

# Learn to Build Awesome Apps with Angular 2



Strong grasp on how to **construct** a **single** feature in Angular 2

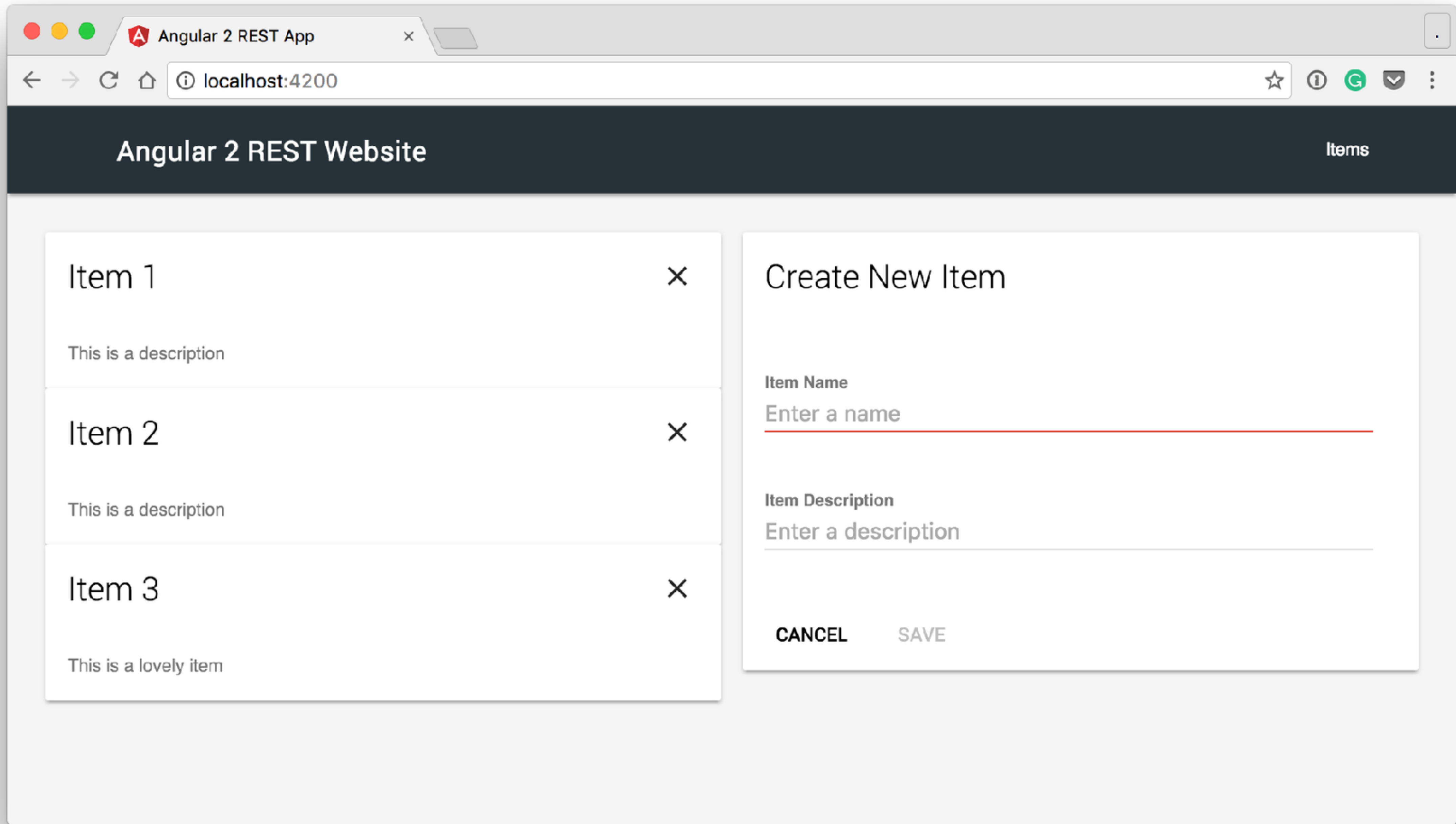
# Agenda

- **The Demo Application**
- **The Angular 2 Big Picture**
- **The Angular CLI**
- **Components**
- **Templates**
- **Services**
- **Routing**

# Getting Started



<https://github.com/onehungrymind/ng2-rest-app>



# The Demo Application

- A simple RESTful master-detail application built using Angular 2 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

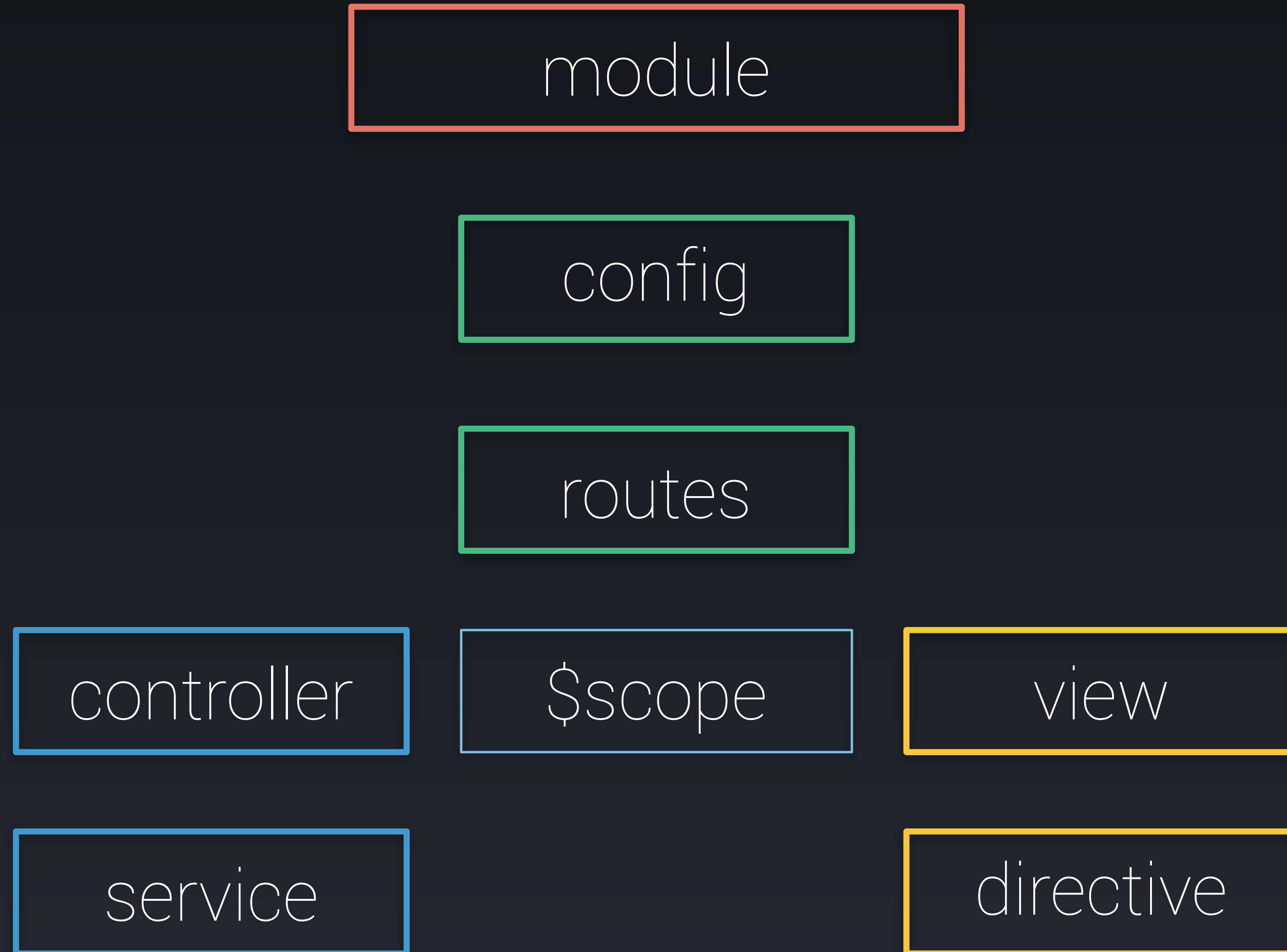
# Why Angular 2?

- Distilled all the best practices of Angular 1.x into Angular 2
- By focusing on standards, we get **twice** the power with **half** the framework
- Dramatically improved change detection with a relentless focus on speed and performance
- Reactive mechanisms baked into the framework 🥰
- Teamwork! The Angular team is working with some really smart people from other projects to make Angular and web development awesome

# The Main Building Blocks

- Modules
- Routes
- Components
- Services

# The Angular 1.x Big Picture



# The Angular 2 Big Picture

module

routes

component

service

# The Angular 2 Big Picture

module

routes

component

service

# ES6 Modules

- Uses ES6 module syntax
- Angular 2 applications use modules as the core mechanism for composition
- Modules export things that other modules can import
- Keep your modules fine-grained and self-documenting

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

# Modules



# @NgModule

- The organizational mechanism used within an Angular application
- **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,
    HttpClientModule,
    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 2 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: '',      component: ItemsComponent },
  {path: 'items', component: ItemsComponent},
  {path: '*',     component: ItemsComponent }
];

