Build Better Apps

with Angular 2

Understand how to compose multiple, non-trivial features in Angular 2

Agenda

- Routing
- Component Composition
- Directives
- Forms
- Server Communication
- Pipes

ANGULAR 2 with NGRX

Item 1

This is a description

Item 2

This is a description

Item 3

This is a lovely item

Create New Item

Item Name

Enter a name

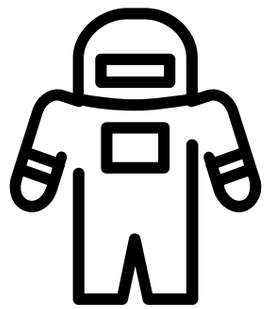
Item Description

Enter a description

CANCEL SAVE

Challenges

- Using the **items** feature as a reference, create the file structure for a **widgets** feature



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

```
<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']);**

```
<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

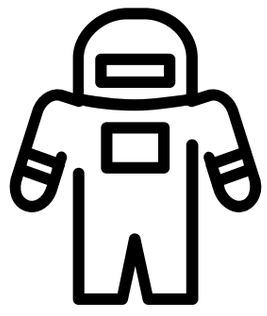
```
@RouteConfig([
  {
    path:'/another-component/...',
    name: 'AnotherComponent',
    component: AnotherComponent
  }
])
export class App {}
```

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

Child Routes

Challenges

- Create a route to the **widgets** feature
- Use **routeLink** to navigate to the **widgets** feature
- Create a method in the **items** component that imperatively navigates to that route
- Add both route parameters and query parameters to the **widgets** route



Component Composition

Component Composition

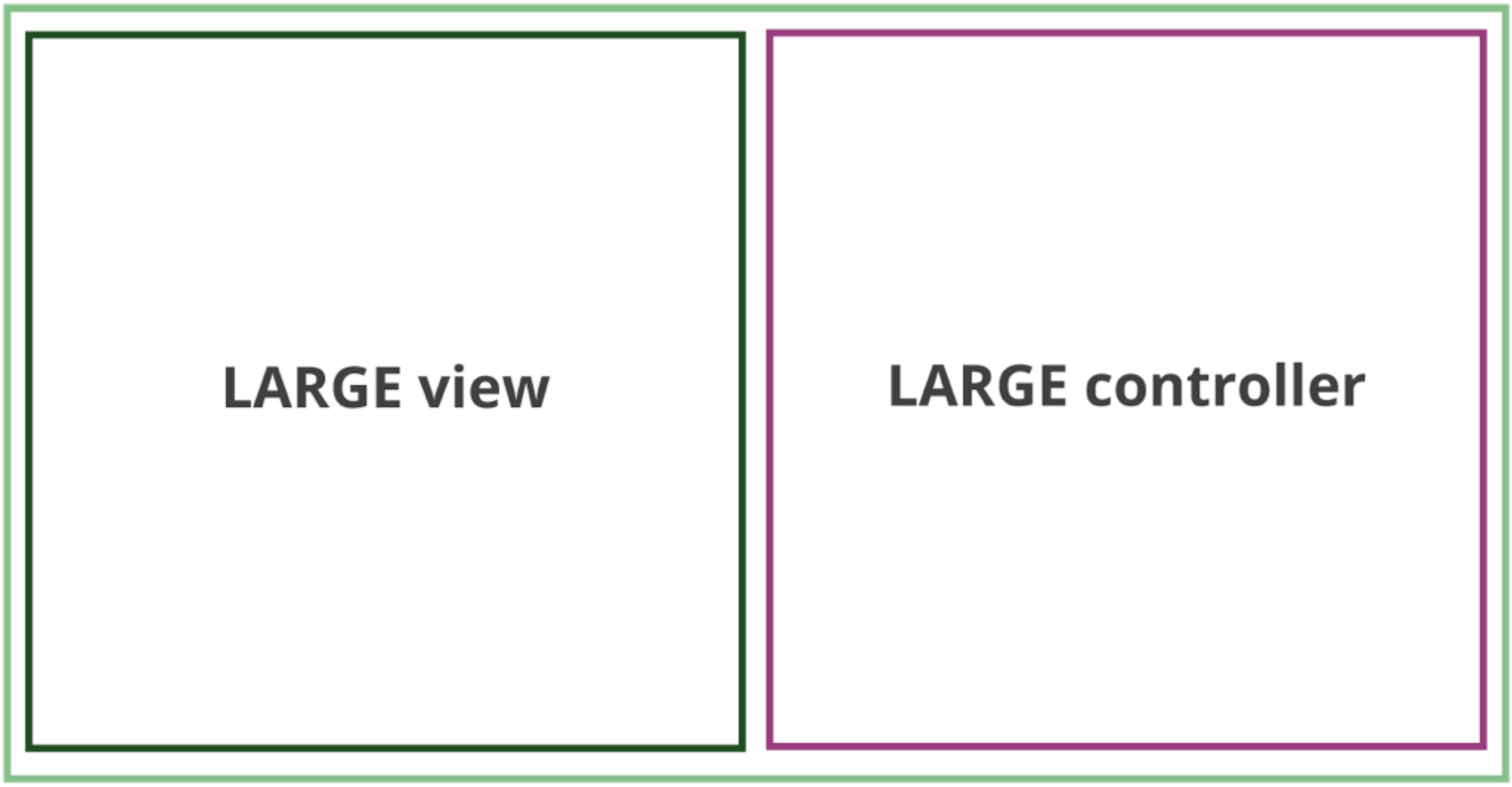
- Component System Architecture
- Clear contract with @Input and @Output
- @Input
- @Output
- Smart Components and Dumb Components
- View Encapsulation

