'shortDate' }}
```

```
<pre>{{model | json}}</pre>
```

```
{{ total | currency:'USD':true:'1.2-2' }}
```

<https://angular.io/docs/ts/latest/api/#?apiFilter=pipe>

# Custom Pipes

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'colorRange' })
export class ColorRangePipe implements PipeTransform {

  transform(value: number): string {
    if (value <= 25) {
      return 'red';
    }
    if (value <= 75) {
      return 'yellow';
    }
    return 'green';
  }
}
```

Lab 4

Pipes

# Directives

- With Angular directives we add logic to our HTML such as if statements and for loops

# Three Kinds of Directives

- Directives with templates known as *Components*
- Directives that manipulate DOM by changing behavior and appearance known as *Attribute Directives*
- Directives that create and destroy DOM elements known as *Structural Directives*

# Component Directive

- A component is a directive that has an associated template

app.component.ts

```
@Component({
  selector: 'pm-app',
  template:
    <div><h1>{{pageTitle}}</h1>
      <pm-products></pm-products>
    </div>
})
export class AppComponent { }
```

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateUrl:
    'app/products/product-list.component.html'
})
export class ProductListComponent { }
```

# Building Better Components

- Components are one of the key building blocks of an Angular application
- The cleaner the components are the better the application

# How Can We make Our Components Better?

- Strong Typing and Interfaces
- Encapsulating Styles
- Lifecycle Hook

# Strong Typing and Interfaces

- One of the benefits of using TypeScript is strong typing

```
class Greeter {  
    greeting:string; ←  
    products : any [] ←  
    constructor(message: string) {  
        this.greeting = message;  
    } ←  
    greet():string { ←  
        return "Hello, " + this.greeting;  
    } ←  
}
```

- Every property has a type
- Every parameter has a type
- Every method has a return type

In some cases we have a property or a method that doesn't have a predefined type

# Strong Typing and Interfaces

- One of the benefits of using TypeScript is strong typing

```
class Greeter {  
    greeting:string;  
    products : any []  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet():string {  
        return "Hello, " + this.greeting;  
    }  
}
```

Using 'any' negates the  
benefits of strong typing

# Strong Typing and Interfaces

- To specify a custom type we can define an Interface

```
interface Greetable {  
    greet(message: string): void;  
}
```

```
function helloEnglish(g: Greetable) {  
    g.greet('Hello there!'); // OK  
    g.greet(42); // Not OK -- 42 is not a string  
    g.greep('Hi'); // Not OK -- 'greep' is not a member of 'Greetable'  
}
```

# Strong Typing and Interfaces

- Interfaces are only a compile-time construct and have no effect on the generated code
  - ES 2015 does not support it yet

# Encapsulating Styles

```
import { Component } from '@angular/core';

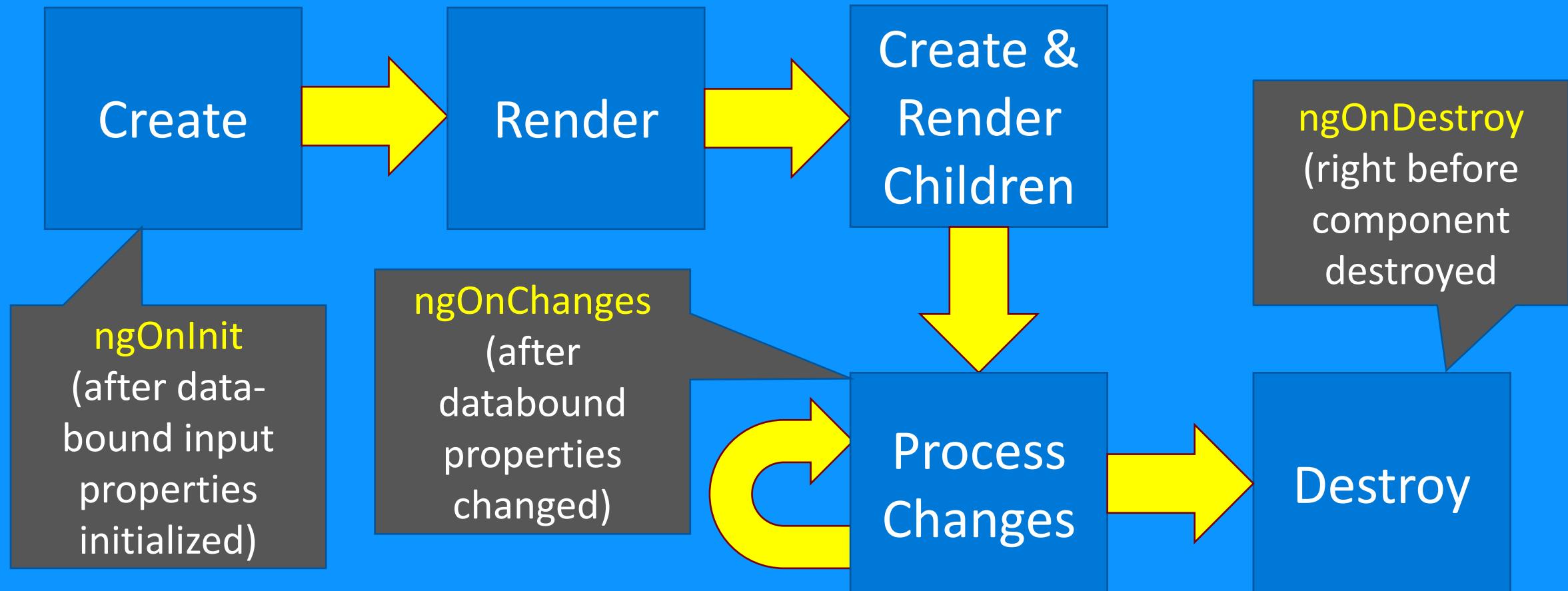
@Component({
  selector: 'my-app',
  templateUrl: 'app/home/about.component.html',
  styleUrls: [ 'app/home/about.component.css' ]
})
export class AboutComponent {
  public pageTitle = "About";
}
```

Encapsulated Styles

# Lifecycle Hooks

- A component lifecycle is managed by Angular

# Component Lifecycle



# Lifecycle Hooks

```
export class App implements OnInit{  
  constructor(){  
    //called first time before the ngOnInit()  
  }  
  ngOnInit(){  
    //called after the constructor and called after the first ngOnChanges()  
  }  
}
```

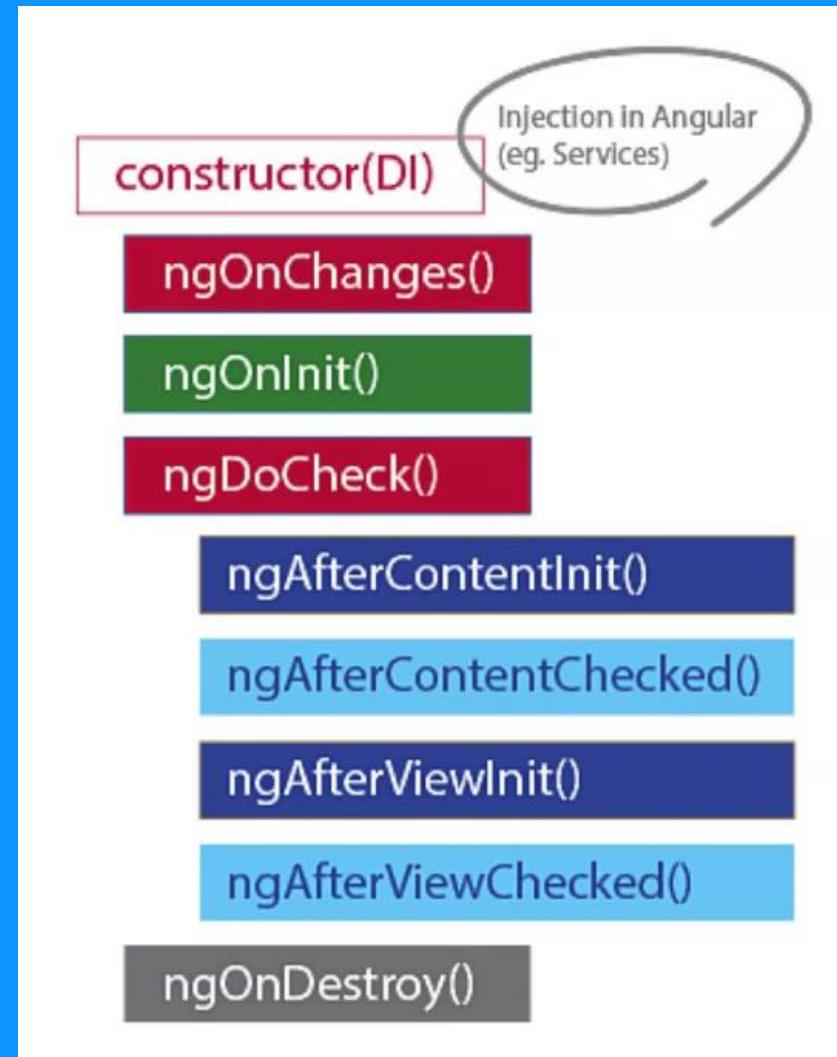
# Constructor VS ngOnInit

Constructor	ngOnInit
Called when component is created	Called when component is created
ES2015/TypeScript Feature	Angular Feature
Out of Angular's control	Under Angular's Control
Not a suitable hook to let you know when Angular has finished initializing the component	Suitable hook to let you know when Angular has finished initializing the component
This is when we want to utilize dependency injection – essentially for “wiring up” dependencies into the component	ngOnInit is purely there to give us a signal that Angular has finished initializing the component. This phase includes the first pass at Change Detection against the properties that we may bind to the component itself – such as using an @Input() decorator. Due to this, the @Input() properties are available inside ngOnInit, however are undefined inside the constructor, by design

# Constructor VS ngOnInit

Constructor	ngOnInit
The constructor should only be used to initialize class members but shouldn't do actual “work”	The bulk of an application’s initialization should go here as there is the certainty that Angular has initialized the component correctly and developers can start doing any logic they need in OnInit versus doing things when your component hasn’t finished loading properly

# Constructor VS ngOnInit



# Structuring Components

Simple Component

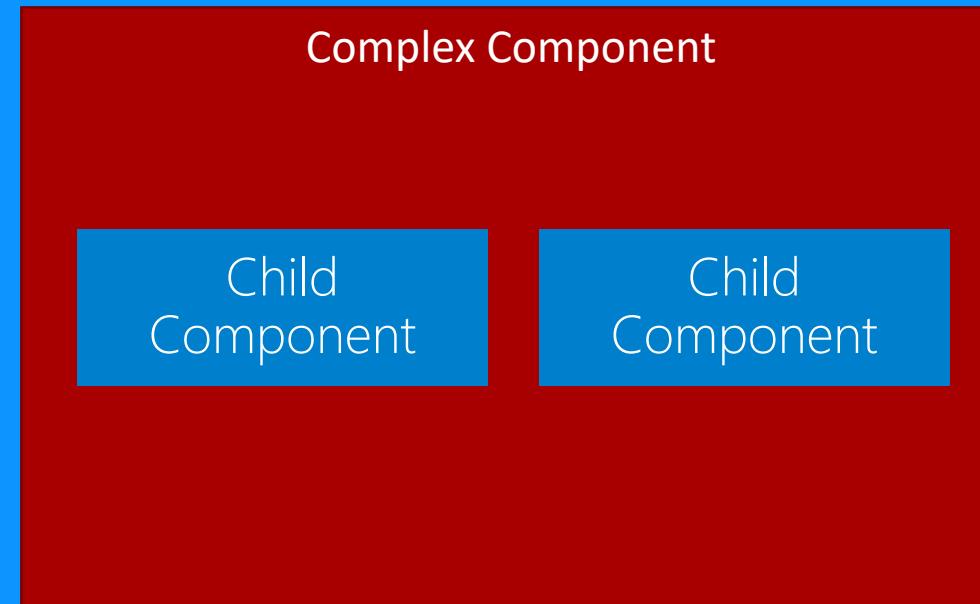
Template

Complex Component

Child  
Component

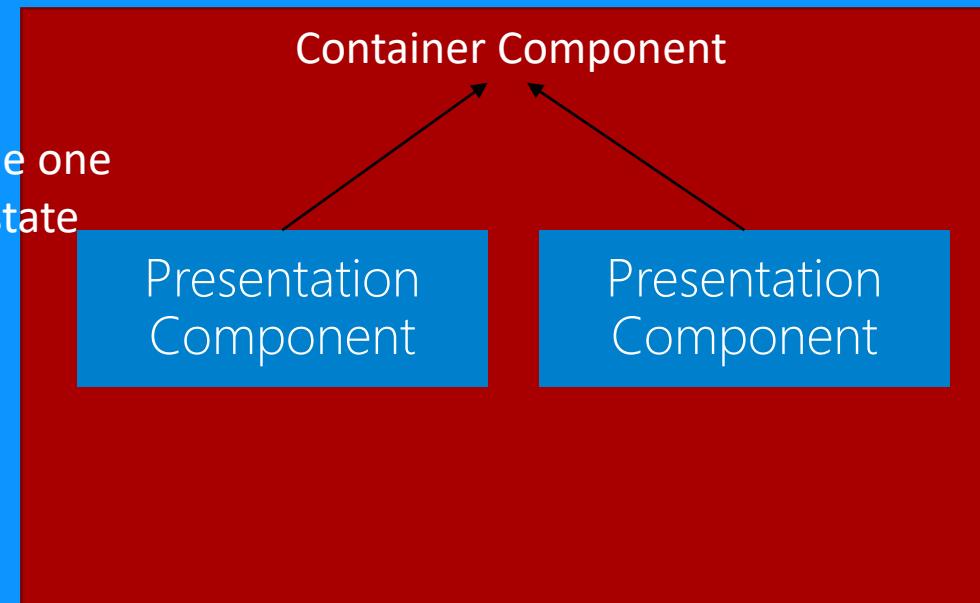
Child  
Component

# Container And Presentation Components



# Container And Presentation Components

- Container component is the one responsible for retrieving state
- Presentation component, presents/render state



# Building Nested Components

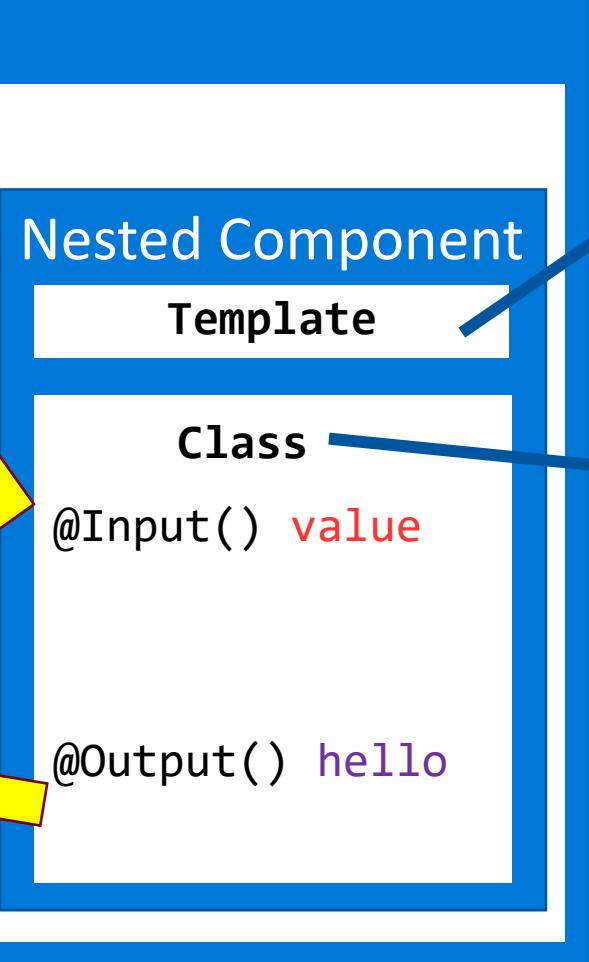
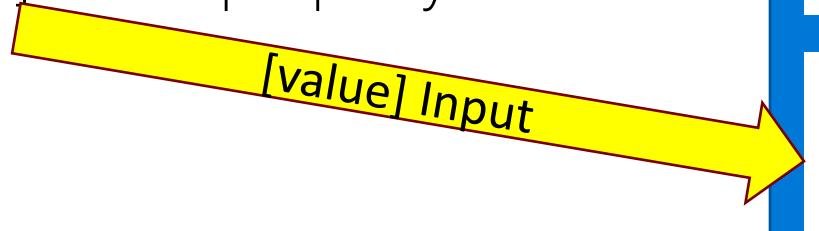
- What makes a component nestable?
  - Its template manages a fragment of a larger view
  - It has a selector
  - It optionally communicates with its parent component (the container)

# Communication Between Parent and Child Components

## Container Component

- Template
- <my-nested-component [value]=someproperty>

```
(hello)='onHello($event)'  
</my-nested-component>
```



```
<div (click)='update()'>  
    ...  
</div>
```

```
@Output() hello:  
EventEmitter<string> = new  
EventEmitter<string>();  
  
update() {  
    this.hello.emit('hi');  
}
```

# Communication Between Parent and Child Components

## Child Component

```
import { Component, OnChanges, Input,
         Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
  styleUrls: ['app/shared/star.component.css']
})
export class StarComponent implements OnChanges {
  @Input() rating: number;
  starWidth: number;
  @Output() ratingClicked: EventEmitter<string> =
    new EventEmitter<string>();

  ngOnChanges(): void {
    // Convert x out of 5 starts
    // to y out of 86px width
    this.starWidth = this.rating * 86 / 5;
  }

  onClick(): void {
    this.ratingClicked.emit(`The rating ${this.rating} was clicked!`);
  }
}
```

```
<div class="crop"
      [style.width.px]="starWidth"
      [title]="rating"
      (click)="onClick()">
  <div style="width: 86px">
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
  </div>
</div>
```

# Communication Between Parent and Child Components

## Child Component

```
import { Component, OnChanges, Input,
         Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
  styleUrls: ['app/shared/star.component.css']
})
export class StarComponent implements OnChanges {
  @Input() rating: number;
  starWidth: number; |----->
  @Output() ratingClicked: EventEmitter<string> =
    new EventEmitter<string>();

  ngOnChanges(): void {
    // Convert x out of 5 starts
    // to y out of 86px width
    this.starWidth = this.rating * 86 / 5;
  }

  onClick(): void {
    this.ratingClicked.emit(`The rating ${this.rating} was clicked!`);
  }
}
```

Child Component  
Watches for  
changes to input  
properties

```
<div class="crop"
      [style.width.px]="starWidth"
      [title]="rating"
      (click)="onClick()">
  <div style="width: 86px">
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
  </div>
</div>
```

## Parent Component Template

```
<td>{{ product.productCode | lowercase }}</td>
<td>{{ product.releaseDate }}</td>
<td>{{ product.price | currency:'USD':true:'1.2-2' }}</td>
<td>
  <ai-star>[rating]='product.starRating'
        (ratingClicked)='onRatingClicked($event)'
  </ai-star>
</td>
```

# Communication Between Parent and Child Components

## Child Component

```
import { Component, OnChanges, Input,
         Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
  styleUrls: ['app/shared/star.component.css']
})
export class StarComponent implements OnChanges {
  @Input() rating: number;
  starWidth: number;
  @Output() ratingClicked: EventEmitter<string> =
    new EventEmitter<string>();

  ngOnChanges(): void {
    // Convert x out of 5 starts
    // to y out of 86px width
    this.starWidth = this.rating * 86 / 5;
  }

  onClick(): void {
    this.ratingClicked.emit(`The rating ${this.rating} was clicked!`);
  }
}
```

Child Component triggers changes through the output properties

```
<div class="crop"
      [style.width.px]="starWidth"
      [title]="rating"
      (click)="onClick()">
  <div style="width: 86px">
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
  </div>
</div>
```

## Parent Component Template

```
<td>{{ product.productCode | lowercase }}</td>
<td>{{ product.releaseDate }}</td>
<td>{{ product.price | currency:'USD':true:'1.2-2' }}</td>
<td>
  <ai-star [rating]='product.starRating'
            (ratingClicked)='onRatingClicked($event)'>
  </ai-star>
</td>
```

# Communication Between Parent and Child Components

## Parent Component

```
@Component({
  templateUrl: 'app/products/product-list.component.html',
  styleUrls: ['app/products/product-list.component.css']
})
export class ProductListComponent implements OnInit {
  pageTitle: string = 'Product List';
  imageWidth: number = 50;
  imageMargin: number = 2;
  showImage: boolean = false;
  listFilter: string;
  errorMessage: string;

  products: IProduct[];

  constructor(private _productService: ProductService) {

  }

  toggleImage(): void {
    this.showImage = !this.showImage;
  }

  ngOnInit(): void {
    this._productService.getProducts()
      .subscribe(products => this.products = products,
                error => this.errorMessage = <any>error);
  }
}

onRatingClicked(message: string): void {
  this.pageTitle = 'Product List: ' + message;
}
```

The event triggered from the child component executes the `onRatingClicked` in the parent component. The parameter is passed using the `$event`

```
<td>{{ product.productCode | lowercase }}</td>
<td>{{ product.releaseDate }}</td>
<td>{{ product.price | currency:'USD':true:'1.2-2' }}</td>
<td>
  <ai-star [rating]='product.starRating'
            (ratingClicked)='onRatingClicked($event)'>
  </ai-star>
</td>
```

# Demo

Communication Between Parent and Child Components

# Lab 5

Communication Between Parent and Child Components

# Attribute directives

- Applied as attributes to elements
- They are used to manipulate the DOM in all kinds of different ways except creating or destroying them

# Attribute directives

- They can help us achieve one of the following tasks:
  - **Apply conditional styles and classes to elements**  
`<p [style.color]="blue">Directives are awesome</p>`
  - **Hide and show elements, conditionally (different from creating and destroying elements)**  
`<p [hidden]="shouldHide">Directives are awesome</p>`
  - **Dynamically changing the behavior of a component based on a changing property**

# Custom Attribute directives

- Angular provides clean and simple APIs to create custom directives. Part
- You will find yourself creating more custom attribute directives than structural directives

Demo

Custom Directives

# Structural Directives

- Modifies the layout of the view. Here are two built in structural directives:
  - `*ngIf` – destroys the DOM element whereas **hidden** hides the element
  - `*ngFor`

```
<div class="alert alert-danger" *ngIf='errorMessage'>
    {{errorMessage}}
</div>
<tr *ngFor='let book of bookTitles'>
    <td>{{book}}</td>
</tr>
```

# Structural Directives

- Wait... Didn't we just say that an Angular module defines the boundary or context within which the component resolves its directives and dependencies?