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

# Routing

# Components

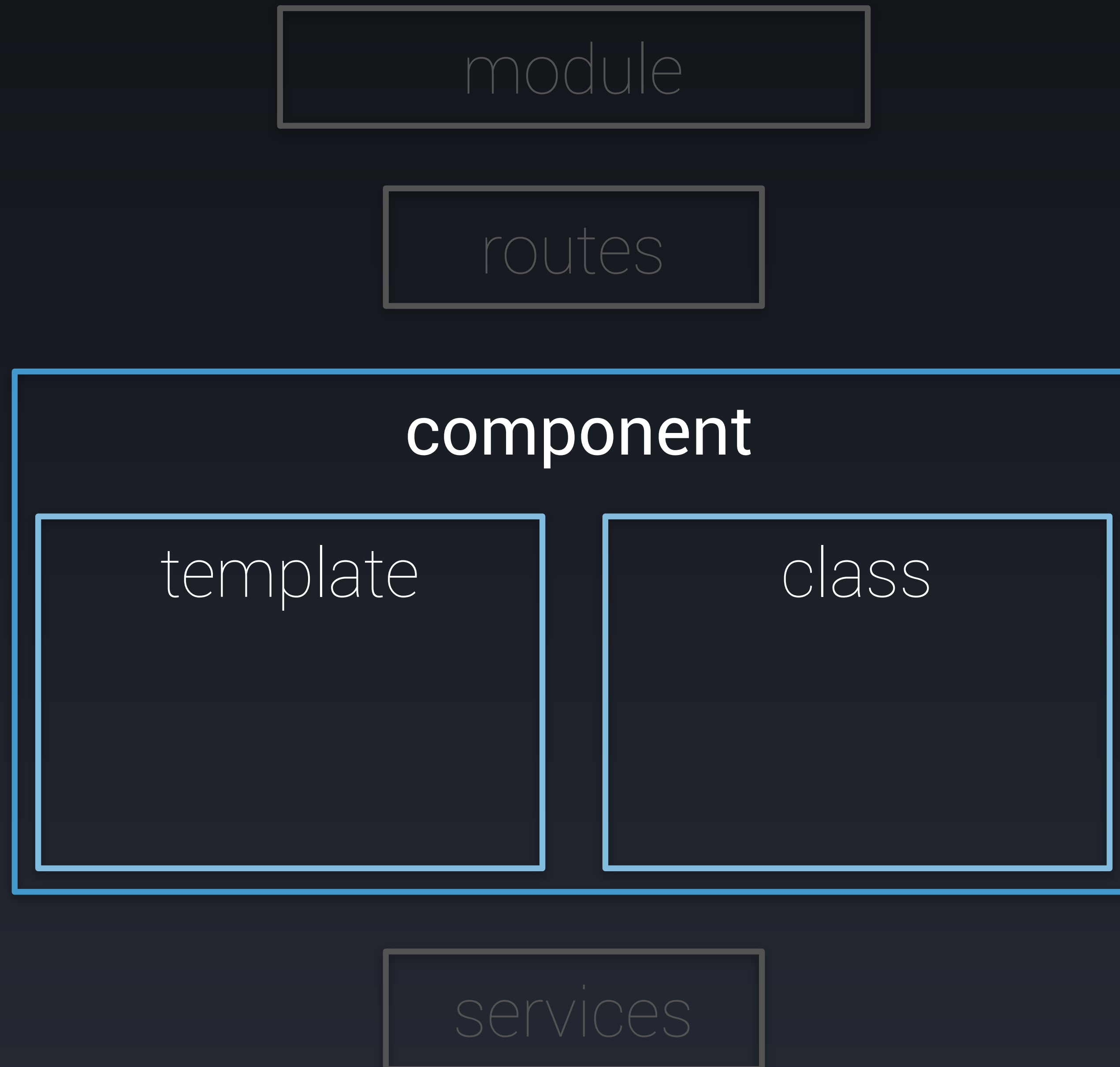
module

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
- Hook into the component lifecycle 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 2 leverages native DOM events and properties which dramatically reduces the need for a ton of built-in directives
- Angular 2 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();
}
```

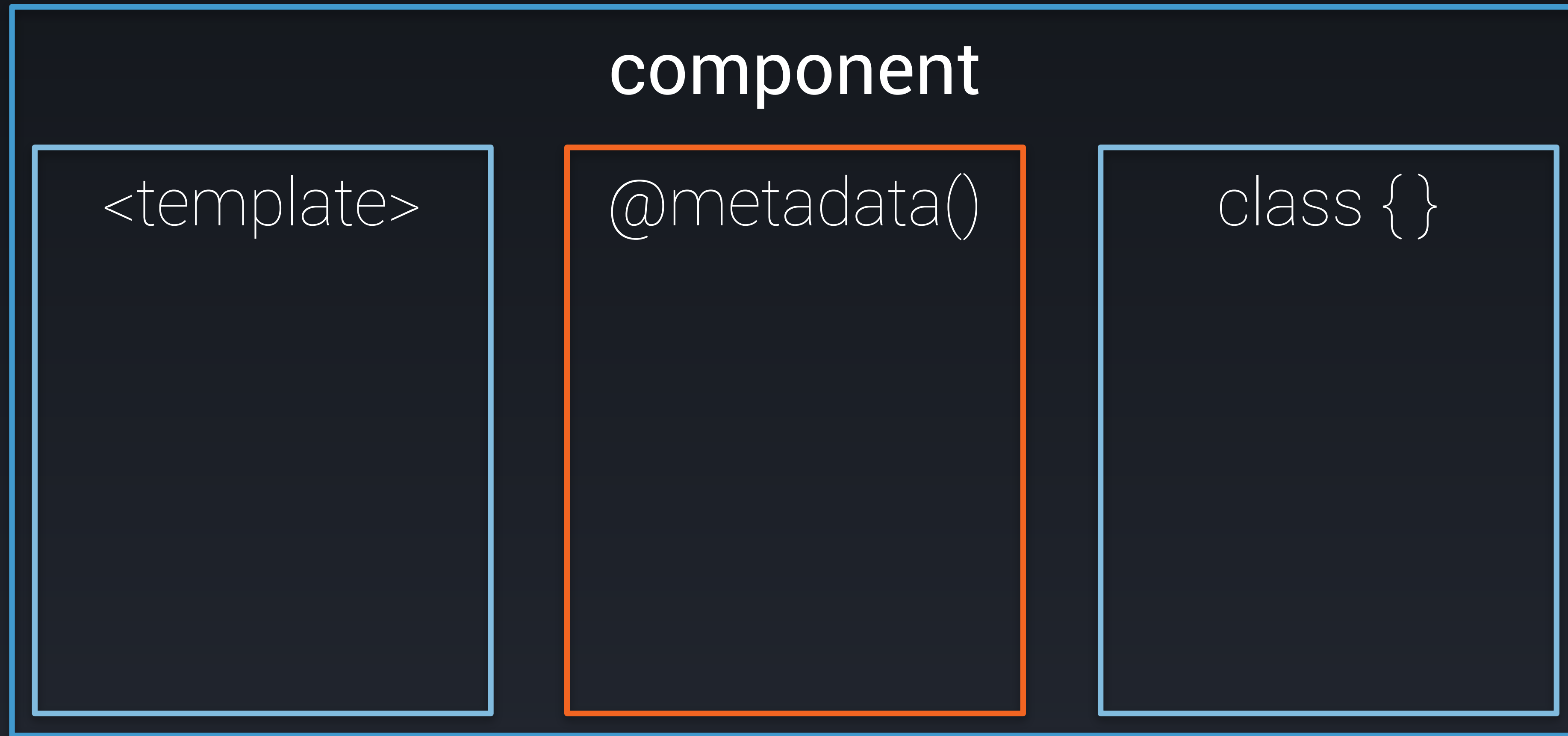
# Templates

