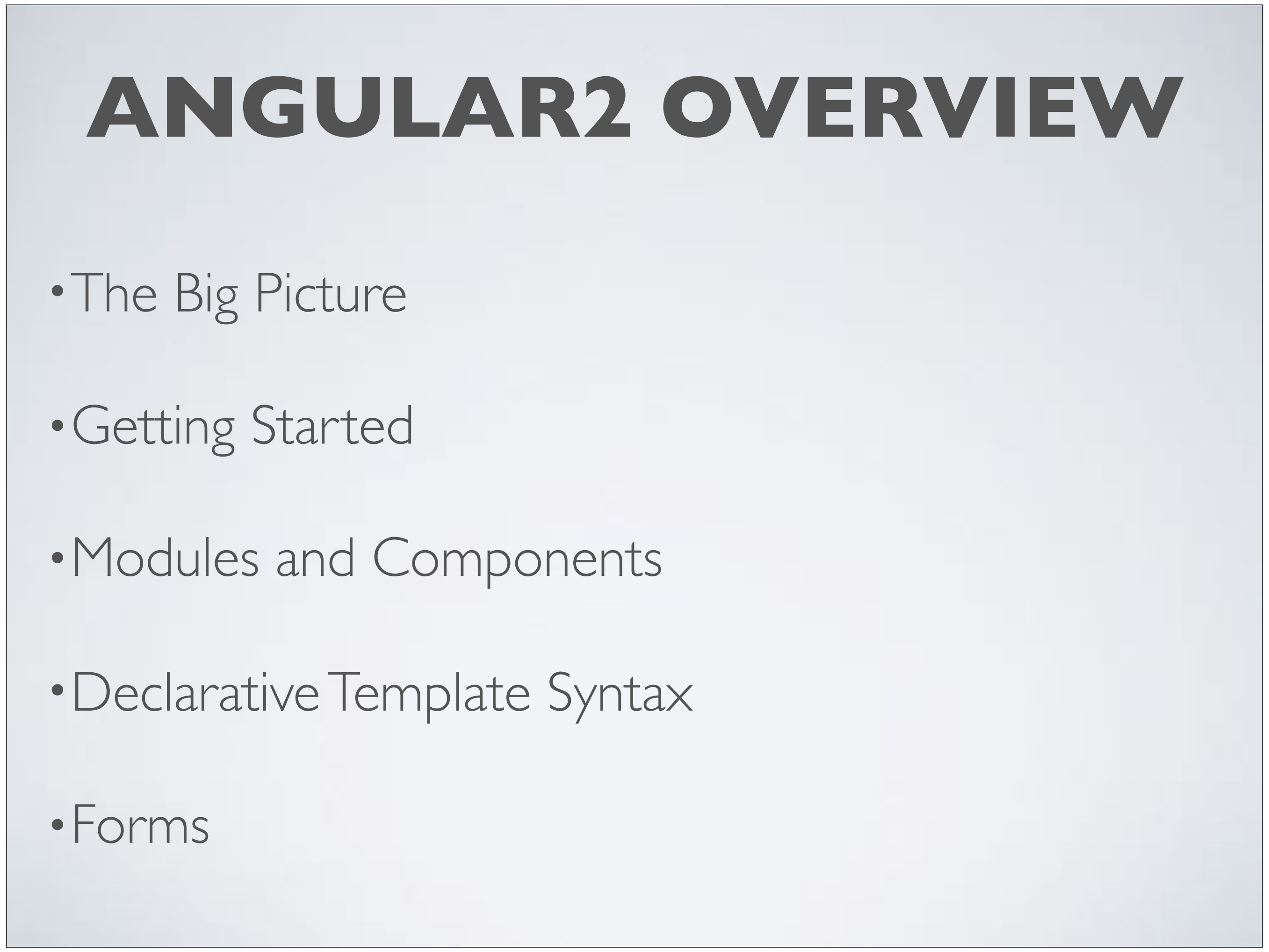
FORMS IN ANGULAR

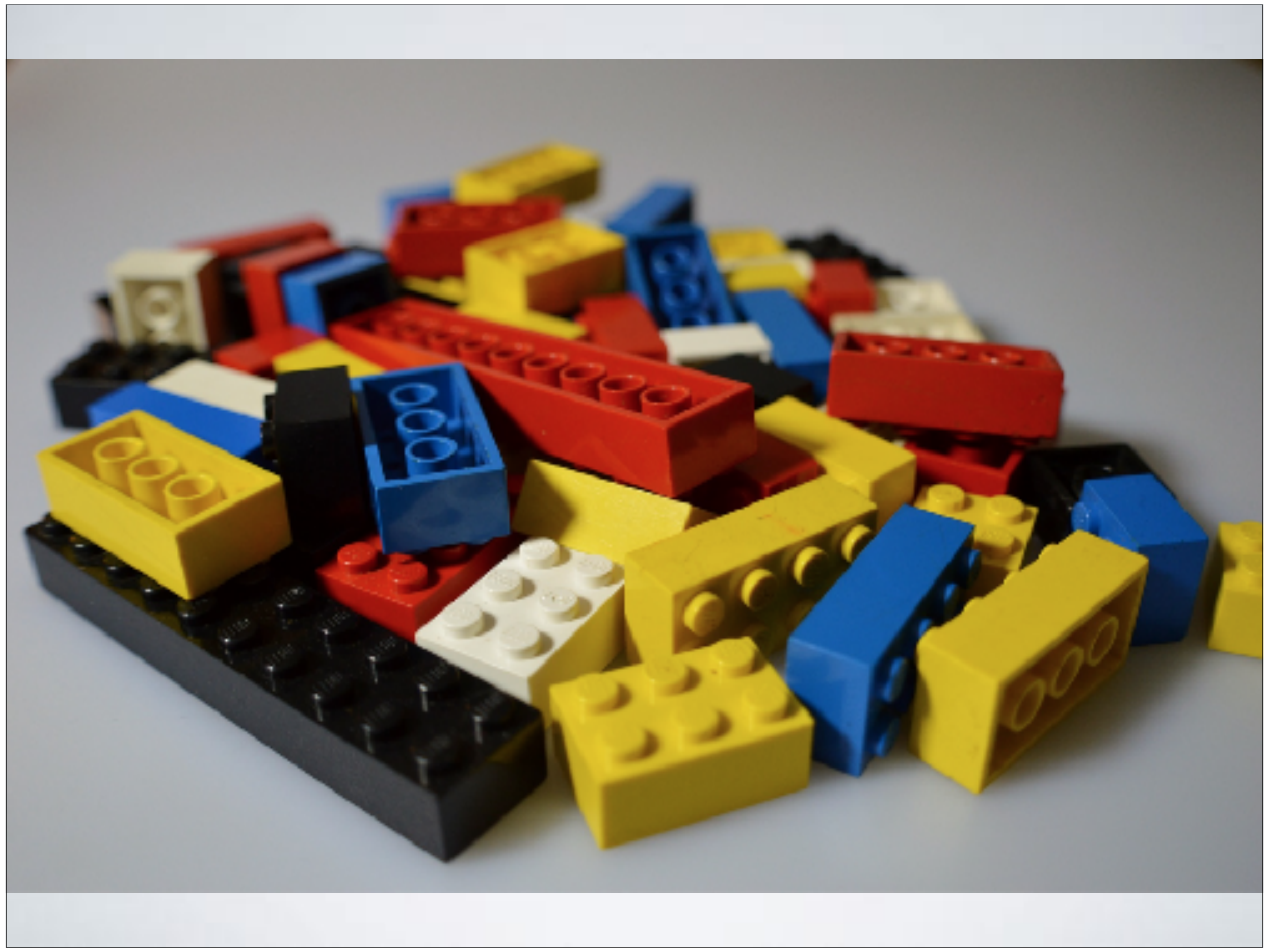
Hello Cluj. I'm Alex Lakatos, a Mozilla volunteer which helps other people volunteer. I want to talk to you today about Angular forms. What's a form you ask? A form creates a cohesive, effective, and compelling data entry experience. An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.



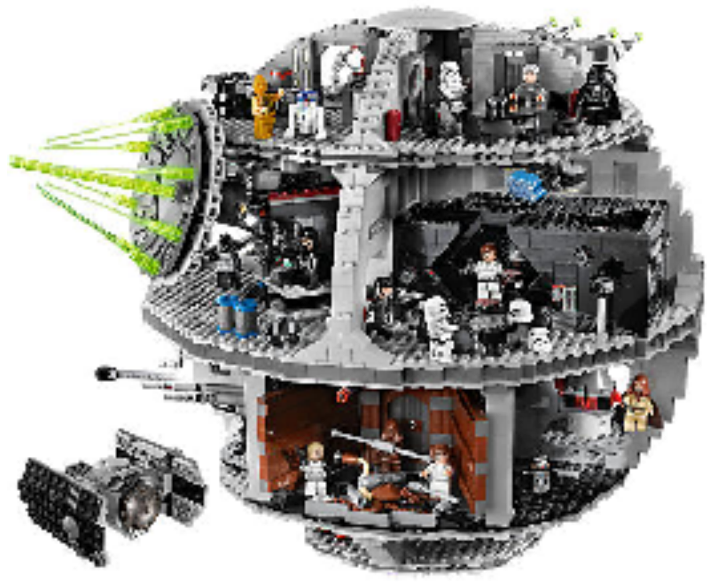
ANGULAR2 OVERVIEW

- The Big Picture
- Getting Started
- Modules and Components
- Declarative Template Syntax
- Forms

Angular 2 is all about building blocks. So if you’re someone who likes to build things this is a fun story to hear about!

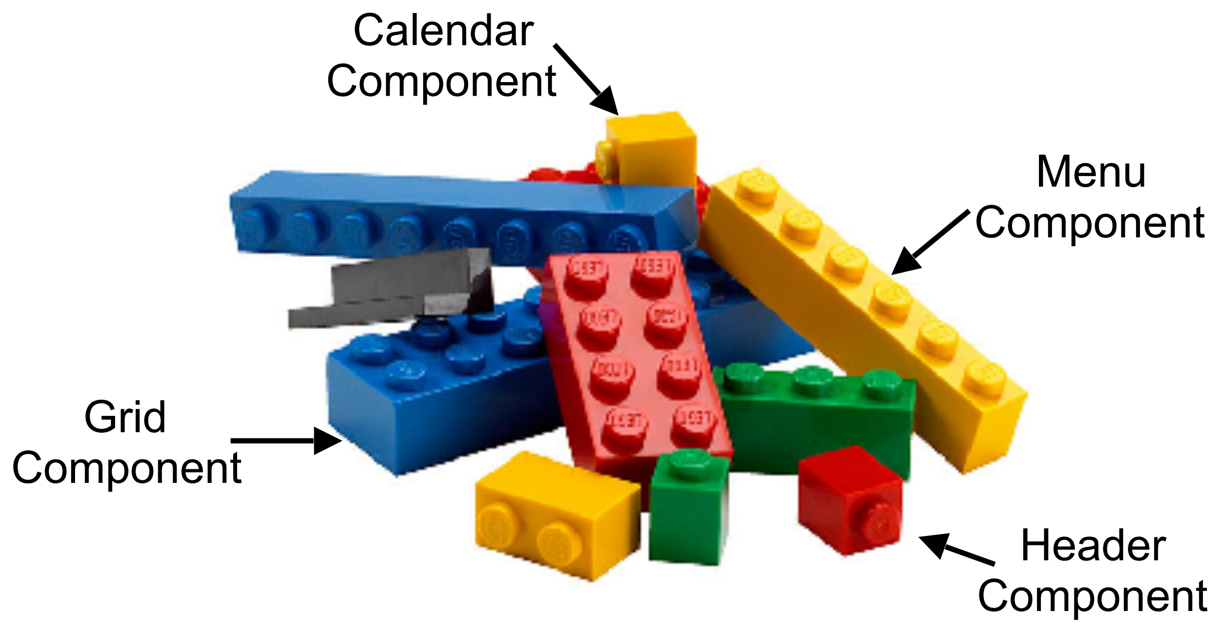


Now, these aren’t just any type of building block. These legos can be re-used in various ways and have the power to build applications that can target multiple devices and browsers.

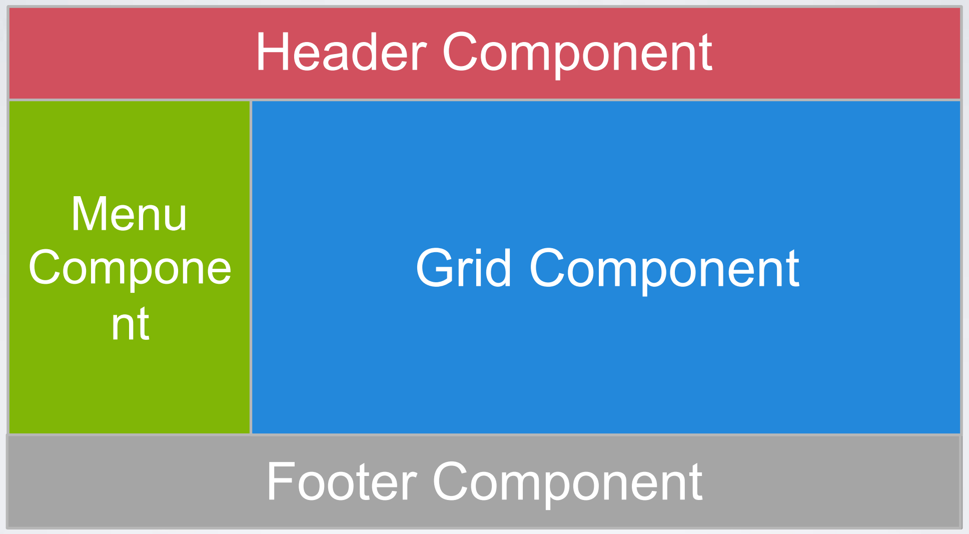


When the building blocks are combined you can build some cool things! Being recruited by the dark side? Build the next enterprise-scale app for the death star (since I’m sure they’ll build another one).

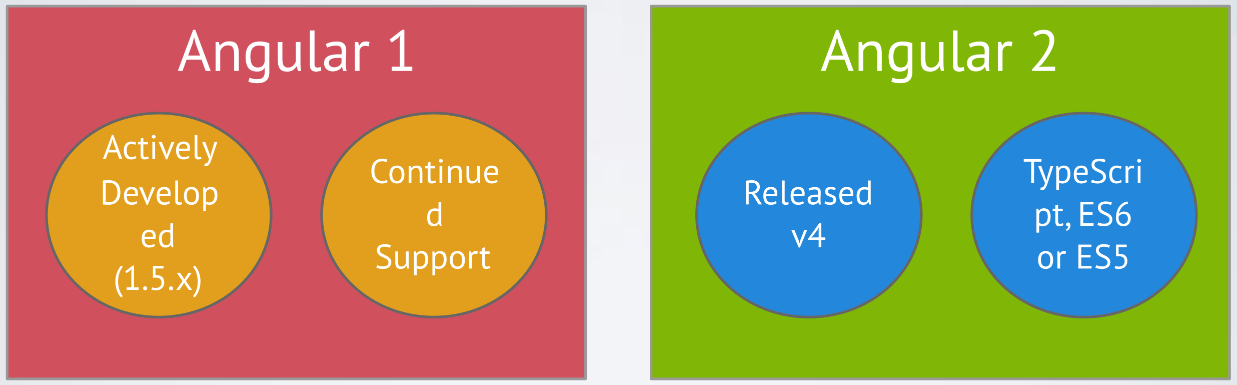
Being recruited by the rebel alliance to build an app? You can do that too of course.

