Angular 9

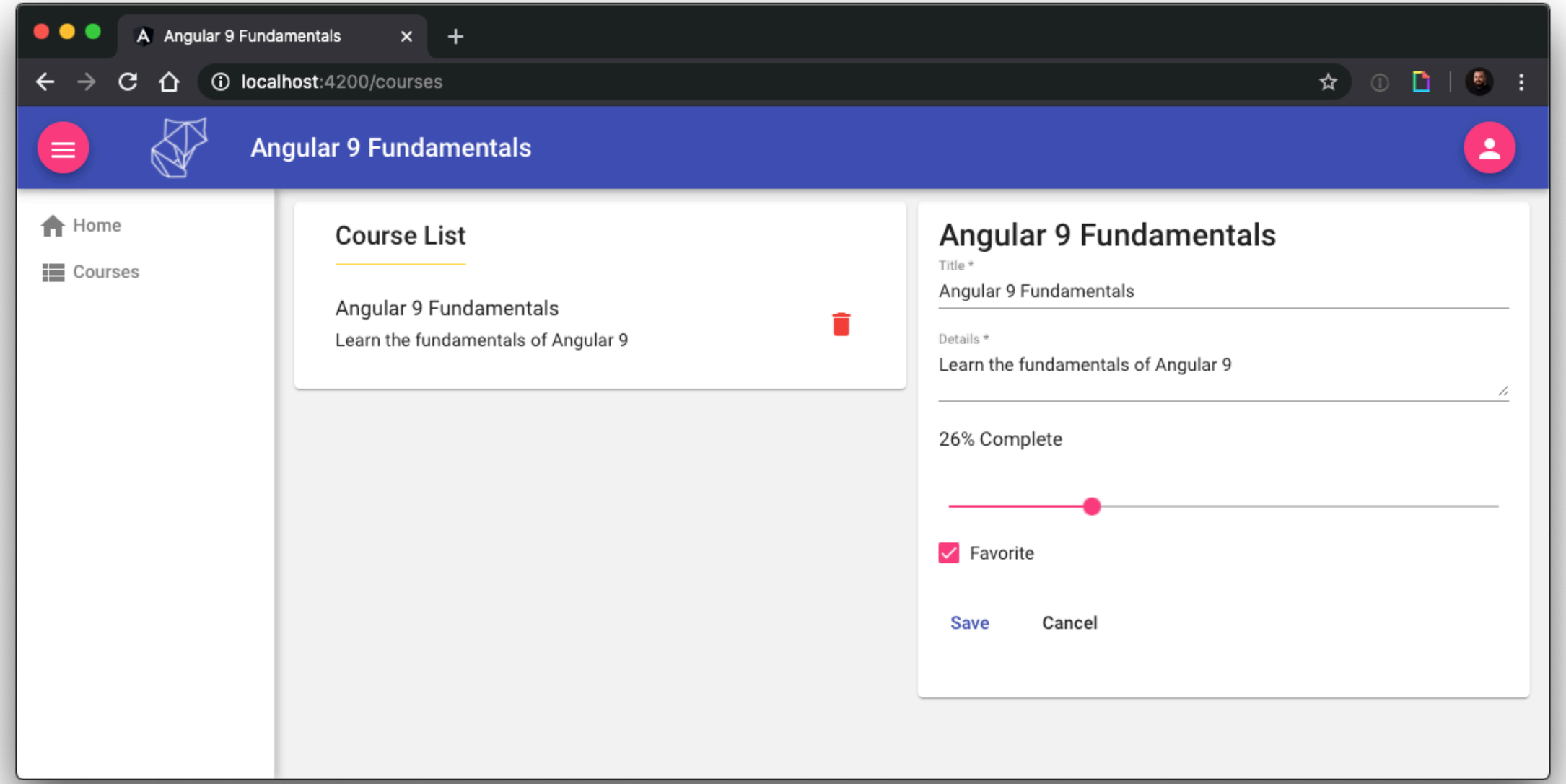
FUNDAMENTALS!

Agenda

- [ ] **Hello Angular**
- [ ] **Component Fundamentals**
- [ ] **Template Driven Forms**
- [ ] **Angular Services**
- [ ] **Server Communication**
- [ ] **Component Driven Architecture**
- [ ] **Angular Routing**
- [ ] **Unit Testing Fundamentals**

Getting **Started**

https://github.com/onehungrymind/angular9-fundamentals-workshop


Angular 9 Fundamentals
localhost:4200/courses
Angular 9 Fundamentals
Home
Courses
Course List
Angular 9 Fundamentals
Learn the fundamentals of Angular 9
Angular 9 Fundamentals
Title *
Angular 9 Fundamentals
Details *
Learn the fundamentals of Angular 9
26% Complete
Favorite
Save
Cancel

The Big Picture

aka How to impress your Angular friends at a dinner party

Why **Angular**?

Angular follows
common and **familiar**
enterprise patterns and
conventions

Angular is a **“batteries included”** framework

Angular ships with **tooling** to accelerate the **developer workflow**

Angular has a rich and vibrant **ecosystem**

Angular has a **proven** track record

The Angular 1.x Big Picture

module

config

routes

controller

$scope

view

service

directive

The **Simplified** 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
- Use ES6 module syntax
- Modules export things that other modules can import

```
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 { }
```

ES6 Modules

```
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 { }
```

ES6 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
- **exports** define a list of modules the module makes available
- **bootstrap** defines the component that should be bootstrapped

```
@NgModule({
  declarations: [
    AppComponent,
    ItemsComponent,
    ItemsListComponent,
    ItemDetailComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    AppRoutingModule
  ],
  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** directive
- 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

```ts
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 AppRoutingModule {
}
```

Routing

Components

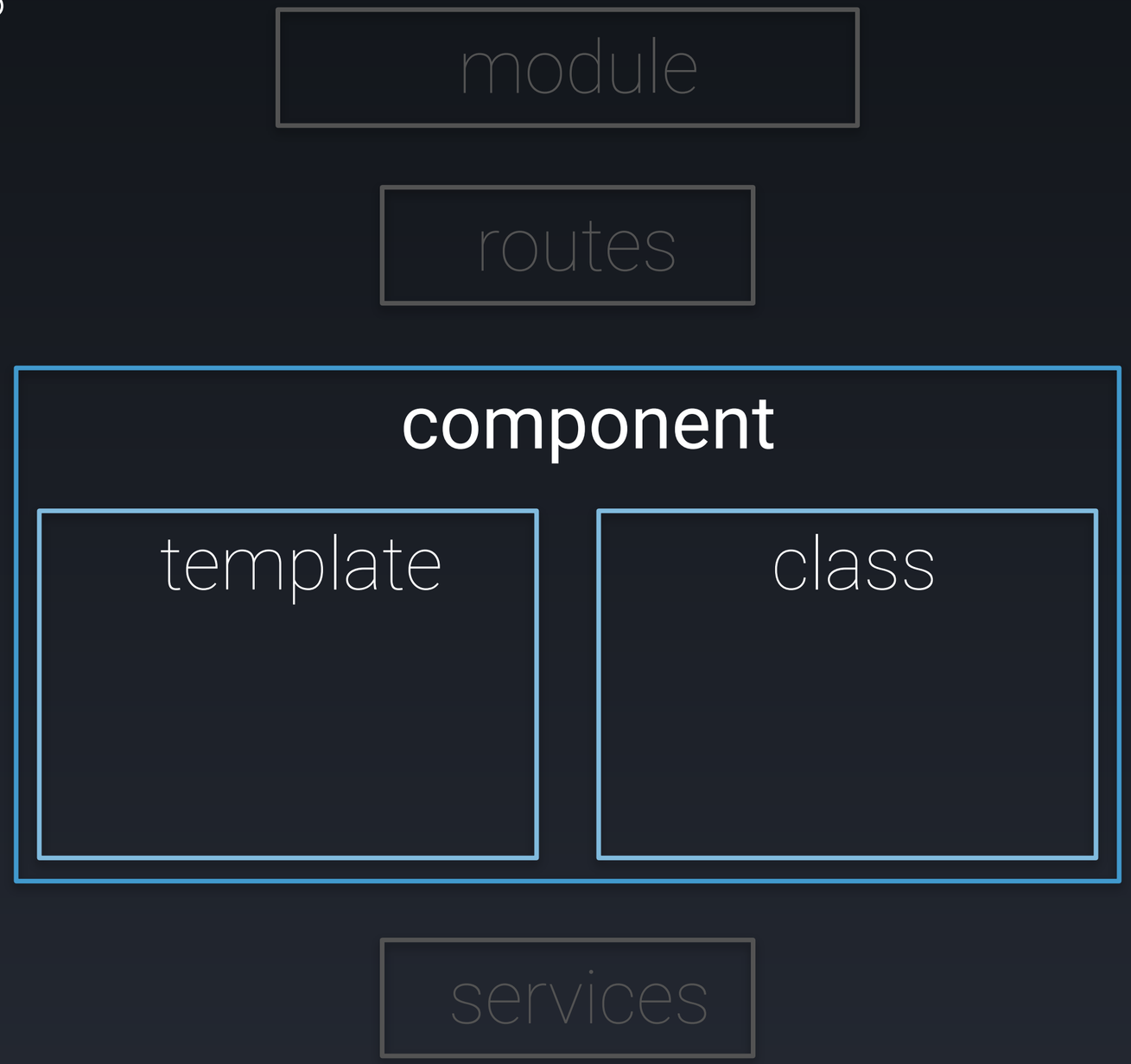
module

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: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.getItems();
  }

  getItems() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => 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

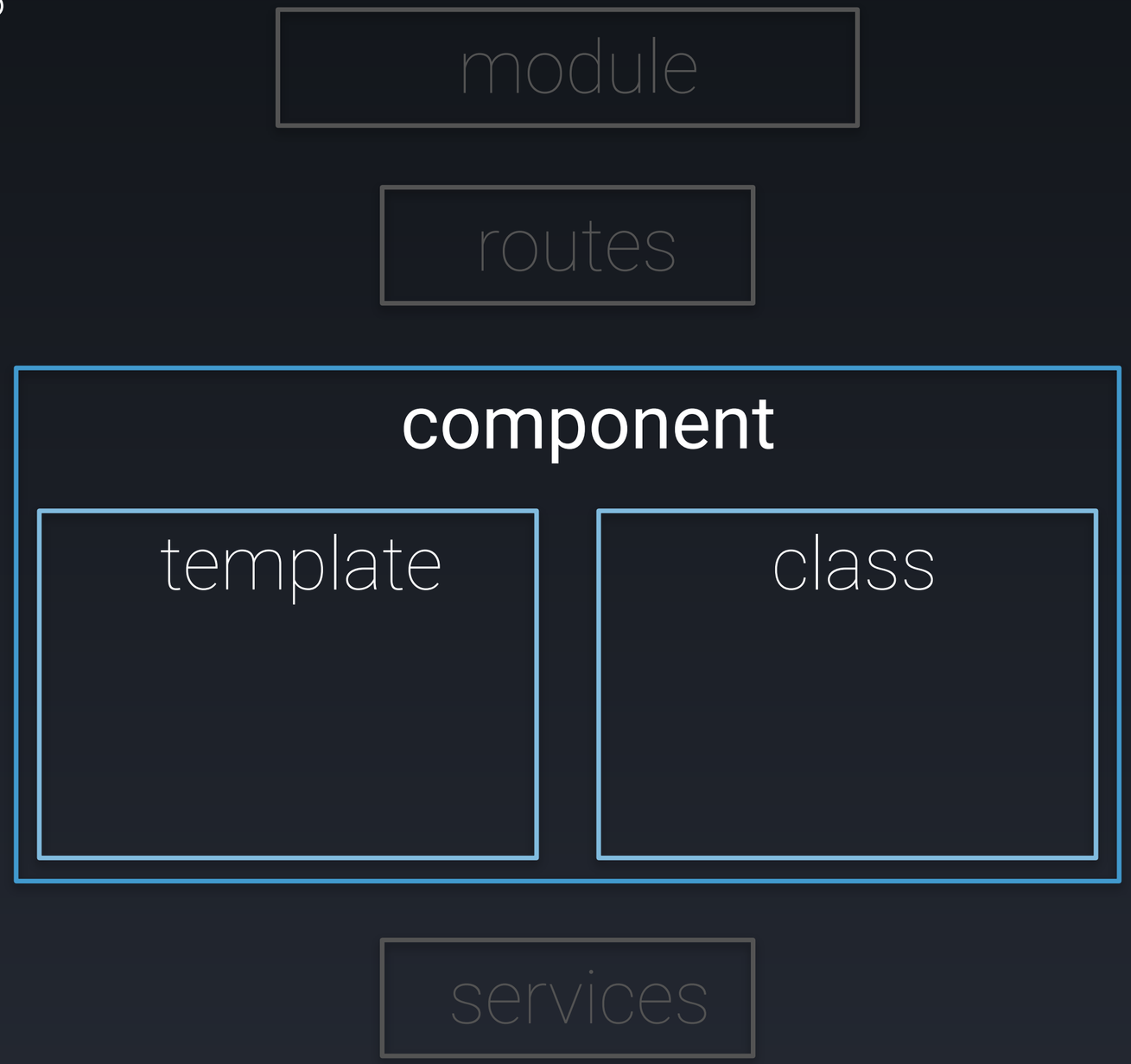
```
@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