```
@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

# 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**, **styleUrl**, **animations**, etc

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

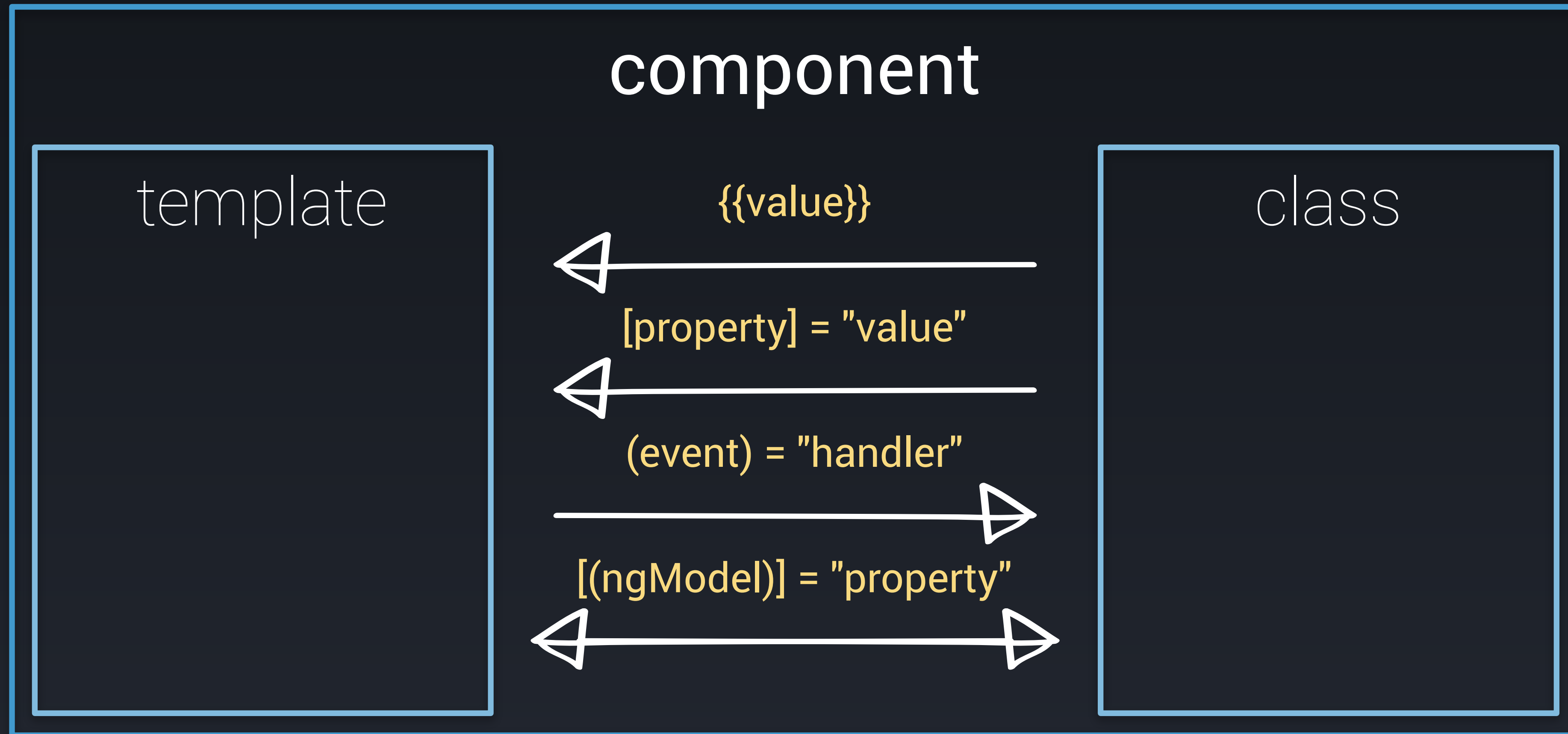
# Metadata



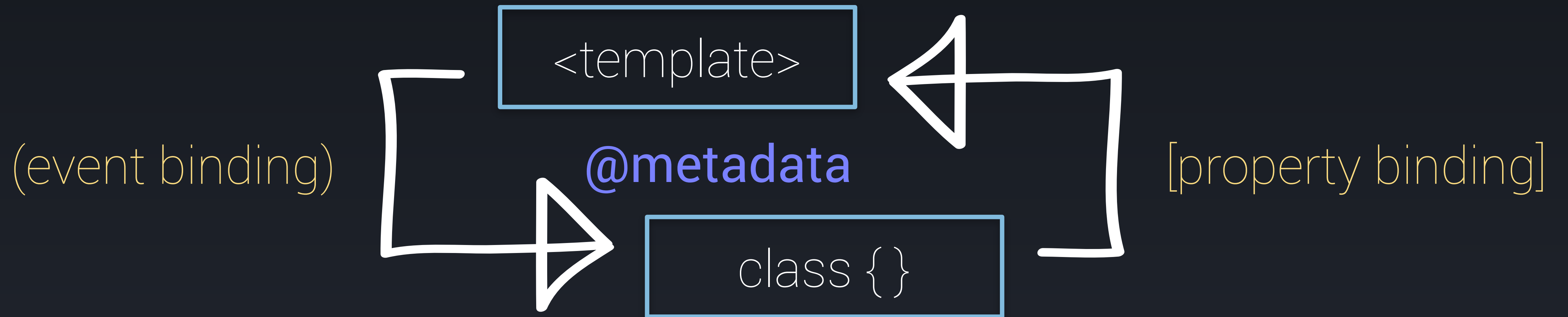
# Data Binding

- 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



```
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let experiment of experiments" [experiment]="experiment"></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: '[blinker]'})
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) {}

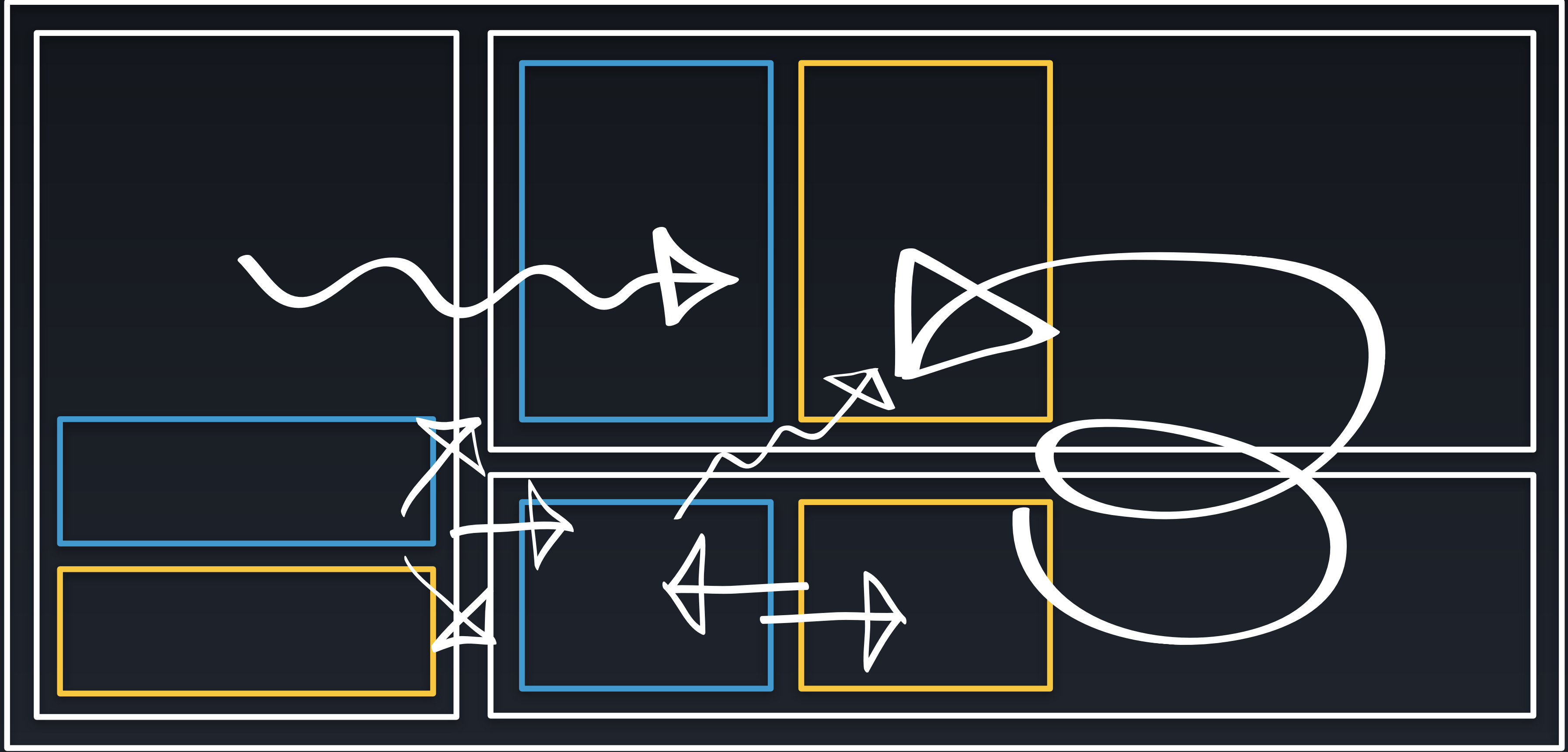
  loadItems() {
    return this.http.get(BASE_URL)
      .map(res => res.json())
      .toPromise();
  }
}
```

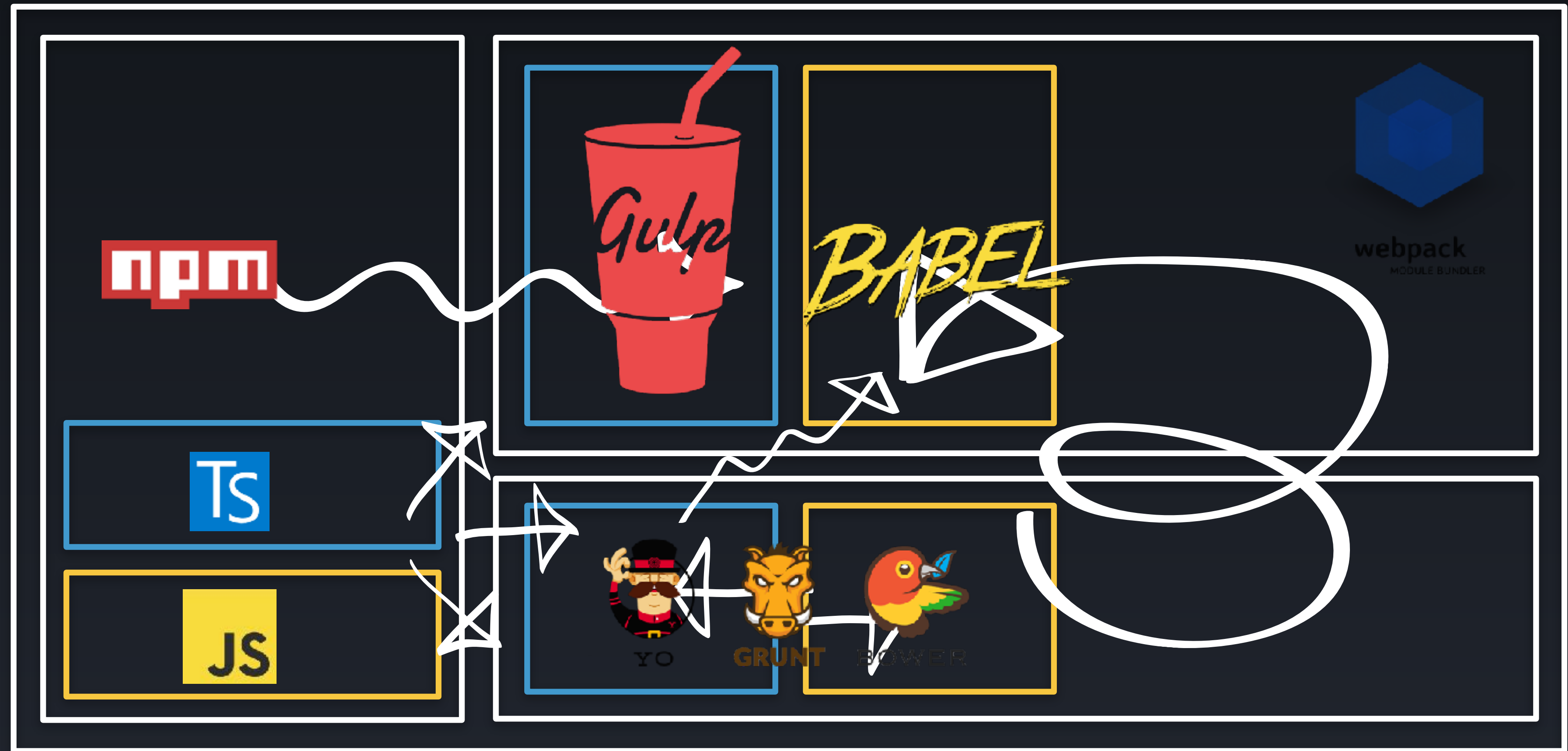
# Services

# Challenges

- Identify the major Angular 2 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
- **BONUS** Create an interface for the new property and type it in the **ItemsService** and your consuming component

# The Angular CLI















A terminal window with a dark background and a light gray title bar. The title bar contains three colored window control buttons (red, yellow, green) on the left. The terminal displays four lines of text, each starting with a green arrow pointing right, followed by a blue tilde (~) and a command. The commands are: `npm install -g angular-cli`, `ng new my-dream-app`, `cd my-dream-app`, and `ng serve`.

```
→ ~ 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 component my-new-component # using the alias
```

**Generating a component**

```
ng generate service my-new-service  
ng g service 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

- Scaffold out a **gizmo component**
- Scaffold out a **gizmo service**
- Run the tests
- Build the application
- **BONUS: Create a gizmo route**

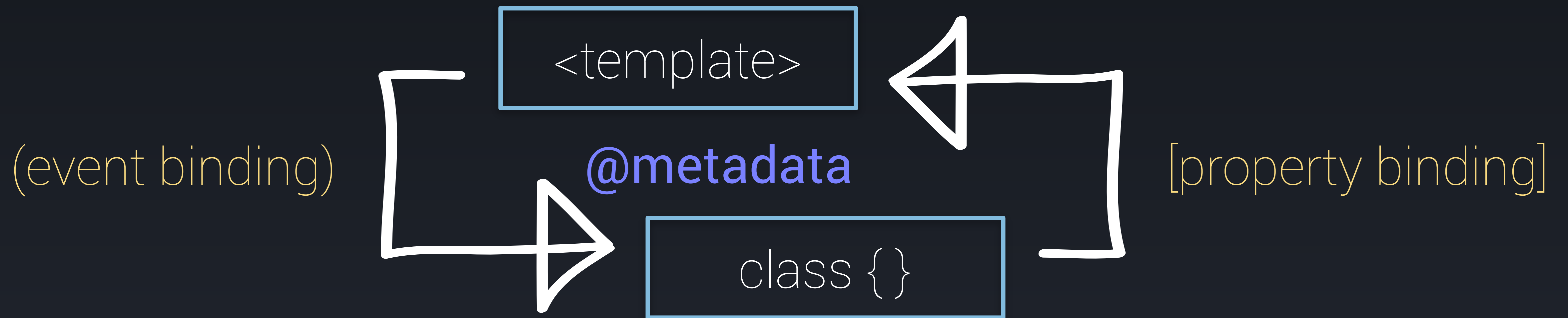
**NOTE: Use the Angular CLI for all of the tasks above**

# Component Fundamentals

# Component Fundamentals

- Anatomy of a Component
- **Class Import Decorate Enhance Repeat**
- Enhance with properties and methods
- Enhance with injectables
- Lifecycle Hooks

# Anatomy of a Component





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 2 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



```
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 **app** component
- **BONUS** Create a simple route to view the **widgets** component by itself