Angular History Lesson

tiny app == tiny view + tiny controller

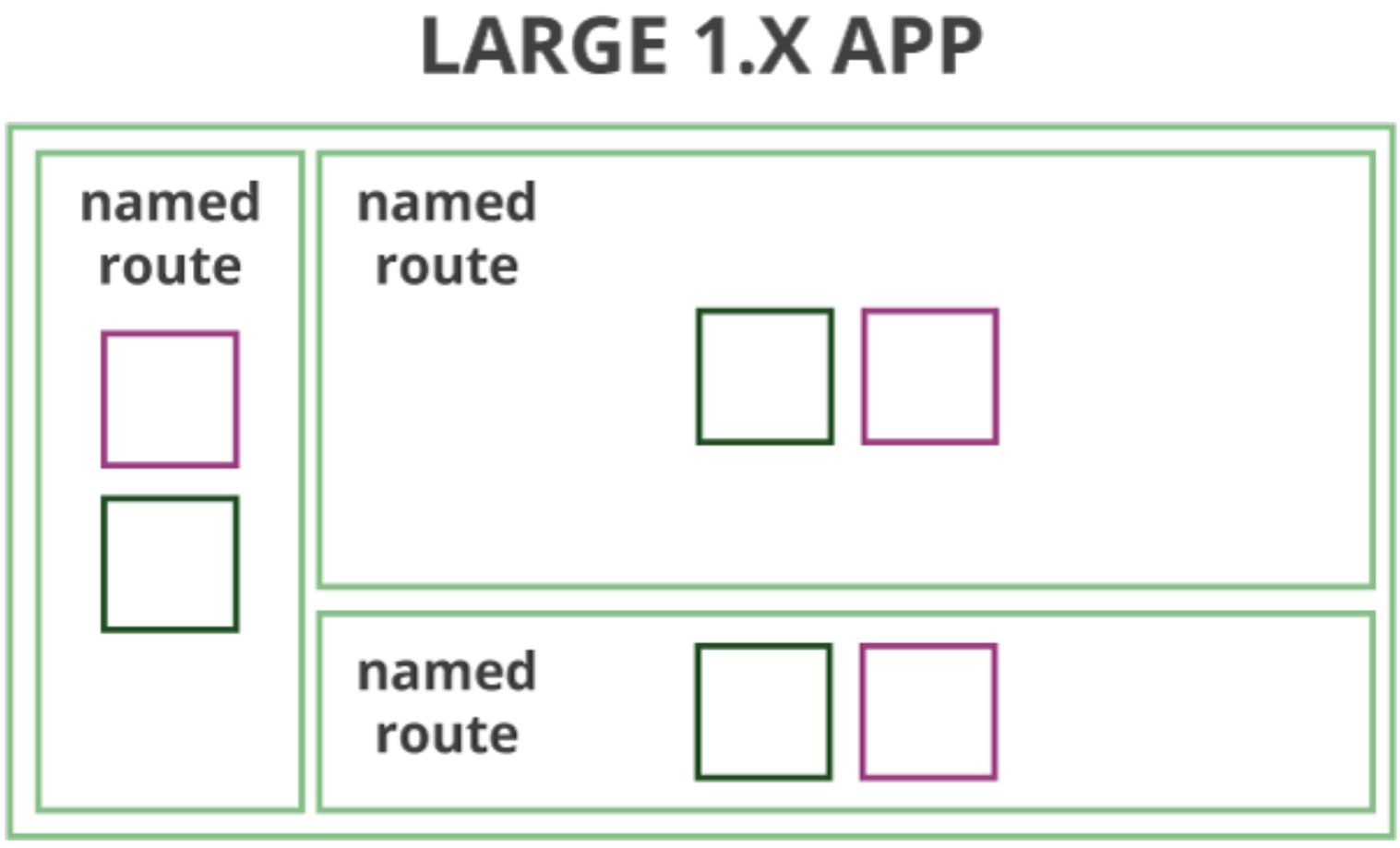
Hello Angular 1.x

Let's Get Serious

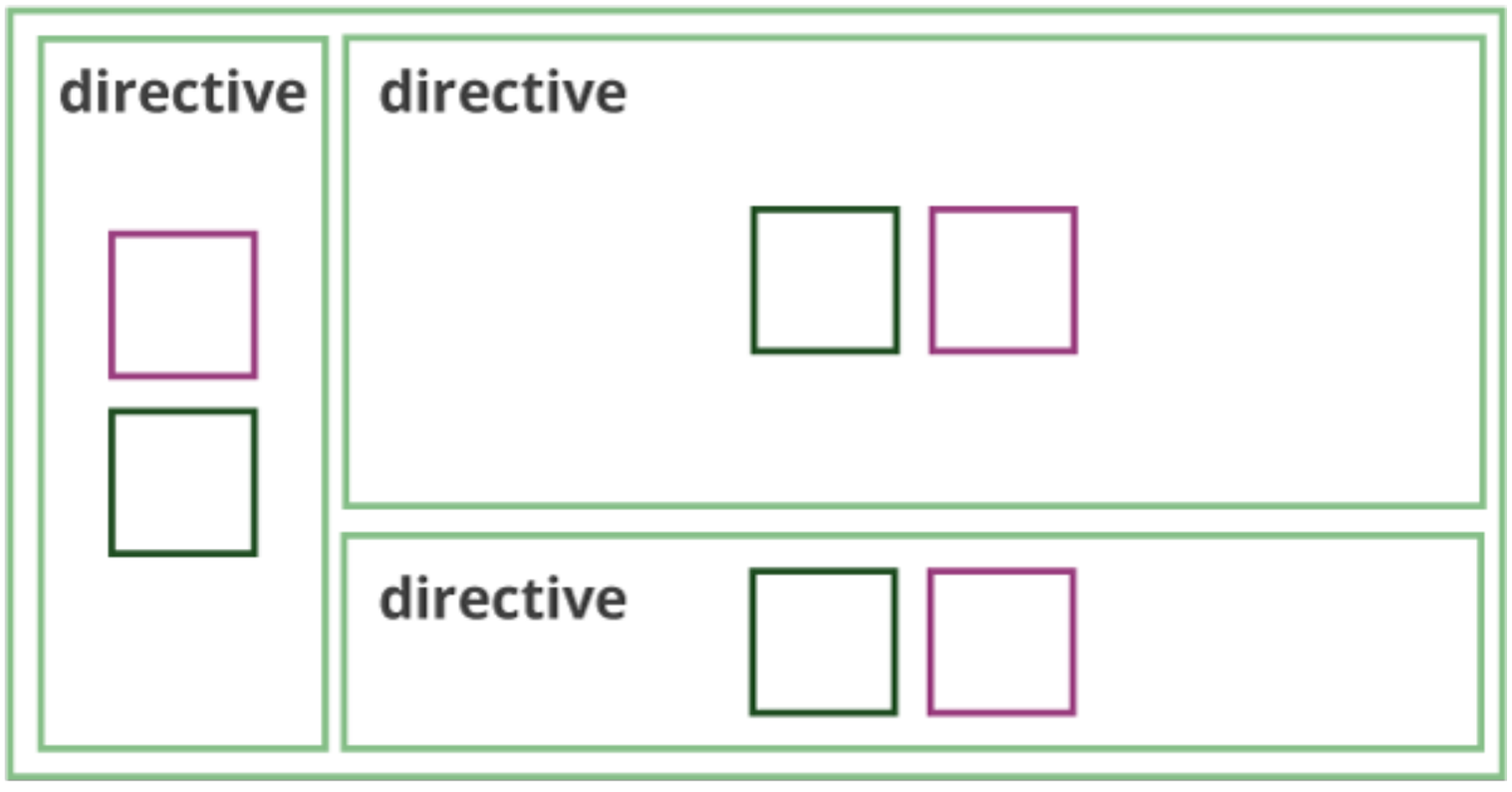