External Template

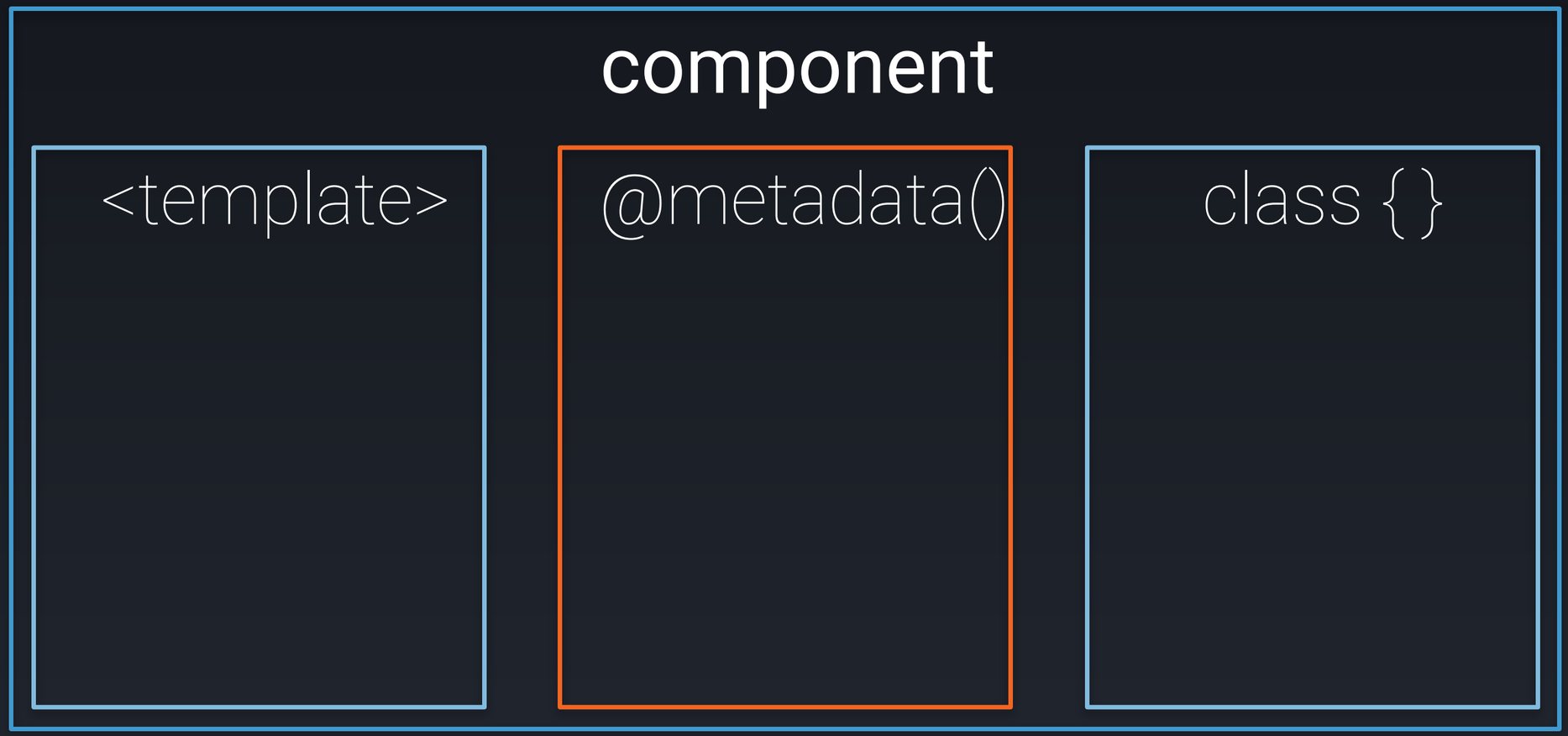
```
@Component({
  selector: 'app-items-list',
  template: `
  <div *ngFor="let item of items" (click)="selected.emit(item)">
    <div>
      <div><h2>{{item.name}}</h2></div>
      <div>{{item.description}}</div>
    </div>
  </div>
  `,
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent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

Inline 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

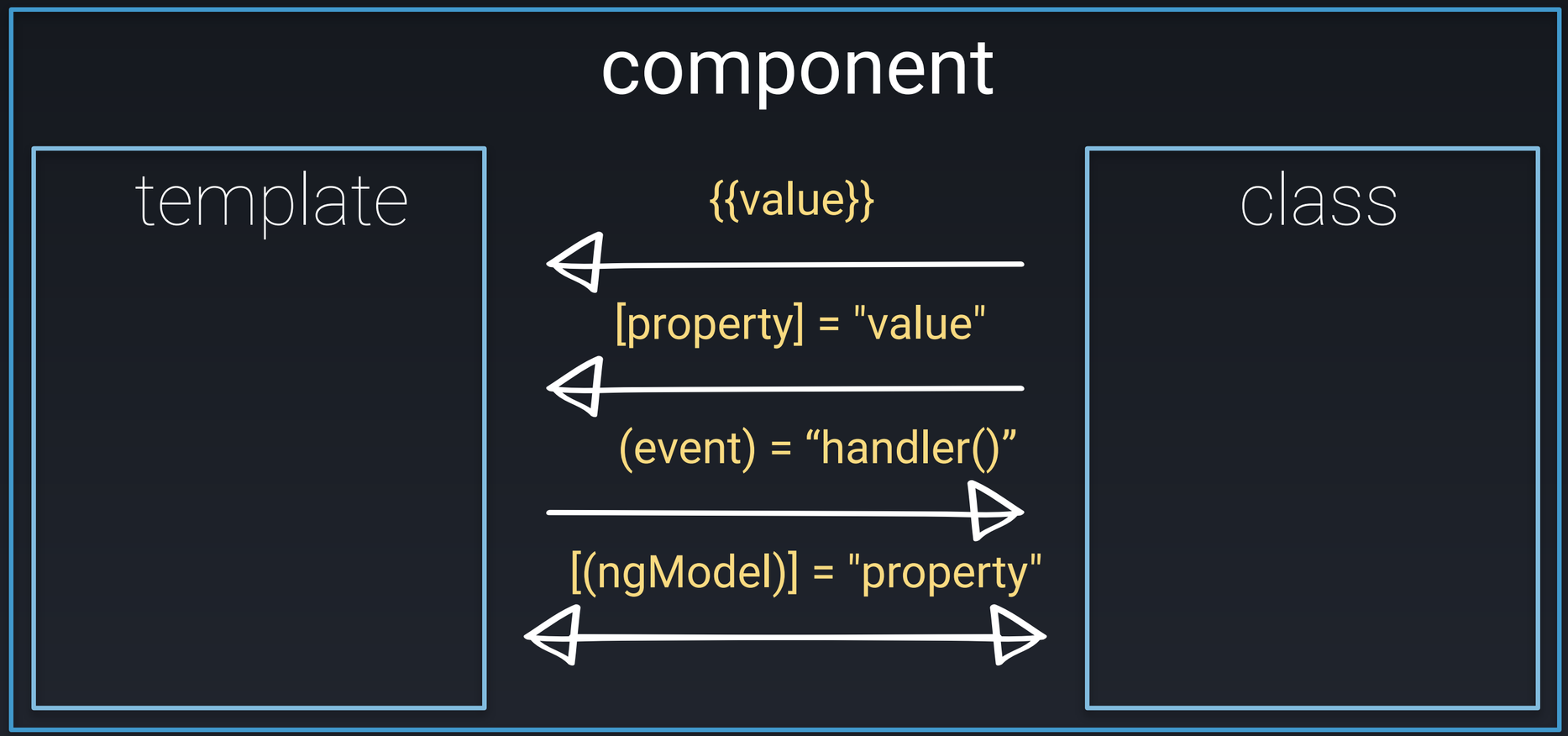
```
@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.css']
})
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Inline Metadata

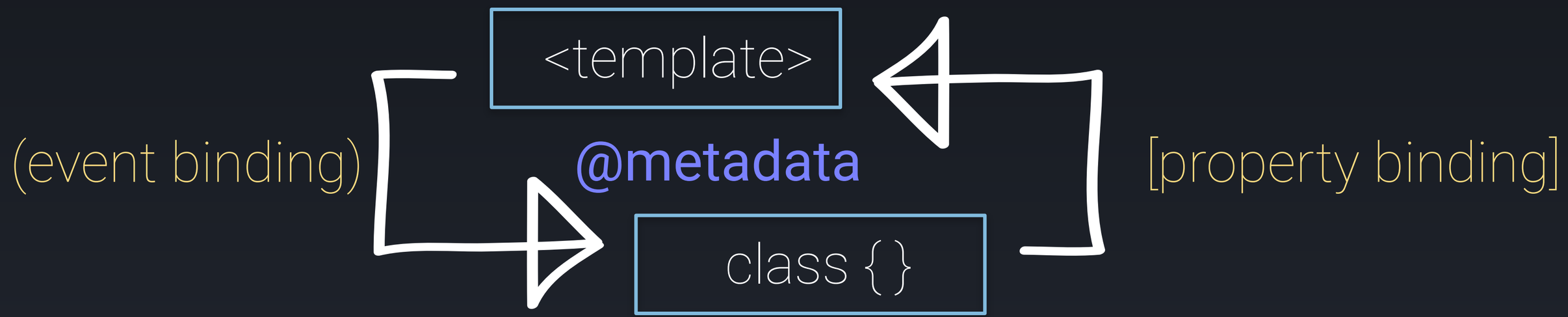
Data Binding

- 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 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" (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

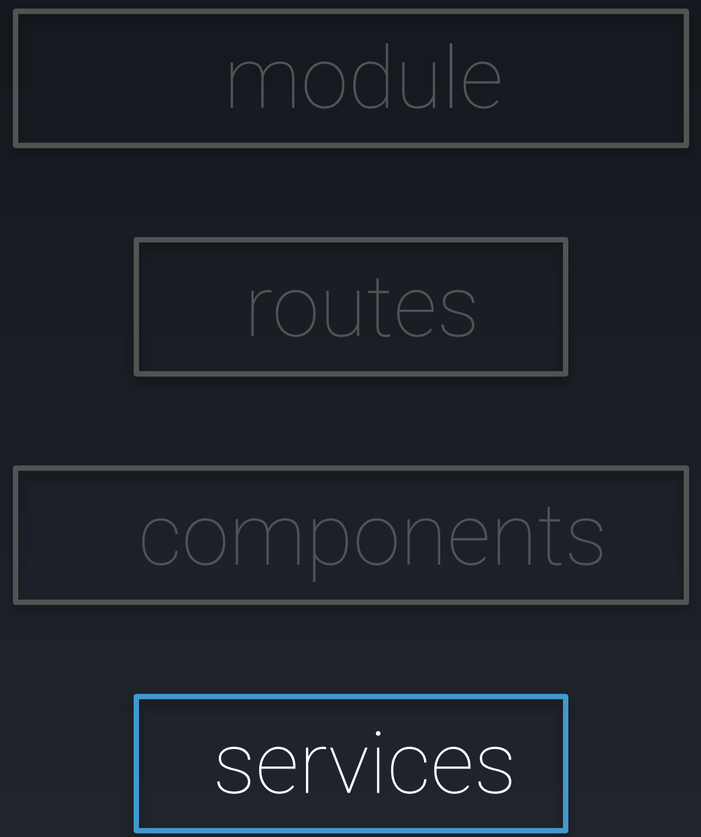
```ts
import { Directive, ElementRef } from '@angular/core';

@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



Services

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

```ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Item } from './item.model';

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

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

  loadItems() {
    return this.http.get(BASE_URL);
  }
}
```

Services

BONUS! TypeScript Time!

