Build Better Apps

with Angular 2

Strong grasp on how to build a single, basic feature in Angular 2

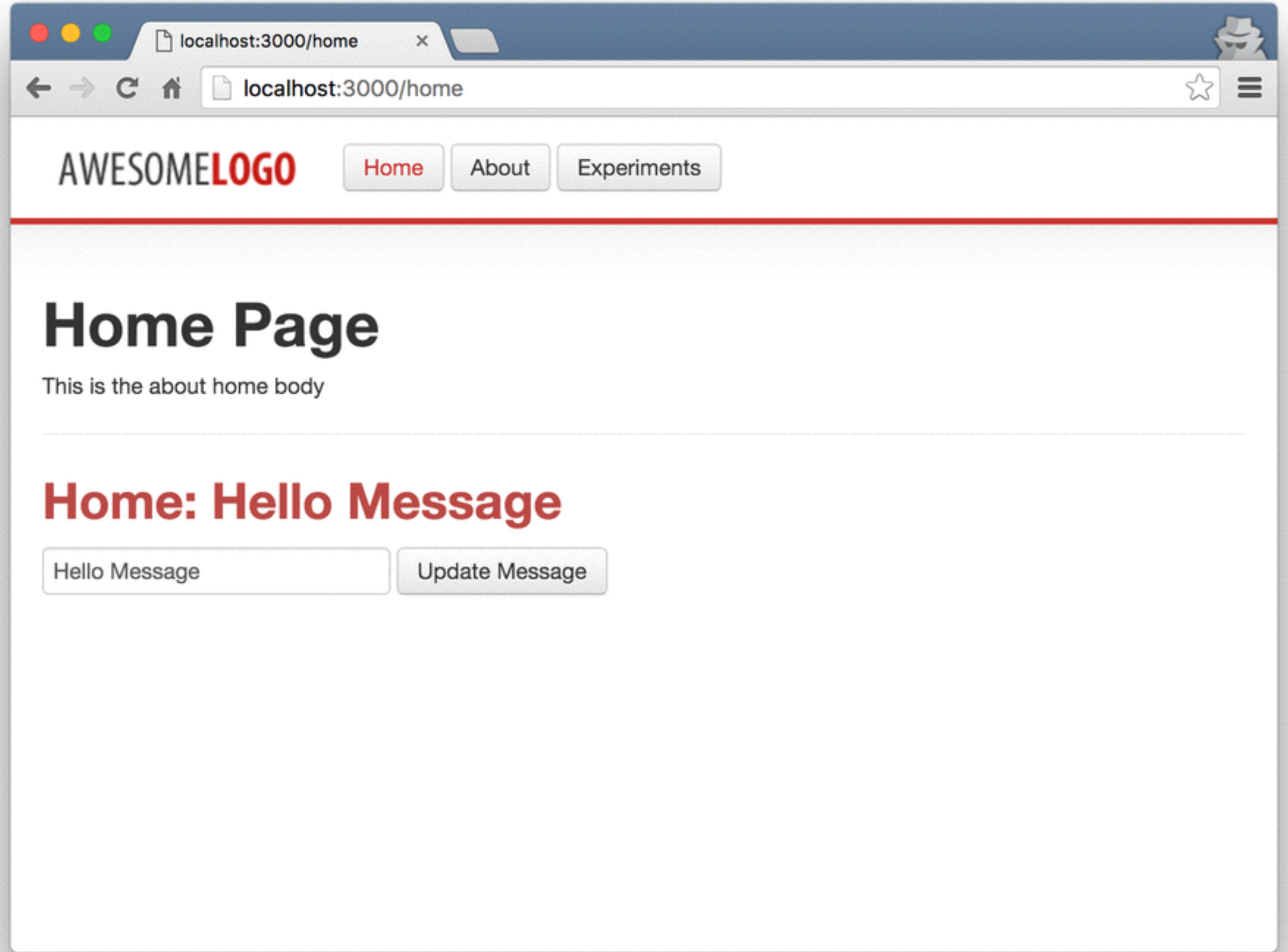
Agenda

- The Angular 2 Big Picture
- Prerequisite Primer in Tooling
- Component Fundamentals
- Templates
- Services
- Routing

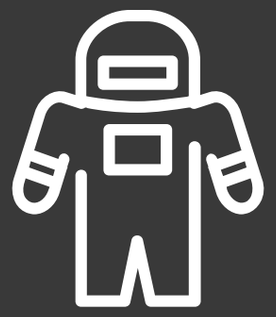
The Angular 2 Big Picture

The Demo Application

- A simple web application with basic features
- We will be building out a new **widgets** feature
- Feel free to use the existing code as a reference point
- Please explore! Don't be afraid to try new things!


localhost:3000/home
localhost:3000/home
AWESOMELOGO
Home
About
Experiments
Home Page
This is the about home body
Home: Hello Message
Hello Message
Update Message