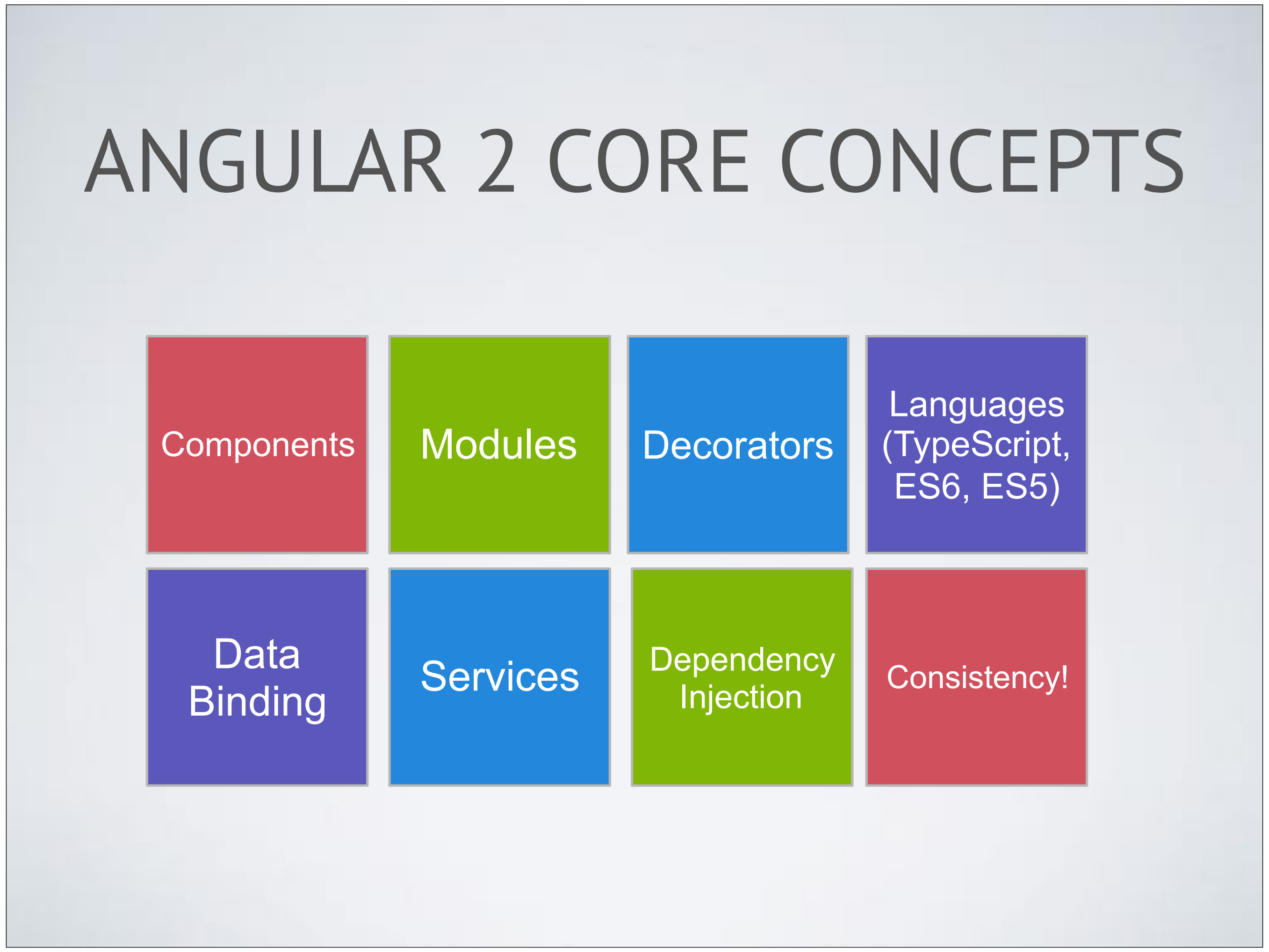

Calendar Component
Menu Component
Grid Component
Header Component

BUILDING SPAS WITH COMPONENTS



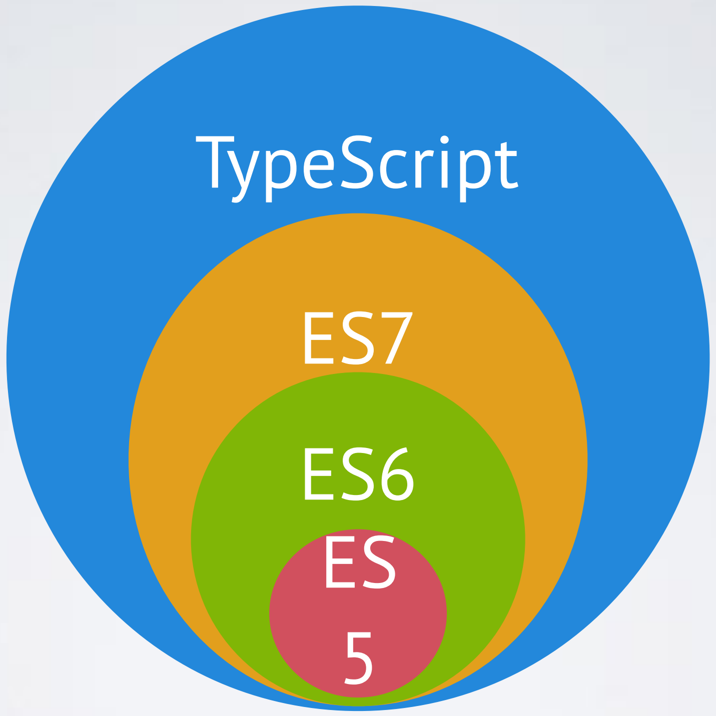
BUILDING SPAS & COMPONENTS WITH ANGULAR





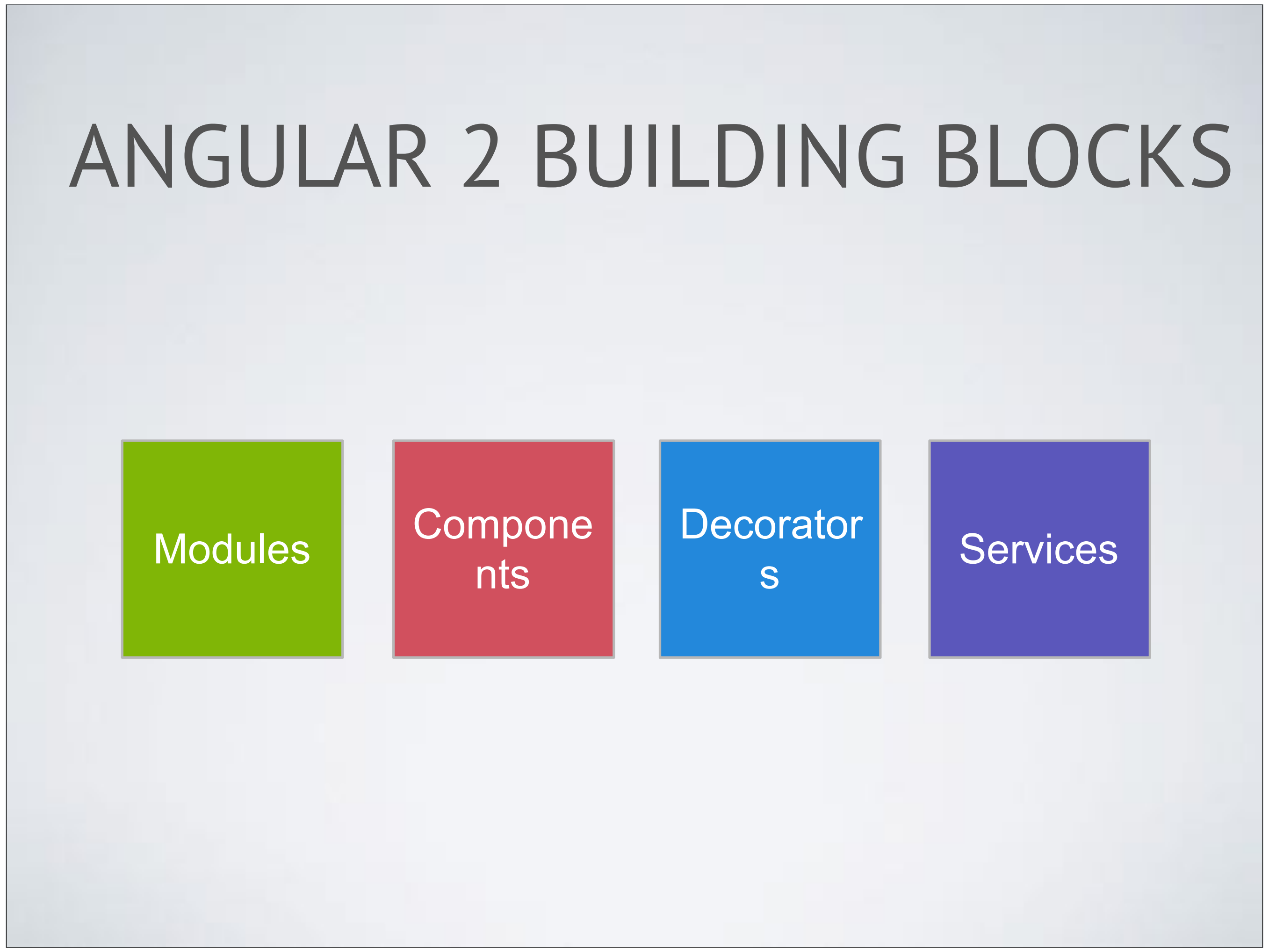
ANGULAR 2 CORE CONCEPTS

Components	Modules	Decorators	Languages (TypeScript, ES6, ES5)
Data Binding	Services	Dependency Injection	Consistency!


TypeScript
ES7
ES6
ES
5

GETTING STARTED

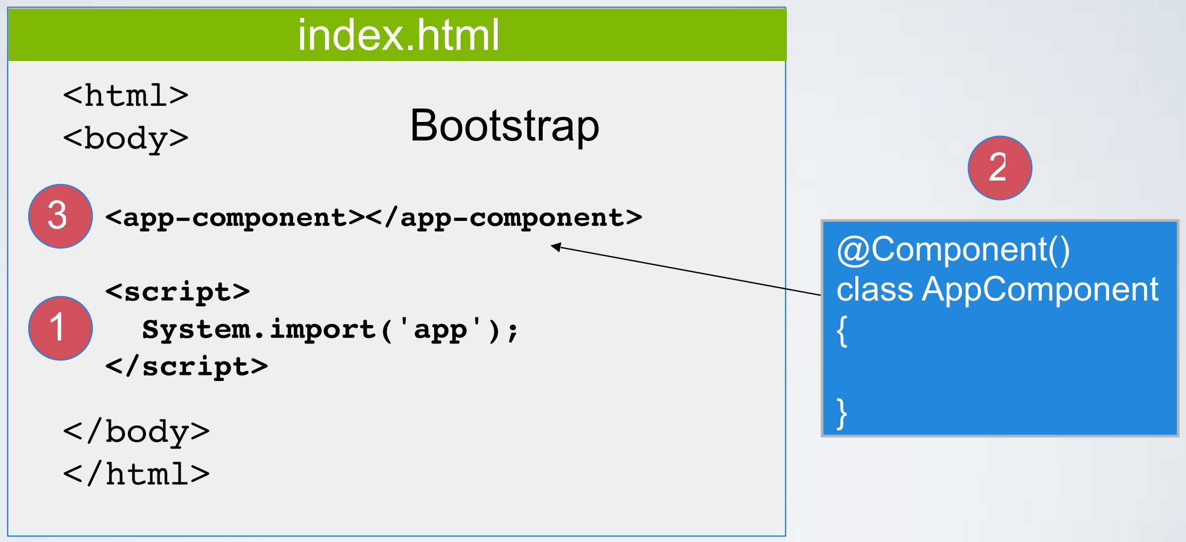
Angular 2 is all about building blocks. So if you're someone who likes to build things this is a fun story to hear about!



ANGULAR 2 BUILDING BLOCKS

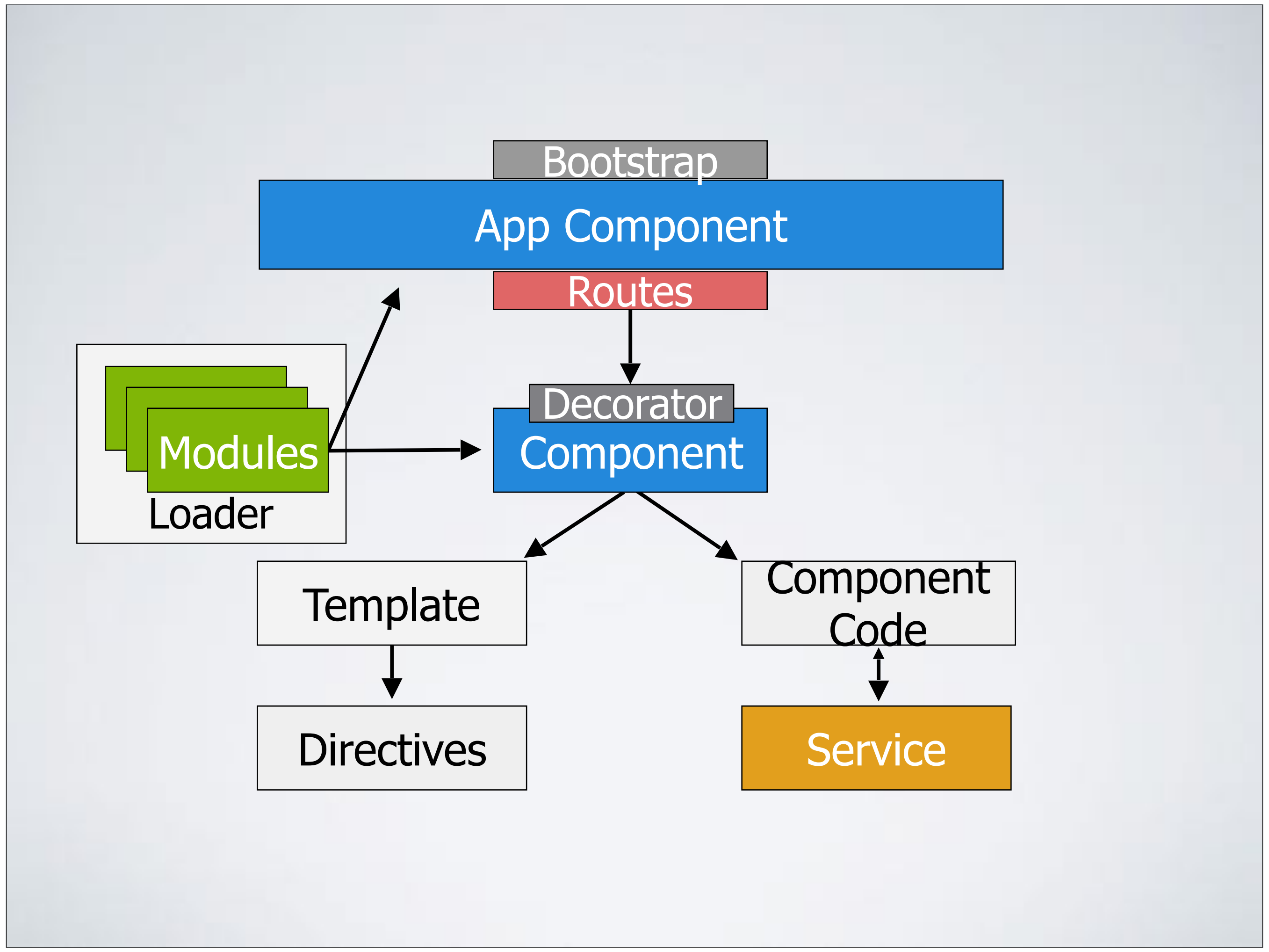
- Modules
- Compone
  nts
- Decorator
  s
- Services