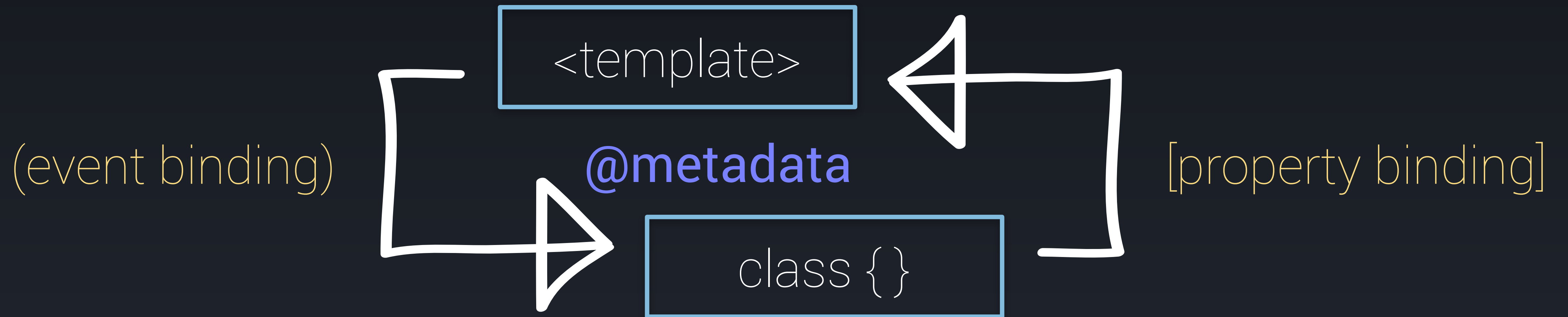
# Template Fundamentals



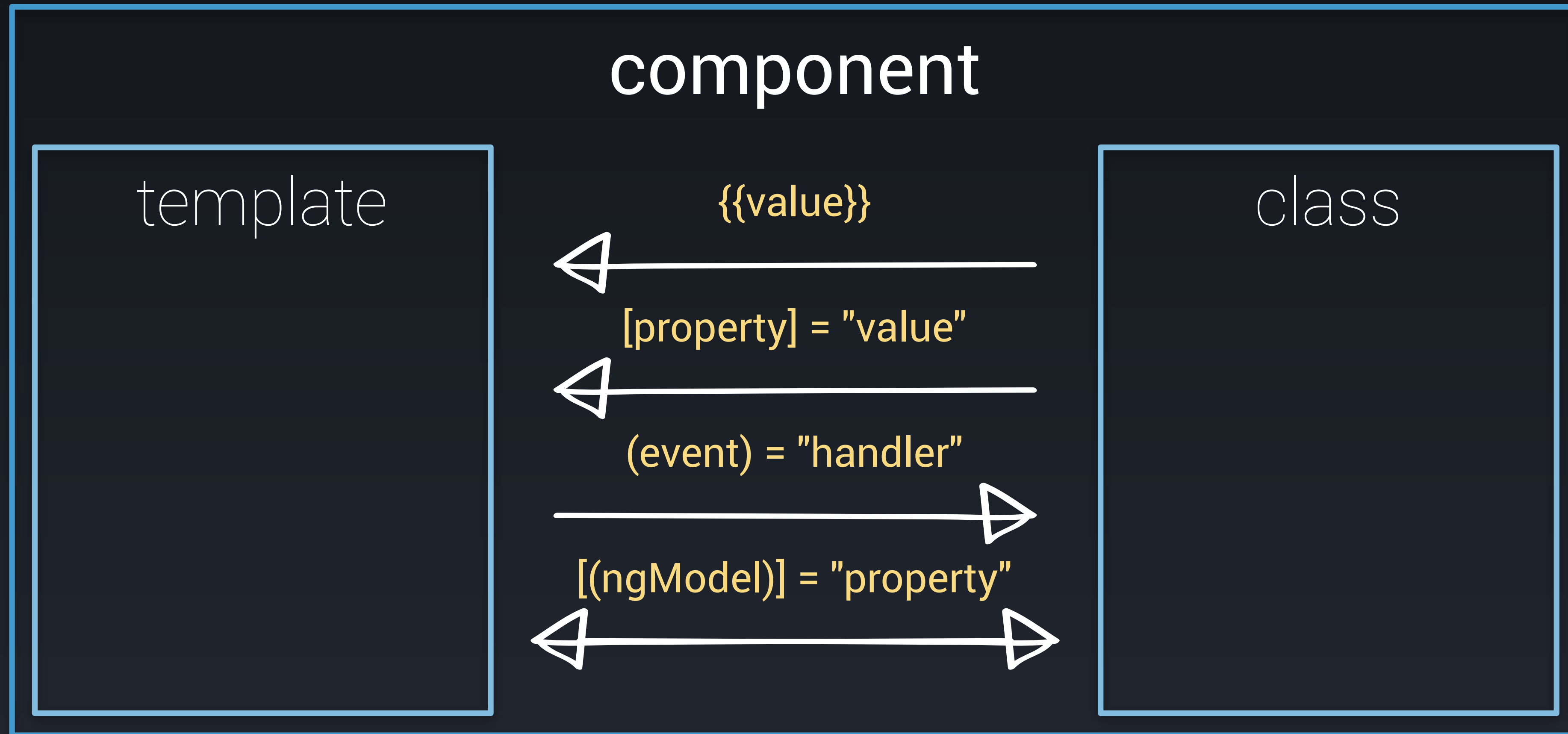
# Template Fundamentals

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

# Templates



# 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

```
<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



# Two-way Binding

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



