Learn to Build Awesome[r] Apps with Angular

Feature

Features

Applications

Feature

Features

Applications

Feature

Features

Applications

http://bit.ly/awesomer-survey

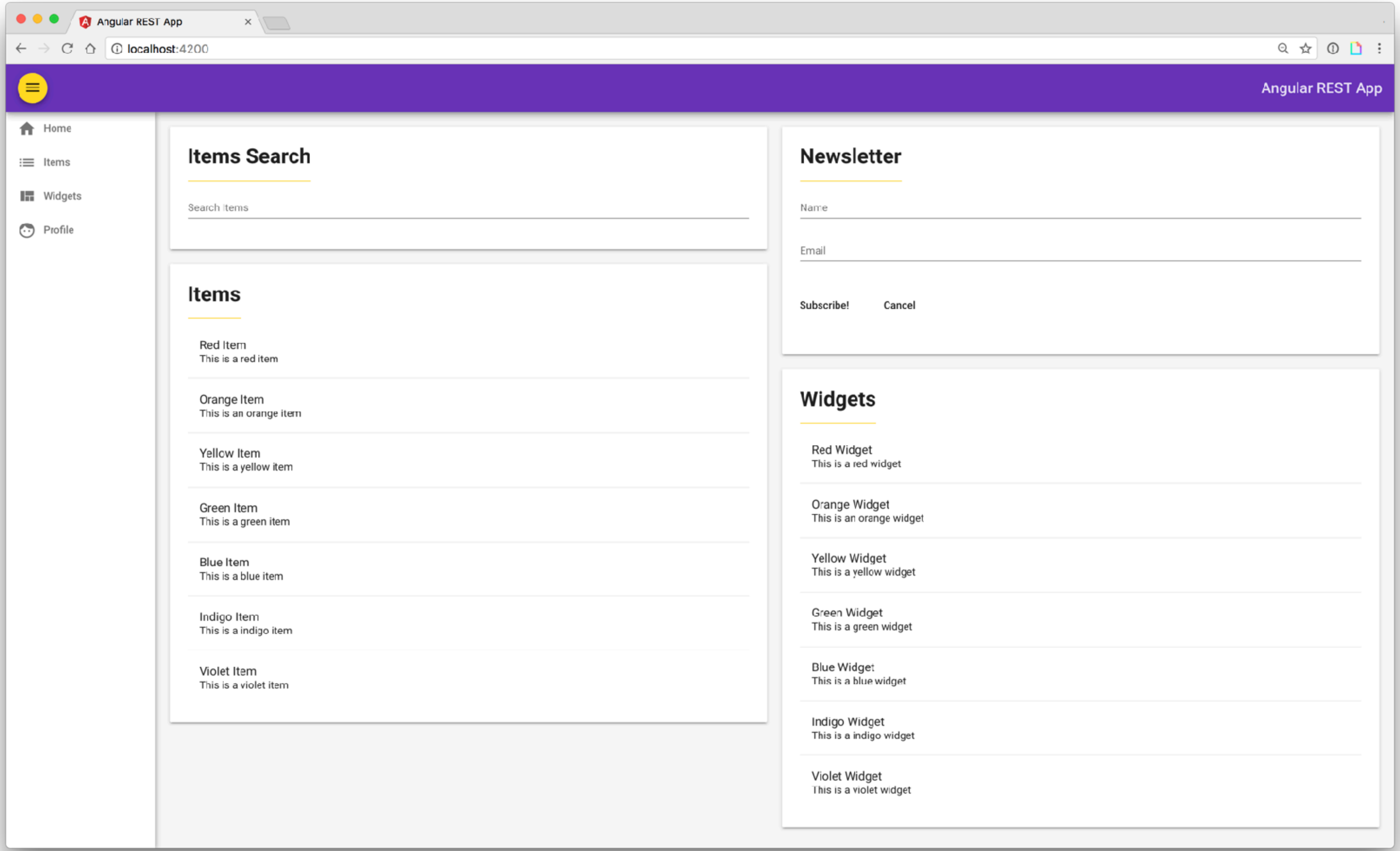
Strong grasp on how to **construct** a **single** feature in Angular

Agenda

- The Demo Application
- The Angular Big Picture
- The Angular CLI
- Components
- Templates
- Services
- Template Driven Forms

Getting **Started**

https://github.com/onehungrymind/angular-rest-app



Items Search

Search Items

Items

Red Item
This is a red item

Orange Item
This is an orange item

Yellow Item
This is a yellow item

Green Item
This is a green item

Blue Item
This is a blue item

Indigo Item
This is a indigo item

Violet Item
This is a violet item

Newsletter

Name

Email

Subscribe! Cancel

Widgets

Red Widget
This is a red widget

Orange Widget
This is an orange widget

Yellow Widget
This is a yellow widget

Green Widget
This is a green widget

Blue Widget
This is a blue widget

Indigo Widget
This is a indigo widget

Violet Widget
This is a violet widget

The Demo Application

- A simple RESTful master-detail application built using Angular and the Angular CLI
- We will be building out a new **widgets** feature
- Feel free to use the existing **items** feature as a reference point
- Please explore! Don't be afraid to try new things!