```
<div class="alert alert-danger" *ngIf='errorMessage'>
    {{errorMessage}}
</div>
<tr *ngFor='let book of bookTitles'>
    <td>{{book}}</td>
</tr>
```

# Structural Directives

- How will our application find this `*ngIf` directive?
  - `BrowserModule` exposes these directives

```
12  @NgModule({  
13    imports: [  
14      BrowserModule,
```

```
<div class="alert alert-danger" *ngIf='errorMessage'>  
  {{errorMessage}}  
</div>  
<tr *ngFor='let book of bookTitles'>  
  <td>{{book}}</td>  
</tr>
```

# Services

- Components are great, but what do we do with data or logic that is not associated with a specific view or that we want to share across components?
- We Build Services

# Services

- Two ways to utilize a service from within a component
  - Creating a local instance which makes it hard to test

**Service**

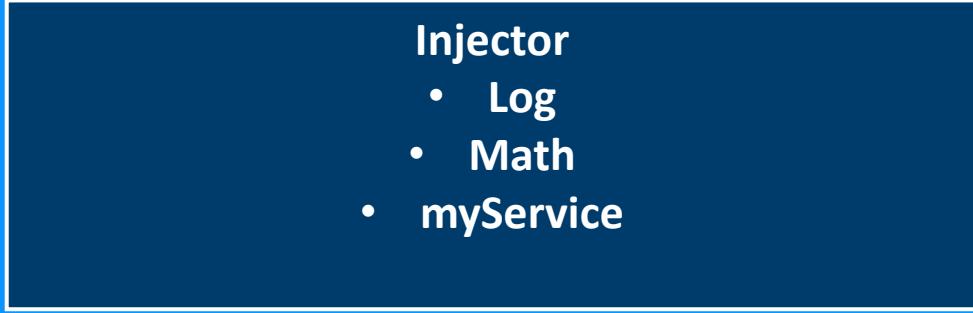
```
export class MyDataService  
{}
```

**Component**

```
Let svc = new  
MyDataService();
```

# Services

- Two ways to utilize a service from within a component
  - Register the service with Angular which creates a single instance of the service class (singleton)



Injector

- Log
- Math
- myService



Service  
`export class MyDataService {}`



Component

# Services

- If you register it twice (in different components) then you get 2 instances (no longer a singleton). The injector will keep track of both.

# Services

## Angular Injector

http

routeParams

log

myDataService

myConstants

Component

Register a provider

## Define your service

```
@Injectable()  
export class MyDataService {  
  getCustomers(): ICustomer[] {...}  
}
```

## Use your service

```
constructor(private _myDataService:  
  MyDataService) {  
}
```

# Services

```
private _myDataService;  
constructor(myDataService : MyDataService) {  
    _myDataService = myDataService;  
}
```

```
constructor(private _myDataService: MyDataService)  
{  
}
```

This is such a common pattern  
that TypeScript defined a  
shorthand syntax for all of  
this code

# Register a Provider

Inside a component

```
import { MyDataService } from './data.service';

@Component({
  selector: 'my-app',
  template: '<h1>My First Angular App</h1> ',
  providers: [myDataService]
})
export class AppComponent {}
```

Could also be added to the  
Angular Module providers if  
needed across all the  
components within an  
Angular Module

# Register a Provider

## Root Injector

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class DataService {

  constructor() { }

}
```

# Lab 6

Build a Service

# AJAX

NOTE: Option to fallback to promises

## Angular 1

\$http – returns a Promise

### Promise

returns a value when:  
done,  
upon error,  
during progress

## Angular

Http – return Observable

### Observable

returns multiple values over time  
can be cancelled  
supports array operators (map,  
filter, etc.)

# Promises VS. Observables

Promises	Observables
Provides a single future value	Emits multiple values over time
Not lazy. By the time you have a promise its on its way to being resolved	Lazy. Observables won't emit values until they are subscribed to
Not cancellable (resolved or rejected and only once)	Cancellable by unsubscribing
	Supports map, filter, reduce, and similar operators

# Observables

- Not supported in JavaScript yet
  - Proposed feature for ES 2016
- Must use Reactive Extensions (RxJS) when using observables with Angular
- RxJS 5 was rewritten to improve performance and also conforms to the ES2016 Observable spec
- RxJs uses the Observer Pattern

# The Observer Pattern

- Let's take a Tv station that is broadcasting and multiple Tvs that are receiving the signal as an example
- Usually there is a so called subject. The subject is the Tv station. On the other hand, there are also observers which are the Tvs in this example

# The Observer Pattern

- To get notified about new data, the observer subscribes to the subject
- One subject can have many subscribed observers
- It is the responsibility of the subject, to publish new data to the clients. Observers can not send any data back. They also don't know about possible other subscribers

# RxJs

- The Rx implementation follows the observer pattern, but some terms and details are different
- First of all, there are two different versions of the 'subject' from the pattern
  - The first one is called **Subject**
  - The second one is called **Observable**

# RxJs

- With the subject, you can subscribe to messages, but also push new data to the stream
- Similar to the television station, but with a key of the front door. You can walk in and broadcast your favorite show whenever you want
- The subject should not be passed around in your application, to avoid confusion where the new data came from. For that, it should be converted to an observable

# RxJs

- On the other hand, there is the observable
- The observable is much more limited. It only allows subscribing to the subject, but is missing the required methods to publish new messages
- It can be passed around in your application, since it does not allow publishing new data

# RxJs

- An observable can be in one of two states. Hot and Cold
- Your observable is hot, if there are actually subscribers subscribed to it
- Otherwise the observable is cold

# RxJs

- Some functions don't do anything if there is no subscriber to the result
- For example the angular HttpClient does not send the request, if you don't subscribe to the result of the request

# RxJs

- The subscribe() method returns a subscription
- Make sure to unsubscribe from that at some point if you no longer need it. Otherwise you create immortal objects and memory leaks

```
import { Subject } from "rxjs/Subject";
import { Observable } from "rxjs/Observable";
import { Subscription } from "rxjs/Subscription";

private subscription: Subscription
ngOnInit(){
  this.subscription = new Subject().subscribe();
}

ngOnDestroy(){
  this.subscription.unsubscribe();
}
```

# RxJs

- RxJs also provides some operators that can be used to modify observable streams
- These operators are where it can get very complex quickly
- You don't have to use them all and in most cases you won't

# RxJs

- Examples of useful operators
- map: You provide it with a function, that gets applied to all elements that enter the stream, before they reach the subscriber

```
let subject = new Subject<string>();

subject.map(value => value + "Appended by the mapping operator").subscribe(
  value => {
    // value is the value of the received data
    console.log(value);
  },
  e => {
    console.error(e)
  },
  () =>{
    console.log('complete')
  }
);

subject.next('String to be broadcasted to subscribers...');

subject.complete();
```

# RxJs

- **filter**: Filter the elements of a stream using a condition. If the condition returns false, the element is not sent to the subscribers

```
let subject = new Subject<string>();

subject.map(value => value + "Appended by the mapping operator")
    .filter(value => {
        return value.endsWith('...') ? true: false;
    })
    .subscribe(
        value => {
            // value is the value of the received data
            console.log(value);
        },
        e => {
            console.error(e)
        },
        () =>{
            console.log('complete')
        }
    );
    subject.next('String to be broadcasted to subscribers...');
    subject.complete();
```

# Demo

RxJS Subject vs Observable  
RxJS Operators  
RxJS Observables vs Promises

# Lab 7

Http Service

# Reactive Programming

- Programming with focus on data streams
  - Instead of reading into an array and then binding to the array we work with observables and then bind to these observables
- By working with observables we can leverage the extensive list of operators available with RxJS
  - We transform, compose, and combine streams to better manage multiple data sources
- We define actions streams to easily react to user or application actions

# Async Pipe

- Subscribes to the Observable when component is initialized
- Returns each emitted value
- When a new item is emitted, component is marked to be checked for changes
- Unsubscribes when component is destroyed. No need to explicitly unsubscribe inside the ngOnDestroy
- Since the async pipe handles our subscription we no longer need to subscribe in the subscription code
- To use async pipe to our service code. Only our component and template needs to be

# Async Pipe

- There are several benefits of writing reactive templates:
  - **Memory leaks are less likely (depending on the type of Observables we use)**
  - **More readable code**
  - **Less potential issues with multiple subscriptions at the level of the service layer**
  - **Less state in our components**

# Async Pipe

## Non Reactive Programming

Recommended pattern to append observable properties with the \$ to make them easier to spot in your code.

## Reactive Programming

```
16 export class CollectionComponent implements OnInit {  
17   pageTitle: string;  
18   showOperatingHours: boolean;  
19   openingTime: Date;  
20   closingTime: Date;  
21   searchTerm$ = new Subject<string>();  
22 +  books$: Observable<Ibook[]>;  
23  
24 +  constructor(private _snackBar: MatSnackBar, private _dataService: DataSe  
● 25    this.openingTime = new Date();  
26    this.openingTime.setHours(10, 0);  
27    this.closingTime = new Date();  
28    this.closingTime.setHours(15, 0);  
29 }  
30  
31 ngOnInit() {  
32 +  this.books$ = this._dataService.getBooks()  
33 +    .pipe(  
34 +      catchError(error => {  
35 +        this.updateMessage(<any>error, 'ERROR');  
36 +        return EmptyError;  
37 +      })  
38 +    );  
39 +  
40   this._dataService.search(this.searchTerm$)  
41     .subscribe(books => {  
42 +       this.books$ = of(books);  
43     });  
44 }
```



# Async Pipe

## Non Reactive Programming

```
src > app > collection > collection.component.html > div > div > mat-list > mat-list-item
10 |   <mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime | date:'shortTime'}} (M-F)</mat-card-content>
11 |   </mat-card>
12 | </div>
13 | <div>
14 |   Search Books By Title or Author:
15 |   <input (keyup)="searchTerm$.next($event.target.value)">
16 |   <mat-list>
17 |     <mat-list-item *ngFor="let book of books">
18 |       <mat-icon mat-list-icon>book</mat-icon>
19 |       <h3 mat-line>
20 |         <strong>{{book.title}}</strong>
21 |       </h3>
22 |       <div>
23 |         <button mat-button (click)="openDialog(book.id)">
24 |           <i class="material-icons">pageview</i>
25 |           Dialog
26 |         </button>
27 |         <button mat-button (click)="openRoute(book.id)">
28 |           <i class="material-icons"> pageview</i>
29 |           Route
30 |         </button>
31 |         <button mat-button (click)="delete(book)">
32 |           <i class="material-icons">delete_forever</i>
33 |           Delete
34 |         </button>
35 |       </div>
36 |       <p mat-line>
37 |         <span>{{book.author}}</span>
38 |       </p>
39 |       <p mat-line>
40 |         <my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">
41 |         </my-rating>
42 |       </p>
43 |       <p mat-line>
44 |         <span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Chec
45 |           </p>
46 |         </mat-list-item>
47 |       </mat-list>
48 |       <div class="text-right add-btn">
49 |         <button mat-raised-button color="primary" (click)="addBook()">
50 |           <i class="material-icons">add_box</i> ADD BOOK</button>
51 |         </div>
52 |       </div>
```

## Reactive Programming (Reactive Templates)

```
src > app > collection > collection.component.html > div > div > mat-list > mat-list-item
10 |   <mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime | date:'shortTime'}} (M-F)</mat-card-content>
11 |   </mat-card>
12 | </div>
13 | <div>
14 +   <div *ngIf="books$ | async as books; else nobooks">
15 |   Search Books By Title or Author:
16 |   <input (keyup)="searchTerm$.next($event.target.value)">
17 |   <mat-list>
18 |     <mat-list-item *ngFor="let book of books">
19 |       <mat-icon mat-list-icon>book</mat-icon>
20 |       You, 11 hours ago * Uncommitted changes
21 |       <h3 mat-line>
22 |         <strong>{{book.title}}</strong>
23 |       </h3>
24 |       <div>
25 |         <button mat-button (click)="openDialog(book.id)">
26 |           <i class="material-icons">pageview</i>
27 |           Dialog
28 |         </button>
29 |         <button mat-button (click)="openRoute(book.id)">
30 |           <i class="material-icons"> pageview</i>
31 |           Route
32 |         </button>
33 |         <button mat-button (click)="delete(book)">
34 |           <i class="material-icons">delete_forever</i>
35 |           Delete
36 |         </button>
37 |       </div>
38 |       <p mat-line>
39 |         <span>{{book.author}}</span>
40 |       </p>
41 |       <p mat-line>
42 |         <my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">
43 |         </my-rating>
44 |       </p>
45 |       <p mat-line>
46 |         <span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Chec
47 |           </p>
48 |         </mat-list-item>
49 +       </div>
50 +     <ng-template #nobooks>
51 +       <div>No Books Available</div>
52 +     </ng-template>
53 |     <div class="text-right add-btn">
54 |       <button mat-raised-button color="primary" (click)="addBook()">
55 |         <i class="material-icons">add_box</i> ADD BOOK</button>
56 |       </div>
57 |     </div>
```

# Handling Errors

- When working with Observables its important to catch errors
- Errors stop the Observable stream and thus prevents it from emitting any more items and hence cannot be used anymore

# RxJS Operator: catchError

- Gracefully handles errors in an observable sequence and passes in a handleError method:

```
catchError(this.handleError)
```

- Two Strategies for handling errors with Observables:

1. **Catch and Replace**
2. **Catch and Rethrow**

# Catch and Replace Strategy

```
const http$ = this.http.get<Course[]>('/api/courses');

http$  
  .pipe(  
    catchError(err => of([]))  
  )  
  .subscribe(  
    res => console.log('HTTP response', res),  
    err => console.log('HTTP Error', err),  
    () => console.log('HTTP request completed.')  
  );
```

## Let's break down the implementation of the catch and replace strategy:

- We are passing to the catchError operator a function, which is the error handling function
- The error handling function is not called immediately, and in general, it's usually not called
- Only when an error occurs in the input observable of catchError, will the error handling function be called
- If an error happens in the input stream, this function is then returning an observable built using the of([]) function
- The of() function builds an observable that emits only one value ([])) and then it completes
- The error handling function returns the recovery observable (of([])), that gets subscribed to by the catchError operator
- The values of the recovery observable are then emitted as replacement values in the output observable returned by catchError

# Catch and Replace Strategy

```
const http$ = this.http.get<Course[]>('/api/courses');

http$  
  .pipe(  
    catchError(err => of([]))  
  )  
  .subscribe(  
    res => console.log('HTTP response', res),  
    err => console.log('HTTP Error', err),  
    () => console.log('HTTP request completed.')  
  );
```

As the end result, the http\$ Observable will not error out anymore! Here is the result that we get in the console:

- HTTP response []
- HTTP request completed.

As we can see, the error handling callback in subscribe() is not invoked anymore. Instead, here is what happens:

- The empty array value [] is emitted
- The http\$ observable is then completed

As we can see, the replacement Observable was used to provide a default fallback value ([]]) to the subscribers of http\$, despite the fact that the original Observable did error out.

Notice that we could have also added some local error handling, before returning the replacement Observable!

# Catch and Rethrow Strategy

```
const http$ = this.http.get<Course[]>('/api/courses');

http$  
  .pipe(  
    catchError(err => {  
      console.log('Handling error locally and rethrowing it...', err);  
      return throwError(err);  
    })  
  )  
  .subscribe(  
    res => console.log('HTTP response', res),  
    err => console.log('HTTP Error', err),  
    () => console.log('HTTP request completed.')  
  );
```

## Let's break down step-by-step the implementation of the Catch and Rethrow Strategy:

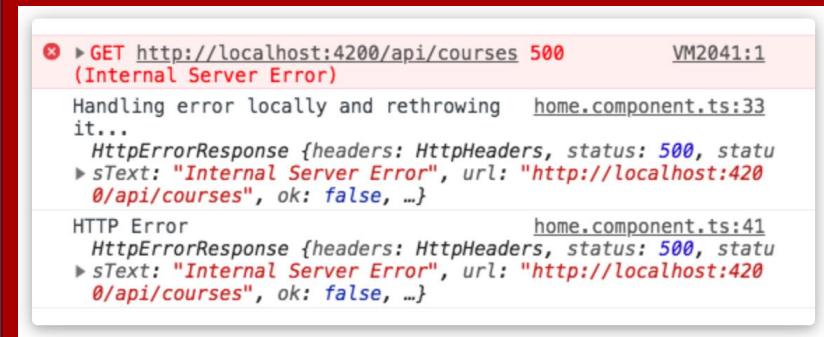
- Just like before, we are catching the error, and returning a replacement observable
- But this time around, instead of providing a replacement output value like [], we are now handling the error locally in the catchError function
- In this case, we are simply logging the error to the console, but we could instead add any local error handling logic that we want, such as for example showing an error message to the user
- We are then returning a replacement observable that this time was created using throwError
- Throwerror creates an observable that never emits any value (**Observable<never>**). Instead, it errors out immediately using the same error caught by catchError
- This means that the output observable of catchError will also error out with the exact same error thrown by the input of catchError
- This means that we have managed to successfully rethrow the error initially thrown by the input observable of catchError to its output observable
- The error can now be further handled by the rest of the observable chain, if needed

# Catch and Rethrow Strategy

```
const http$ = this.http.get<Course[]>('/api/courses');

http$  
  .pipe(  
    catchError(err => {  
      console.log('Handling error locally and rethrowing it...', err);  
      return throwError(err);  
    })  
  )  
  .subscribe(  
    res => console.log('HTTP response', res),  
    err => console.log('HTTP Error', err),  
    () => console.log('HTTP request completed.')  
  );
```

If we now run the code above, here is the result that we get in the console:



As we can see, the same error was logged both in the catchError block and in the subscription error handler function, as expected

# Handling Errors With Async Operator

## Non Reactive Programming

We need to add code to replace the observable using the EmptyError since the async pipe now expects an observable

We are returning EmptyError, but at the same time we are raising an error using the Angular Material Snackbar component

## Reactive Programming (Reactive Templates)

```
94  getBooks(): void {  
95 +   this.books$ = this._dataService.getBooks()  
96 +     .pipe(  
97 +       catchError(error => {  
98 +         this.updateMessage(<any>error, 'ERROR');  
99 +         return EMPTY;  
100 +      })  
101 +    );  
102 }
```

# Handling Errors With Async Operator

We need to add code to replace the observable  
We are returning EmptyError, but at the same time we  
are raising an error using the Angular Material Snackbar  
component

## Catch and Replace Error Handling Strategy

collection.component.ts

```
getBooks(): void {
  this.books$ = this._dataService.getBooks()
    .pipe(
      catchError(error => {
        this.updateMessage(<any>error, 'ERROR');
        return EMPTY;
      })
    );
}
```

# Both Error Handling Strategies In Action

## Catch and Rethrow Error Handling Strategy

data.service.ts

```
getBooks(query?: string): Observable<Ibook[]> {
  return this._http.get<Ibook[]>(this._booksUrl+"/GetBooks")
    .pipe(
      map((books:Ibook[]) => {
        if (query != null && query.length > 0) {
          books = books.filter(
            data =>
              data.author.includes(query) ||
              data.title.includes(query)
          )
        }
        return books;
      }),
      catchError(this.handleError)
    );
}
```

## Catch and Replace Error Handling Strategy

collection.component.ts

```
getBooks(): void {
  this.books$ = this._dataService.getBooks()
    .pipe(
      catchError(error => {
        this.updateMessage(<any>error, 'ERROR');
        return EmptyError;
      })
    );
}
```

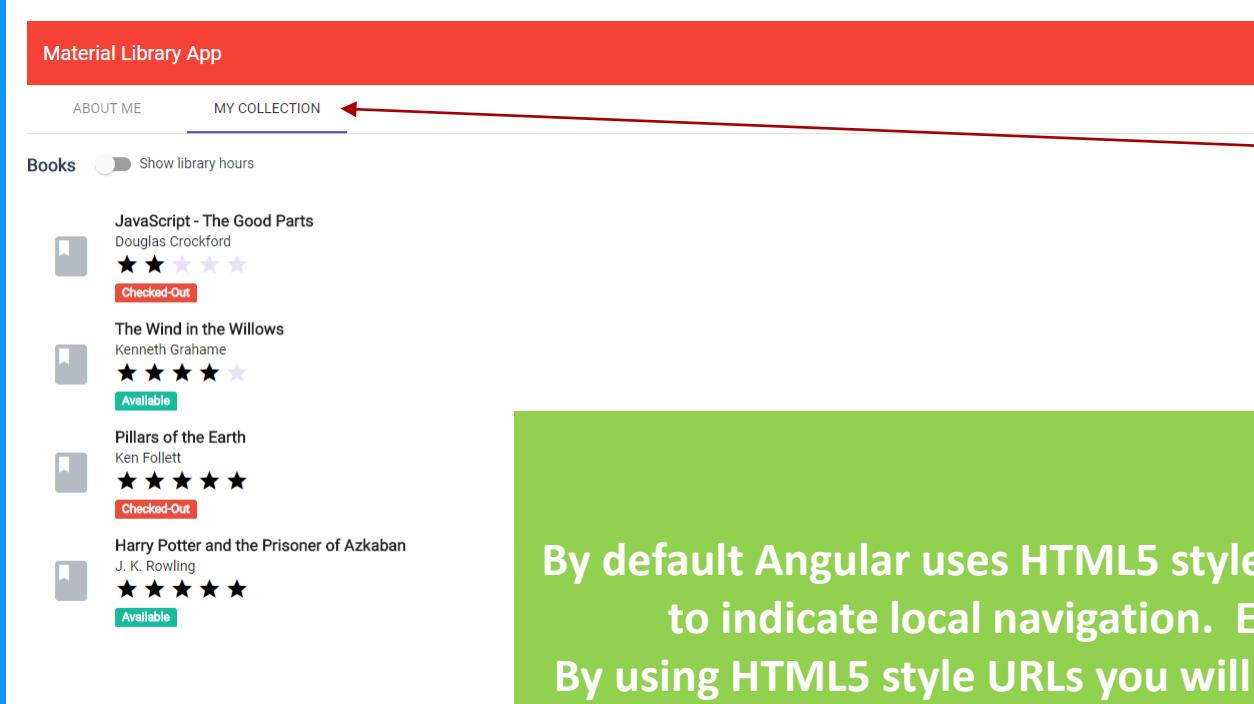
# Lab 8

## Reactive Programming

# Routing

- Configure a route for each component that wants to display its view
- Define options/actions
  - e.g. Menu item
- Tie a route to each option/action
- Activate the route based on user action
- Activating a route displays the component's view

# Routing



We tie a route to each menu option using a built in directive called routerLink

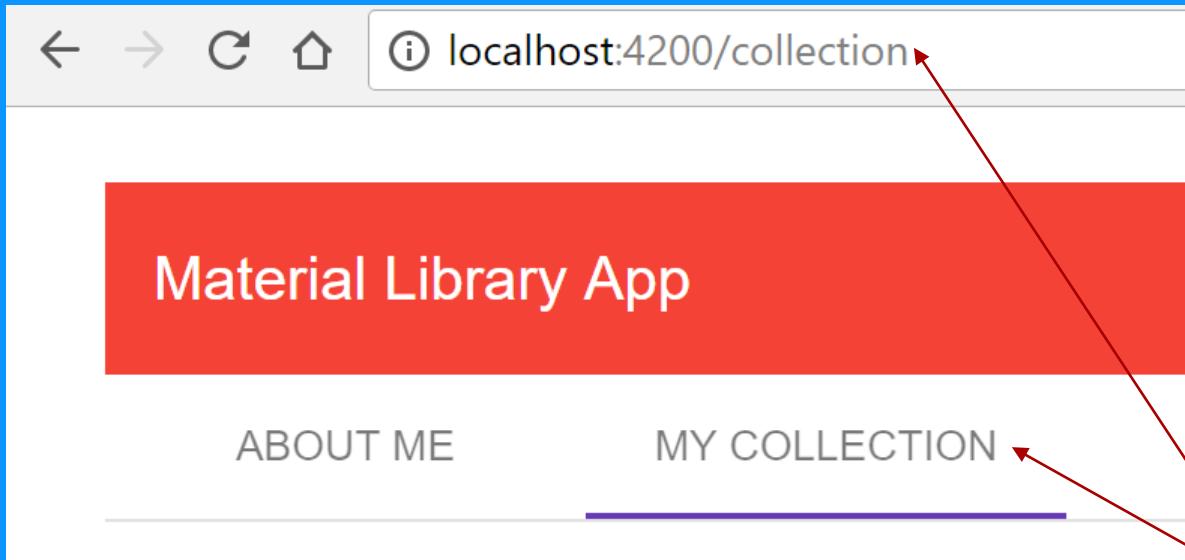
```
<a [routerLink]="/collection"> My Collection</a>
```

The ['collection'] portion is a template expression that returns a link parameters array. E.g. we could have passed a product id as the second parameter in the array

By default Angular uses HTML5 style URLs which doesn't require # symbol to indicate local navigation. E.g. www.myservice.com/library  
By using HTML5 style URLs you will need to configure your web server to support URL rewriting

Angular also supports # style routing which does not require URL rewriting

# Routing



Notice that the collection component doesn't have a selector as its not nested but instead used as part of routing

We tie a route to each menu option using a built in directive called `routerLink`

```
<a [routerLink]="/collection"> My Collection </a>
```

When clicking the menu option the address bar reflects the change

The angular router looks for a route definition matching the path segment below  
`{ path: 'collection', component: CollectionComponent }`

```
@Component({  
  templateUrl: './collection.component.html',  
  styleUrls: ['./collection.component.css']  
})  
export class CollectionComponent implements OnInit {
```

# Configuring Routes

- Angular application has one router that is managed by Angular's router service
- Before we use the service we need to register the router service provider in an angular module

```
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
    RouterModule, // RouterModule
    ProductModule
  ],
  providers: [
    ...
  ]
})
```

Similar to the **HttpModule** the router provider is registered inside the **RouterModule**

# Configuring Routes

- The router must be configured with a list of routes

```
const routes: Routes = [
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: 'collection',
    component: CollectionComponent
  },
  {
    path: '',
    redirectTo: '/about',
    pathMatch: 'full'
  }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

We set the list of route definitions that are available for our application using the `forRoot` method

The order of the paths in the array matters.  
More specific routes should always come before less specific routes like the `**`

# Configuring Routes

- The router must be configured with a list of routes

```
const routes: Routes = [  
  {  
    path: 'about',  
    component: AboutComponent  
  },  
  {  
    path: 'collection',  
    component: CollectionComponent  
  },  
  {  
    path: '',  
    redirectTo: '/about',  
    pathMatch: 'full'  
  }  
];  
  
export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Each definition specifies a route object

Path: Defines the URL path segment.

