Let's Learn Angular!

with companion codelab

Keith Hall
2016-10-14

What is Angular?

***Angular** is a development platform for building mobile and desktop applications. Angular lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's binding and dependency injection eliminate much of the code you would otherwise have to write.*

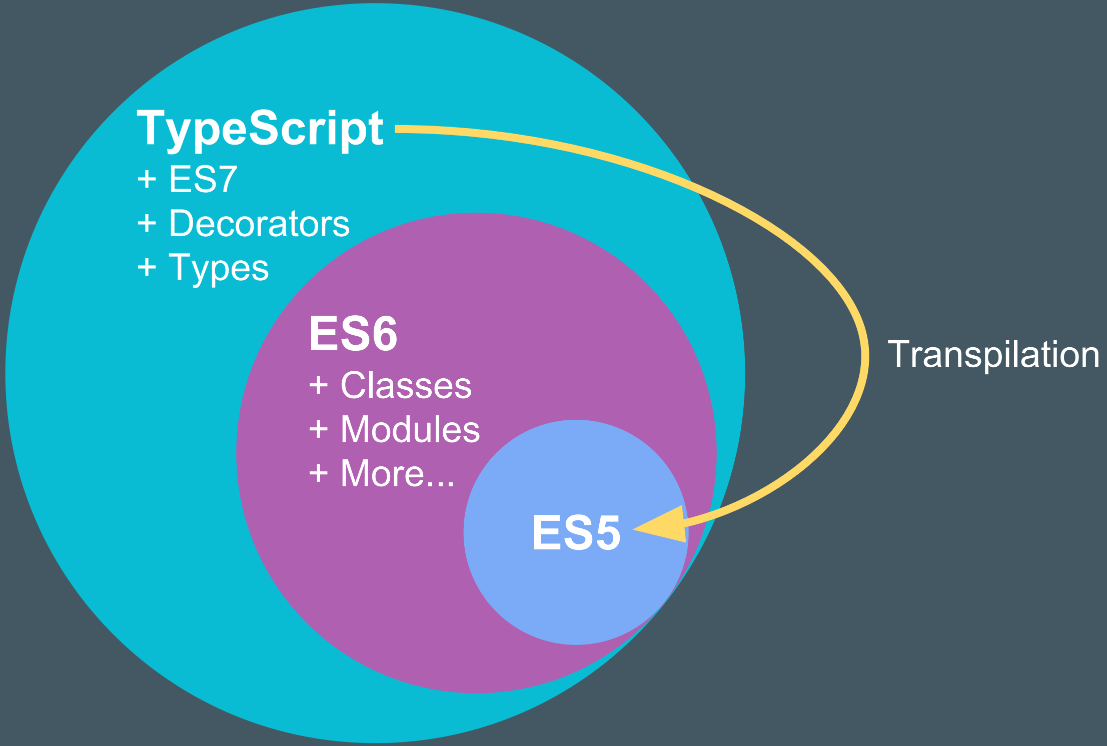
What is Angular?

Angular *is a development platform for building mobile and desktop applications. Angular lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's binding and dependency injection eliminate much of the code you would otherwise have to write.*

Platform Benefits

PWA	Native	Desktop
Optimization	Universal	Code Splitting
Templates	CLI	IDE Support
Testing	Animation	Accessibility

Why TypeScript?

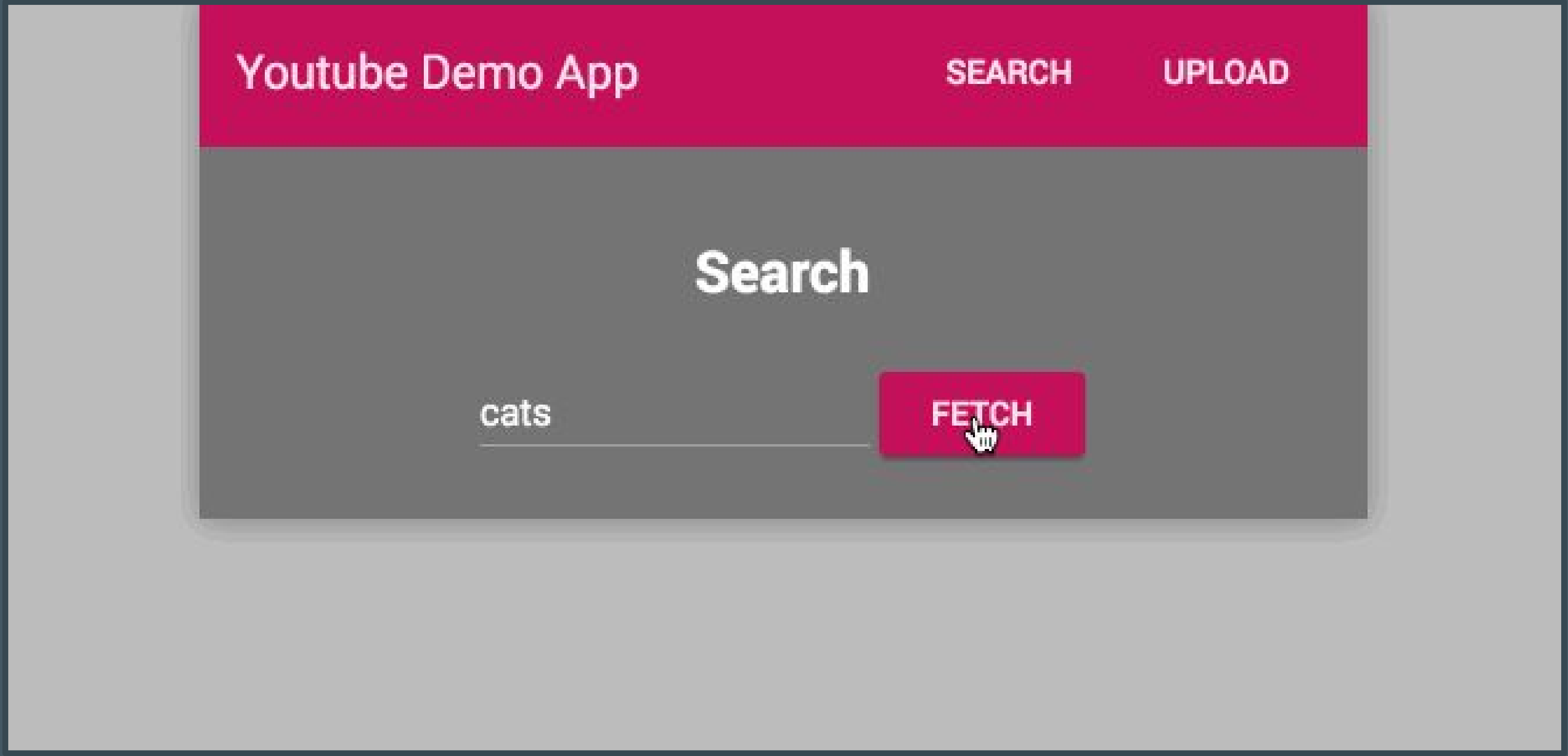


Playground: typescriptlang.org/play

Codelab

goo.gl/fIfx7z

So what are we going to build?



Content

Bootstrapping your application

Component Anatomy - Inline Template

@Component decorator ----►
Selector ----►
Template ----►
Class declaration ----►

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

@Component({
  selector: 'hello-world',
  template: '<h1>Hello World!</h1>',
})
export class HelloWorldComponent {}
```

hello-world.component.ts

Bootstrapping your application

Component Anatomy - Template File

@Component decorator ----►

Selector ----►

Template ----►

Class declaration ----►

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

@Component({
  moduleId: module.id,
  selector: 'hello-world',
  templateUrl: './hello-world.component.html',
})
export class HelloWorldComponent {}
```

hello-world.component.ts

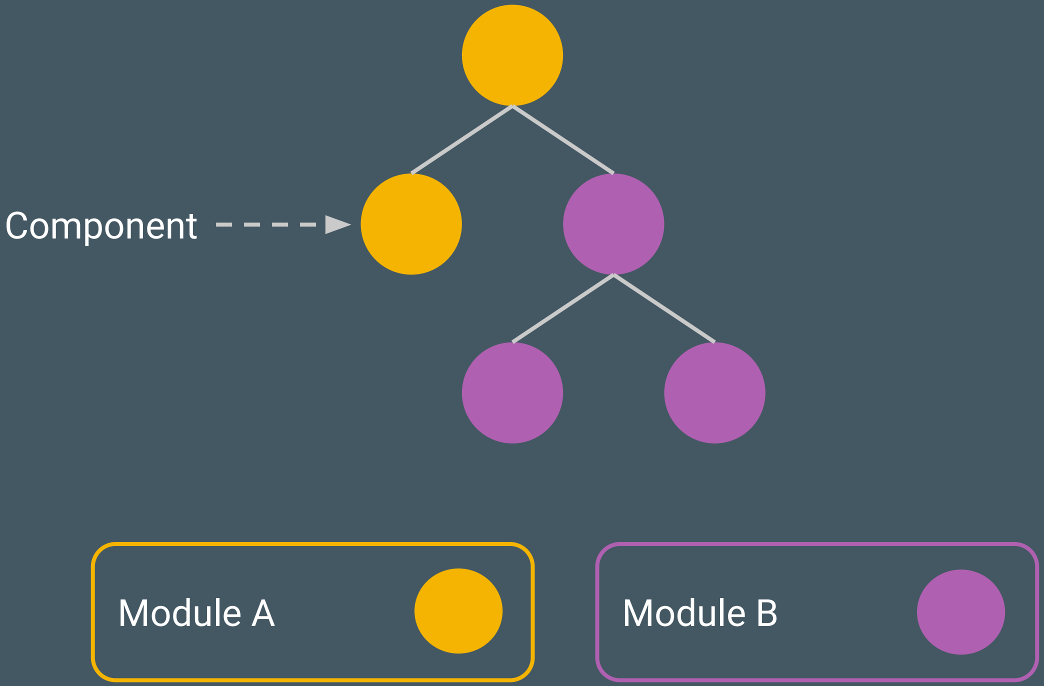
Required for relative imports for some module loaders, e.g. SystemJS.

```
<h1>Hello World!</h1>
```