```ts
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Basic Component

```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Strong Types

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

Interface

```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Field Assignment

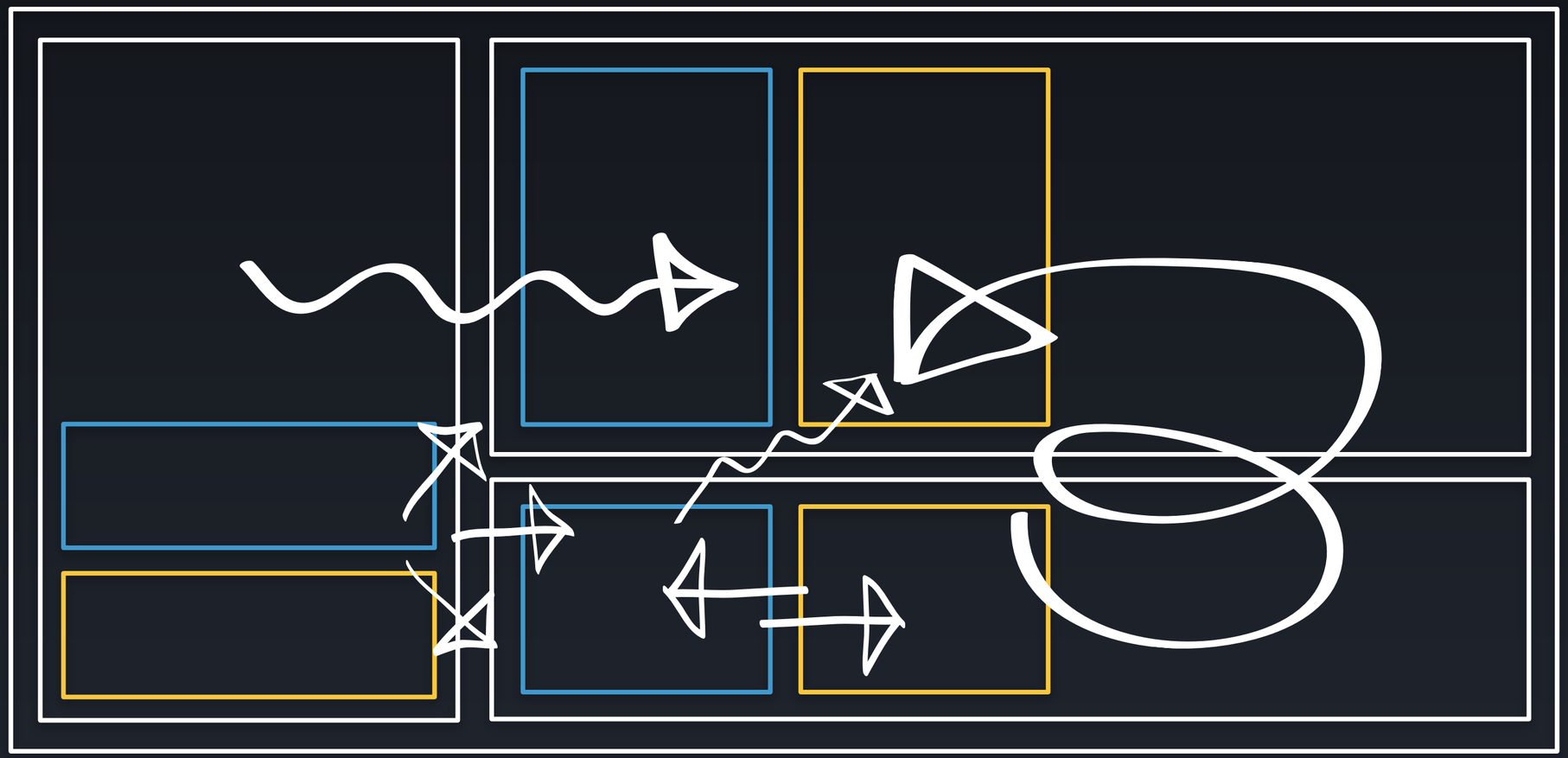
```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

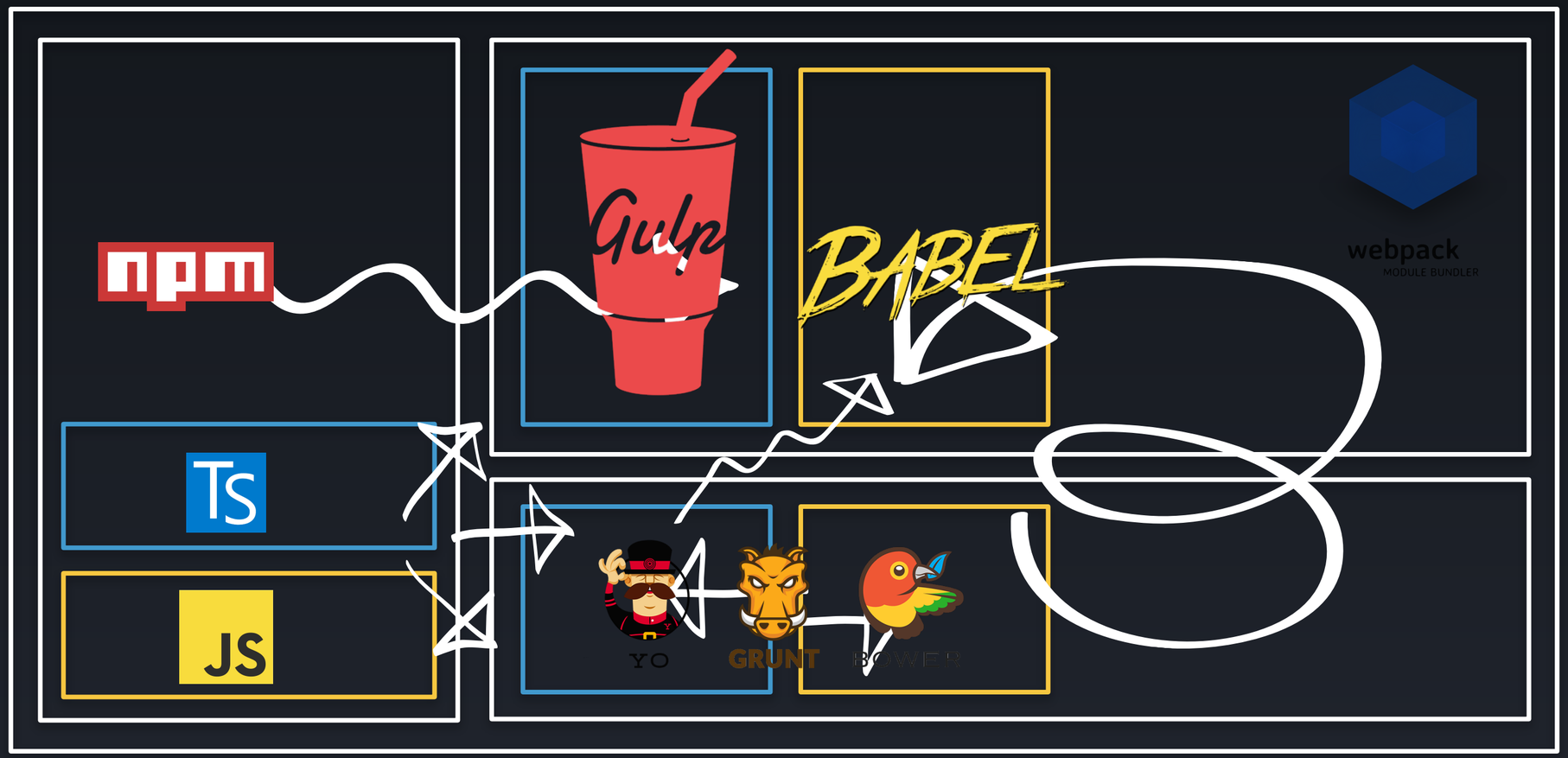
Constructor Assignment

```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Implements Interface

The Angular CLI




npm
Gulp
BABEL
webpack
MODULE BUNDLER
TS
JS
YO
GRUNT
BOWER


NGULAR 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 to GitHub pages
- Linting
- CSS preprocessor support
- AOT support
- Lazy routes
- **Fully extensible schematics**

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

Installing the CLI

```
ng new my-project
cd my-project
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
```

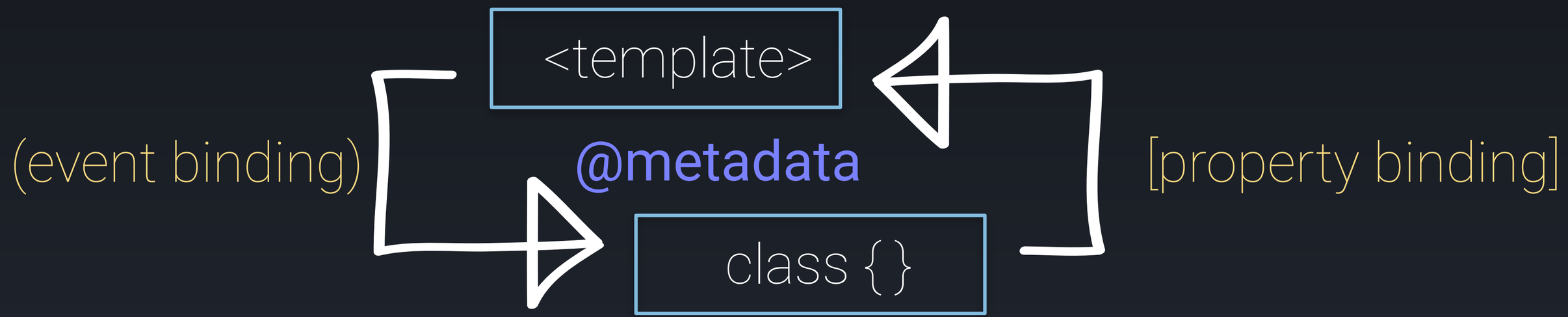
Running tests

```
ng lint
```

Linting

Component Fundamentals

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 can use by decorating it with 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

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

Exposing a Component

```
export class ItemsComponent {
  items: Item[];
  selectedItem: Item;

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

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

Properties and Methods

```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => 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

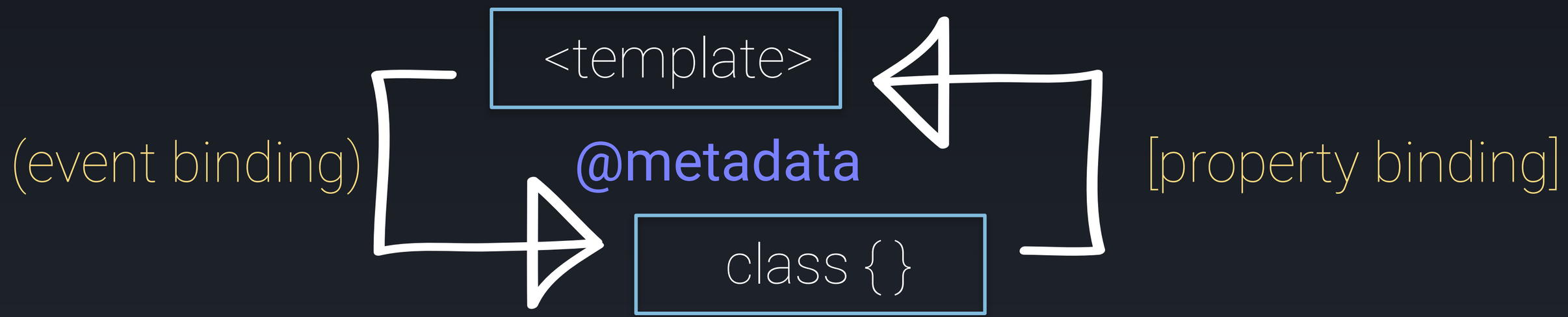
- **ngOnChanges** called when an input or output binding value changes
- **ngOnInit** called after the first ngOnChanges
- **ngDoCheck** handles developer's custom change detection
- **ngAfterContentInit** called after component content initialized
- **ngAfterContentChecked** called after every check 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.

```ts
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

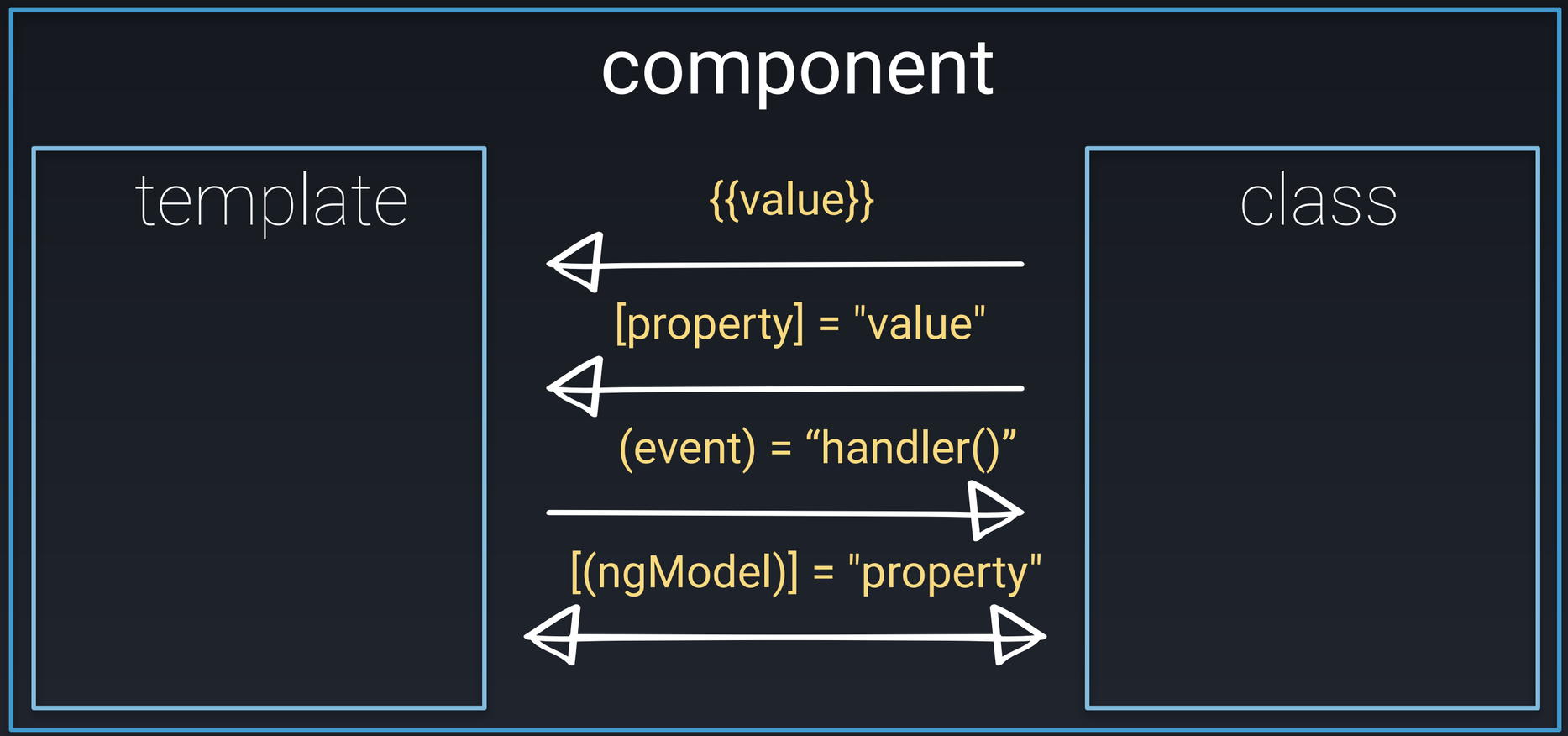
Lifecycle Hooks

Template Fundamentals

Templates



Data Binding



Property Binding

- Flows data from the component to an element
- Created with brackets ****
- 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($event)”></button>**
- Information about the target event is carried in the **$event** parameter

