Microsoft Services

Angular

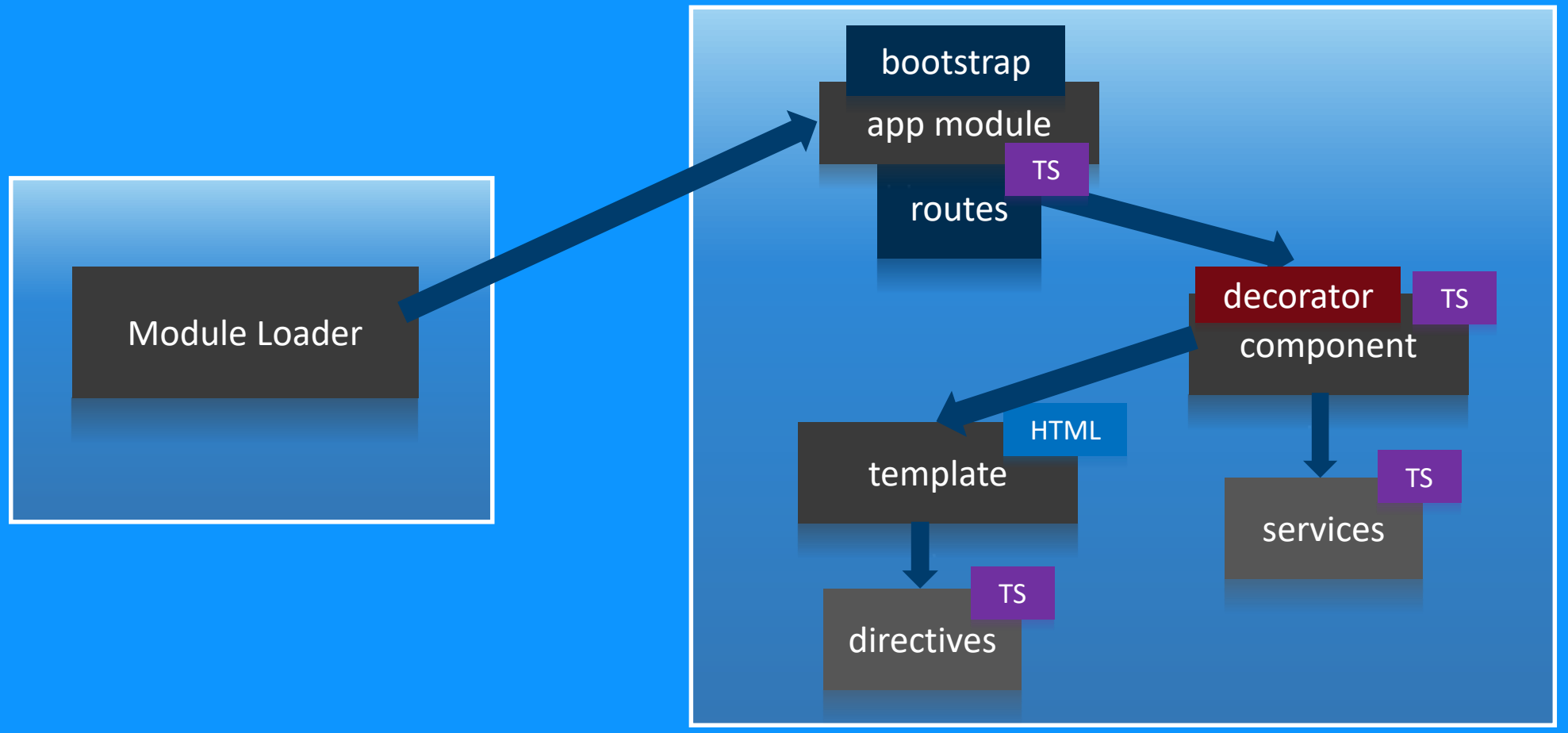
Laurie Atkinson
Senior Consultant
Development Advisory Services

Microsoft

Agenda

- Set Up Environment
- Modules
- Components
  - Templates
  - Classes
  - Data Binding
- Pipes
- Directives
- Services
- Http Service
- Routing
- Forms
- Lazy Loading Modules
- Unit Testing
- ASP.NET Core Web API

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

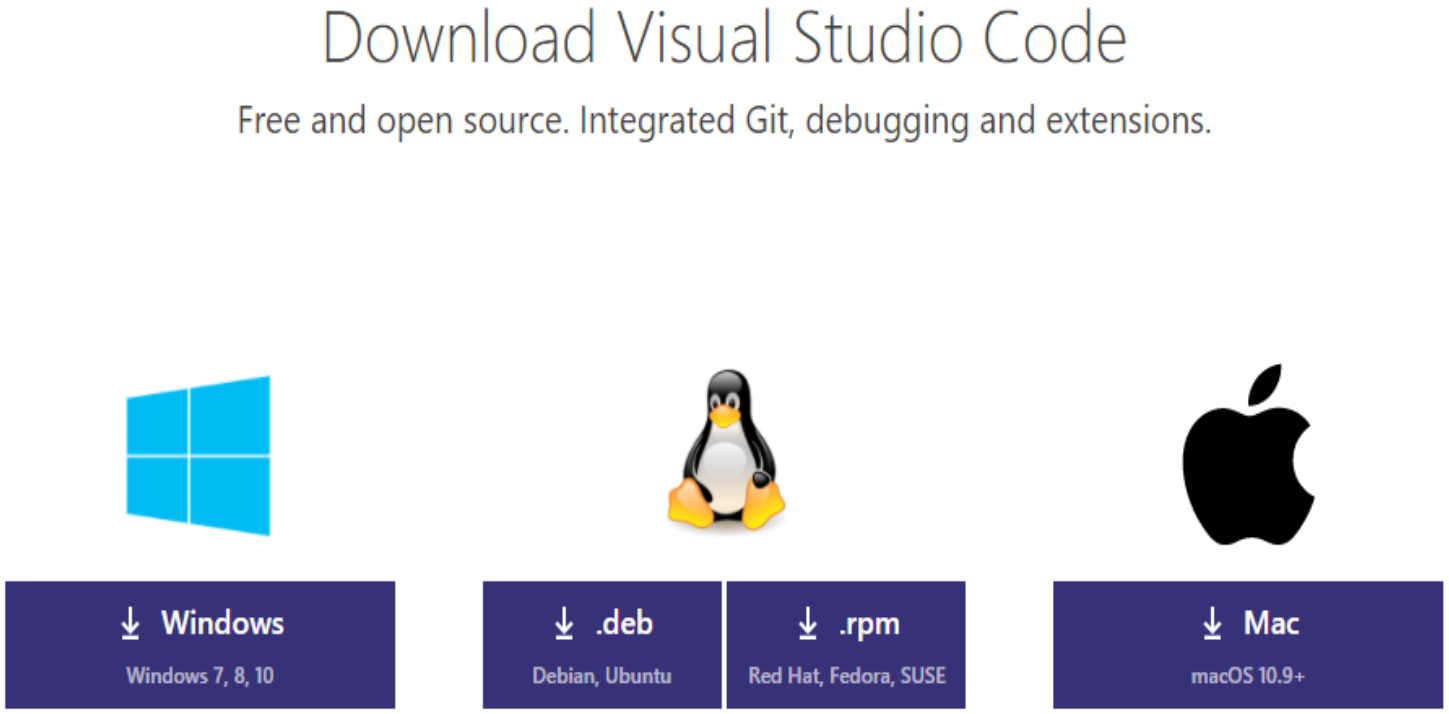
- Pick the language
  - JavaScript - ES5 (supported in most browsers)
  - JavaScript - ES6, now called ES2015 (must be transpiled)
  - TypeScript – superset of JavaScript (must be transpiled)
- Pick an editor
  - Visual Studio
  - Visual Studio Code
  - and more…

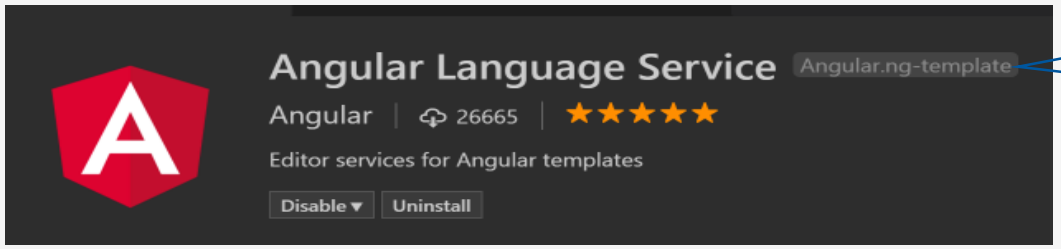
Set Up the environment

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

Set Up the Environment





Tip:
Recommended Extension

Install Git Client

git --everything-is-local

Search entire site...

- 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 →

Logos

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

View Logos →

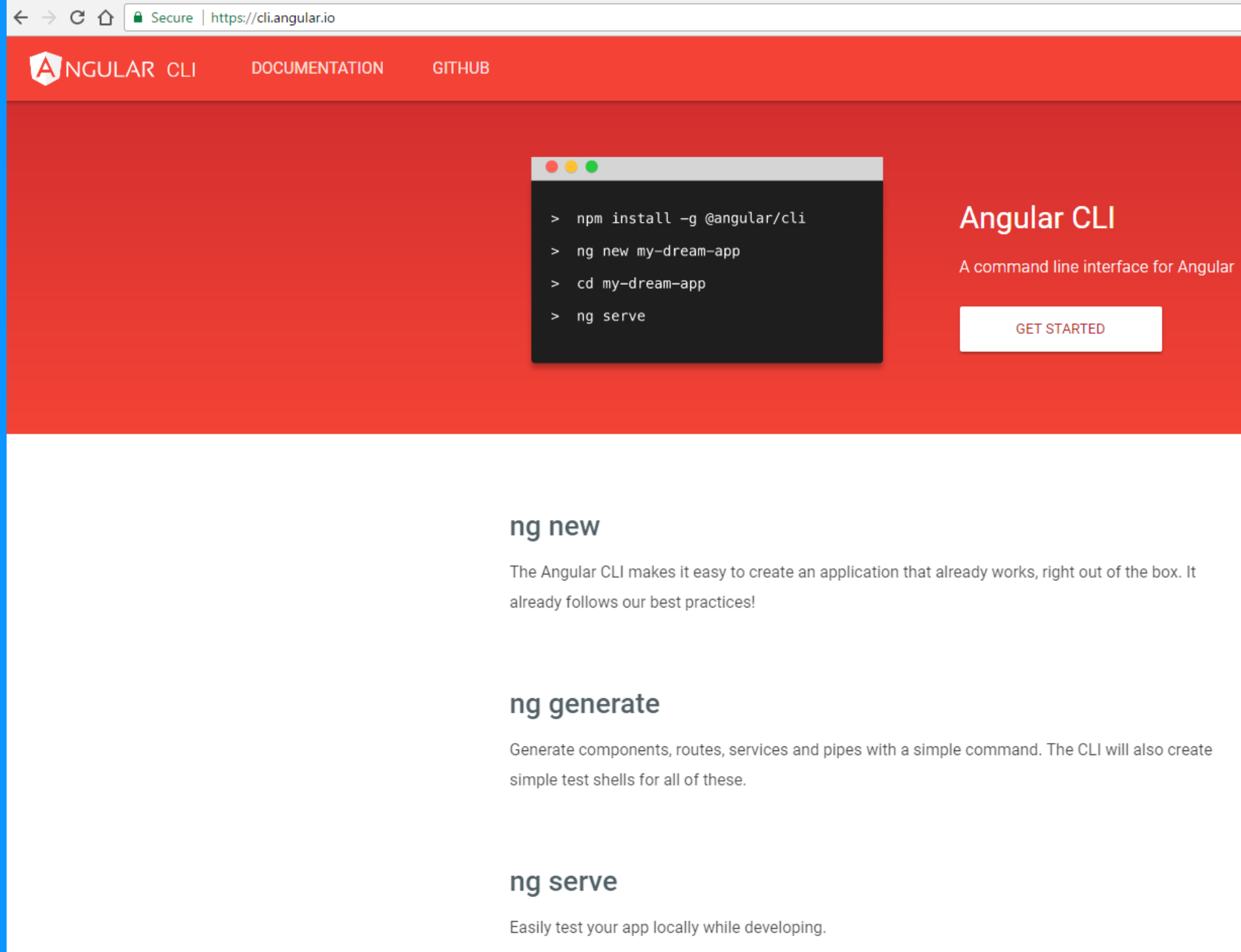
Git via Git

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.

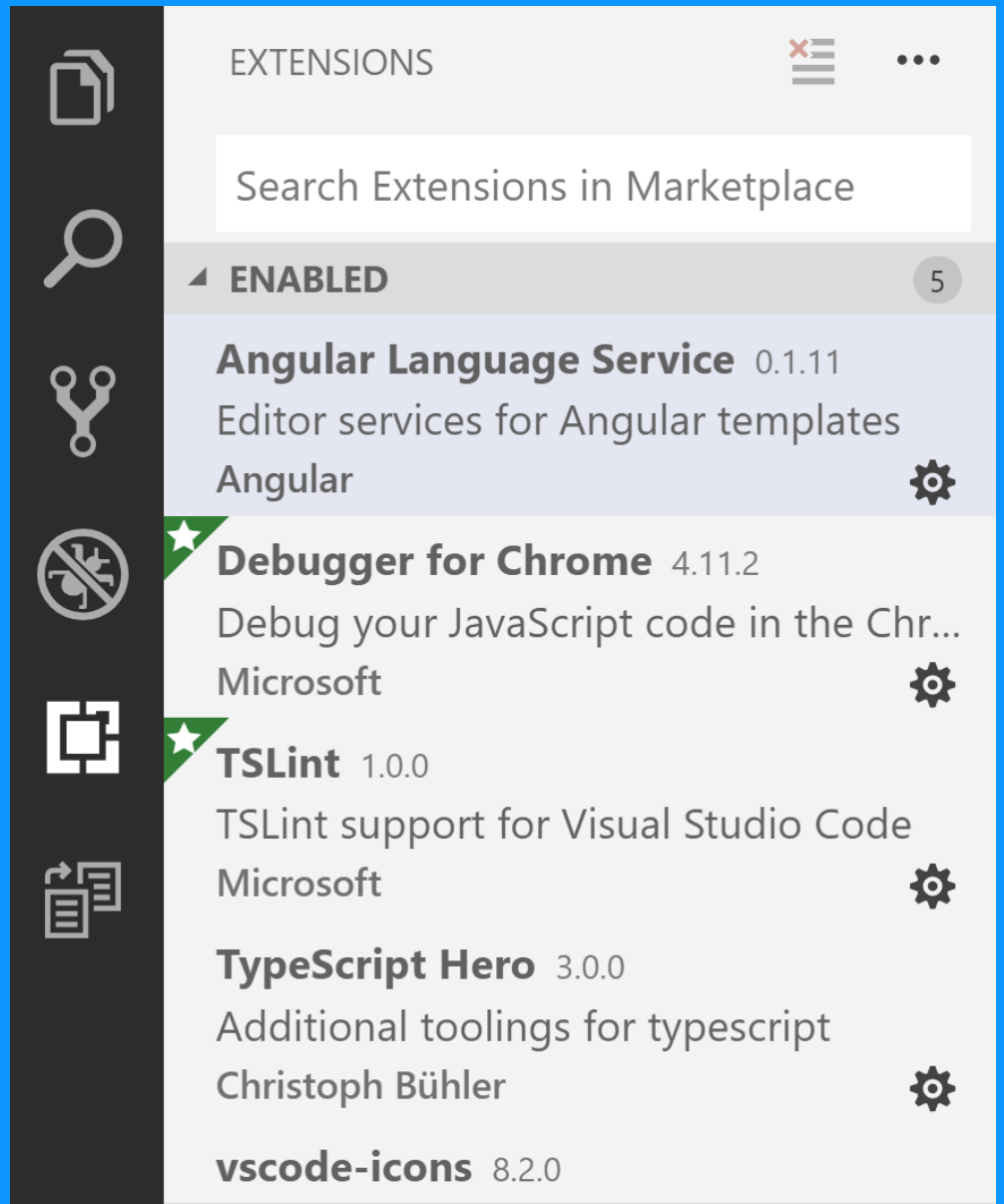
Set Up the Environment



Lab 1

Set Up the Project

VS Code Must Have Extensions



Run Angular CLI Repos Directly In Your Browser

StackBlitz

Angular Modules

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

ES 2015 Modules

book.ts

```
export class Book {
}
```

library.ts

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

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

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
...