```
<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

```
<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



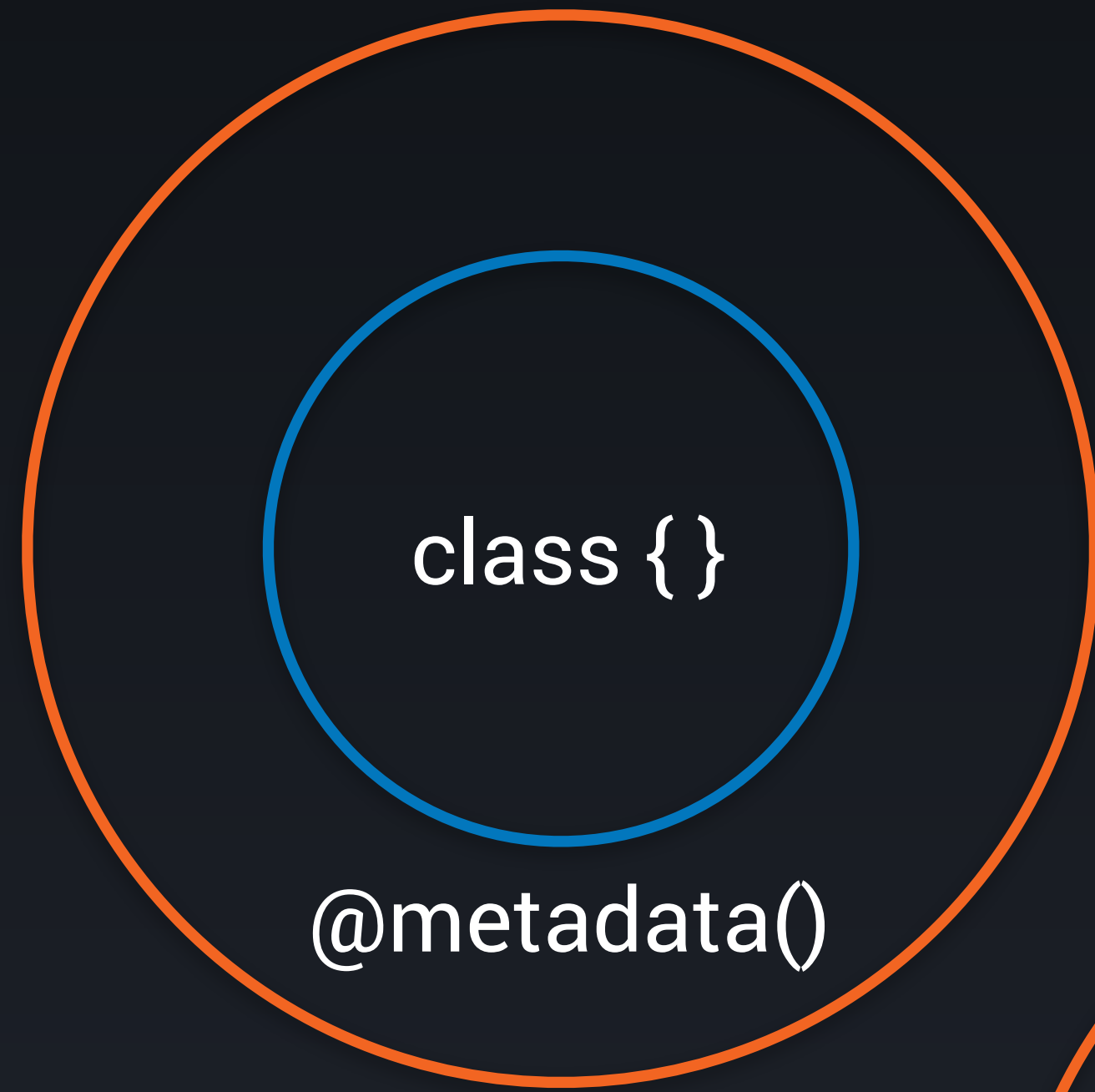
# Demonstration

# Challenges

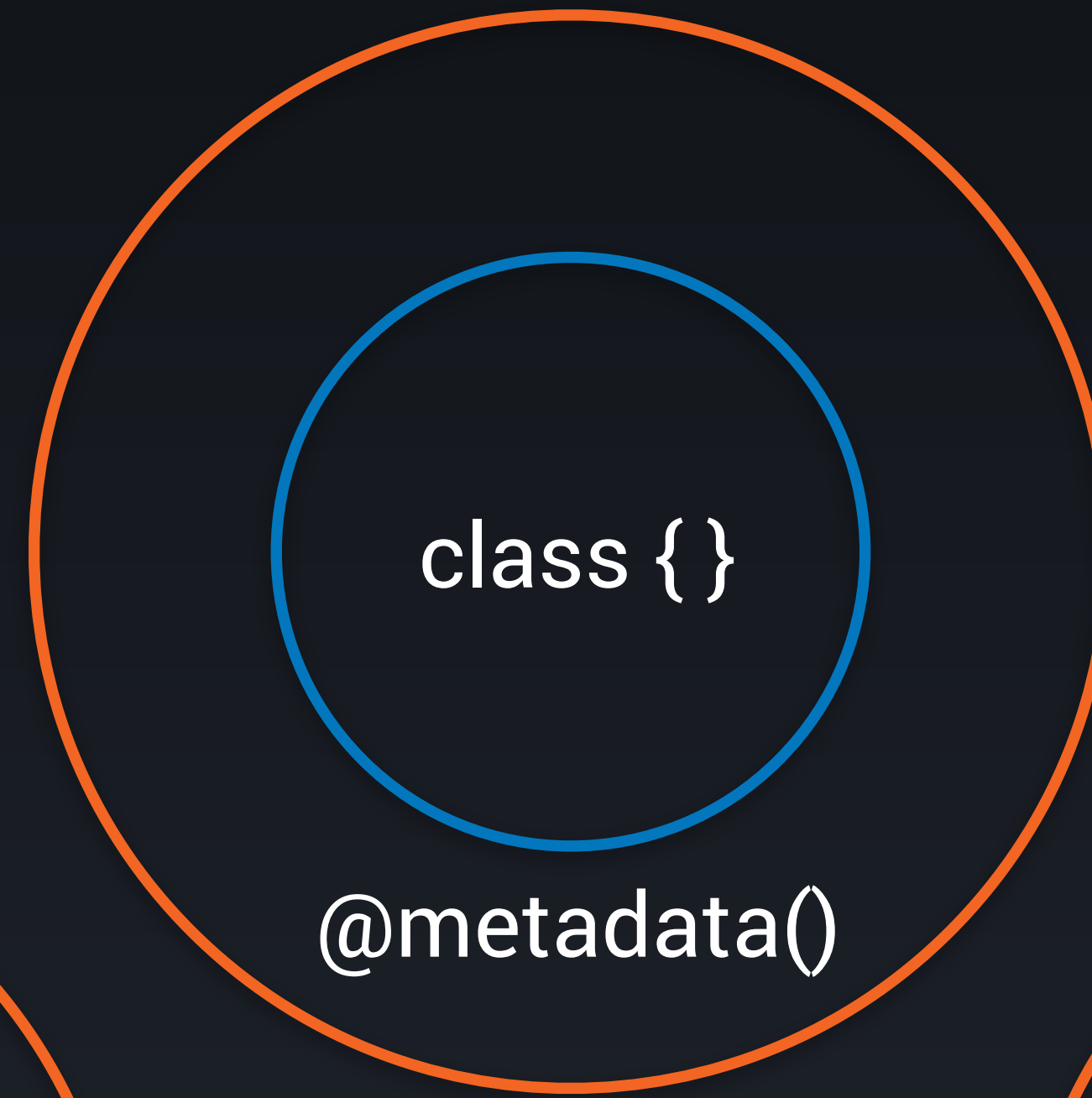
- Create a **widgets** collection in the **widget** component with mock objects
- Create a **selectedWidget** property in the widget component
- Display the **widgets** collection in the template using **ngFor**
- Use an **event binding** to set a selected widget
- Display the **widget** properties using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

**ACTION ITEM!** Go to <http://bit.ly/workshop-snippets> to save on typing

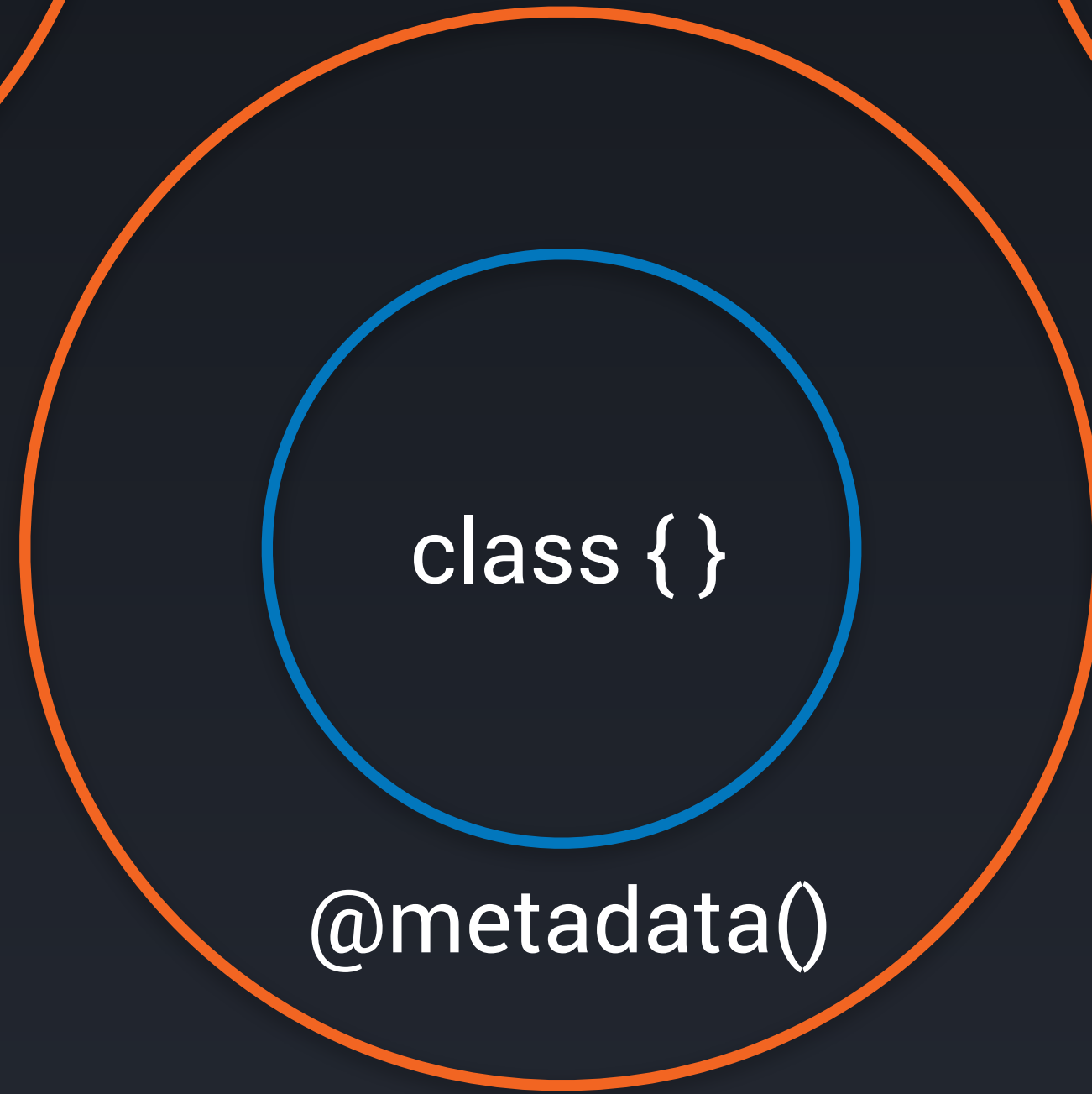
# Services



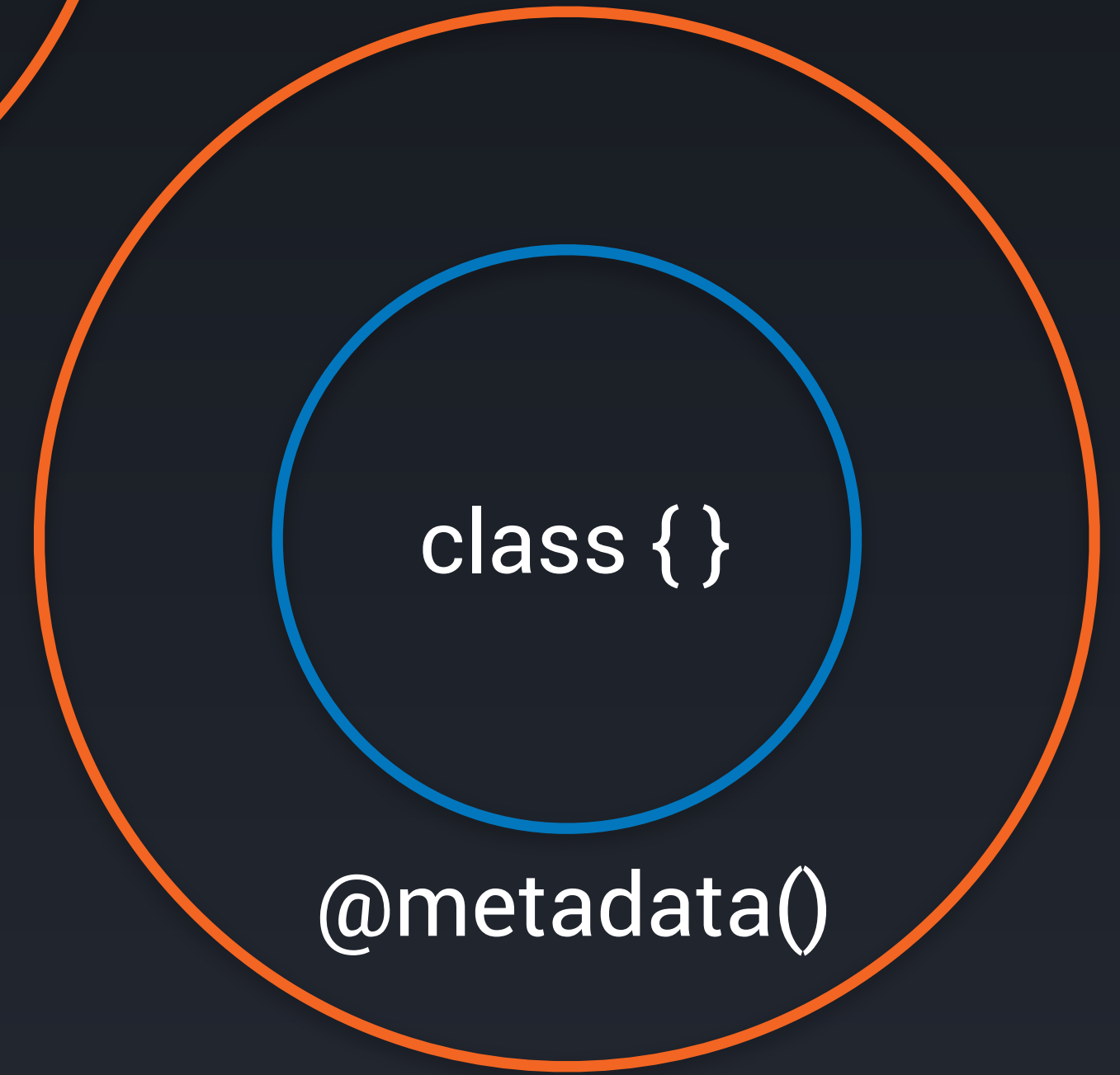
**component**



**service**



**directive**



**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,
    HttpClientModule,
    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**

# Routing

# Routing

- Defining Routes
- Navigating Routes
- Route Params
- Child Routes
- **Named Routes**
- **Lazy Loading Routes**

# Defining Routes

- Routes are defined as a collection of **Route** objects
- The simplest configuration defines a **path** and a **component**
- We then pass our **routes** collection into **RouterModule.forRoot** which returns a configured **Router** module

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

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

## Defining Routes

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

## Exposing Routes

```
<main>  
  <router-outlet></router-outlet>  
</main>
```

# Loading Routes



# Navigating Routes

- We can use the **routerLink** directive to navigate to routes within our template
- We can use **router.navigate** to navigate to routes programmatically

```
<header>
  <div>
    <span>{{title}}</span>
    <nav>
      <a [routerLink]="links.items">Items</a>
      <a [routerLink]="links.widgets">Widgets</a>
    </nav>
  </div>
</header>
```

# Navigating Routes

```
setItemAsFeatured(item: Item) {  
  this.unsetFeaturedItem();  
  
  this.saveItem(Object.assign({}, item, {featured: true}));  
  
  this.router.navigate(['featured', item.id], {relativeTo: this.route});  
}
```

## Navigating Routes Programmatically

# Route Params

- We use a colon to indicate a route parameter such as **/items/:id** which will resolve to something like **/items/4**
- We can then use the **ActivatedRoute** service to get information about the current route
- The **ActivatedRoute.params** returns an observable with the **required** and **optional** parameters for the route
- We can also use **ActivatedRoute.snapshot.params** if we only need the *initial* value of the parameter

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

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

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

## Route Params

```
diffFeaturedItems(items: Item[]) {  
  const supposedlyFeaturedID = this.route.snapshot.firstChild.params['id'];  
  
  if (supposedlyFeaturedID) {  
    let supposedlyFeaturedItem =  
      items.find(item => item.id === +supposedlyFeaturedID);  
  
    if (!supposedlyFeaturedItem.featured) {  
      this.setItemAsFeatured(supposedlyFeaturedItem);  
    }  
  }  
}
```

# Route Params

# Child Routes

- To define a child route, we add the **children** property to our **routes** collection
- Child routes will be loaded within the **router-outlet** of its parent component
- For relative navigation to a child route programmatically, use **relativeTo** within the **router.navigate** method call

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

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

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

## Child Routes