hello-world.component.html

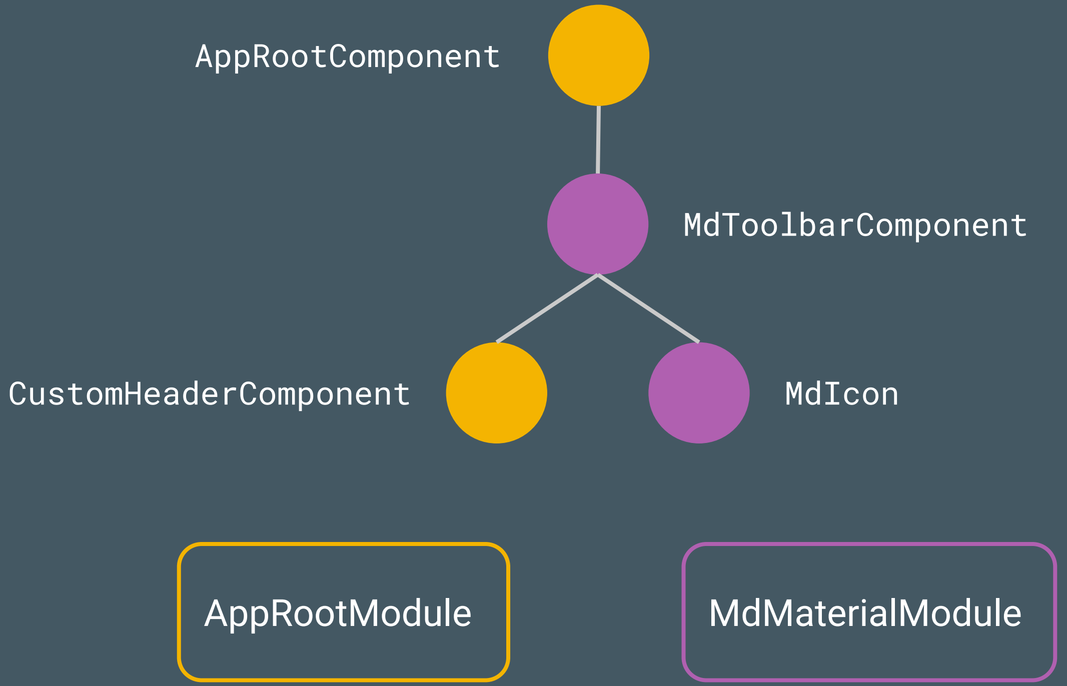
Bootstrapping your application

Modules and Component Trees



Bootstrapping your application

Example: Using Modules



Bootstrapping your application

Module Anatomy

@NgModule decorator ----►
Components to include ----►
Root component(s) ----►
Class declaration ----►

```
import { NgModule } from '@angular/core';
import { HelloWorldComponent }
    from './hello-world.component';

@NgModule({
  declarations: [ HelloWorldComponent ],
  bootstrap: [ HelloWorldComponent ],
})
export class AppModule {}
```

app.module.ts

Bootstrapping your application

Module Bootstrapping

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

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

main.ts

```
<body>
  <!-- Typically our “root” component has the selector
       <app-root></app-root>. We're using <hello-world></hello-word>
       as tradition dictates. -->
  <hello-world>Loading...</hello-world>
  <script>...</script>
</body>
```

index.html

Bootstrapping your application

Module Imports and Exports

```
import { NgModule } from '@angular/core';
import { CarouselComponent }
    from './carousel.component';

@NgModule({
  declarations: [ CarouselComponent ],
  exports: [ CarouselComponent ],
})
export class CarouselModule {}
```

../vendor/carousel.module.ts

```
import { NgModule } from '@angular/core';
import { HelloWorldComponent }
    from './hello-world.component';
import { CarouselModule }
    from '../vendor/carousel.module';

@NgModule({
  imports: [ CarouselModule ],
  declarations: [ HelloWorldComponent ],
  bootstrap: [ HelloWorldComponent ],
})
export class AppModule {}
```

app.module.ts

Bootstrapping your application

Current Structure

```
hello-world.component.html
hello-world.component.ts
index.html
main.ts
app.module.ts
                              app/
```

Bootstrapping your application

Milestone #1

1. Open **app.component.html** and add 'Hello, YouTube!'.
2. Hook up **app.component.html** to the **YoutubeApp** component.
3. Declare **YoutubeApp** in the **AppModule** in **main.ts**.
4. Bootstrap **AppModule** in **main.ts**.
5. Add a **youtube-app** tag to **index.html**.

Content

Writing Effective Templates

Person Component

```
import { Person } from './person.model';

@Component({
  moduleId: module.id,
  selector: 'user-profile',
  templateUrl: './user-profile.component.html'
})
export class UserProfileComponent {
  person: Person = {
    name: 'Alice',
    birthday: 'April 6, 1980',
    emailAddress: 'alice@foo.com',
    photoUrl: 'alice.jpg',
    getBiography: () => {...}
  };
}
```

user-profile.component.ts

Writing Effective Templates

Text Interpolation

```html
<!-- Use property value from person object from component instance. -->
<h2>Profile for {{person.name}}</h2>

<!-- Use method on person object from component instance. -->
<h2>Profile for {{person.getBiography()}}</h2>

<!-- Use property on person object from component instance. -->
<img alt="Photo of {{person.name}}" ...>
```

user-profile.component.html

```html
<user-profile></user-profile>
```

users.component.html

Writing Effective Templates

Property Binding (one-way)

```
<!-- Bind [property] to the template express property. -->
<img [src]=”person.photoUrl” ...>

<!-- Yes, this is valid HTML syntax. -->
<input [value]=”person.emailAddress”>

<!-- This also works on your own components! -->
<birthday-card [date]=”person.birthday”>
```

user-profile.component.html

```
<user-profile></user-profile>
```

users.component.html

Writing Effective Templates

Reference Binding

```html
<!-- You can define a variable that points to an element or
     Component instance by using a hash. -->
<div>
  <!-- userName variable is available globally in this template. -->
  <input #userName>
  <!-- Remember, input elements have a value property. -->
  <button (click)=”isTaken(userName.value)”>
    Check if taken
  </button>
</div>
```

register.component.html

Writing Effective Templates

Shortcut Property Binding

```
<!-- Add class `.important` if `isImportant` evaluates to true. -->
<img [class.important]="isImportant" ...>

<!-- If `isWide` evaluates to true, set the style to `widePx`.  -->
<input [style.width.px]="isWide ? widePx : narrowPx">

<!-- You can also set attributes as opposed to properties. -->
<div [attr.aria-disabled]="isDisabled">
```

Event Binding

```
<!-- When user clicks the button, call the `saveUser` function on the
     component instance and pass the the underlying event. -->
<button (click)=”saveUser($event)”>

<!-- You can also create events for custom components. Here we have a
     depleted event, and it's going to call the `soundAlarm` function
     on the component instance when it fires.  -->
<coffee-maker (depleted)=”soundAlarm('loud')”>

<!-- There are also shortcut event bindings! The submit function on the
     component instance will be called when the user presses control
     and enter. -->
<textarea (keydown.control.enter)=”submit()”></textarea>
```

Writing Effective Templates

Conditional Display

```
<!-- Some directives change the structure of the component tree.
     ngIf conditionally shows/hides a section of the UI. -->
<section *ngIf="isSectionVisible">Howdy!</section>

<!-- Note the * and that it is case-sensitive! -->
```

Writing Effective Templates

Conditional Display

```
<!-- ngFor dynamically changes the structure too!
     Note again the * and case-sensitivity of the directive. -->
<ul>
  <li *ngFor="let player of team.roster">
    {{player.name}}
  </li>
</ul>
```

Similar to ES6, iterates over collections.
Note the use of "`let`" and "`of`".

Writing Effective Templates

Milestone #2

1. Add the **SearchVideos** component to **SearchModule**.
2. Add a fetch button to **search-videos.component.html**.
3. Add a click handler to **search-videos.component.ts**.
4. Display the number of **FAKE_RESULTS** in **search-videos.component.html**.
5. Show the description for each result in **search-videos.component.html**.

Content

Providing code using dependency injection

Recap

Dependency Injection is a way to provide dependencies to your code instead of hard-coding them.