@NgModule({
  declarations: [AppComponent, MyComboboxComponent,
                 CollapsibleDirective, CustomCurrencyPipe],
  imports: [BrowserModule],
  providers: [UserService, LessonsService]
})
export class ExampleModule {

}
```

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, what does Angular exactly do with this information?

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, 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

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

Angular Modules

```
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 })
14 export class AppModule {
15
16 }
17
```

The root module has the conventional name of AppModule

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.

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

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 })
14 export class AppModule {
15
16 }
17
```

Making Angular Modules More Readable

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
...

@NgModule({
  declarations: [AppComponent, MyComboboxComponent,
                 CollapsibleDirective, CustomCurrencyPipe],
  imports: [BrowserModule],
  providers: [UserService, LessonsService],
  bootstrap: [AppComponent]

})
export class AppModule {

}
```

Before
Including the full list of components within the app.module.ts

```
export const myComponents = [
  MyComboboxComponent,
  MyDropdownComponent
];

export const myDirectives = [
  CollapsibleDirective,
  SearchOnTypeDirective
];

export const myPipes = [
  CustomCurrencyPipe,
  CustomDatePipe
];
```

```
@NgModule({
  declarations: [
      ...myComponents,
      ...myDirectives,
      ...myPipes
  ],
  ...
})
export class AppModule {


}
```

After
Using Spread operator

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

- This is the current version of the HomeModule, and there is actually something wrong in this definition:

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

}

```

Feature Modules

- To see what the problem here is, let's try to use a core Angular directive in the Home component template, like for example ngStyle
- 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'.

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

}
```

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

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

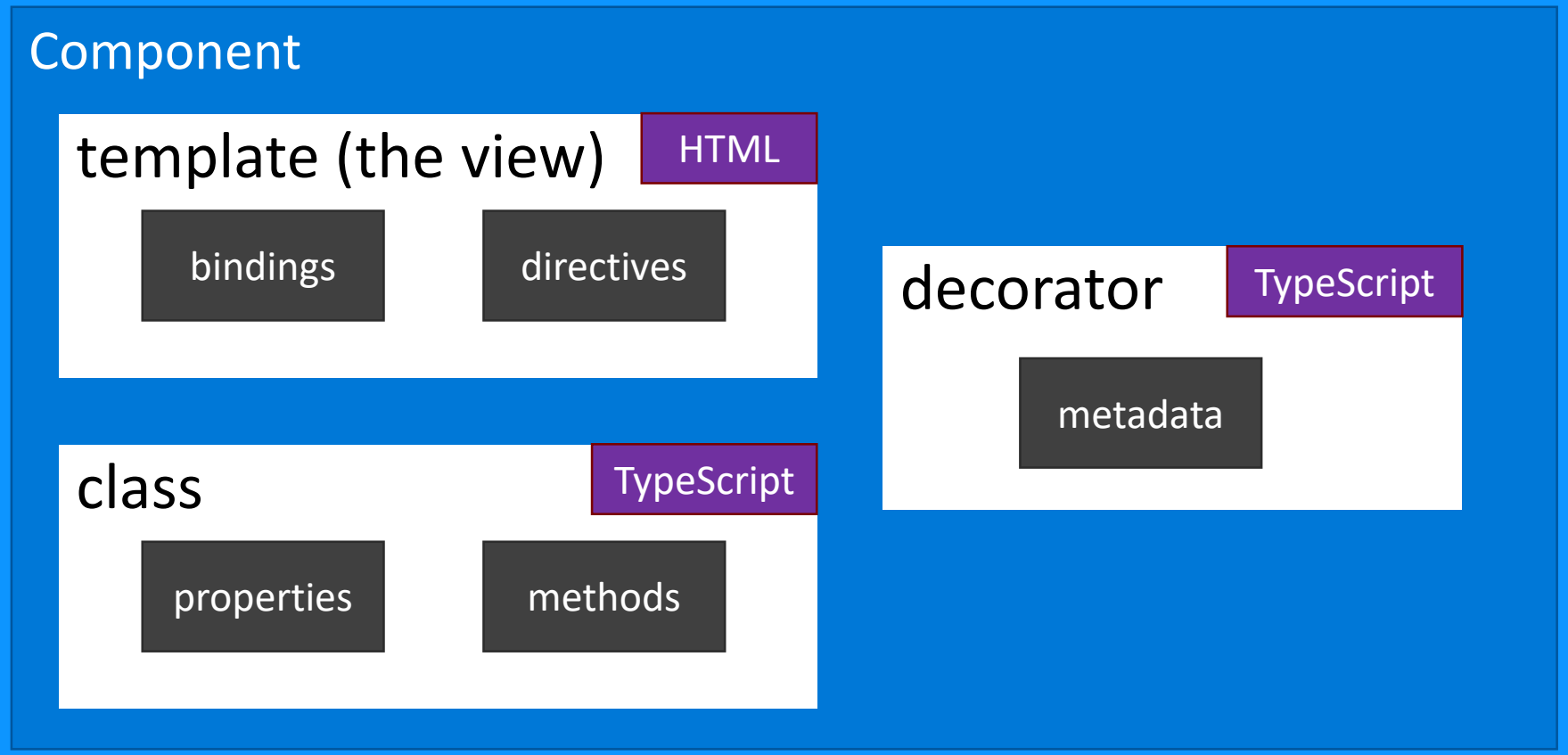
```
@NgModule({
    declarations: [Home],
    exports: [Home],
    providers: [LessonsService]
})
export class HomeModule {

}

```

```
import {CommonModule} from "@angular/common";

@NgModule({
    imports: [CommonModule],
    declarations: [Home],
    exports: [Home],
    providers: [LessonsService]
})
export class HomeModule {
    ...
}

```

Components

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

Components


Component
template (the view)
HTML
bindings
directives
class
TypeScript
properties
methods
decorator
TypeScript
metadata

Components

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

Directive name

Optional stylesheets

Include on the page using a directive

```
<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 then one module is also weird as you are saying that depending which module you chose 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 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="w
  <link rel="icon" type="image/x-i
  <script type="text/javascript" s
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

Main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platf

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

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

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

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 { }
```

App.component.html

```
<div class="container">
  <md-toolbar color="warn">
    <span>{{title}}</span>
    <span class="fill-remaining-space"></span>
    <img src="assets/img/books-icon.png"
         class="center-block"
         style="max-height:100px" />
  </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';
}
```

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

Interpolation

```
{{2 + 2}}
<h1>{{pageTitle}}</h1>
{{amount > 0 ? 'Positive' : 'Negative'}}
{{calculateResult()}}
<span class="{{getClass()}}">
    Hello
</span>
```

Expression

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

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 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

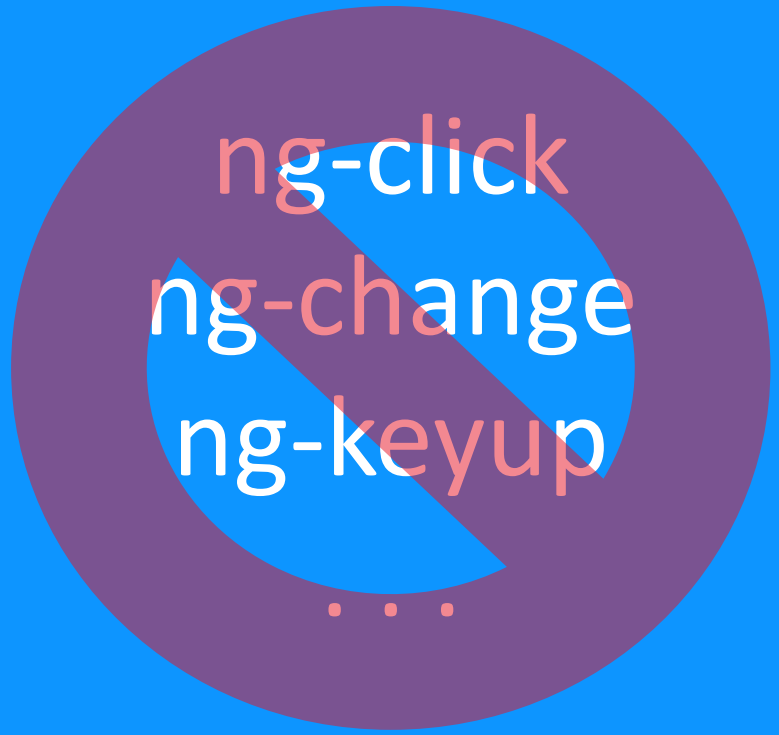
Interpolation and Property Binding Security

- *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

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

- In the example below Angular will look for a component that has a selector called ‘pm-products’

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 { }
```

Component Directive

- But we can have many components in our application that Angular needs to parse

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 { }
```

Component Directive

- Angular looks inside the Angular module that owns this component
  - Each component can belong to one and only one Angular module

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 { }
```

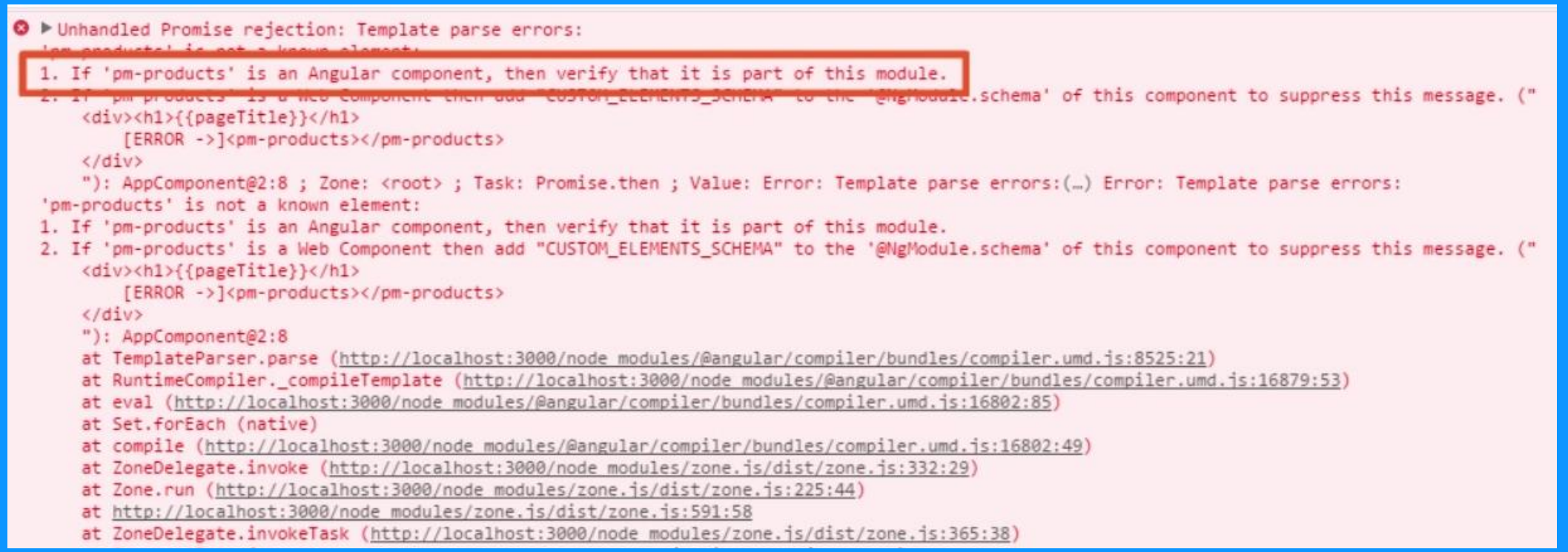
Component Directive

- App.module.ts will have to include the ProductListComponent as part of the declarations array

```
1   import { NgModule } from '@angular/core';
2   import { BrowserModule } from '@angular/platform-browser';
3
4   import { AppComponent }  from './app.component';
5   import { ProductListComponent }  from './products/product-list.component';
6
7   @NgModule({
8     imports: [ BrowserModule ],
9     declarations: [
10      AppComponent,
11      ProductListComponent ],
12    bootstrap: [ AppComponent ]
13  })
14  export class AppModule { }
15
```

Component Directive

- If you forget to include the ProductListComponent in the declarations array the following error will be thrown



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 do more with our components?