http://bit.ly/fem-ng2-simple-app



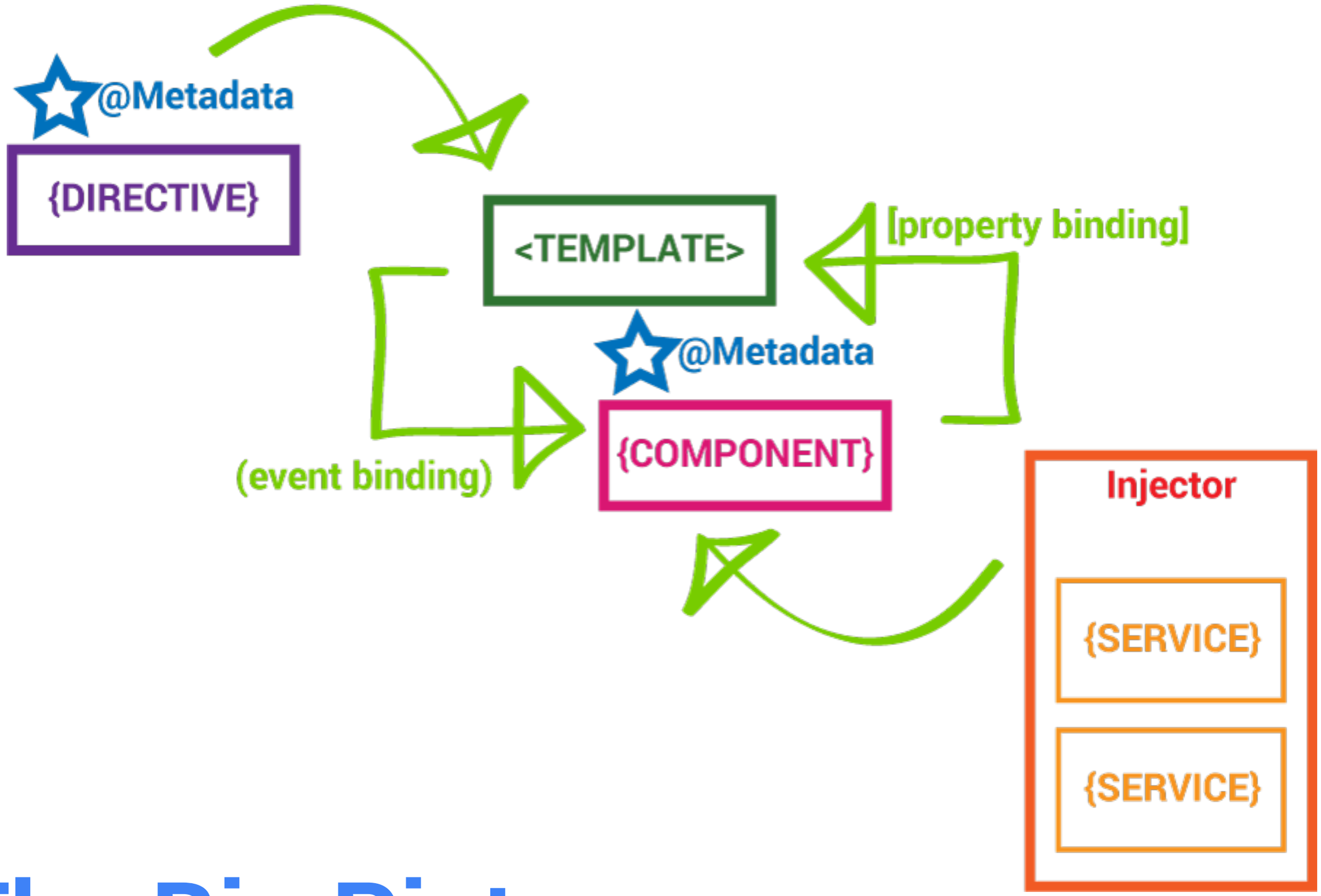
http://onehungrymind.com/fem-examples/

http://bit.ly/fem-ng2-no-ts

So why Angular 2?

Why Angular 2?

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



The Big Picture

The Main Building Blocks

- Module
- Component
- Metadata
- Template
- Data Binding
- Service
- Directive
- Dependency Injection

Bootstrapping the App

- Import the **bootstrap** module
- Import your top-level component
- Import application dependencies
- Call **bootstrap** and pass in your top-level component as the first parameter and an array of dependencies as the second

```
import {bootstrap} from 'angular2/platform/browser';
import {ROUTER_PROVIDERS} from 'angular2/router';
import {AppComponent} from './app.component';

bootstrap(AppComponent, [
  ROUTER_PROVIDERS
]);
```

Bootstrap

Module

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

```
// In home.component.ts
export class HomeComponent { }

// In app.component.ts
import {HomeComponent} from './home/home.component';
```

Module

Component

- Components are just ES6 classes
- Providers (Services) are injected in the constructor
- Need to explicitly define providers and directives within the component decoration
- Hook into the component lifecycle with hooks
- Properties and methods of the component class are available to the template

```
export class HomeComponent implements OnInit{
  title: string = 'Home Page';
  body:  string = 'This is the about home body';
  message: string;

  constructor(private _stateService: StateService) { }

  ngOnInit() {
    this.message = this._stateService.getMessage();
  }

  updateMessage(m: string): void {
    this._stateService.setMessage(m);
  }
}
```

Component

Metadata

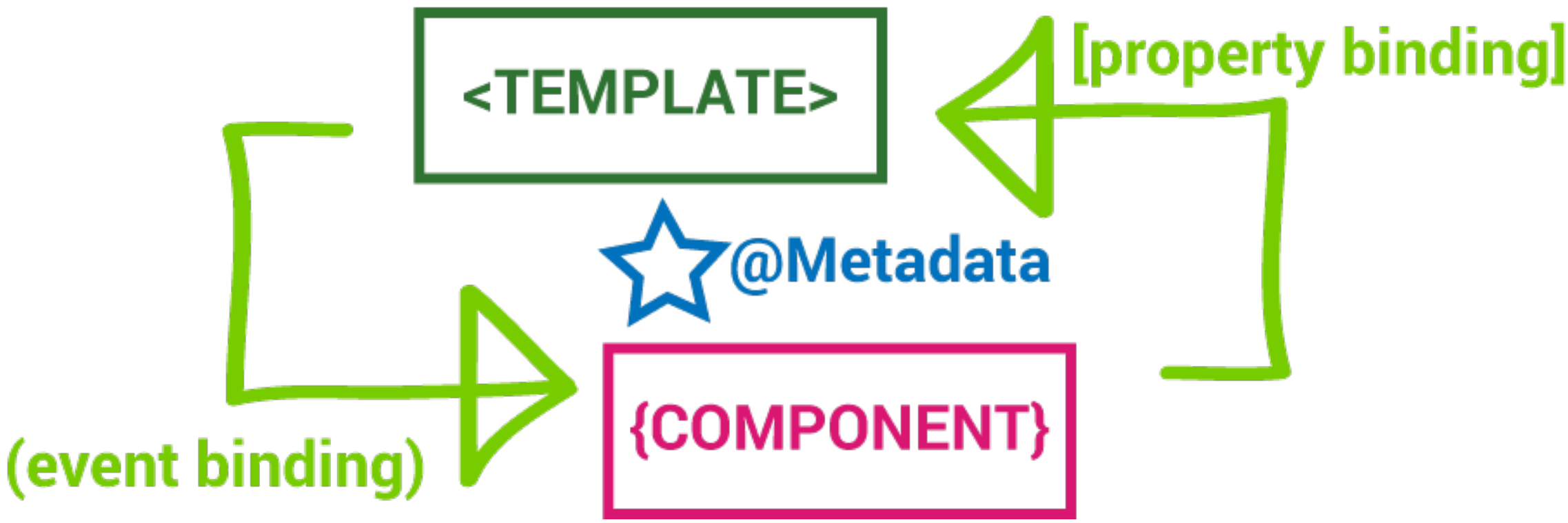
- Metadata allows Angular to process a class
- We can attach metadata with TypeScript using decorators
- Decorators are just functions
- Most common is the **@Component()** decorator
- Takes a config option with the selector, template(Url), providers, directives, pipes and styles

```
@Component({
  selector: 'home',
  templateUrl: 'app/home/home.component.html'
})
export class HomeComponent{ }
```

Metadata

Template

- A template is HTML that tells Angular how to render a component
- Templates include data bindings as well as other components and directives
- Angular 2 leverages native DOM events and properties which dramatically reduces the need for a ton of built-in directives
- Angular 2 leverages shadow DOM to do some really interesting things with view encapsulation


<TEMPLATE>
[property binding]
@Metadata
{COMPONENT}
(event binding)