We can also have a route parameter using the colon. E.g. {path: 'product/:id,...')}

The empty path '' defines the default route

\*\* denotes a wildcard path. In this example we are redirected to the welcome page if the requested url does not match any prior paths

No leading slashes in the paths

# Configuring Routes

- The router must be configured with a list of routes

```
const routes: Routes = [
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: 'collection',
    component: CollectionComponent ←
  },
  {
    path: '',
    redirectTo: '/about',
    pathMatch: 'full'
  }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Each definition specifies a route object

Component: Component associated with the route

# Configuring Routes

- The router must be configured with a list of routes

```
const routes: Routes = [
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: 'collection',
    component: CollectionComponent
  },
  {
    path: '',
    redirectTo: '/about', ←
    pathMatch: 'full'
  }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Each definition specifies a route object

redirectTo: redirects to page. It requires a **pathMatch** to tell the router how to match the url path segment to the path of a route

By setting **pathMatch** to 'full' we are telling the router that we only want this default route when the entire client side portion of the path is empty

pathMatch:  
full | prefix

# Configuring Routes

- The base element tells the router how to compose navigation URLs
- In our example since the app folder is the application root we will set the href for the base tag to /

```
<!DOCTYPE html>
<html>

<head lang="en">
  <base href="/">
```

# Placing the Views

- When a route is activated the associated component's view is displayed

```
<nav md-tab-navbar>
  <a md-tab-link
    *ngFor="let link of navLinks"
    [hidden]="!link.path"
    [routerLink]="link.path"
    routerLinkActive #rla="routerLinkActive"
    [active]="rla.isActive">
    {{link.label}}
  </a>
</nav>
<router-outlet></router-outlet>
```

To specify where we want the routing component to display its view we use the `<router-outlet>`

# Lab 9

## Routing

# Advanced Routing

- Passing parameters to a route
- Activating a route with code
- Protecting routes with guards

# Passing Parameters To A Route

- Sometimes we need to pass a parameter in the route
  - E.g. `http://localhost:3000/collection/1`
- First step is to configure the routes with parameters
  - E.g. `{ path : 'collection/:id', component : BookDetailComponent }`
- With the route definition we need to add a `[routerLink]` which takes a parameter
  - E.g. `<a [routerLink] = "[ '/collection' , book.bookId ] ">`

# Passing Parameters To A Route

- We access the router parameter using the ActivatedRoute service
  - import {ActivatedRoute } from '@angular/router'
- Two ways to retrieve the parameter
  - **snapshot**: If we only need to get the initial value of the parameter
    - constructor( private \_route : ActivatedRoute ) {  
    console.log(this.\_route.snapshot.params['id']);  
}
    - For example if the user is always returned to the library before navigating to a new book so the snapshot approach would be sufficient
  - **observable**: If you expect the parameter to change without leaving the page then you will need to use an observable. For example if we had a next button on the book detail page to display the next book the url will change to the next book's id

# Activating A Route With Code

- So far we have been activating routes using the routerLink directive
  - E.g. <a [routerLink] = “[ ‘/collection’ , book.bookId ] ”>
- Routes can be activated using code
  - A typical use case is when you have a save button which would require executing some code to save the data and then route

```
onBack(): void {
    this._router.navigate(['/library']);
}
```

```
<div class='panel-footer'>
    <a class='btn btn-default' (click)='onBack()' style='width:80px'>
        <i class='glyphicon glyphicon-chevron-left'></i> Back
    </a>
</div>
```

# Lab 10

Passing Parameters To a Route and Activating a Route  
with Code

# Protecting Routes With Guards

- You want to limit routes to specific users
  - E.g. admin only
- You want to confirm a navigation operation
  - E.g. asking the user to save before navigating away from an edit page
- Angular router provides several guards
  - **CanActivate**: Guard navigation to a route
  - **CanDeactivate**: Guard navigation away from a route
  - **Resolve**: Pre-fetch data before activating a route
  - **CanLoad**: Prevent asynchronous routing
- The guards service provider must be provided at the Angular Module level

# Protecting Routes With Guards

```
@Injectable()
export class BookGuardService {

  constructor(private _router: Router, private _dataService: DataService) { }

  canActivate(route: ActivatedRouteSnapshot) {          Wael Kdouh, a year ago •
    // parse the book id from the route.
    // Note:+ ensures that the value is returned as number and not string
    const id:number = +route.params.id;

    if (isNaN(id)) {
      // start a new navigation to redirect to list page
      this._router.navigate(['/collection']);
      // abort current navigation
      return false;
    };

    return this._dataService.canActivate(id)
      .pipe(
        map(result => {
          if (result) {
            || return true;
          }
          this._router.navigate(['/collection']);
          return of(false);
        }),
        catchError(() => {
          this._router.navigate(['/collection']);
          return of(false);
        })
      );
  }
}
```

```
{
  path: ':id',
  canActivate: [BookGuardService],
  component: BookDetailComponent
}
```

# Lab 11

Protecting Routes with Guards

# Forms

- A form creates a cohesive, effective, and compelling data entry experience
- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	

```
@Component({  
  templateUrl: 'app/books/library.component.html'  
})
```

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	
Unit Test Against DOM	

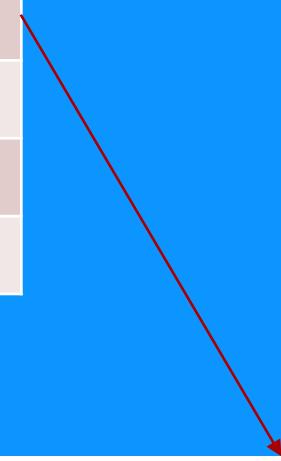


This could be difficult and slow in a lot of circumstances  
For example the validation might exist in HTML5 attributes like 'require' 'minlength'  
'maxlength' which is usually up to the browser to take care of that

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	



```
@Component({
  templateUrl: 'app/books/library.component.html'
})
```

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	Create a Form Model in TypeScript (this must be in sync with the template)

You are somewhat repeating yourself

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	Create a Form Model in TypeScript (this must be in sync with the template)
	Unit Test Against Form Model

Can lead to faster unit testing

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	Create a Form Model in TypeScript (this must be in sync with the template)
	Unit Test Against Form Model
	Validation in Form Model

Validation doesn't necessarily need to be in the DOM  
It can be put directly in the Form Model

# Forms

- Angular Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	
Unit Test Against DOM	

# Forms

- Steps involved in building a form
  - Build an Angular form with a component and template
  - Two-way data bind with [(ngModel)] syntax for reading and writing values to input controls
  - Track the change state and validity of form controls using ngModel in combination with a form
  - Provide strong visual feedback using special CSS classes that track the state of the controls
  - Display validation errors to users and enable/disable form controls
  - Use template reference variables for sharing information among HTML elements

# Form Component

- There's nothing special about a form component
- Nothing form-specific, nothing to distinguish it from any component we've written before

# Form Component

- There's nothing special about a form component
- Nothing form-specific, nothing to distinguish it from any component we've written before

```
@NgModule({  
  imports: [BrowserModule, FormsModule, HttpClientModule, routing],  
  declarations: [AppComponent, AboutComponent, LibraryComponent],  
  bootstrap: [AppComponent],  
  providers: [BookDetailGuard]  
})  
export class AppModule { }
```

We need to add the `FormsModule` to the array of imports before we can use forms

This gives our application access to all of the template-driven forms features, including `ngModel`

# Two-way data binding with ngModel

```
<div class="form-group">
  <label for="title">Title</label>
  <input type="text" class="form-control" id="title" required [(ngModel)]="book.title" name="title">
</div>
```

Enables Two-way binding on the input element

Defining a name attribute is a requirement when using [(ngModel)] in combination with a form

Internally Angular creates FormControl and registers them with an NgForm directive that Angular attached to the <form> tag. Each FormControl is registered under the name we assigned to the name attribute

# Track change-state and validity with ngModel

- A form isn't just about data binding. We'd also like to know the state of the controls on our form
- Using ngModel in a form gives us more than just two way data binding. It also tells us if the user touched the control, if the value changed, or if the value became invalid

# Track change-state and validity with ngModel

- The ngModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. We can leverage those class names to change the appearance of the control and make messages appear or disappear

# Track change-state and validity with ngModel

State	Class if true	Class if false
Control has been visited	<code>ng-touched</code>	<code>ng-untouched</code>
Control's value has changed	<code>ng-dirty</code>	<code>ng-pristine</code>
Control's value is valid	<code>ng-valid</code>	<code>ng-invalid</code>

# Add Custom CSS for Visual Feedback

## New Book

TODO: remove this: {"id":26,"title":"","author":"","isCheckedOut":false,"rating":1}

Title



Author



Rating



Submit

New Book

Invalid State

## New Book

TODO: remove this: {"id":26,"title":"","author":"","isCheckedOut":false,"rating":1}

Title



Author



Rating



Submit

New Book

Valid State

# Add Custom CSS for Visual Feedback

```
.ng-valid[required], .ng-valid.required {  
    border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
    border-left: 5px solid #a94442; /* red */  
}
```

# Show and Hide Validation Error messages

## New Book

```
{"id":26,"title":"","author":"","isCheckedOut":false,"rating":1}
```

Title

Title is required

Validation Message

# Show and Hide Validation Error messages

```
<div class="form-group">
  <label for="title">Title</label>
  <input type="text" class="form-control" id="title" required [(ngModel)]="book.title"
    name="title" #titleSpy="ngModel">
  <div [hidden]="titleSpy.valid || titleSpy.pristine" class="alert alert-danger">
    Title is required
  </div>
  <br>TODO: remove this: {{titleSpy.className}}
</div>
```

We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called titleSpy and gave it the value "ngModel"

Why is the reference variable set to "ngModel"?

A directive's exportAs property tells Angular how to link the reference variable to the directive. We set name to ngModel because the ngModel directive's exportAs property happens to be "ngModel"

# Show and Hide Validation Error messages

```
<div class="form-group">
  <label for="title">Title</label>
  <input type="text" class="form-control" id="title" required [(ngModel)]="book.title"
    name="title" #titleSpy="ngModel">
  <div [hidden]="titleSpy.valid || titleSpy.pristine" class="alert alert-danger">
    Title is required
  </div>
  <br>TODO: remove this: {{titleSpy.className}}
</div>
```

Now we can control visibility of the "Title" error message by binding properties of the title control to the message `<div>` element's `hidden` property

In this example, we hide the message when the control is valid or pristine; pristine means the user hasn't changed the value since it was displayed in this form

# What About Resetting The Form?

## New Book

TODO: remove this: {"id":26,"title":"","author":"","isCheckedOut":false,"rating":1}

Title

Author

Rating

Submit

New Book

```
<button type="button"  
       class="btn btn-default"  
       (click)="newBook();">New Book  
</button>
```

Data Entered

## New Book

TODO: remove this: {"id":27,"title":"","author":"","isCheckedOut":false,"rating":1}

Title

Title is required

Author

Rating

Submit

New Book

The form remembers that we entered data before clicking New Book. Replacing the book object did not restore the pristine state of the form controls

# What About Resetting The Form?

## New Book

TODO: remove this: {"id":27,"title":"Angular 2 Rocks","author":null,"isCheckedOut":null,"rating":null}

Title

Author

Rating

SubmitNew Book

```
<button type="button"  
       class="btn btn-default"  
       (click)="newBook();  
       newBookForm.reset()">New Book  
</button>
```

We have to clear all of the flags imperatively which we can do by calling the form's reset() method after calling the newBook() method

## New Book

TODO: remove this: {"id":27,"title":null,"author":null,"isCheckedOut":null,"rating":null}

Title



Author

Rating

SubmitNew Book

Now clicking "New Book" both resets the form and its control flags. The required message doesn't show up as now the title field is pristine

# Submit The Form With ngSubmit

```
<button type="submit" class="btn btn-default">Submit</button>
```

The Submit button at the bottom of the form does nothing on its own but it will trigger a form submit because of its type (type="submit")

```
<form (ngSubmit)="onSubmit()" #newBookForm="ngForm">
```

To make "form submit" useful, the <form> tag needs NgSubmit

# Submit The Form With ngSubmit

## The NgForm directive

What `NgForm` directive? We didn't add an `NgForm` directive!

Angular did. Angular creates and attaches an `NgForm` directive to the `<Form>` tag automatically.

The `NgForm` directive supplements the `Form` element with additional features. It holds the controls we created for the elements with `ngModel` directive and `name` attribute and monitors their properties including their validity. It also has its own `valid` property which is true only if every contained control is valid.

```
<form (ngSubmit)="onSubmit()" #newBookForm="ngForm">
```

## What about `#newbookForm` template reference?

The variable `newbookForm` is now a reference to the `NgForm` directive that governs the form as a whole

# Submit The Form With ngSubmit

## The NgForm directive

What `NgForm` directive? We didn't add an `NgForm` directive!

Angular did. Angular creates and attaches an `NgForm` directive to the `<form>` tag automatically.

The `NgForm` directive supplements the `form` element with additional features. It holds the controls we created for the elements with `ngModel` directive and `name` attribute and monitors their properties including their validity. It also has its own `valid` property which is true only if every contained control is valid.

```
<button type="submit" class="btn btn-default"
        [disabled]="!newBookForm.form.valid">
    Submit
</button>
```

Bind the button's disabled property to the form's over-all validity via the `newBookForm` variable

# Lab 12

Adding Forms

# Lazy Loading Modules

- Another advantage of using modules to group related pieces of functionality of our application is the ability to load those pieces on demand using Lazy Loading
- Lazy Loading modules helps us decrease the startup time

# Lazy Loading Modules

- With lazy loading our application does not need to load everything at once, it only needs to load what the user expects to see when the app first loads
- Modules that are lazily loaded will only be loaded when the user navigates to their routes

# Lazy Loading Modules

app/app.routing.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { EagerComponent } from './eager.component';

const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Whenever we try to go to the path `lazy`, we are going to lazy load a module conveniently called `LazyModule`

There's a few important things to notice here:

We use the property `loadChildren` instead of `component`

We define not only the path to the module but the name of the class as well

# Lazy Loading Modules

```
import { NgModule } from '@angular/core';
import { LazyComponent } from './lazy.component';
import { routing } from './lazy.routing';

@NgModule({
  imports: [routing],
  declarations: [LazyComponent]
})
export class LazyModule {}
```

There's nothing special about LazyModule other than it has its own routing

# Lazy Loading Modules

app/lazy/lazy.routing.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { LazyComponent } from './lazy.component';

const routes: Routes = [
  { path: '', component: LazyComponent }
];

export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

Notice that we use the method call `forChild` instead of `forRoot` to create the routing object

We should always do that when creating a routing object for a feature module, no matter if the module is supposed to be eagerly or lazily loaded

Demo

Lazy Loading

Lab 13

Lazy Loading

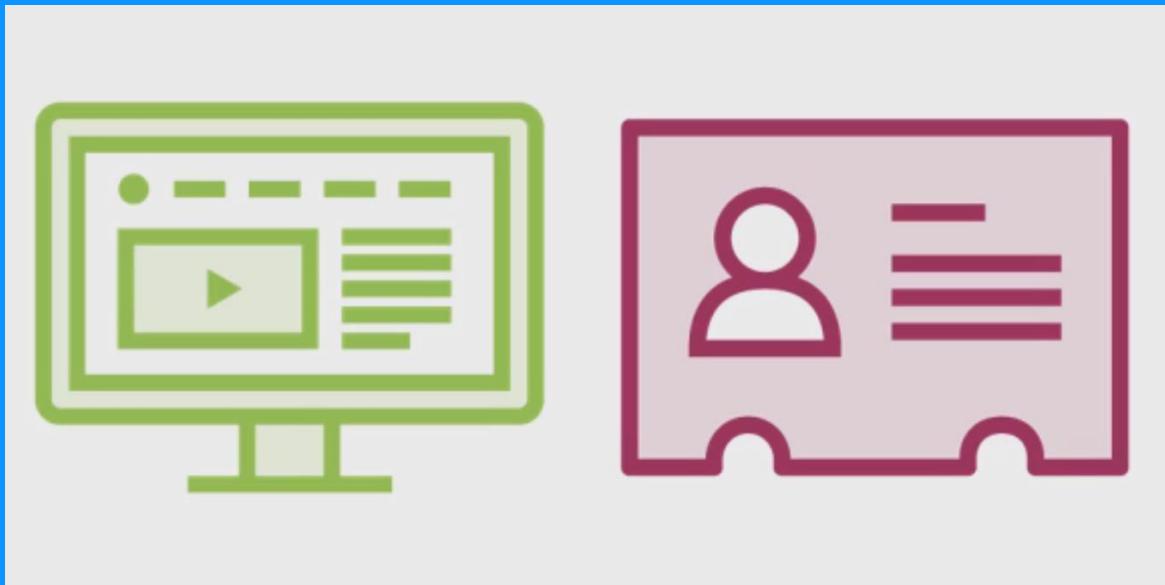


# What is State?



View State

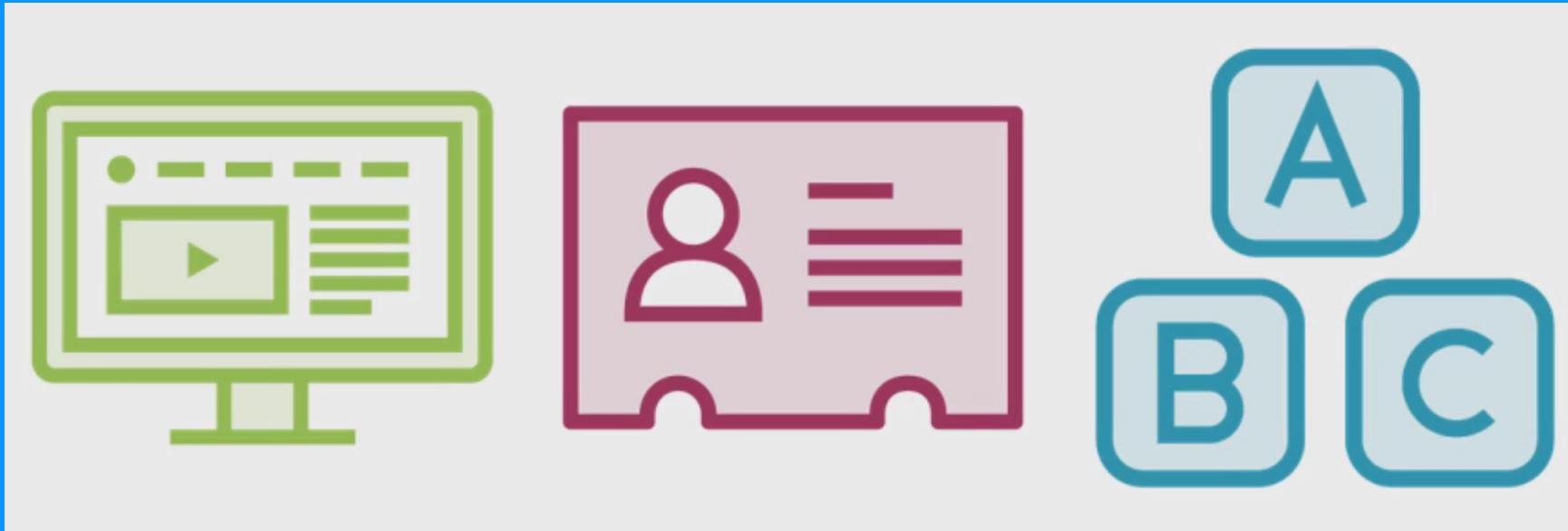
# What is State?



View State

User  
Information

# What is State?

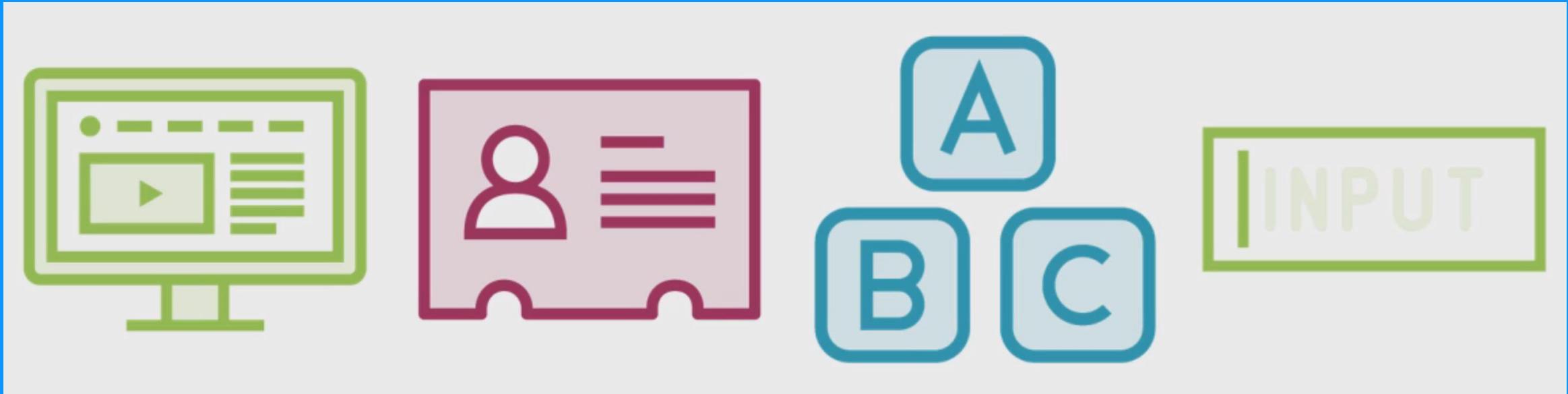


View State

User  
Information

Entity Data

# What Is State?



View State

User  
Information

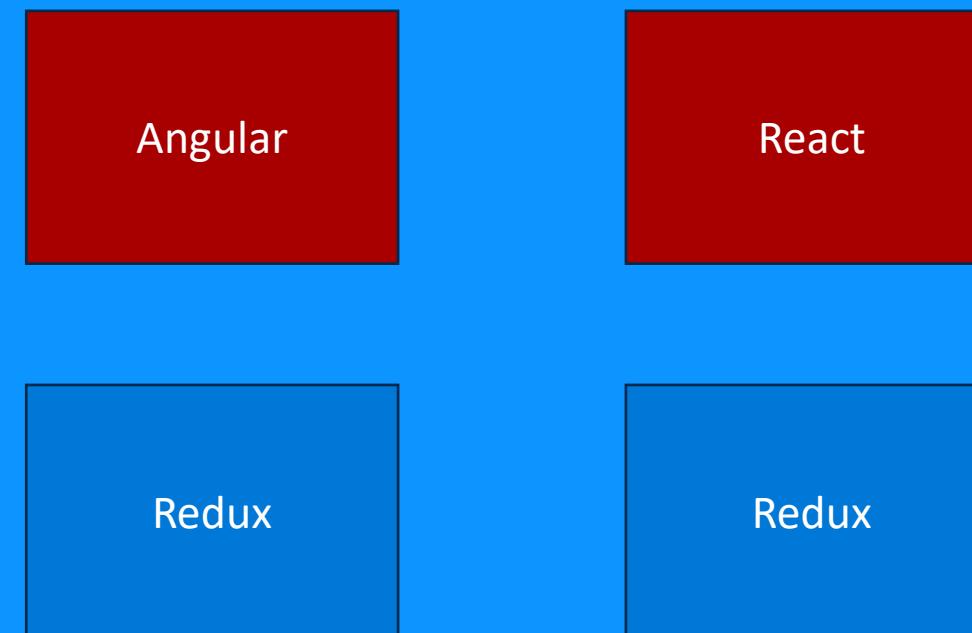
Entity Data

User  
Selection  
and Input

# State Management With Redux

- Isn't Redux a React thing?