- Use strong typing and interfaces
- Build “Smart” and “Dumb” Components
- Encapsulate styles
- Use Lifecycle Hooks

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

Strong Typing and Interfaces

- Interfaces: TypeScript's Swiss Army Knife

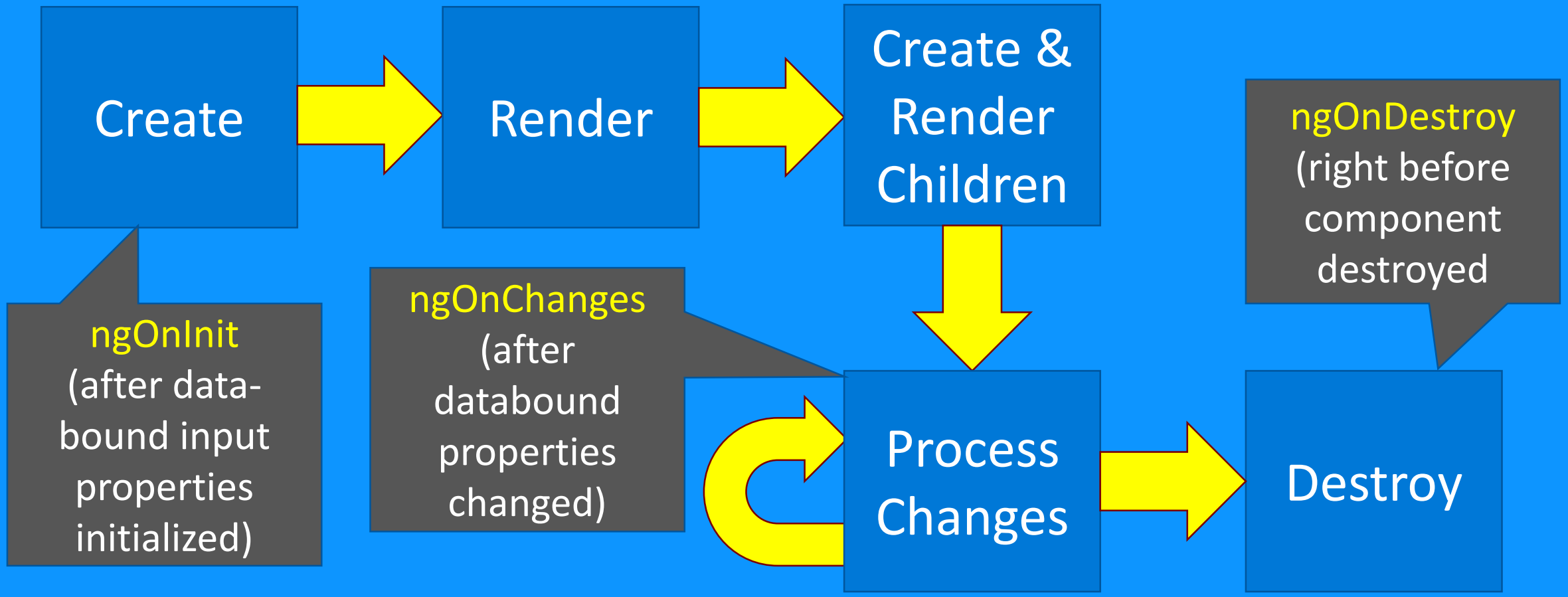
Encapsulating Styles

```
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";
}
```

Lifecycle Hooks

- A component lifecycle is managed by Angular

Component Lifecycle



Lifecycle Hooks

```
import { Component, OnInit } from 'angular2/core';
@Component({ ... })
export class AboutComponent implements OnInit {
  private pageTitle = "About";
  ngOnInit() {
    // do some initialization
    // This is a best practice to do initialization in ngOnInit
    // instead of the constructor
    // Components are easier to test and debug when their
    // constructors are simple and all real work (especially
    // calling a remote server) is handled in a separate method

  }
}
```

Using a Component

- There are two ways to use a component
  - As a directive: App Component or Nested Component

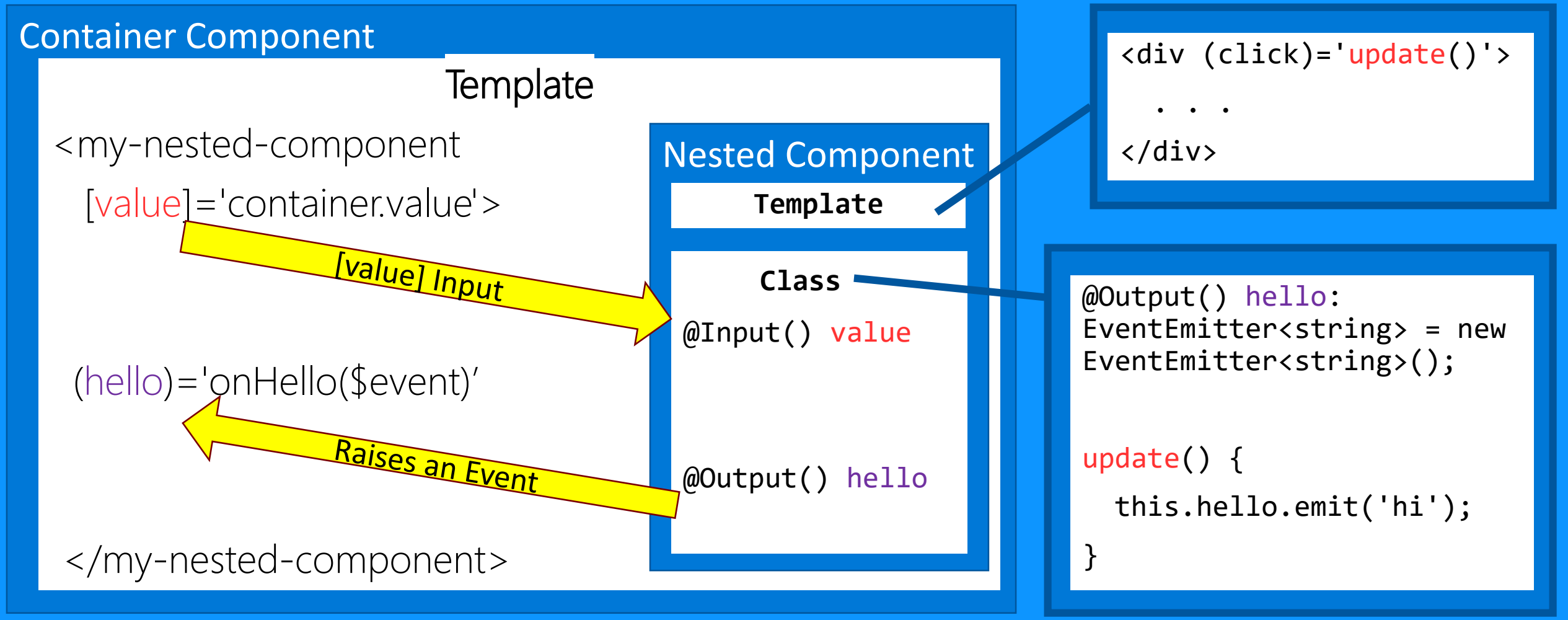
```
<body>
    <pm-app>Loading App...</pm-app>
</body>
```

  - As a routing target: This makes it appear to the user that they have navigated to another view

Building Nested Components

- What makes a component nest-able?
  - 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



Communication between Parent and Child Components

Child Component

```typescript
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!`);
    }
}
```

```html
<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

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**

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

```
@NgModule({
  imports: [
    BrowserModule,
```

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

Custom Attribute directives

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

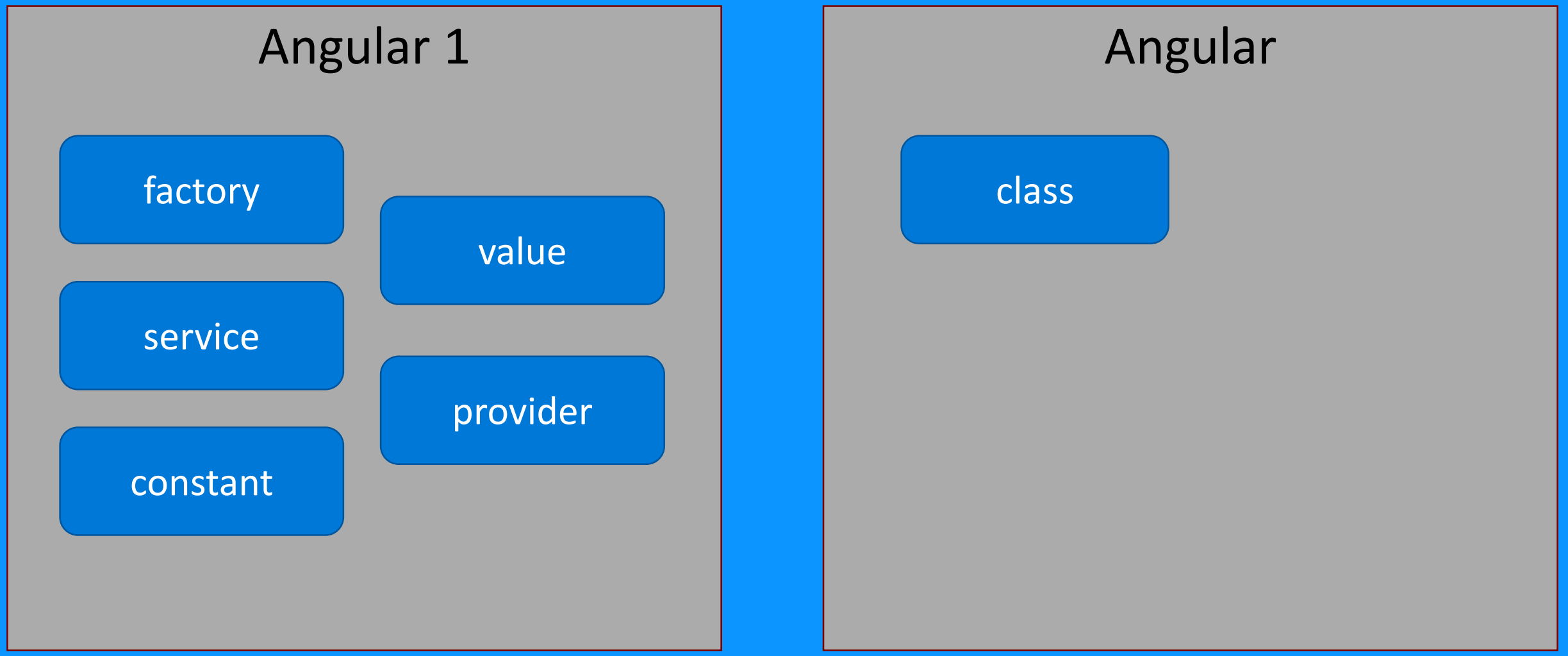
Demo

Custom Directives

Lab 5

Communication Between Parent and Child Components

Services


Angular 1
factory
service
constant
value
provider
Angular
class

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

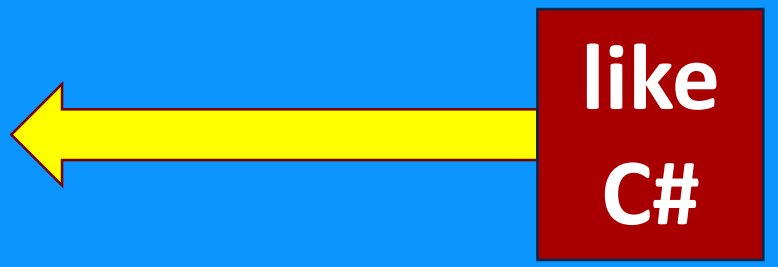
```
export class MyDataService {}
```

Component

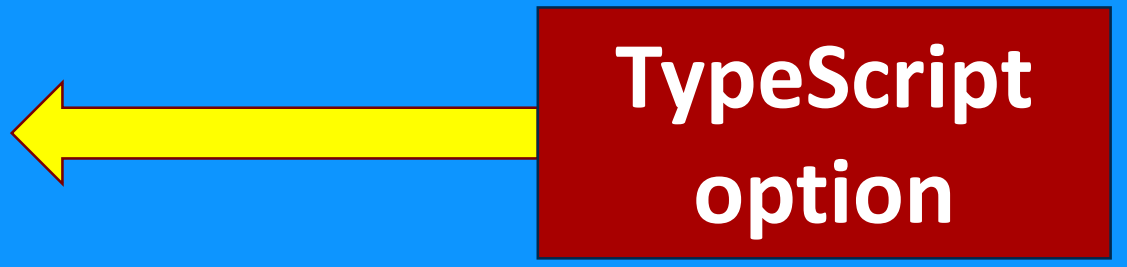
```
let svc = new MyDataService ();
```

TypeScript Constructor Syntax

```
class Person {
  firstName: string;
  lastName: string;
  constructor(firstName: string, lastName: string)
{
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

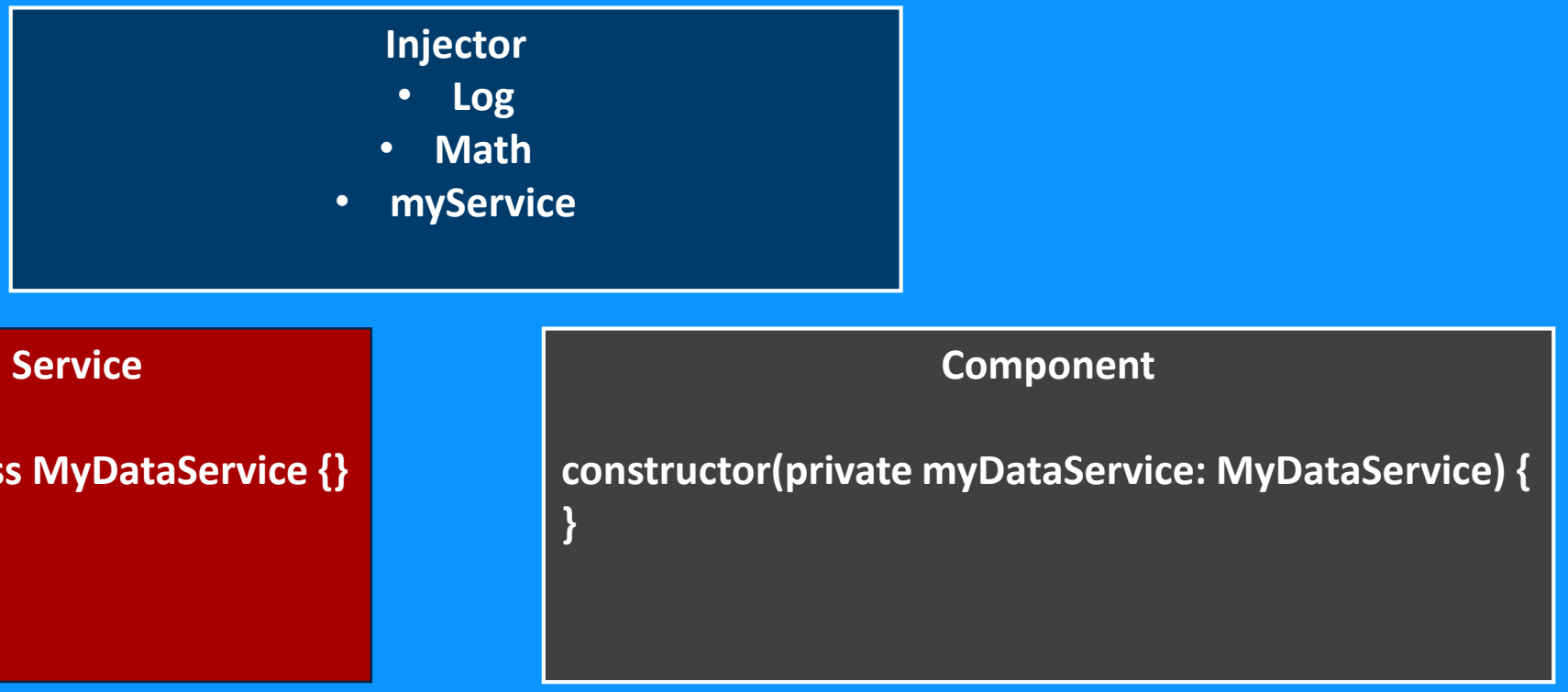



```
class Person {
    constructor(private firstName: string,
                private lastName: string) {
    }
}
```



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)



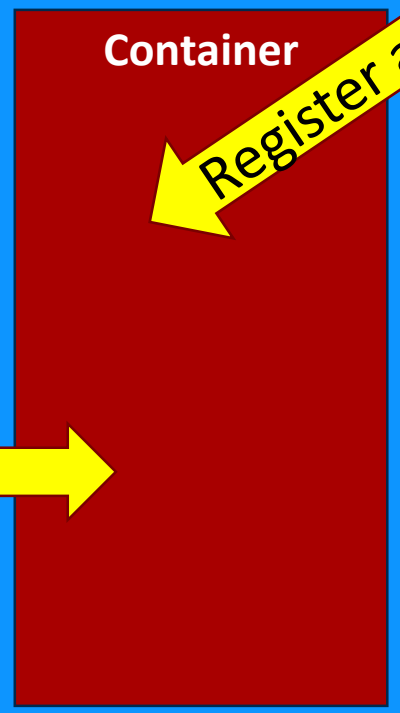
Services

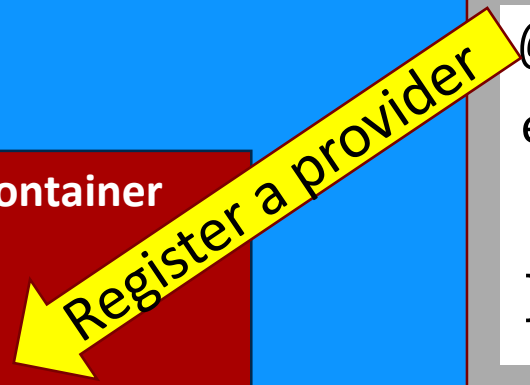
- If you register it twice (in different components) then you get 2 instances (no longer a singleton). The injector will keep track of both.