```
<div>
  <div>
    <router-outlet></router-outlet>
  </div>
  <div>
    <app-items-list [items]="items"
      (selected)="selectItem($event)"
      (deleted)="deleteItem($event)"
      (featured)="setItemAsFeatured($event)">
    </app-items-list>
  </div>
  <div>
    <app-item-detail (saved)="saveItem($event)"
      (cancelled)="resetItem($event)"
      [item]="selectedItem">
      Select an Item
    </app-item-detail>
  </div>
</div>
```

## Child Routes

# Demonstration

# Challenges

- Add a method to the **widgets service** to select a **widget** by **id**
- Define a **route** with an **id** parameter that we can use to pre-populate the **widgets component**
- Navigate to the parameterized widgets route using **routerLink**
- Navigate to parameterized widgets route using **router.navigate**

**BONUS!** View Encapsulation

Unfortunately the "C" in  
CSS is "cascade"

Ideally the "C" in CSS  
would be **"component"**

Shadow DOM fixes CSS and DOM. It introduces **scoped styles** to the web platform.

# Native View Encapsulation

Set with **ViewEncapsulation.Native**

Uses the browser's native shadow DOM

The component's styles are included within the shadow DOM

**YOLO!**



Summary									
Calculation of support for currently selected criteria									
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49: 50%						
		47: 0%	51: 50%			9.2: 0%		4.4: 50%	
8: 0%	13: 0%	48: 0%	52: 50%	9.1: 0%		9.3: 0%		4.4.4: 50%	
11: 0%	14: 0%	49: 0%	53: 100%	10: 75%	39: 50%	10: 25%	all: 0%	52: 50%	51: 50%
		50: 0%	54: 100%	TP: 25%	40: 100%				
		51: 0%	55: 100%		41: 100%				
		52: 0%	56: 100%						

About that shadow DOM...

# Emulated View Encapsulation

Set with **ViewEncapsulation.Emulated**

This is the default mode

Emulates shadow DOM by preprocessing and renaming CSS

**This is cash money!**

```
<!-- original dom -->
<hero-details>
  <h2>Mister Fantastic</h2>
  <hero-team>
    <h3>Team</h3>
  </hero-team>
</hero-detail>
```

```
<!-- rendered dom -->
<hero-details _nghost-pmm-5>
  <h2 _ngcontent-pmm-5>Mister Fantastic</h2>
  <hero-team _ngcontent-pmm-5 _nghost-pmm-6>
    <h3 _ngcontent-pmm-6>Team</h3>
  </hero-team>
</hero-detail>
```