Challenges

- Make sure you can run the application

The Big Picture

Why **Angular?**

Well defined
best practices

Streamlined development workflow

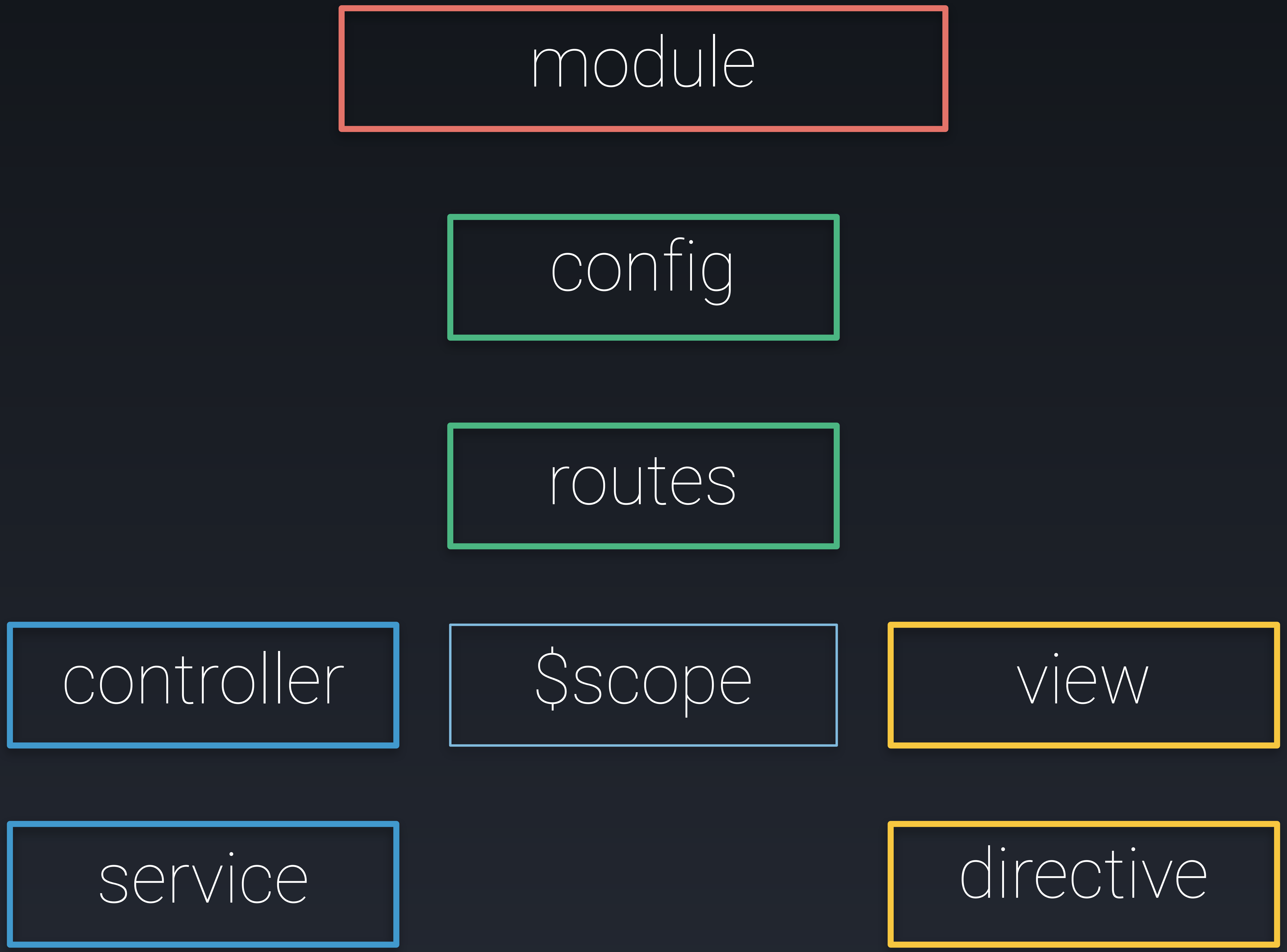
Rich ecosystem

Standards gives us **twice** the power with **half** the framework

Reactive FTW! 😍

Teamwork FTW! 😍

The Angular 1.x Big Picture



The Angular Big Picture

module

routes

component

service

The Angular Big Picture

module

routes

component

service

ES6 Modules

- ES6 modules provide organization at a **language** level
- Uses ES6 module syntax
- Modules export things that other modules can import

```
import { Component, OnInit } from '@angular/core';
import { ItemsService, Item } from '../shared';

export class ItemsComponent implements OnInit {}
```

Modules

@NgModule

- Provides organization at a **framework** level
- **declarations** define *view classes* that are available to the module
- **imports** define a list of modules that the module needs
- **providers** define a list of services the module makes available
- **bootstrap** defines the component that should be bootstrapped

```
@NgModule({
  declarations: [
    AppComponent,
    ItemsComponent,
    ItemsListComponent,
    ItemDetailComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    Ng2RestAppRoutingModule
  ],
  providers: [ItemsService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

@NgModule

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';
import { AppModule } from './app/';

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

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

Bootstrapping

The Angular Big Picture

module

routes

components

services

Routing

- Routes are defined in a route definition table that in its simplest form contains a **path** and **component** reference
- Components are loaded into the **router-outlet** component
- We can navigate to routes using the **routerLink** directive
- The router uses **history.pushState** which means we need to set a **base-ref** tag to our **index.html** file

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ItemsComponent } from './items/items.component';

const routes: Routes = [
  {path: '',       redirectTo: '/items', pathMatch: 'full' },
  {path: 'items', component: ItemsComponent},
  {path: '**',     redirectTo: '/items', pathMatch: 'full'}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})
export class Ng2RestAppRoutingModule { }
```