Services

Angular Injector

- http
- routeParams
- log
- myDataService
- myConstants





Define your service

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

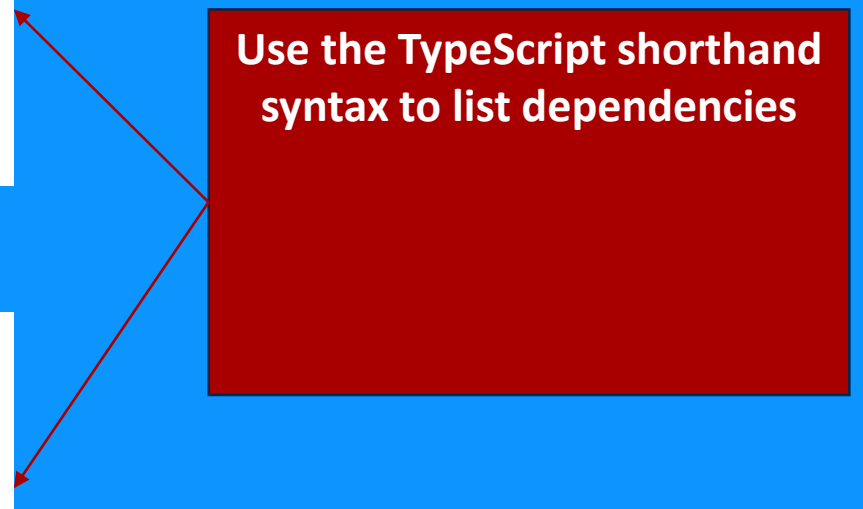
Use your service

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

Services

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

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



Service Provider Choices

- Part of the service's own metadata, (in the @Injectable() decorator) making that service available everywhere
- Register with specific modules (in the @NgModule() decorator)
- Register with specific components, (in the @Component() decorator)

Register a Provider

Root Injector

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

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

  constructor() { }
}
```

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 {}
```

Register a Provider

Inside a module

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


@NgModule({
    providers: [MyDataService]
})
export class MyFeatureModule {}
```

Demo

Services

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 a 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 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

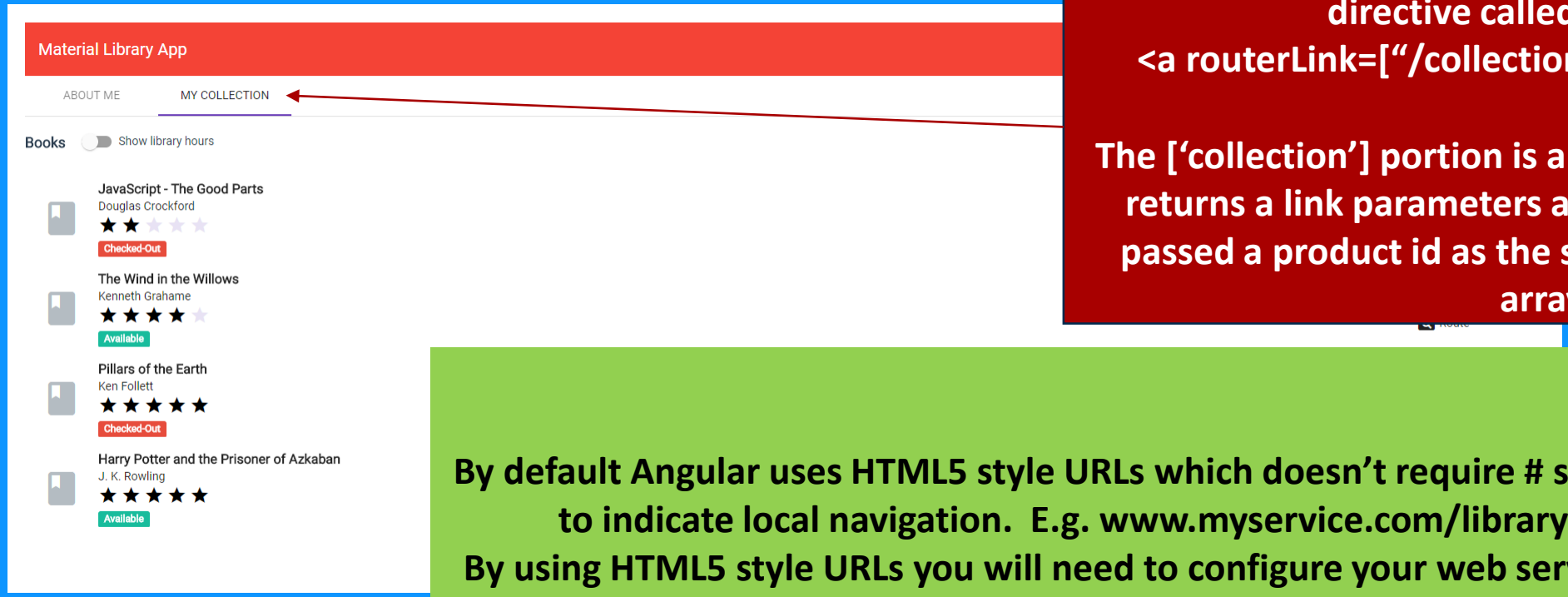
Lab 7

Http Service

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



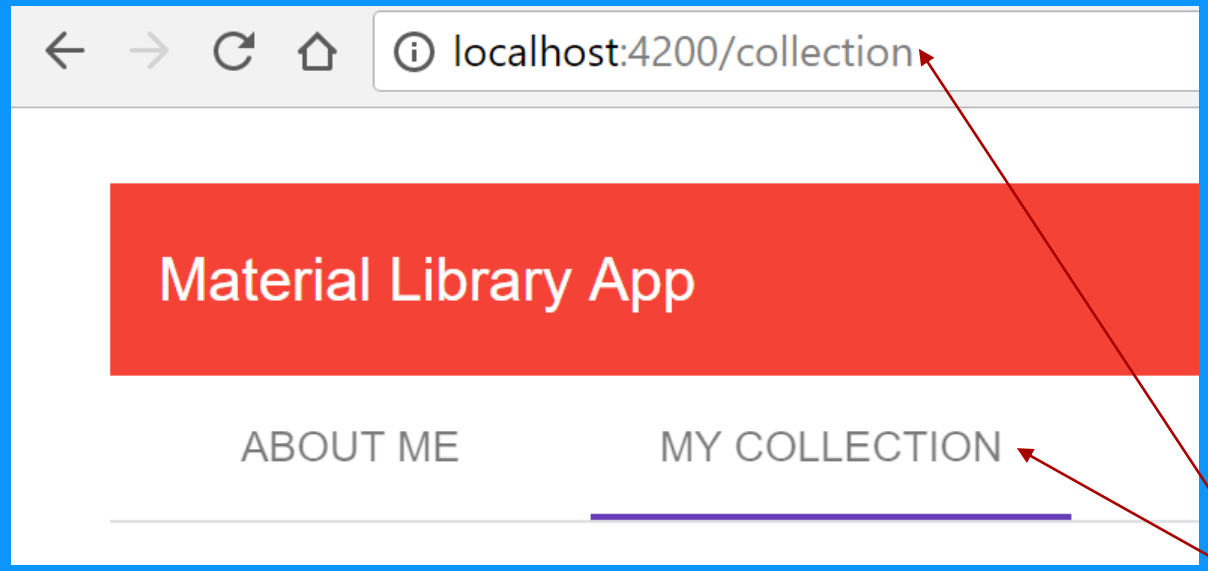
We tie a route to each menu option using a built in directive called routerLink
<a routerLink=["/collection"]> My Collection

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



We tie a route to each menu option using a built in directive called routerLink
**<a routerLink="/collection"> My Collection **

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 {
```

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

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,
    ProductModule
  ],
```

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-nav-bar>
  <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 8

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

Demo

Route Parameters

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 9

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

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, Router } from '@angular/router';

@Injectable()
export class BookDetailGuard implements CanActivate {

    constructor(private _router: Router) {
    }

    canActivate(route: ActivatedRouteSnapshot): boolean {
        let id = +route.url[1].path;
        if (isNaN(id) || id < 1) {
            alert('Invalid book Id');
            // start a new navigation to redirect to list page
            this._router.navigate(['/library']);
            // abort current navigation
            return false;
        };
        return true;
    }
}
```

```
@NgModule({
    imports: [BrowserModule, Forms
    declarations: [AppComponent, A
    bootstrap: [AppComponent],
    providers: [BookDetailGuard]
})
```

```
{
  path: 'collection/:id',
  canActivate: [BookDetailGuard],
  component: BookDetailComponent
},
```

Route Resolves

- Route resolves are nothing more than a way to pre-fetch the data a component needs before it is initialized

- For example if you have a component whose only role is to display a chart of daily sales for the month, then there is no point in rendering the view or loading this component before the sales data is available

Route Resolves

- In fact, many charting libraries will throw an error if you try to initialize a chart before supplying it with data (and so will *ngFor)

- You can work around this problem by hiding the html with an *ngIf or temporarily supplying an empty array until the necessary data is loaded

Route Resolves

- While you can get by without resolves, implementing them helps make code more readable and maintainable by:
  - Eliminating confusing clutter in your component's template and code.
  - Clarifying your intent by showing which data must be pre-fetched

Route Resolves

- Despite these benefits, many sites avoid using resolves in favor of displaying most of the component and showing a spinner in the sections for which data is still being loaded