Providing code using dependency injection

Comparison

Without Dependency Injection

```
export class MyComponent {
  service: MyService;

  constructor() {
    this.service = new MyService();
  }

}
```

With Dependency Injection

```
export class MyComponent {
  /**
   * Typescript shorthand makes `service`
   * available to component instance.
   */
  constructor(
      public service: MyService) {}
}
```

Providing code using dependency injection

Implementation - Step #1

Mark a class as injectable

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

@Injectable()
export class MyService {
   ...
}
```

Providing code using dependency injection

Implementation - Step #2

Provide the injectable

```
import { NgModule } from '@angular/core';
import { UnitConverterService }
    from '../services/unit-converter.service';
import { UnitConversionComponent }
    from './unit-conversion.component';

@NgModule({
  declarations: [ UnitConversionComponent ],
  provides: [ UnitConverterService ],
})
export class AppModule {}
```

Providing code using dependency injection

Implementation - Step #3

Consume the injectable

```
import { Component } from '@angular/core';
import { UnitConverterService }
    from '../services/unit-converter.service';

@Component({...})
export class UnitConversionComponent {
  constructor(converter: UnitConverterService) {}
}
```

unit-conversion.component.ts

Providing code using dependency injection

Handling services

Promises and callbacks

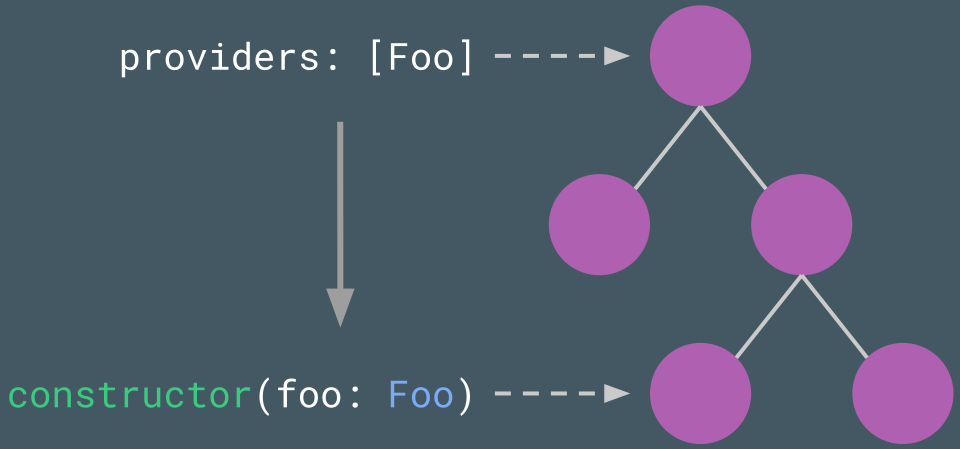
```
import { Component } from '@angular/core';
import { UnitConverterService }
    from '../services/unit-converter.service';

@Component({...})
export class UnitConversionComponent {
  constructor(public converter: UnitConverterService) {}

  getUnit(fromUnit: string, value: number) {
    this.converter.doStuff();
  }
}
```

unit-conversion.component.ts

Providing code using dependency injection

Availability in Modules

Providers declared by a module are available to all of its components



Providing code using dependency injection

Milestone #3

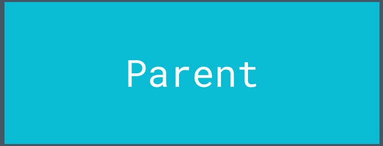
1. Replace the fake data with the results from **YoutubeService**.

2. Update the text binding to show the video description.

Content

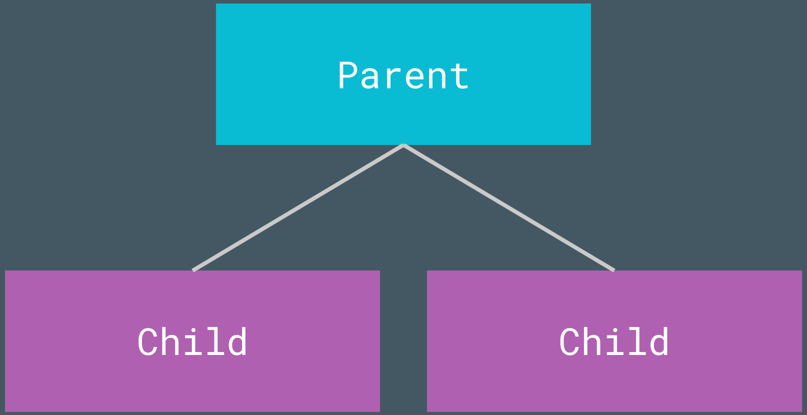
Composing your app with a tree of components

Single Component



Composing your app with a tree of components

Single Component



Composing your app with a tree of components

Instantiating Sub-components

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

@Component({
  selector: 'parent',
  template: '<child>...</child>'
})
export class ParentComponent {}
```

parent.component.ts

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

@Component({
  selector: 'child',
  template: '<p>I'm a child!</p>'
})
export class ChildComponent {}
```

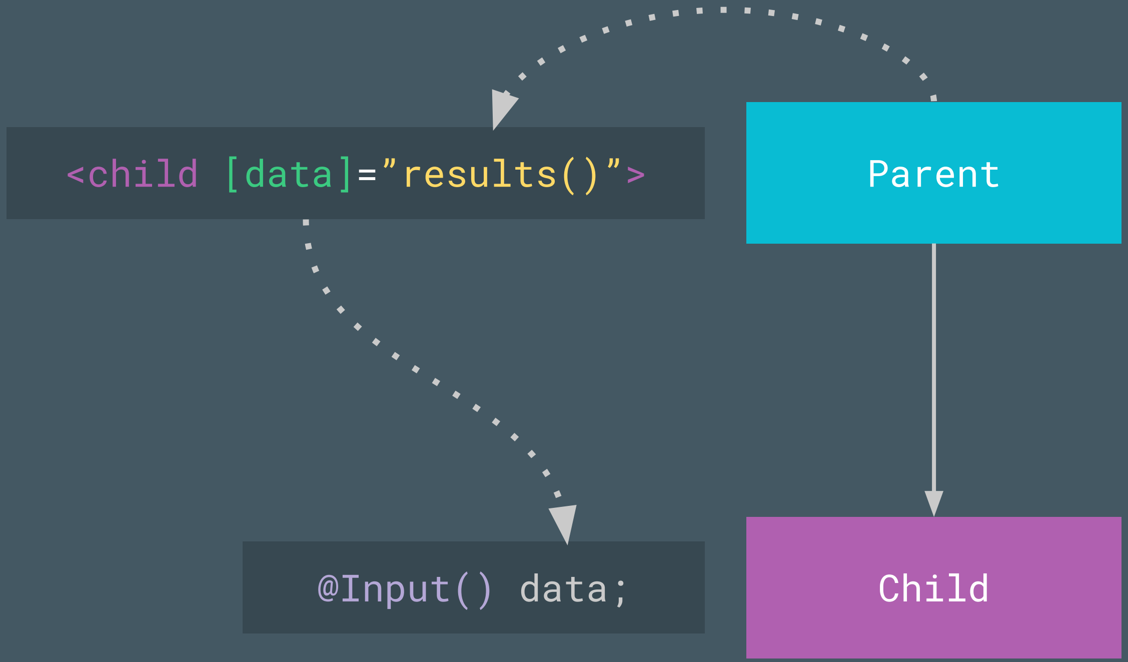
child.component.ts

```
import { NgModule } from '@angular/core';
import { ChildComponent } from './child.component';
import { ParentComponent } from './parent.component';

@NgModule({
  declarations: [ ChildComponent, ParentComponent ]
})
export class AppModule {}
```

app.module.ts

Composing your app with a tree of components

Passing data from parent to child



```
<child [data]="results()">
```

```
@Input() data;
```

Composing your app with a tree of components

Passing data from parent to child

```
import { Component }
    from '@angular/core';
import { Result }
    from './result.model';

@Component({
  selector: 'parent',
  template: '<child [data]="results()">
            </child>'
})
export class Parent {
  results(): Result[] {...}
}
```

parent.component.ts

```
import { Component, Input }
    from '@angular/core';
import { Result }
    from './result.model';