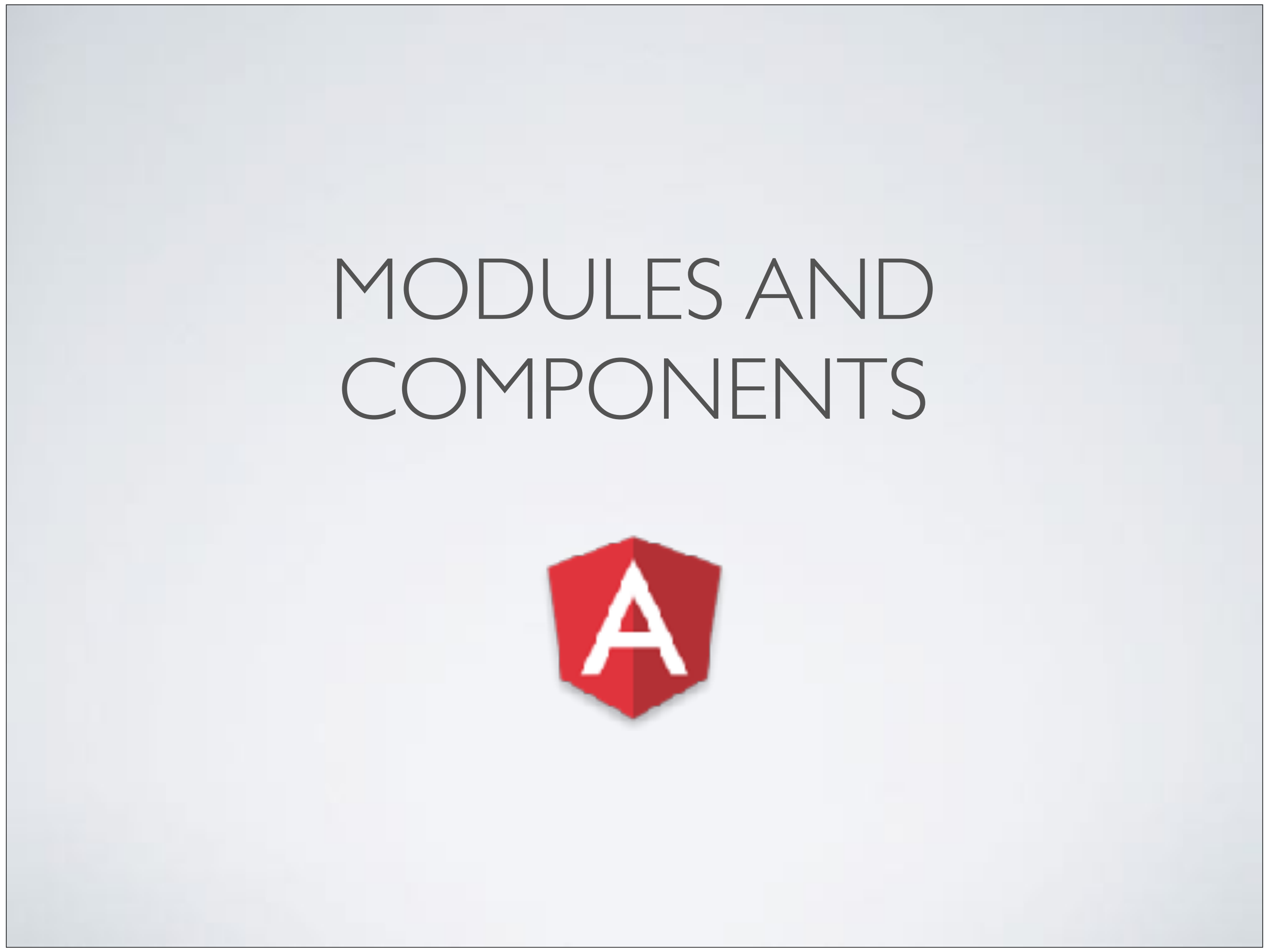
STEPS TO GET STARTED



```
<html>
<body>

<app-component></app-component>

  <script>
    System.import('app');
  </script>

</body>
</html>
```

```
@Component()
class AppComponent
{

}
```


Bootstrap
App Component
Routes
Modules
Loader
Decorator
Component
Template
Component Code
Directives
Service



MODULES AND COMPONENTS

Angular 2 is all about building blocks. So if you're someone who likes to build things this is a fun story to hear about!

WHAT IS A COMPONENT?

- A component is a reusable object

- Made up of: Code | HTML Template
- Has a "selector": <customers></customers>

WHAT'S IN A COMPONENT CLASS?

imports

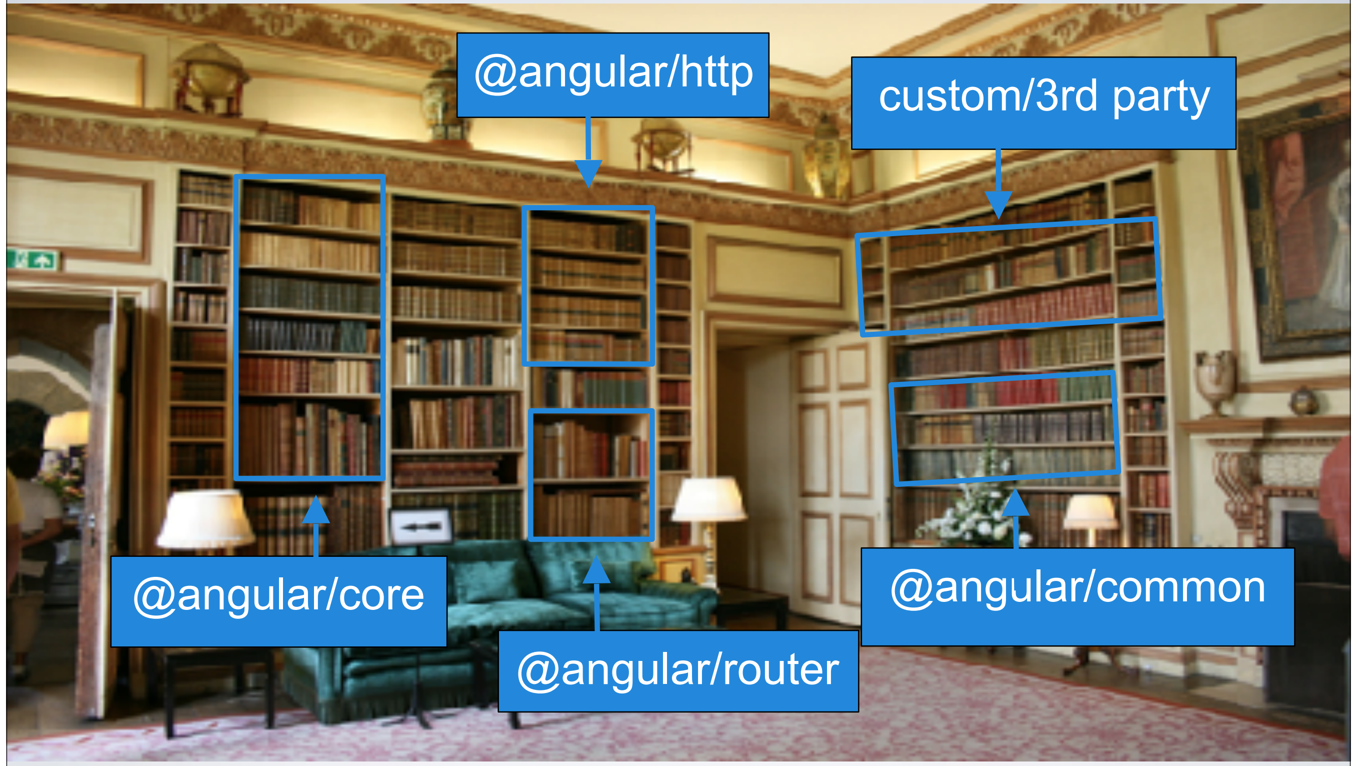
```
import { Component } from '@angular/core';
import { DataService } from '../services/data.service';
```

decorators

```
@Component({
  selector: 'customers',
  templateUrl: 'customers.component.html'
})
```

class

```
export class CustomersComponent {

}
```


@angular/http
custom/3rd party
@angular/core
@angular/common
@angular/router

EXPORTING CUSTOM MODULES

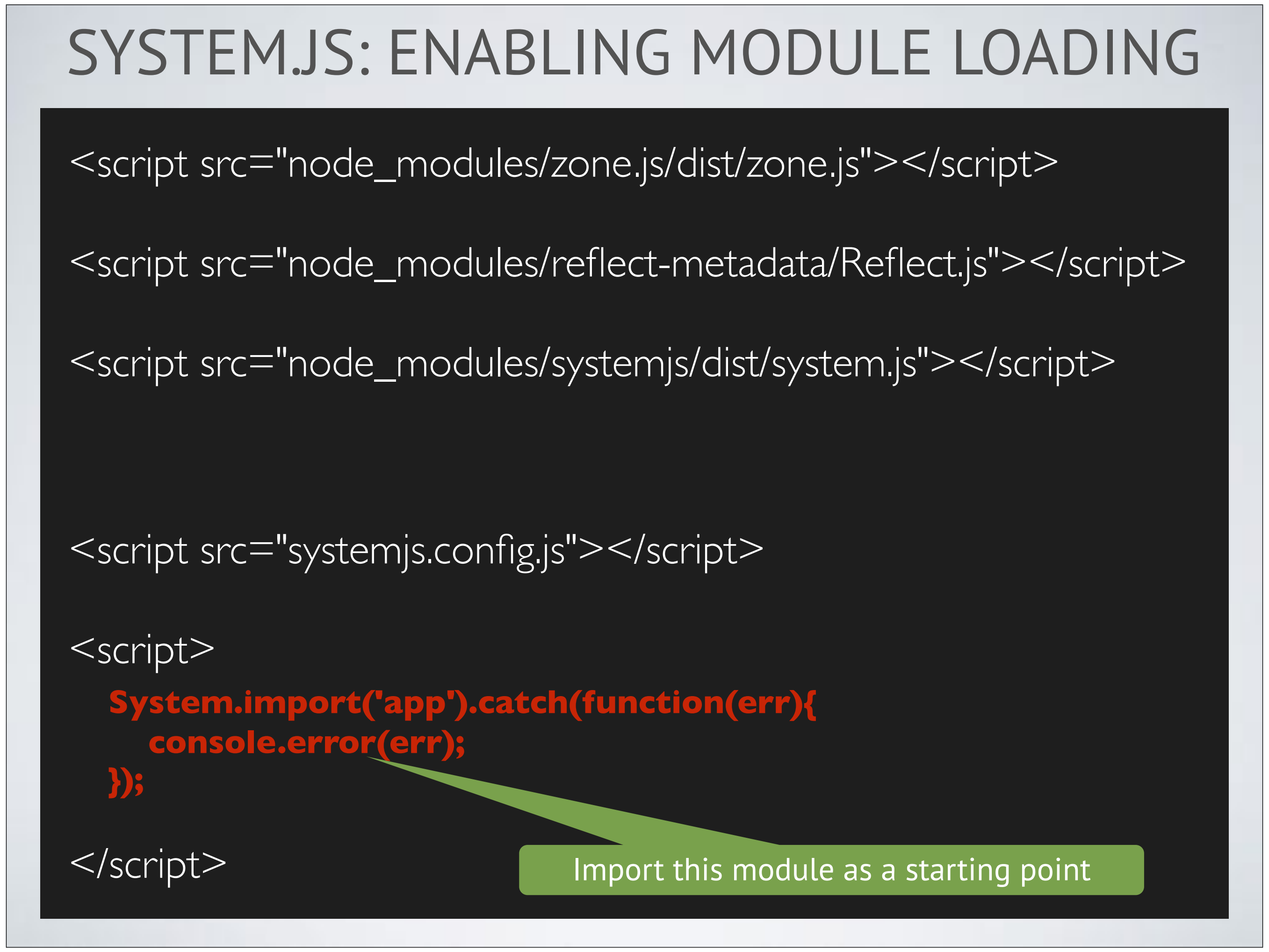
```
export class DataService {
   ...
}
```

IMPORTING MODULES

customers.component.ts

Imported module is relative to current file.

```
import { Component  } from '@angular/core';
import { DataService } from '../shared/services/data.service';
...
```