Let's Get Realistic

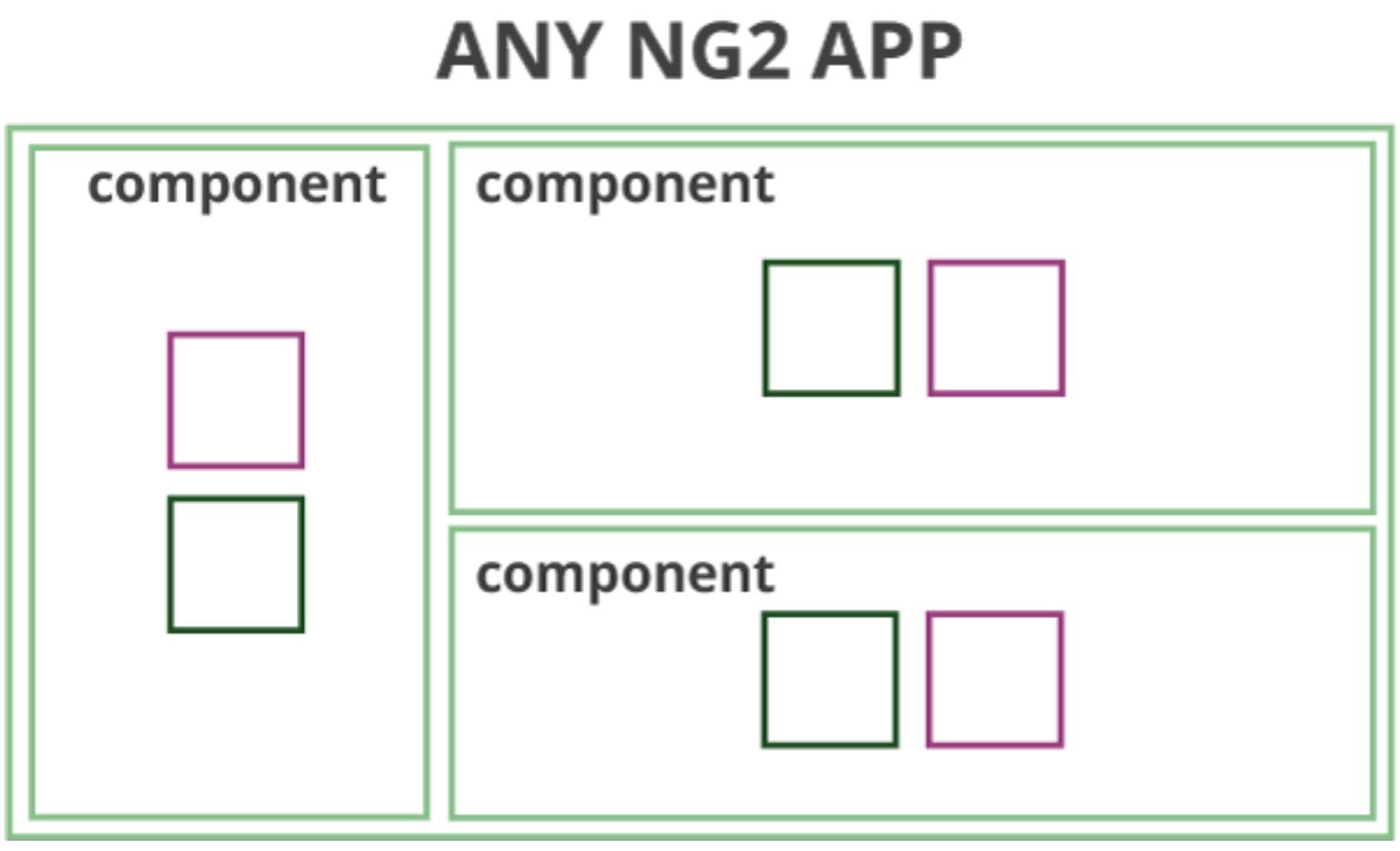
Two Solid Approaches



Named Routes



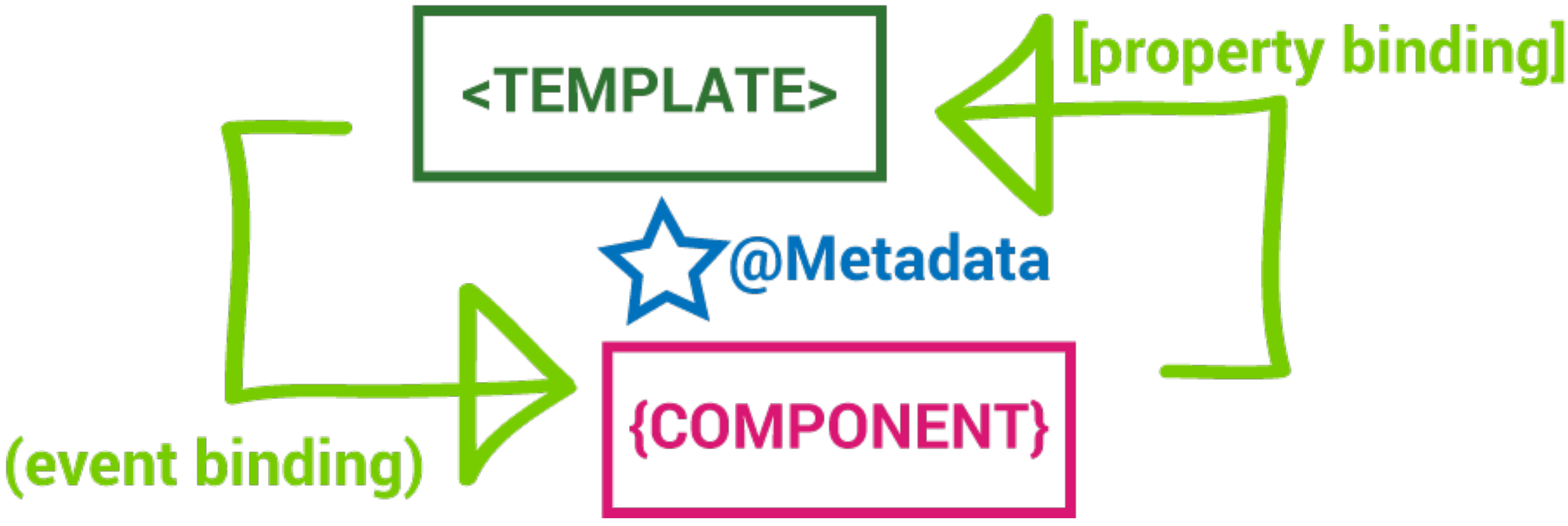
Directives



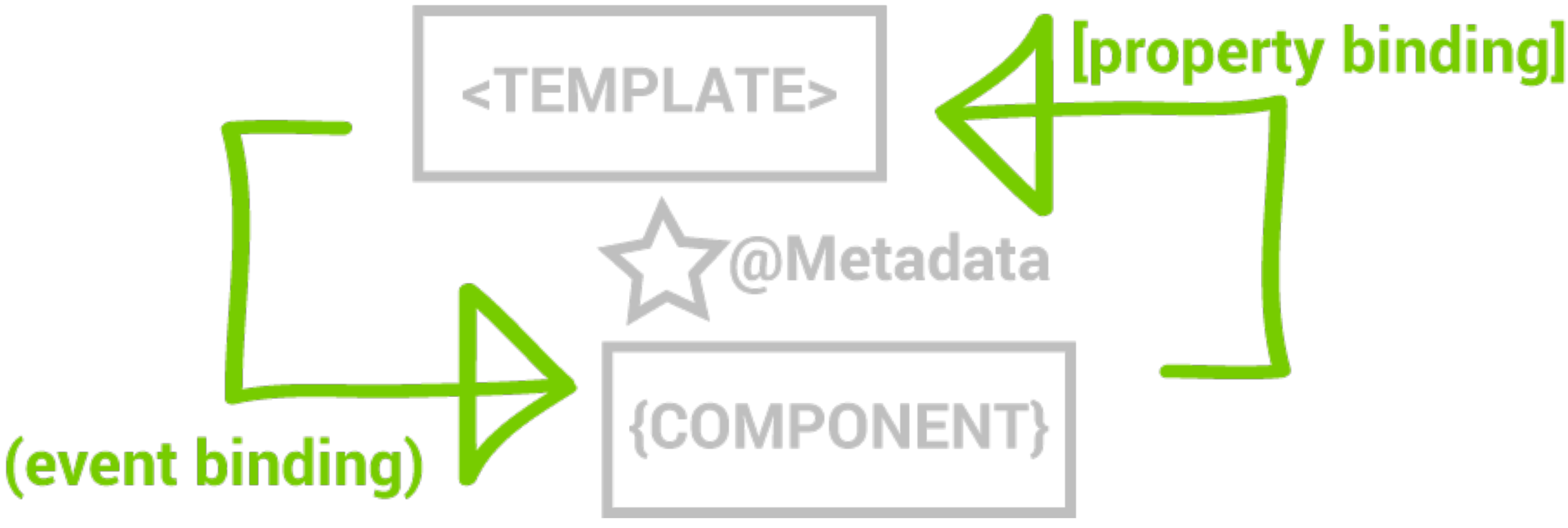
Components

Component System Architecture

- Components are small, encapsulated pieces of software that can be reused in many different contexts
- Angular 2 strongly encourages the component architecture by making it easy (and necessary) to build out every feature of an app as a component
- Angular components contain their own templates, styles, and logic so that they can easily be ported elsewhere