Template

```
@Component({
  selector: 'experiment',
  templateUrl: './experiment.detail.component.html',
  styles: [`
    .experiment {
      cursor: pointer;
      outline: 1px lightgray solid;
      padding: 5px;
      margin: 5px;
    }
  `]
})
```

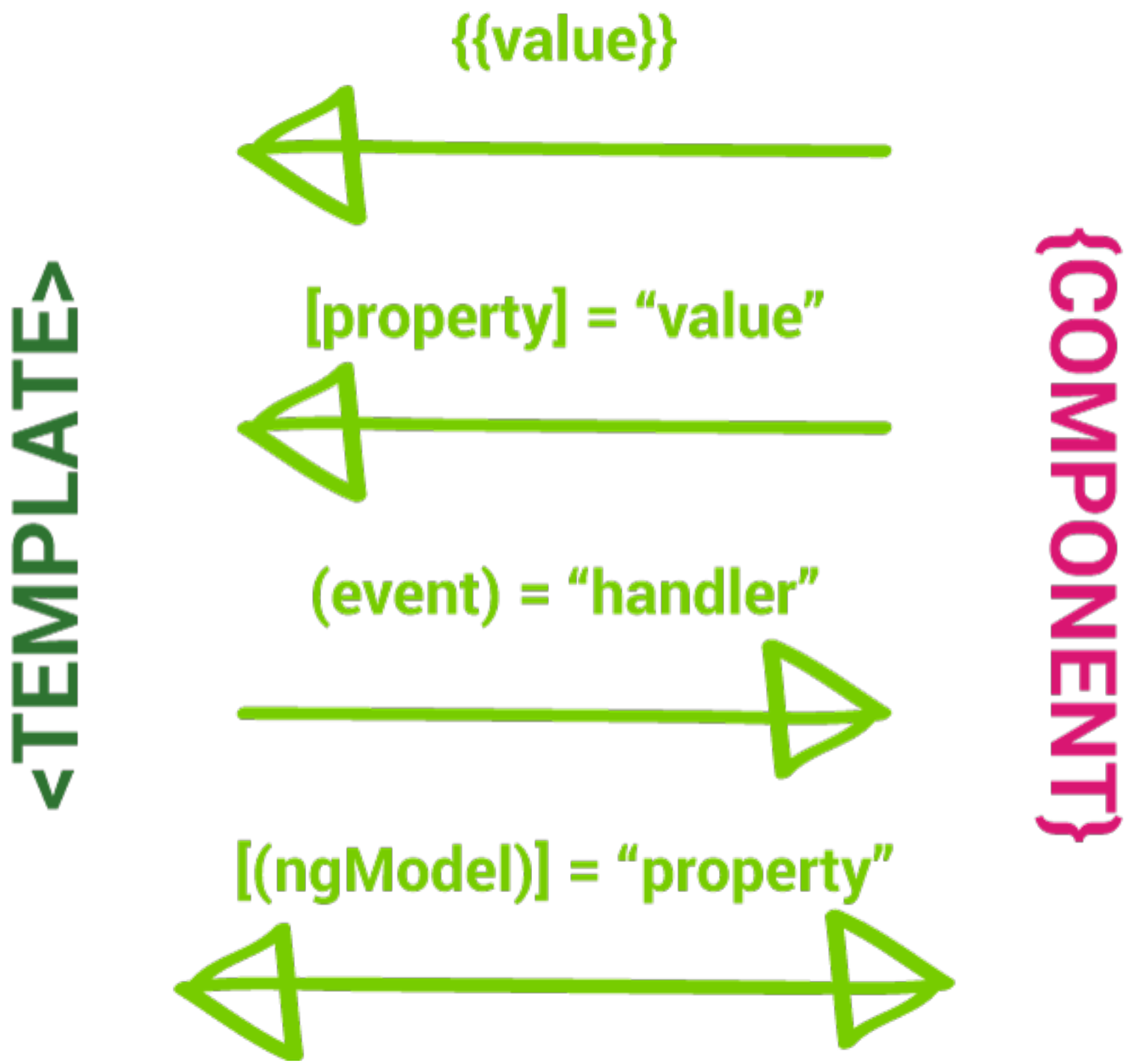
Template

```
@Component({
  selector: 'experiment',
  template: `
  <div class="experiment" (click)="doExperiment()">
    <h3>{{ experiment.name }}</h3>
    <p>{{ experiment.description }}</p>
    <p><strong>{{experiment.completed}}</strong></p>
  </div>
  `,
  styles: [`
    .experiment {
      cursor: pointer;
      outline: 1px lightgray solid;
      padding: 5px;
      margin: 5px;
    }
  `]
})
```

Template

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

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

Data Binding

Service

- A service is just a class
- Should only do one specific thing
- Take the burden of business logic out of components
- Decorate with @Injectable when we need to inject dependencies into our service

```
import {Injectable} from 'angular2/core';
import {Experiment} from './experiment.model';

@Injectable()
export class ExperimentsService {
  private experiments: Experiment[] = [];

  getExperiments(): Experiment[] {
    return this.experiments;
  };
}
```

Service

Directive

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

```
import { Directive, ElementRef } from 'angular2/core';

@Directive({
  selector: '[femBlinker]'
})

export class FemBlinker {
  constructor(element: ElementRef) {
    let interval = setInterval(() => {
      let color = element.nativeElement.style.color;
      element.nativeElement.style.color
        = (color === '' || color === 'black') ? 'red' : 'black';
    }, 300);

    setTimeout(() => {
      clearInterval(interval);
    }, 10000);
  }
}
```

Directive

Dependency Injection

- Supplies instance of a class with fully-formed dependencies
- Maintains a container of previously created service instances
- To use DI for a service, we register it as a provider in one of two ways: when bootstrapping the application, or in the component metadata

```
// experiments.service.ts
import {Injectable} from 'angular2/core';

@Injectable()
export class ExperimentsService { }

// experiments.component.ts
import {ExperimentsService} from '../common/
experiments.service';
import {StateService} from '../common/state.service';

export class ExperimentsComponent {
  constructor(
    private _stateService: StateService,
    private _experimentsService: ExperimentsService) {}
}
```

Dependency Injection

Change Detection

- Checks for changes in the data model so that it can re-render the DOM
- Changes are caused by events, XHR, and timers
- Each component has its own change detector
- We can use **ChangeDetectionStrategy.OnPush** along with immutable objects and/or observables.
- We can tell Angular to check a particular component by injecting **ChangeDetectorRef** and calling **markForCheck()** inside the component