@Component({
  selector: 'child',
  template: '<p *ngFor="let result of
            data">{{result}}</p>'
})
export class Child {
  @Input() data: Result[];
}
```

child.component.ts

Composing your app with a tree of components

Milestone #4

1. Add the **SearchResultCard** component to the **SearchModule**.

2. Instantiate one child component per search result from the API in **`search-videos.component.html`**.

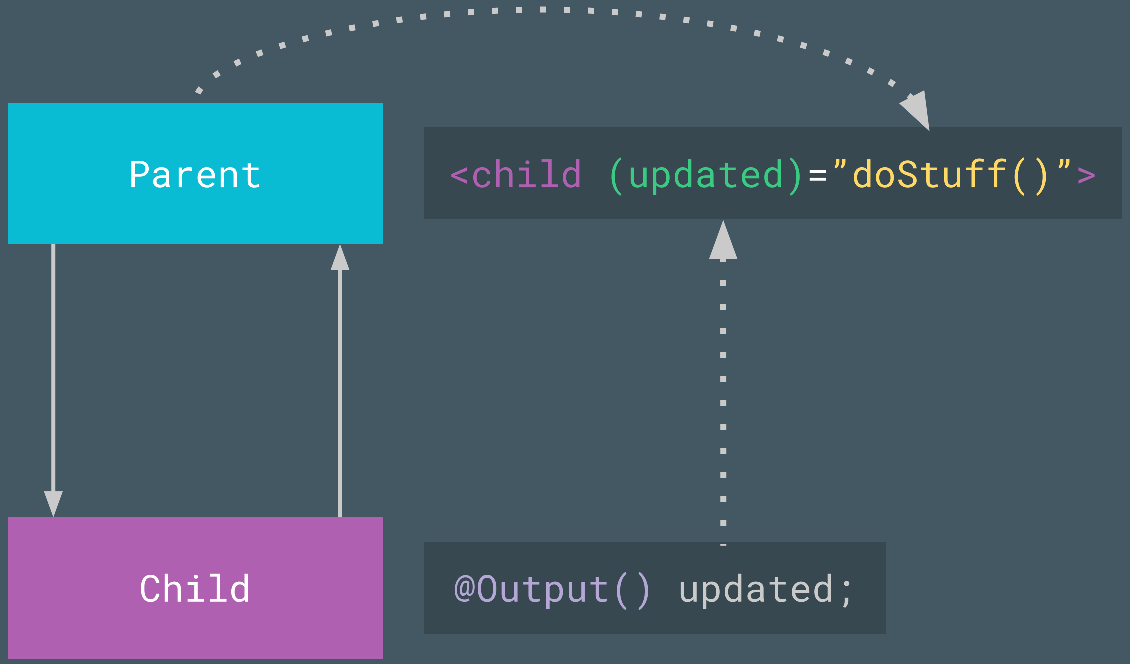
3. Pass the search result into the **SearchResultCard** component in **`search-result-card.component.ts`**.

4. Render the search result description, viewCount, likeCount, and thumbnail in its template, **`search-result-card.component.html`**.

Content

Handling custom events

Passing data between components

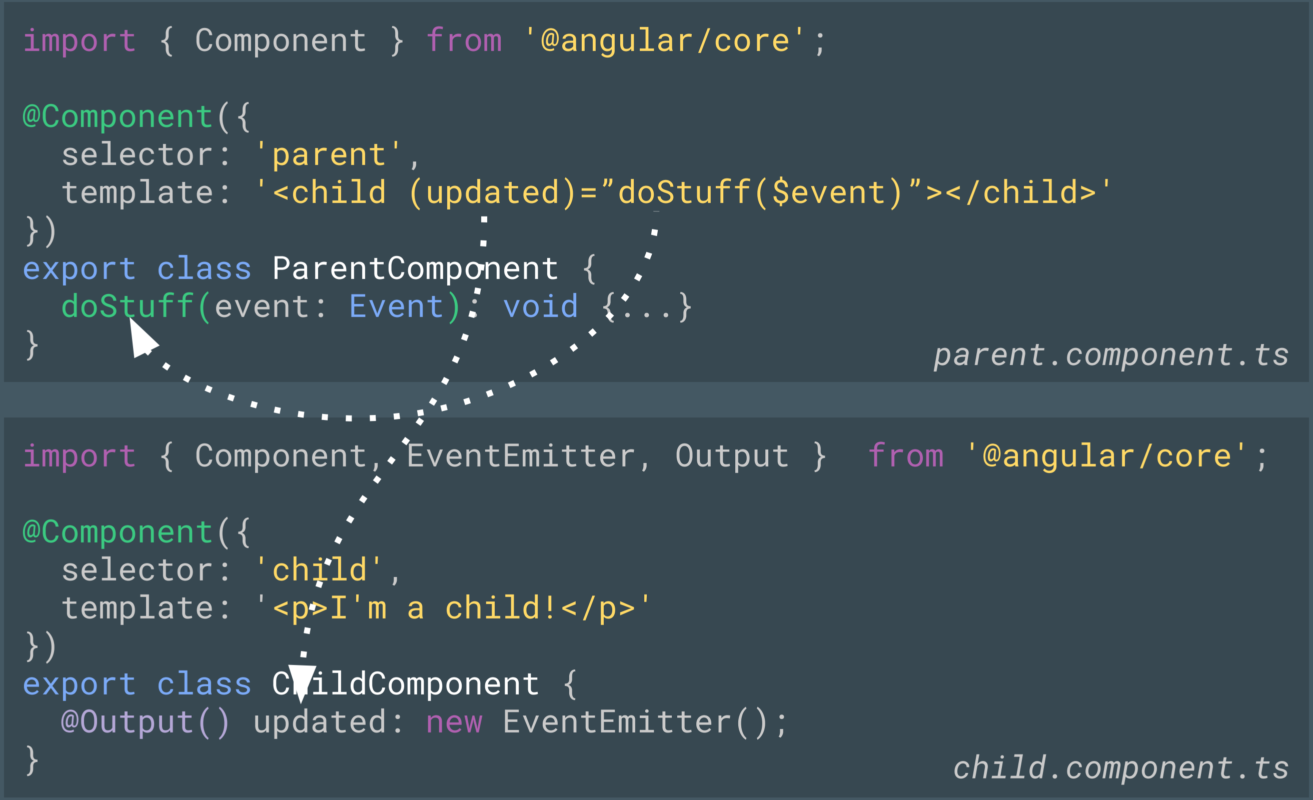


```
<child (updated)="doStuff()">
```

```
@Output() updated;
```

Handling custom events

Passing data from child to parent



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

@Component({
  selector: 'parent',
  template: '<child (updated)=”doStuff($event)”></child>'
})
export class ParentComponent {
  doStuff(event: Event): void {...}
}
```

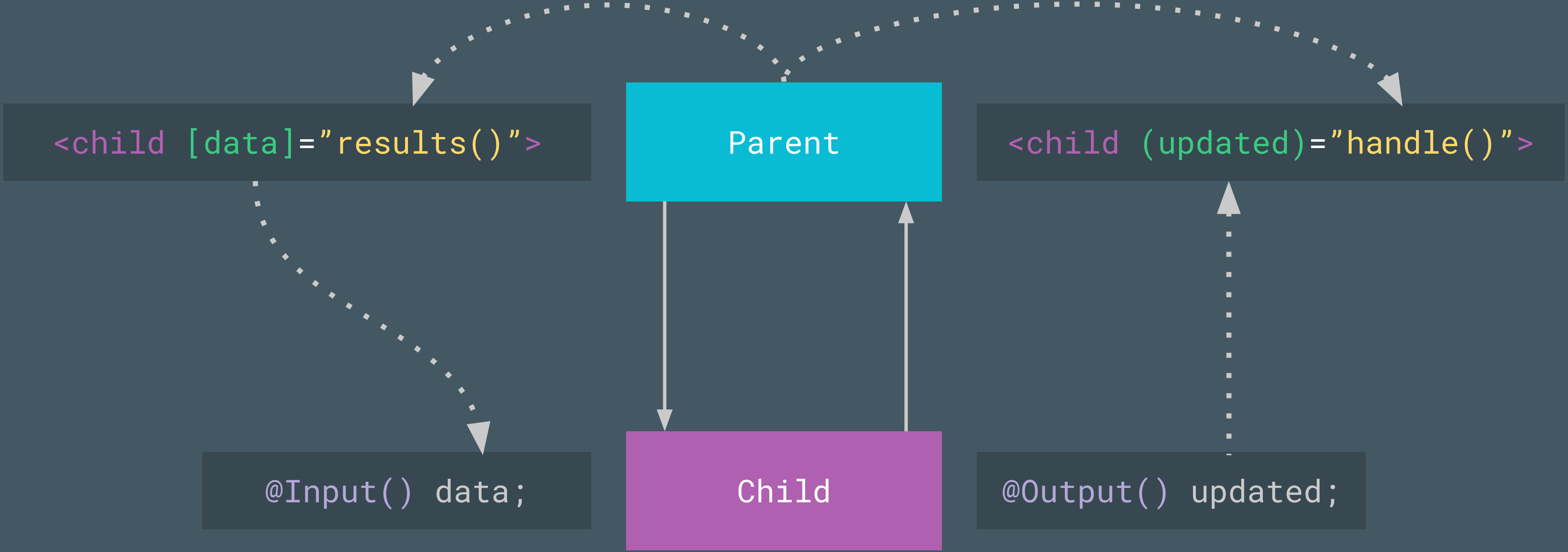
parent.component.ts

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

@Component({
  selector: 'child',
  template: '<p>I'm a child!</p>'
})
export class ChildComponent {
  @Output() updated: new EventEmitter();
}
```

child.component.ts

Handling custom events

Passing data between components



```
<child [data]="results()">
```

```
<child (updated)="handle()">
```

```
@Input() data;
```

```
@Output() updated;
```

Handling custom events

Passing data from child to parent

```
import { Component, Output } from '@angular/core';
import { CoffeeEvent } from './coffee-event.model';

export enum BrewState {
  NOT_BREWING,
  BREWING
}

@Component({
  selector: 'coffee-maker',
  template: '<p>I'm a coffee maker!</p>'
})
export class CoffeeMaker {
  @Output() depleted: new EventEmitter<CoffeeEvent>();
  onPour(newState: BrewState): void {
    this.depleted.emit(new CoffeeEvent(...));
  }
}
```

coffee-maker.component.ts

Handling custom events

Milestone #5

1. Add the **Thumbs** component to the **SearchModule**.
2. Instantiate the **Thumbs** component in the **SearchResultCard** component in **`search-result-card.component.ts`**.
3. Emit an event when the thumb states change in **thumbs.ts**.
4. Use event binding to update the like/dislike counts in **`search-result-card.component.html`**.

Content

Testing your code

Writing a simple test