```
<!-- rendered css -->
[_nghost-pmm-5] {
  display: block;
  border: 1px solid black;
}
h3[_ngcontent-pmm-6] {
  background-color: white;
  border: 1px solid #777;
}
```

# Emulated Shadow DOM

# No View Encapsulation

Set with **ViewEncapsulation.None**

This offers no view encapsulation

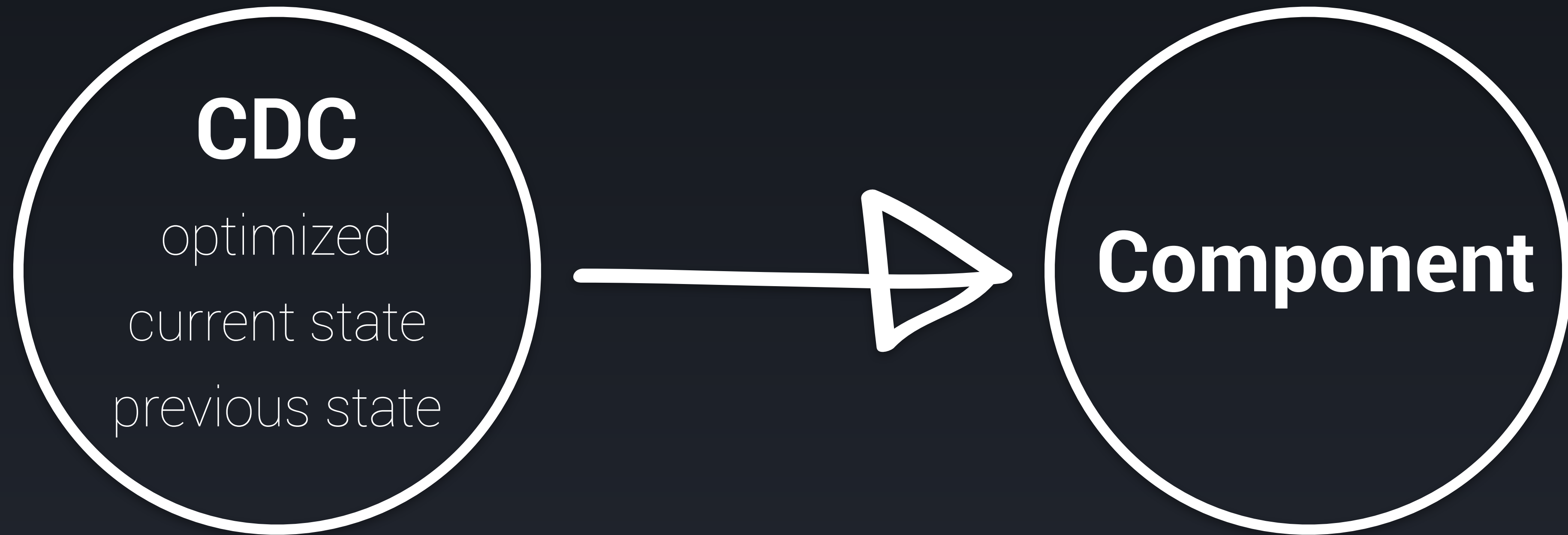
The equivalent of pasting your styles directly into the HTML

**Sad panda!**

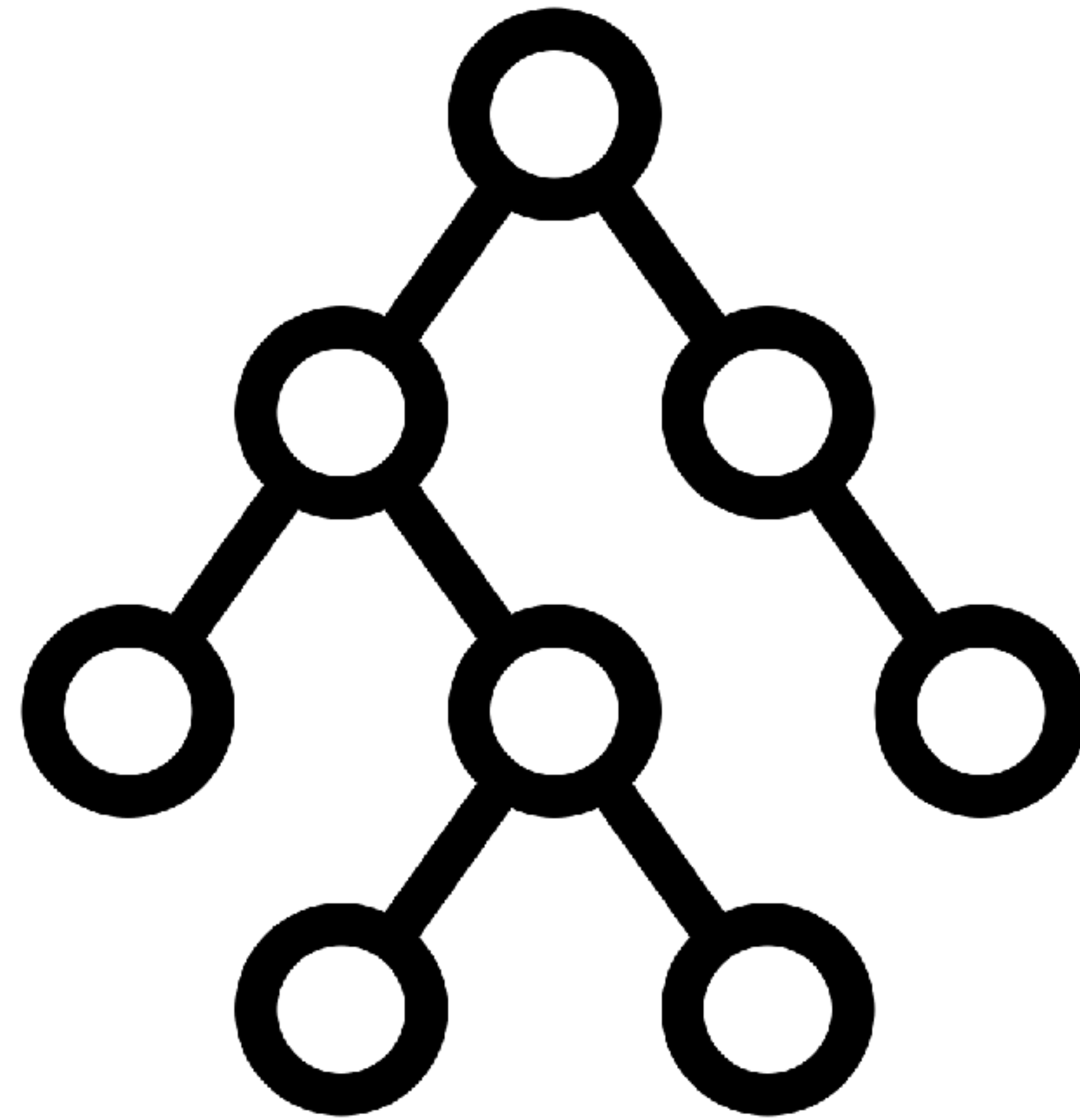
**BONUS!** Change Detection



Zone.js

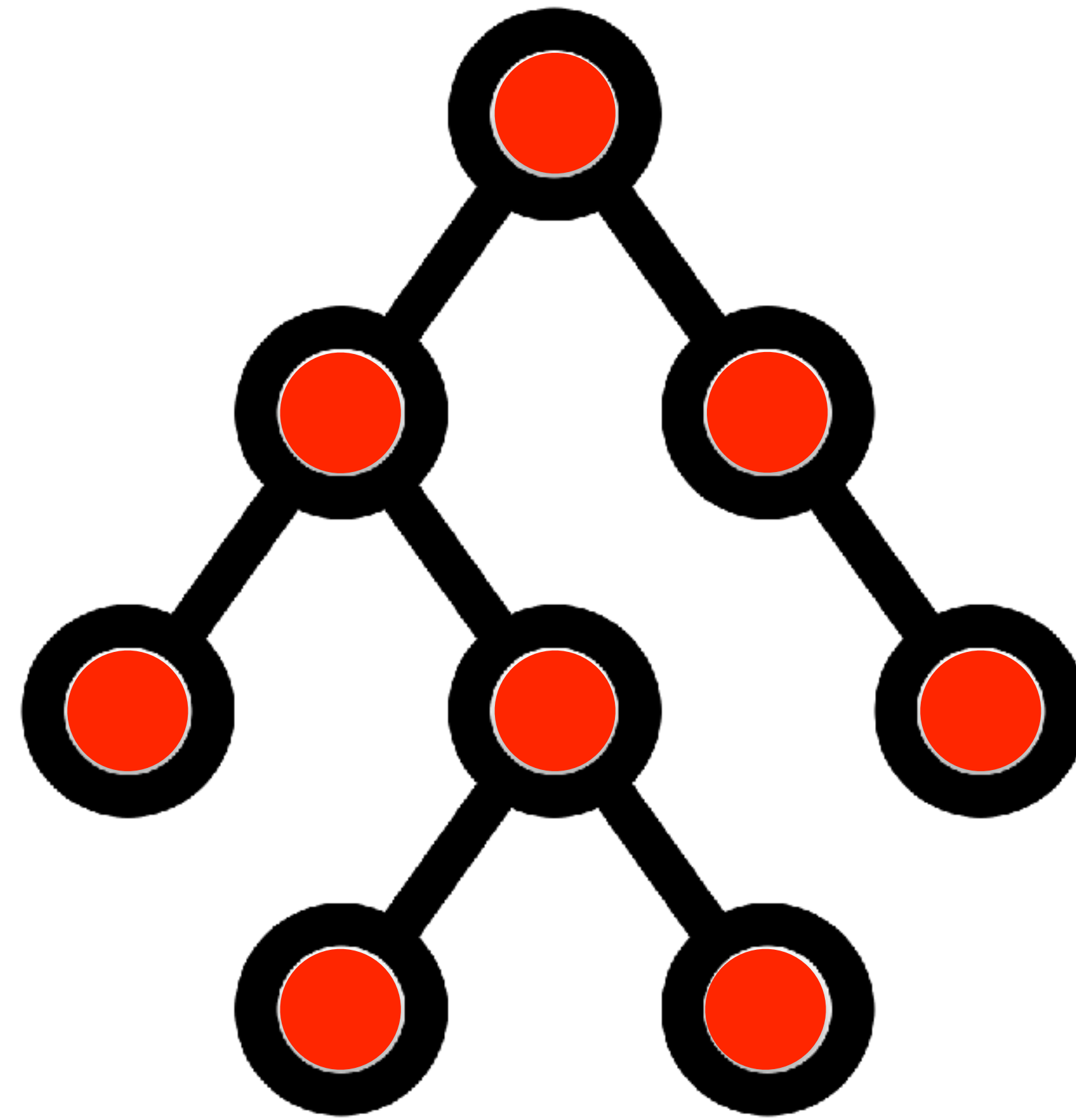


# Change Detection Classes

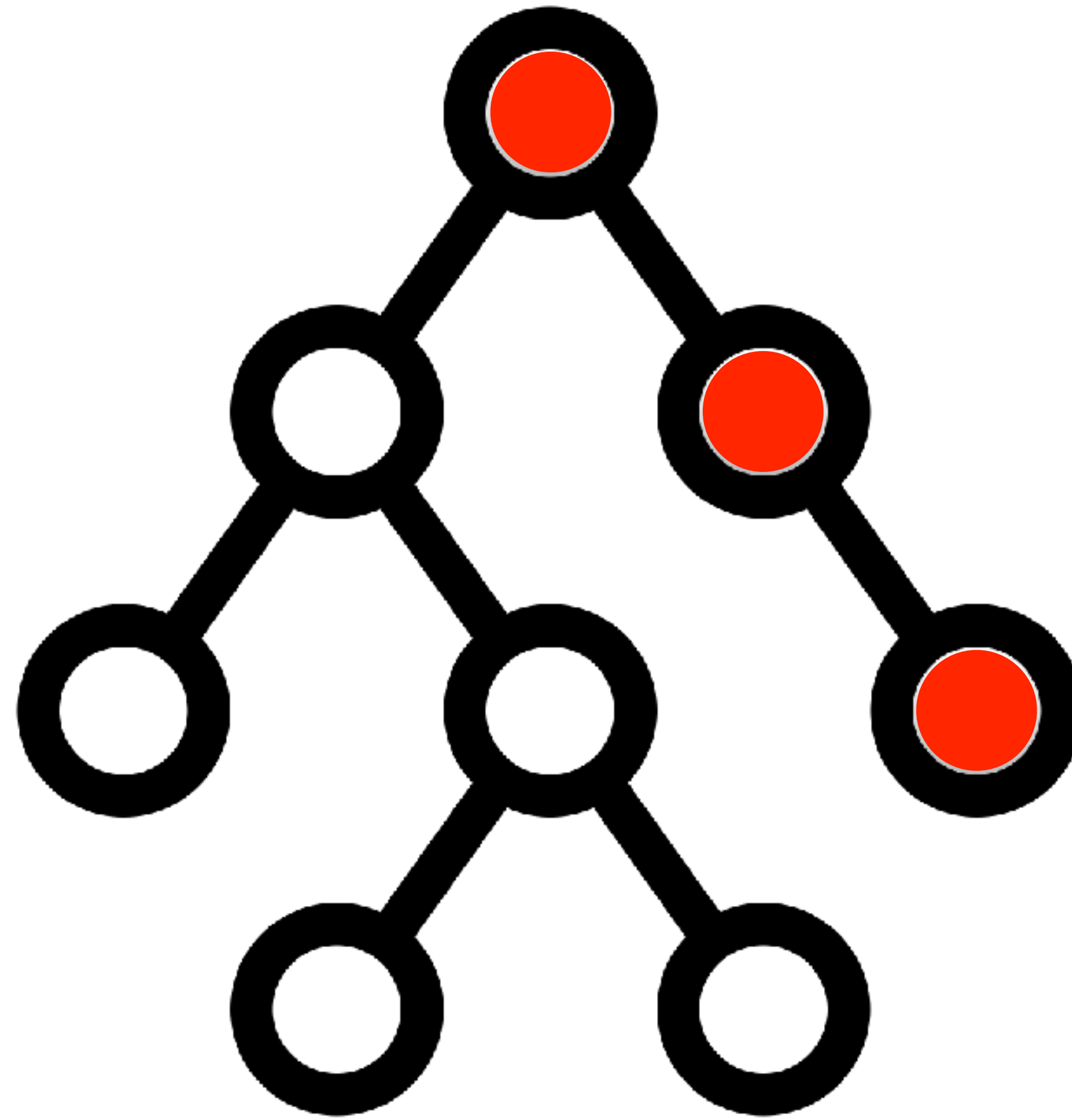


Detecting Change

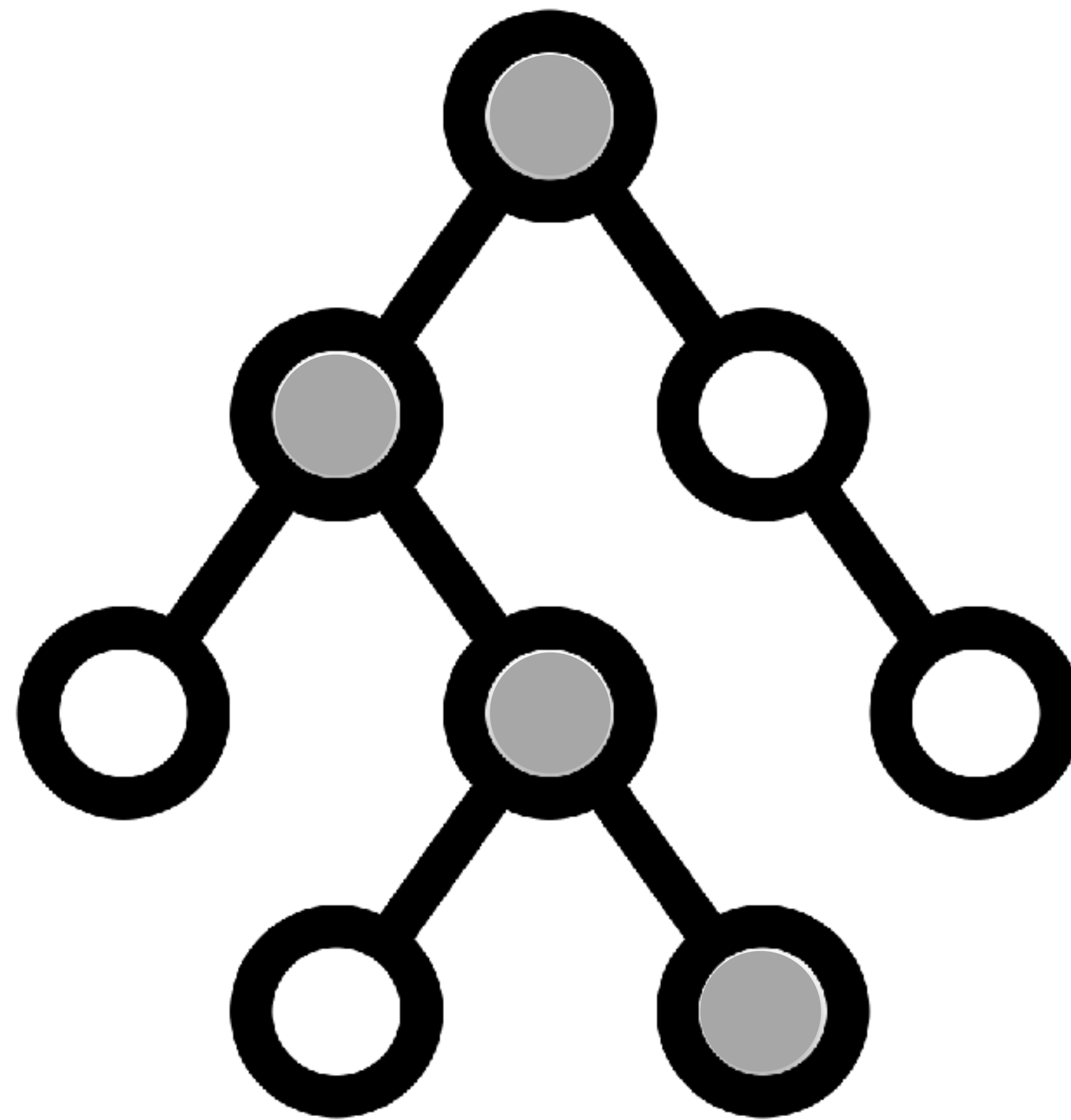




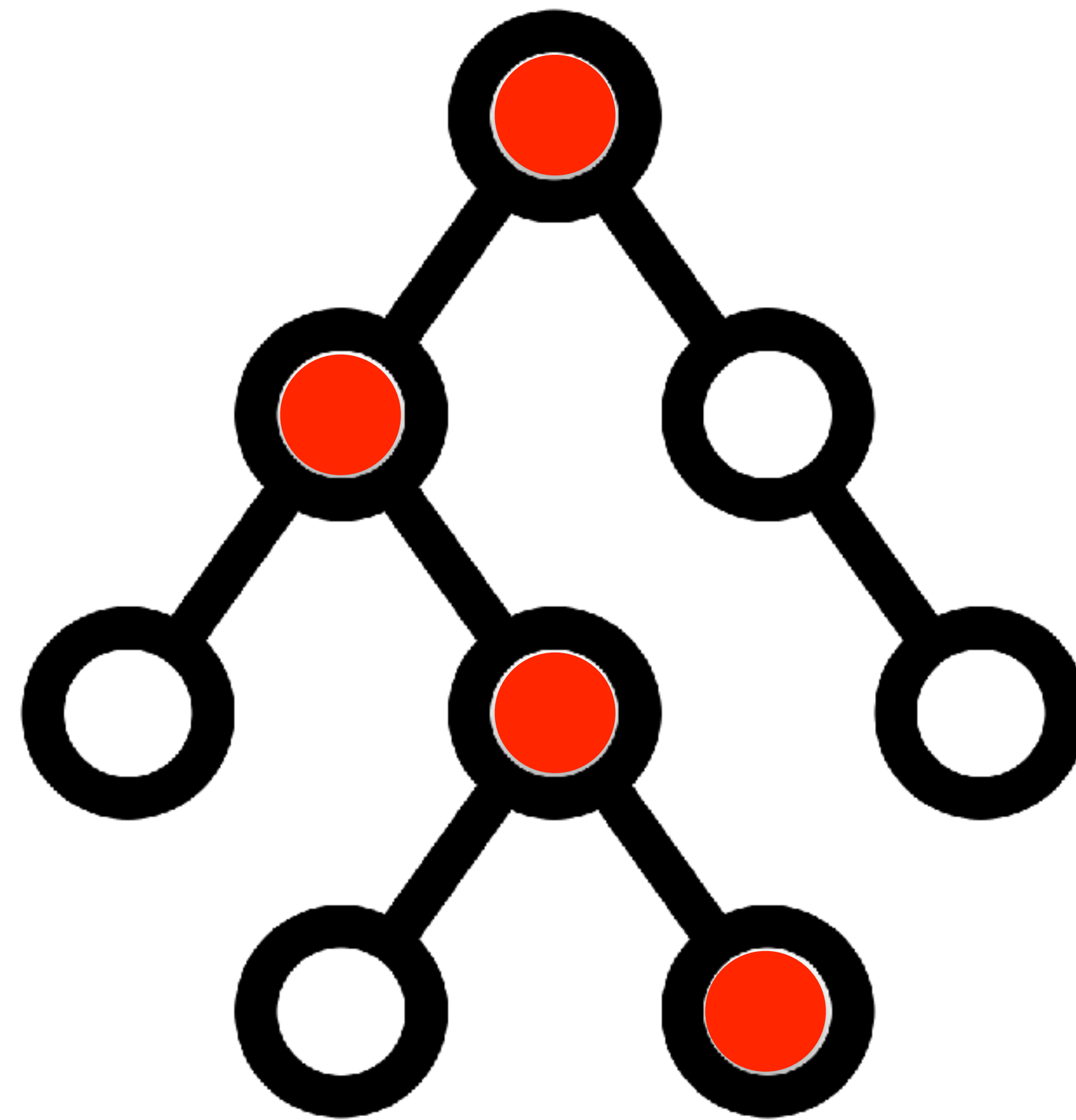
Default Change Detection



# OnPush Change Detection



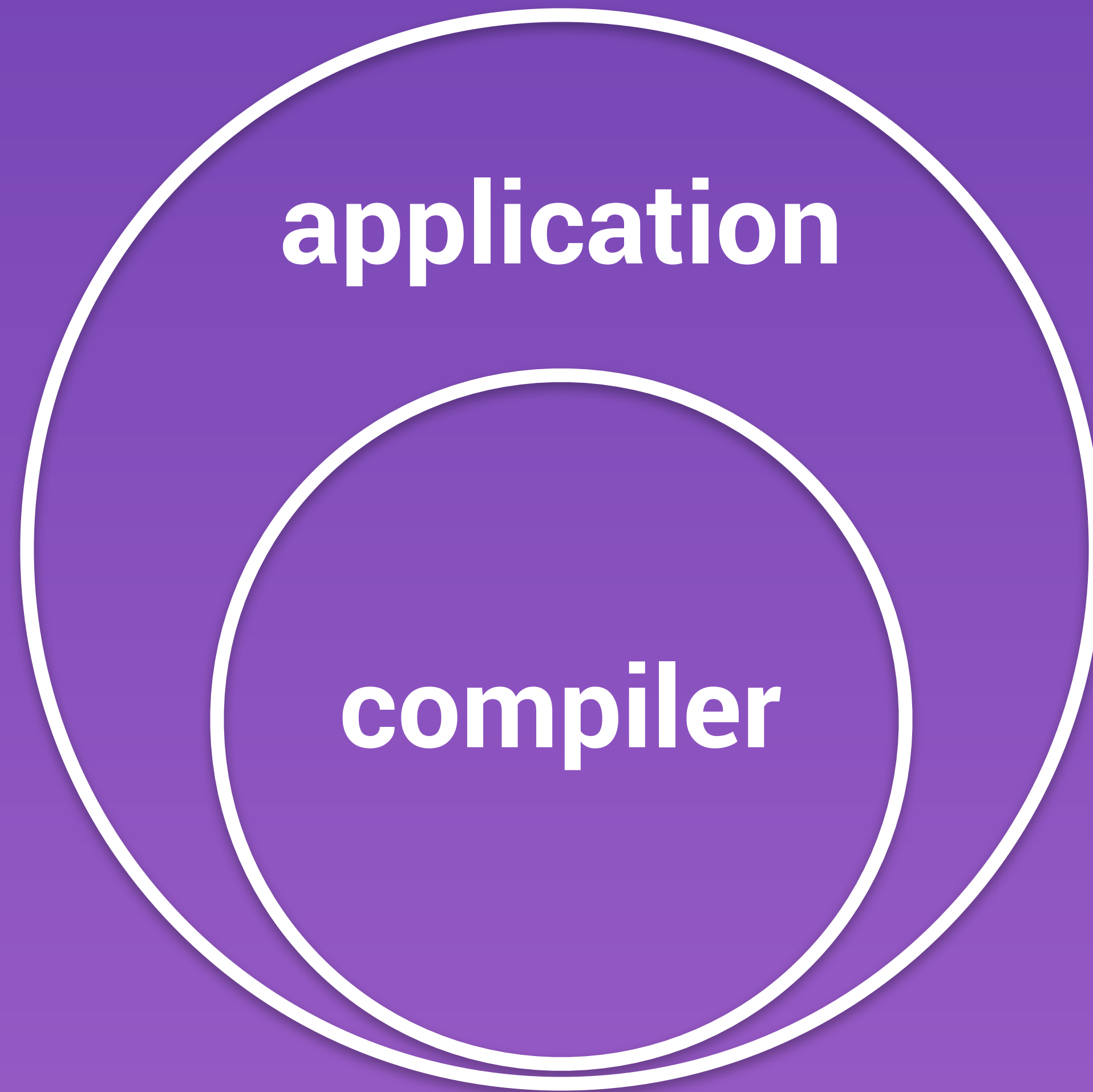
Observables



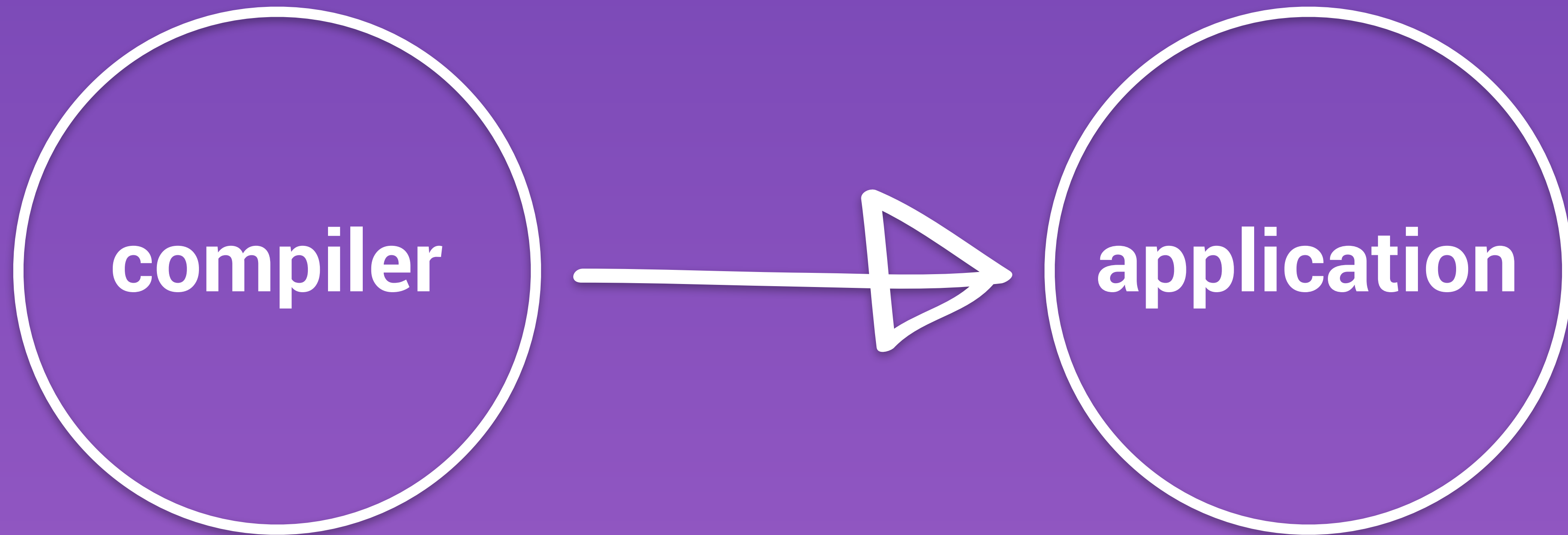
Observables

3-10x Faster

**BONUS!** AOT Compilation



JIT Compilation



AOT Compilation





smaller payload

fewer async requests

**faster rendering**

# Benefits of AOT