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

export interface AppStore {
  items: Item[];
  selectedItem: Item;
};
```

Code Sample

Testing

- Angular wraps Jasmine methods
- Import all necessary Jasmine methods from **angular2/testing**
- Import the classes to test
- Include providers by importing **beforeEachProviders** and then calling it with a method that returns an array of imported providers
- Inject providers by calling **inject([arrayOfProviders], (providerAliases) => {})** inside a **beforeEach** or **it** block

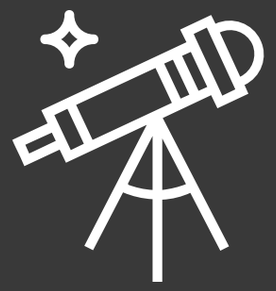
```
import { describe, it, expect } from 'angular2/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  it('should be a function', () => {
    expect(typeof AppComponent).toBe('function');
  });
});
```

Tests!

Architectural Best Practices

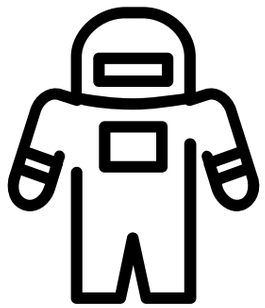
- Include all files pertinent to a component in the same folder
- Remember CIDER for creating components: (Create class, Import dependencies, Decorate class, Enhance with composition, Repeat for sub-components
- Keep templates small enough to put in the main component file directly
- Delegate business logic from the component to a provider
- Don’t be afraid to split a component up if it is growing too large
- Constantly consider change detection implications as you develop an app

Demonstration



Challenges

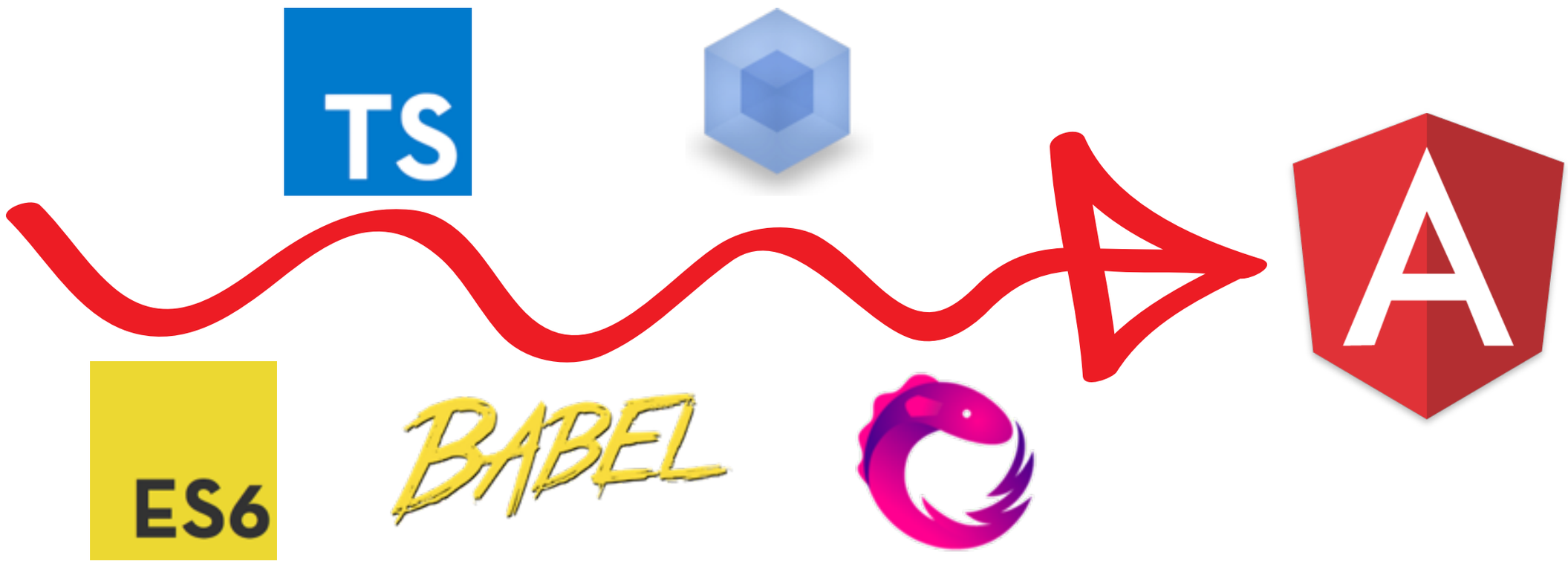
- Make sure that you can run the sample application
- Identify the major Angular 2 pieces in the sample application
- Add a new property to one of the feature components and bind to it in the view
- Add a new property to the **StateService** and consume it in a component
- **BONUS** Create an interface for the new property and type it in the **StateService** and your consuming component



Prerequisite Primer in Tooling

Tooling Primer

- Module Loading
- Webpack
- ES6
- ES5
- TypeScript
- Typings

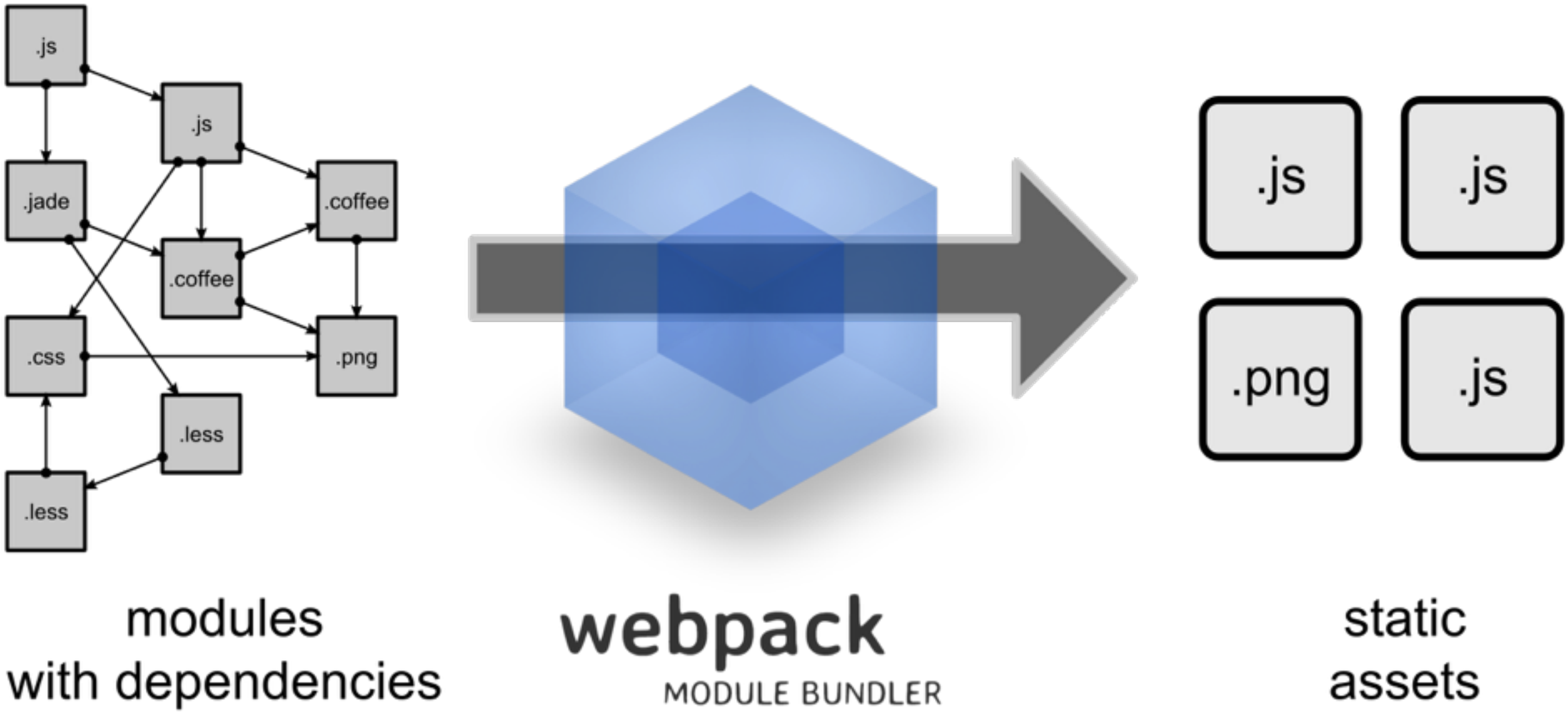

TS
ES6
BABEL

Module Loading

- Modular code is not required to develop with Angular, but it is recommended
- Allows us to easily use specific parts of a library
- Erases collisions between two different libraries
- We don’t have to include script tags for everything
- Because modules are not supported, we have to translate a module (file) to a pseudo module (wrapped function)

Webpack

- One of the most popular module loaders
- Allows us to include any sort of file as a module (CSS, JSON, etc.)
- Useful not only for module loading, but also the entire build process


.js
.js
.jade
.coffee
.coffee
.css
.png
.less
.less
.js
.js
.png
.js
modules
with dependencies
webpack
MODULE BUNDLER
static
assets

Webpack

ES6

- ES6/ES2015 is the latest standard for Javascript
- Comes with many helpful additions, such as a module system, new array methods, classes, multi-line templates, and arrow functions
- The most important features for us are modules and classes
- Although we don’t have to use it, it greatly enhances the development experience
- **Classes and modules FTW!**

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `(${this.x}, ${this.y})`;
  }
}
```