Routing

Components

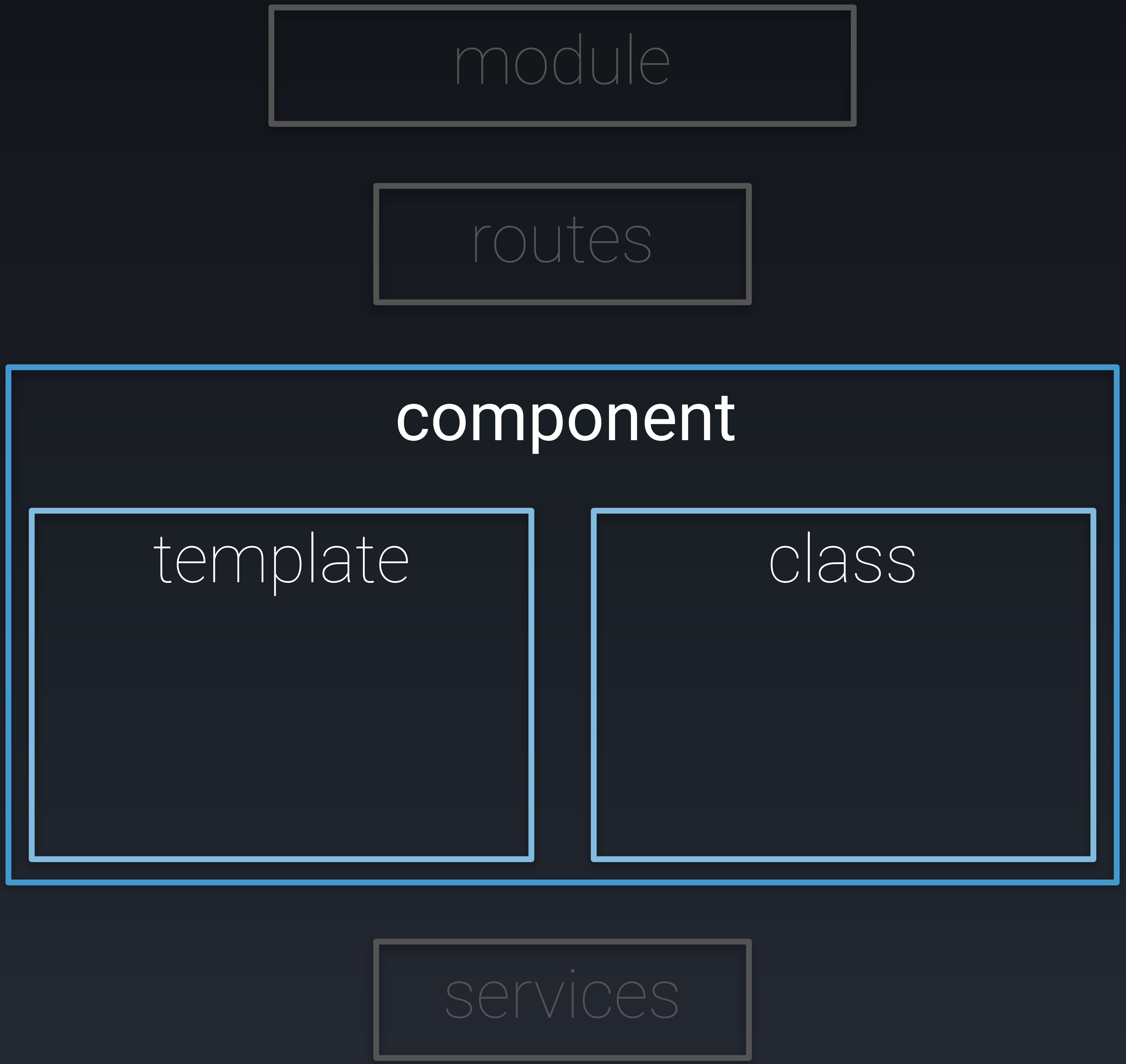
module

routes

components

services

Components



Component Classes

- Components are just ES6 classes
- Properties and methods of the component class are available to the template
- Providers (Services) are injected in the constructor
- The component lifecycle is exposed with hooks

```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Components

Templates

- A template is HTML that tells Angular how to render a component
- Templates include data bindings as well as other components and directives
- Angular leverages native DOM events and properties which dramatically reduces the need for a ton of built-in directives
- Angular leverages shadow DOM to do some really interesting things with view encapsulation

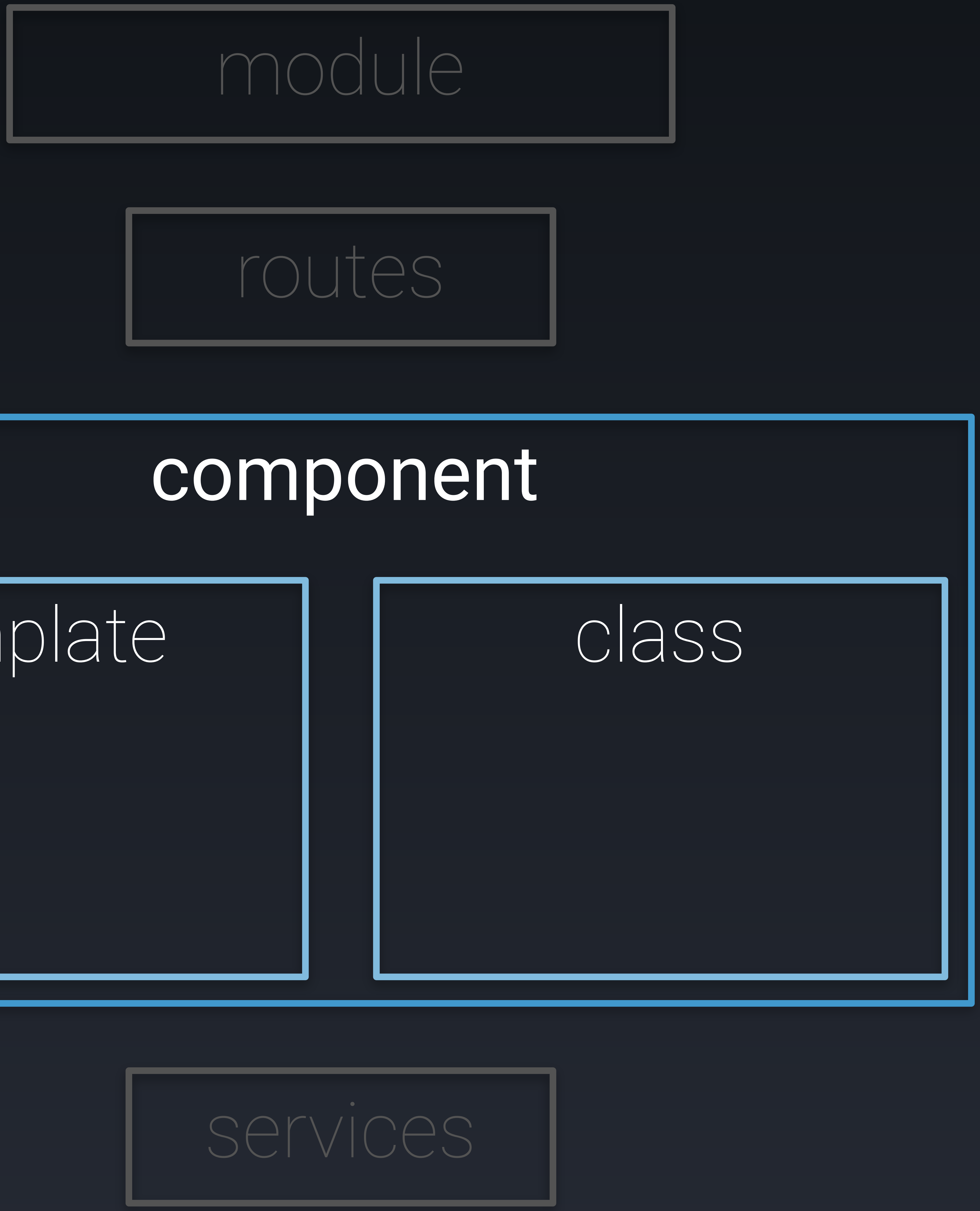
```
@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Templates

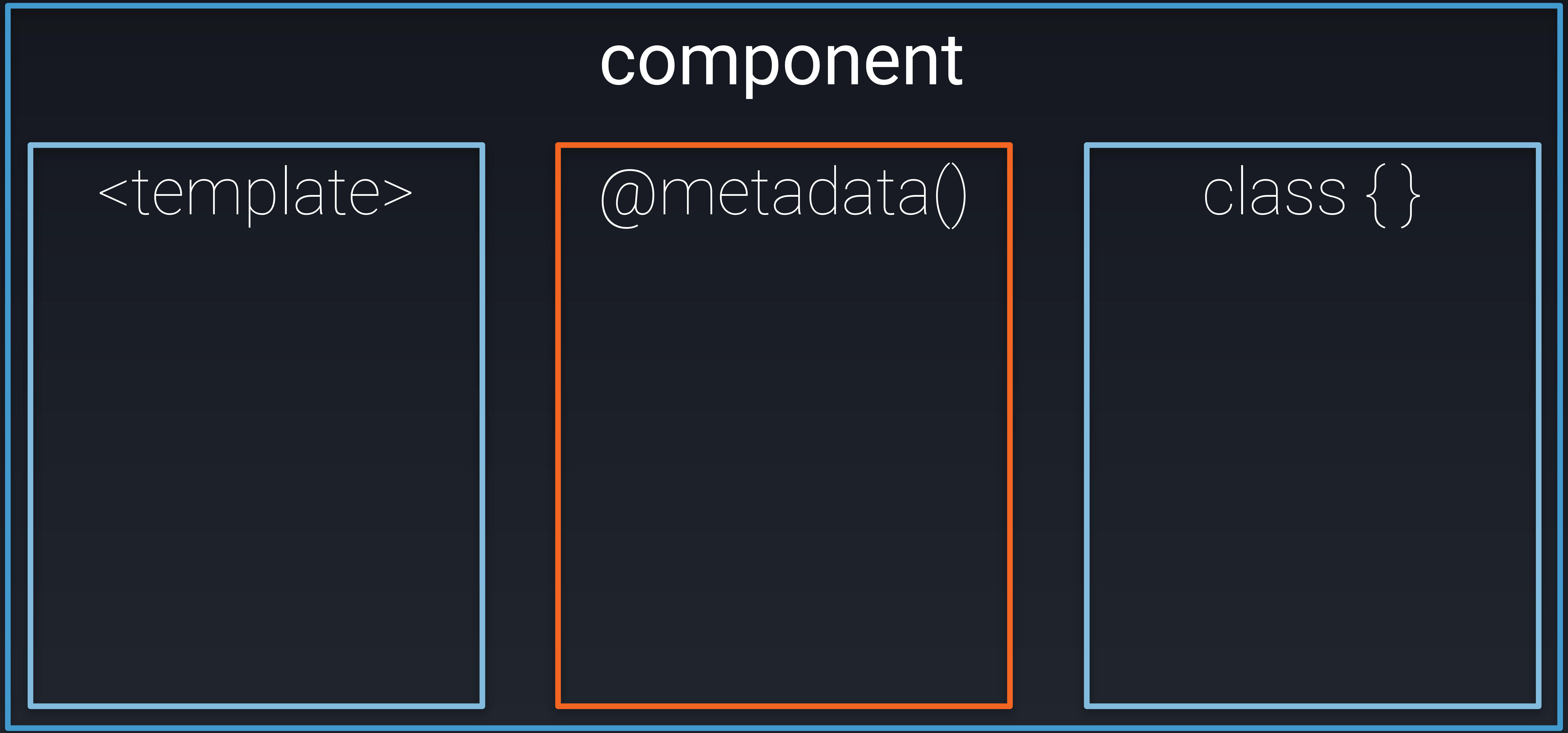
```
@Component({
  selector: 'app-items-list',
  template: `
  <div *ngFor="let item of items" (click)="selected.emit(item)">
    <div>
      <h2>{{item.name}}</h2>
    </div>
    <div>
      {{item.description}}
    </div>
    <div>
      <button (click)="deleted.emit(item); $event.stopPropagation();">
        <i class="material-icons">close</i>
      </button>
    </div>
  </div>
  `,
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Templates

Components



Metadata



Metadata

- Metadata allows Angular to process a class
- We can attach metadata with TypeScript using decorators
- Decorators are just functions
- Most common is the **@Component()** decorator
- Takes a config option with the **selector**, **templateUrl**, **styles**, **styleUrls**, **animations**, etc