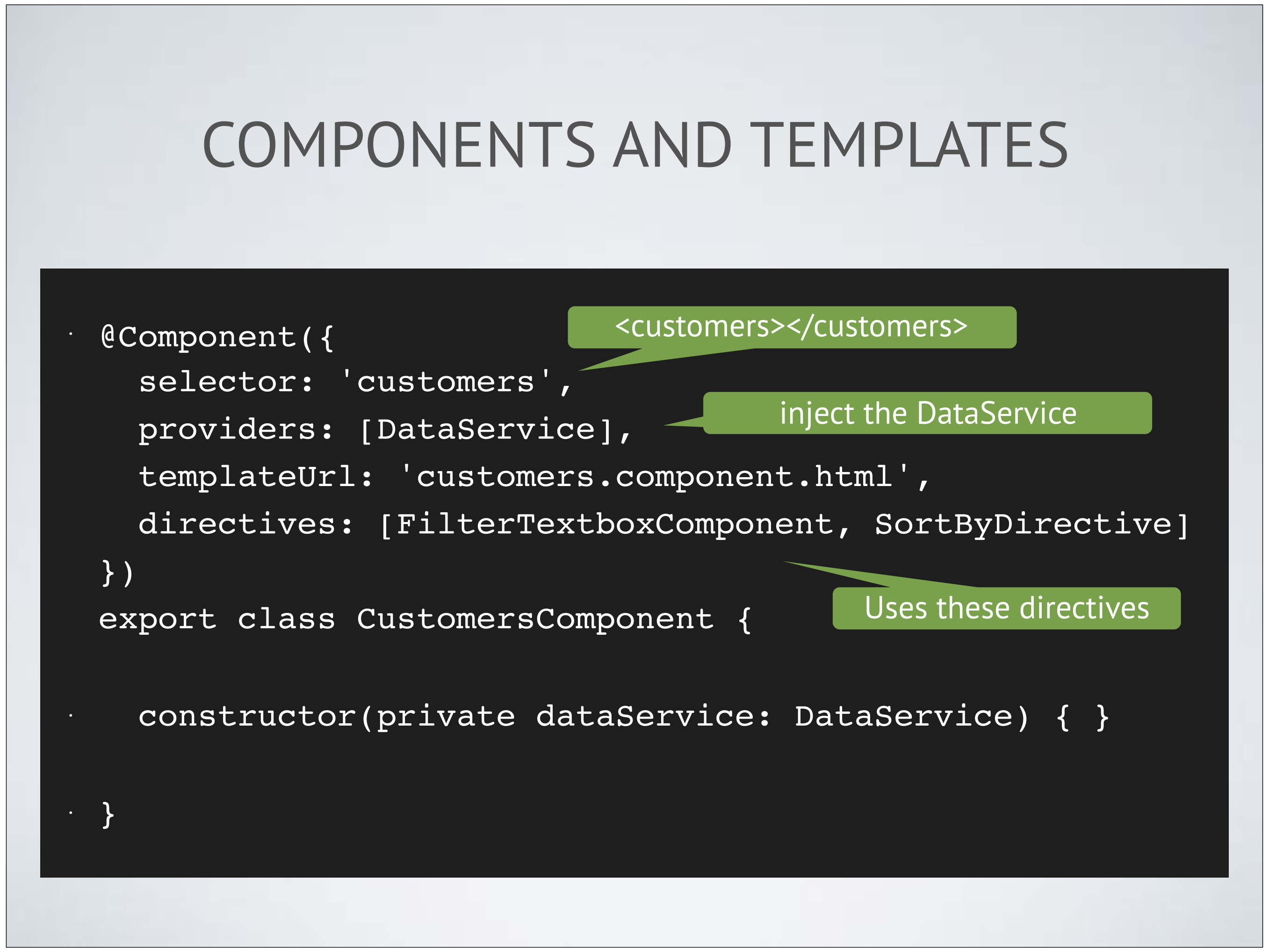


SYSTEM.JS: ENABLING MODULE LOADING

```
<script src="node_modules/zone.js/dist/zone.js"></script>

<script src="node_modules/reflect-metadata/Reflect.js"></script>

<script src="node_modules/systemjs/dist/system.js"></script>



<script src="systemjs.config.js"></script>

<script>
   System.import('app').catch(function(err){
      console.error(err);
   });

</script>
```

Import this module as a starting point

DECLARATIVE TEMPLATE SYNTAX

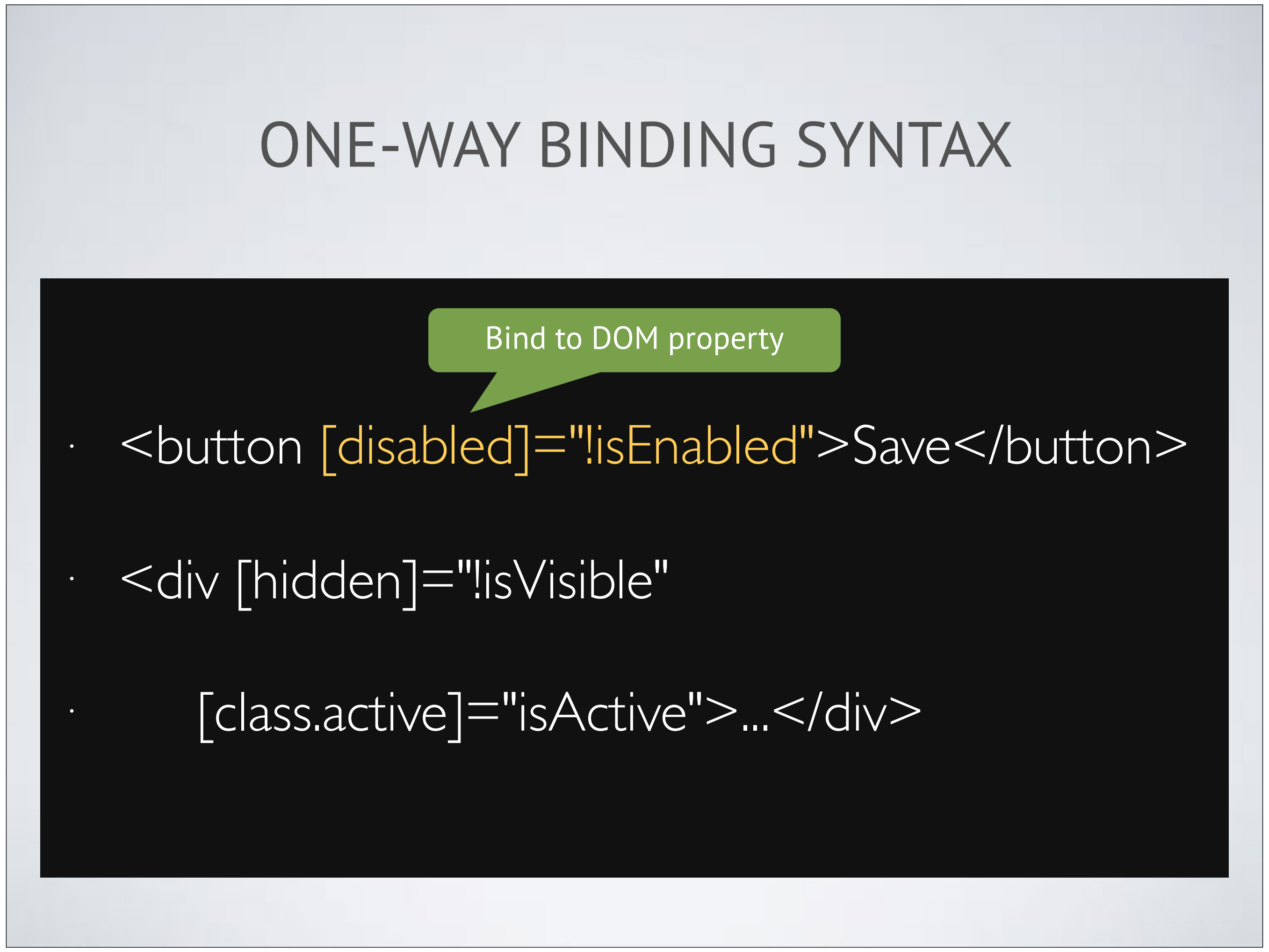


COMPONENTS AND TEMPLATES

```
@Component({
  selector: 'customers',
  providers: [DataService],
  templateUrl: 'customers.component.html',
  directives: [FilterTextboxComponent, SortByDirective]
})
export class CustomersComponent {

  constructor(private dataService: DataService) { }

}
```

<customers></customers>
inject the DataService
Uses these directives

Angular 2 apps are built using components
Provides reuse and consistency
Components rely on decorators to define metadata
Can use directives for rendering in component templates



ONE-WAY BINDING SYNTAX

Bind to DOM property

```
<button [disabled]="!isEnabled">Save</button>

<div [hidden]="!isVisible"

     [class.active]="isActive">...</div>
```

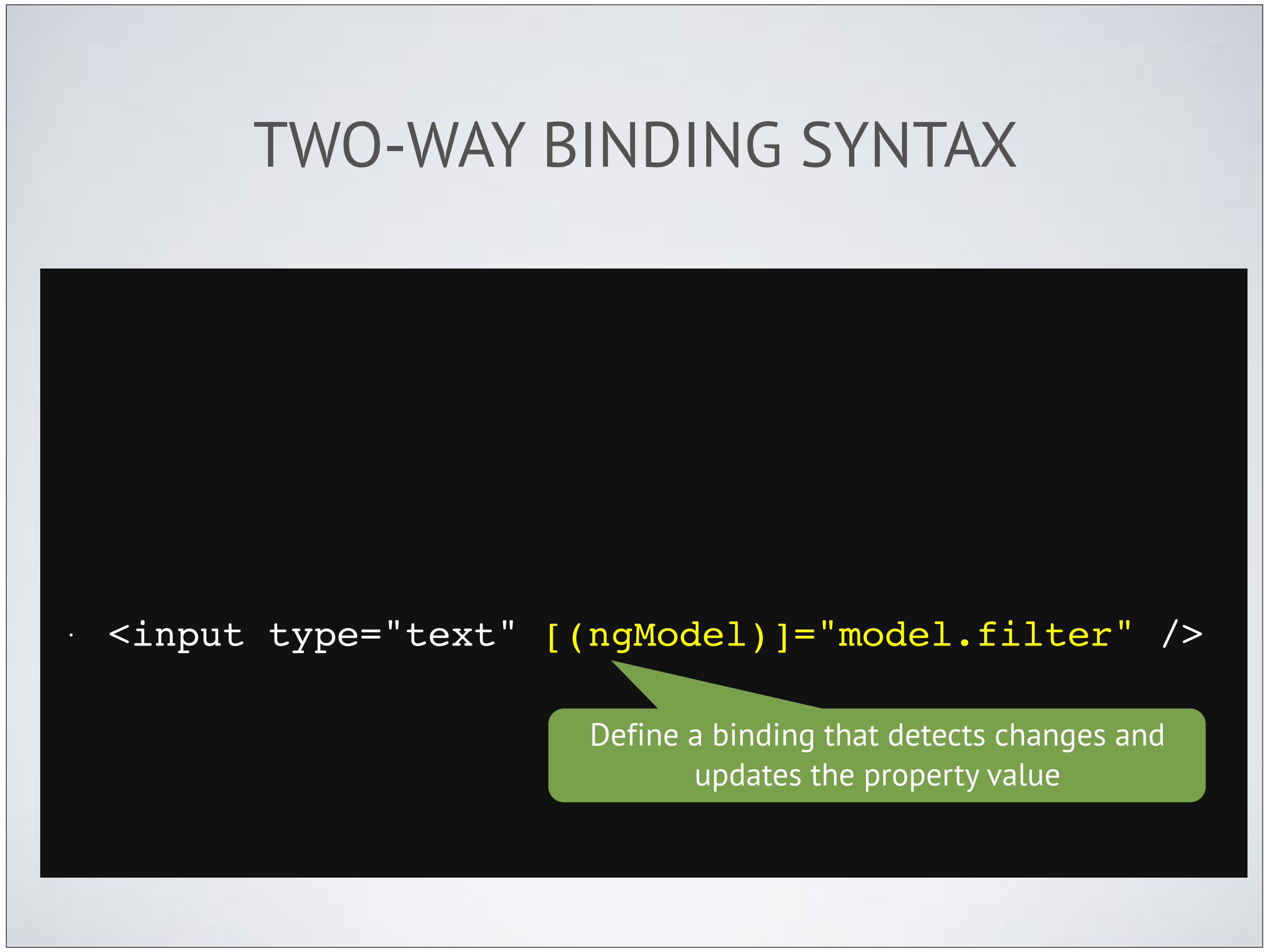
Binding and change detection relies on a tree of components (no cycles)
Create one-way bindings using [property] or bind-property syntax

HANDLING EVENTS

Handle click event

- `<button (click)="save()">Save</button>`
- `<th sort-by="firstName" (sorted)="sort($event)">...</th>`

banana in a box
Angular 2 simplifies working with events and eliminates the need for many directives
Handle an event using the (event) or on-event syntax



TWO-WAY BINDING SYNTAX

```
<input type="text" [(ngModel)]="model.filter" />
```

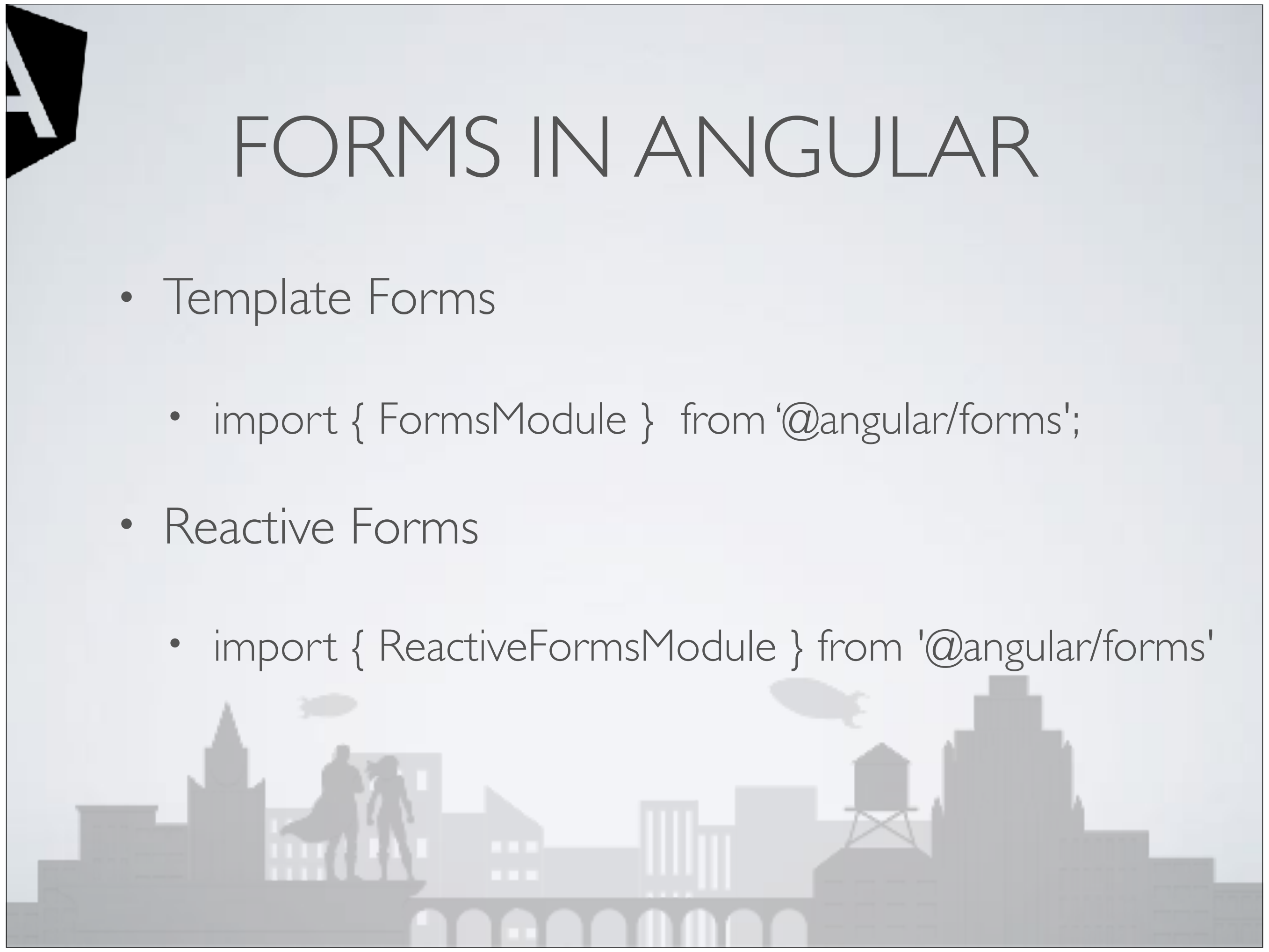
Define a binding that detects changes and updates the property value

Create two-way bindings using [(property)] or bindon-property syntax
Change triggers an event that Angular uses to update the property value