ES6 Class

TypeScript

- Is a typed superset of Javascript
- Is a compiled language, so it catches errors before runtime
- Includes the features of ES6 but with types, as well as better tooling support
- TypeScript allows us to decorate our classes via **@<Decorator>** the syntax
- **Classes, modules, types, interfaces and decorators FTW!**

```typescript
interface ClockInterface {
  currentTime: Date;
  setTime(d: Date);
}

class Clock implements ClockInterface  {
  currentTime: Date;
  setTime(d: Date) {
    this.currentTime = d;
  }
  constructor(h: number, m: number) { }
}
```

TypeScript Class

Typings

- We use the **typings** NPM package to handle the type definitions associated with third-party libraries
- By creating a **postinstall** script in **package.json**, we can install the appropriate typings immediately after all NPM packages have been downloaded
- To install a definition file for a particular library, run **typings install -g <package> --ambient --save**
- Treat the **typings.json** file just like the **package.json** file

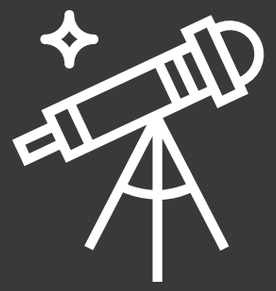
Angular 2 with ES6

- Almost the same as TypeScript without types and interfaces
- Define dependency paramenters explicitly for DI to work properly
- Use the same build system, just switch out your transpiler (babel). Or use your TypeScript compiler and just not use TypeScript features
- Babel has experimental features that TypeScript does not

Angular 2 with ES5

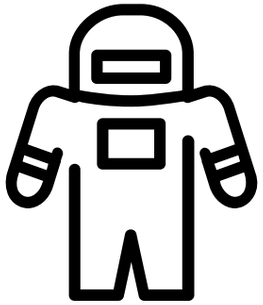
- Supported natively
- No module system like angular 1.x. Use IIFE’s or other 3rd party module system
- Exposes a global ng namespace with methods to build application
- No need for a build system or transpiler
- No type files or configs
- Documentation is lacking
- ot recommended

Demonstration



Challenges

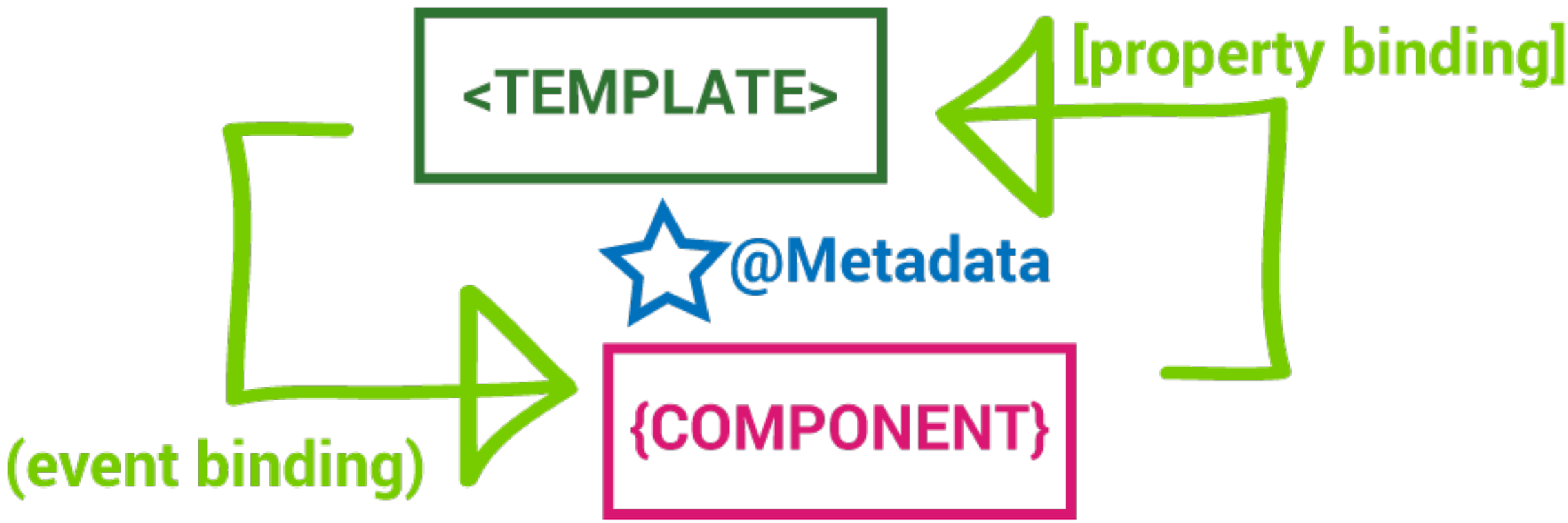
- We'll play this by ear. :D



Component Fundamentals

Component Fundamentals

- Class
- Import
- Decorate
- Enhance
- Repeat
- Lifecycle Hooks


<TEMPLATE>
[property binding]
@Metadata
{COMPONENT}
(event binding)