```
describe('SimpleTest', () => {
  beforeEach(() => {
    /* Perform common setup for each test. */
  });

  it('should perform some behavior', () => {
    /* Perform additional setup for this test. */
    expect(simple.value).toEqual('expected value');
  });
};
```

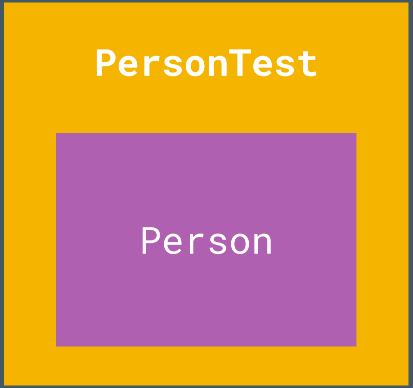
simple.spec.ts

Testing your code

Anatomy of a component under test

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

@Component({
  selector: 'test-wrapper',
  template: '<person [input]=”data”>'
})
export class TestPersonComponent {
  data: string = '';
}
```

person.spec.ts



Testing your code

Writing a component test

```
import { Component } from '@angular/core';
import { async, TestBed } from '@angular/core/testing';
import { PersonComponent } from './person.component';

/* We've seen this defined already... */
@Component({...})
export class TestPersonComponent {...}

describe('TestPersonComponent', () => {
  /* Setup the test bed. */
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ TestPersonComponent, PersonComponent ]
    });
    TestBed.compileComponents();
  }));
});
```

person.spec.ts

Needed for compiling

Creates a special NgModule ----►

Same as in NgModule ----►

Make components available ----►

Testing your code

Writing a component test

```
                              import { Component } from '@angular/core';
                              import { async, TestBed } from '@angular/core/testing';
                              import { PersonComponent } from './person.component';

                              /* We've seen this defined already... */
                              @Component({...})
                              export class TestPersonComponent {...}

                              describe('PersonTest', () => {
                                beforeEach(...);

                                it('should contain some text', () => {
                                  let fixture = TestBed.createComponent(TestPersonComponent);
                                  fixture.componentInstance.data = 'foo';
   Trigger change detection ----► fixture.detectChanges();
Get the test native element ----► const testElement = fixture.nativeElement;
        Perform assertions ----► expect(testElement.textContent).toContain('foo');
                                });
                              });
```

person.spec.ts

Writing a component test (complete)

```
import { Component } from '@angular/core';
import { async, TestBed } from '@angular/core/testing';
import { PersonComponent } from './person.component';

@Component({
  selector: 'test-wrapper',
  template: '<person [input]=”data”>'
})
export class TestPersonComponent {
  data: string = '';
}

describe('PersonTest', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ TestPersonComponent, PersonComponent ]
    });
    TestBed.compileComponents();
  }));

  it('should contain some text', () => {
    let fixture = TestBed.createComponent(TestPersonComponent);
    fixture.componentInstance.data = 'foo';
    fixture.detectChanges();
    let testElement = fixture.debugElement.nativeElement;
    expect(testElement.textContent).toContain('foo');
  });
});
```

person.spec.ts

Testing your code

Milestone #6

1. Create a test component that wraps your component under test (`search-result-card.spec.ts`).

2. Declare all the components used in the test.

3. Assert that the test component contains both the text from the template and fake data from the stubbed service

Content

Transforming data using pipes

Reminder: Text Interpolation

```
<p>Annual Budget: {{department.annualBudget}}</p>
```

departments.component.html

```
Annual Budget: 123456
```

Browser Rendering

Transforming data using pipes

Let's fix that...

```html
<p>Annual Budget: {{department.annualBudget | currency}}</p>
```

departments.component.html

```
Annual Budget: $1234.56
```

Browser Rendering

Transforming data using pipes

How do pipes work?

1) Pipes can transform **any** input type to any output type
2) Pipes can be used in **any** Angular binding expression

```
<!-- Here we pipe team.roster through a custom “notInjured” pipe,
     then render each player that isn't injured. -->
<ul>
  <li *ngFor=”let player of (team.roster | notInjured)”>
    {{player.name}}
  </li>
</ul>
```

Transforming data using pipes

Creating Pipelines

Multiple pipes can be chained together in the same expression

```
<!-- “2016-02-04T20:16:26+00:00" -->
<p>{{birthday | fullDate | uppercase}}</p>

<!-- Then:    “Feb 4th, 2016” -->
<!-- Finally: “FEB 4TH, 2016” -->
```

Transforming data using pipes

Pipes with arguments

Pipes can accept parameters -- use a colon `:` to delimit.

```
<p>Your budget is {{budget | currency:”CAD”}}</p>

<p>Your truncated name is {{name | substring:1:4}}</p>

<!-- Maryanne -> Mary -->
```

Transforming data using pipes

Creating a pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'substring'})
export class SubstringPipe implements PipeTransform {
  transform(value: string, start: number, end: number): string {
    return (value || '').slice(start, end);
  }
}
```

substring.pipe.ts

```
interface PipeTransform {
  transform(value: any, ...args: any[]): ay {}
}
```

Transforming data using pipes

Consuming a pipe

```
import { NgModule } from '@angular/core';
import { SubstringPipe } from './pipes/substring.pipe';

@NgModule({
  declarations: [ SubstringPipe ]
})
export class AppModule {}
```

app.module.ts

Transforming data using pipes

Built-in Pipes

```
<!-- json: renders a Javascript object in JSON. -->
{{jsObject | json}}

<!-- date: converts a date object into another format. -->
{{today | date:”dd/MM/yyyy”}}

<!-- async: renders resolved value returned by promise or
            observable. -->
{{promise | async}}
```

See more built-in pipes at angular.io

Transforming data using pipes

Milestone #7

1. Implement the **FuzzyTime** pipe. It will take an ISO-8061 date string and output the relative time ago, e.g. 1 year ago.

2. Use the **FuzzyTime** pipe in **SearchResultCard** to show how long ago each video was published.

Hint 1: you can use `new Date(publishDateString)`. To subtract dates in TypeScript, you'll need to call the `getTime()` method on the date instance first.

Hint 2: You don't need to implement advanced logic here, just learn how to use pipes!

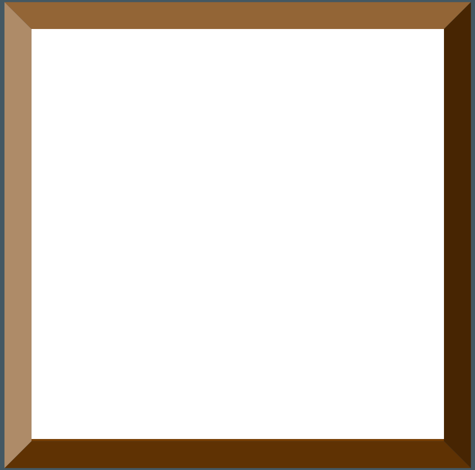
Content

Projecting content into your components

Content vs. View

```html
<picture-frame>


</picture-frame>
```

my-photos.component.html

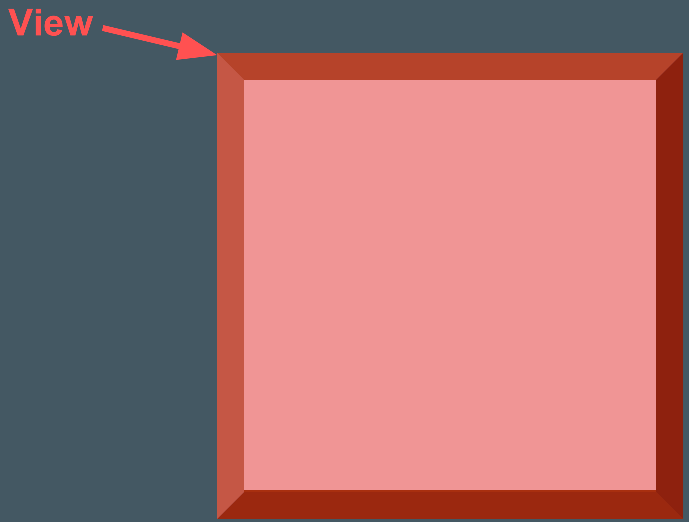


Projecting content into your components

Content vs. View

```
<picture-frame>


</picture-frame>
```

my-photos.component.html



Projecting content into your components

Content vs. View

```
<picture-frame>

  <img src="mal.png">

</picture-frame>
```

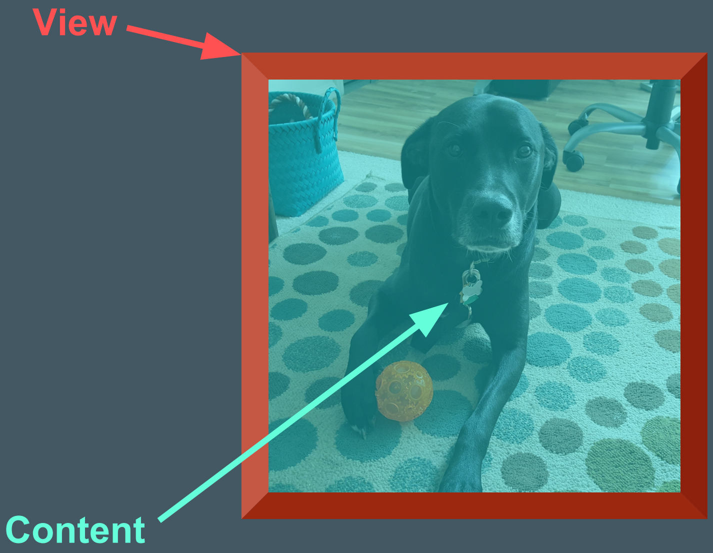
my-photos.component.html

Projecting content into your components

Content vs. View