```
@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent implements OnInit { }
```

Metadata

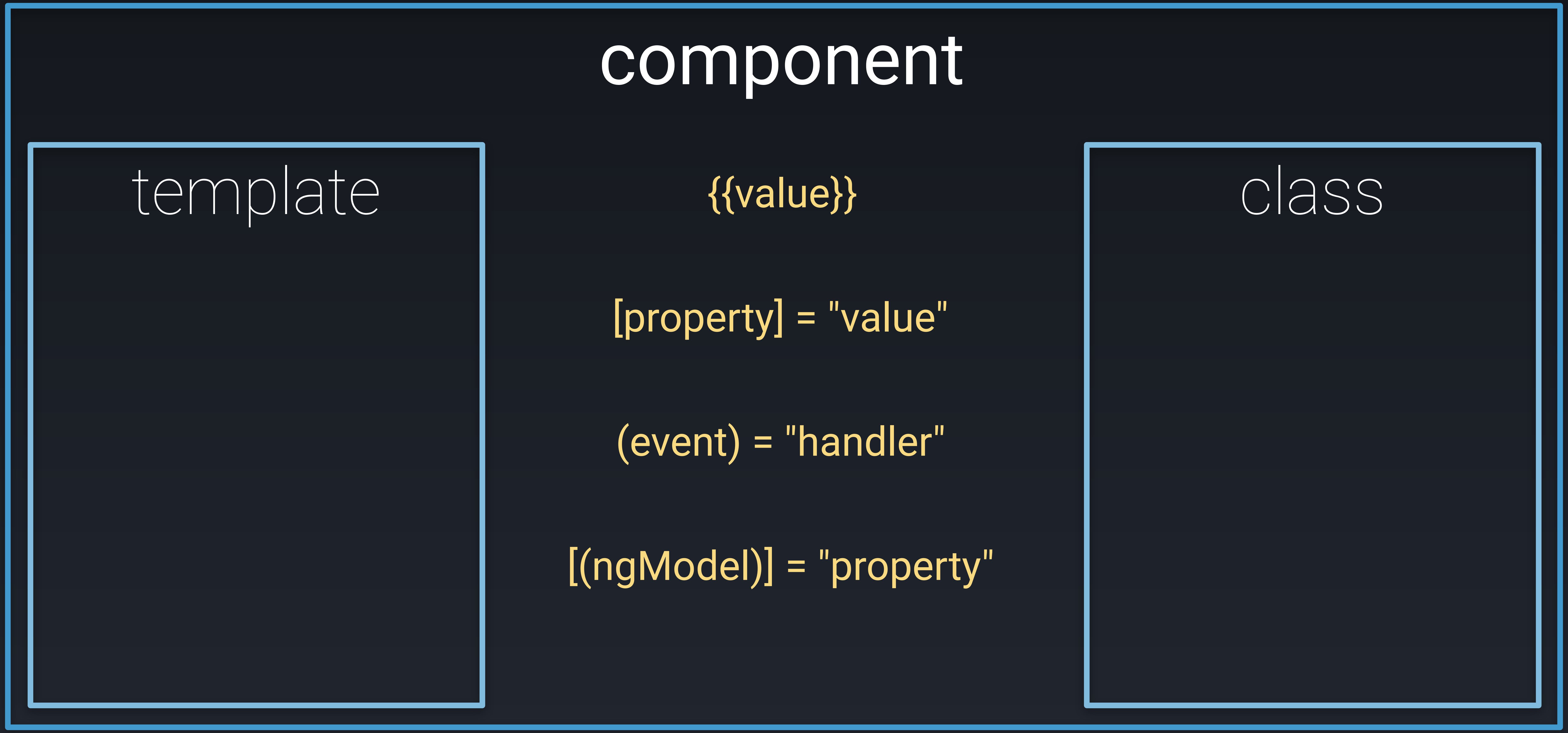
```
@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Metadata

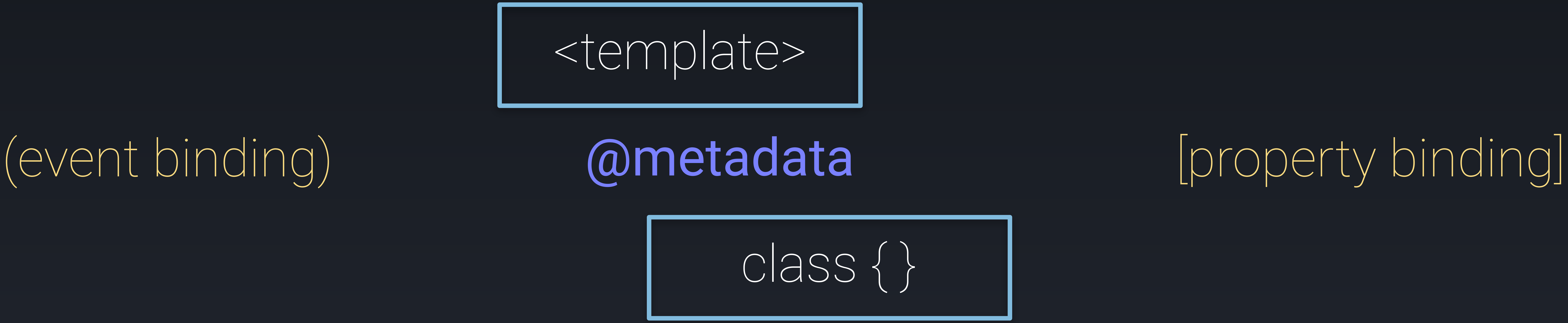
Data Binding

- Enables data to flow from the component to template and vice-versa
- Includes interpolation, property binding, event binding, and two-way binding (property binding and event binding combined)
- The binding syntax has expanded but the result is a much smaller framework footprint

Data Binding



Data Binding



```html
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let e of experiments" [experiment]="e"></experiment>
<hr/>
<div>
 <h2 class="text-error">Experiments: {{message}}</h2>
  <form class="form-inline">
    <input type="text" [(ngModel)]="message" placeholder="Message">
    <button type="submit" class="btn" (click)="updateMessage(message)">Update Message</button>
  </form>
</div>
```

Data Binding

BUT! What about directives?

Directives

- A directive is a class decorated with **@Directive**
- A component is just a directive with added template features
- Built-in directives include structural directives and attribute directives

```
import { Directive, ElementRef } from '@angular/core';

@Directive({selector: 'blink'})
export class Blinker {
  constructor(element: ElementRef) {
    // All the magic happens!
  }
}
```

Directives

```
import { Directive, ElementRef } from '@angular/core';

@Directive({selector: 'blink'})
export class Blinker {
  constructor(element: ElementRef) {
    // All the magic happens!
  }
}
```

Directives

Services

module

routes

components

services

Services

- A service is *generally* just a class
- Should only do one specific thing
- Take the burden of business logic out of components
- It is considered best practice to always use **@Injectable** so that metadata is generated correctly

```
import { Injectable } from '@angular/core';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

const BASE_URL = 'http://localhost:3000/items/';

@Injectable()
export class ItemsService {
  constructor(private http: Http) {}

  loadItems() {
    return this.http.get(BASE_URL)
      .map(res => res.json())
      .toPromise();
  }
}
```

Services

BONUS! TypeScript Time!

```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Component

```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Types

```
export interface Item {
  id: number;
  name: string;
  description: string;
}
```

Interface

```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Field Assignment