```
<button type="button" (click)="saved.emit(selectedItem)">Save</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"

```
<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

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

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**

```
<span *ngIf="currentPortfolio.id; else prompt">Editing {{originalName}}</span>
<ng-template #prompt>Create</ng-template>
```

Local Template Variable

```
<form #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

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! -->
<h2>The null hero's name is {{nullHero?.firstName}}</h2>
```

Safe Navigation Operator

Template Driven Forms

```
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 #nameRef="ngModel" [(ngModel)]="selectedItem.name"
  name="name" 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 #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

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

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

ngForm

```
<form #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

```
<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**

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

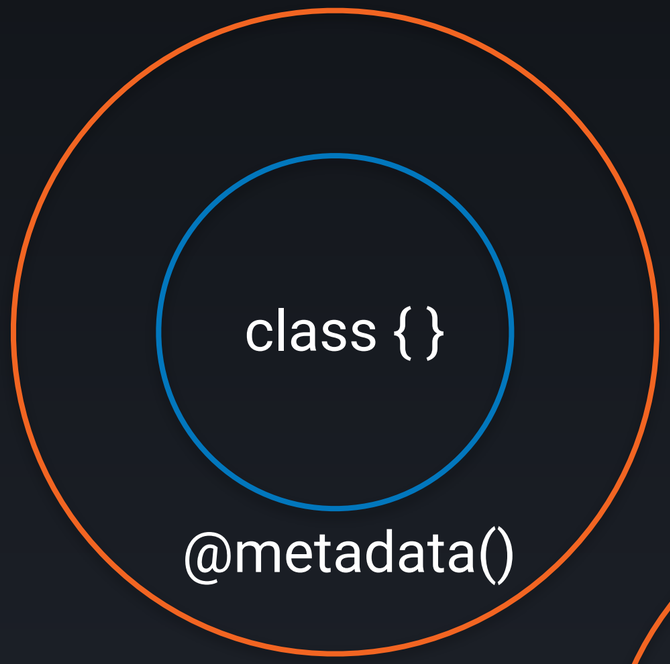
```
input.ng-valid {
  border-bottom: 1px solid green;
}
```

Validation Styles

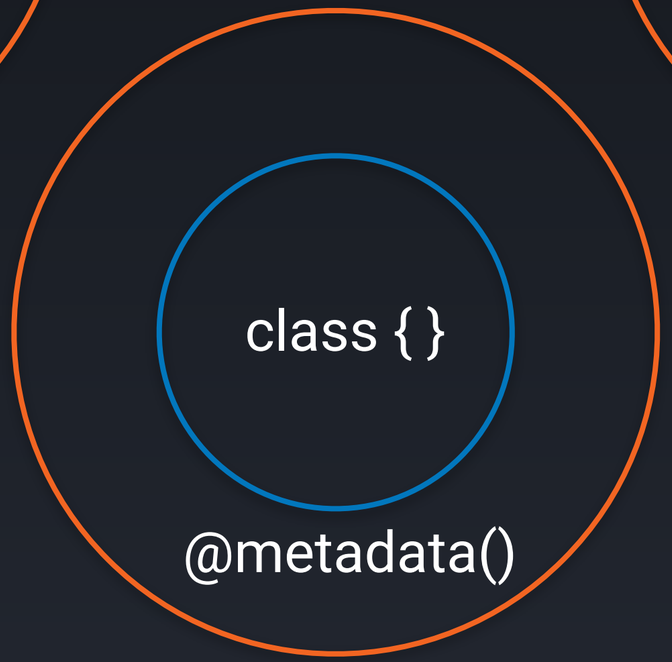
Angular **Services**

Everything is just a class

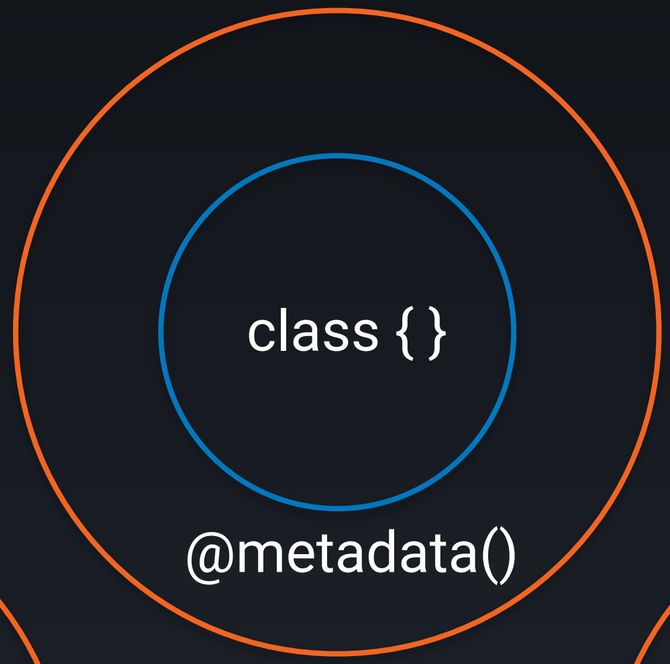
WAT.



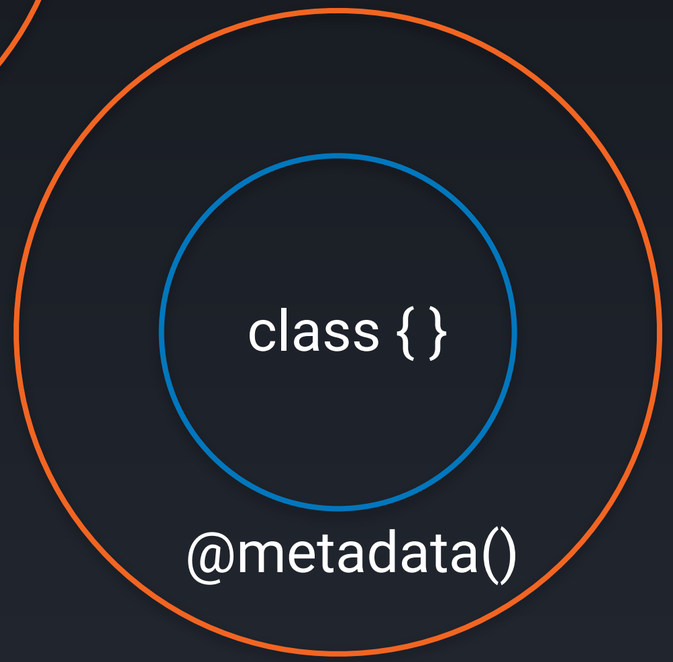
component



directive



service



pipe

Just a class!

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

  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,
    AppRoutingModule
  ],
  providers: [ItemsService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Exposing a Service

```
export class ItemsComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Consuming a Service

Server Communication

The HTTP Module

- Simplifies usage of the XHR and JSONP APIs
- API conveniently matches RESTful verbs
- Returns an observable

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

HttpClientModule

The HTTP Module Methods

- **request**: performs any type of http **request**
- **get**: performs a request with **GET** http method
- **post**: performs a request with **POST** http method
- **put**: performs a request with **PUT** http method
- **delete**: performs a request with **DELETE** http method
- **patch**: performs a request with **PATCH** http method
- **head**: performs a request with **HEAD** http method

```
loadItems() {
  return this.http.get(BASE_URL);
}

createItem(item: Item) {
  return this.http.post(`${BASE_URL}`, item);
}

updateItem(item: Item) {
  return this.http.patch(`${BASE_URL}${item.id}`, item);
}

deleteItem(item: Item) {
  return this.http.delete(`${BASE_URL}${item.id}`);
}
```

HTTP Methods

Observable.subscribe

- We finalize an observable stream by subscribing to it
- The **subscribe** method accepts three event handlers
- **onNext** is called when new data arrives
- **onError** is called when an error is thrown
- **onComplete** is called when the stream is completed

```
loadItems() {
  return this.http.get(BASE_URL);
}
```

http.get

```ts
export class ItemsComponent {
  items: Item[];
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  getItems() {
    this.itemsService.loadItems()
      .subscribe((items: Item[]) => this.items = items);
  }
}
```

Observable.subscribe

Headers

- HttpClient methods have an optional parameter which contains options for configuring the request
- This options object has a **headers** property which is an **HttpHeaders** object
- We can use the **HttpHeaders** object to set additional parameters like **Content-Type**

```
uploadFile(file: File, url: string) {
  const headers = new HttpHeaders().set('Content-Type', file.type);
  const options = {
    headers,
    reportProgress: true,
    observe: 'events',
    responseType: 'text' as 'text',
  };

  const req = new HttpRequest('PUT', url, file, options);
  return this.http.request(req);
}
```

Headers and Options

Error Handling

- We use the **catchError** operator to well… catch the error
- We can create a general error handling method that can process the error internally while surfacing a user friendly error via **throwError**

```
private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error('Error Status: ', error.status);
    console.error('Error Details: ', error.error);
  }
  // return an observable with a user-facing error message
  return throwError('Something bad happened; please try again later.');
};
```

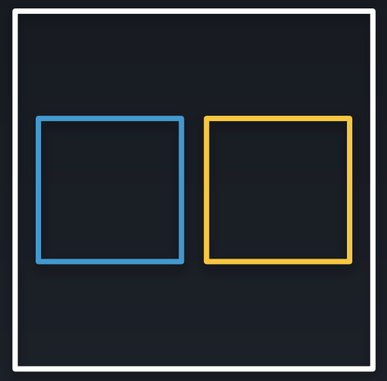
Handle the Error