Lab 10

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

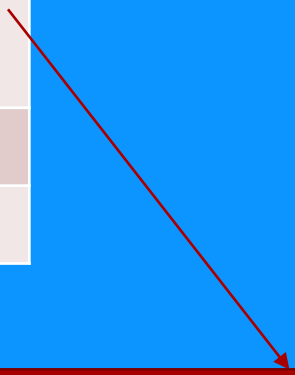
Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	

```
@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, HttpModule, 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 FormControls 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

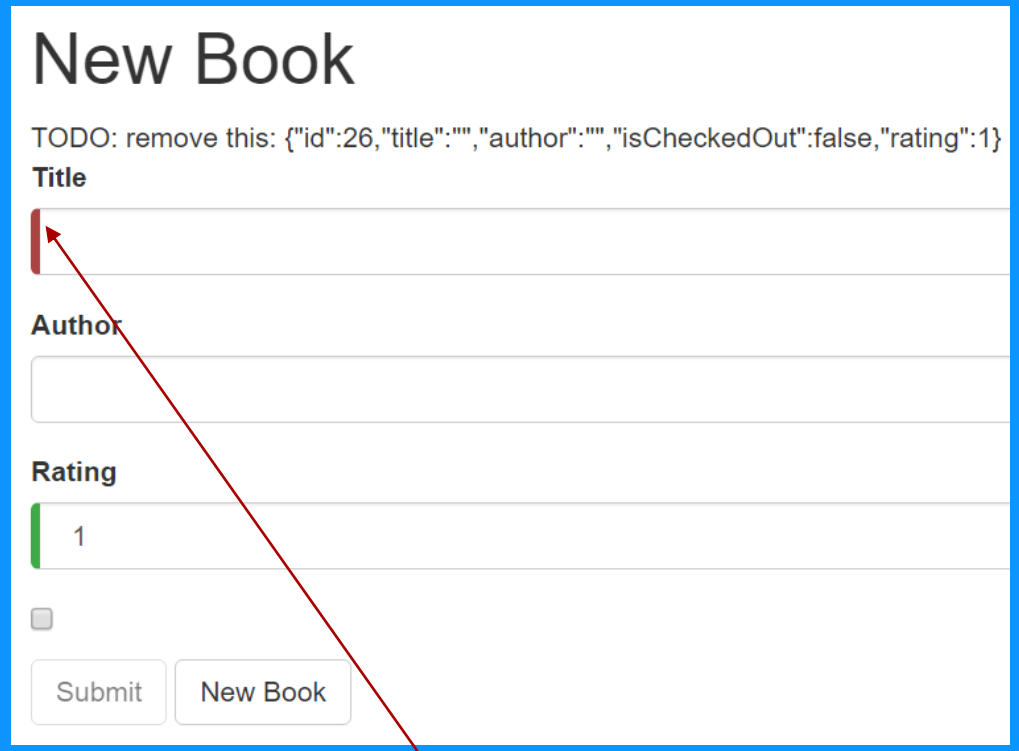
Track change-state and validity with ngModel

- 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

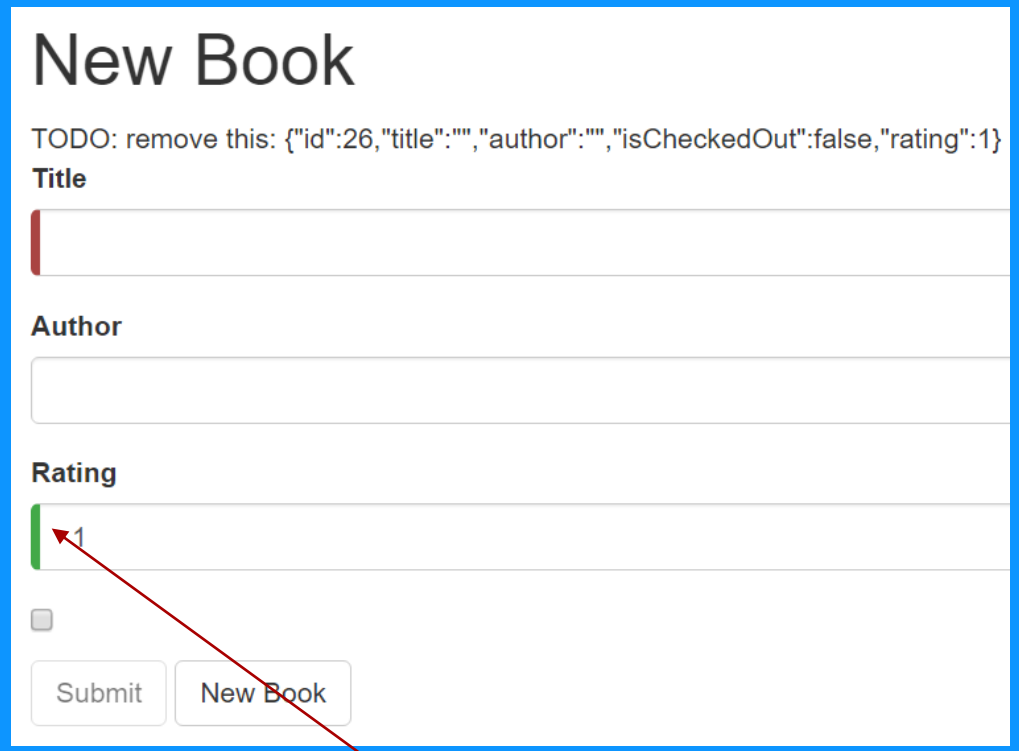
Track change-state and validity with ngModel

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

Add Custom CSS for Visual Feedback



Invalid State

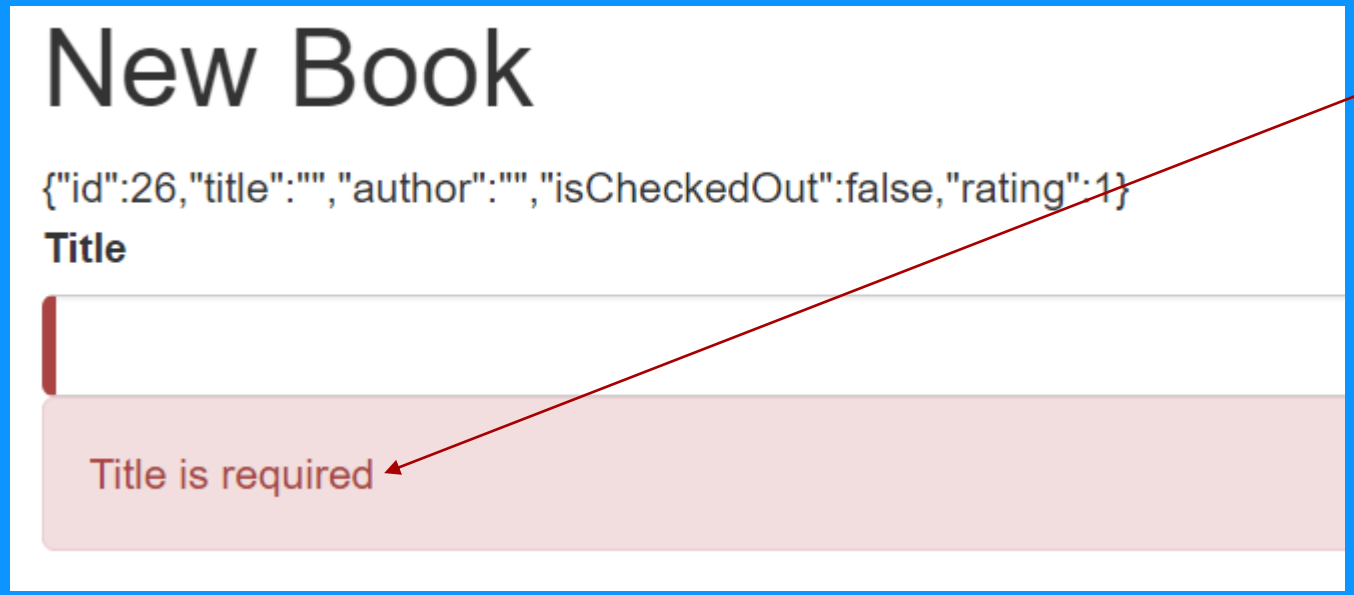


Valid State

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



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"

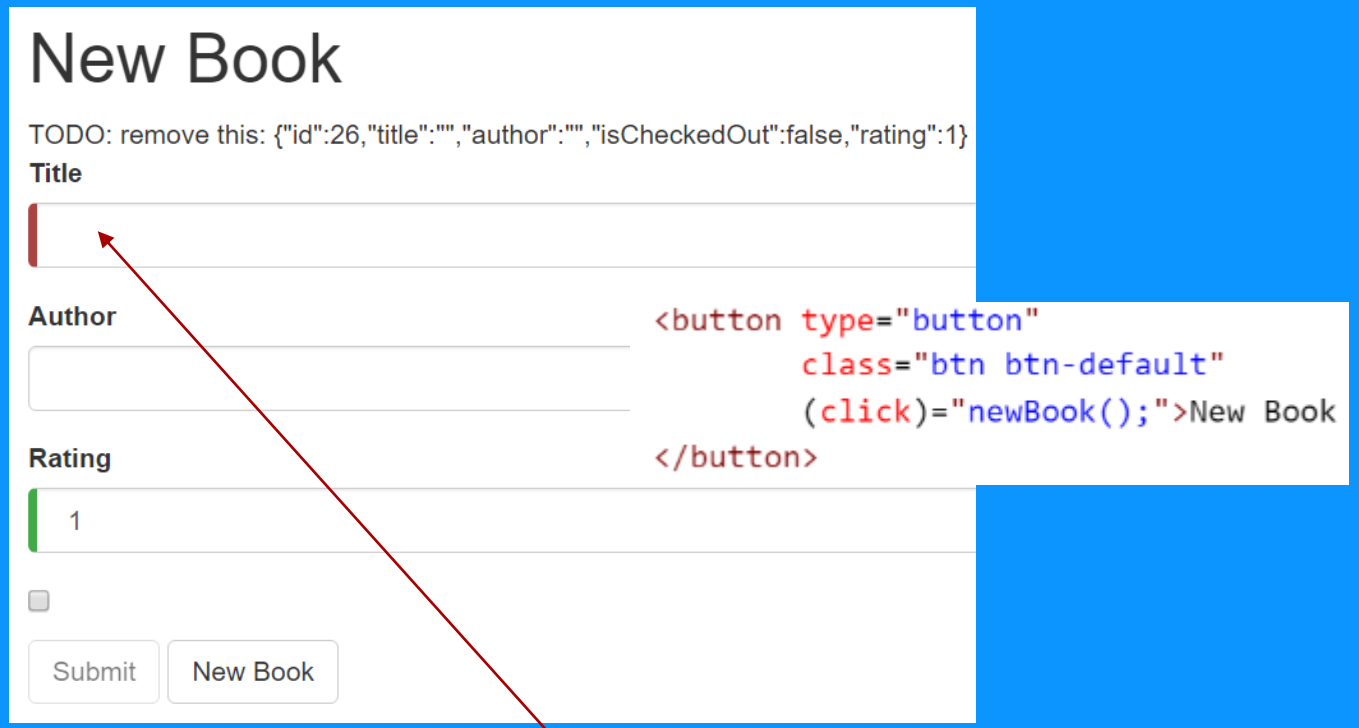
Show and Hide Validation Error messages

```
<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 element's hidden property

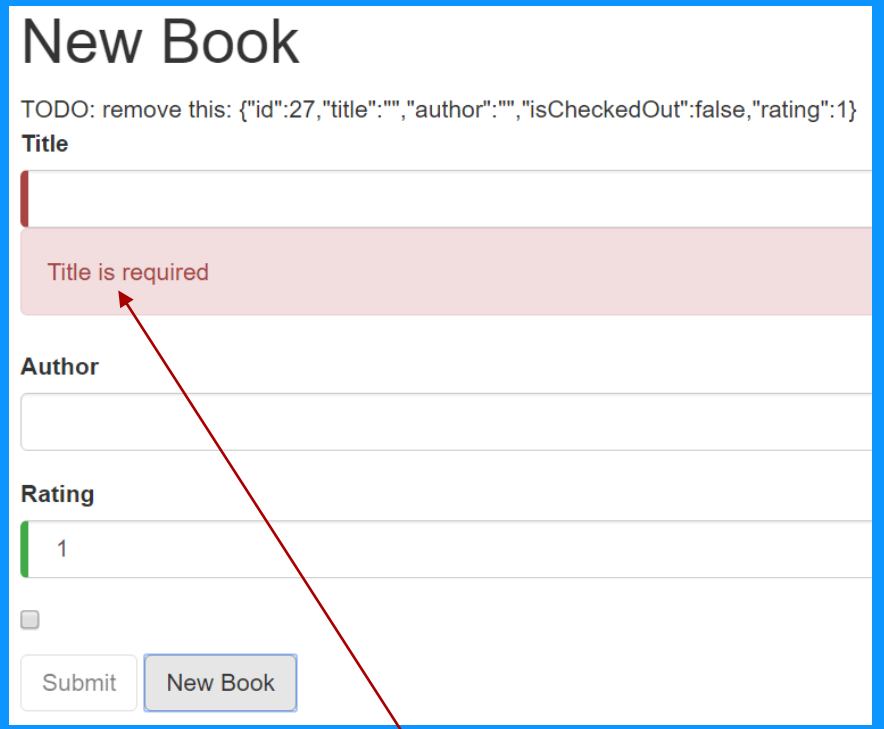
In this example, we hide the message when the control is valid or pristine; pristine means the user hasn't changed the value since it was displayed in this form

What About Resetting The Form?



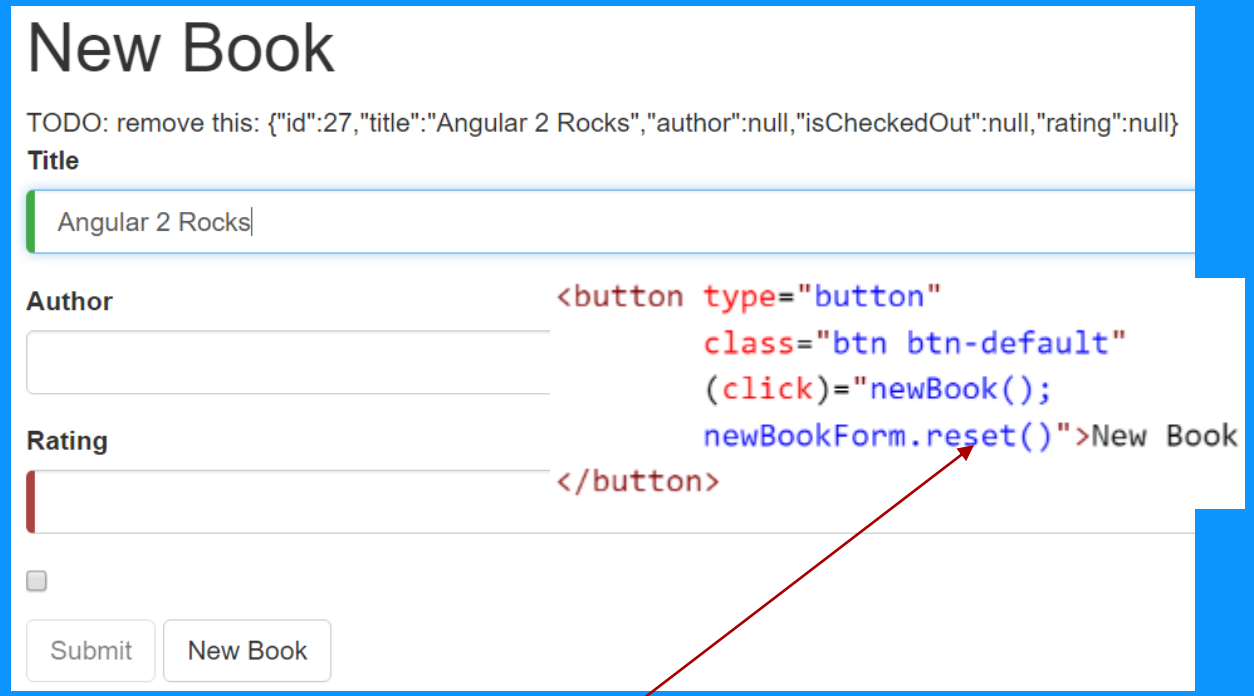

```
<button type="button"
        class="btn btn-default"
        (click)="newBook();">New Book
</button>
```

Data Entered



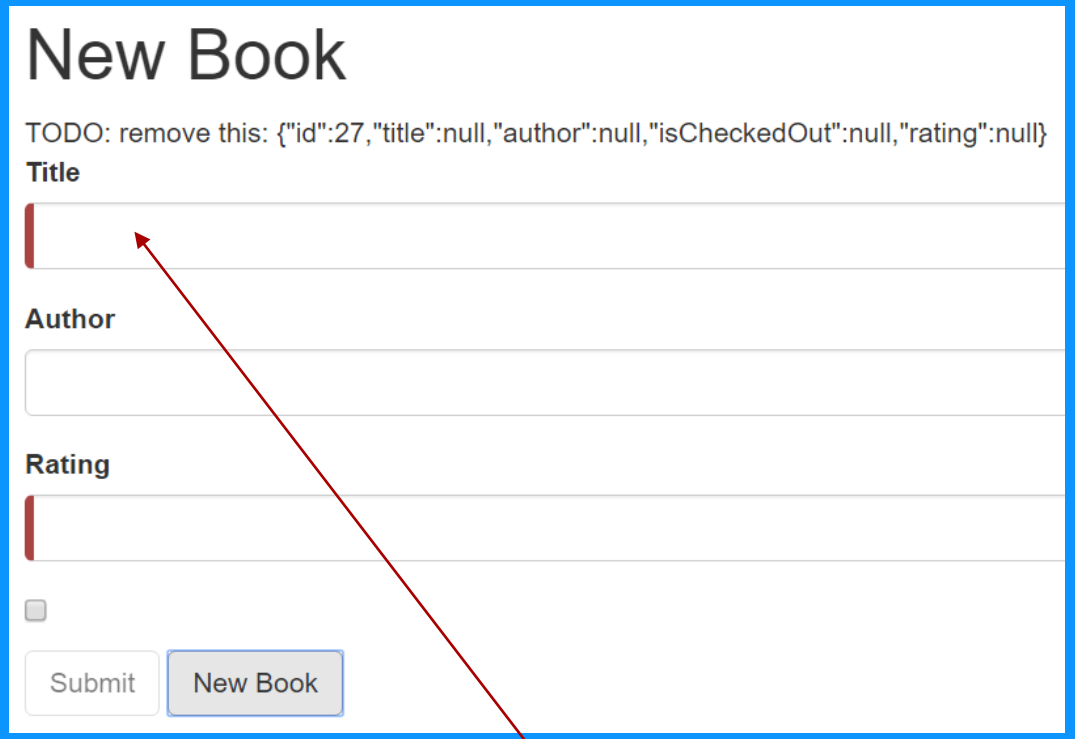
The form remembers that we entered data before clicking New Book. Replacing the book object did not restore the pristine state of the form controls