```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Constructor Assignment

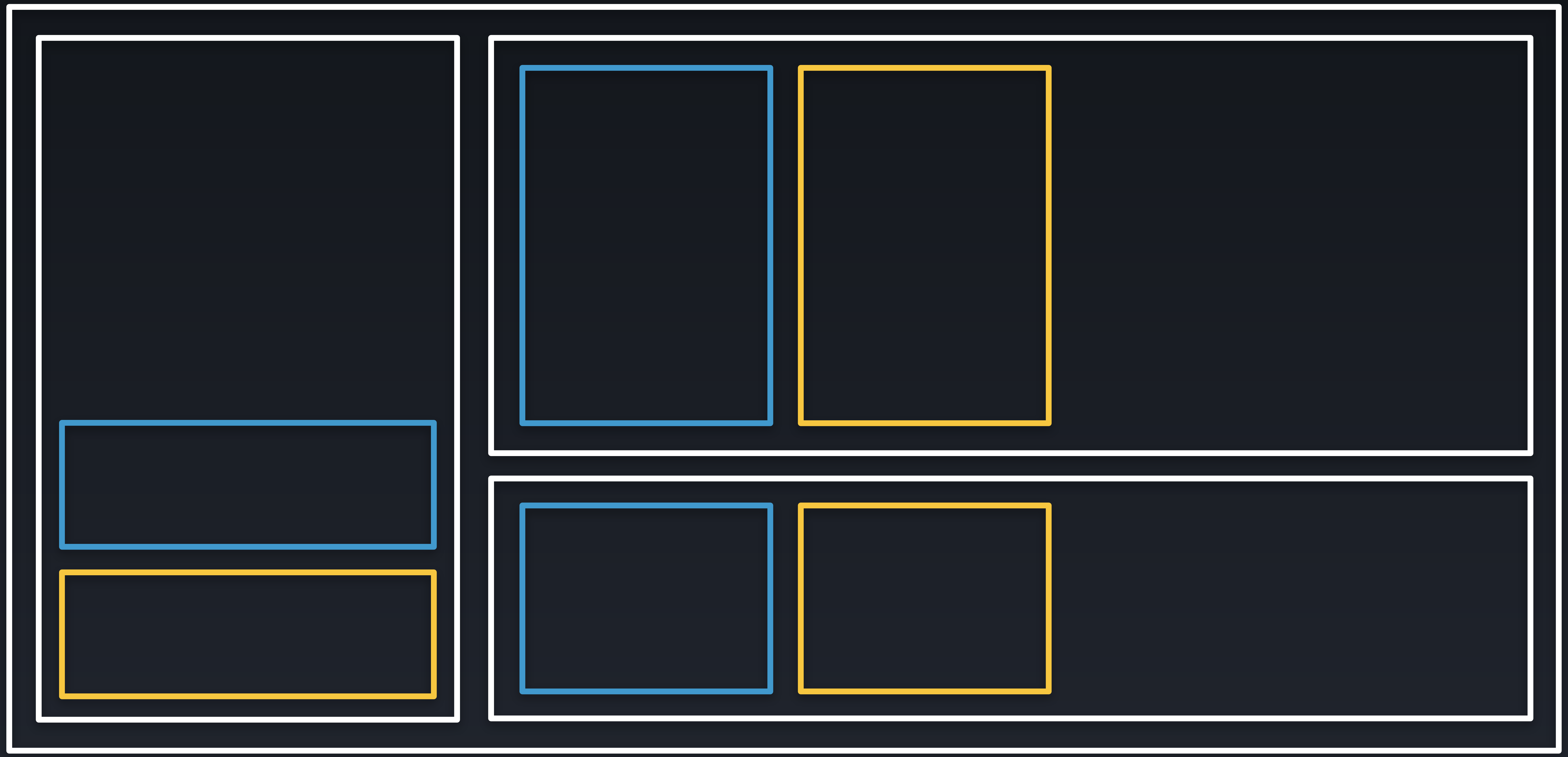
```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

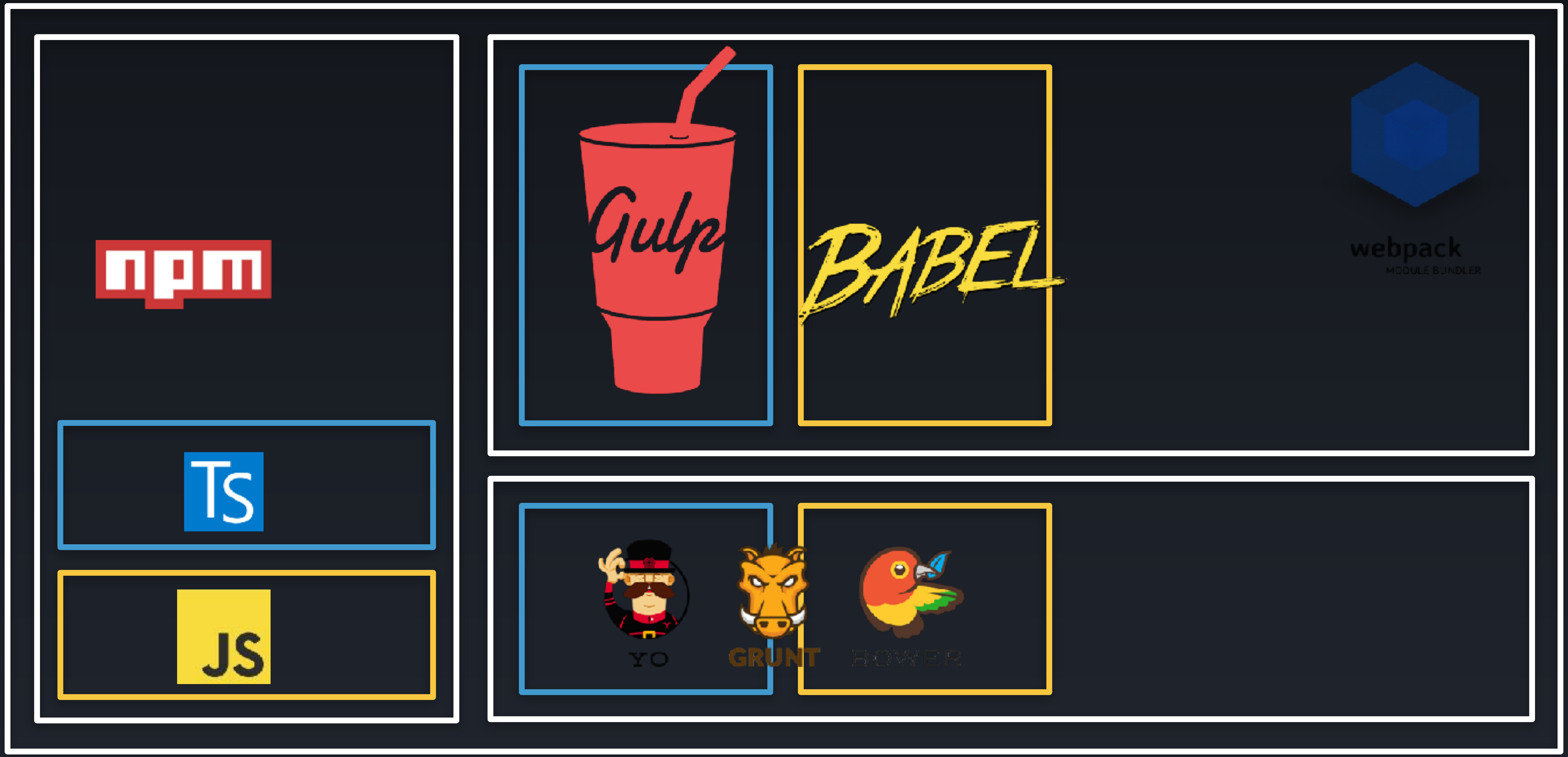
Implements Interface

Challenges

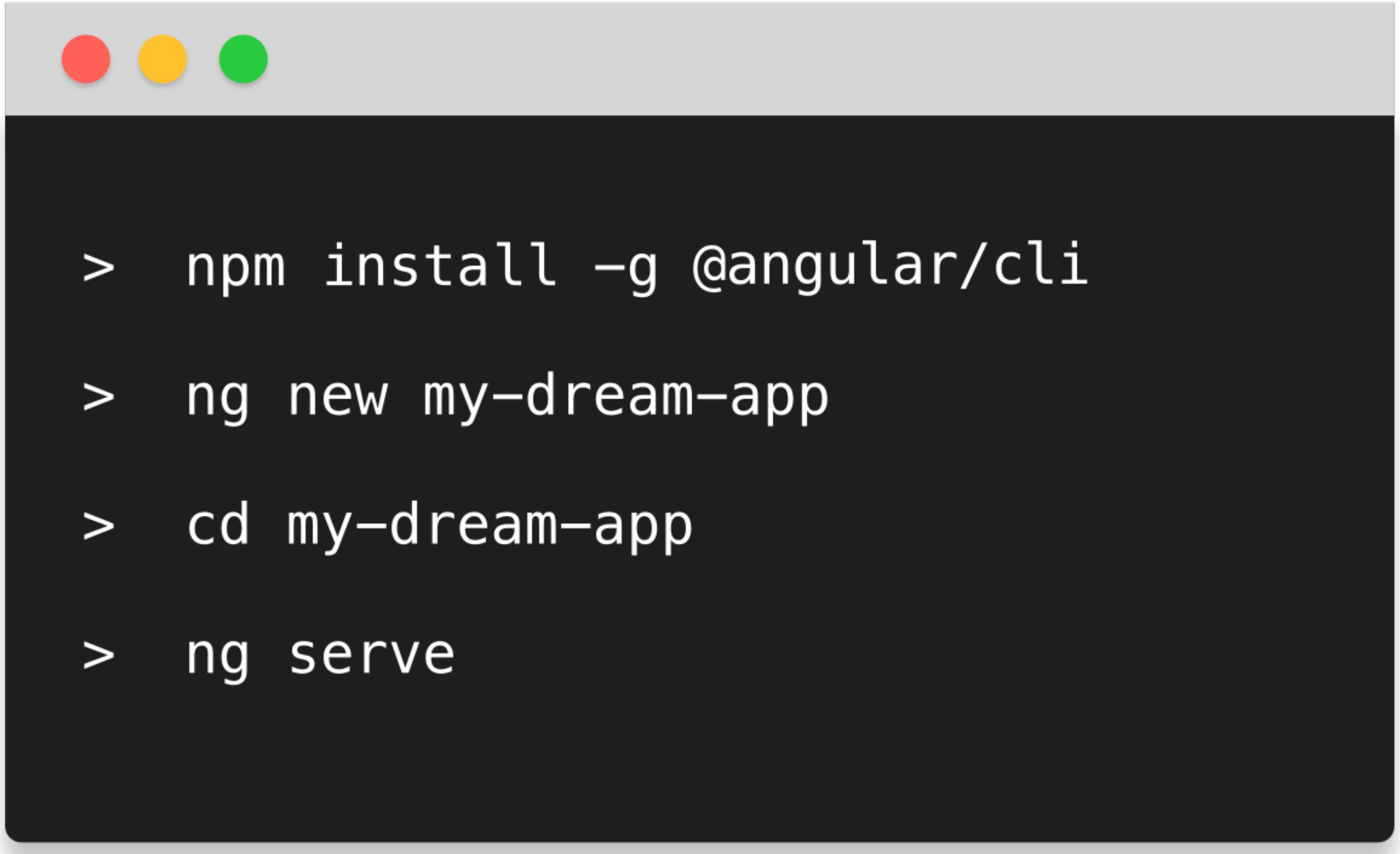
- Identify the major Angular pieces in the sample application
- Add a new property to the **Items** component and bind to it in the view
- Add a new property to the **ItemsService** and consume it in a component

The Angular CLI




npm
TS
JS
Gulp
BABEL
webpack
MODULE BUNDLER
YO
GRUNT
BOWER

ANGULAR CLI