BUILT-IN DIRECTIVES

Angular 2 directive that generates a template

```
<tr *ngFor="let customer of filteredCustomers">
    <td>{{ customer.firstName }}</td>
    <td>{{ customer.lastName }}</td>
</tr>

<div *ngIf="customer">{{ customer.details }}</
div>
```

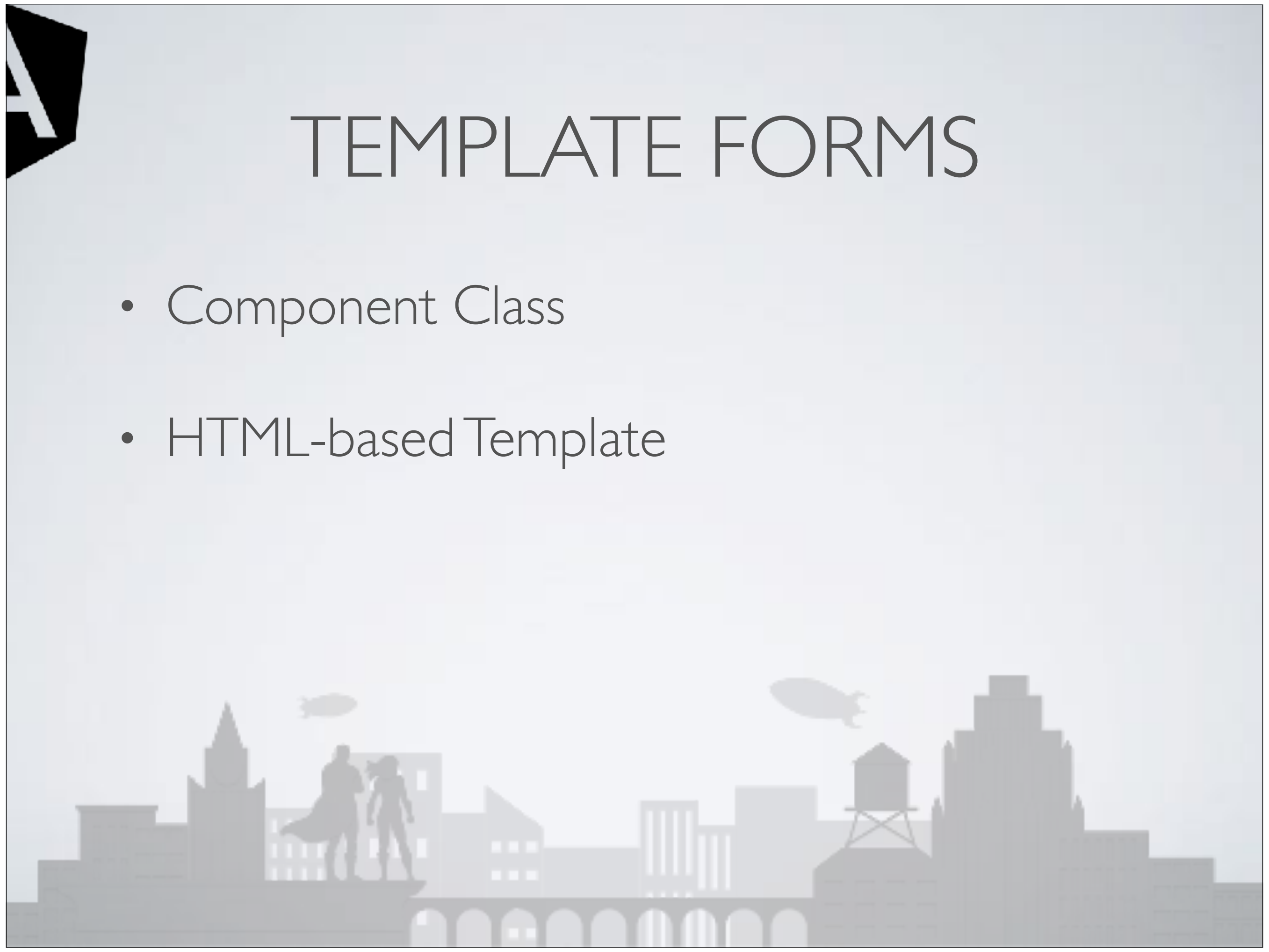


FORMS IN ANGULAR

- Template Forms
  - import { FormsModule } from '@angular/forms';
- Reactive Forms
  - import { ReactiveFormsModule } from '@angular/forms'

Angular offers two form-building technologies: reactive forms and template-driven forms. The two technologies belong to the @angular/forms library and share a common set of form control classes.

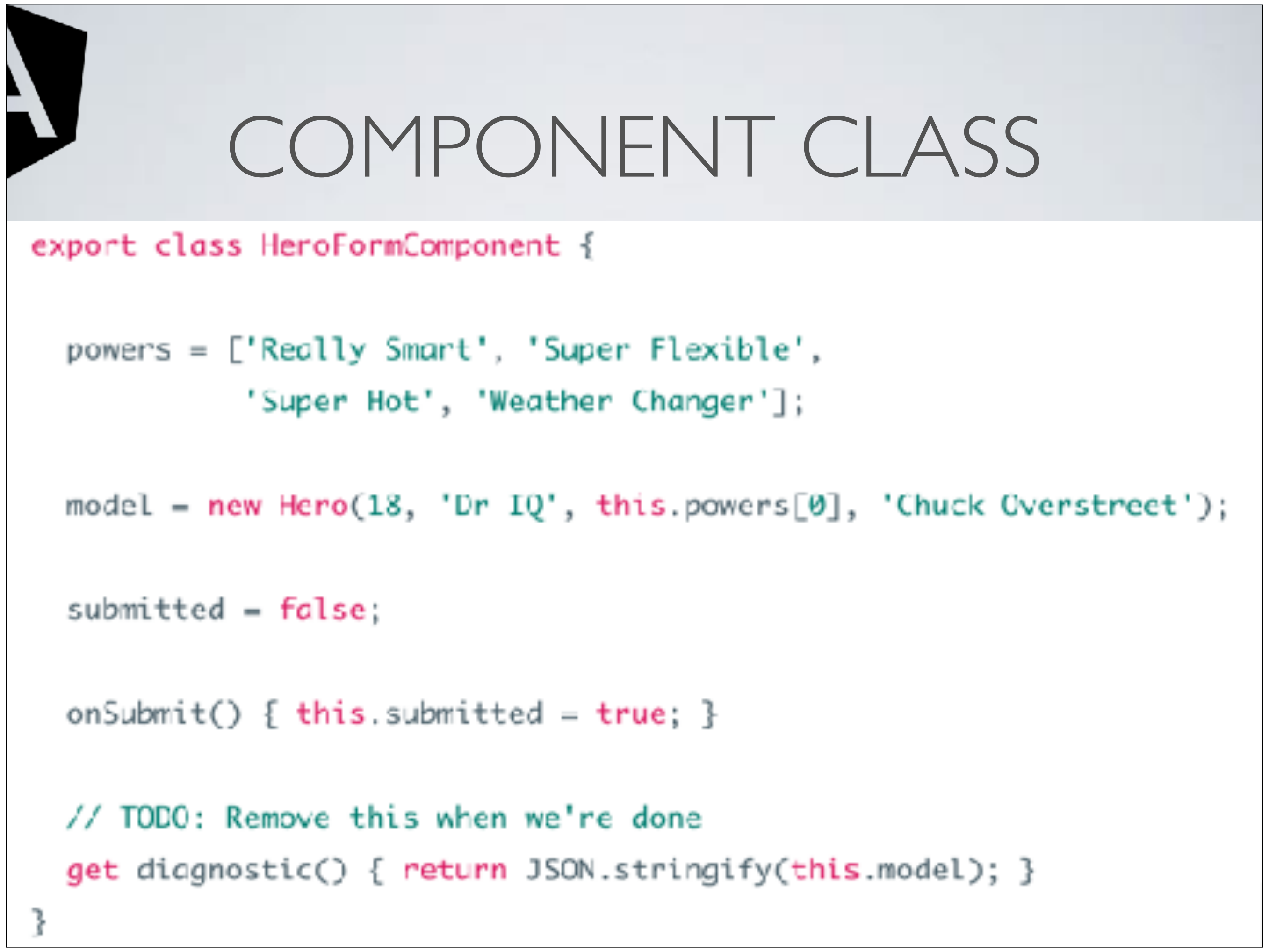
But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the ReactiveFormsModule and the FormsModule.



TEMPLATE FORMS

- Component Class
- HTML-based Template

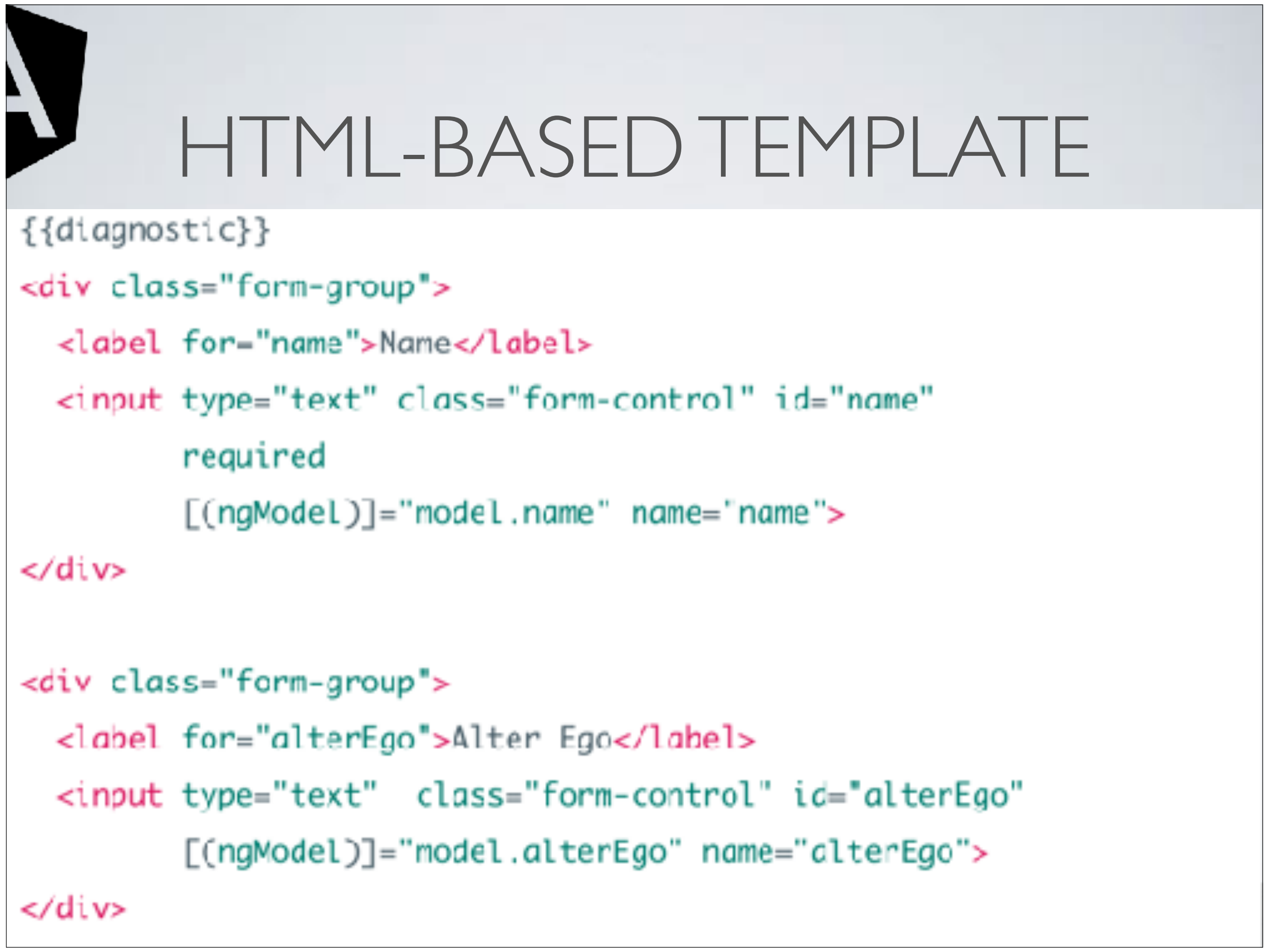
An Angular form has two parts: an HTML-based template and a component class to handle data and user interactions programmatically. We begin with the class because it states, in brief, what our form can do.



COMPONENT CLASS

```
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
            'Super Hot', 'Weather Changer'];

  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');

  submitted = false;

  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```

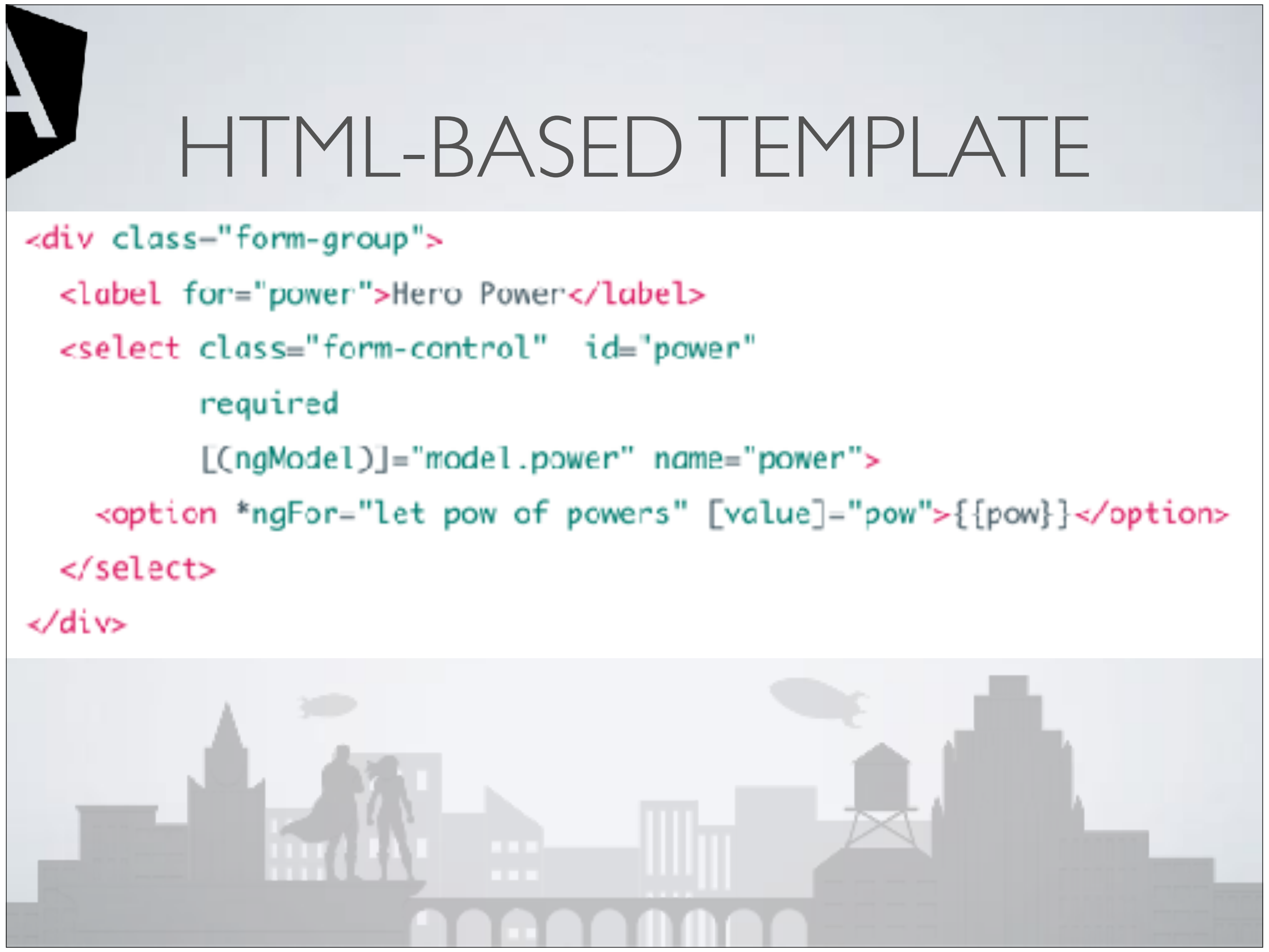
There's nothing special about this component, nothing form-specific, nothing to distinguish it from any component you've written before.



HTML-BASED TEMPLATE