What About Resetting The Form?




```
<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



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 11

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

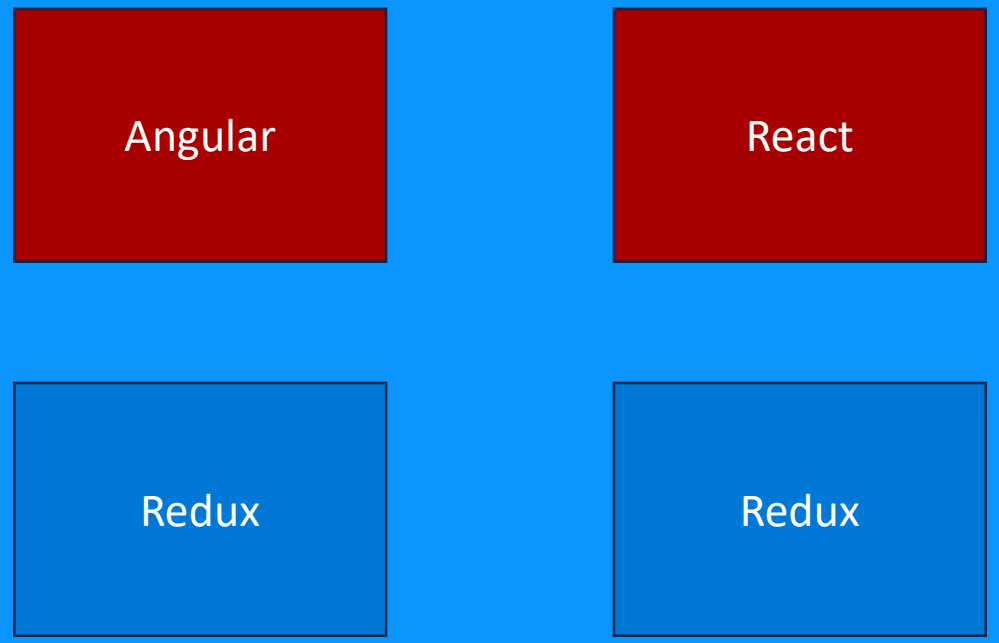
Lazy Loading

Lab 12

Lazy Loading

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



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

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

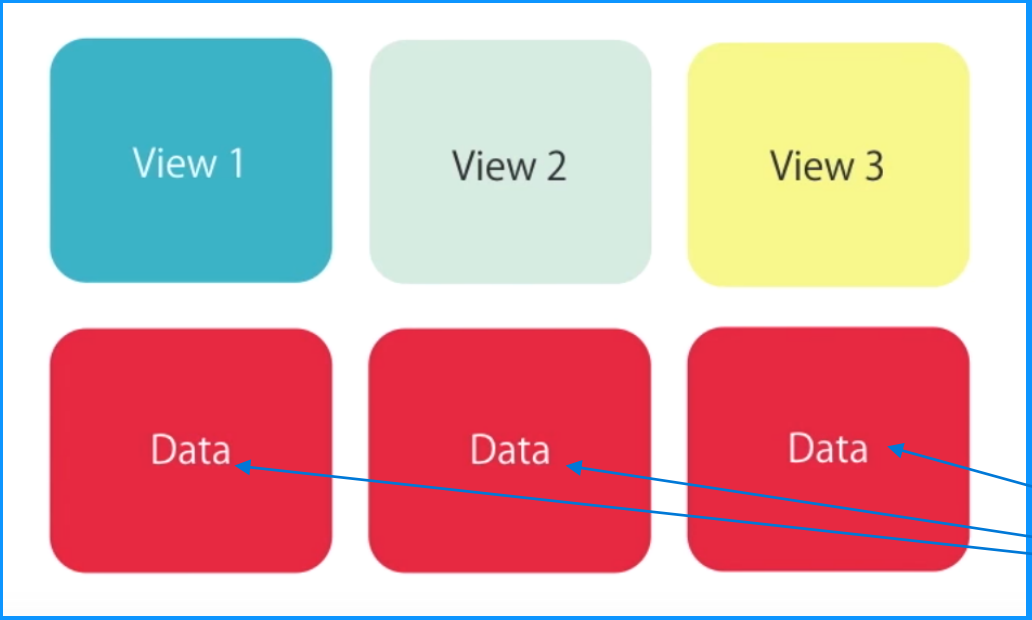
What is The Problem That Redux is Trying To Solve?

```
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?

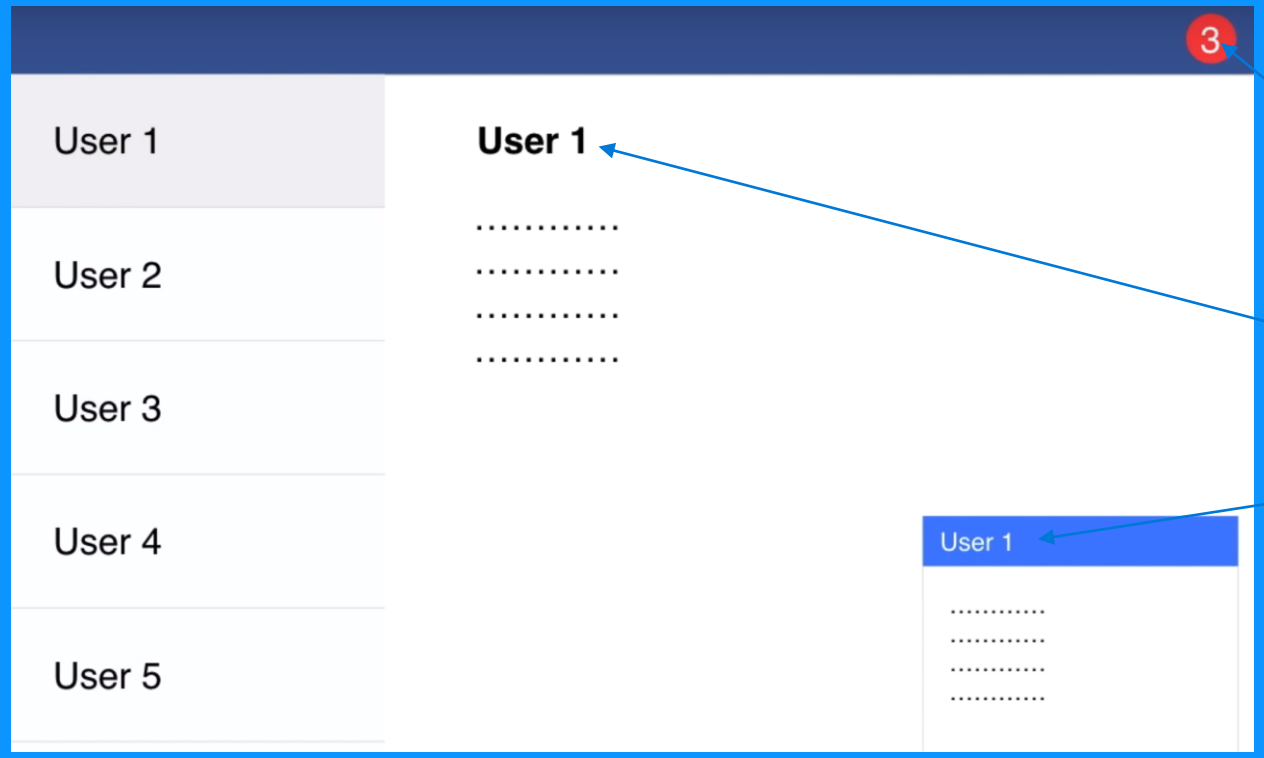


This could be a problem when you have multiple views that are working with the same piece of data and do not have the parent-child relationship

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

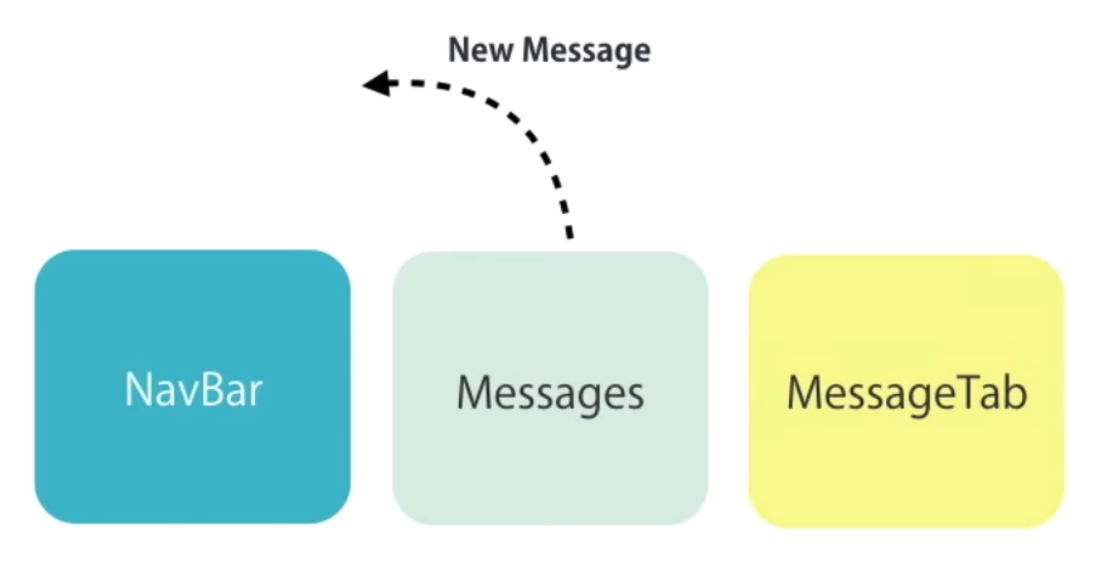
So when a view updates a model we need to do some extra work to keep the other views in sync

What is The Problem That Redux is Trying To Solve?



Three different views that don't have parent-child relationship

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

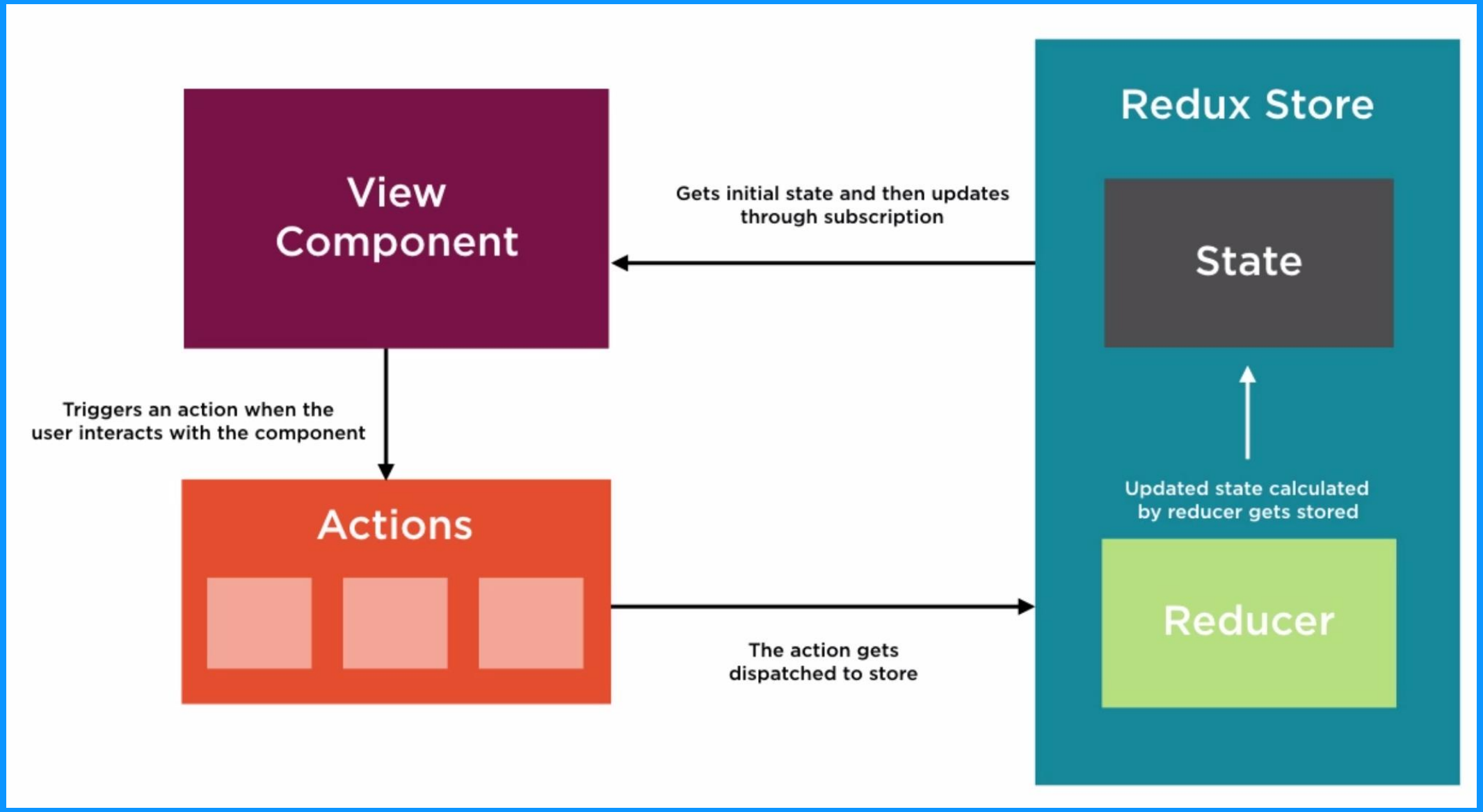
Unpredictable

What is The Problem That Redux Solves?