Component

Class !== Inheritance

Class

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

```
export class ExperimentsComponent { }
```

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 'angular2/core';

export class ExperimentsComponent {}
```

Import

Decorate

- We turn our class into something Angular 2 can use by decorating it with a Angular specific metadata
- Use the **@<decorator>** syntax to decorate your classes
- The most common class decorators are **@Component**, **@Injectable**, **@Directive** and **@Pipe**
- You can also decorate properties and methods within your class
- The two most common member decorators are **@Input** and **@Output**

```
import {Component} from 'angular2/core';

@Component({
  selector: 'experiments',
  templateUrl: './experiments.component.html'
})
export class ExperimentsComponent {}
```

Decorate

Enhance

- This is an iterative process that will vary on a per-case basis but the idea is to start small and build your component out
- Enhance with composition by adding methods, inputs and outputs, injecting services, etc.
- Remember to keep your components small and focused

```
import {Component} from 'angular2/core';
import {Experiment} from '../common/experiment.model';
import {ExperimentsService} from '../common/experiments.service';
import {StateService} from '../common/state.service';

@Component({
  selector: 'experiments',
  templateUrl: 'app/experiments/experiments.component.html'
})
export class ExperimentsComponent {
  title: string = 'Experiments Page';
  body: string = 'This is the about experiments body';
  message: string;
  experiments: Experiment[];

  constructor(
    private _StateService: StateService,
    private _ExperimentsService: ExperimentsService) {}

  updateMessage(m: string): void {
    this._StateService.setMessage(m);
  }
}
```

Enhance

Repeat

- Angular provides a framework where building subcomponents is not only easy, but also strongly encouraged
- If a component is getting too large, do not hesitate to break it into separate pieces and repeat the process

```
import {ExperimentDetailComponent}
  from './experiment-details/experiment.detail.component';

@Component({
  selector: 'experiments',
  templateUrl: 'app/experiments/experiments.component.html',
  directives: [ExperimentDetailComponent]
})
export class ExperimentsComponent { }
```

Repeat

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
- Only available in TypeScript
- 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

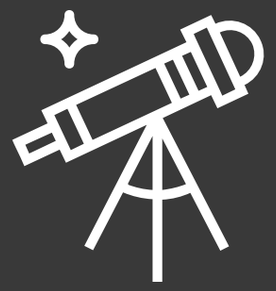
Lifecycle Hooks (cont.)

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

```
import {Component, OnInit} from 'angular2/core';

export class ExperimentsComponent implements OnInit {
  constructor(
    private _StateService: StateService,
    private _ExperimentsService: ExperimentsService) {}