```
{{diagnostic}}
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name"
         required
         [(ngModel)]="model.name"  name="name">
</div>

<div class="form-group">
  <label for="alterEgo">Alter Ego</label>
  <input type="text"  class="form-control" id="alterEgo"
         [(ngModel)]="model.alterEgo" name="alterEgo">
</div>
```

Notice that we also added a name attribute to our <input> tag and set it to "name" which makes sense for the hero's name. Any unique value will do, but using a descriptive name is helpful. Defining a name attribute is a requirement when using [(ngModel)] in combination with a form.

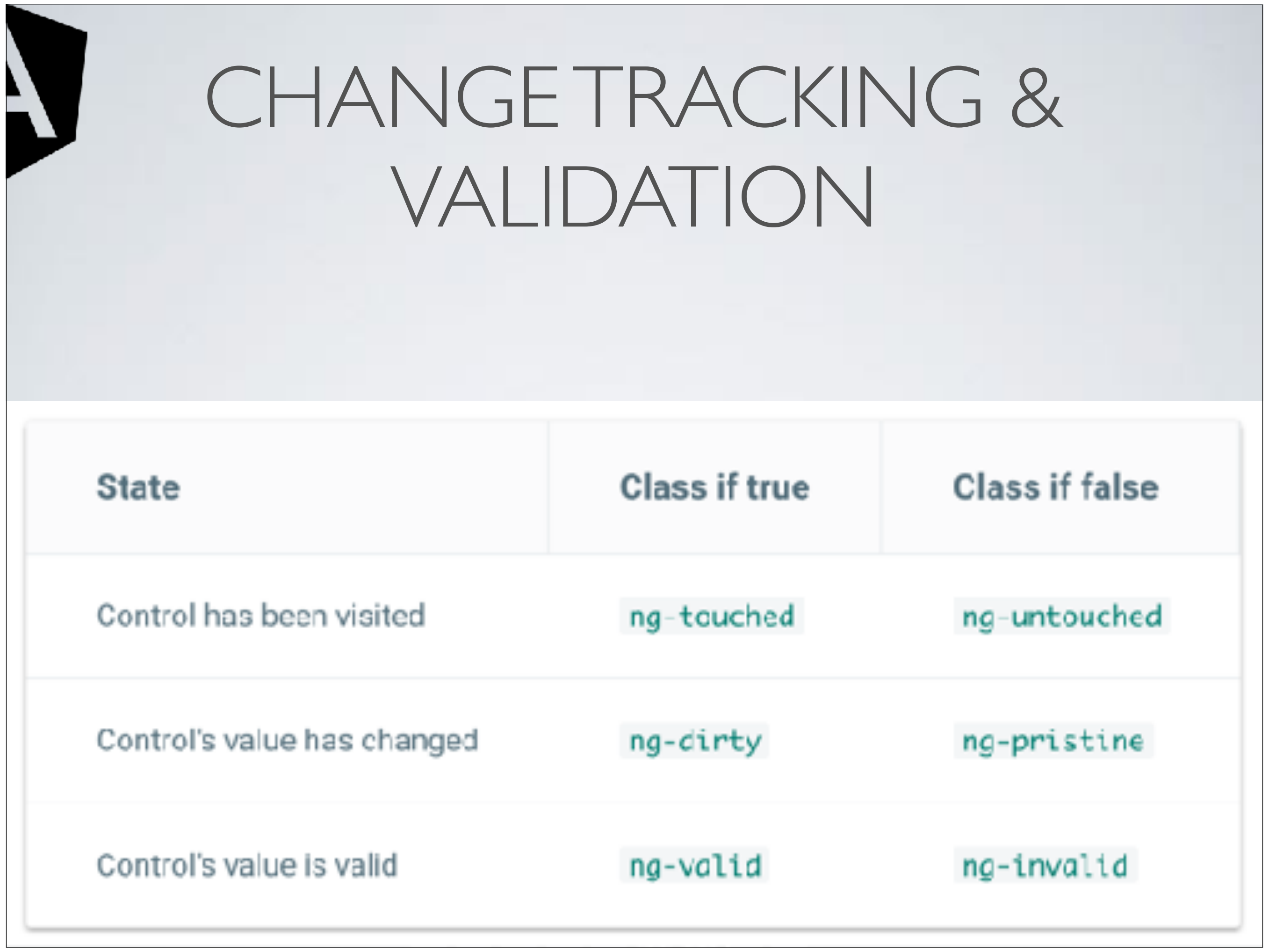


HTML-BASED TEMPLATE

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control"  id="power"
          required
          [(ngModel)]="model.power" name="power">
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

Our hero must choose one super power from a fixed list of Agency-approved powers. We maintain that list internally (in HeroFormComponent).

We'll add a select to our form and bind the options to the powers list using ngFor.



CHANGE TRACKING & VALIDATION

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

A form isn't just about data binding. We'd also like to know the state of the controls in our form.

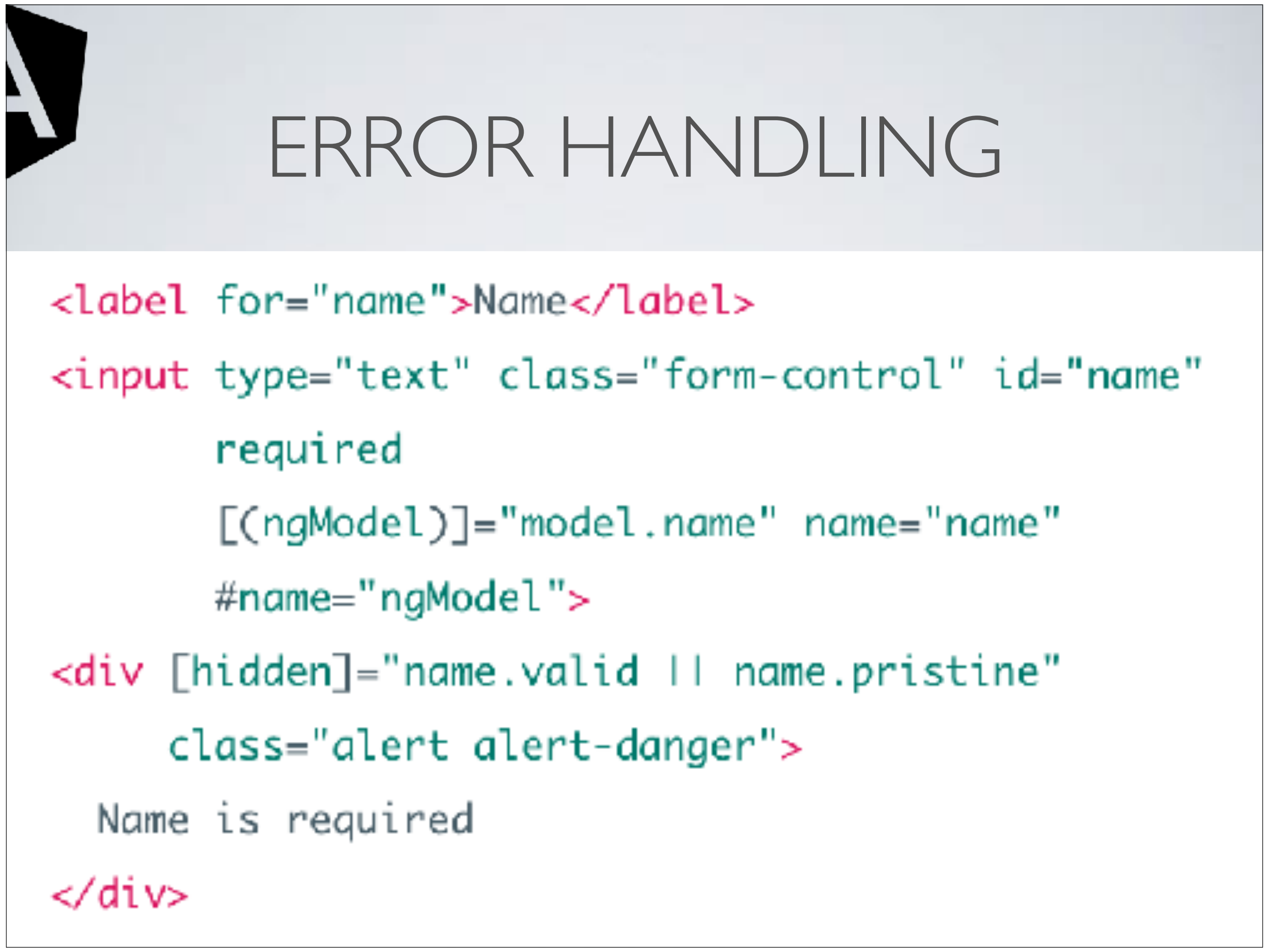
Using ngModel in a form gives us more than just a two way data binding. It also tells us if the user touched the control, if the value changed, or if the value became invalid.

The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. We can leverage those class names to change the appearance of the control.

CHANGE TRACKING & VALIDATION

```css
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}


.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

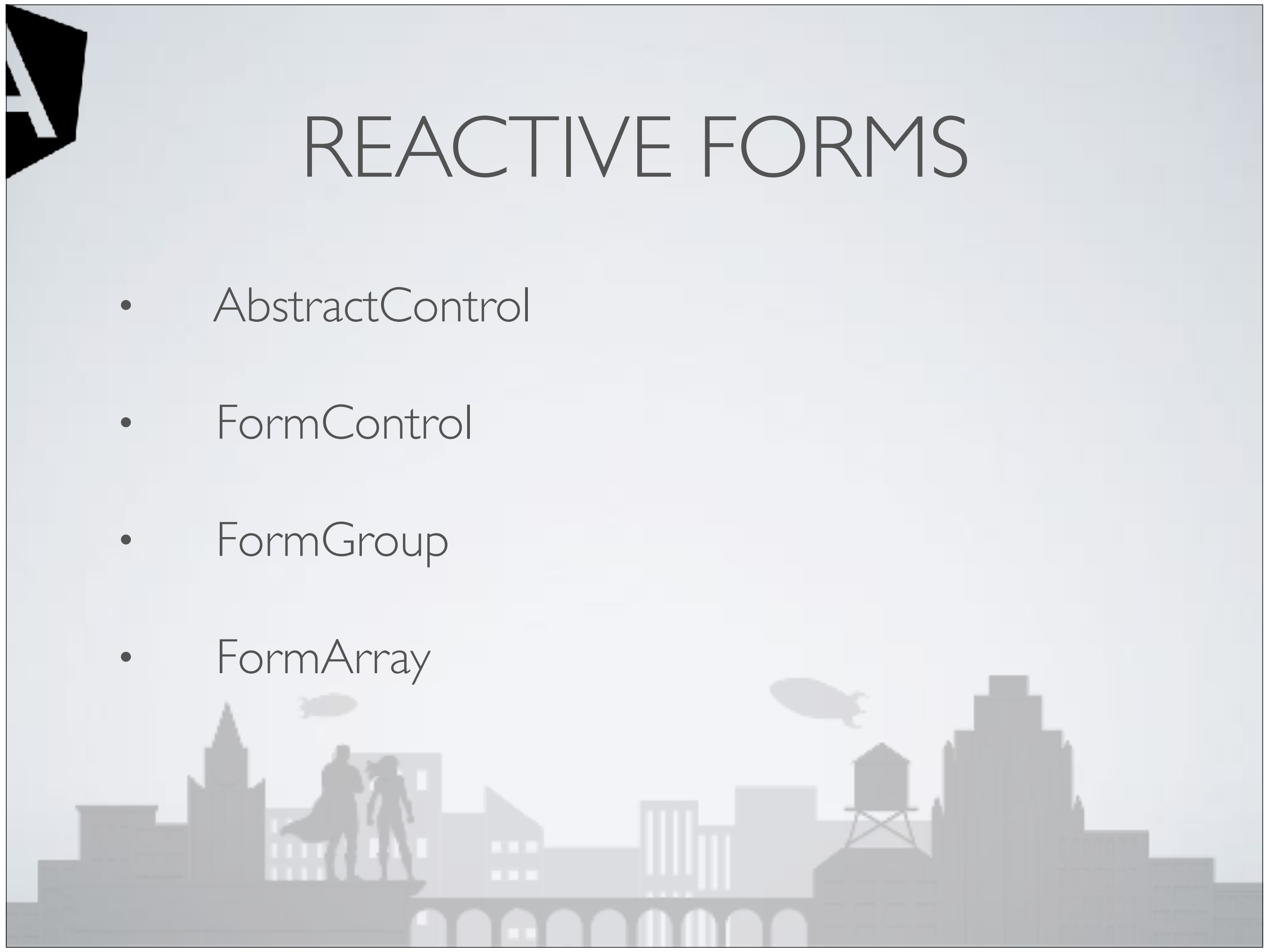
We can use CSS to display a visual indicator of the validity state of the form.