- It is part of facebook's open source project but it can be used with Angular or any other web framework



Angular

React

Redux

Redux

# State Management With Redux

- What is Redux?

Redux attempts to make state mutations predictable and easy to debug by imposing certain restrictions on how and when updates can happen

# State Management With Redux

- State is difficult to manage in most applications
- Most bugs can be traced back to state changes that were not applied or misinterpreted

# State Management With Redux

- Do not use it if you are building a simple application as it would be an overkill
- Its purpose is to make state mutations predictable which only makes sense if you have a decent amount of state

# State Management With Redux

- Do not use it if you are building a simple application as it would be an overkill
- Its purpose is to make state mutations predictable which only makes sense if you have a decent amount of state

# Redux Benefits

- Predictable state management
- Enjoyable developer workflow
- Makes it easy to implement features like undo/redo
- Great Tooling

# When To Use Redux

- Independent copies of the same data in multiple places
- Multiple views that need to work with the same data and be in sync
- Data can be updated by multiple users
- Data can be updated by multiple actors

# Redux Core Principles

1. Single source of truth – you should be able to look at the redux state and that should tell you exactly why your views are in the state that they are in

No more state spread all over your app with you playing the detective to track down data. Its all in one big store and you can easily take a peak at it

# Redux Core Principles

2. State is read only – when you want to update the state of your application, you will actually make a copy of the state and change just the bits that should be updated

# Redux Core Principles

3. Pure functions drive state changes – a pure function doesn't have any side effects. It can be called multiple times with the same input and should always return the same output

Makes the code easy to test and reason about

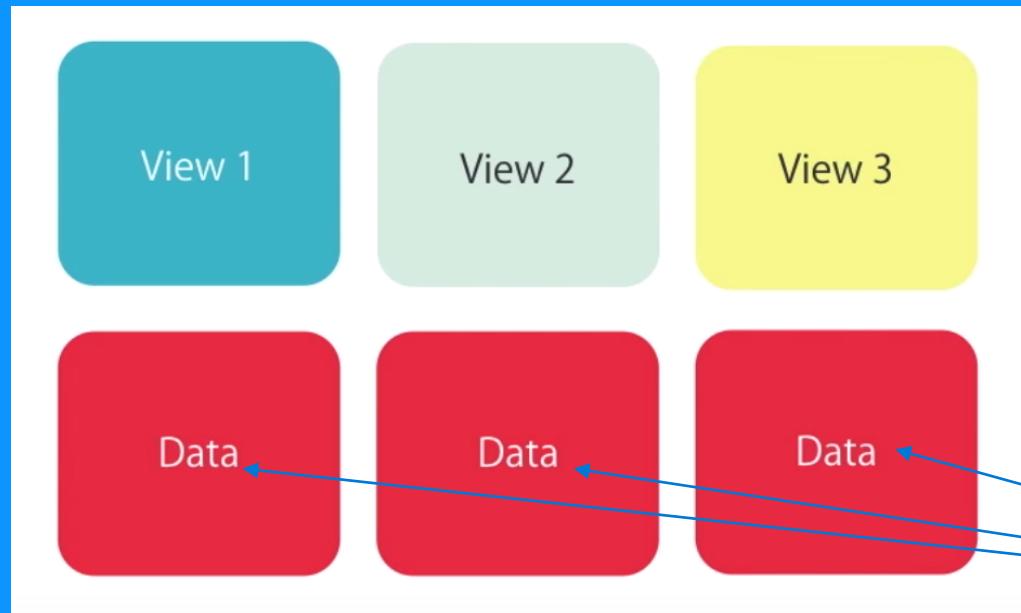
# What is The Problem That Redux is Trying To Solve?

```
export class MessagesComponent {  
  messages: any[];  
  
  postMessage(message) {  
    ...  
  }  
}
```

Without the Redux architecture  
each component maintains the  
state and the logic behind the view

This aligns perfectly with the  
encapsulation principle of object-  
oriented programming

# What is The Problem That Redux is Trying To Solve?

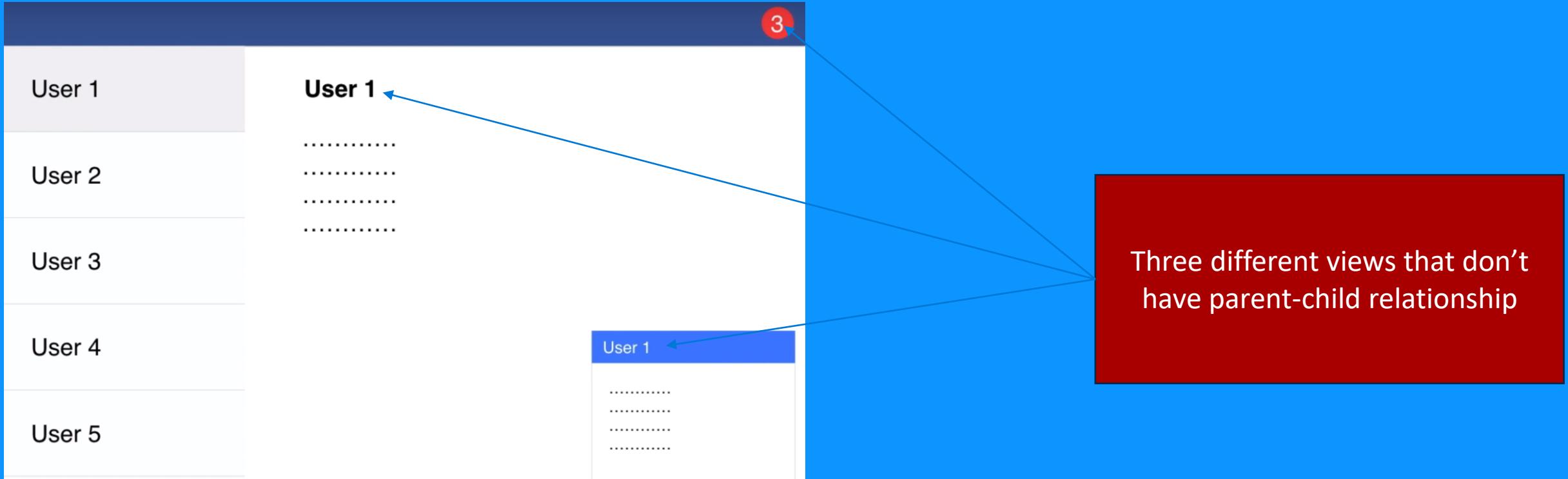


This could be a problem when you have multiple views that are working with the same piece of data and do not have the parent-child relationship

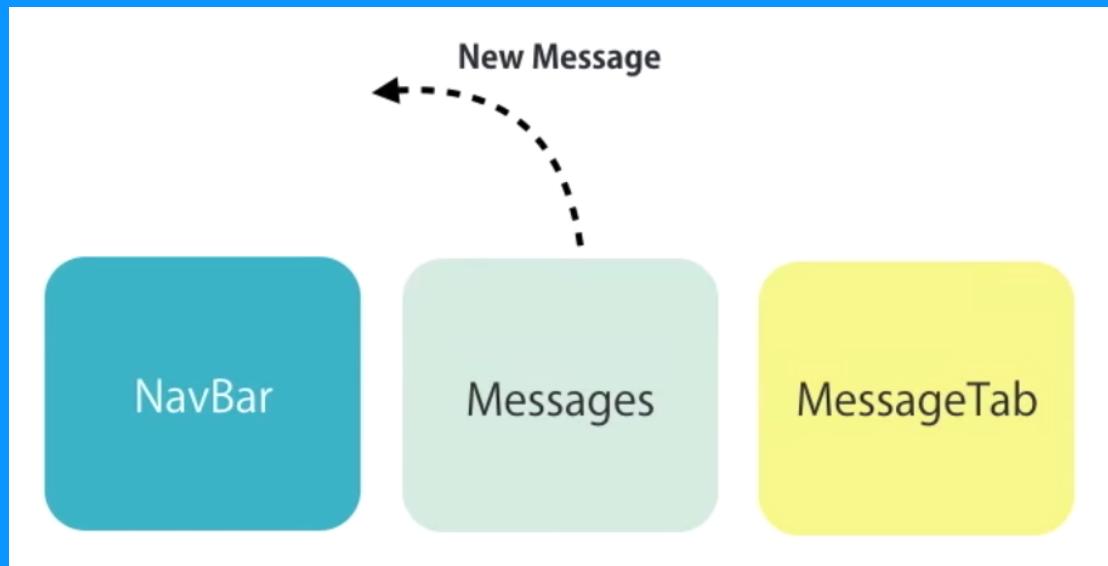
In these situations we often have multiple copies of the same data that are independent of each other

So when a view updates a model we need to do some extra work to keep the other views in sync

# What is The Problem That Redux is Trying To Solve?



# What is The Problem That Redux Solves?

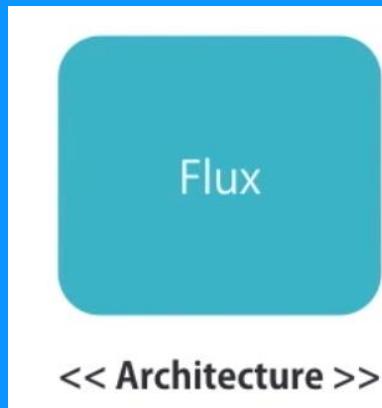


A common solution is to use Events, but sooner or later that will turn into an event spaghetti code

When there is a bug you have to figure out how the data is flowing and how the application state is updated in multiple places

**Unpredictable**

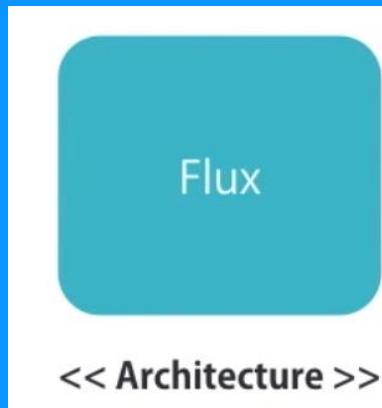
# What is The Problem That Redux Solves?



Facebook had this problem back in 2014 and as a result they introduced the Flux architecture

Redux is light implementation of this architecture that provides an elegant way to maintain the application's state in a **predictable** way

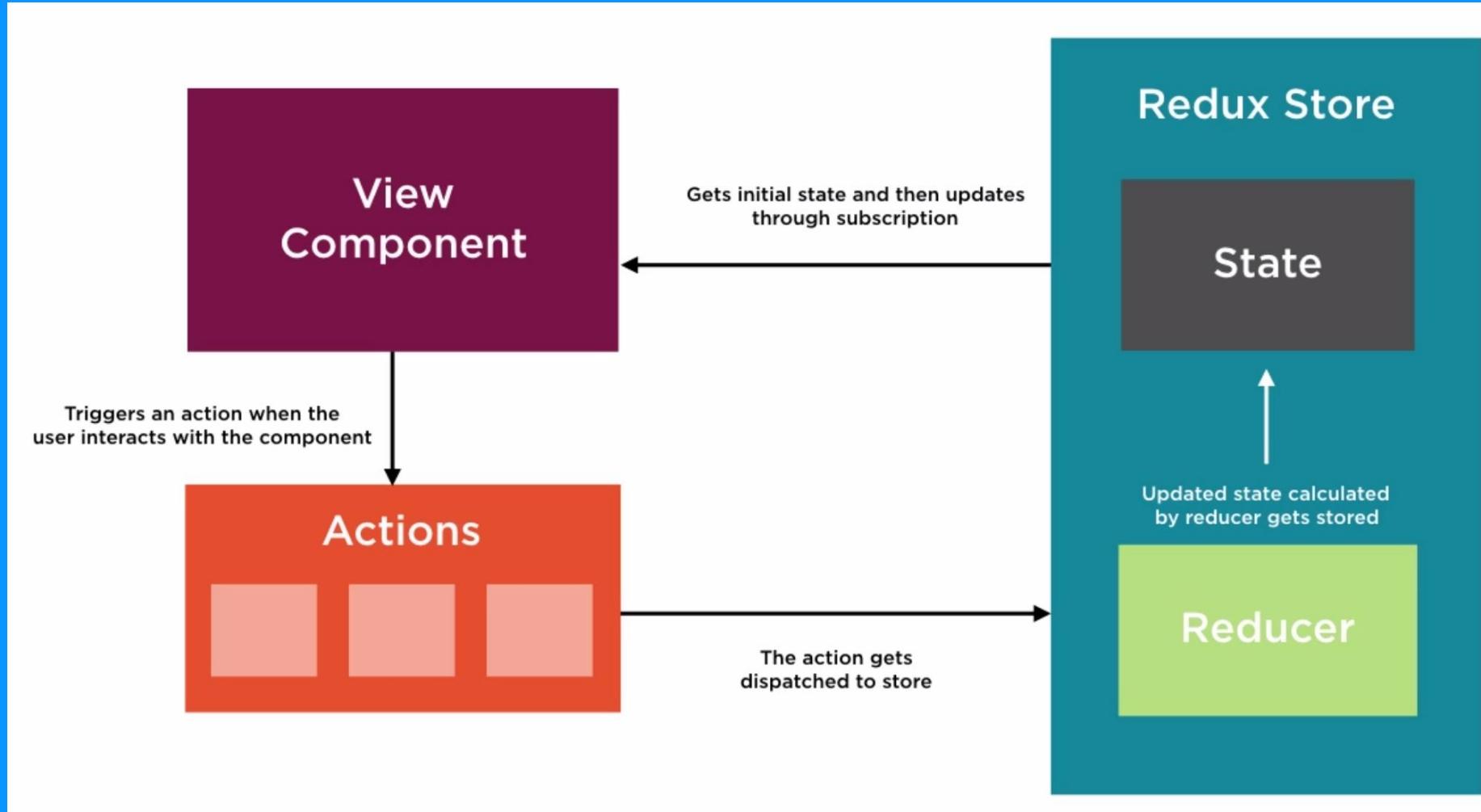
# What is The Problem That Redux Solves?



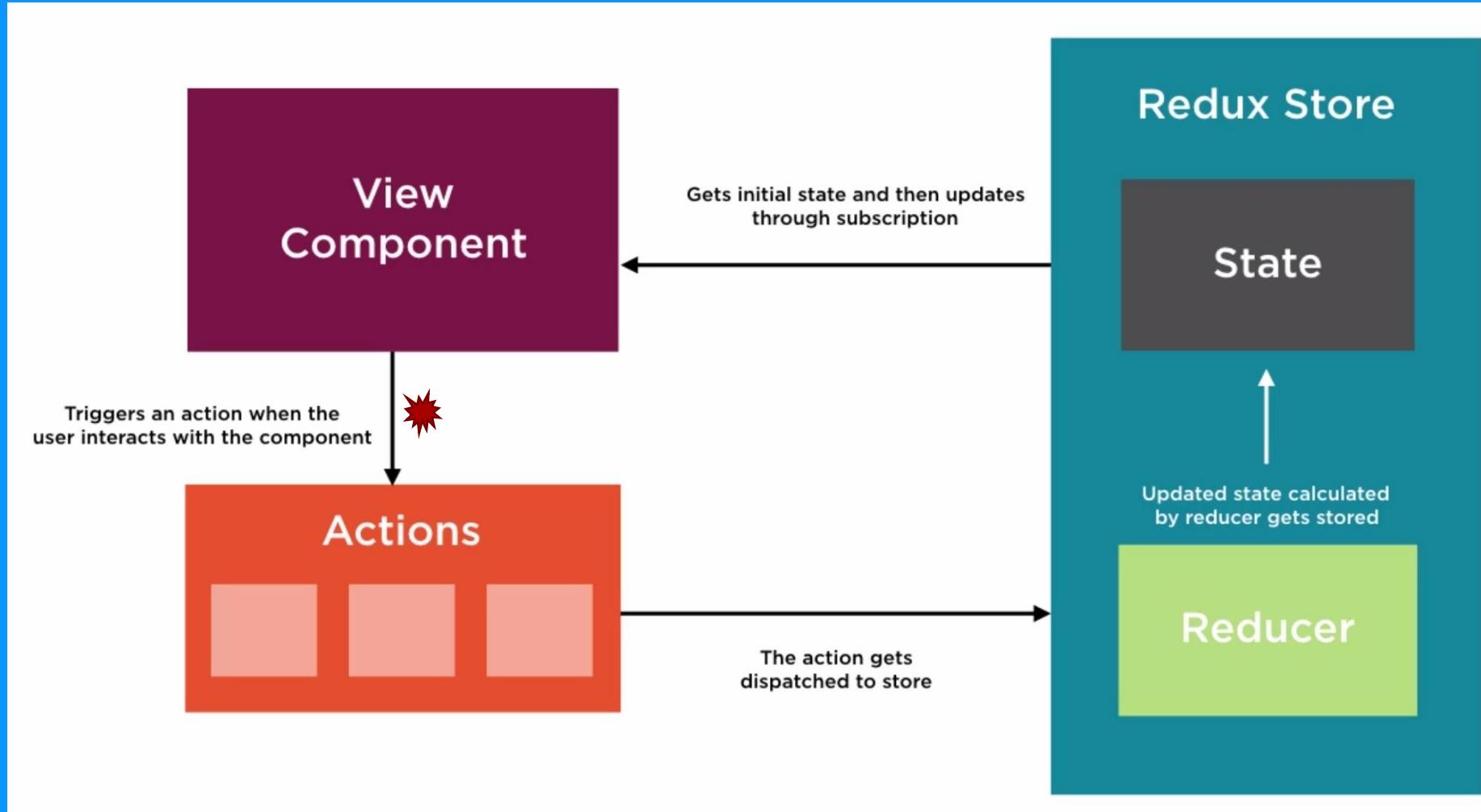
Facebook had this problem back in 2014 and as a result they introduced the Flux architecture

Redux is light implementation of this architecture that provides an elegant way to maintain the application's state in a **predictable** way