```
<picture-frame>

  <img src="mal.png">

</picture-frame>
```

my-photos.component.html

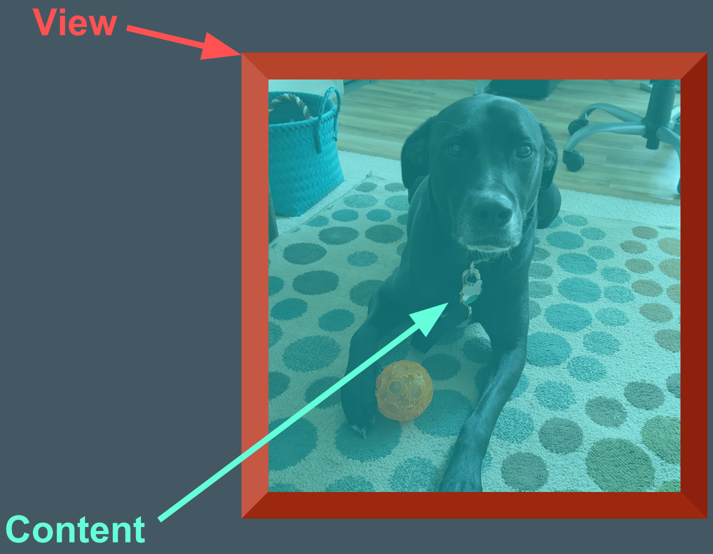


Projecting content into your components

Content vs. View

```
<picture-frame>

  <img src="mal.png">

</picture-frame>
```

my-photos.component.html



For the picture frame component, the frame rendered is the component **view**, and the photo is the **content**.

Projecting content into your components

Content vs. View

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

@Component({
  selector: 'picture-frame',
  template: '<div class=”photo”>
               <ng-content></ng-content>
             </div>'
})
export class PictureFrameComponent { ... }
```

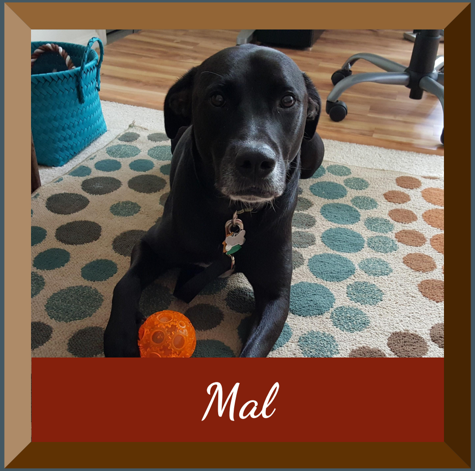
picture-frame.component.ts

Projecting content into your components

Multiple Contents vs. View

```
<picture-frame>

  <img src="mal.png">

  <p class="name">Mal</p>

</picture-frame>
```

my-photos.component.html

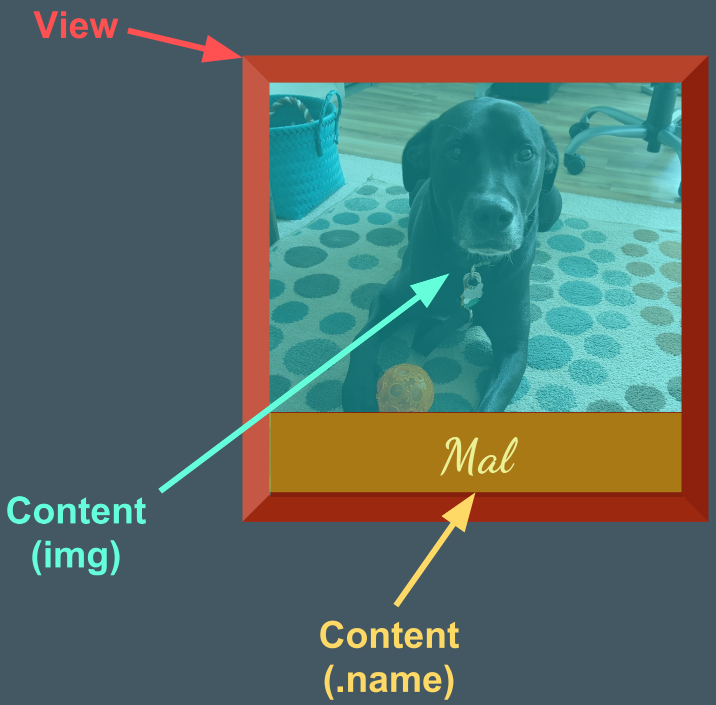


Projecting content into your components

Multiple Contents vs. View

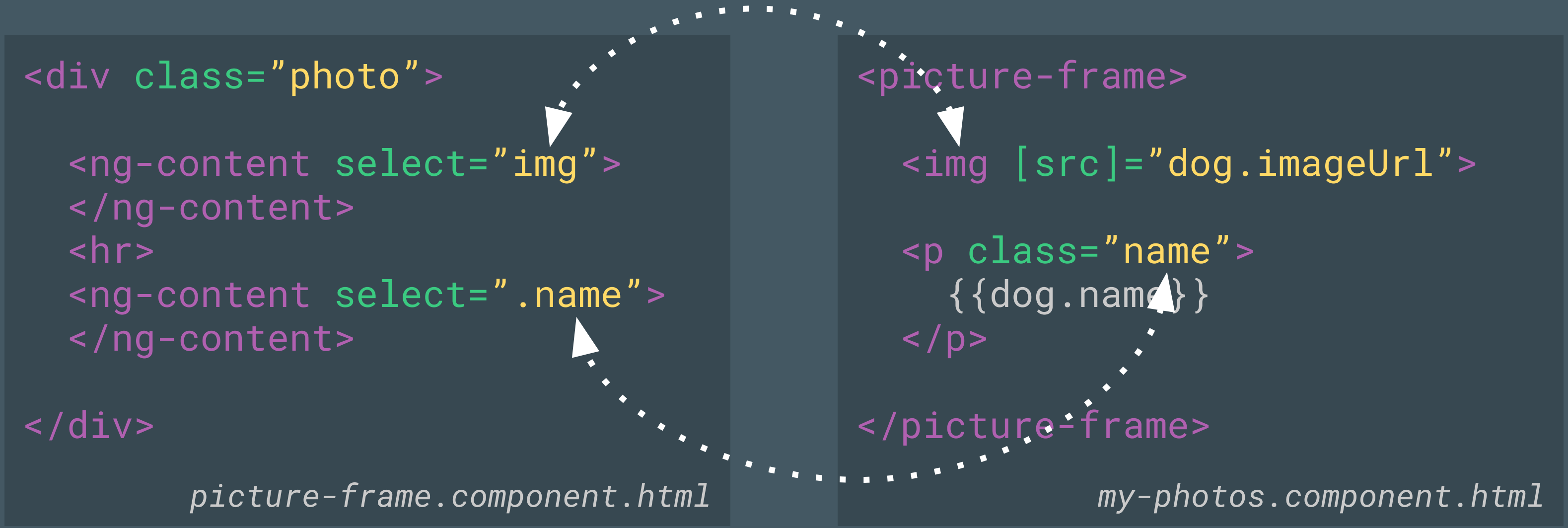
```
<picture-frame>

  <img src="mal.png">

  <p class="name">Mal</p>

</picture-frame>
```

my-photos.component.html



Projecting content into your components

Multiple Contents vs. View



```
<div class="photo">

  <ng-content select="img">
  </ng-content>
  <hr>
  <ng-content select=".name">
  </ng-content>

</div>
```

picture-frame.component.html

```
<picture-frame>

  <img [src]="dog.imageUrl">

  <p class="name">
    {{dog.name}}
  </p>

</picture-frame>
```

my-photos.component.html

Projecting content into your components

Milestone #8

1. Implement **TogglePanelComponent** with a button that toggles between two different sections by editing **toggle-panel.component.ts** and **toggle-panel.component.html**.

2. Export it from **ToggleModule** and import it into **AppModule**.

3. Use **TogglePanelComponent** to switch between video votes and descriptions on the search result card.

4. Keep all video data in **search-result-card.component.html**, and project using two **<ng-content>** directives into the toggle panel.

Hint: Don't forget Event Binding!

Content

Coupling components tightly

Loose vs. Tight Coupling

Loose Coupling

- Good for reusable, isolated components
- Use: @Input / @Output and template syntax

Tight Coupling

- For components that work together
- Stronger type safety
- Use: constructor injection, @ContentChildren, @ViewChildren

Coupling components tightly

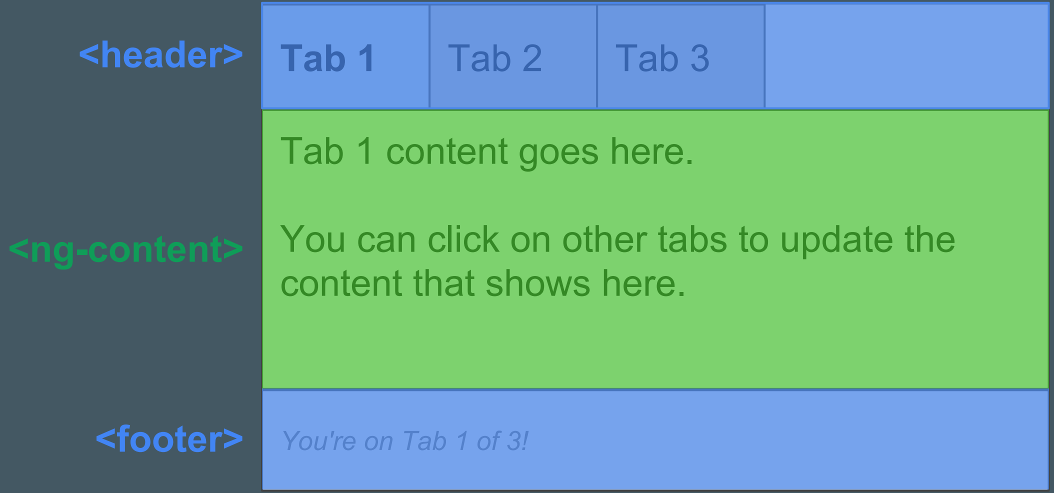
Example: Tabs

```
<tab-group>
  <tab>Tab 1 content...</tab>
  <tab>Tab 2 content...</tab>
  <tab>Tab 3 content...</tab>
</tab-group>
```