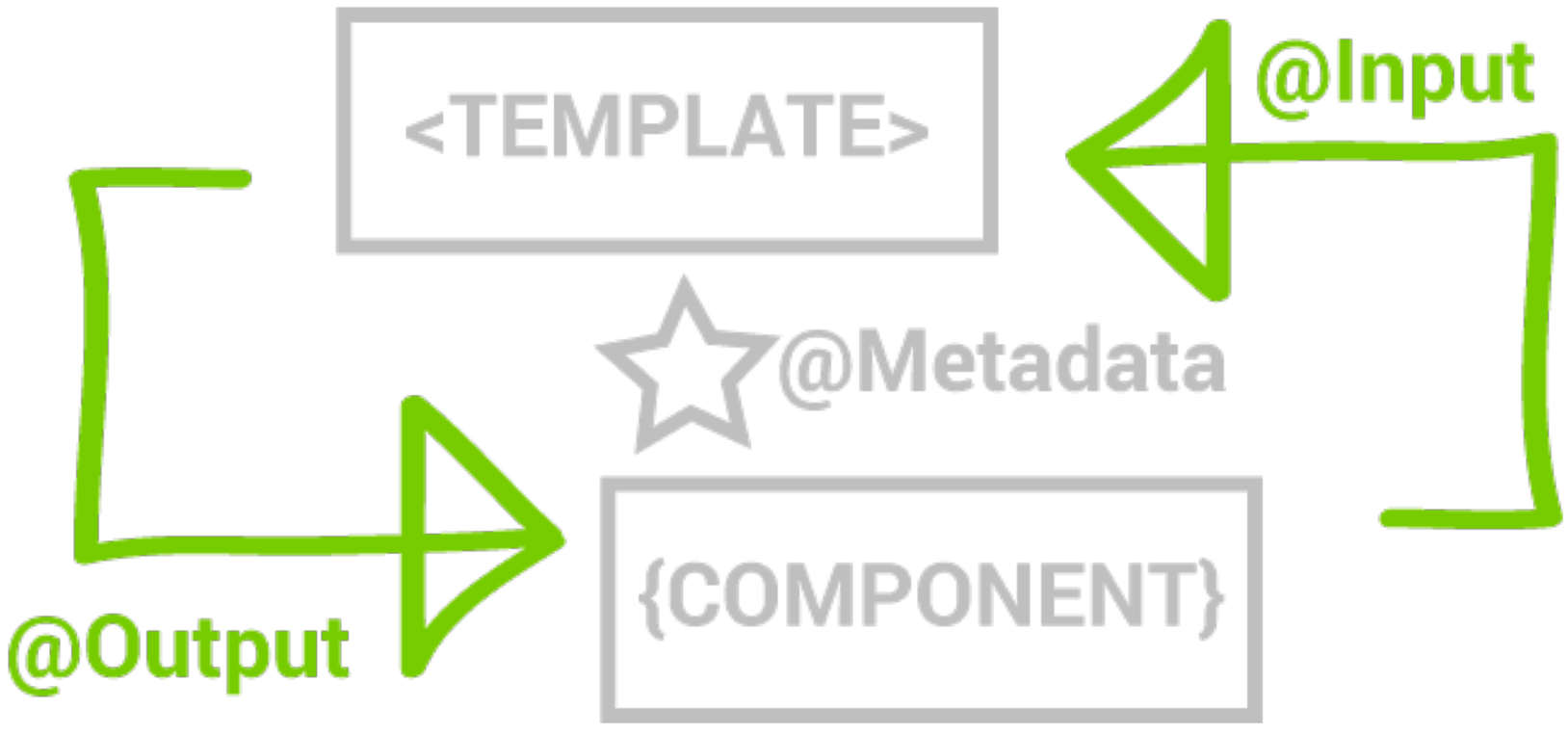
Data Binding



Custom Data Binding

Component Contracts

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



Component Contract

```
<div>
  <item-detail
    (saved)="saveItem($event)"
    (cancelled)="resetItem($event)"
    [item]="selectedItem">Select an Item</item-detail>
</div>
```

Component Contract

@Input

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

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

@Component({
  selector: 'my-component',
  template: `
    <div>Greeting from parent:</div>
    <div>{{greeting}}</div>
  `
})

export class MyComponent {
  @Input() greeting: String = 'Default Greeting';
}
```

@Input

```
import { Component } from 'angular2/core';
import { MyComponent } from './components/my.component';

@Component({
  selector: 'app',
  template: `
    <my-component [greeting]="greeting"></my-component>
    <my-component></my-component>
  `,
  directives: [ MyComponent ]
})

export class App {
  greeting: String = 'Hello child!';
}
```

@Input

@Output

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

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

@Component({
  selector: 'my-component',
  template: `<button (click)="greet()">Greet Me</button>`
})
export class MyComponent {
  @Output() greeter: EventEmitter = new EventEmitter();

  greet() {
    this.greeter.emit('Child greeting emitted!');
  }
}
```

@Output

```
@Component({
  selector: 'app',
  template: `
    <div>
      <h1>{{greeting}}</h1>
      <my-component (greeter)="greet($event)"></my-component>
    </div>
  `,
  directives: [MyComponent]
})

export class App {
  greet(event) {
    this.greeting = event;
  }
}
```