```
npm install @angular/compiler-cli @angular/platform-server --save
```

**Compile with AOT**

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "es2015",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  },

  "files": [
    "app/app.module.ts",
    "app/main.ts",
    "./typings/index.d.ts"
  ],

  "angularCompilerOptions": {
    "genDir": "aot",
    "skipMetadataEmit" : true
  }
}
```

# Compile with AOT

```
node_modules/.bin/ngc -p tsconfig-aot.json
```

**Compile with AOT**

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
import { AppModule } from './app.module';
```

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

// Becomes

```
import { platformBrowser } from '@angular/platform-browser';  
import { AppModuleNgFactory } from '../aot/app/app.module.ngfactory';
```

```
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

# Compile with AOT

```
import rollup      from 'rollup'
import nodeResolve from 'rollup-plugin-node-resolve'
import commonjs    from 'rollup-plugin-commonjs';
import uglify      from 'rollup-plugin-uglify'

export default {
  entry: 'app/main.js',
  dest: 'dist/build.js', // output a single application bundle
  sourceMap: false,
  format: 'iife',
  plugins: [
    nodeResolve({jsnext: true, module: true}),
    commonjs({
      include: 'node_modules/rxjs/**',
    }),
    uglify()
  ]
}
```

# Tree Shaking and Rollups

```
npm install rollup rollup-plugin-node-resolve rollup-plugin-commonjs rollup-plugin-uglify --save-dev
```

# Tree Shaking and Rollups

```
<body>  
  <my-app>Loading...</my-app>  
</body>  
  
<script src="dist/build.js"></script>
```

# Tree Shaking and Rollups







**@simpulton**



**Thanks!**