```
loadItems() {
  return this.http.get(BASE_URL)
  .pipe(
    catchError(this.handleError)
  );
}
```

Catching the Error

Component Driven Architecture

A Brief History of Angular



tiny app == tiny view + tiny controller

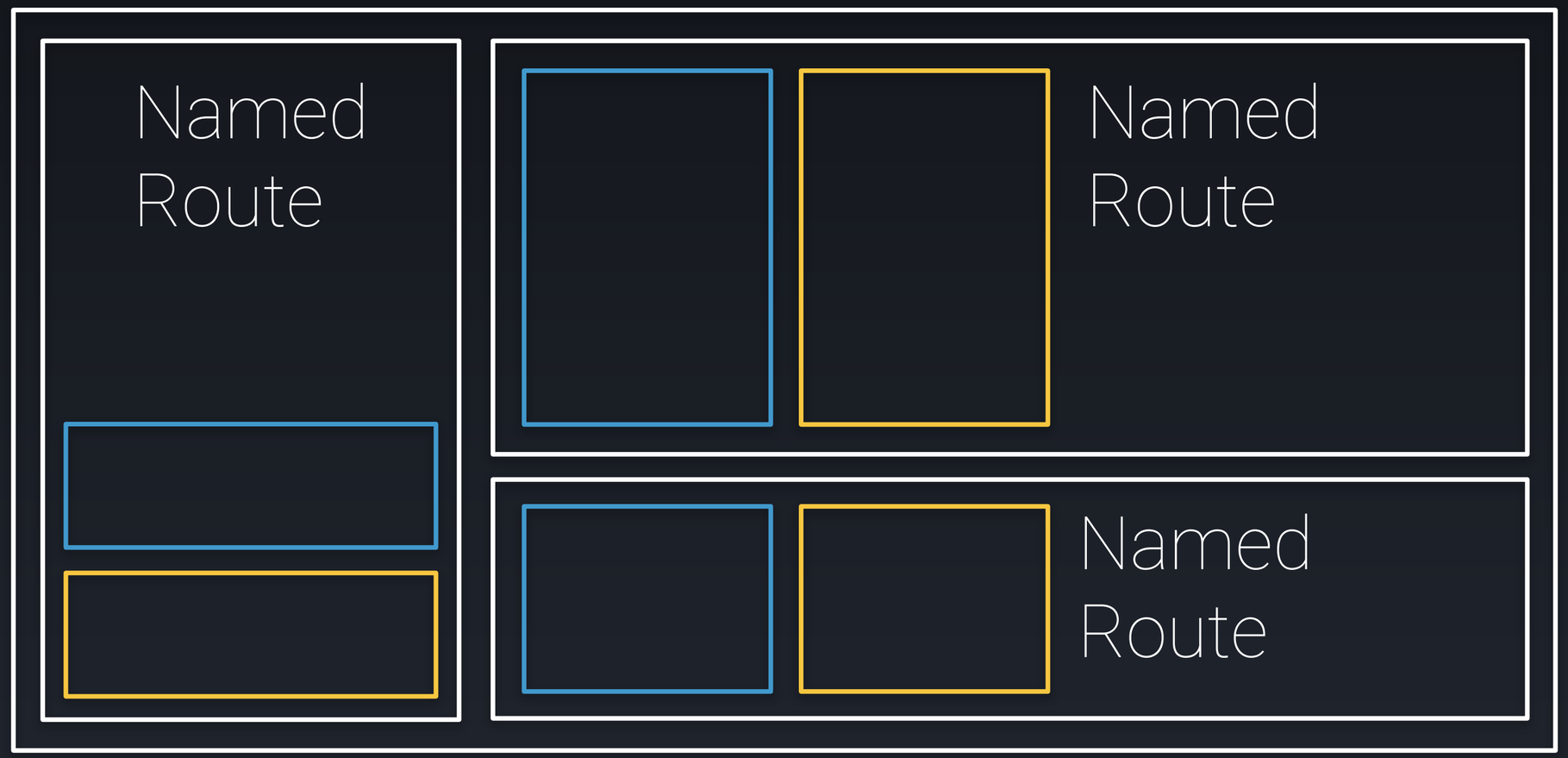

Growing
View
Growing
Controller
Growing Application

Realistic Application

Growing
View

Growing
Controller

Uh oh!



Large 1.x Application

Large 1.x Application

Any Angular Application

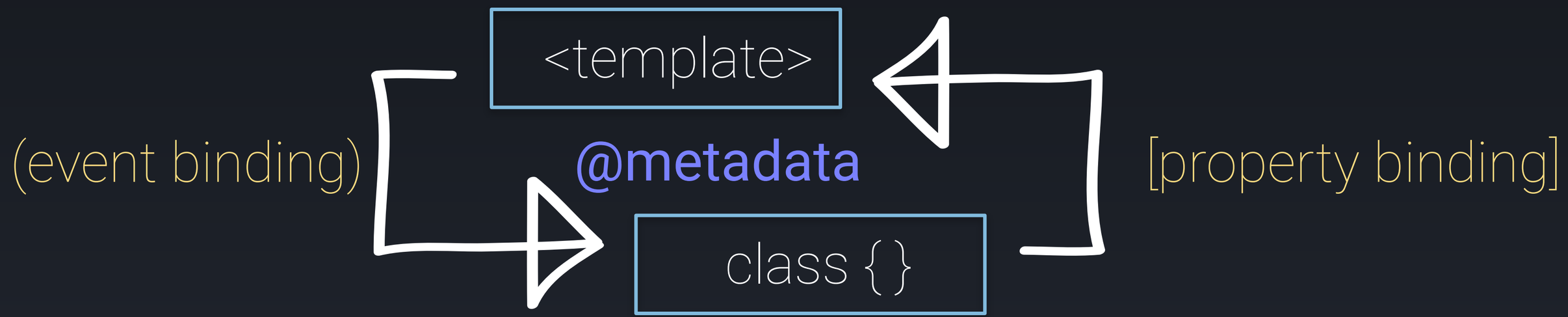
This helps us solve the problem of **structure**

This helps us solve
the problem of
communication

Component Driven Architecture

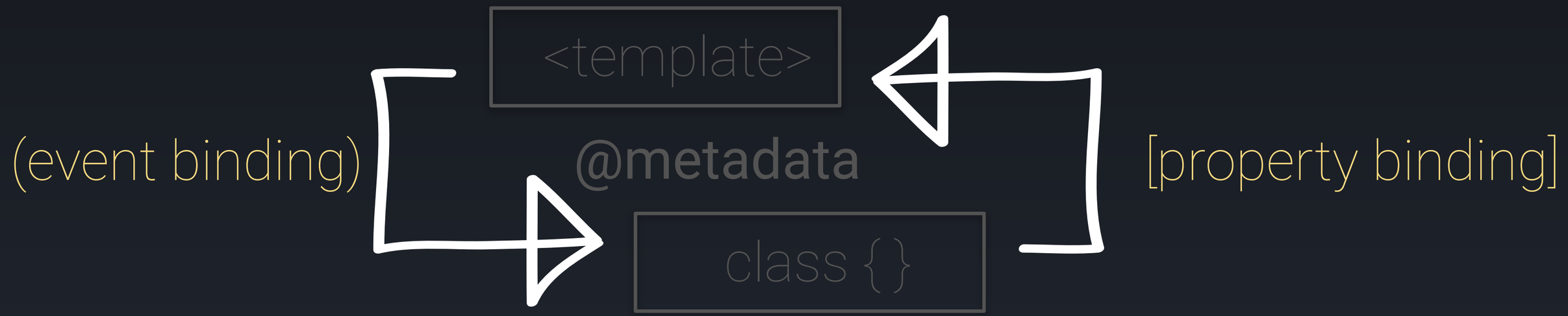
- Components are small, encapsulated pieces of software that can be reused in many different contexts
- Angular strongly encourages the component architecture by making it easy (and necessary) to build out every feature of an app as a component
- Angular components self encapsulated building blocks that contain their own templates, styles, and logic so that they can easily be ported elsewhere

Data Binding

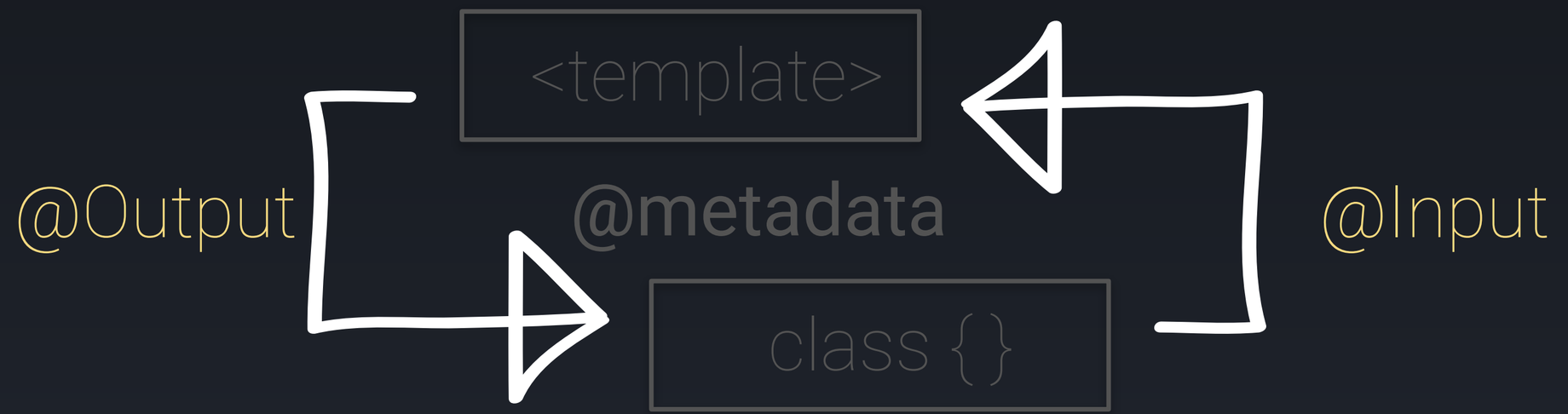


What if we could define **custom** properties and events to bind to?

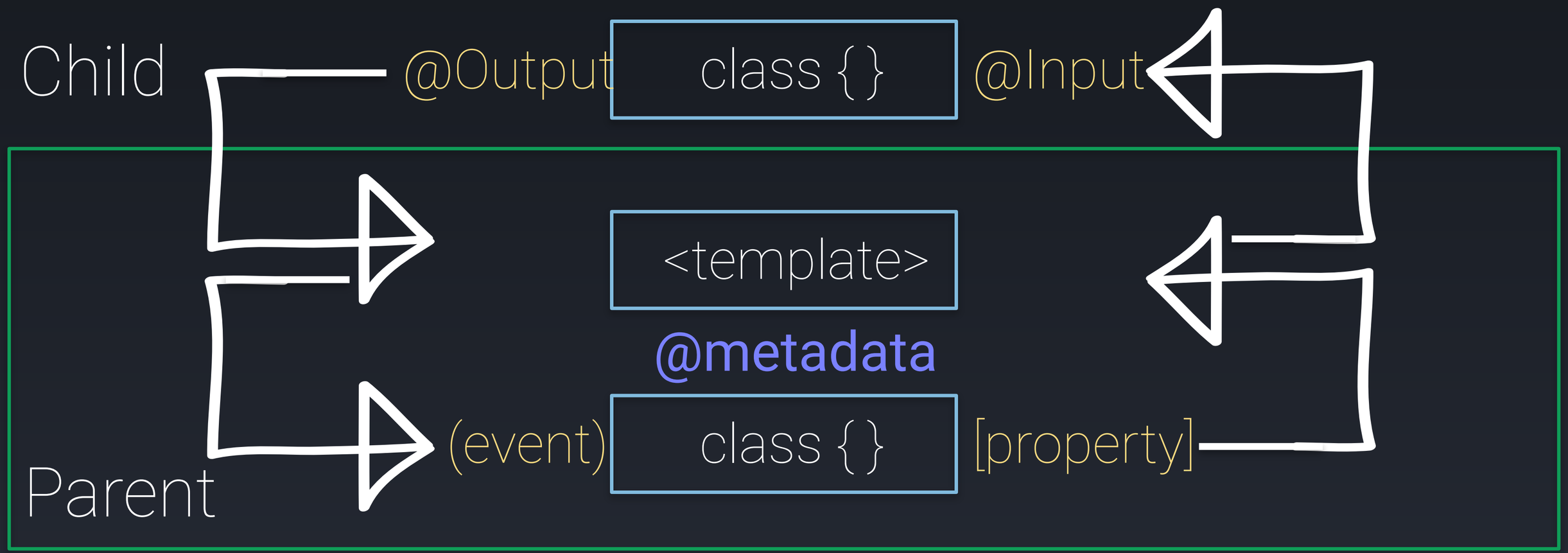
Custom Data Binding



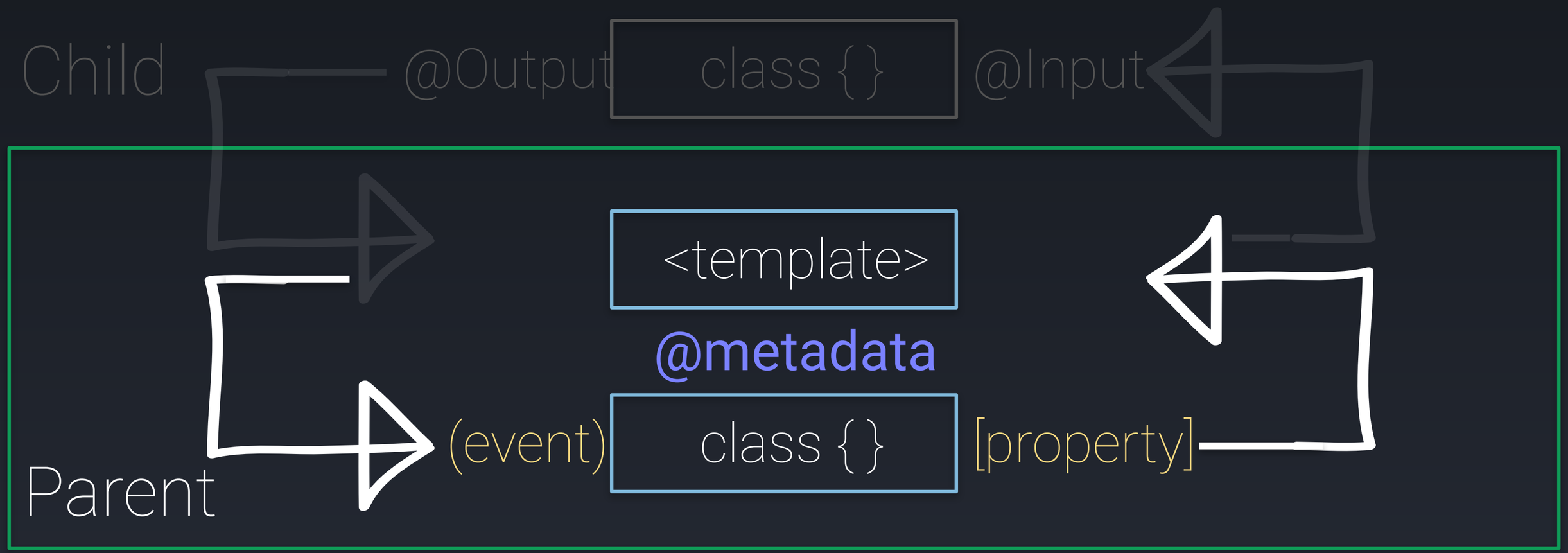
Component Contract



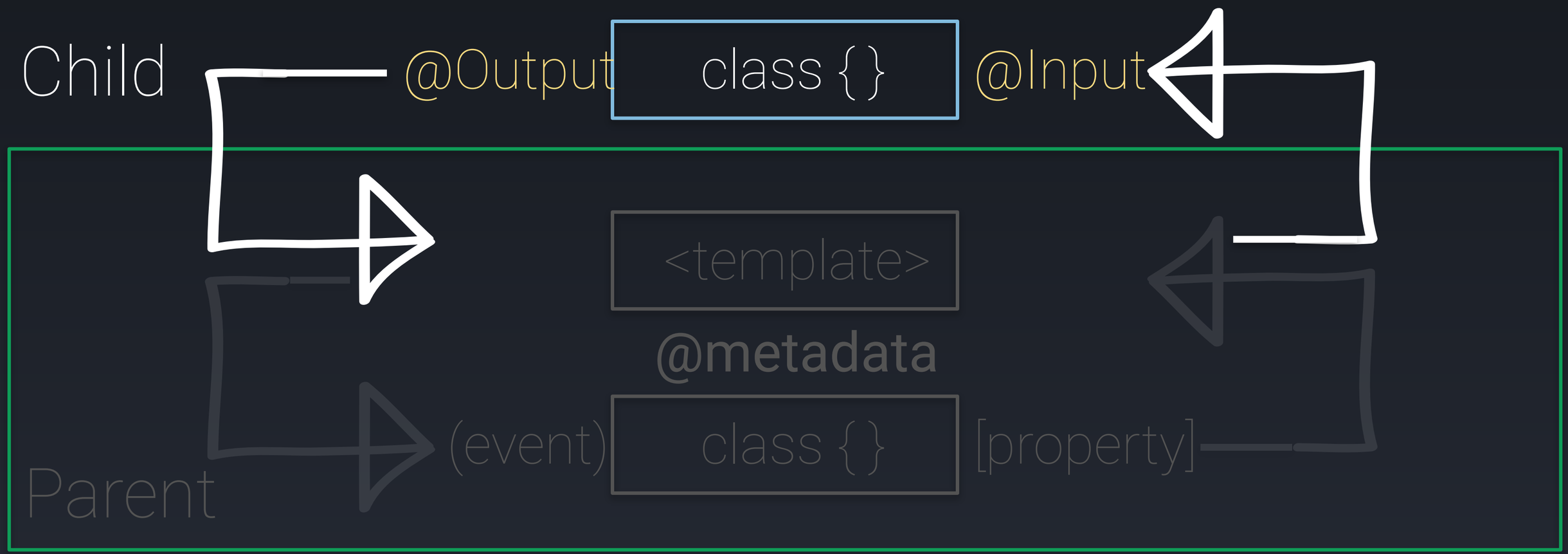
Parent and Child



Parent and Child



Parent and **Child**



@Input

- Allows data to flow from a parent component to a child component
- Defined inside a component via the **@Input** decorator: **@Input() someValue: string;**
- Bind in parent template: **<component [someValue]="value"></component>**
- We can alias inputs: **@Input('alias') someValue: string;**

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'my-component',
  template: `
  <div>Greeting from parent:</div>
  <div>{{greeting}}</div>
  `
})
export class MyComponent {
  @Input() greeting: String = 'Default Greeting';
}
```