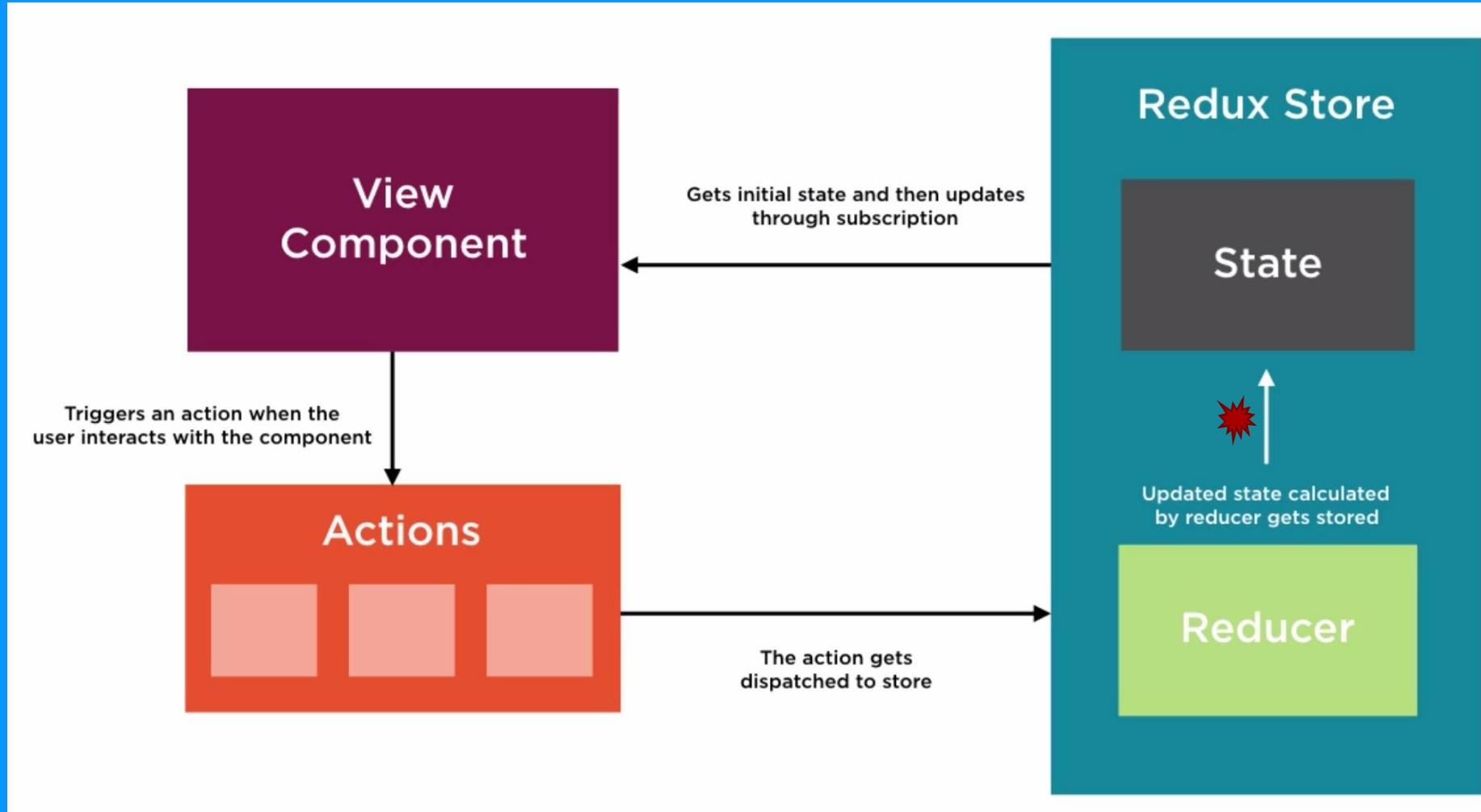
# Redux Pattern



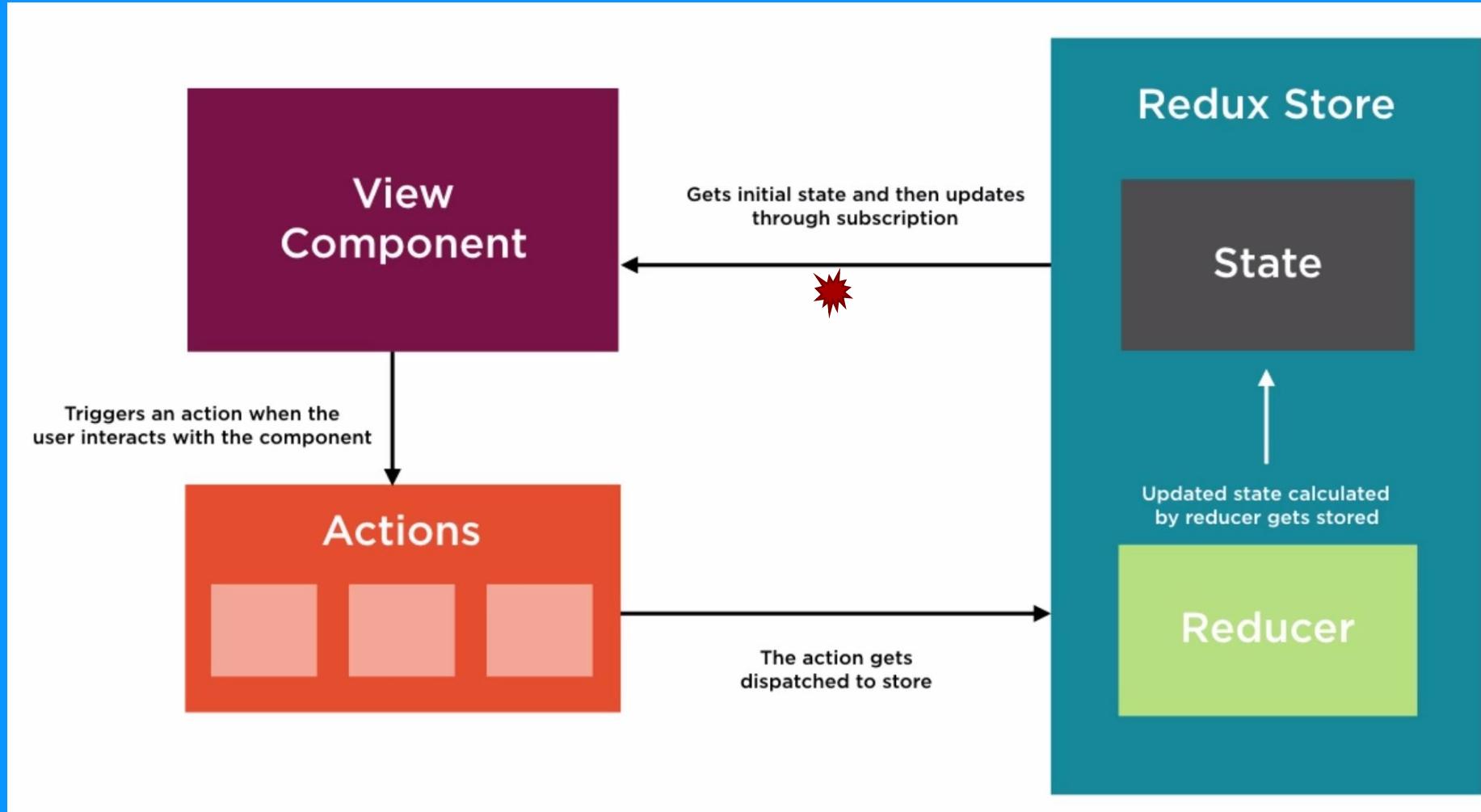
# Redux Pattern



# Redux Pattern



# Redux Pattern



Demo

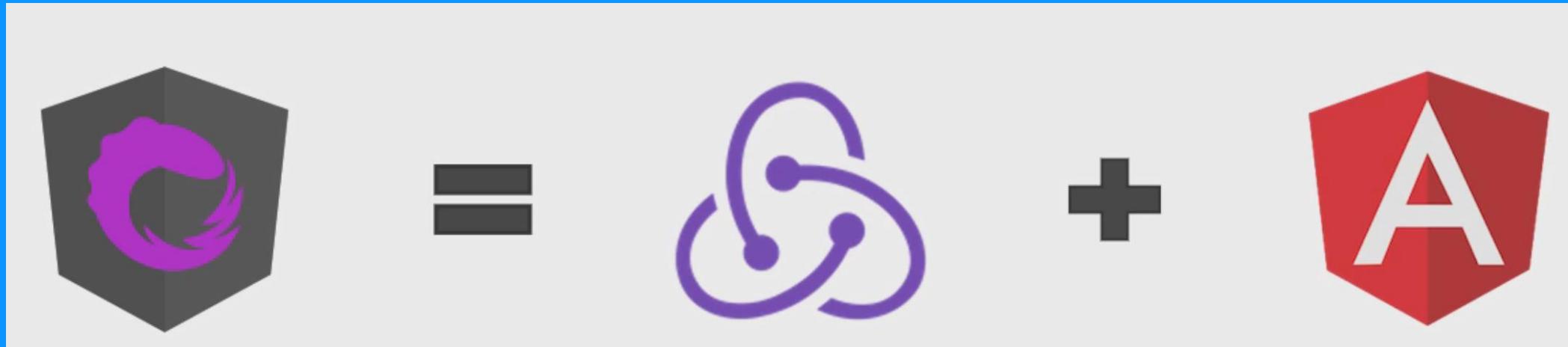
Redux Without Angular

# What is NgRx?

The screenshot shows the GitHub repository page for `ngrx/platform`. At the top, there are several status indicators: Sponsor, Used by 127, Watch 214, Star 5,202, Fork 1,333. Below the header, there are navigation links for Code, Issues (79), Pull requests (13), Actions, Projects (1), Wiki, Security, and Insights. A red dashed circle highlights the 'Issues' link. The repository title is 'Reactive libraries for Angular' with a link to <https://ngrx.io>. Below the title, there are several blue buttons with red text: 'ngrx', 'is', 'angular', 'observables', 'reactive', 'state-management', and 'redux'. A red dashed circle highlights the 'angular' button. Below these buttons, there are summary statistics: 881 commits, 21 branches, 0 packages, 39 releases, 214 contributors, and an MIT license. A red dashed circle highlights the 'branches' link. The main content area shows a list of recent commits, each with a small icon, the author, the commit message, and the time it was made. A red dashed circle highlights the first commit by 'Edouardbozon'.

Author	Commit Message	Time Ago
Edouardbozon and brandonroberts	docs: remove outdated alert for runtime checks (#2120)	2 days ago
.circleci	ci: fix filters on tags for publishing to npm and deployment of docs (#...)	23 days ago
.github	chore: add sponsorship button to OpenCollective (#1912)	4 months ago
build	chore: release 8.0.0	3 months ago
docs	docs: remove extra character in legacy effects testing guide (#1609)	6 months ago
modules	docs: add some guidance on when to use strict serializability for act...	6 days ago
projects	docs: remove outdated alert for runtime checks (#2120)	2 days ago
tools	build: point to RxJS 6.4	3 months ago
.bazelignore	test(example-app): add cypress tests (#1565)	7 months ago
.bazelrc	chore: update dependencies to Angular version 8.0.0 (#1898)	4 months ago
.dockerignore	docs(example): update ngrx.io for docker and offline support (#1806)	5 months ago
.editorconfig	docs: Use HTTPS in URLs where applicable (#1115)	last year
.gitignore	docs: Use HTTPS in URLs where applicable (#1115)	last year
.prettierignore	docs: add store tutorial example and tweak build to support examples (#...)	10 months ago

# What is NgRx?

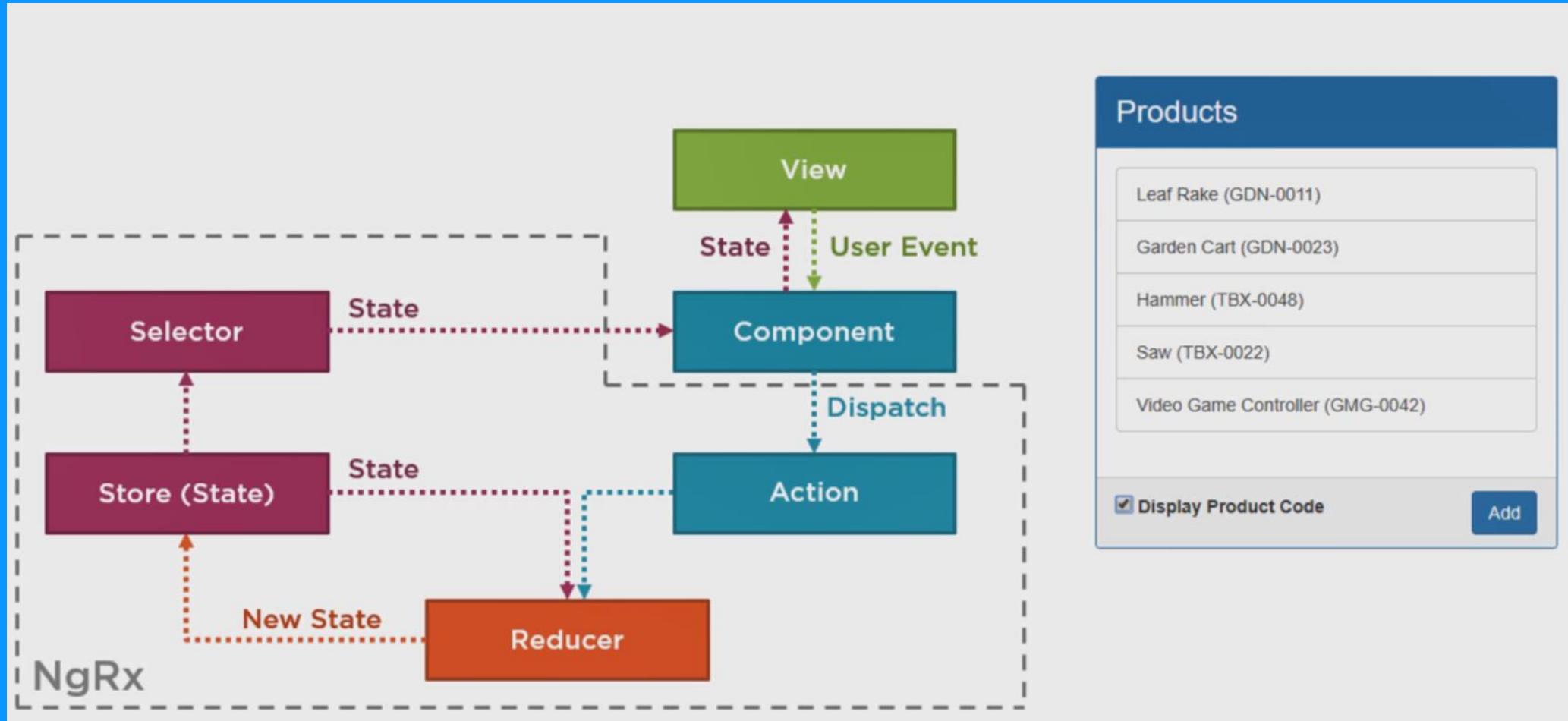


NgRx

Redux  
Pattern

Angular

# What is NgRx?



# Why Use NgRx?

**Scenario 1:** Use the store to retain UI state between router views when there is state everywhere

Service1

```
export class Service1 {  
  Property1:value1,  
  Property2:value2  
}
```

# Why Use NgRx?

**Scenario 1:** Use the store to retain UI state between router views when there is state everywhere

Service1

Service2

```
export class Service2 {  
  Property1:value1,  
  Property2:value2  
}
```

# Why Use NgRx?

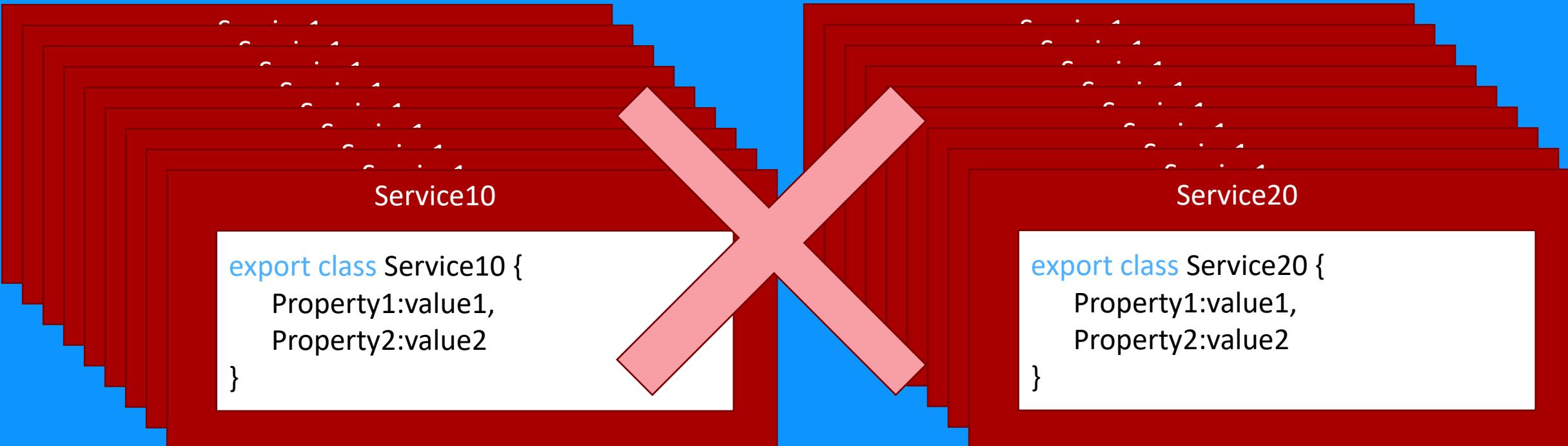
**Scenario 1:** Use the store to retain UI state between router views when there is state everywhere



```
export class Service10 {
  Property1:value1,
  Property2:value2
}
```

# Why Use NgRx?

**Scenario 1:** Use the store to retain UI state between router views when there is state everywhere



```
Service10
export class Service10 {
  Property1:value1,
  Property2:value2
}
```

```
Service20
export class Service20 {
  Property1:value1,
  Property2:value2
}
```

# Why Use NgRx?

Scenario 2: Use the store to provide a client side cache when there is excessive http requests

## Product Service

```
export class ProductService {
  private productUrl = 'api/products'

  constructor(private http:HttpClient) {}
  getProducts():Observable<Product[]> {
    return this.http.get<product[]>(this.productUrl);
  }
}
```

# Why Use NgRx?

Scenario 2: Use the store to provide a client side cache when there is excessive http requests

Product Service

```
export class ProductService {  
  private products: Product[] = [];  
  
  constructor(private http: HttpClient) {}  
  
  getProducts(): Observable<Product[]> {  
    return this.http.get('https://fakestoreapi.com/products');  
  }  
}
```

Product List Component

```
export class ProductListComponent {  
  products: Product[] = [];  
  
  constructor(private productService: ProductService) {}  
  
  ngOnInit(): void{  
    this.productService.getProducts()  
      .subscribe(products => this.products = products)  
  }  
}
```

# Why Use NgRx?

## Scenario 3: Use the store when there are complex component interactions

When user selects a product in the product list

Acme Product Management    Home    Product List    Add Product    Welcome Deborah    Log Out

Number of months on the market: 9

**Products**

- Garden Cart (GDN-0023) (highlighted)
- Hammer (TBX-0048)
- Leaf Rake (GDN-0011)
- Saw (TBX-0022)
- Video Game Controller (GMG-0042)

**Product Detail: Garden Cart**

Name: Garden Cart  
Code: gdn-0023  
Description: 15 gallon capacity rolling garden cart  
Availability: March 18, 2017  
Price: \$32.99  
5 Star Rating: ★★★★

[Edit](#)

# Why Use NgRx?

**Scenario 3:** Use the store when there are complex component interactions

Notify the shell component to display the correct number of months

Acme Product Management    Home    Product List    Add Product    Welcome Deborah    Log Out

Number of months on the market: 9 1

**Products**

- Garden Cart (GDN-0023)
- Hammer (TBX-0048)
- Leaf Rake (GDN-0011)
- Saw (TBX-0022)
- Video Game Controller (GMG-0042)

**Product Detail: Garden Cart**

Name: Garden Cart  
Code: gdn-0023  
Description: 15 gallon capacity rolling garden cart  
Availability: March 18, 2017  
Price: \$32.99  
5 Star Rating: ★★★★

[Edit](#)

# Why Use NgRx?

**Scenario 3:** Use the store when there are complex component interactions

Acme Product Management    Home    Product List    Add Product    Welcome Deborah    Log Out

Number of months on the market: 9

**Products**

- Garden Cart (GDN-0023)
- Hammer (TBX-0048)
- Leaf Rake (GDN-0011)
- Saw (TBX-0022)
- Video Game Controller (GMG-0042)

**Product Detail: Garden Cart** 2

Name: Garden Cart  
Code: gdn-0023  
Description: 15 gallon capacity rolling garden cart  
Availability: March 18, 2017  
Price: \$32.99  
5 Star Rating: ★★★★

Edit

Notify the detail page to display the product details

# Why Use NgRx?

## Scenario 3: Use the store when there are complex component interactions

Acme Product Management    Home    Product List    Add Product    Welcome Deborah    Log Out

Number of months on the market: 9

Products 3

- Garden Cart (GDN-0023)
- Hammer (TBX-0048)
- Leaf Rake (GDN-0011)
- Saw (TBX-0022)
- Video Game Controller (GMG-0042)

Product Detail: Garden Cart

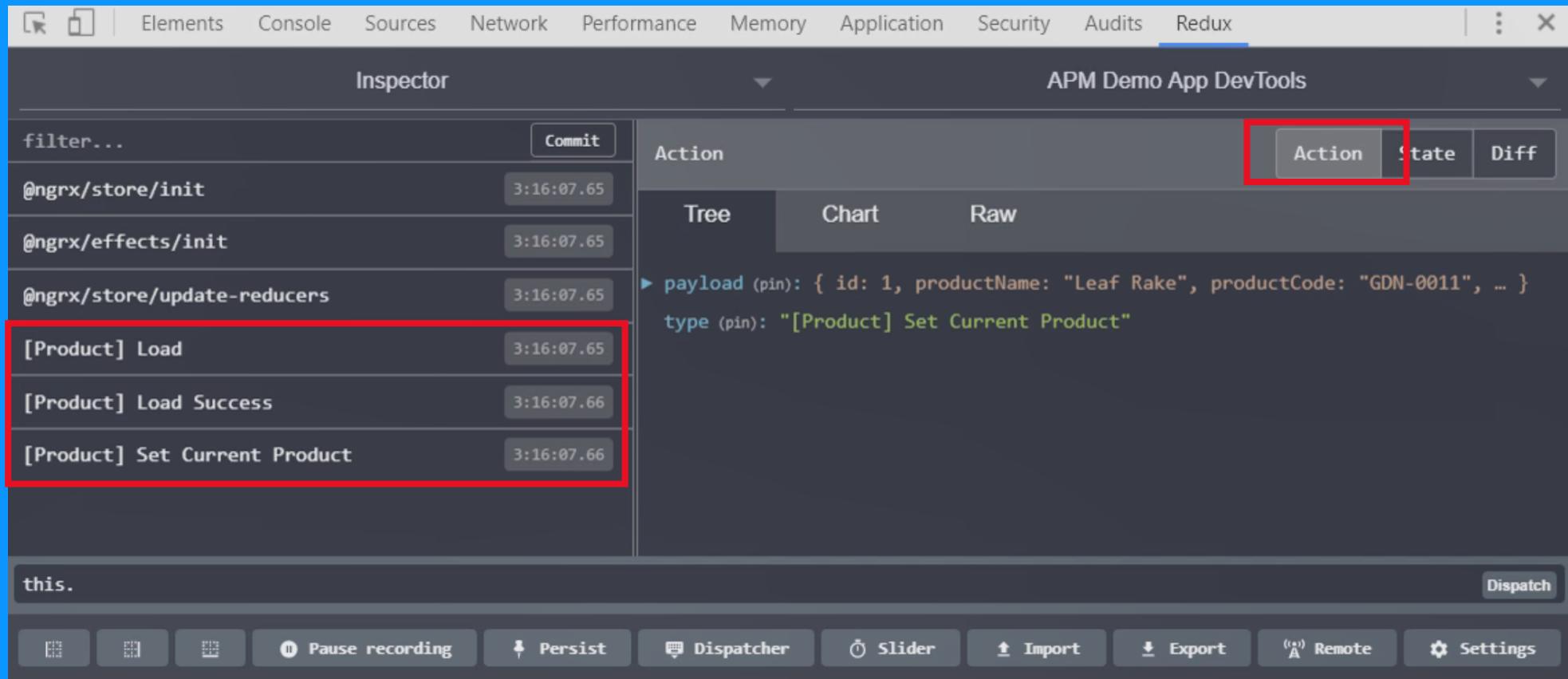
Name: Garden Cart  
Code: gdn-0023  
Description: 15 gallon capacity rolling garden cart  
Availability: March 18, 2017  
Price: \$32.99  
5 Star Rating: ★★★★

Edit

Notify the list itself to ensure that the selected product is highlighted

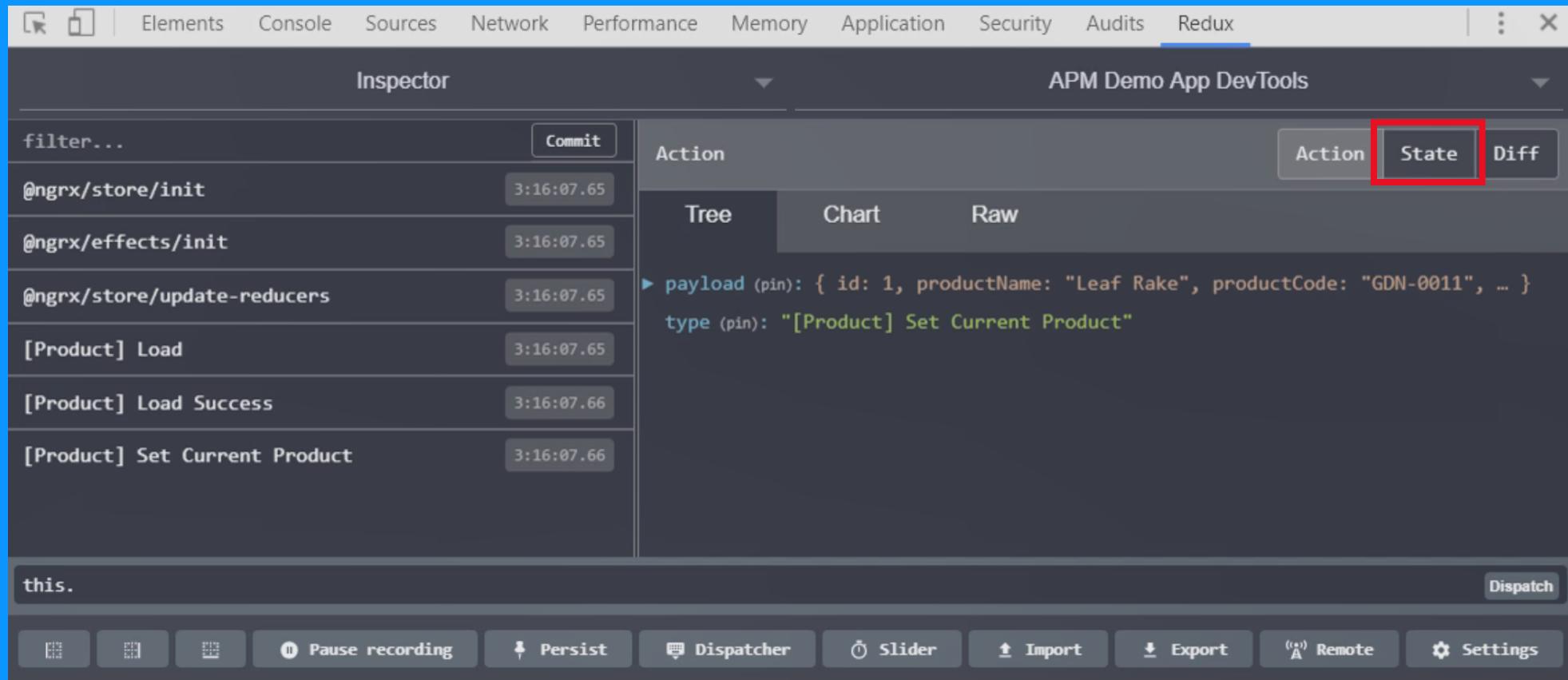
# Why Use NgRx?