@Output

Smart and Dumb Components

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

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

Dumb Component

```
export class App implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() { }

  resetItem() { }

  selectItem(item: Item) { }

  saveItem(item: Item) { }

  replaceItem(item: Item) { }

  pushItem(item: Item) { }

  deleteItem(item: Item) { }
}
```

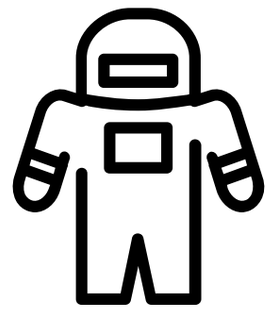
Smart Component

View Encapsulation

- Allows styles to be scoped only to one single component
- There are three types of view encapsulation in Angular 2
- **ViewEncapsulation.None:** styles are global like any other HTML document
- **ViewEncapsulation.Emulated:** style scope is mimicked by adding attributes to the elements in each components' template
- **ViewEncapsulation.Native:** uses the Shadow DOM to insert styles

Challenges

- Create a dumb **widgets-list** and **item-details** component using **@Input** and **@Output**
- Create a **widgets** service and hardcode a **widgets** collection
- Consume the collection in the **widgets** component and pass it to the **widgets-list** component
- Select a **widget** from the **widgets-list**
- Display the selected **widget** in **item**-details



Directives

Directives

- What is a Directive?
- Attribute Directives
- Structural Directives
- Custom Directives
- Accessing the DOM

What is a Directives?

- A directive is responsible for modifying a dynamic template
- A component is a specific kind of directive with a template
- For the sake of conversation… a directive is a component without a template

Structural and Attribute Directives

Creating a Directive

Creating a Structural Directive

Accessing the DOM

- First import **ElementRef**: **import { ElementRef } from 'angular2/core'**
- Then inject it into the directive class constructor: **constructor(el: ElementRef) {}**
- Access properties directly on the directive's DOM **element by using element.nativeElement.property**: **el.nativeElement.style.backgroundColor = 'yellow';** or **el.nativeElement.innerText = 'Some Text';**

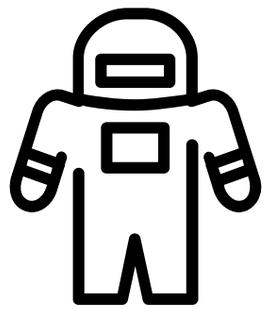
```
export class FemBlinker {
  constructor(private _element: ElementRef) {
    let interval = setInterval(() => {
      let color = _element.nativeElement.style.color;
      _element.nativeElement.style.color
        = (color === '' || color === 'black') ? 'red' : 'black';
    }, 300);

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

Accessing the DOM

Challenges

- Pending



Forms

Forms

- ngModel
- ngSubmit
- FormBuilder
- Validation

ngModel

- Implements Angular’s two-way binding syntax e.g. **[(ngModel)]**
- One of the few directives that actually uses two-way data binding
- Allows us to bind inputs to a model defined by an interface in TypeScript Interface: **export interface User { id: number; name: string; };**
- Component: **class UserCmp { user: User }**
- HTML: **<input type="text" [(ngModel)]="user.name" />**

Binding data to an Input

Getting an Input's State

Get State from a Group of Inputs

Leveraging formControls and formGroups

FormBuilder

ngSubmit

- Just like any other event binding
- Binds to the native submit event (generally by clicking a button with type of submit or by pressing enter)
- When form is submitted, calls whatever component method we pass it

Submitting a Form

Validation

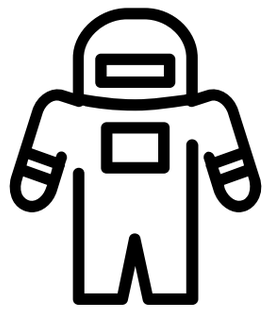
- Set a local template variable to the value **ngForm**
- Angular resets the local template variable to the **ngControl** directive instance. In other words, the local template variable becomes a handle on the **ngControl** object for this input box.

<input type="text" required [(ngModel)]="user.name" ngControl="name" #name="ngForm" >
**<div [hidden]="name.valid || name.pristine" class="alert alert-danger">Name is required **

Custom Validators

Challenges

- Create a form for **widget-details**
- Bind the form to a **widget** object sent in via **@Input**
- Submit the details of the form to the parent component
- BONUS Add in a custom validator



Server Communication

Server Communication

- The HTTP Module
- Methods
- Observable.toPromise
- Error Handling
- Header

The HTTP Module

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

The HTTP Module Methods

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

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

```
createItem(item: Item) {
  return this.http.post(`${BASE_URL}`, JSON.stringify(item), HEADER)
    .map(res => res.json())
    .toPromise();
}
```

HTTP Methods