@Input

```
@Component({
  selector: 'app',
  template: `
  <my-component [greeting]="greeting"></my-component>
  <my-component></my-component>
  `
})
export class App {
  greeting = 'Hello child!';
}
```

Parent Component

@Output

- Exposes an **EventEmitter** property that emits events to the parent component
- Defined inside a component via the @Output decorator: **@Output() showValue = new EventEmitter();**
- Bind in parent template: **<cmp (someValue)="handleValue()"></cmp>**

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'my-component',
  template: `<button (click)="greet()">Greet Me</button>`
})
export class MyComponent {
  @Output() greeter = new EventEmitter();
  greet() {
    this.greeter.emit('Child greeting emitted!');
  }
}
```

@Output

```
@Component({
  selector: 'app',
  template: `
  <div>
    <h1>{{greeting}}</h1>
    <my-component (greeter)="greet($event)"></my-component>
  </div>
  `
})
export class App {
  private greeting;
  greet(event) {
    this.greeting = event;
  }
}
```

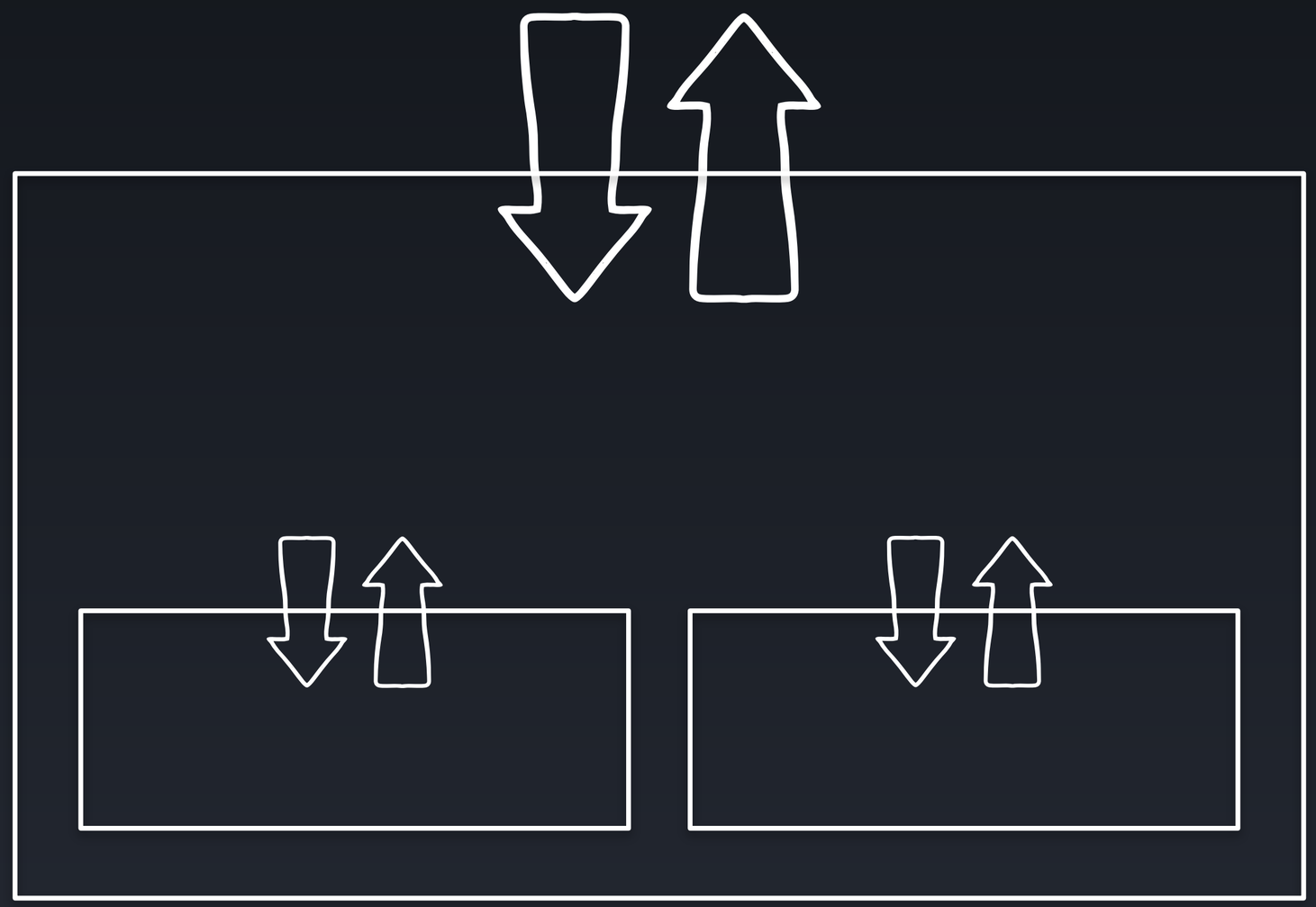
Parent Component

Component Contracts

- Represents an agreement between the software developer and software user – or the supplier and the consumer
- **Inputs** and **Outputs** define the interface of a component
- These then act as a contract to any component that wants to consume it
- Also act as a visual aid so that we can infer what a component does just by looking at its inputs and outputs

```
<app-items-list [items]="items"
                (selected)="selectItem($event)"
                (deleted)="deleteItem($event)">
</app-items-list>
```

Component Contract



Nice Neat Containers

Container and Presentational Components

- Container components are connected to services
- Container components know how to load their own data, and how to persist changes
- Presentational components are fully defined by their bindings
- All the data goes in as inputs, and every change comes out as an output
- Create as few container components/many presentational components as possible

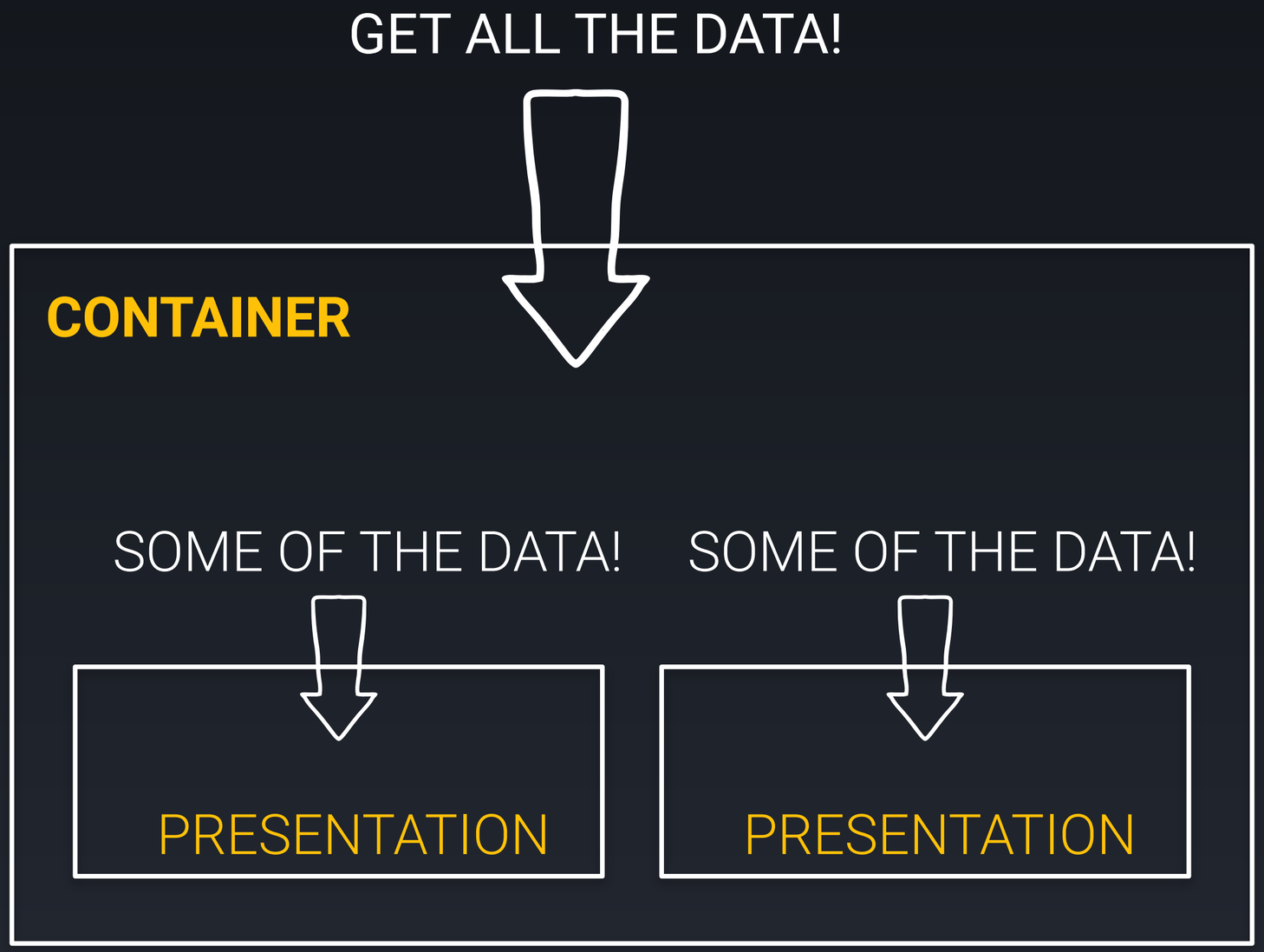
```ts
export class ItemsListComponent {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Presentational Component