  ngOnInit() {
    this.experiments = this._ExperimentsService.getExperiments();
    this.message = this._StateService.getMessage();
  }
}
```

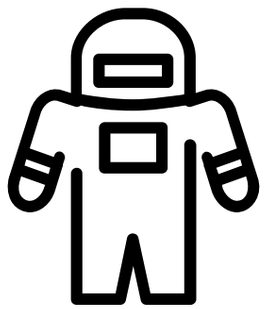
Lifecycle Hooks

Demonstration



Challenges

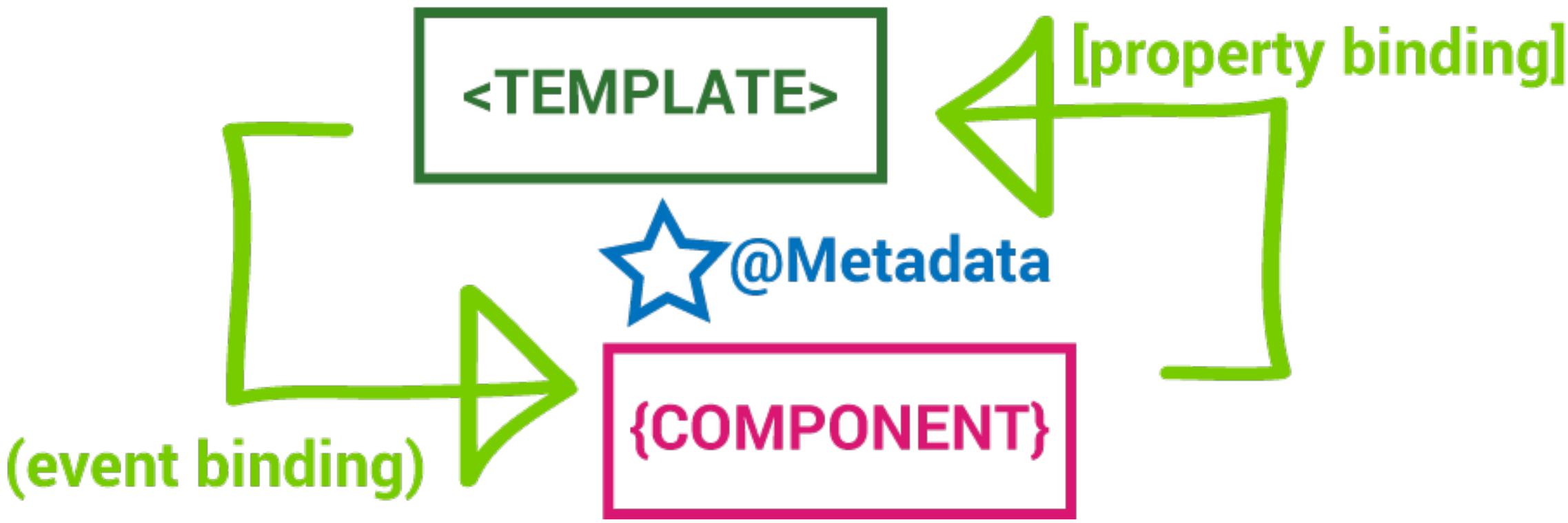
- Create the file structure for a new **widgets** feature
- Create the ES6 class for the **widgets** component
- Import the appropriate modules into the **widgets** component
- Decorate the **widgets** component to use the **widgets** template
- Display the **widgets** component in the **home** component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



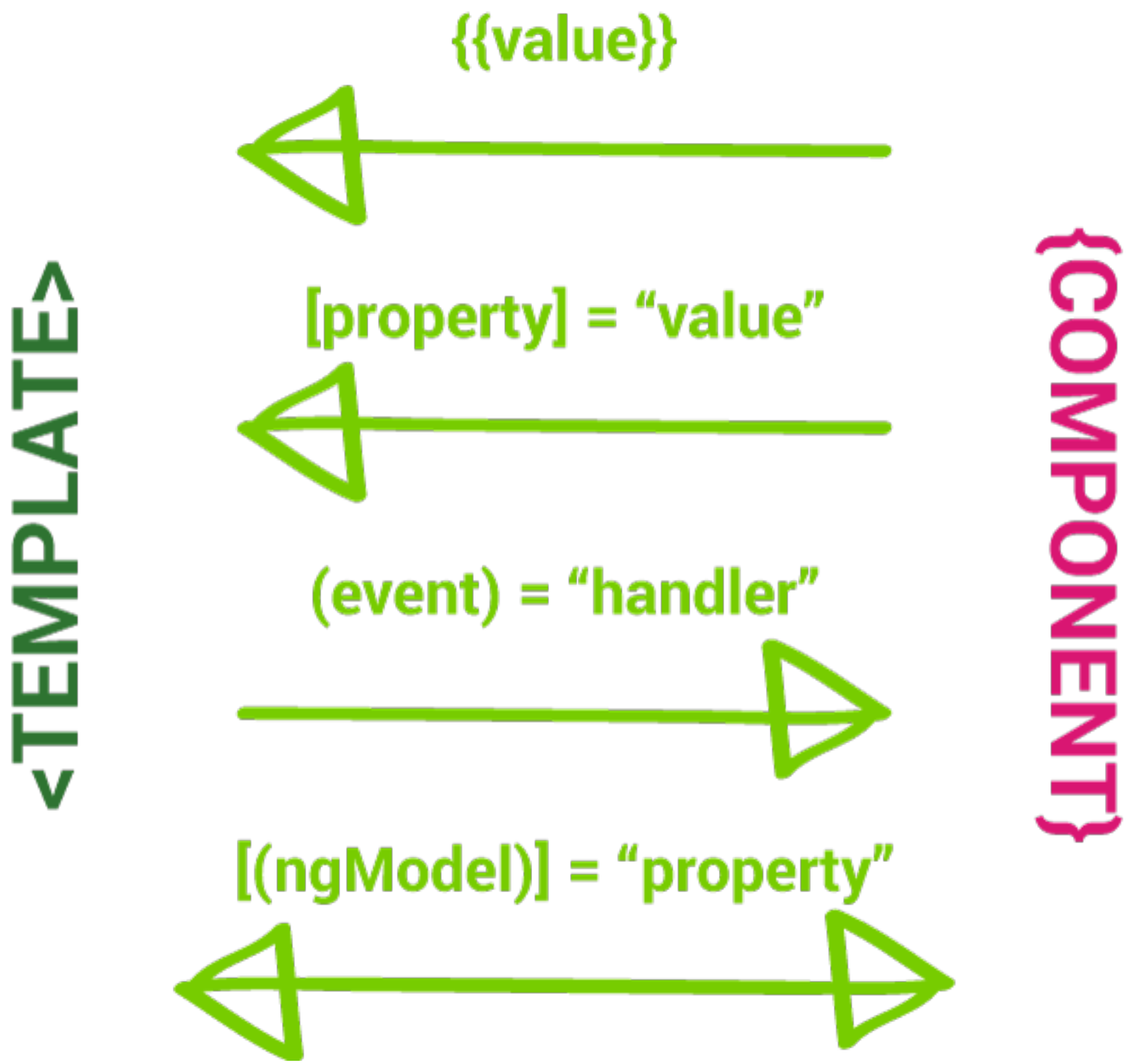
Templates

Templates

- Interpolation
- Method Binding
- Property Binding
- Two Way Binding
- Hashtag Operator
- Asterisk Operator
- Elvis Operator (?.)


<TEMPLATE>
[property binding]
@Metadata
{COMPONENT}
(event binding)

Template



Data Binding

Interpolation

- Allows us to bind to component properties in out template
- Defined with the double curly brace syntax: **{{ propertyValue }}**
- We can bind to methods as well
- Angular converts interpolation to property binding

```
<span>{{interpolatedValue}}<span>
```

Interpolation

Property Bindings

- Flows data from the component to an element
- Created with brackets ****
- Canonical form is **bind-attribute** e.g. ****
- When there is no element property, prepend with **attr** e.g. **[attr.colspan]**

Property Bindings (cont.)

Don’t use the brackets if:

- the target property accepts a string value
- the string is a fixed value that we can bake into the template
- this initial value never changes

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

Property Bindings

Event Bindings

- Flows data from an element to the component
- Created with parentheses **<button (click)="foo()"></button>**
- Canonical form is **on-event** e.g. **<button on-click="foo()"></button>**
- Get access to the event object inside the method via **$event** e.g. **<button (click)="callFoo($event)"></button>**

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

Event Bindings

Two-way Bindings

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

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

Two-way Bindings

Asterisk Operator

- Asterisks indicate a directive that modifies the HTML
- It is syntactic sugar to avoid having to use template elements directly

```
<div *ngIf=”userIsVisible”>{{user.name}}</div>
```

```
<template [ngIf]="userIsVisible">
  <div>{{user.name}}</div>
</template>
```

Asterisk Bindings

Hashtag Operator

- The hashtag (#) defines a local variable inside our template
- Template variable is available on the same element, sibling elements, or child elements of the element on which it was declared
- To consume, simply use it as a variable without the hashtag

```
<p *ngFor="#name of names">{{name}}</p>
```

Hashtag Operator

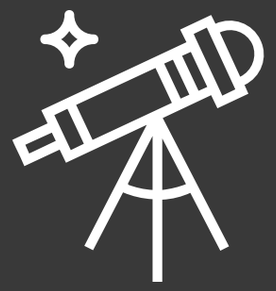
Elvis Operator

- 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 elvis operator is a simple, easy way to guard against null and undefined properties

```
<md-input-container>
  <label>Type to see the value</label>
  <input md-input type="text" #input />
</md-input-container>

<strong>{{input?.value}}</strong>
```

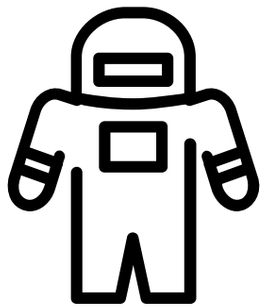
Elvis Operator

Demonstration



Challenges

- Flesh out the **widgets** template with the following:
  - A template expression via interpolation
  - A property binding
  - An event binding
  - A two-way binding
- **BONUS** use a local variable via #, use a built-in directive via *, and use the elvis operator with **setTimeout** to demonstrate a temporarily null or undefined value



Services

Services

- Services
- @Injectable
- Injecting Services

Just a Class

- Similarly to components, services are just a class
- We define our service's API by creating methods directly on the class
- We can also expose public properties on our class if need be