dashboard.component.html

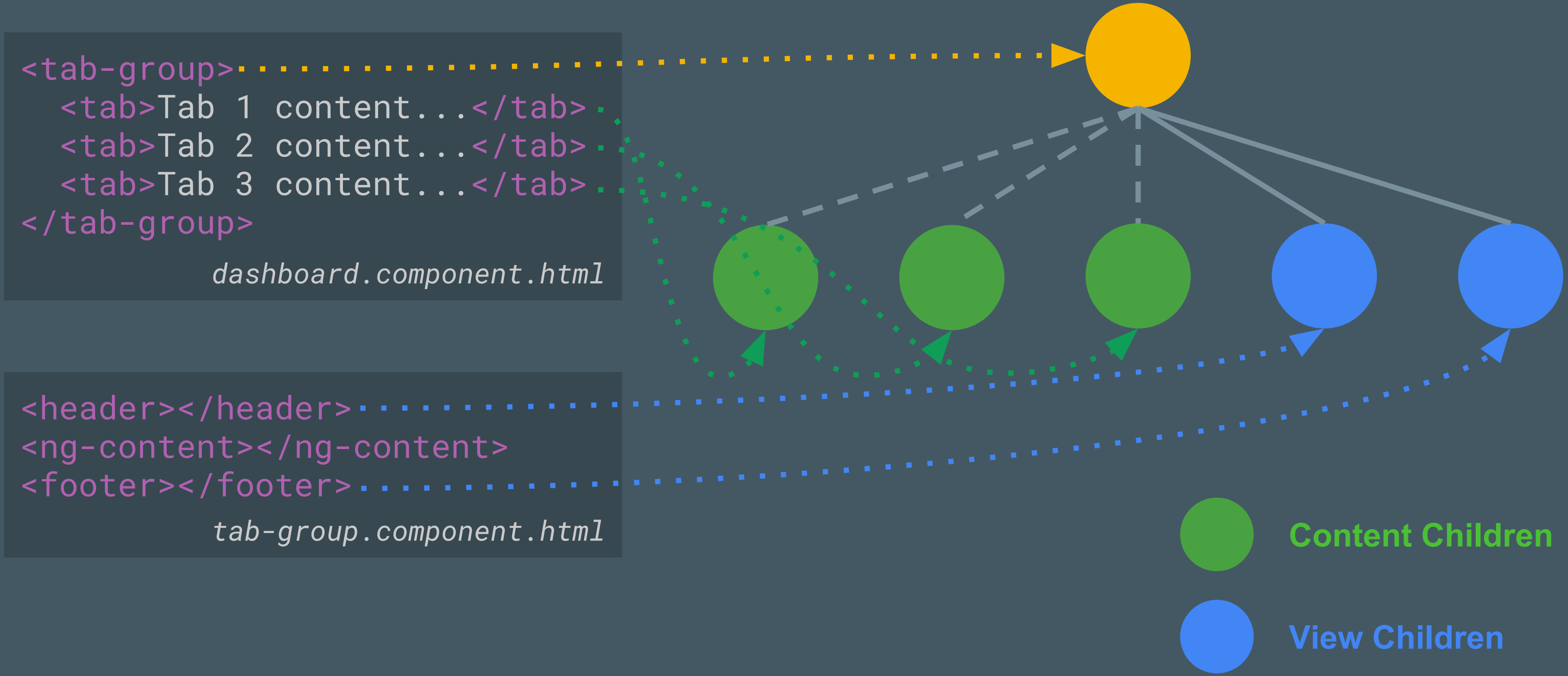
```
<header></header>
<ng-content></ng-content>
<footer></footer>
```

tab-group.component.html



Coupling components tightly

Content Children vs. View Children



```
<tab-group>
  <tab>Tab 1 content...</tab>
  <tab>Tab 2 content...</tab>
  <tab>Tab 3 content...</tab>
</tab-group>
```

dashboard.component.html

```
<header></header>
<ng-content></ng-content>
<footer></footer>
```

tab-group.component.html

Coupling components tightly

Injecting a Parent Component

```
import { Component } from '@angular/core';
import { TabGroupComponent } from './tab-group.component';

@Component({
  selector: 'tab',
  templateUrl: 'tab.component.html'
})
export class TabComponent {
  constructor(tabGroup: TabGroupComponent) {}
}
```

tab.component.ts

Coupling components tightly

Querying for child components

```
import { ContentChildren, Component, QueryList, ViewChildren }
    from '@angular/core';
import { TabComponent } from './tab.component';
import { InputComponent } from '../common/input.component';

@Component({
  selector: 'tab-group',
  templateUrl: 'tab-group.component.html'
})
export class TabGroupComponent {
  @ContentChildren(TabComponent) tabs: QueryList<TabComponent>;
  @ViewChildren(InputComponent) inputs: QueryList<InputComponent>;
}
```

tab-group.component.ts

You need to use the `.toArray()` method on QueryList to iterate for ES5!

Coupling components tightly

Querying for single child components

```
import { ContentChild, Component, ViewChild } from '@angular/core';
import { CoffeePotComponent } from './coffee-pot.component';
import { HeatingCoilComponent } from './heating-coil.component';

@Component({
  selector: 'coffee-maker',
  templateUrl: 'coffee-maker.component.html'
})
export class CoffeeMakerComponent {
  @ContentChild(CoffeePotComponent) pot: CoffeePotComponent;
  @ViewChild(HeatingCoilComponent) heater: HeatingCoilComponent;
}
```

coffee-maker.component.ts

Coupling components tightly

Revisiting our TabGroupComponent

```
import { ContentChildren, Component, QueryList, ViewChild }
    from '@angular/core';
import { TabComponent } from './tab.component';
import { HeaderComponent } from './header.component';
import { FooterComponent } from './footer.component';

@Component({
  selector: 'tab-group',
  templateUrl: 'tab-group.component.html'
})
export class TabGroupComponent {
  @ContentChildren(TabComponent) tabs: QueryList<TabComponent>;
  @ViewChild(HeaderComponent) header: HeaderComponent;
  @ViewChild(FooterComponent) footer: FooterComponent;
}
```

tab-group.component.ts

Coupling components tightly

Milestone #9

1. If the video description has the word “music”, display “Turn Up Your Speakers!”, otherwise display “Learn more about cats on our channel!”. Implement this feature using **VideoAnnotationComponent** and put it in **search-result-card.component.html**.

2. Query for all cards in **search-videos.component.ts**.

3. Coordinate so that only one video is playing at any given time.

4. Don't forget to update **search.module.ts**!

Review: @Input and @Output

Content

Utilizing lifecycle hooks

When is my data ready?

```html
<shoe-picker [heelStyle]=”filter.heel”></shoe-picker>
```

shoe-picker.component.html

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

@Component({
  selector: 'shoe-picker',
  templateUrl: 'shoe-picker.component.html'
})
export class ShoePickerComponent {
  @Input() heelStyle: string;
  constructor() {
    console.log(this.heelStyle); ✗
  }
}
```

shoe-picker.component.ts

Component instance created before the inputs are updated with the bound values...

Utilizing lifecycle hooks

When is my data ready?

```
<shoe-picker [heelStyle]=”filter.heel”></shoe-picker>
```

shoe-picker.component.html

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

@Component({
  selector: 'shoe-picker',
  templateUrl: 'shoe-picker.component.html'
})
export class ShoePickerComponent implements OnInit {
  @Input() heelStyle: string;
  ngOnInit() {
    console.log(this.heelStyle); ✓
  }
}
```

shoe-picker.component.ts

Utilizing lifecycle hooks

What's available?

`OnInit`	**Called when Angular has *finished initializing* the component.** Used for *reading* initial bound values.
`OnDestroy`	**Called when Angular is *destroying* the component.** Used for any *clean-up before* the component is destroyed.

Utilizing lifecycle hooks

What's available?

OnChanges	**Called *after every change* to input properties and *before processing* any content or child views.** Passed a map of the changed values.
DoCheck	**Called *every time* input properties are checked for changes.** Used to replace Angular's change detection with custom checks. **NOT COMMONLY USED -- PROCEED WITH CAUTION!**

Utilizing lifecycle hooks

What's available?

`AfterContentInit`	Called after the component's **content** is initialized
`AfterViewInit`	Called after the component's **view** is initialized
`AfterContentChecked`	Called after each check of the component's **content**
`AfterViewChecked`	Called after each check of the component's **view**

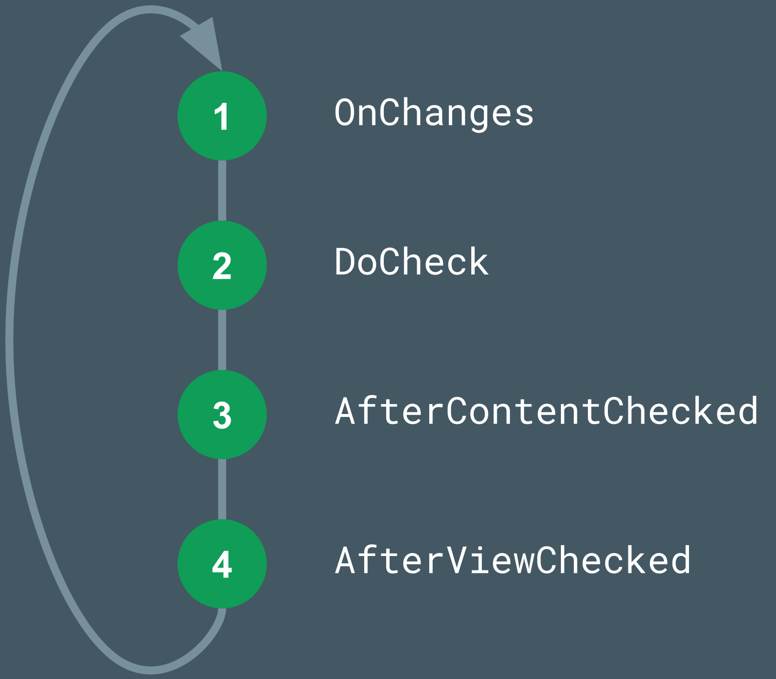
Utilizing lifecycle hooks

Lifecycle Hook Order

1. `OnChanges`
2. `OnInit`
3. `DoCheck`
4. `AfterContentInit`
5. `AfterContentChecked`
6. `AfterViewInit`
7. `AfterViewChecked`
8. `OnDestroy`

Utilizing lifecycle hooks

Lifecycle Traversal 1 .. n-1



Directives vs. Components

- Directives in Angular 2 are the superclass of Component.
- A Component is just a directive with template.
- Directives are also controlled by a selector in HTML.