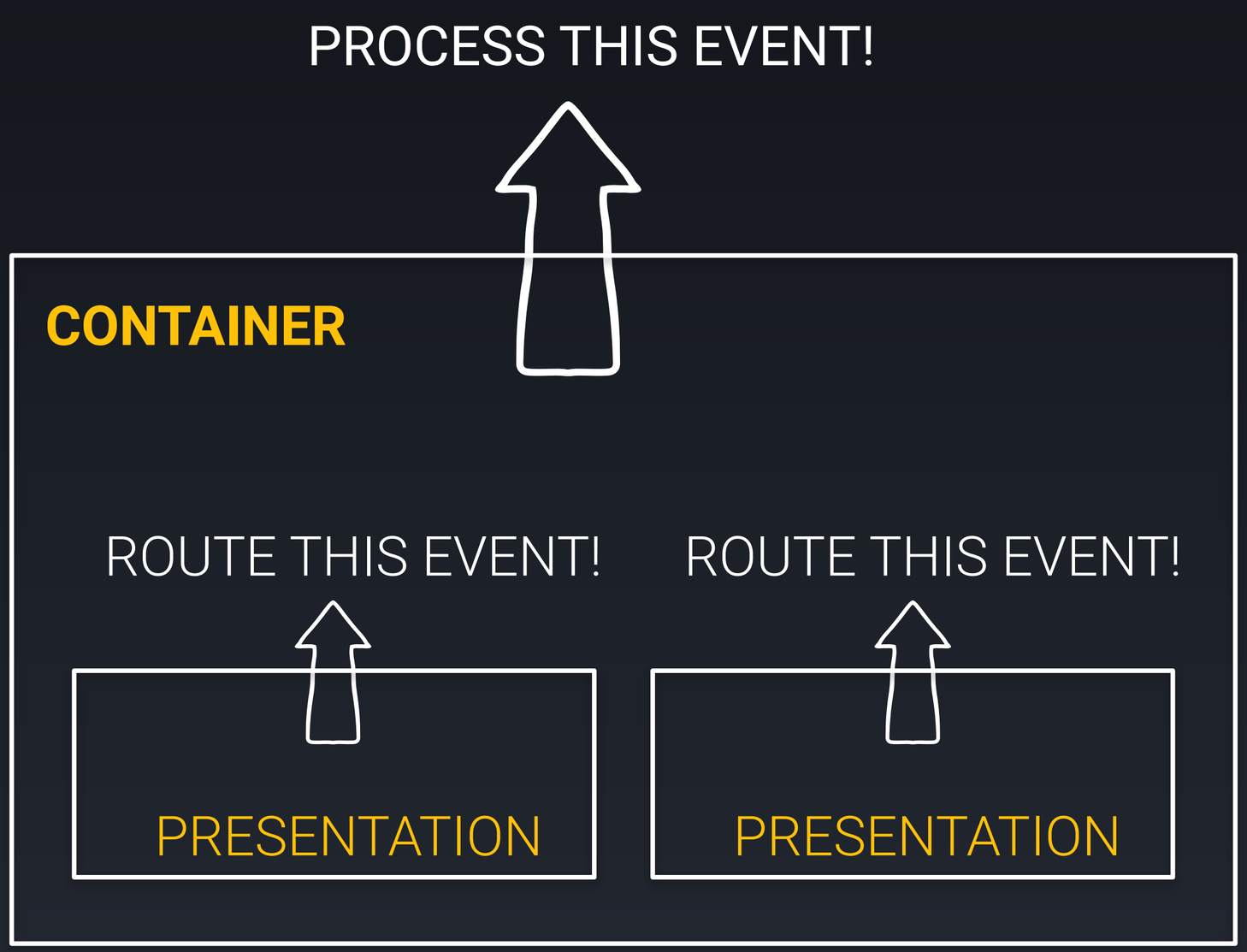
```
export class ItemsComponent implements OnInit {
  items: Array<Item>;
  selectedItem: Item;
  constructor(private itemsService: ItemsService) { }
  ngOnInit() { }
  resetItem() { }
  selectItem(item: Item) { }
  saveItem(item: Item) { }
  replaceItem(item: Item) { }
  pushItem(item: Item) { }
  deleteItem(item: Item) { }
}
```

Container Component

State flows **down**


GET ALL THE DATA!
CONTAINER
SOME OF THE DATA!
SOME OF THE DATA!
PRESENTATION
PRESENTATION

Events flows **up**


PROCESS THIS EVENT!
CONTAINER
ROUTE THIS EVENT!
ROUTE THIS EVENT!
PRESENTATION
PRESENTATION

You **basically**
understand redux

WAT.

Angular **Routing**

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

```ts
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

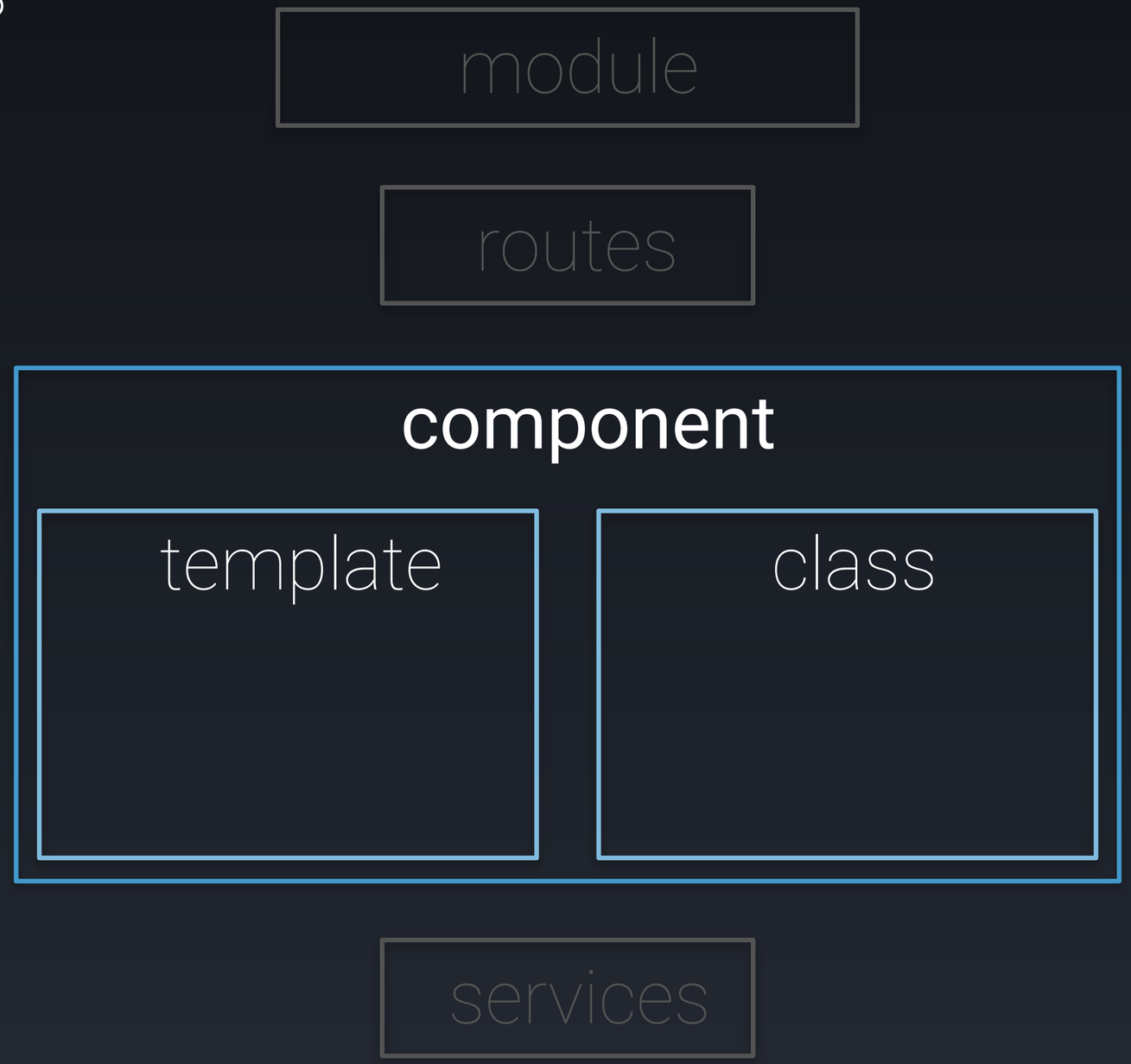
module

routes

components

services

Components



Testing Fundamentals

TESTING **IS**

HARD!

WRITING SOFTWARE IS HARD!

WAT.

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand.

Out of the Tarpit - Ben Mosely Peter Marks

We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.

Out of the Tarpit - Ben Mosely Peter Marks

Complexity and purgatory

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {
  switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.reCalculateTotal(testWidget);

testService.mode = 'update';
testService.reCalculateTotal(testWidget);

testService.mode = 'delete';
testService.reCalculateTotal(testWidget);
```

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.reCalculateTotal(testWidget);

testService.mode = 'update';
testService.reCalculateTotal(testWidget);

testService.mode = 'delete';
testService.reCalculateTotal(testWidget);
```

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {

  switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }


  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {

   switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }


  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

TESTING **IS**

HARD!

Testing can be summarized with some **basic patterns**

Small methods are
easier to test

Pure methods are

easier to test

Focus on testing just what that method does

Don't use **real**
services

Don't use **real**
services,
use a **test double**

Don't use **real**

services,

use a **stub**

Don't use **real**

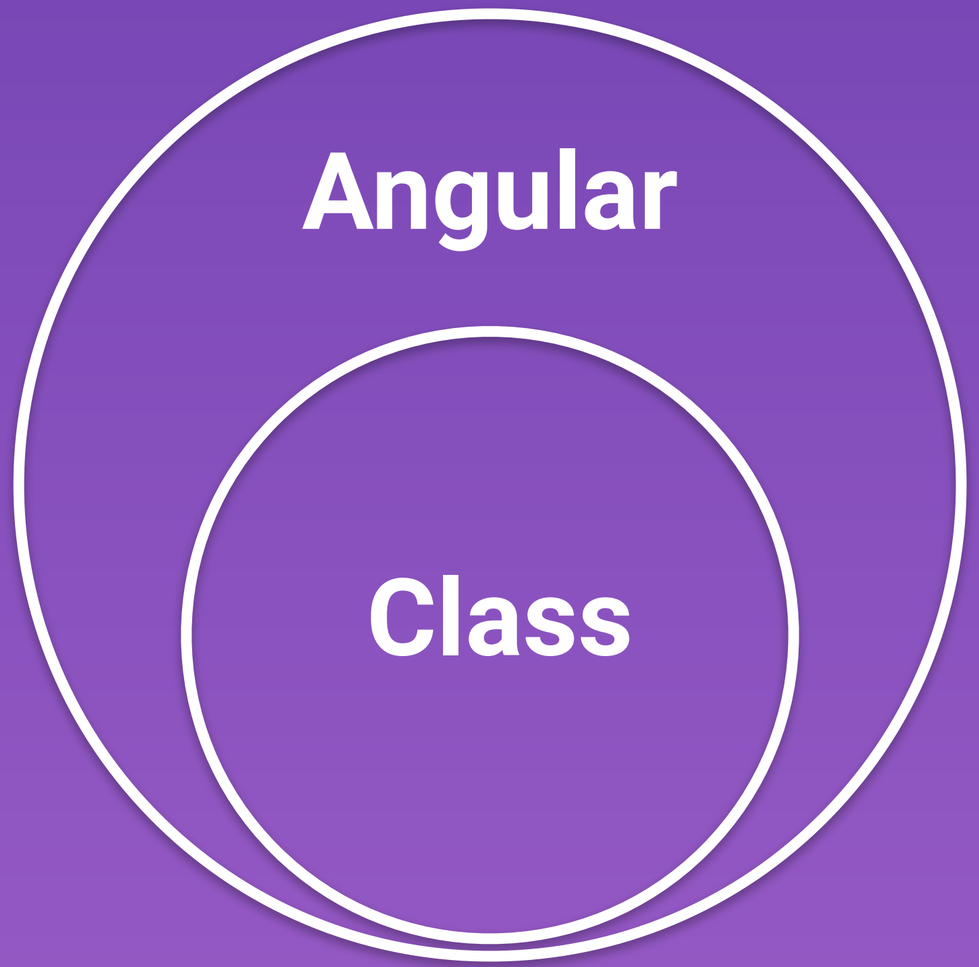
services,

use a **spy**

Faking and spying are both great options

Faking and **spying** are both **great** options, start with what is **easiest**

Faking and **spying** are both **great** options, you can use **both**



Basic Structure

Utilities vs **Isolated**

Two Approaches

The Testing Big Picture

Karma

Jasmine

Testing Utilities

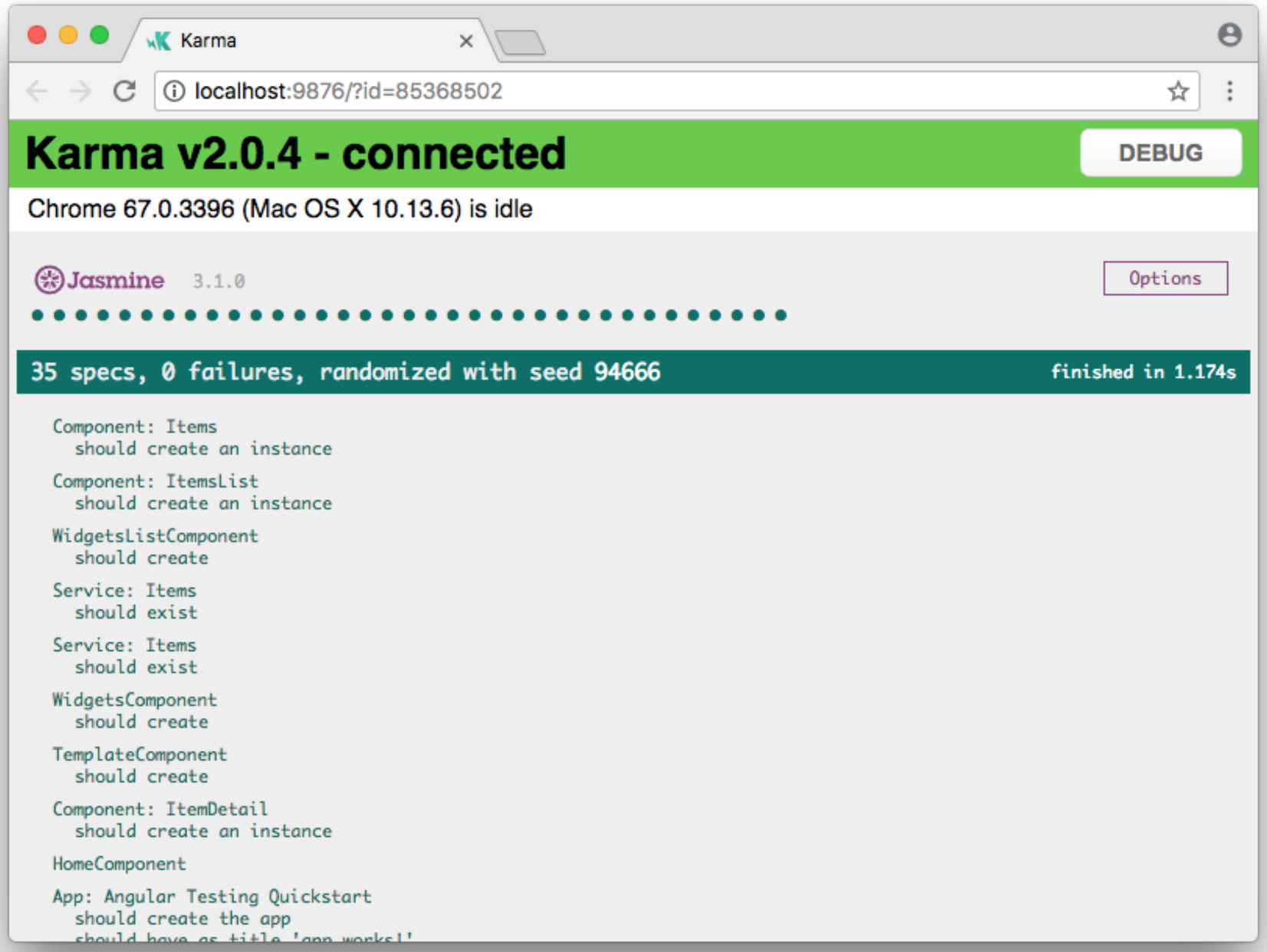
Code

Your **First** Test

Karma

- Karma is the test runner that is used to execute Angular unit tests
- You can manually install and configure Karma
- Karma is **installed** and **configured** by default when you create a project with the Angular CLI
- Karma is configured via the **karma.conf.js** file
- Tests (specs) are identified with a **.spec.ts** naming convention