```
>  npm install -g @angular/cli

>  ng new my-dream-app

>  cd my-dream-app

>  ng serve
```

Angular CLI !== Crutch

Includes

- Fully functional project generation THAT JUST WORKS!
- Code generator for components, directives, pipes, enums, classes, modules and services
- Build generation
- Unit test runner
- End-to-end test runner
- App deployment GitHub pages
- Linting
- CSS preprocessor support
- AOT support
- Lazy routes
- **Extensible blueprints coming soon**

```
npm install -g @angular/cli
```

Installing the CLI

```
ng new PROJECT_NAME
cd PROJECT_NAME
ng serve
```

Generating a project

```
ng generate component my-new-component
ng g c my-new-component # using the alias
ng g c my-new-component -m app.module.ts
```

Generating a component

```
ng generate service my-new-service
ng g s my-new-service # using the alias
```

Generating a service

```
ng build
```

Generating a build

```
ng test
ng e2e
```

Running tests

```
ng lint
```

Linting

```
ng github-pages:deploy --message "Optional
commit message"
```

Deploying the app

Challenges

- Check out the **00-start** branch
- Scaffold out a **gizmo component** with the module flag
- Scaffold out a **gizmo service**
- Run the tests
- Build the application
- **BONUS: Create a gizmo route**

NOTE: Use the Angular CLI for ALMOST all of the tasks above

Component
Fundamentals

Component Fundamentals

- Anatomy of a Component
- **C**lass **I**mport **D**ecorate **E**nhance **R**epeat
- Enhance with properties and methods
- Enhance with injectables
- Lifecycle Hooks

Anatomy of a Component

<template>

(event binding)

@metadata

[property binding]

class { }

Class !== Inheritance

Class Definition

- Create the component as an ES6 class
- Properties and methods on our component class will be available for binding in our template

```
export class ItemsComponent {}
```

Class

Import

- Import the core Angular dependencies
- Import 3rd party dependencies
- Import your custom dependencies
- This approach gives us a more fine-grained control over the managing our dependencies

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

Import

Class Decoration

- We turn our class into something Angular can use by decorating it with a Angular specific metadata
- Use the **@Component** syntax to decorate your classes
- You can also decorate properties and methods within your class
- The two most common member decorators are **@Input** and **@Output**

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

@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent {}
```

Decorate

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

@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent {
  items: Array<Item>;
  selectedItem: Item;

  constructor() {}

  resetItem() {
    let emptyItem: Item = {id: null, name: '', description: ''};
    this.selectedItem = emptyItem;
  }

  selectItem(item: Item) {
    this.selectedItem = item;
  }
}
```

Properties and Methods

```typescript
import { Component, OnInit } from '@angular/core';
import { ItemsService, Item } from '../shared';

@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Injecting a Dependency

Lifecycle Hooks

- Allow us to perform custom logic at various stages of a component's life
- Data isn't always immediately available in the constructor
- The lifecycle interfaces are optional. We recommend adding them to benefit from TypeScript's strong typing and editor tooling
- Implemented as class methods on the component class

Lifecycle Hooks Continued

- **ngOnChanges** called when an input or output binding value changes
- **ngOnInit** called after the first ngOnChanges
- **ngDoCheck** handles developer's custom change detection
- **ngAfterContentInit** called after component content initialized
- **ngAfterContentChecked** called after every check of component content
- **ngAfterViewInit** called after component's view(s) are initialized
- **ngAfterViewChecked** called after every check of a component's view(s)
- **ngOnDestroy** called just before the directive is destroyed.