## Great Tooling Support



# Why Use NgRx?

## Great Tooling Support



# Don't Use NgRX If...

## You are new to Angular

- It's recommended that you master Angular and RxJS Observables first

## Dealing with a simple application

- The complexity of NgRX may outweigh the benefits

## Have an established state management pattern

- If it's not broke, don't fix it

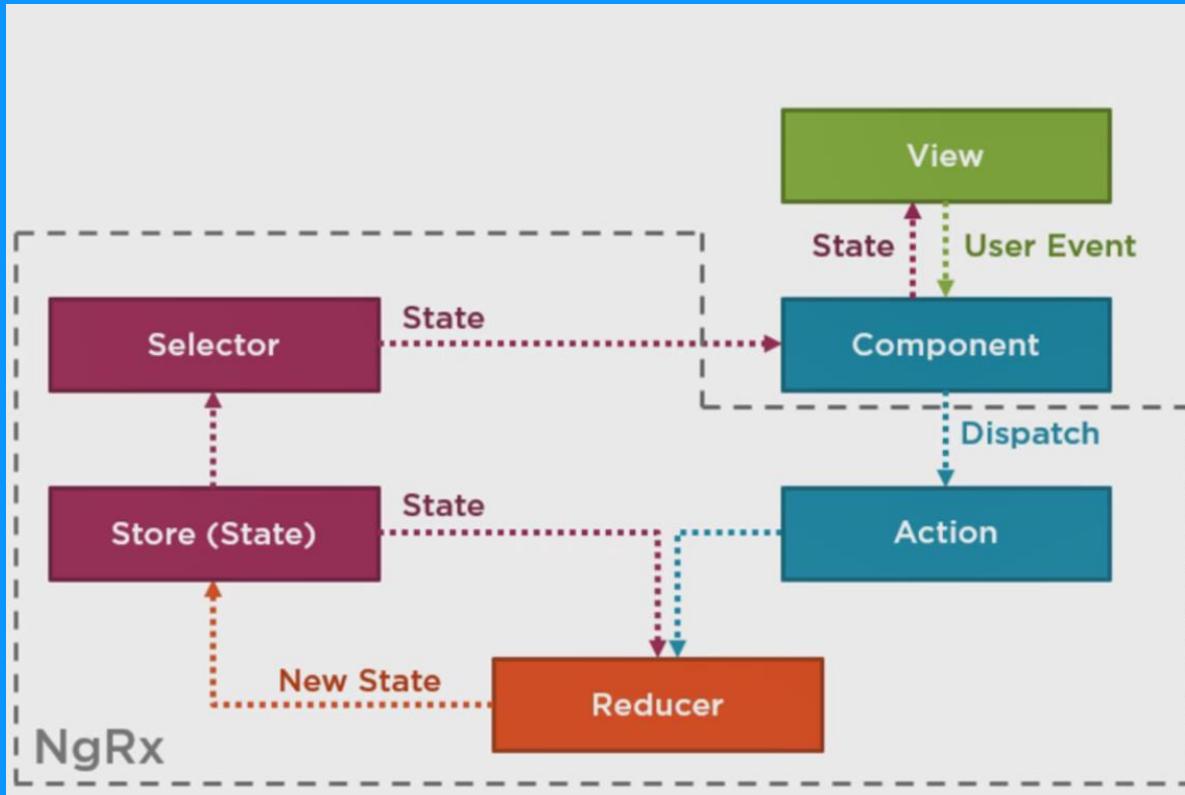
Demo

## Angular NgRx

- Install @ngrx/store

# Initializing The NgRX Store

In NgRX we have one store for all of the state

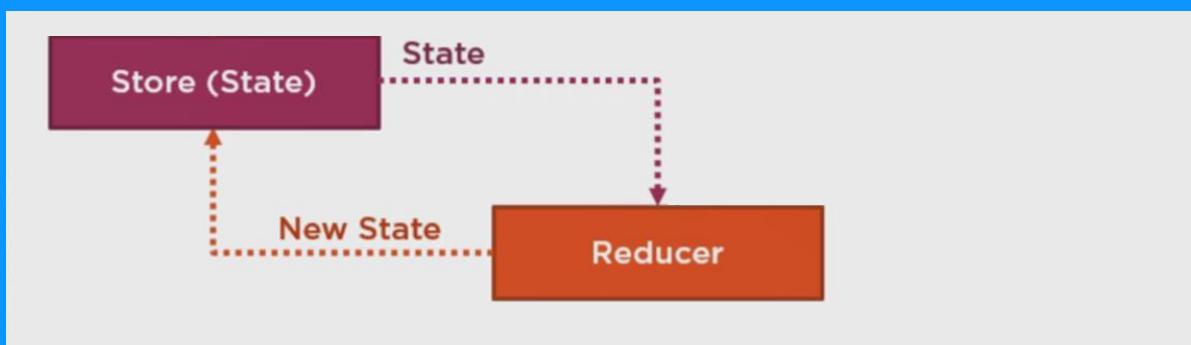


## App Module

```
import { StoreModule }  
       from '@ngrx/store';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    RouterModule.forRoot(appRoutes),  
    ...  
    StoreModule.forRoot(reducer)  
  ],  
  declarations: [ ... ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

# Initializing The NgRX Store

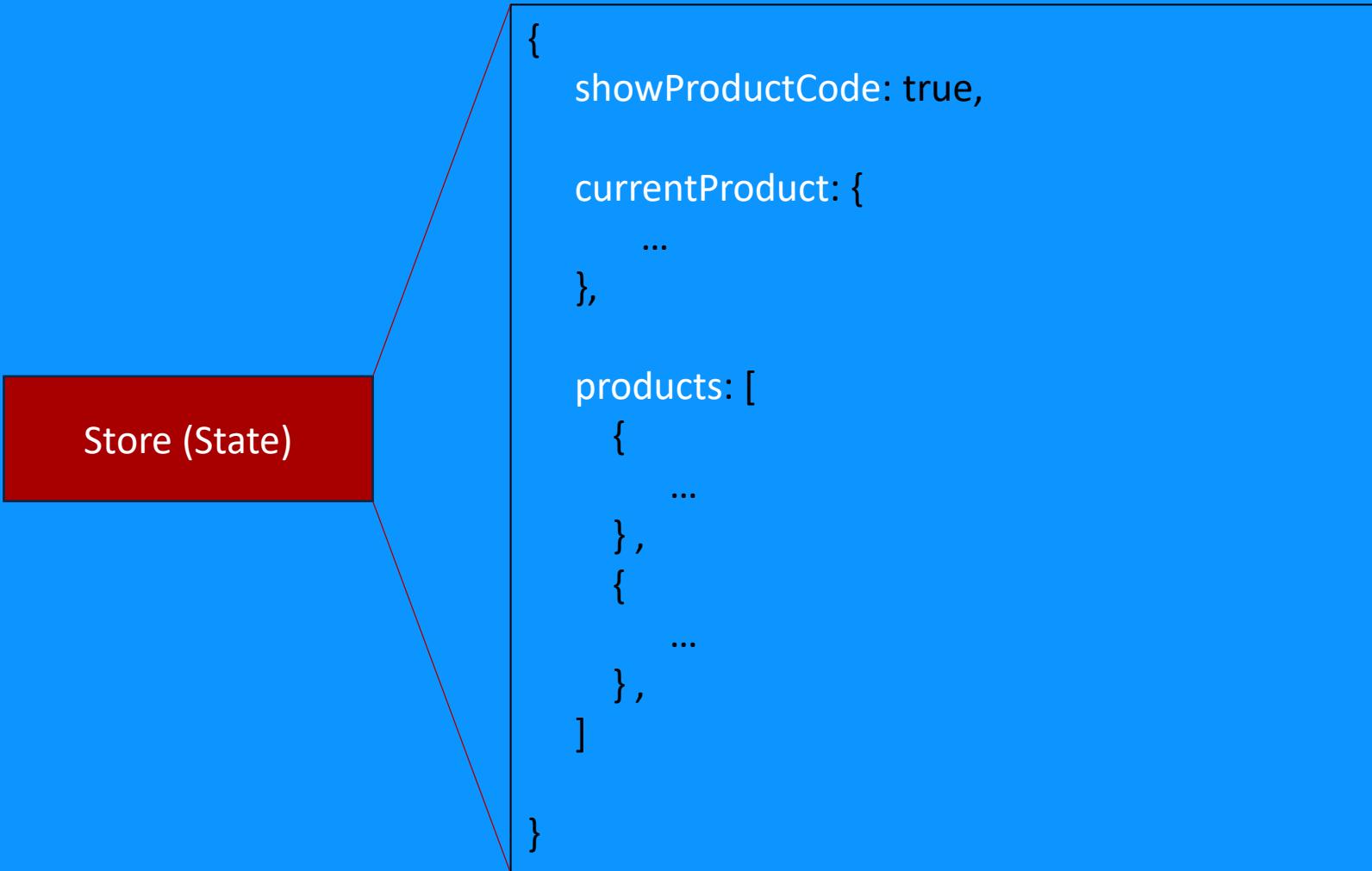
We could build a single reducer to work with that state



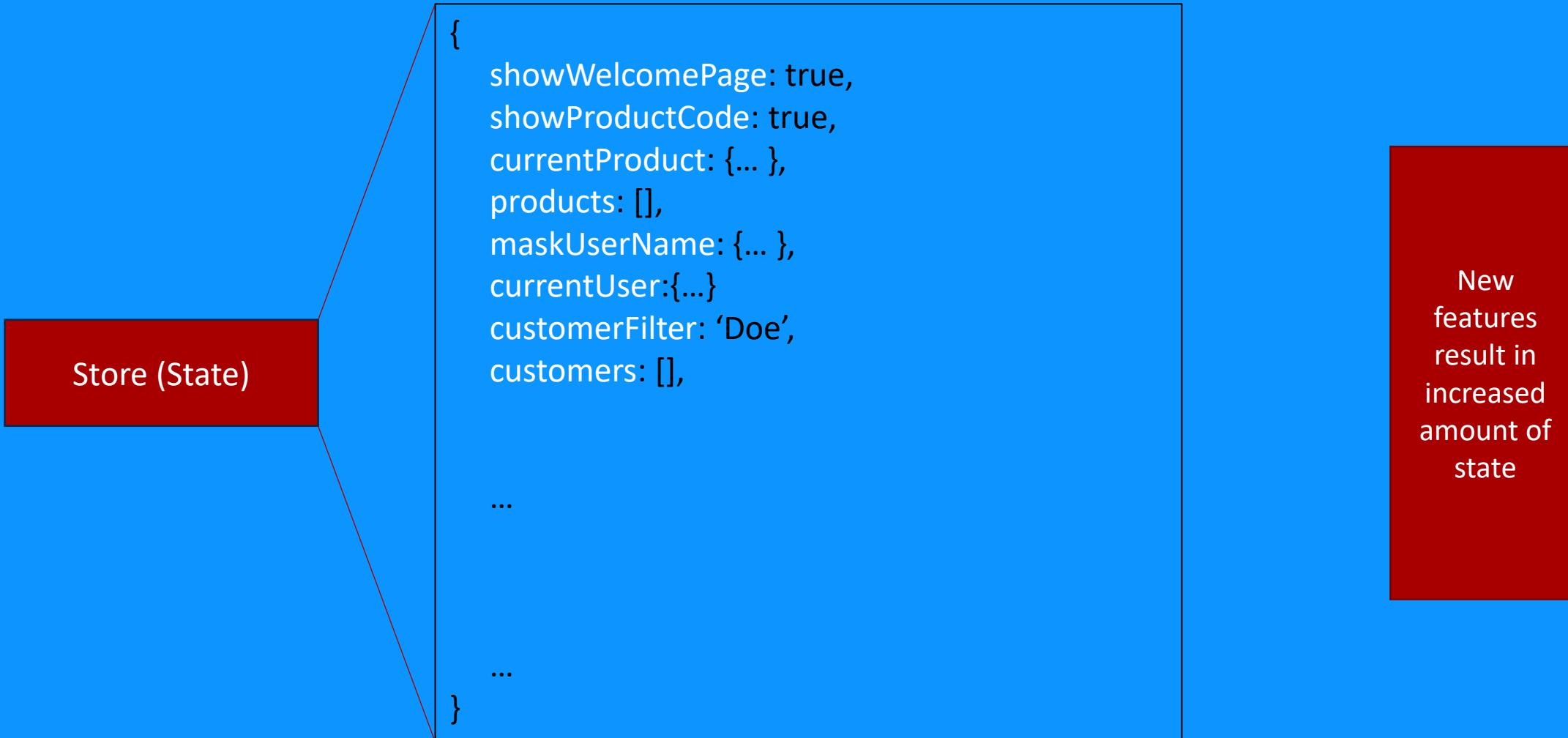
We don't want one large flat state

We don't want one huge reducer to manage all of this state

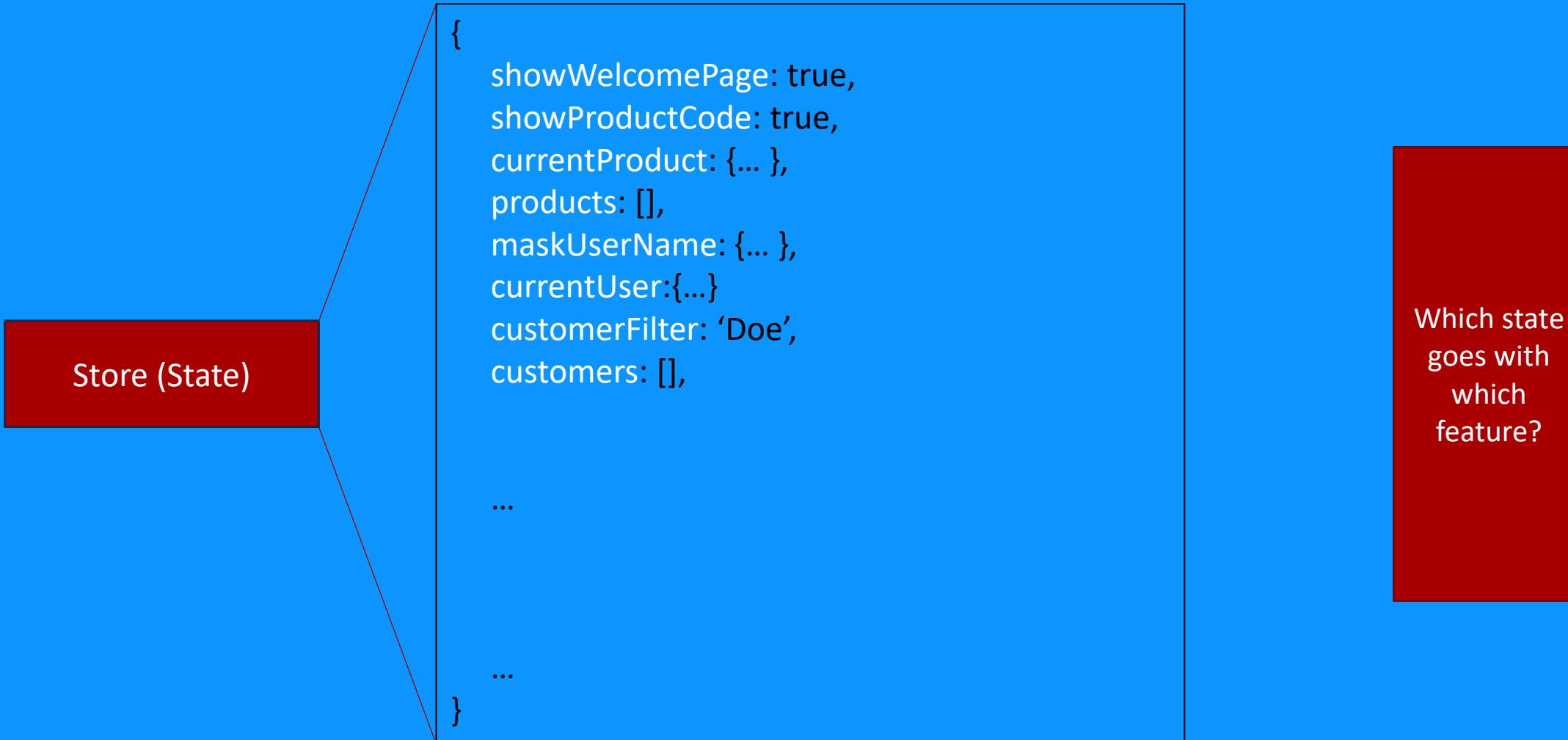
# NgRX Store State Structure



# NgRX Store State Structure



# NgRX Store State Structure



# NgRX Store State Structure

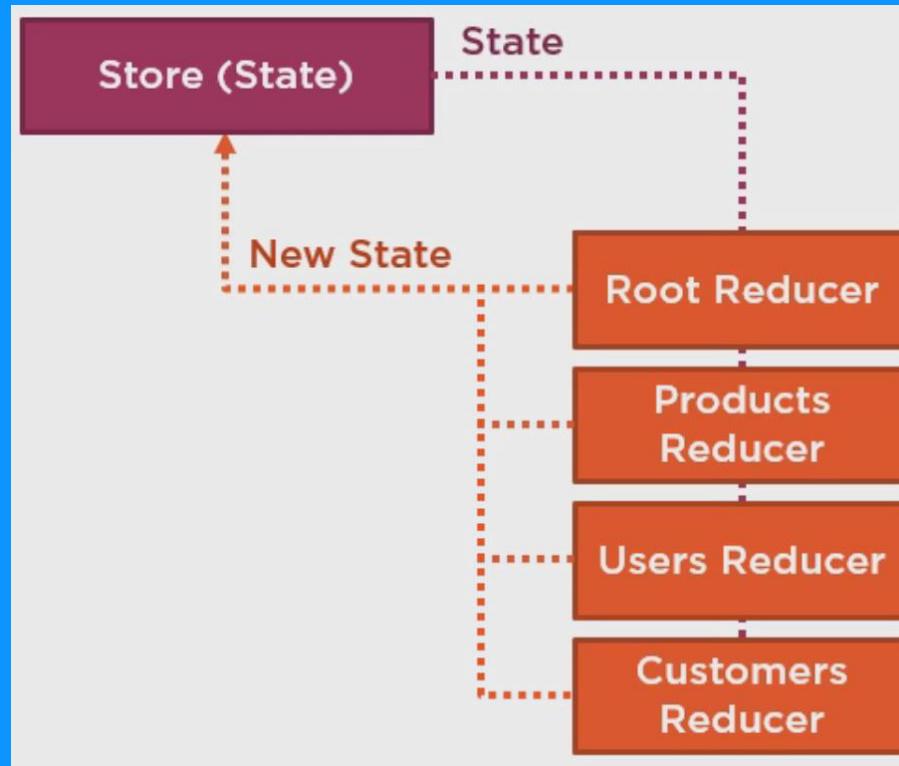
Store (State)

```
{  
  app: {  
    showWelcomePage: true,  
  },  
  products: {  
    showProductCode: true,  
    currentProduct: {...},  
    products: [],  
  },  
  users: {  
    maskUserName: {...},  
    currentUser: {...},  
  }  
  customers: {  
    customerFilter:'Doe'  
    customers: [],  
  }  
}
```

Angular applications are organized into feature modules, so it makes sense to layout the state by feature as well.

# Initializing The NgRX Store

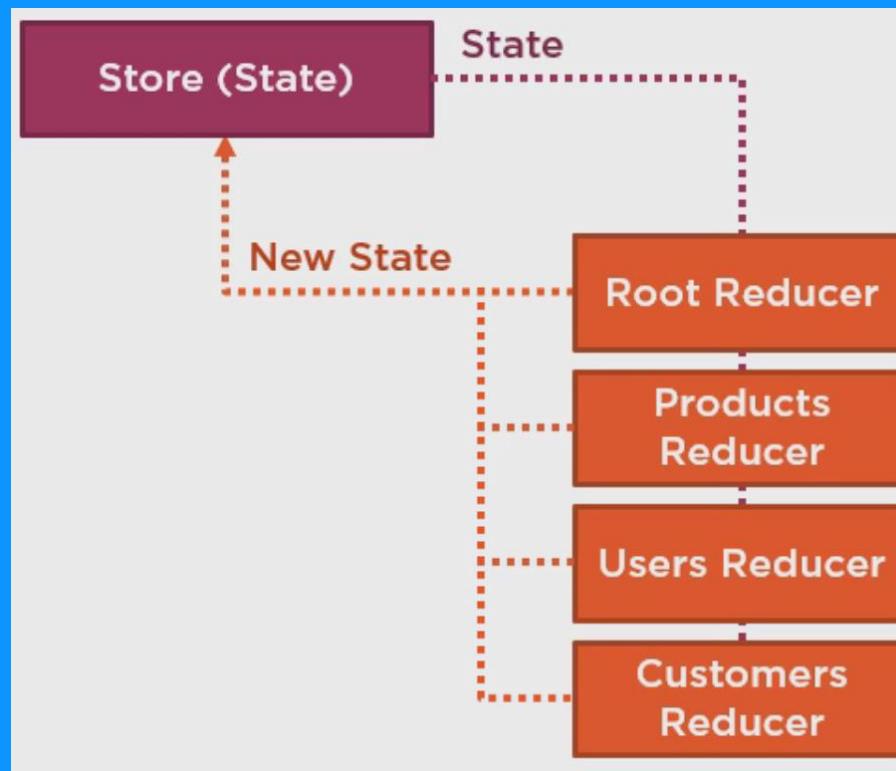
Define multiple reducers. One for each feature's slice of state.



```
{
  app: {
    showWelcomePage: true,
  },
  products: {
    showProductCode: true,
    currentProduct: {...},
    products: [],
  },
  users: {
    maskUserName: {...},
    currentUser: {...},
  },
  customers: {
    customerFilter:'Doe'
    customers: [],
  }
}
```

# Initializing The NgRX Store

Define multiple reducers. One for each feature's slice of state.



Each reducer is more focused



Easier to build



Easier to Maintain



Easier to Test



Data is never created for a module that is not loaded

# Feature Module State Composition

Initialize root app state using `forRoot` method

## App Module

```
import { StoreModule }  
       from '@ngrx/store';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    RouterModule.forRoot(appRoutes),  
    ...  
    StoreModule.forRoot(reducer)  
  ],  
  declarations: [ ... ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

Initialize each feature's state using `forFeature` method

## Product Module

```
import { StoreModule }  
       from '@ngrx/store';  
  
@NgModule({  
  imports: [  
    SharedModule,  
    RouterModule.forChild(productRoutes),  
    ...  
    StoreModule.forFeature('products', reducer)  
  ],  
  declarations: [ ... ],  
  providers: [ ... ]  
})  
export class ProductModule { }
```

# Feature Module State Composition

Notice the similarity between RouterModule and StoreModule

## App Module

```
import { StoreModule }  
      from '@ngrx/store';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    RouterModule.forRoot(appRoutes),  
    ...  
    StoreModule.forRoot(reducer)  
  ],  
  declarations: [ ... ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

## Product Module

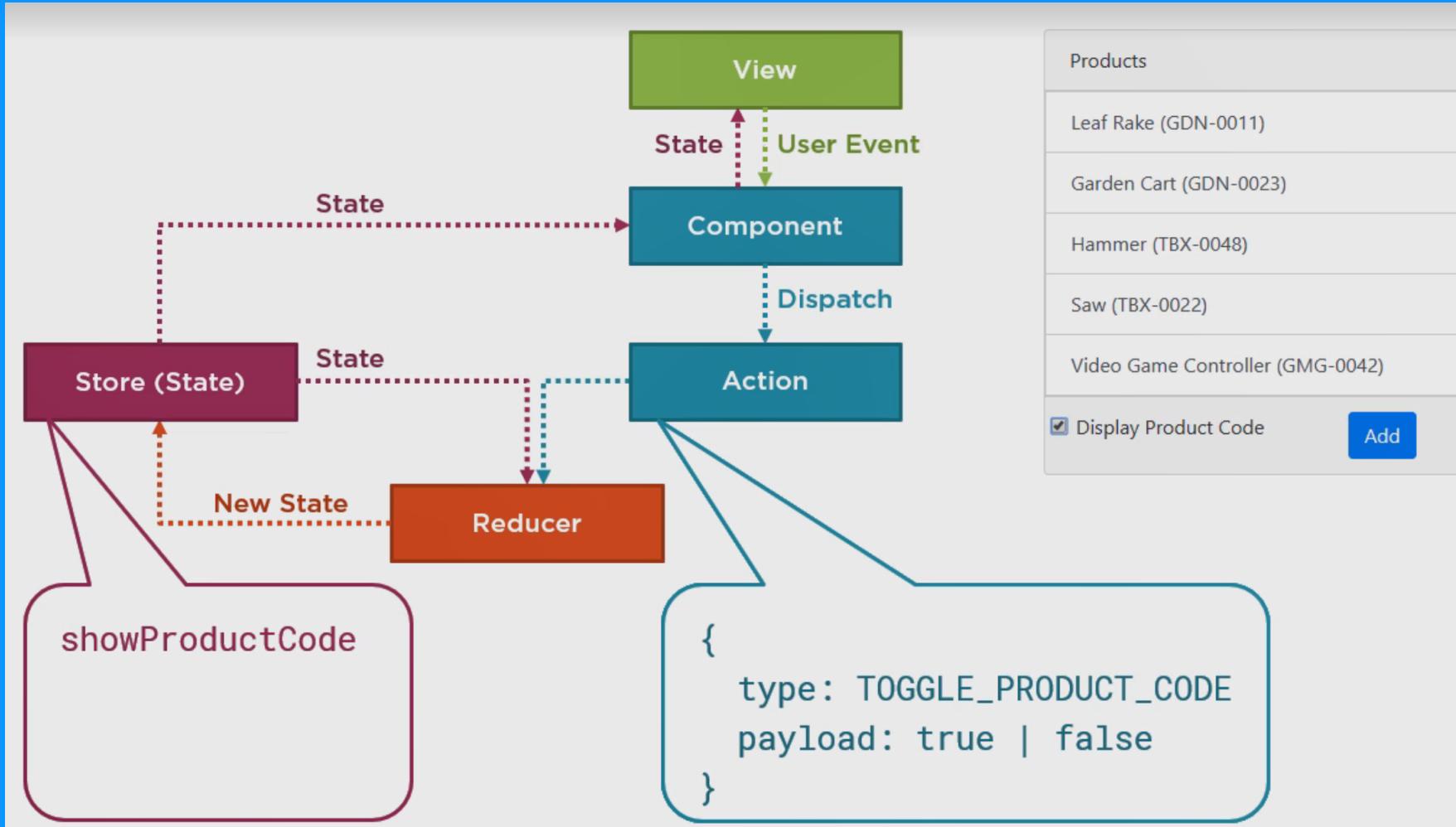
```
import { StoreModule }  
      from '@ngrx/store';  
  