```
export class StateService {
  private _message = 'Hello Message';

  getMessage(): string {
    return this._message;
  };

  setMessage(newMessage: string): void {
    this._message = newMessage;
  };
}
```

Just a Class

@Injectable

- We decorate our service class with the **@Injectable** to mark our class as being available to the Injector for creation
- Injector will throw **NoAnnotationError** when trying to instantiate a class that does not have **@Injectable** marker

```
import {Injectable} from 'angular2/core';

@Injectable()
export class StateService {
   private _message = 'Hello Message';

  getMessage(): string {
    return this._message;
  };

  setMessage(newMessage: string): void {
    this._message = newMessage;
  };
}
```

@Injectable

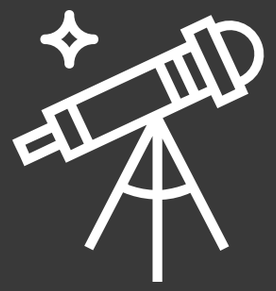
Injecting a Service

- Injecting a service is as simple as importing the service class and then defining it within the consumer's constructor parameters
- Just like components, we can inject dependencies into the constructor of a service
- There can be only one instance of a service type in a particular injector but there can be multiple injectors operating at different levels of the application's component tree. Any of those injectors could have its own instance of the service.

```
import {Component} from 'angular2/core';
import {StateService} from '../common/state.service';

@Component({
  selector: 'home',
  templateUrl: 'app/home/home.component.html'
})
export class HomeComponent {
  title: string = 'Home Page';
  body:  string = 'This is the about home body';
  message: string;

  constructor(private _stateService: StateService) { }

  ngOnInit() {
    this.message = this._stateService.getMessage();
  }

  updateMessage(m: string): void {
    this._stateService.setMessage(m);
  }
}
```

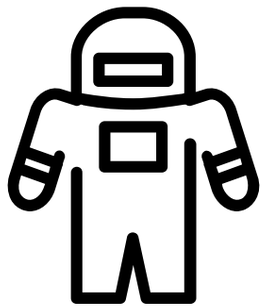
Injecting a Service

Demonstration



Challenges

- Create a **widgets** service class with a **widgets** collection
- Decorate it with **@Injectable()**
- Inject it into the **widgets** component and consume the **widgets** collection
- **BONUS** create a second helper service to use it within the **widgets** service



Router

Router

- Component Router
- Navigating Routes
- Route Parameters
- Query Parameters
- Child Routes

Component Router

- Import **ROUTE_PROVIDERS, ROUTE_DIRECTIVES**, and the **RouteConfig** decorator
- Set a base href in the head tag of your HTML like so: **<base href="/">**
- Configuration is handled via a decorator function (generally placed next to a component) by passing in an array of route definition objects
- Use the router-outlet directive to tell Angular where you want a route to put its template **<router-outlet></router-outlet>**

```
@RouteConfig([
  {path: '/home', name: 'Home', component: HomeComponent, useAsDefault: true},
  {path: '/about', name: 'About', component: AboutComponent},
  {path: '/experiments', name: 'Experiments', component: ExperimentsComponent}
])
export class AppComponent {}
```

@RouteConfig

```html
<div id="container">
  <router-outlet></router-outlet>
</div>
```

RouterOutlet

Navigating Routes

- Add a **routerLink** attribute directive to an anchor tag
- Bind it to a template expression that returns an array of route link parameters **<a [routerLink]="['Users']">Users**
- Navigate imperatively by importing **Router**, injecting it, and then calling **.navigate()** from within a component method
- We pass the same array of parameters as we would to the **routerLink** directive **this._router.navigate(['Users']);**

```html
<div id="menu">
  <a [routerLink]="['/Home']" class="btn">Home</a>
  <a [routerLink]="['/About']" class="btn">About</a>
  <a [routerLink]="['/Experiments']" class="btn">Experiments</a>
</div>
```

RouterLink

```
export class App {
  constructor(private _router: Router) {}
  navigate(route) {
    this._router.navigate([`/${route}`]);
  }
}
```

Router.navigate

Query Parameters

- Denotes an optional value for a particular route
- Do not add query parameters to the route definition **{ path:'/users', name: UserDetail, component: UserDetail }**
- Add as a parameter to the routerLink template expression just like router params: **<a [routerLink]="['Users', {id: 7}]"> {{user.name}} **
- Also accessed by injecting **RouteParams** into a component

```
<div>
  <button [routerLink]="['./MyComponent', {id: 1}]">
    My Component Link</button>
  <button [routerLink]="['./AnotherComponent', {queryParam: 'bar'}]">
    Another Component Link</button>
</div>
```

QueryParam

```
import { Component } from 'angular2/core';
import { RouteParams } from 'angular2/router';

@Component({
  selector: 'my-component',
  template: `<h1>my component ({{routeParams.get('id')}})!</h1>`
})

export class MyComponent {
  constructor(routeParams: RouteParams) {
    this.routeParams = routeParams;
  }
}
```

RouteParams

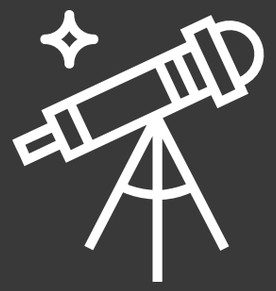
Child Routes

- Ideal for creating reusable components
- Components with child routes are “ignorant” of the parents’ route implementation
- In the parent route config, end the path with **/…**
- In the child config, set the path relative to the parent path
- If more than one child route, make sure to set the default route

```
@RouteConfig([
  {
    path:'/another-component/...',
    name: 'AnotherComponent',
    component: AnotherComponent
  }
])
export class App {}

@RouteConfig([
  {
    path:'/first',
    name: 'FirstChild',
    component: FirstSubComponent
  }
])
export class AnotherComponent {}
```

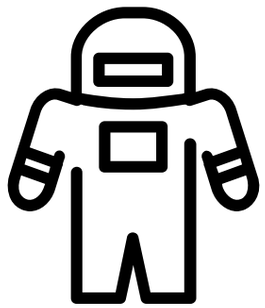
Child Routes

Demonstration



Challenges

- Create a route to the **widgets** feature
- Use **routeLink** to navigate to the **widgets** feature
- Create a method in the **home** component that imperatively navigates to that route
- Add both route parameters and query parameters to the **widgets** route
- **BONUS** create a **widget-item** child component with a child route definition



Resources

http://onehungrymind.com/

Thanks!