```
updateItem(item: Item) {
  return this.http.put(`${BASE_URL}${item.id}`,
    JSON.stringify(item), HEADER)
    .map(res => res.json())
    .toPromise();
}
```

```
deleteItem(item: Item) {
  return this.http.delete(`${BASE_URL}${item.id}`)
    .map(res => res.json())
    .toPromise();
}
```

HTTP Methods

What is an Observable

- A lazy event stream which can emit zero or more events
- Composed of subjects and observers
- A subject performs some logic and notifies the observer at the appropriate times

Observable vs Promise

- Observables are lazy – they do not run unless subscribed to while promises run no matter what
- Observables can define both setup and teardown aspects of asynchronous behavior
- Observables are cancellable
- Observables can be retried, while a caller must have access to the original function that returned the promise in order to retry

Observable.subscribe

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

```
source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));
```

Observable.subscribe

Observable.toPromise

Diving into observables can be intimidating
We can chain any HTTP method (or any observable for that matter) with **toPromise**
Then we can use **.then** and **.catch** to resolve the promise as always

```
this.http.get('users.json')
  .toPromise()
  .then(res => res.json().data)
  .then(users => this.users = users)
  .catch(this.handleError);
```

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

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

Code Sample

Error Handling

We should **always** handle errors
Chain the **.catch** method on an observable
Pass in a callback with a single **error** argument

```
this.http.get('users.json')
  .catch(error => {
    console.error(error);
    return Observable.throw(error.json().error || 'Server error');
  });
```

```
getItems() {
  return this._http.get('fileDoesNotExist.json')
    .map(result => result.json())
    .catch(this.handleError);
}
```

```
private handleError (error: Response) {
  console.error(error);
  return Observable.throw(error.json().error || 'Server error');
}
```

Observable.catch

Headers

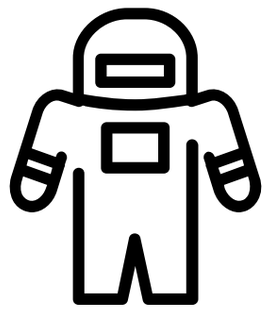
Import Headers and RequestOptions:
import {Headers, RequestOptions} from 'angular2/http';

Headers are an instance of the **Headers** class
Pass in an object with each header as a key-value pair
Then pass this **Headers** instance into a new **RequestOptions** instance

```
let headers = new Headers({ 'Content-Type': 'application/json' });
let options = new RequestOptions({ headers: headers });
this.http.post('users.json', '{}', options);
```

Challenges

- Create a method inside the service that uses HTTP to get **widgets.json**
- Return that method from the service and subscribe to it in the component so you can display the data inside the component's template
- Change the method to use promises and update the component accordingly
- **BONUS** create another method that **POST**s to **widgets.json**



Pipes

Pipes

- What are Pipes?
- Built-in Pipes
- Custom Pipes
- Async Pipes

What are Pipes?

A pipe takes in data as input and transforms it to a desired output

We use them in our templates with interpolation:
** User created on {{ created_at | date }} **

Include parameters to a pipe by separating them with a colon:
** User created on {{ created_at | date:"MM/dd/yy" }} **

Pipes are chain-able:
** User created on {{ created_at | date | uppercase }} **

Built-in Pipes

DatePipe

** User created on {{ user.created_at | date }} **

UpperCasePipe

** Middle Initial: {{ user.middle | uppercase }} **

LowerCasePipe

** Username: {{ user.username | lowercase }} **

CurrencyPipe

** Price Plan: {{ user.plan.price | currency }} **

PercentPipe

** Data Usage: {{ user.usage | percent }} **

Custom Pipes

Import the pipe decorator and **PipeTransform** interface:
import { Pipe, PipeTransform } from 'angular2/core';

Create a class that implements the **PipeTransform** interface and includes a **transform** method:

```
export class ReversePipe implements PipeTransform {
  transform(value:string, args:string[]) : any {
    return value.reverse();
  }
}
```

Async Pipe

Resolves async data (observables/promises) directly in the template

Skips the process of having to manually subscribe to async methods in the component and **then** setting those values for the template to bind to

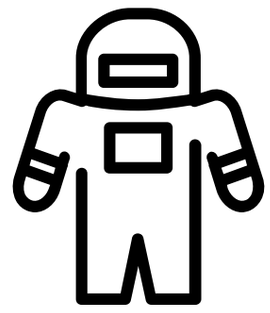
Component attribute:

```
asyncAttribute<string> = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Promise resolved'), 500);
})
```

Template: ** Async result: {{asyncAttribute | async}} **

Challenges

- Use two or more built-in pipes to transform data in the template
- Create a custom pipe that filters an array of strings based on a particular letter
- Create an asynchronous method or attribute on the component and bind to it in the template



Thanks!