@NgModule({  
  imports: [  
    SharedModule,  
    RouterModule.forChild(productRoutes),  
    ...  
    StoreModule.forFeature('products', reducer)  
  ],  
  declarations: [ ... ],  
  providers: [ ... ]  
})  
export class ProductModule { }
```

Demo

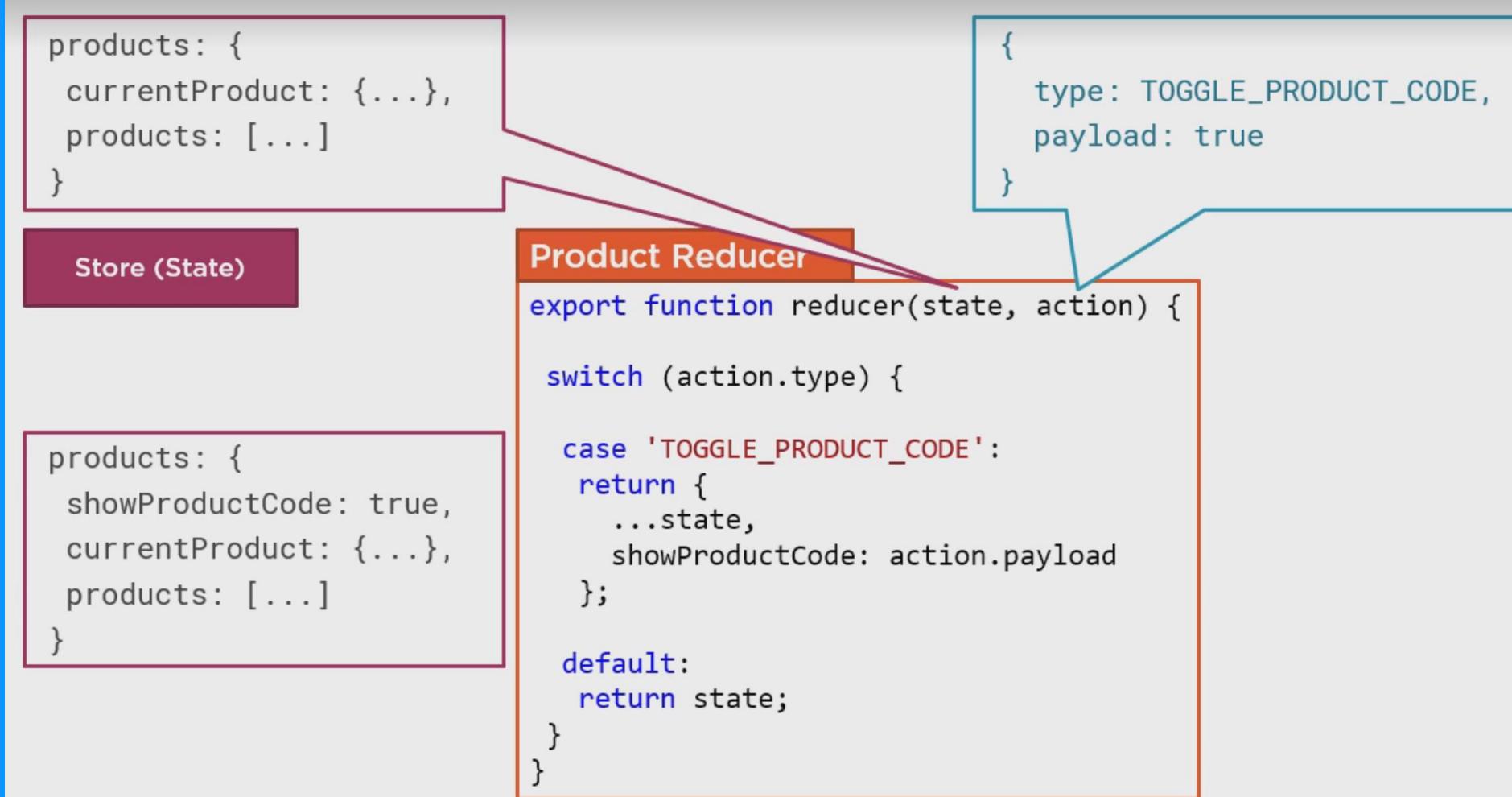
## Angular NgRx

- Install @ngrx/store
- Initialize the store

# Defining The State And Actions



# Building A Reducer To Process Actions

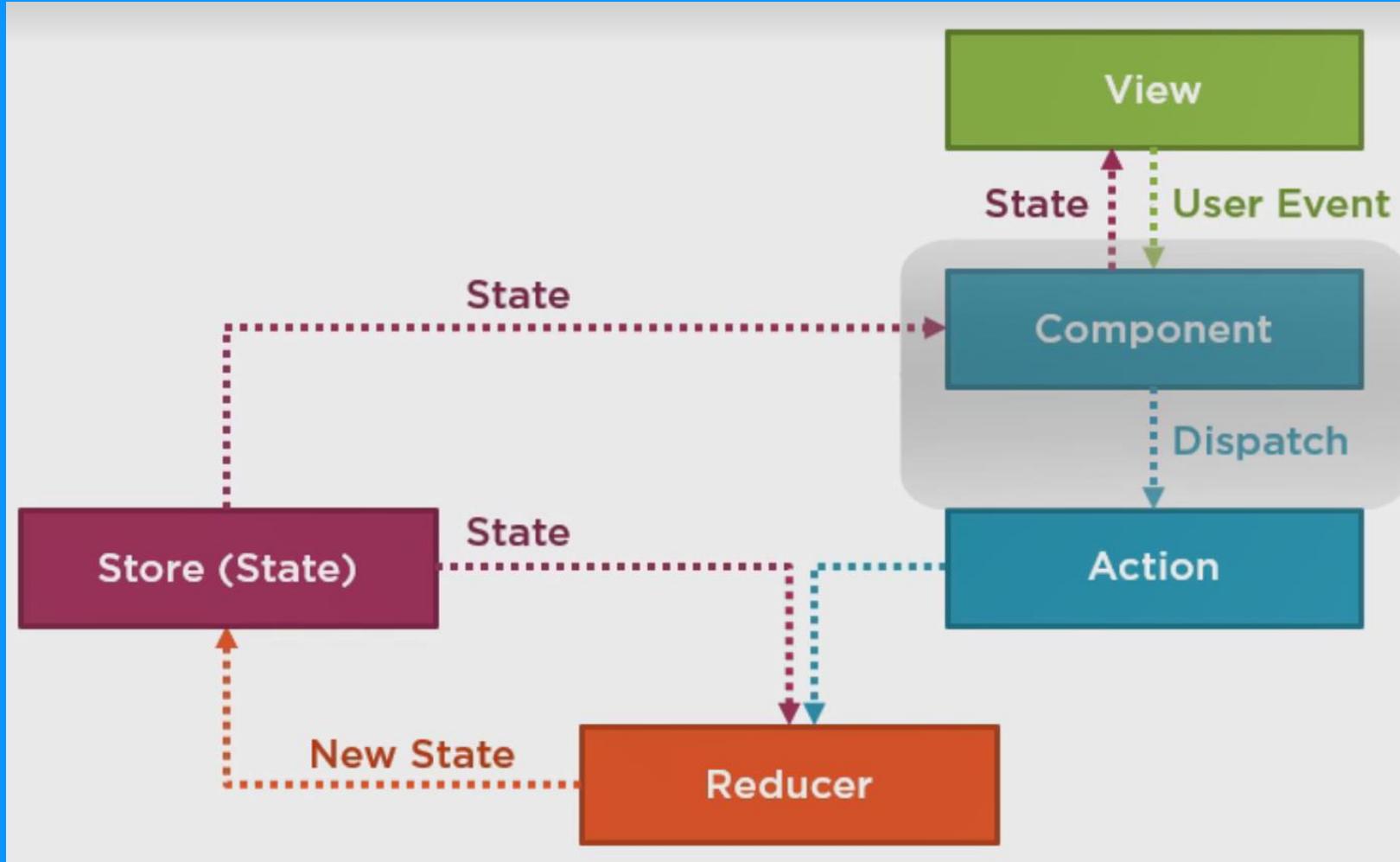


# Demo

## Angular NgRx

- Install `@ngrx/store`
- Initialize the store
- **Build a reducer to process actions and set store state**

# Dispatching An Action To Change State

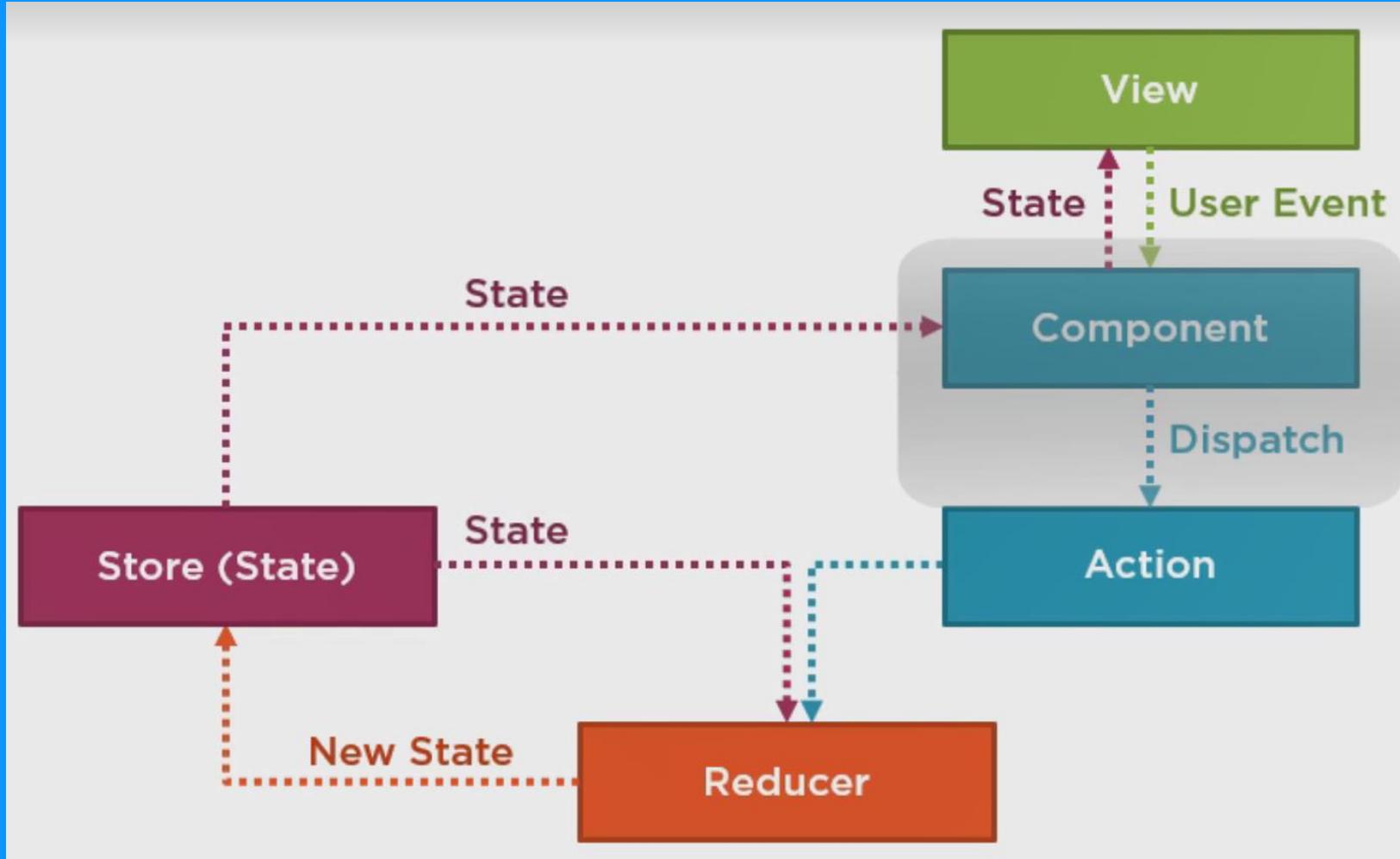


# Demo

## Angular NgRx

- Install `@ngrx/store`
- Initialize the store
- Build a reducer to process actions  
and set store state
- **Dispatch an action to change state**

# Subscribing To the Store To Get State Changes



# Demo

## Angular NgRx

- Install `@ngrx/store`
- Initialize the store
- Build a reducer to process actions and set store state
- Dispatch an action to change state
- **Subscribe To the Store To Get State Changes**

# Debugging NgRX With Redux DevTools

- Install browser Redux DevTools extension
- Install `@ngrx/store-devtools`
- Initialize `@ngrx/store-devtools` module

Demo

Debugging NgRX

# Ahead-Of-Time (AOT) vs Just-In-Time (JIT)

- An Angular application consists largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by the Angular compiler

# Ahead-Of-Time (AOT) vs Just-In-Time (JIT)

- There is actually only one Angular compiler. The difference between AOT and JIT is a matter of timing and tooling
- With AOT, the compiler runs once at build time using one set of libraries; with JIT it runs every time for every user at runtime using a different set of libraries

# Just-In-Time (JIT)

- Good for development

- Ship the Angular compiler to the browser, and dynamically compile the application

```
import {platformBrowserDynamic} from "@angular/platform-browser-dynamic";  
  
@NgModule({...})  
export class AppModule {  
  
}  
  
platformBrowserDynamic().bootstrapModule(AppModule);
```

This will compile all templates and bootstrap the application. This will come at the expense of a much larger application bundle, but that is OK when the server is actually running in your own development machine

# Just-In-Time (JIT)

- Compilation can uncover many component-template binding errors. JIT compilation discovers them at runtime, which is late in the process

# Ahead-Of-Time (AOT)

- Good for production
- Use the module information to do ahead of time compilation

# Ahead-Of-Time (AOT)

- Advantages of AOT

- Faster rendering
- With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.
- Fewer asynchronous requests. The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.
- Smaller Angular framework download size
- There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

# Ahead-Of-Time (AOT)

- Advantages of AOT

- Detect template errors earlier. The AOT compiler detects and reports template binding errors during the build step before users can see them.
- Better security. AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

# Ahead-Of-Time (AOT)

- You can introduce AOT manually using the [following](#) steps
- Or use the Angular CLI to make the process transparent:
  - **ng serve** will serve the app in Just In Time Mode
  - **ng build --prod && ng serve --prod** will serve the app in Ahead Of Time compilation mode

# Comparing Dev and Prod Build Targets

	ng build	ng build --prod
Environment	environment.ts	environment.prod.ts
Cache-busting	only images referenced in css	all build files
Source maps	generated	not generated
Extracted CSS	global CSS output to .js	yes, to css file(s)
Uglification	no	yes
Tree-Shaking	no	yes
AOT	no	yes
Bundling	yes	yes

# Ahead-Of-Time (AOT)

- You can also use the --build-optimizer flag when building to further optimize the build
- Since Angular CLI 1.5, the Build Optimizer is on by default when doing a production build.

Demo

Ahead Of Time Compilation

# CLI Environment Options

- The new angular-cli has the concept of different environments like development (dev) and production (prod)
- By default two environments are created
  - environment.ts
  - environment.prod.ts

Demo

## CLI Environment Options

# Debugging

- There are various methods to debug an Angular application:
  - Console
  - Debugger
  - JSON Pipe
  - Augury
  - Logging

# Debugging

- Ensure that development mode is enabled as it allows errors to be displayed in the console and the usage of breakpoints
- When you open your console in debugging mode, you would normally see this message pop up:

```
Angular 2 is running in the development mode. Call enableProdMode() to enable the production mode
```

Demo

Debugging Angular

# Testing

- We have three types of tests

Unit Tests

Integration Tests

End to End Tests

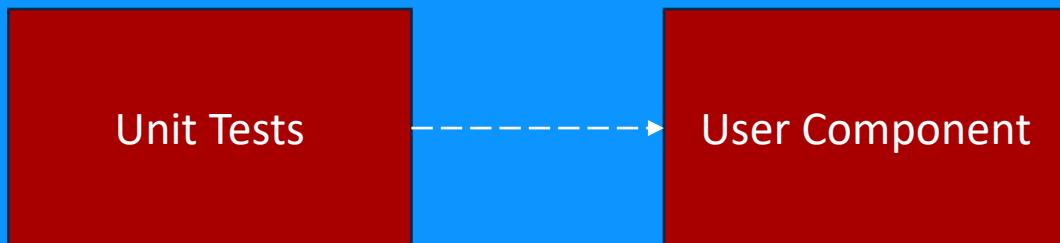
# Unit Tests

- Test a single “unit” of code
- “Unit” can mean a different thing to different people
- Generally the accepted unit of code is a single class

# Unit Tests

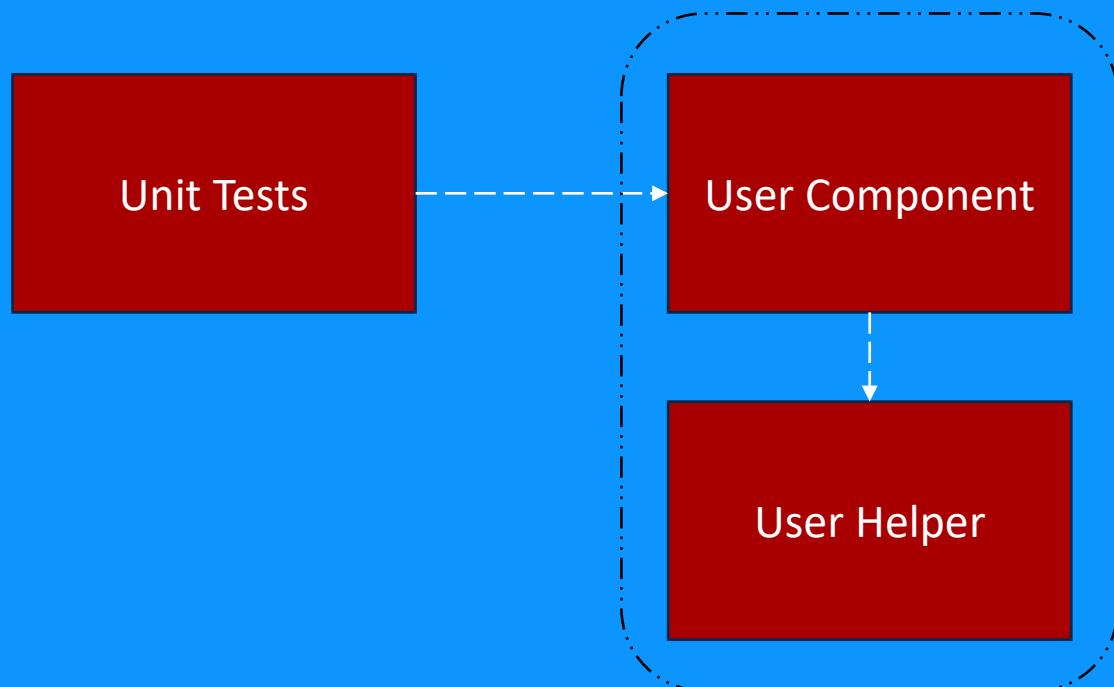
- In Angular world this means testing your Component in isolation without its Template and any other resources that touch external resources (e.g. file system, database, API endpoints)

# Unit Tests



# Unit Tests

- Again there could be an argument of what is a unit. For example if you have a little helper class you may test it together with your component and consider it a unit test



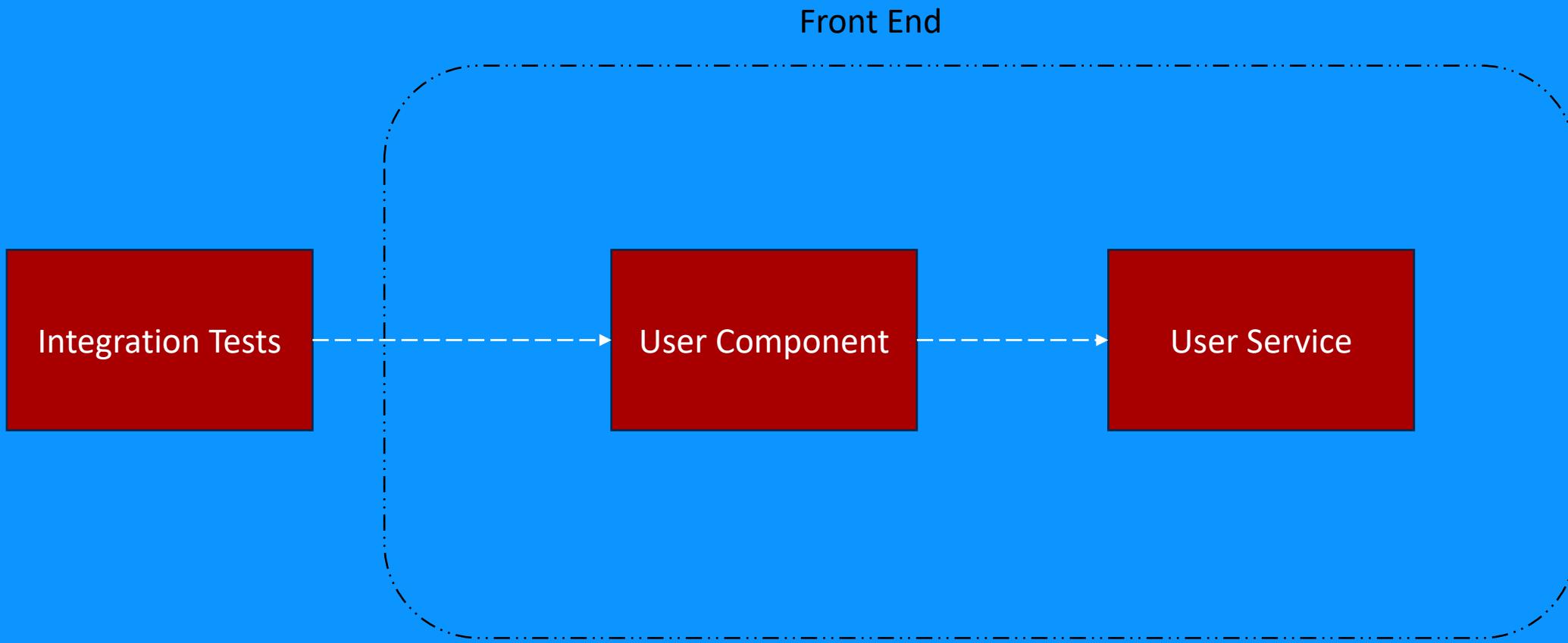
# Unit Tests

- Easiest to write
- Fast
- Doesn't give us much confidence

# Integration Tests

- More than a unit but less than the complete application
- It usually means at least two units are working together
- The concept is typically vague and can mean a lot of different things to a lot of different people

# Integration Tests

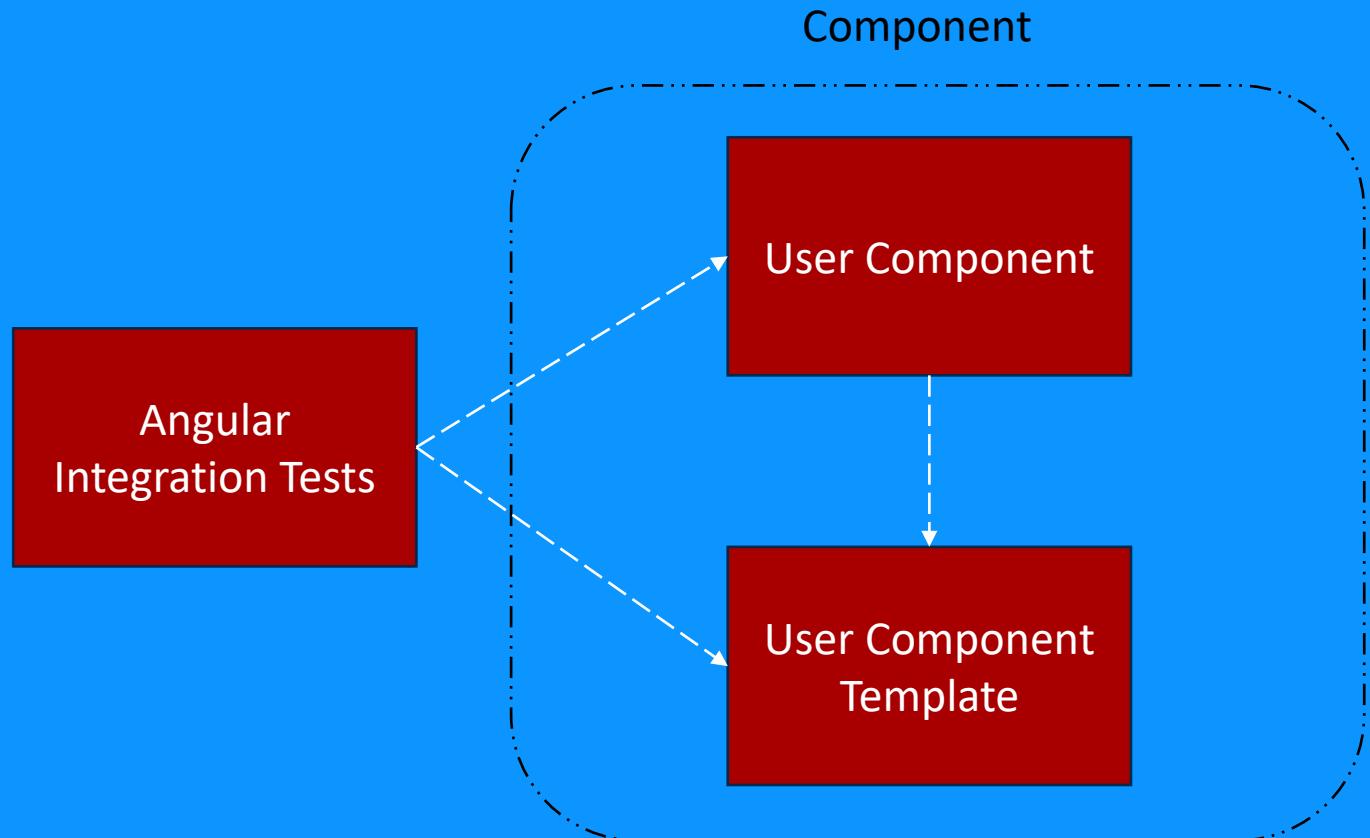


# What About Testing A Component With Its Template?

- Components in Angular have templates
- So Angular has tooling that allows for a special kind of test which they call Angular integration test
- It uses the template and the component together to make sure that these two pieces are working correctly together

# What About Testing A Component With Its Template?

- Is this truly an integration test or is it just another kind of unit test?
- We have two different pieces which are the same unit of code
- That is up to you decide but the name of this type of testing is officially called integration testing

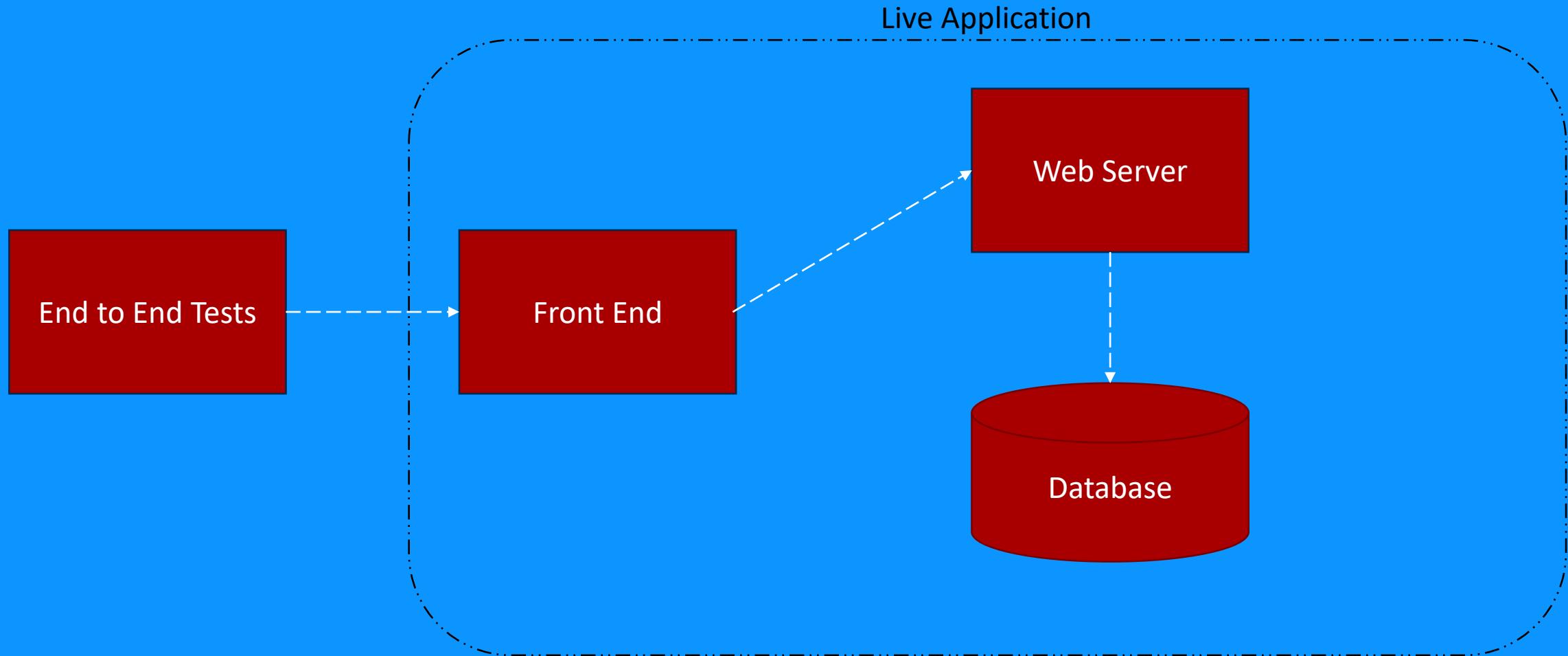


# End to End Tests

- Test the entire application as a whole

- We simulate a real user
- We launch the application in the browser
- Navigate between pages
- Maybe Even log in

# End to End Tests



# End to End Tests

- More confidence
- Very slow
  - Each test is going to launch the application in the browser, navigate to the home page, then perhaps do a few clicks here and there and do something
- Very fragile
  - A simple change in the application markup can break these tests even if the application is working properly

# Ideal Testing Plan

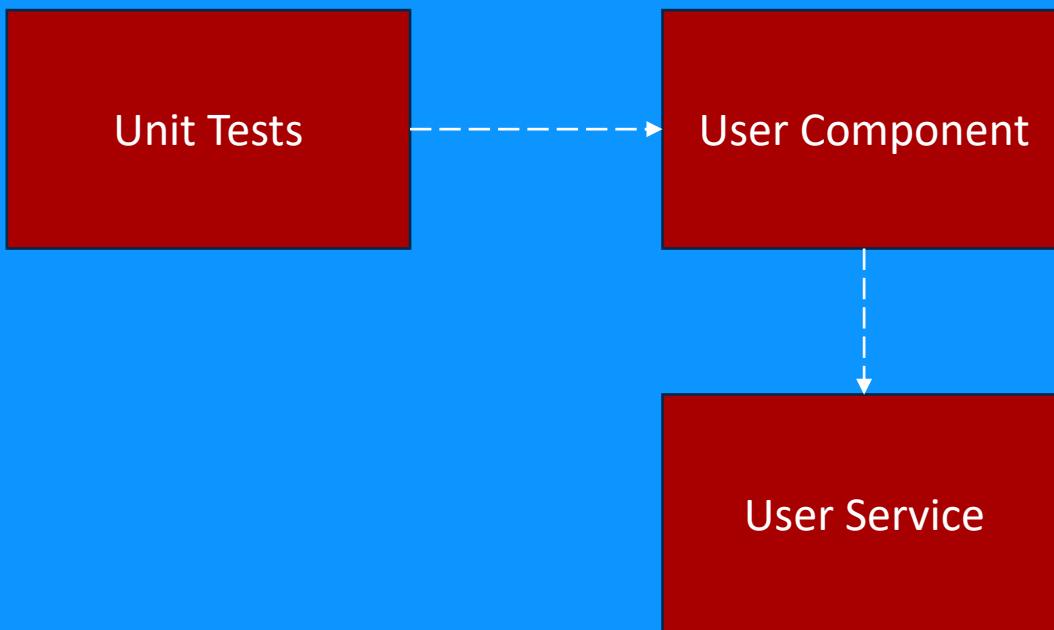
A lot of Unit/Integration Tests

Few  
End to end Tests

Only for the key functions  
of the application

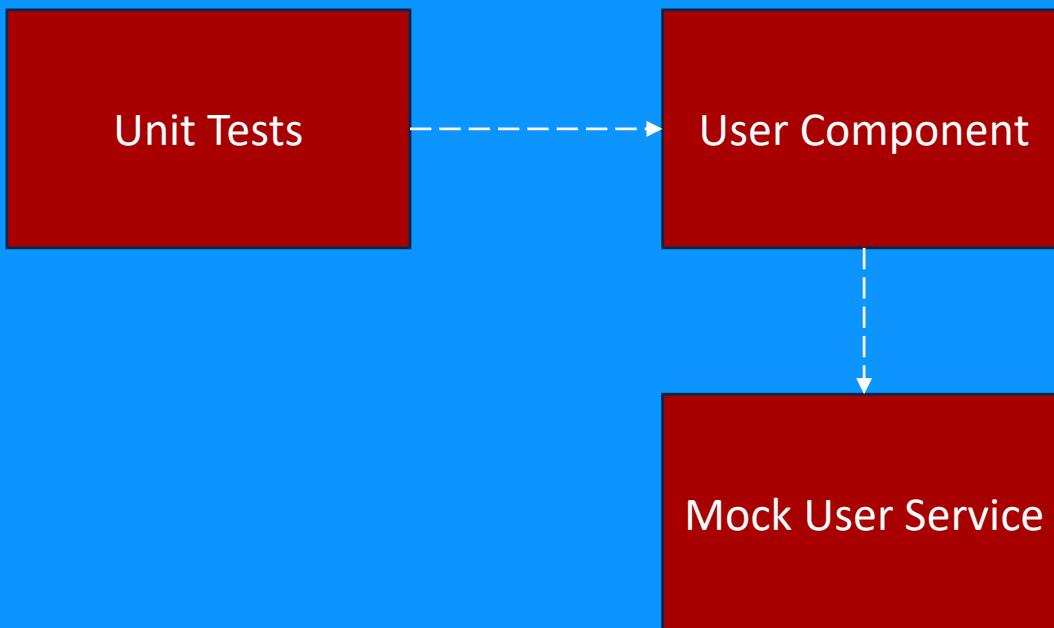
# Mocking

- Important concept in unit testing as it ensures that we are only testing a single unit of code at a time
- Most classes have dependencies and don't usually work by themselves
  - E.g. a user service is injected into the component



# Mocking

- When writing a unit test you don't want to use the real service for several reasons
  - You are just trying to test the user component
  - User service might make http calls which you don't want in your unit tests
- Mocks allow you to draw a boundary around your user component
  - Instead of using the real user service you will use the mock user service



# Types of Mocks

- Although mock is used as a generic term there are several types of objects that do various things related to mocking
  - Dummies are the simplest type. Dummies are just objects that are used in place of a real ones. For example if a method call requires a parameter that is an object but doesn't care what that object is
  - Stubs are objects that have controllable behavior. If we call a certain method on a stub we can decide in our test what value that method call will return
  - Spies are objects that keep track of which of its methods were called and how many times they were called and what parameters were used for each call. Most of the time these are the type of objects that we use when we need a mock. Many times the boundaries between the three mock types can be a little blurred. We may use objects that have behaviors of both spies and stubs
  - True Mocks
    - More complex objects that verify that they were used in a specific way
    - For example they can check only a specific method was called and was called only once and it had some very specific parameters. They are able to do this to themselves
    - They are bit more difficult to work with and are usually overkill for what most unit tests need

# Testing In Angular

# Types of Angular Tests

- Isolated tests are basic unit tests
  - We isolate a single unit of code like a class of a component, service, or pipe
  - We construct the class by hand and we give it its construction parameters ourselves
- Integration tests are a bit more complex
  - In an angular integration test we actually create a module in which we put just the code we are going to test
  - Generally just one component. But we actually test that in the context of an Angular module
  - This is used so we can test the component with its template
  - Two types of Integration Tests
    - Shallow – We only test a single component without its child components
    - Deep – Test the component along with its child components

# Testing Tools for Angular

- In Angular the CLI sets up testing and it uses two different tools
- Jasmine
  - The tool used to make sure that the tests are working as expected using expectations
  - Its also used to create mocks
- Karma
  - Is a test runner which executes unit tests in a browser (supports headless browsers as well)

# Other Testing Tools

- Mocha/chai/etc
  - Those are replacements for Jasmine
  - They are popular and easy to drop in and replace Jasmine with
- Sinon
  - Specialized Mocking library
  - If you find that the Mocking capabilities of Jasmine are insufficient then you can replace them with Sinon
- TestDouble
  - Competitor to Sinon that is gaining some popularity but is still less popular than Sinon
- Wallaby
  - This is a paid tool
  - It allows you to see the code coverage for your tests right in your IDE
- Cypress
  - Traditionally considered to be an end to end testing tool
  - There are development capabilities to do integration types of testing

Demo

Unit Testing

Demo

Mocking

# Angular Testing APIs

- TestBed
  - Configures and initializes environment for unit testing and provides methods for creating components and services in unit tests
- ComponentFixture
  - Has a bit more properties than what a component has
  - Has a property called nativeElement which gets a handle to the DOM element that represents the container for the template
- nativeElement
  - Is a standard html DOM element. This is the same API that you would use inside of old plain JavaScript if you were manipulating the DOM
  - For example you can use the nativeElement.querySelector('a') to choose an anchor element

# Angular Testing APIs

- debugElement
  - Is a wrapper around the DOM node that has a set of functionalities that is very similar to the nativeElement
  - Compared to nativeElement, it exposes additional properties that can be used for other purposes. E.g. if the element has a directive then the debugElement provides a way to access that directive
  - Another scenario where the debugElement comes in handy if for example you want to get access to the component that the element belongs to
  - Even though this can be achieved via the fixture sometimes its still handy if you have multiple components and you want to detect the exact component that the element belongs to
- detectChanges
  - The bindings in the component template won't get updated until change detection runs
  - Runs change detection and updates any bindings that may exist on the component

Demo

Unit Testing In Angular

Demo

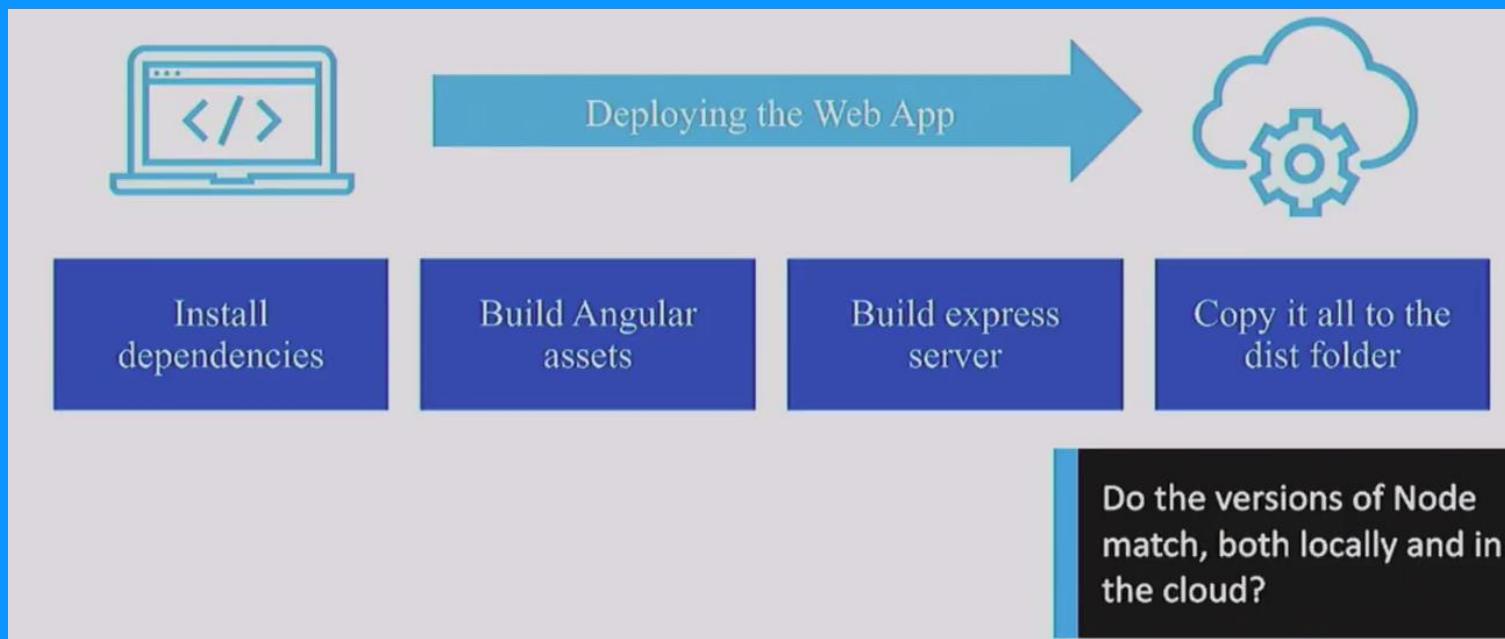
Unit Testing Integration With Azure DevOps

# Lab 14

Testing

# Containerization

- Code works on my machine
- How do we make it work everywhere?



# Containerization

- Automate the build and deploy steps with consistency and confidence

# Containerization

- Docker Containers

- Dockerfile

- Self contained script
- Provides a recipe for creating and running our app
- Configurable to adapt to different environments (what if I needed to hit different databases or use different keys under different environments?)

# Running Angular in Docker

- After creating a *Dockerfile* I should still have the ability to *Build*, *Run*, and *Debug* locally before I push to cloud

# Demo

## Running Angular inside a Docker Container

# Progressive Web Applications (PWA)

- Progressive Web Apps is a name given to a set of new W3C standards that allow any web application to feel and behave very much like a native app on a mobile device:
  - Offline caching with service workers so your app can work without an internet connection
  - Application manifest to define the look and feel of your app (splash screen, icons, name, full screen or not)
  - Install to home screen feature so your web app can be accessed just like any other native app
  - Web notifications so you can send background notifications to the user, just like native mobile apps do!

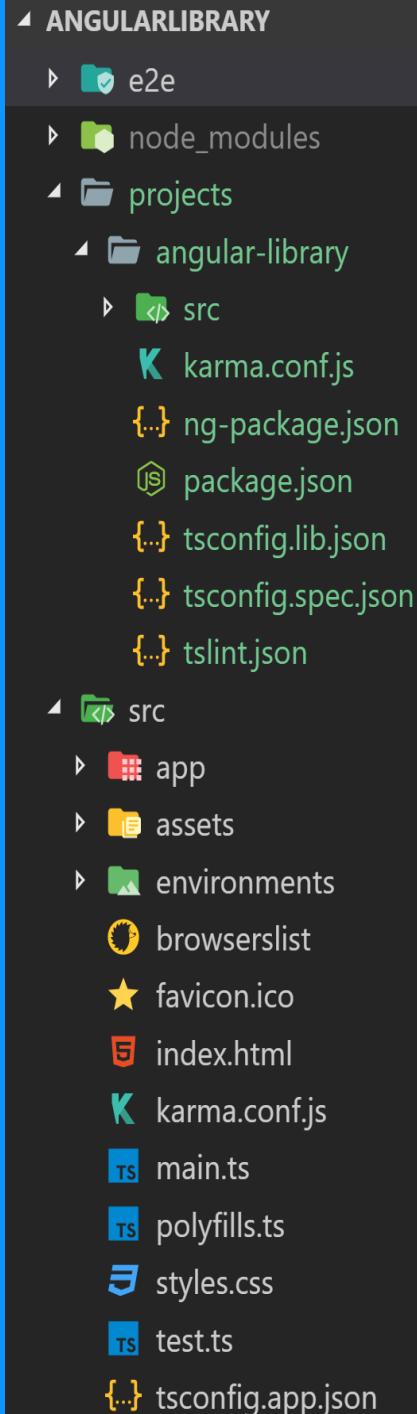
Demo

Angular PWA



# Create an Angular Library

- **ng new** creates a new workspace
- **Workspace** contains two projects:
  1. **Library project** – components and services that will be published as a package
  1. **Application project** – test harness / example usage for the library



# Steps to Get Started

- **ng new AngularLibraryApp**
- Rename the AngularLibraryApp folder to AngularLibrary
- Reopen VS Code in AngularLibraryApp folder

## angular.json

```
"projects": {  
  "AngularLibraryApp": { . . . },  
  "AngularLibraryApp-e2e": { . . . }  
}
```

# Use Angular-CLI To Create Library Project

```
ng generate library AngularLibrary --prefix=la
```

- Generates starter code for library in projects/angular-library folder
- **angular.json** - adds a new **AngularLibrary** project
- **package.json** - adds dev-dependencies for **ng-packagr**
- **tsconfig.json** - adds reference to the **angular-library** build path

## angular.json

```
"projects": {  
  "AngularLibraryApp": { . . . },  
  "AngularLibraryApp-e2e": { . . . },  
  "AngularLibrary": {  
    "root": "projects/angular-library",  
    "sourceRoot": "projects/angular-library/src",  
    "projectType": "library",  
    "prefix": "la",  
    "architect": {  
      "build": {  
        "builder": "@angular-devkit/build-ng-packagr:build",  
        "options": {  
          "tsConfig": "projects/angular-library/tsconfig.lib.json",  
          "project": "projects/angular-library/ng-package.json"  
        }  
      },  
    },  
  },  
},
```

# Build library

```
ng build AngularLibrary
```

- Automatically does a production build into dist folder

# Consume library

## app.module.ts

```
import { AngularLibraryModule } from 'angular-library';
```

```
@NgModule({  
  declarations: [AppComponent],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    AngularLibraryModule  
,  
    providers: [],  
    bootstrap: [AppComponent]  
)  
export class AppModule { }
```

## app.component.html

```
<la-AngularLibrary></la-AngularLibrary>
```

# Demo

## Consume Angular Library In Test Harness

# Add To The Library

```
ng generate component second-one --project=AngularLibrary
```

- Then add component to exports section of the library's module
- Also add component to entry file: public\_api.ts

# Demo

## Adding Another Component To Angular Library

# Package The library

## package.json

```
"scripts": {  
  . . .  
  "pack_lib": "cd dist/angular-library && npm pack"  
},
```

**npm run pack\_lib**

Generates angular-library-0.0.1.tgz

```
"scripts": {  
  . . .  
  "pack_lib": "cd dist/angular-library && npm pack",  
  "package": "npm run build_lib && npm run pack_lib"  
},
```

**npm run package**

# Consume Library Package

```
ng new AngularLibraryTest
```

```
npm install ../AngularLibrary/dist/angular-library/angular-library-0.0.1.tgz
```

app.module.ts

```
import { AngularLibraryModule } from 'angular-library';

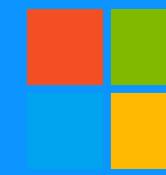
@NgModule({
  . . .,
  imports: [
    . . .,
    AngularLibraryModule
  ],
  . . .,
})
export class AppModule { }
```

app.component.html

```
<la-AngularLibrary></la-AngularLibrary>
```

# Demo

## Consume Angular Library In A New Application



Microsoft