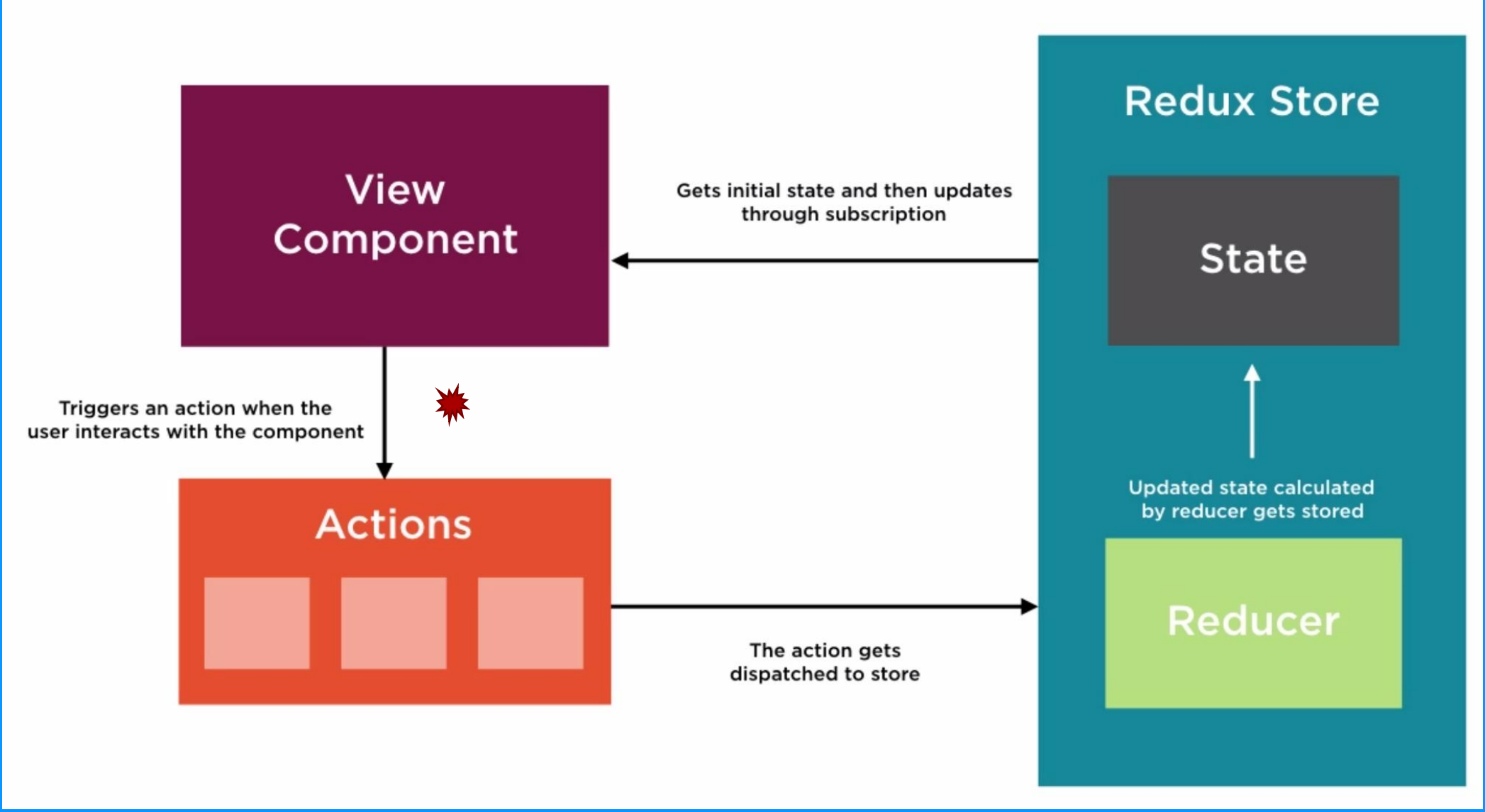
Facebook had this problem back in 2014 and as a result they introduced the Flux architecture