```
➜  angular-testing-quickstart git:(master) ✗  ng test
 10% building modules 1/1 modules 0 active(node:53829) DeprecationWarning: Tapable.plugin is deprecated. Use
 new API on `.hooks` instead
14 07 2018 18:13:10.738:WARN [karma]: No captured browser, open http://localhost:9876/
14 07 2018 18:13:10.741:INFO [karma]: Karma v2.0.4 server started at http://0.0.0.0:9876/
14 07 2018 18:13:10.742:INFO [launcher]: Launching browser Chrome with unlimited concurrency
14 07 2018 18:13:10.753:INFO [launcher]: Starting browser Chrome  14 07 2018 18:13:17.656:WARN [karma]: No c
aptured browser, open http://localhost:9876/
14 07 2018 18:13:17.766:INFO [Chrome 67.0.3396 (Mac OS X 10.13.6)]: Connected on socket R7GOytYJJ8_dOCnHAAAA
 with id 50973291
Chrome 67.0.3396 (Mac OS X 10.13.6): Executed 35 of 35 SUCCESS (1.067 secs / 0.999 secs)
TOTAL: 35 SUCCESS
TOTAL: 35 SUCCESS
TOTAL: 35 SUCCESS
```


Karma
localhost:9876/?id=85368502
Karma v2.0.4 - connected
DEBUG
Chrome 67.0.3396 (Mac OS X 10.13.6) is idle
Jasmine 3.1.0
Options
35 specs, 0 failures, randomized with seed 94666
finished in 1.174s
Component: Items
should create an instance
Component: ItemsList
should create an instance
WidgetsListComponent
should create
Service: Items
should exist
Service: Items
should exist
WidgetsComponent
should create
TemplateComponent
should create
Component: ItemDetail
should create an instance
HomeComponent
App: Angular Testing Quickstart
should create the app

Debugging with Karma

- Use the developer console in the Karma browser window to debug your unit tests
- If something is throwing an error, you will generally see it in the console
- If you need to step through something, you can do some from a breakpoint in the developer tools
- Logging to the console is also a handy tool for observing data and events

```
describe('First spec', () => {
  it('should pass', () => {
    expect(false).toBeTruthy();
  });
});
```

Simple Test Fail

```
describe('First spec', () => {
  it('should pass', () => {
    expect(true).toBeTruthy();
  });
});
```

Simple Test Pass

Basic Component Test

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

@Component({
  selector: 'app-simple',
  template: '<h1>Hello {{subject}}!</h1>'
})
export class SimpleComponent implements OnInit {
  subject: string = 'world';
  constructor() { }
  ngOnInit() { }
}
```

The Component

1. Configure Module

TestBed

- The most important piece of the Angular testing utilities
- Creates an Angular testing module which is an **@NgModule** class
- You can configure the module by calling **TestBed.configureTestingModule**
- Configure the testing module in the **BeforeEach** so that it gets reset before each spec

```
import { TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: any;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    });
  });
});
```

Configure Module

1. Configure Module

2. Create Fixture

TestBed.createComponent

- Creates an instance of the component under test
- Returns a component test fixture
- Calling **createComponent** closes the **TestBed** from further configuration

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: ComponentFixture<SimpleComponent>;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);
  });
});
```

The Fixture

1. Configure Module
2. Create Fixture
3. Get Component Instance

ComponentFixture

- Handle to the test environment surrounding the component
- Provides access to the component itself via **fixture.componentInstance**
- Provides access to the **DebugElement** which is a handle to the component's DOM element
- **DebugElement.query** allows us to query the DOM of the element
- **By.css** allows us to construct our query using CSS selectors

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: ComponentFixture<SimpleComponent>;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
  });
});
```

The Component Instance

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: ComponentFixture<SimpleComponent>;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
  });

  it('sets the `subject` class member', () => {
    expect(component.subject).toBe('world');
  });
});
```

The Component Instance

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: ComponentFixture<SimpleComponent>;
  let de: DebugElement;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
    de = fixture.debugElement;
    fixture.detectChanges();
  });

  it('greets the subject', () => {
    const h1 = de.query(By.css('h1'));
    expect(h1.nativeElement.innerText).toBe('Hello world!');
  });
});
```

The Debug Element

ComponentFixture.detectChanges

- We tell Angular to perform change detection by calling **ComponentFixture.detectChanges**
- **TestBed.createComponent** does not automatically trigger a change detection
- This is intentional as it gives us greater control over how we inspect our components pre-binding and post-binding

```
it('greets the subject', () => {
  const h1 = de.query(By.css('h1'));
  expect(h1.nativeElement.innerText).toBe('Hello world!');
});
```

```
it('updates the subject', () => {
  component.subject = 'developer';
  fixture.detectChanges();
  const h1 = de.query(By.css('h1'));
  expect(h1.nativeElement.innerText).toBe('Hello developer!');
});
```

detectChanges

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

describe('$COMPONENT$', () => {
  let component: $COMPONENT$$END$;
  let fixture: ComponentFixture<$COMPONENT$>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ $COMPONENT$ ]
    })
      .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent($COMPONENT$);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Component Testing Patterns

Component with an External Template

- With an external template, Angular needs to read the file before it can create a component instance. This is problematic because **TestBed.createComponent** is **synchronous**.
- The first thing we do is break our initial **beforeEach** into an **asynchronous beforeEach** call and a **synchronous beforeEach** call
- We then use the **async** testing utility to load our external templates
- And then call **TestBed.compileComponents** to compile our components
- **WebPack users can skip this slide**

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ TemplateComponent ]
  })
  .compileComponents();
}));
```

async

```
beforeEach(() => {
  fixture = TestBed.createComponent(TemplateComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

fixture

Component with a Service Dependency

- Components do not need to be injected with real services
- Use test doubles to stand in for the real service since we are testing the component and not the service
- We can override the provider with **useValue** or **useClass** and provide our custom test double
- Use **debugElement.injector** to get a reference to the service from the component's injector

```
@Component({
  selector: 'app-service',
  template: '<h1>Hello {{subject.name}}!</h1>'
})
export class ServiceComponent implements OnInit {
  subject: {name: string} = this.service.subject;
  constructor(private service: GreetingService) { }
  ngOnInit() { }
}
```

Component

```
export class GreetingService {
  subject: {name: string} = { name: 'world' };
}
```

Service

```
describe('ServiceComponent', () => {
  let component: ServiceComponent;
  let fixture: ComponentFixture<ServiceComponent>;
  let de: DebugElement;
  let greetingServiceStub;
  let greetingService;
});
```

Local Members

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
it('updates component subject when service subject is changed', () => {
  greetingService.subject.name = 'cosmos';
  fixture.detectChanges();
  expect(component.subject.name).toBe('cosmos');
  const h1 = de.query(By.css('h1')).nativeElement;
  expect(h1.innerText).toBe('Hello cosmos!');
});
```

Actual Test

Service with HttpClient

- Testing a service that uses **HttpClient** client is easy with **HttpTestingController** because it allows us to mock and flush requests
- **HttpTestingController** is available through the **HttpClientTestingModule**
- We can define expected request behavior with **match**, **expectOne**, **expectNone** and **verify**
- The above methods returns a **TestRequest** object which we can use to run assertions on and more importantly call **flush** or **error** on it to resolve the request

```ts
import { TestBed, getTestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { RemoteService } from './remote.service';

describe('RemoteService', () => {
  let injector: TestBed;
  let service: RemoteService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [RemoteService]
    });

    injector = getTestBed();
    service = injector.get(RemoteService);
    httpMock = injector.get(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();
  });
});
```

```
it('should fetch all widgets', () => {
  const mockWidgets: Widget[] = [
    {id: 1, name: 'mock', description: 'mock', price: 100},
    {id: 2, name: 'mock', description: 'mock', price: 100},
    {id: 3, name: 'mock', description: 'mock', price: 100}
  ];

  const results = service.all();
  results
    .subscribe((widgets: Widget[]) => {
      expect(widgets.length).toBe(3);
      expect(widgets).toEqual(mockWidgets);
    });

  const req = httpMock.expectOne(`${BASE_URL}`);
  expect(req.request.method).toBe('GET');
  req.flush(mockWidgets);
});
```

```
it('should post a new widget', () => {
  const mockWidget: Widget = {id: null, name: 'new widget', description: 'new widget', price:
100};
  const results = service.create(mockWidget);
  results.subscribe(results => {});

  const req = httpMock.expectOne(`${BASE_URL}`, JSON.stringify(mockWidget));
  expect(req.request.method).toBe('POST');
  req.flush(mockWidget);
});
```

Component with a Remote Service

- To avoid calling a service that makes remote calls, it is usually easiest to start with a very simple mock of the service
- You then can replace the real service with the mock when you configure the testing module
- Because your primary concern is if the component is delegating properly, you can verify this behavior using spies

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { of } from 'rxjs/internal/observable/of';
import { noop } from 'rxjs/internal-compatibility';

import { RemoteComponent } from './remote.component';
import { RemoteService } from './remote.service';

class RemoteServiceStub {
  all() { return of(noop())}
  create() { return of(noop()) }
  update() { return of(noop()) }
  delete() { return of(noop()) }
}
```

```
let component: RemoteComponent;
let fixture: ComponentFixture<RemoteComponent>;
let debugElement: DebugElement;
let service: RemoteService;

beforeEach(async(() => {
  TestBed.configureTestingModule({
      declarations: [
        RemoteComponent
      ],
      providers: [
        {provide: RemoteService, useClass: RemoteServiceStub}
      ]
    })
    .compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(RemoteComponent);
  component = fixture.componentInstance;
  debugElement = fixture.debugElement;
  service = debugElement.injector.get(RemoteService);
  fixture.detectChanges();
});
```

```
it('should call remoteService.all on getWidgets', () => {
  spyOn(service, 'all').and.callThrough();

  component.getWidgets();

  expect(service.all).toHaveBeenCalled();
});
```

```
it('should call remoteService.create on createWidget', () => {
  const mockWidget: Widget = {id: null, name: 'item', description: 'item', price: 100};
  spyOn(service, 'create').and.callThrough();

  component.createWidget(mockWidget);

  expect(service.create).toHaveBeenCalledWith(mockWidget);
});
```

I  YOU!

@simpulton

Thanks!