Utilizing lifecycle hooks

Change Detection

- Angular's change detection traverses the component tree from top to bottom, i.e. depth-first traversal.

- Once a component is checked, its values cannot be updated again in the same cycle.

- If you try, Angular with throw an error in development mode .

Utilizing lifecycle hooks

Milestone #10

1. Create the **`AnalyticsMonitorDirective`** with inputs for **`videoId`** and **`likeCount`**.

2. **`AnalyticsMonitorDirective`** injects the **`AnalyticsTrackerService`** and calls the **`track()`** method whenever its inputs change.

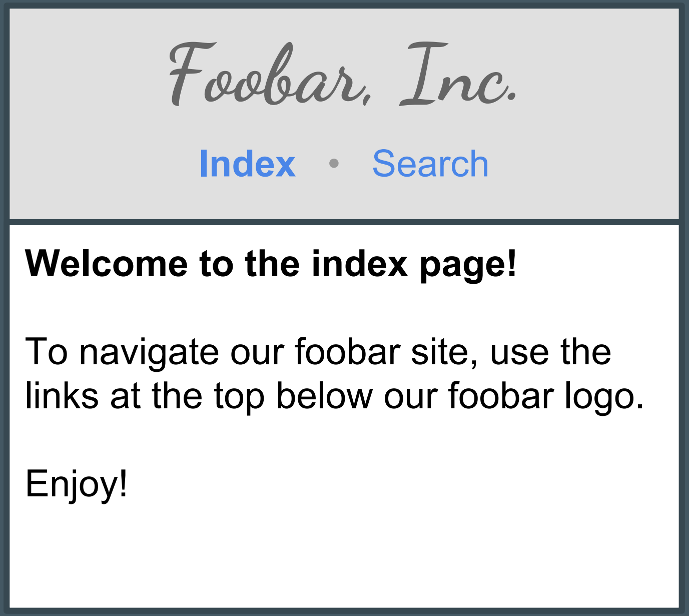
3. Each **`SearchResultCardComponent`** should include an **`AnalyticsMonitorDirective`** in the template.

Review: @Input and @Output

Content

Creating a single-page app with routes

Typical Page Layout


Foobar, Inc.

Index • Search

Welcome to the index page!

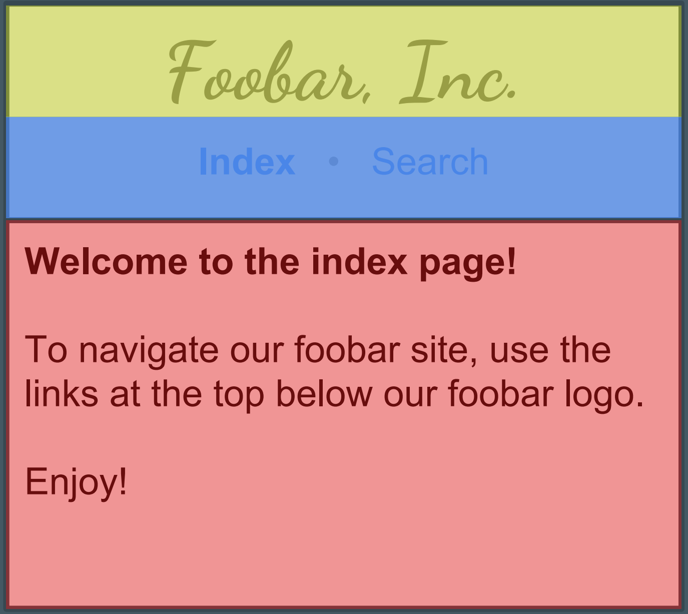
To navigate our foobar site, use the links at the top below our foobar logo.

Enjoy!

Creating a single-page app with routes

Typical Page Layout

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

@Component({
  selector: 'foobar-app',
  template: '<h1 class="logo">Foobar, Inc.</h1>
             <a routerLink="/index">Index</a>
             <a routerLink="/search">Search</a>
             <router-outlet></router-outlet>'
})
export class FoobarAppComponent {}
```
foobar-app.component.ts



Foobar, Inc.

Index • Search

Welcome to the index page!

To navigate our foobar site, use the links at the top below our foobar logo.

Enjoy!

Creating a single-page app with routes

Defining Routes

```
import { RouterModule, Routes } from '@angular/router;
import { IndexComponent } from './index.component';
import { SearchComponent } from './search.component'

const routes: Routes = [
  { path: '', redirectTo: '/', pathMatch: 'full' },
  { path: 'index', component: IndexComponent },
  { path: 'search', component: SearchComponent },
];

export const AppRoutesModule = RouterModule.forRoot(routes);
```

app-routes.module.ts

`forRoot` creates a module that contains the directives, the routes, and the router service itself.

`forChild` creates a module that contains the directives and the routes but not the router service.

Creating a single-page app with routes

Bindings

```
import { NgModule } from '@angular/core';
import { AppRoutesModule } from './app-routes.module';
import { FoobarAppComponent }  from './foobar-app.component';

@NgModule({
  declarations: [ FoobarAppComponent ],
  imports: [ AppRoutesModule ],
  bootstrap: [ FoobarAppComponent ],
})
export class AppModule {}
```

app.module.ts

Creating a single-page app with routes

Recap

1. `RouterOutlet` directive
2. `RouterLink` directive(s)
3. `RouterModule` module (with configuration)

Creating a single-page app with routes

Milestone #11

1. Add the **`router-outlet`** to your **`app.component.html`**.
2. Configure the routes in **`search/search.module.ts`**.
3. Import and configure the **`RouterModule`**.
4. Use the **`routerLink`** directive to switch between the search and upload views.

***Bonus:** In the search results page, implement the ability to take a result full page using params and child routes.*

Content

Handling user data with forms

Building Forms

```
<form>
  <input name=”username” ngModel>
  <input name=”email” ngModel>
  <div ngModelGroup=”address”>
    <input name=”street” ngModel>
    <input name=”zip” ngModel>
  </div>
</form>
```

register.component.html

```
{
  “username”: ”jane”
  “email”: jane.doe@example.com”,
  “address”: {
    “street”: “1234 Main St.”,
    “zip”: “12345”
  }
}
```

Form Object

Handling user data with forms

One-way vs. Two-way Binding

```
// One-way binding
<input name="username" ngModel>

// Two-way binding
<input [ngModel]="username" (ngModelChanges)="username = $event">

// Two-way binding shorthand -- recommended!
<input [(ngModel)]="username">
```

Accessing Forms

```
<form #registerForm=”ngForm”>
  ...
  <span *ngIf=”!registerForm.valid”>
    Form is invalid!
  </span>
</form>
```

register.component.html

```
import { NgForm } from '@angular/forms';

@Component({
  selector: 'register-form',
  templateUrl: 'register.component.html'
})
export class RegisterComponent {
  // Access form object in component methods.
  @ViewChild('registerForm') registerForm: NgForm;
}
```

register.component.ts

Handling user data with forms

Custom Validators

```
<!-- HTML5 validators work... -->
<input name=”username” ngModel required>

<!-- As do custom validators -->
<input name=”email” ngModel validated-email>

<!-- Angular 2 comes with validators built-in for HTML5,
     e.g. required. Custom validators let us do more
     complex validation. -->
```

Handling user data with forms

Custom Validators

```
import { Component, Directive } from '@angular/core';
import { FormControl, NG_VALIDATORS, NgForm } from '@angular/forms';

@Directive({
  selector: 'input[validated-email][ngModel]',  // CSS selector
  providers: [{
    provide: NG_VALIDATORS,                      // Specify it's a validator.
    useValue: EmailValidator.emailCheck,         // Validator function.
    multi: true                                  // Add, don't replace.
  }]
})
export class EmailValidator {
  static emailCheck(control: FormControl): {[errorKey: string]: any} {
    // Do some checks...
    return {'badEmail': 'Email address was invalid'}
  }
}
```

Handling user data with forms

Milestone #12

1. Add a form to the upload page with a title input, description textarea, and a submit button in **`upload-video.component`**.

2. Access the form in **`UploadVideosComponent`**. Hint: Use **`ViewChild`**.

Bonus: Implement validation…

1. Require a title
2. Disable the submit button if the form is invalid.
3. Force “cats” to be in the description.
4. Force “cats” to be in either the title or the description.

We're done!

Solution: goo.gl/0jkm07