Lifecycle Hooks Continued

- **ngOnChanges** called when an input or output binding value changes
- **ngOnInit** called after the first ngOnChanges
- **ngDoCheck** handles developer's custom change detection
- **ngAfterContentInit** called after component content initialized
- **ngAfterContentChecked** called after every check of component content
- **ngAfterViewInit** called after component's view(s) are initialized
- **ngAfterViewChecked** called after every check of a component's view(s)
- **ngOnDestroy** called just before the directive is destroyed.

```
import { Component, OnInit } from '@angular/core';
import { ItemsService, Item } from '../shared';

@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Lifecycle Hooks

Demonstration

Challenges

- Create the file structure for a new **widgets** feature
- Create the ES6 class for the **widgets** component
- Import the appropriate modules into the **widgets** component
- Decorate the **widgets** component to use the **widgets** template
- Display the **widgets** component in the **home** component
- **BONUS Create a simple route to view the widgets component by itself**

Template
Fundamentals

Template Fundamentals

- Property Binding
- Event Binding
- Two-way Binding
- Local Template Variables
- Attribute Directives
- Structural Directives
- Safe Navigation Operator

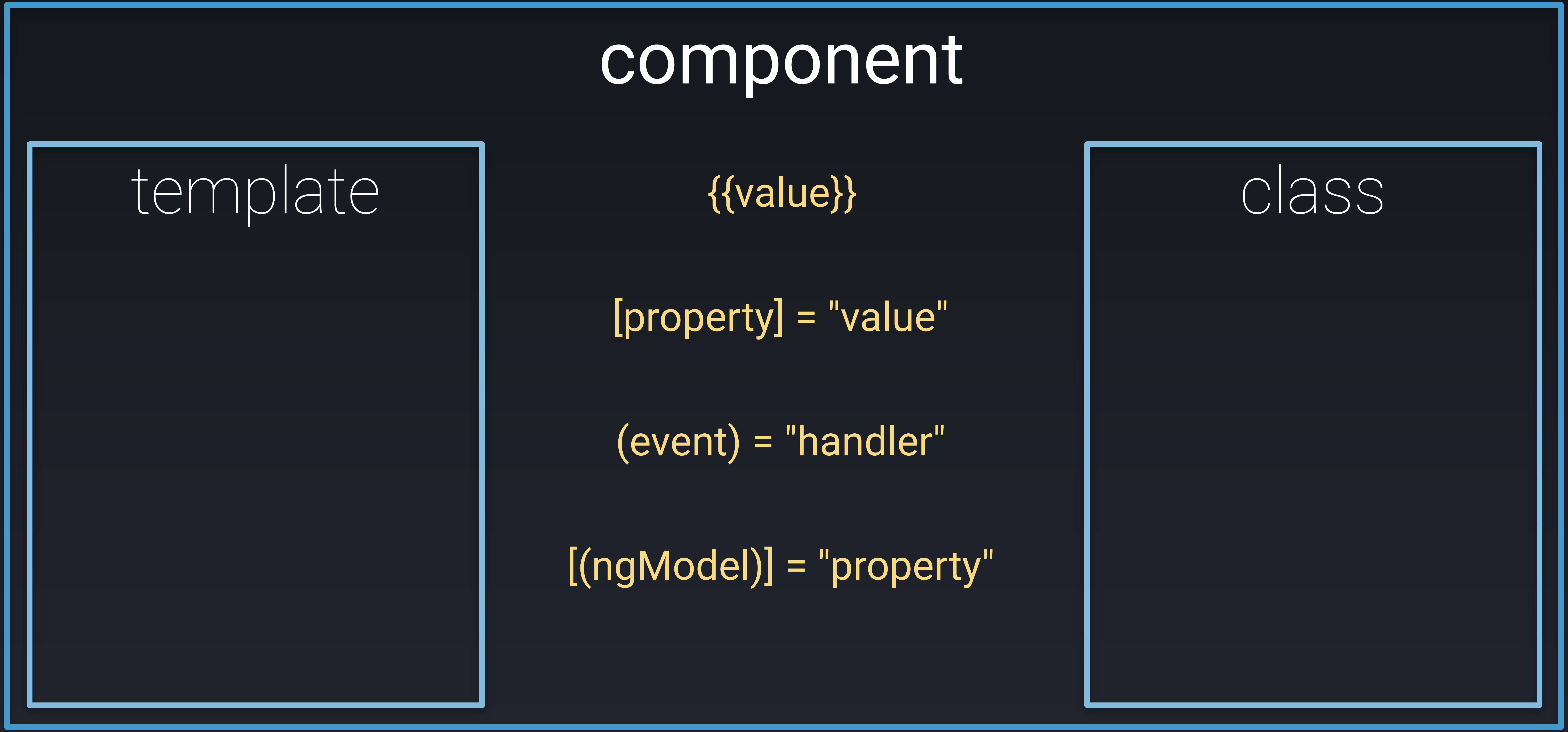
Templates

<template>

(event binding) @metadata [property binding]

class { }

Data Binding



Property Binding

- Flows data from the component to an element
- Created with brackets **<img [src]=”image.src” />**
- The canonical form of **[property]** is **bind-property**
- There are special cases for binding to attributes, classes and styles that look like **[attr.property]**, **[class.className]**, and **[style.styleName]** respectively

```html
<span [style.color]="componentStyle">Some colored text!</span>
```

Property Bindings

Event Binding

- Flows data from an element to the component
- Created with parentheses **<button (click)=”foo($event)”></button>**
- The canonical form of **(event)** is **on-event**
- Information about the target event is carried in the **$event** parameter

```
<button (click)="alertTheWorld($event)">Click me!</button>
```

Event Bindings

Two-way Binding

- Really just a combination of property and event bindings
- Used in conjunction with **ngModel**
- Referred to as "banana in a box"

```html
<label>The awesome input</label>
<input [(ngModel)]="dynamicValue" placeholder="Watch the text update!" type="text">
<label>The awesome output</label>
<span>{{dynamicValue}}</span>
```

Two-way Binding

Local Template Variable

- The hashtag (#) defines a local variable inside our template
- We can refer to a local template variable *anywhere* in the current template
- To consume, simply use it as a variable without the hashtag
- The canonical form of **#variable** is **ref-variable**

```
<form novalidate #formRef="ngForm">
  <label>Item Name</label>
  <input [(ngModel)]="selectedItem.name"
    type="text" name="name" required
    placeholder="Enter a name">

  <label>Item Description</label>
  <input [(ngModel)]="selectedItem.description"
    type="text" name="description"
    placeholder="Enter a description">

  <button type="submit"
    [disabled]="!formRef.valid"
    (click)="saved.emit(selectedItem)">Save</button>
</form>
```

Local Template Variable

Structural Directives

- A structural directive changes the DOM layout by adding and removing DOM elements.
- Asterisks indicate a directive that modifies the HTML
- It is syntactic sugar to avoid having to use template elements directly

```
<div *ngIf="hero">{{hero}}</div>

<div *ngFor="let hero of heroes">{{hero}}</div>

<span [ngSwitch]="toeChoice">
  <span *ngSwitchCase="'Eenie'">Eenie</span>
  <span *ngSwitchCase="'Meanie'">Meanie</span>
  <span *ngSwitchCase="'Miney'">Miney</span>
  <span *ngSwitchCase="'Moe'">Moe</span>
  <span *ngSwitchDefault>other</span>
</span>
```

Structural Directives

```html
<span [ngSwitch]="toeChoice">
  <!-- with *NgSwitch -->
  <span *ngSwitchCase="'Eenie'">Eenie</span>
  <span *ngSwitchCase="'Meanie'">Meanie</span>
  <span *ngSwitchCase="'Miney'">Miney</span>
  <span *ngSwitchCase="'Moe'">Moe</span>
  <span *ngSwitchDefault>other</span>

  <!-- with <template> -->
  <template [ngSwitchCase]="'Eenie'"><span>Eenie</span></template>
  <template [ngSwitchCase]="'Meanie'"><span>Meanie</span></template>
  <template [ngSwitchCase]="'Miney'"><span>Miney</span></template>
  <template [ngSwitchCase]="'Moe'"><span>Moe</span></template>
  <template ngSwitchDefault><span>other</span></template>
</span>
```

Template Tag

Safe Navigation Operator

- Denoted by a question mark immediately followed by a period e.g. **?.**
- If you reference a property in your template that does not exist, you will throw an exception.
- The safe navigation operator is a simple, easy way to guard against null and undefined properties