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

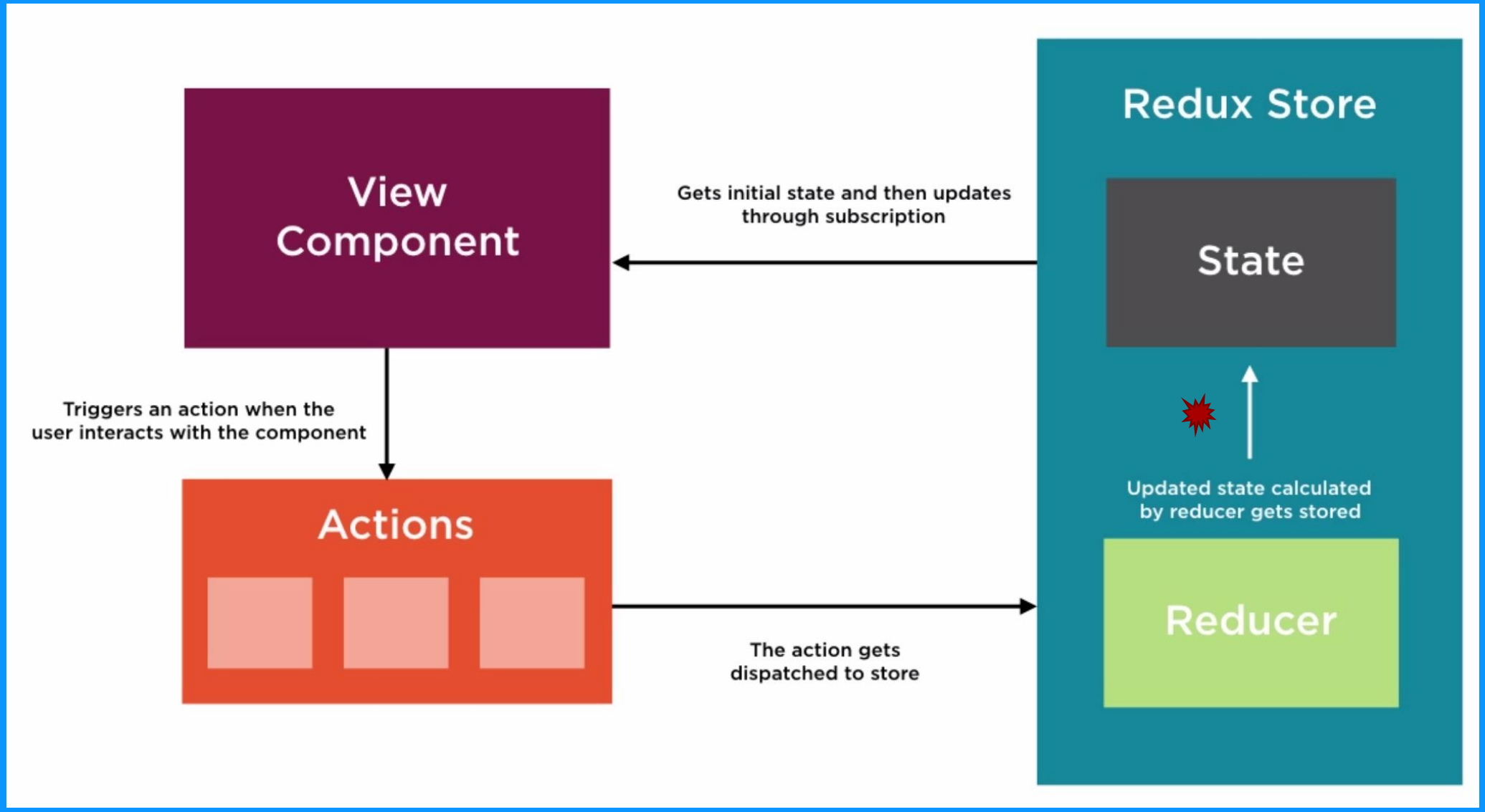
Redux Pattern



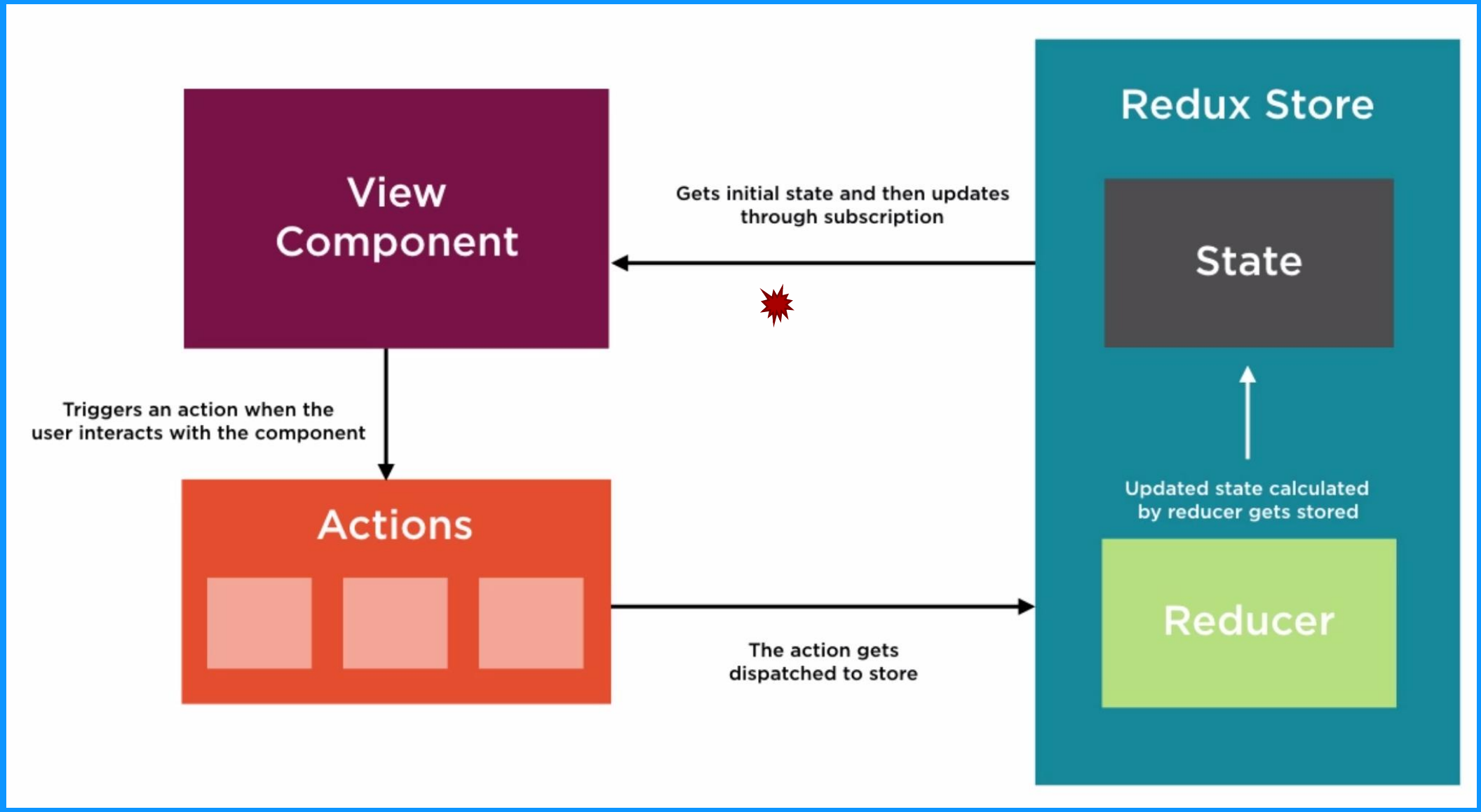
Redux Pattern



Redux Pattern



Redux Pattern



Redux Implementations for Angular

- ngrx/store
- ng2-redux

Demo

Ngrx-store

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
```

Testing

- We have three types of tests

Unit Tests

Integration Tests

End to End Tests

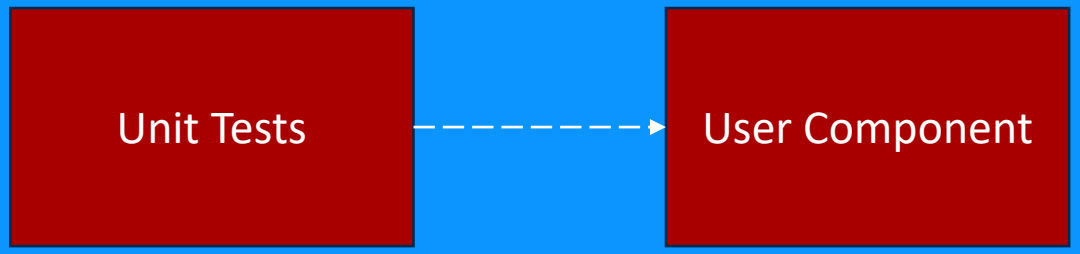
Unit Tests

- 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

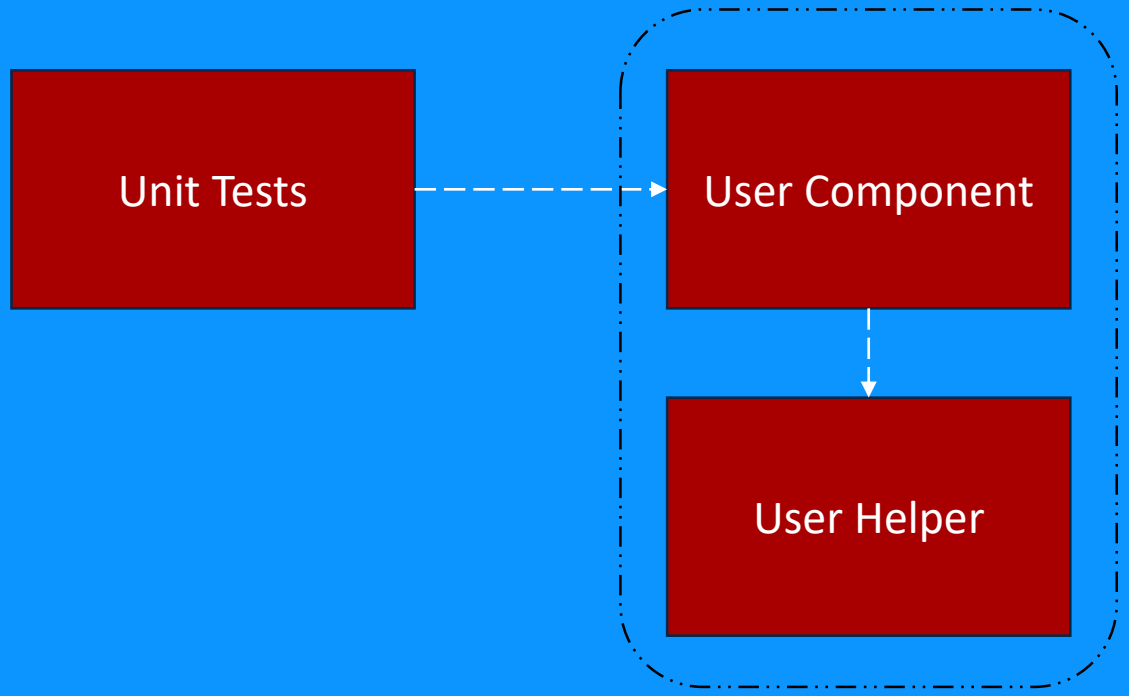
- In Angular world this means testing your Component in isolation without its Template and any other resources that touch external resources (e.g. file system, database, API endpoints)

Unit Tests



Unit Tests

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



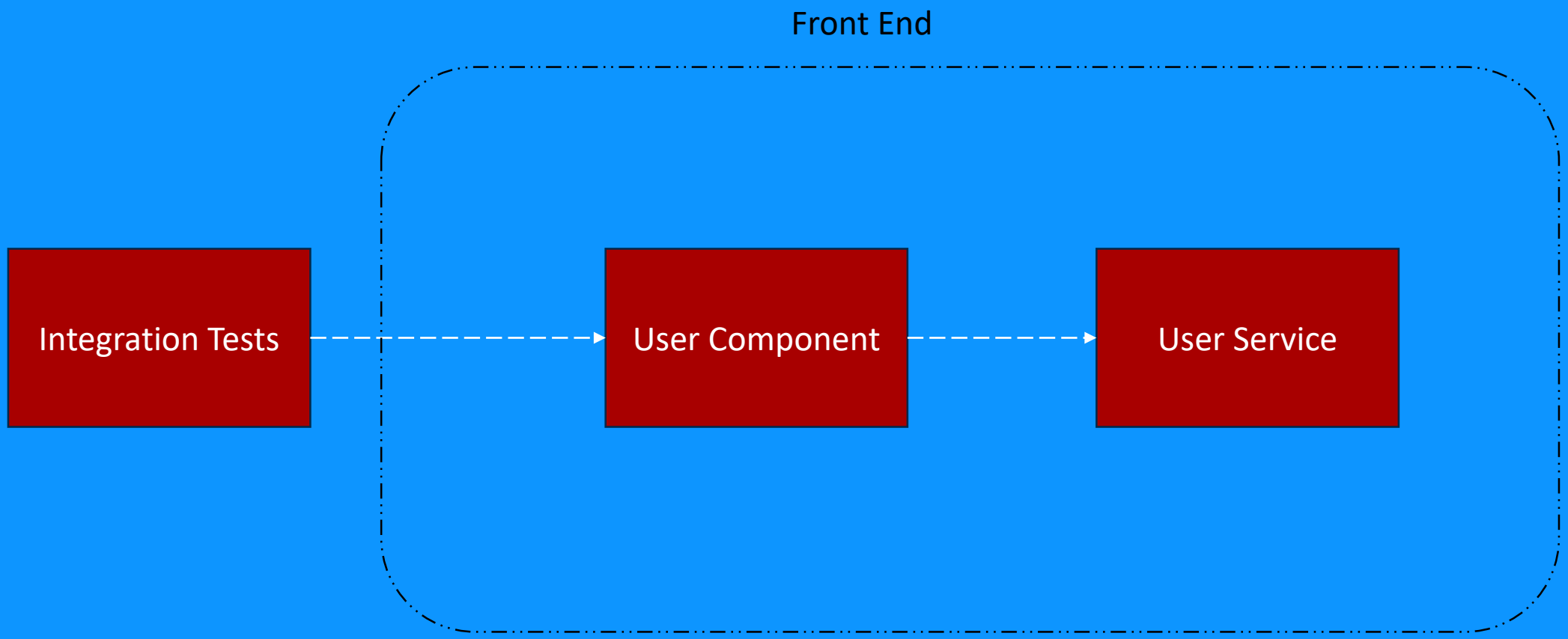
Unit Tests

- 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

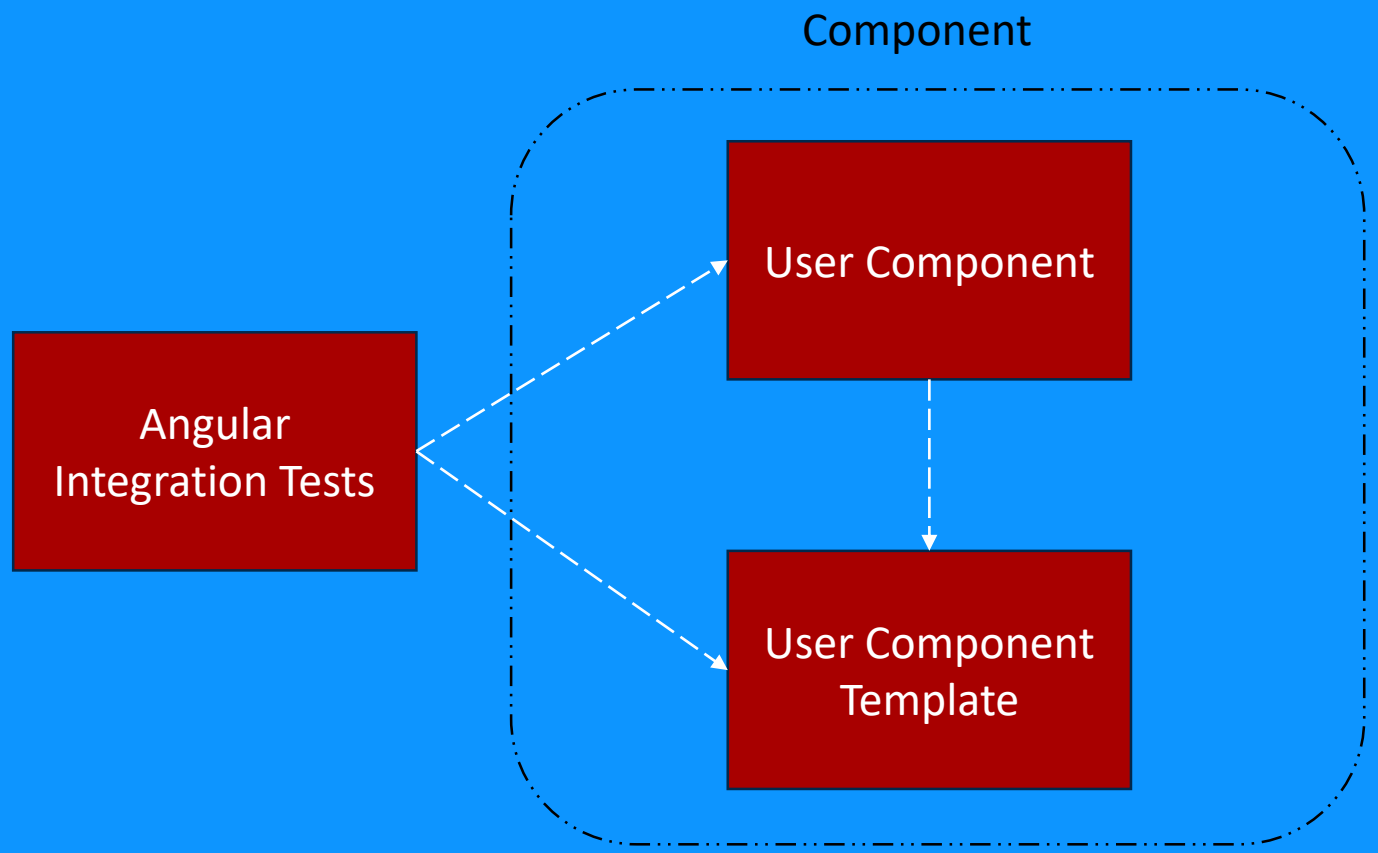


What About Testing A Component With Its Template?

- 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?

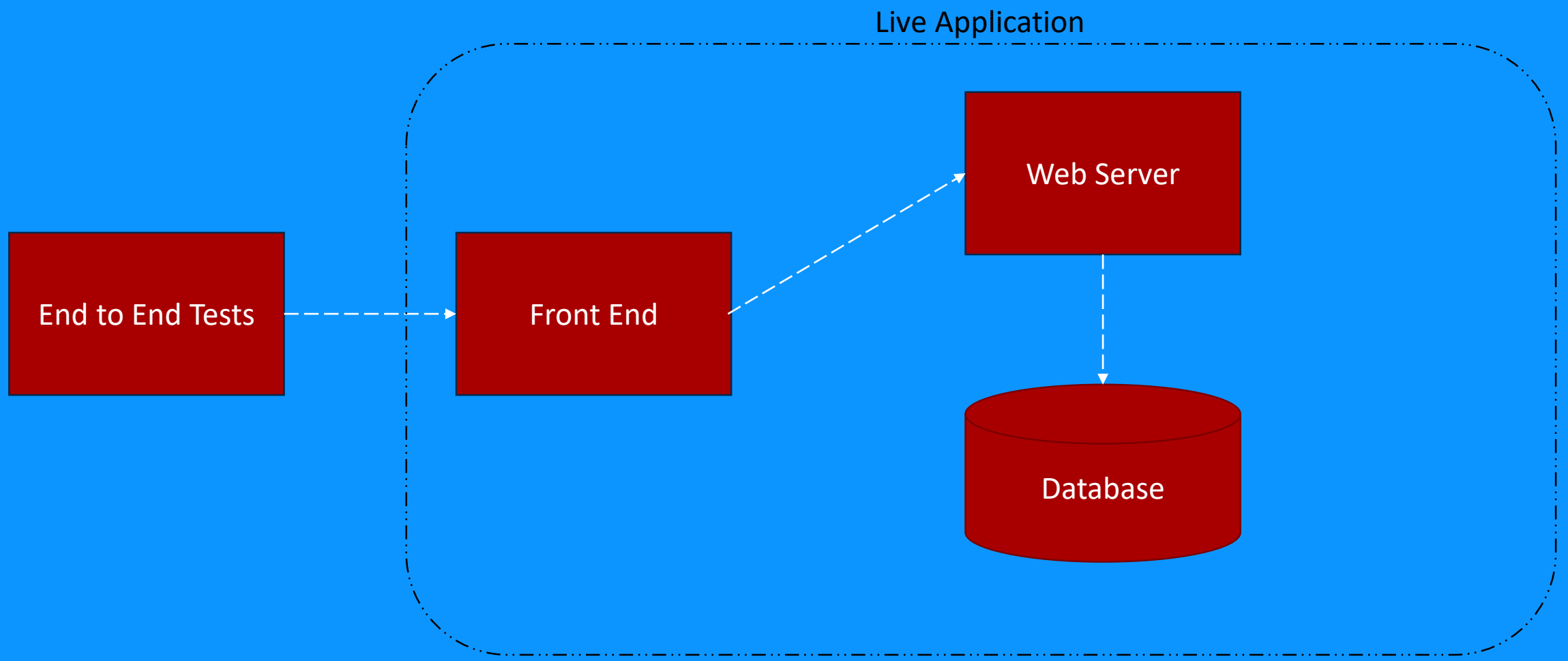
- Is this truly an integration test or is it just another kind of unit test?
- We have two different pieces which are the same unit of code
- That is up to you decide but the name of this type of testing is officially called integration testing



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

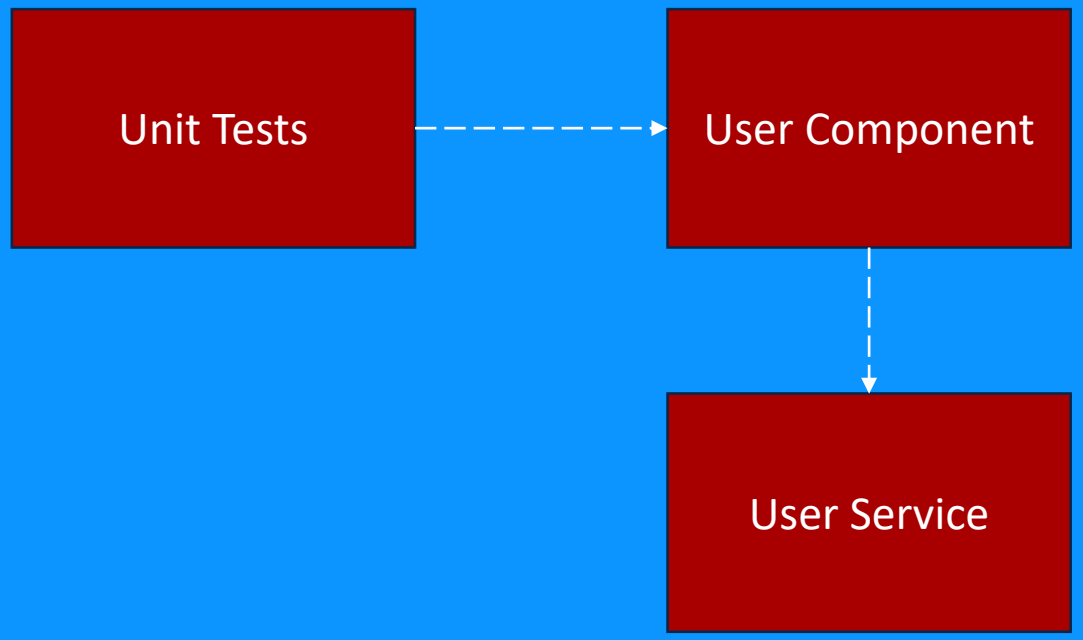
A lot of Unit/Integration Tests

Few
End to end Tests

Only for the key functions
of the application

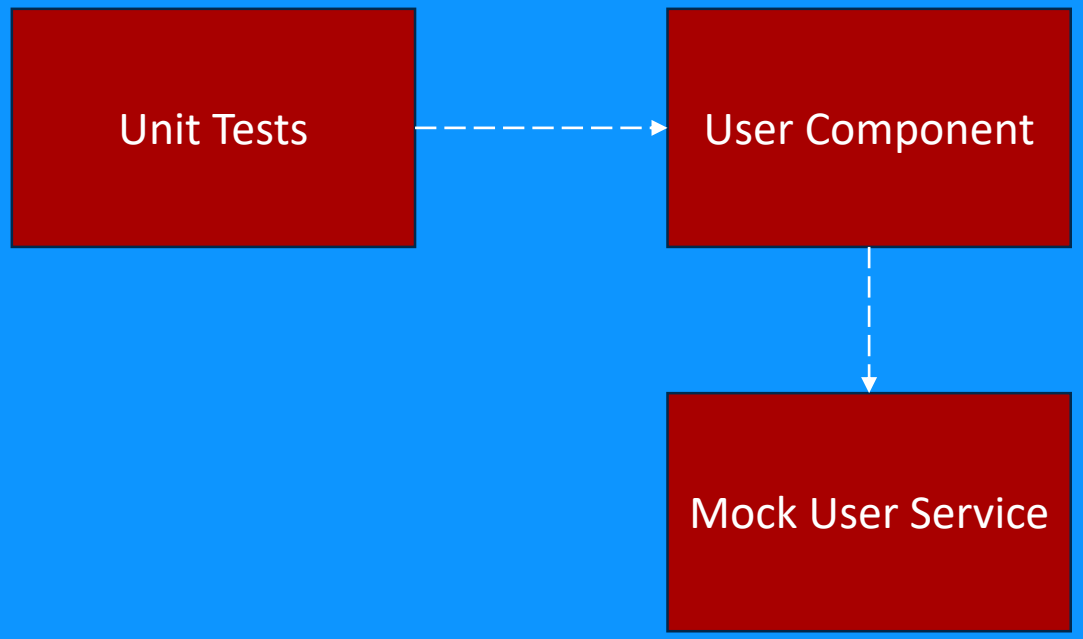
Mocking

- 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 where called and how many times where they 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 on 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 13

Testing

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

Lab 13

Route Resolves


Microsoft