ERROR HANDLING

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
  Name is required
</div>
```

We can do better. The Name input box is required and clearing it turns the bar red. That says something is wrong but we don't know what is wrong or what to do about it.
We can leverage the control's state to reveal a helpful message.
We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called name and gave it the value "ngModel".



REACTIVE FORMS

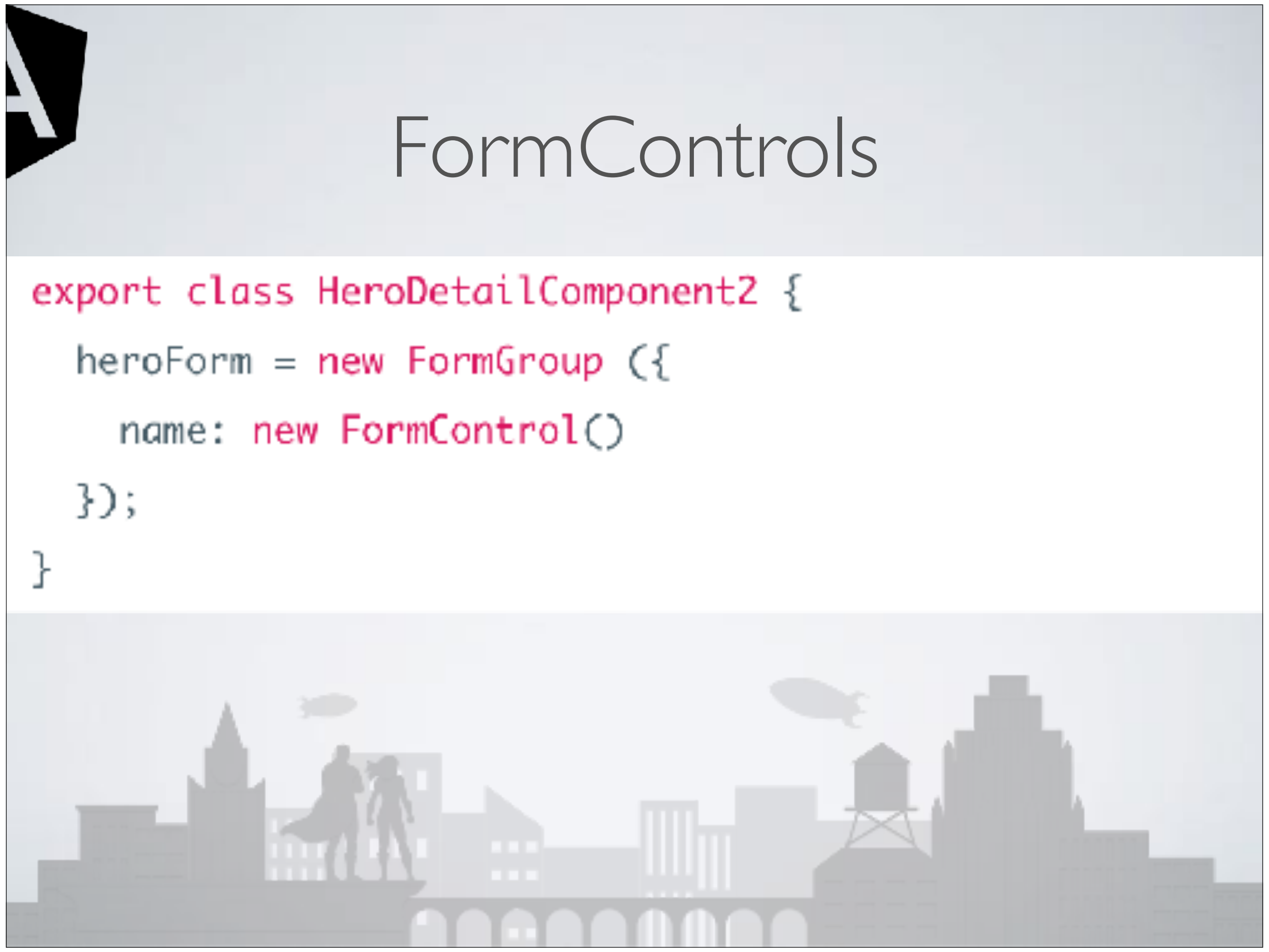
- AbstractControl
- FormControl
- FormGroup
- FormArray

AbstractControl is the abstract base class for the three concrete form control classes: FormControl, FormGroup, and FormArray. It provides their common behaviors and properties, some of which are observable.

FormControl tracks the value and validity status of an individual form control. It corresponds to an HTML form control such as an input box or selector.

FormGroup tracks the value and validity state of a group of AbstractControl instances. The group's properties include its child controls. The top-level form in your component is a FormGroup.

FormArray tracks the value and validity state of a numerically indexed array of AbstractControl instances.



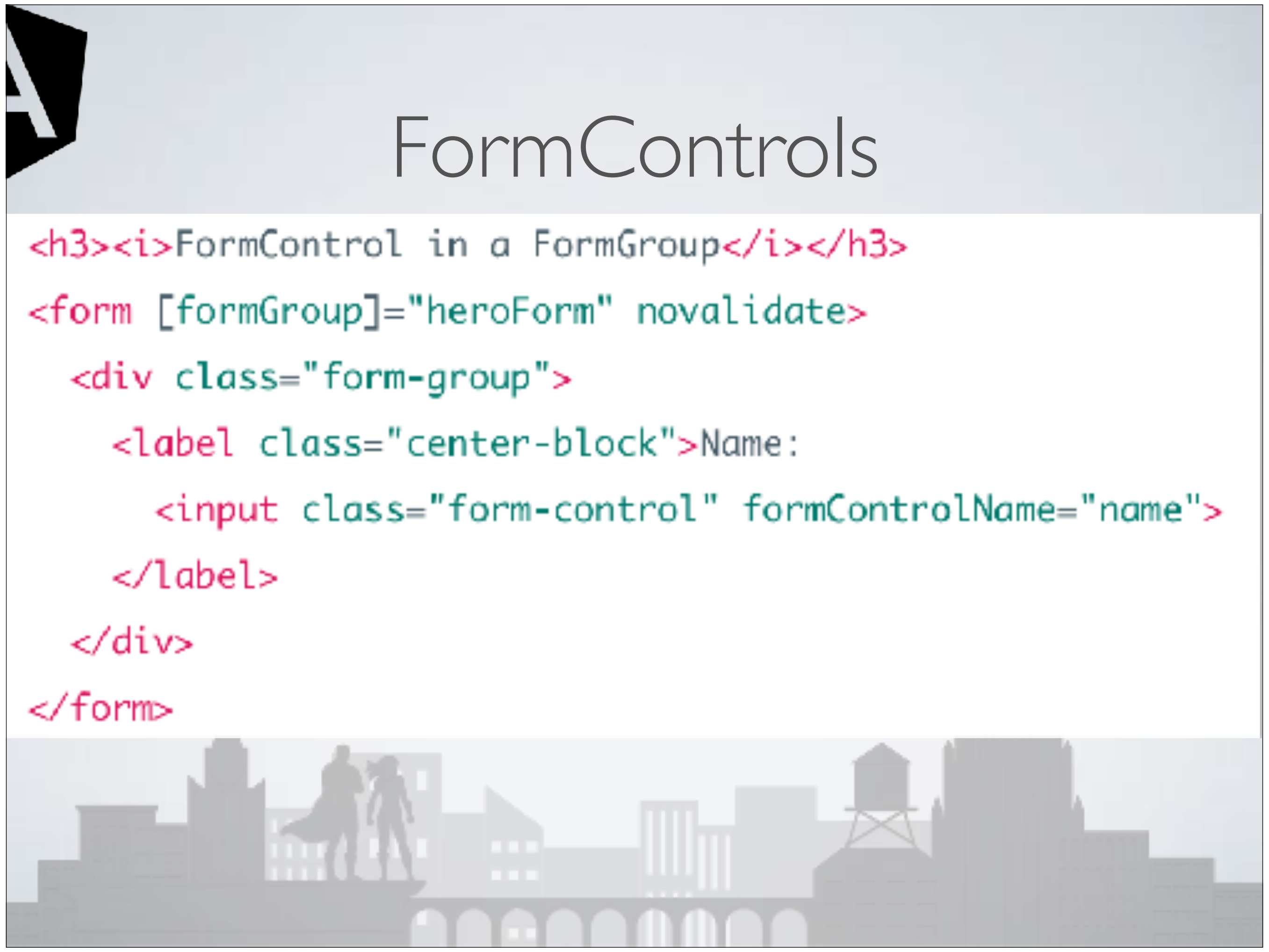
FormControls

```
export class HeroDetailComponent2 {
  heroForm = new FormGroup ({
    name: new FormControl()
  });
}
```

Here you are creating a FormControl called name. It will be bound in the template to an HTML input box for the hero name.

A FormControl constructor accepts three, optional arguments: the initial data value, an array of validators, and an array of async validators.

This simple control doesn't have data or validators. In real apps, most form controls have both.

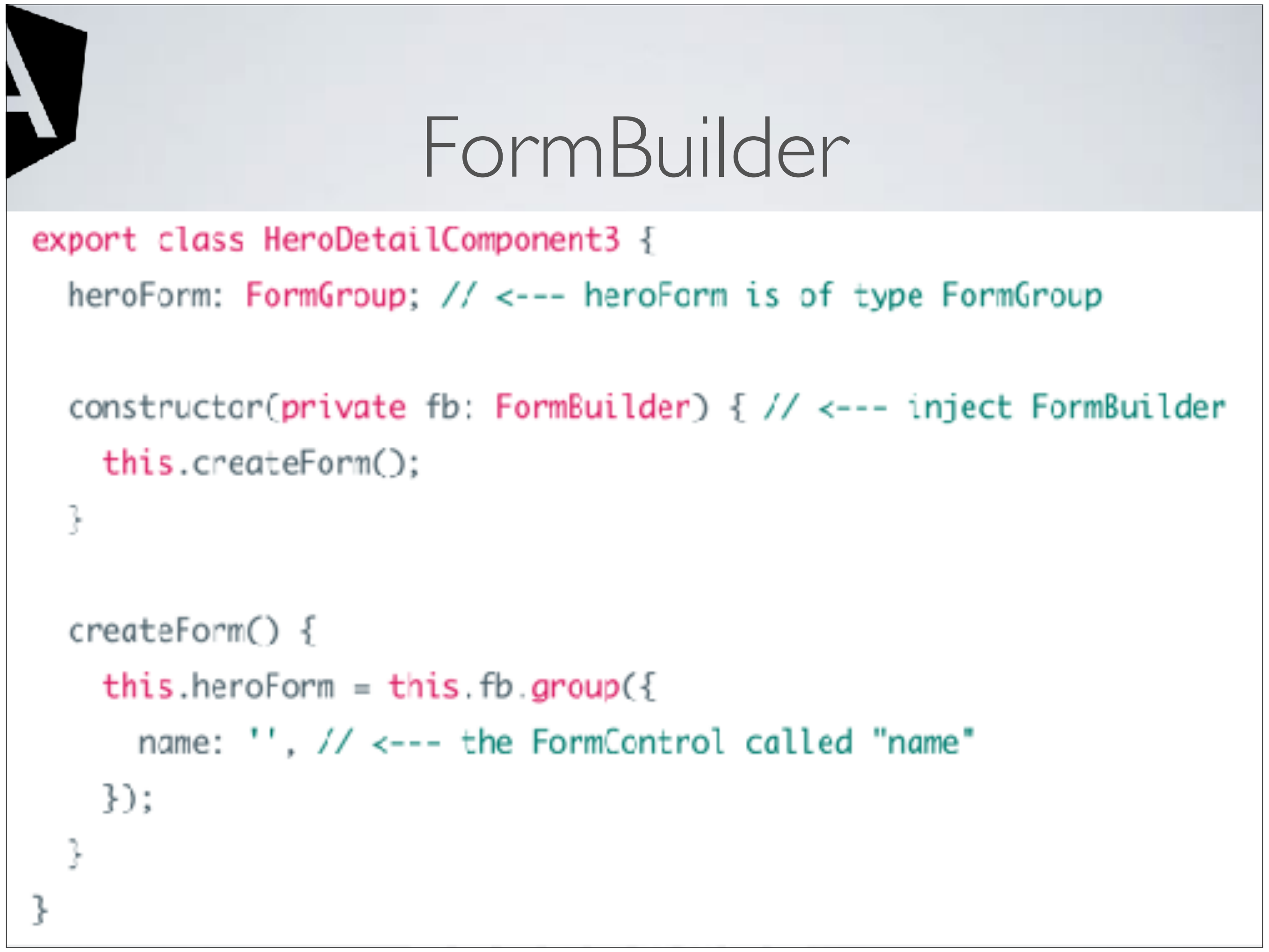


FormControls

```
<h3><i>FormControl in a FormGroup</i></h3>
<form [formGroup]="heroForm" novalidate>
  <div class="form-group">
    <label class="center-block">Name:
      <input class="form-control" formControlName="name">
    </label>
  </div>
</form>
```

Usually, if you have multiple FormControls, you'll want to register them within a parent FormGroup. This is simple to do.
Notice that now the single input is in a form element. The novalidate attribute in the <form> element prevents the browser from attempting native HTML validations.

formGroup is a reactive form directive that takes an existing FormGroup instance and associates it with an HTML element. In this case, it will associate the FormGroup you saved as heroForm with the form element.

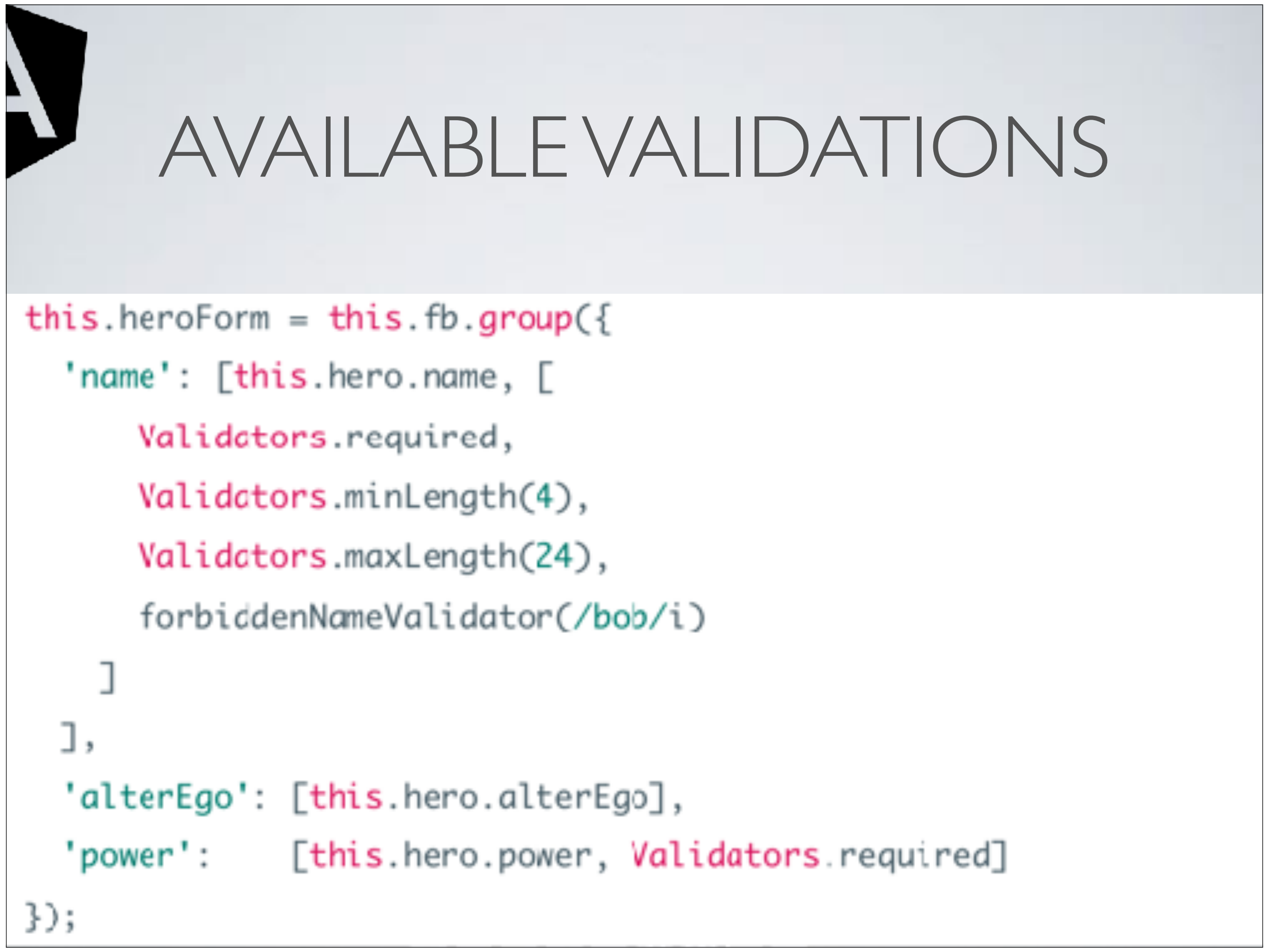


FormBuilder