```
<!-- No hero, no problem! -->
The null hero's name is {{nullHero?.firstName}}
```

Safe Navigation Operator

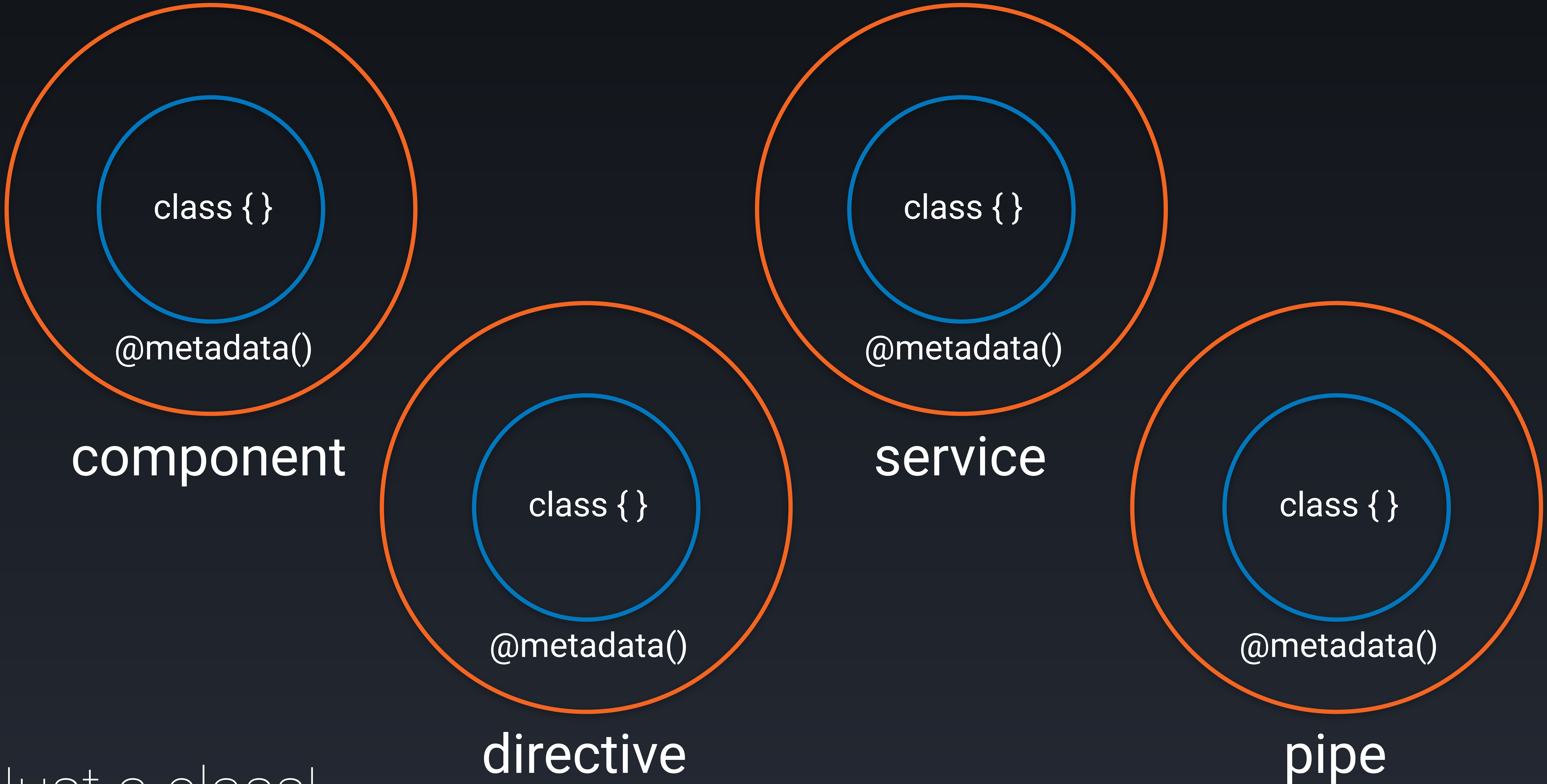
Demonstration

Challenges

- Create a **widgets** collection in the **widget** component with mock objects
- Create a **selectedWidget** property in the widget component
- Display the **widgets** collection in the template using **ngFor**
- Use an **event binding** to set a selected widget
- Display the **widget** properties using **property binding** and **interpolation binding**
- Use **ngIf** to show an alternate message if no widget is selected

ACTION ITEM! Go to http://bit.ly/widgets-snippets to save on typing

Services


class { }
@metadata()
component
class { }
@metadata()
directive
class { }
@metadata()
service
class { }
@metadata()
pipe

Just a class!

Services

- Defining a Service
- Exposing a Service
- Consuming a Service

```
@Injectable()
export class ItemsService {
  constructor(private http: Http) {}

  loadItems() { }

  loadItem(id) { }

  saveItem(item: Item) { }

  createItem(item: Item) { }

  updateItem(item: Item) { }

  deleteItem(item: Item) { }
}
```

Defining a Service

```
@NgModule({
  declarations: [
    AppComponent,
    ItemsComponent,
    ItemsListComponent,
    ItemDetailComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    Ng2RestAppRoutingModule
  ],
  providers: [ItemsService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Exposing a Service

```
import { Component, OnInit } from '@angular/core';
import { ItemsService, Item } from '../shared';

@Component({
  selector: 'app-items',
  templateUrl: './items.component.html',
  styleUrls: ['./items.component.css']
})
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

Consuming a Service

Demonstration

Challenges

- Extract the **widgets collection** to a **widgets service**
- Add the **widgets service** to the **application module** so that it can be consumed
- Inject that **widgets service** into the **widgets component**
- Consume and display the new **widgets collection**
- Create a **widget** interface and strongly type **selectedWidget** and **widgets collection**

Template
Driven Forms

Template Driven Forms

- FormsModule
- Form Controls
- Validation Styles

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
```

FormsModule

ngModel

- Enables two-way data binding within a form
- Creates a **FormControl** instance from a domain model and binds it to a form element
- We can create a local variable to reference the **ngModel** instance of the element

```
<input [(ngModel)]="selectedItem.name"
       name="name" #nameRef="ngModel"
       placeholder="Enter a name"
       type="text">
```

ngModel

Form Controls

- **ngControl** binds a DOM element to a **FormControl**
- **FormControl** is responsible for tracking value and validation status of a single form element
- You can group **FormControl**s together with **FormGroup**
- **ngForm** binds an HTML form to a top-level **FormGroup**
- We can create a local variable to reference the **ngForm** instance of a form
- **ngModelGroup** creates and binds a **FormGroup** instance to a DOM element

```
<form novalidate #formRef="ngForm">
  <div>
    <label>Item Name</label>
    <input [(ngModel)]="selectedItem.name"
      name="name" required
      placeholder="Enter a name" type="text">
  </div>
  <div>
    <label>Item Description</label>
    <input [(ngModel)]="selectedItem.description"
      name="description"
      placeholder="Enter a description" type="text">
  </div>
</form>
```

ngForm

```html
<pre>{{formRef.value | json}}</pre>
<pre>{{formRef.valid | json}}</pre>

<!--
{
  "name": "First Item",
  "description": "Item Description"
}
true
-->
```

ngForm

```html
<form novalidate #formRef="ngForm">
  <fieldset ngModelGroup="user">
    <label>First Name</label>
    <input [(ngModel)]="user.firstName"
      name="firstName" required
      placeholder="Enter your first name" type="text">
    <label>Last Name</label>
    <input [(ngModel)]="user.lastName"
      name="lastName" required
      placeholder="Enter your last name" type="text">
  </fieldset>
</form>
```

ngModelGroup

```html
<div ngModelGroup="user">
  <label>First Name</label>
  <input [(ngModel)]="firstName"
    name="firstName" required
    placeholder="Enter your first name" type="text">
  <label>Last Name</label>
  <input [(ngModel)]="lastName"
    name="lastName" required
    placeholder="Enter your last name" type="text">
</div>
<pre>{{formRef.value | json}}</pre>

<!--
{
  "user": {
    "firstName": "Test",
    "lastName": "Test"
  }
}
-->
```

ngModelGroup

Validation Styles

- Angular will automatically attach styles to a form element depending on its state
- For instance, if it is in a valid state then **ng-valid** is attached
- If the element is in an invalid state, then **ng-invalid** is attached
- There are additional styles such as **ng-pristine** and **ng-untouched**

```css
input.ng-invalid {
    border-bottom: 1px solid red;
}

input.ng-valid {
    border-bottom: 1px solid green;
}
```

Validation Styles

Demonstration

Challenges

- Create a form to display the currently selected **widget**
- Create a button to **save** the edited **widget**
- Create a button to **cancel** editing the **widget**
- Using **ngForm**, add in some validation for editing the **widget** component

@simpulton

Thanks!

WAT.