```
export class HeroDetailComponent3 {
  heroForm: FormGroup; // <--- heroForm is of type FormGroup

  constructor(private fb: FormBuilder) { // <--- inject FormBuilder
    this.createForm();
  }

  createForm() {
    this.heroForm = this.fb.group({
      name: '', // <--- the FormControl called "name"
    });
  }
}
```

The FormBuilder class helps reduce repetition and clutter by handling details of control creation for you.
FormBuilder.group is a factory method that creates a FormGroup.   FormBuilder.group takes an object whose keys and values are FormControl names and their definitions. In this example, the name control is defined by its initial data value, an empty string.

Defining a group of controls in a single object makes for a compact, readable style. It beats writing an equivalent series of new FormControl(...) statements.



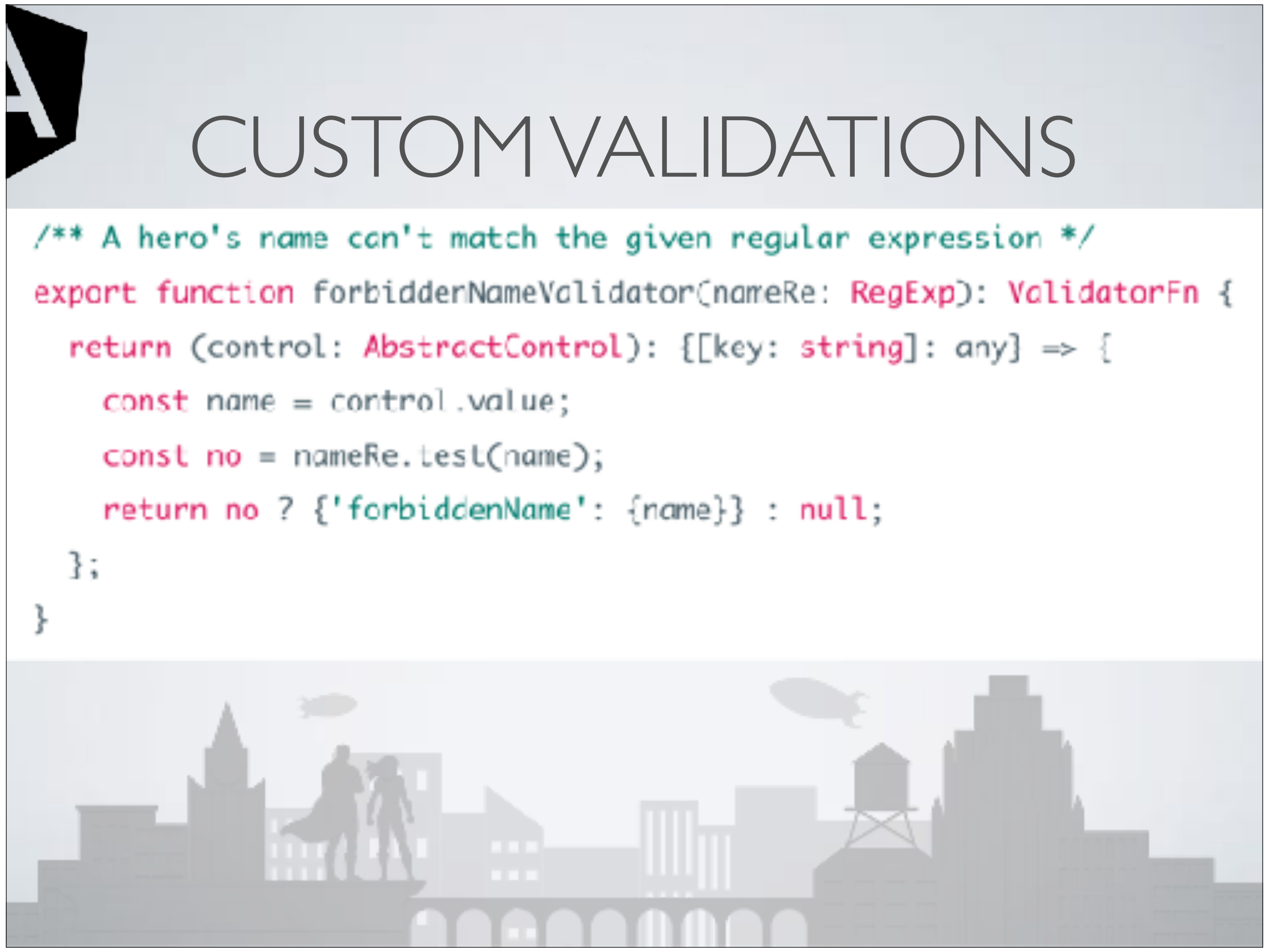
AVAILABLE VALIDATIONS

```
this.heroForm = this.fb.group({
  'name': [this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      Validators.maxLength(24),
      forbiddenNameValidator(/bob/i)
    ]
  ],
  'alterEgo': [this.hero.alterEgo],
  'power':    [this.hero.power, Validators.required]
});
```

The FormBuilder declaration object specifies the three controls of the sample's hero form.

Each control spec is a control name with an array value. The first array element is the current value of the corresponding hero field. The (optional) second value is a validator function or an array of validator functions.

Most of the validator functions are stock validators provided by Angular as static methods of the Validators class. Angular has stock validators that correspond to the standard HTML validation attributes.



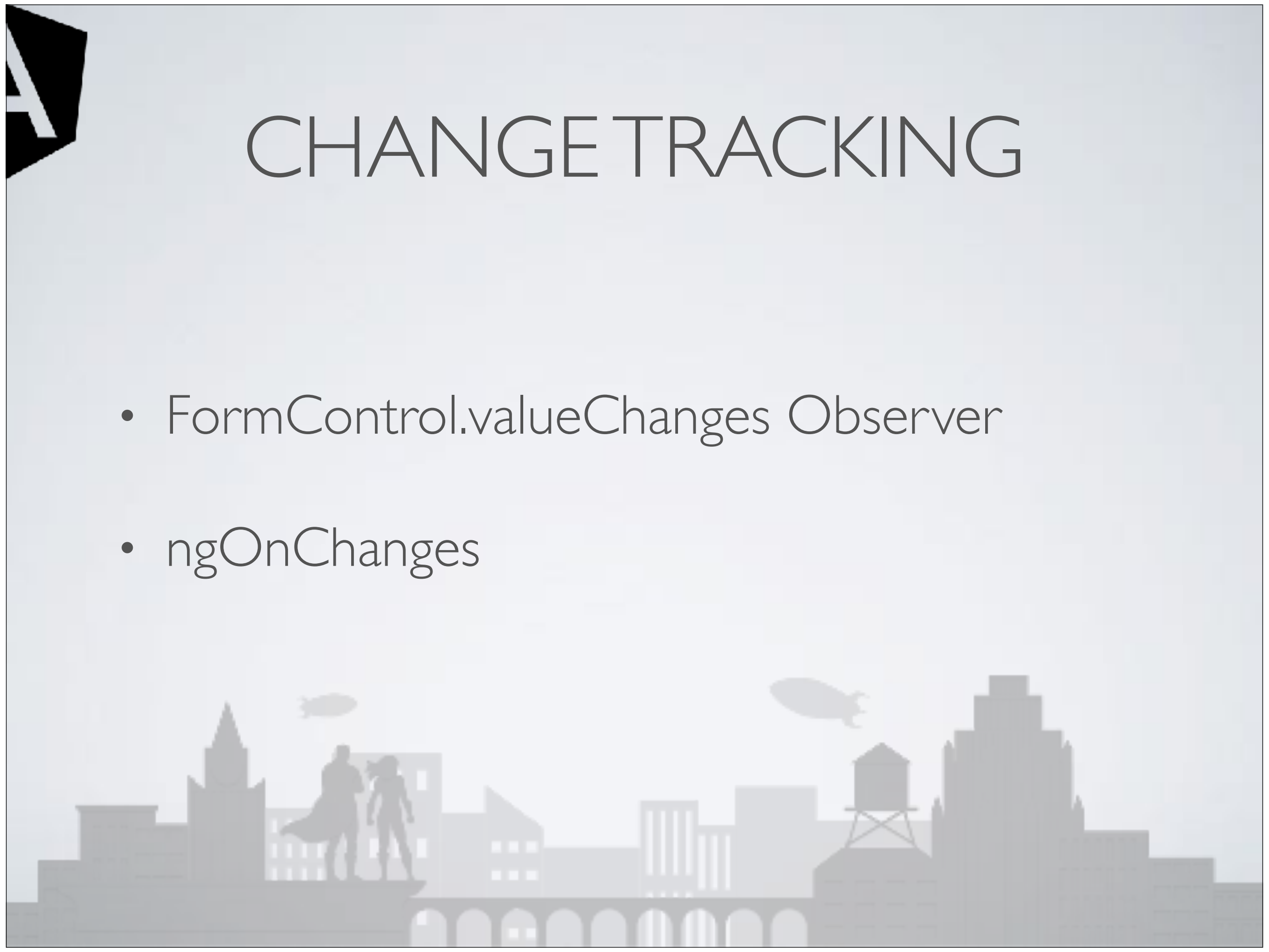
CUSTOM VALIDATIONS

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} => {
    const name = control.value;
    const no = nameRe.test(name);
    return no ? {'forbiddenName': {name}} : null;
  };
}
```

The function is actually a factory that takes a regular expression to detect a specific forbidden name and returns a validator function.

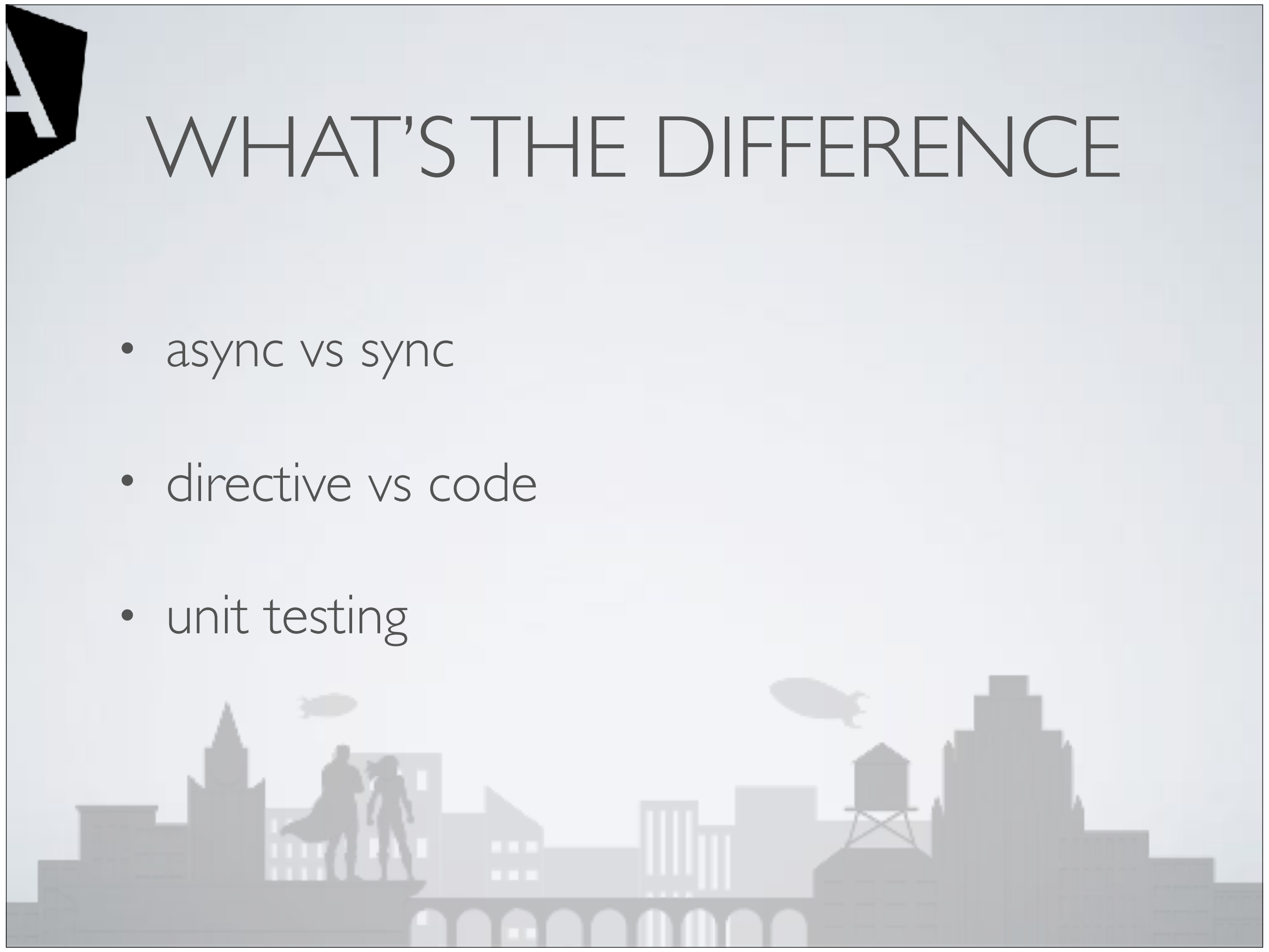
In this sample, the forbidden name is "bob"; the validator rejects any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The forbiddenNamevalidator factory returns the configured validator function. That function takes an Angular control object and returns either null if the control value is valid or a validation error object. The validation error object typically has a property whose name is the validation key ('forbiddenName') and whose value is an arbitrary dictionary of values that we could insert into an error message ({name}).



CHANGE TRACKING

- FormControl.valueChanges Observer
- ngOnChanges



WHAT'S THE DIFFERENCE

- async vs sync
- directive vs code
- unit testing

In reactive forms, you create the entire form control tree in code. You can immediately update a value or drill down through the descendants of the parent form because all controls are always available.
Template-driven forms delegate creation of their form controls to directives. To avoid "changed after checked" errors, these directives take more than one cycle to build the entire control tree. That means you must wait a tick before manipulating any of the controls from within the component class.

The asynchrony of template-driven forms also complicates unit testing. You must wrap your test block in async() or fakeAsync() to avoid looking for values in the form that aren't there yet. With reactive forms, everything is available when you expect it to be.

Which is better, reactive or template-driven?
Neither is "better". They're two different architectural paradigms, with their own strengths and weaknesses. Choose the approach that works best for you. You may decide to use both in the same application.

MULTUMESC!

@Lakatos88

Alex Lakatos