Strong grasp on how to **construct** and **compose** features in Angular
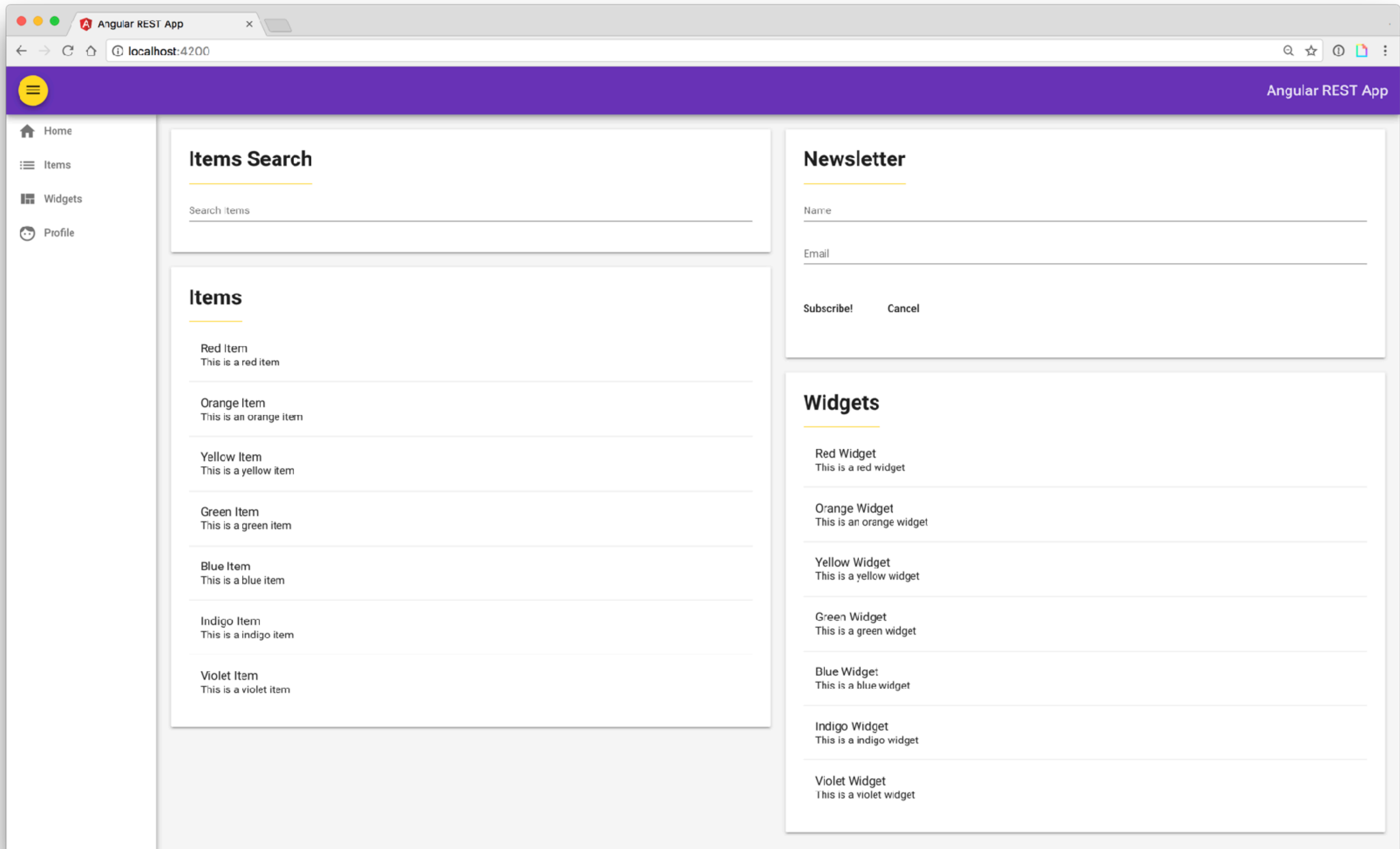
# Agenda

- ⃝ **Review Challenge**
- ⃝ **Template Driven Forms**
- ⃝ **Component Driven Architecture**
- ⃝ **Server Communication**
- ⃝ **Observable Fundamentals**
- ⃝ **Reactive Forms**
- ⃝ **Angular Animations**

# Getting **Started**

https://github.com/onehungrymind/angular-rest-app

**Angular REST App**

Home

Items

Widgets

Profile

## Items Search

Search Items

## Items

Red Item
This is a red item

Orange Item
This is an orange item

Yellow Item
This is a yellow item

Green Item
This is a green item

Blue Item
This is a blue item

Indigo Item
This is a indigo item

Violet Item
This is a violet item

## Newsletter

Name

Email

Subscribe!        Cancel

## Widgets

Red Widget
This is a red widget

Orange Widget
This is an orange widget

Yellow Widget
This is a yellow widget

Green Widget
This is a green widget

Blue Widget
This is a blue widget

Indigo Widget
This is a indigo widget

Violet Widget
This is a violet widget

# The Demo Application

- A simple RESTful master-detail application built using Angular and the Angular CLI
- We will be building out a new **widgets** feature
- Feel free to use the existing **items** feature as a reference point
- Please explore! Don't be afraid to try new things!

# Challenges

- Make sure you can run the application

REVIEW Time!

# The Angular Big Picture

module

routes

component

service

# What is the purpose of NgModule?

module

routes

component

service

# What do we use routes for?

module

routes

components

services

# What role does components play?

module

routes

components

services

# What do we use services for?

module

routes

components

services

# What mechanism is in play here?

```
import { Component, OnInit } from '@angular/core';
import { ItemsService, Item } from '../shared';

export class ItemsComponent implements OnInit {}
```

Modules

# What is the purpose of each of these properties?

```
@NgModule({
  declarations: [
    AppComponent,
    ItemsComponent,
    ItemsListComponent,
    ItemDetailComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    Ng2RestAppRoutingModule
  ],
  providers: [ItemsService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

@NgModule

# What is the entry point to our application?

```typescript
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';
import { AppModule } from './app/';

if (environment.production) {
  enableProdMode();
}

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

**Bootstrapping**

# What is the basic structure of a route?

```typescript
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ItemsComponent } from './items/items.component';

const routes: Routes = [
  {path: '',      component: ItemsComponent },
  {path: 'items', component: ItemsComponent},
  {path: '**',    component: ItemsComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})
export class Ng2RestAppRoutingModule { }
```

**Routing**

# What is does ** do?

```typescript
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ItemsComponent } from './items/items.component';

const routes: Routes = [
  {path: '',      component: ItemsComponent },
  {path: 'items', component: ItemsComponent},
  {path: '**',    component: ItemsComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})
export class Ng2RestAppRoutingModule { }
```

**Routing**

# What are the two main pieces of a component?
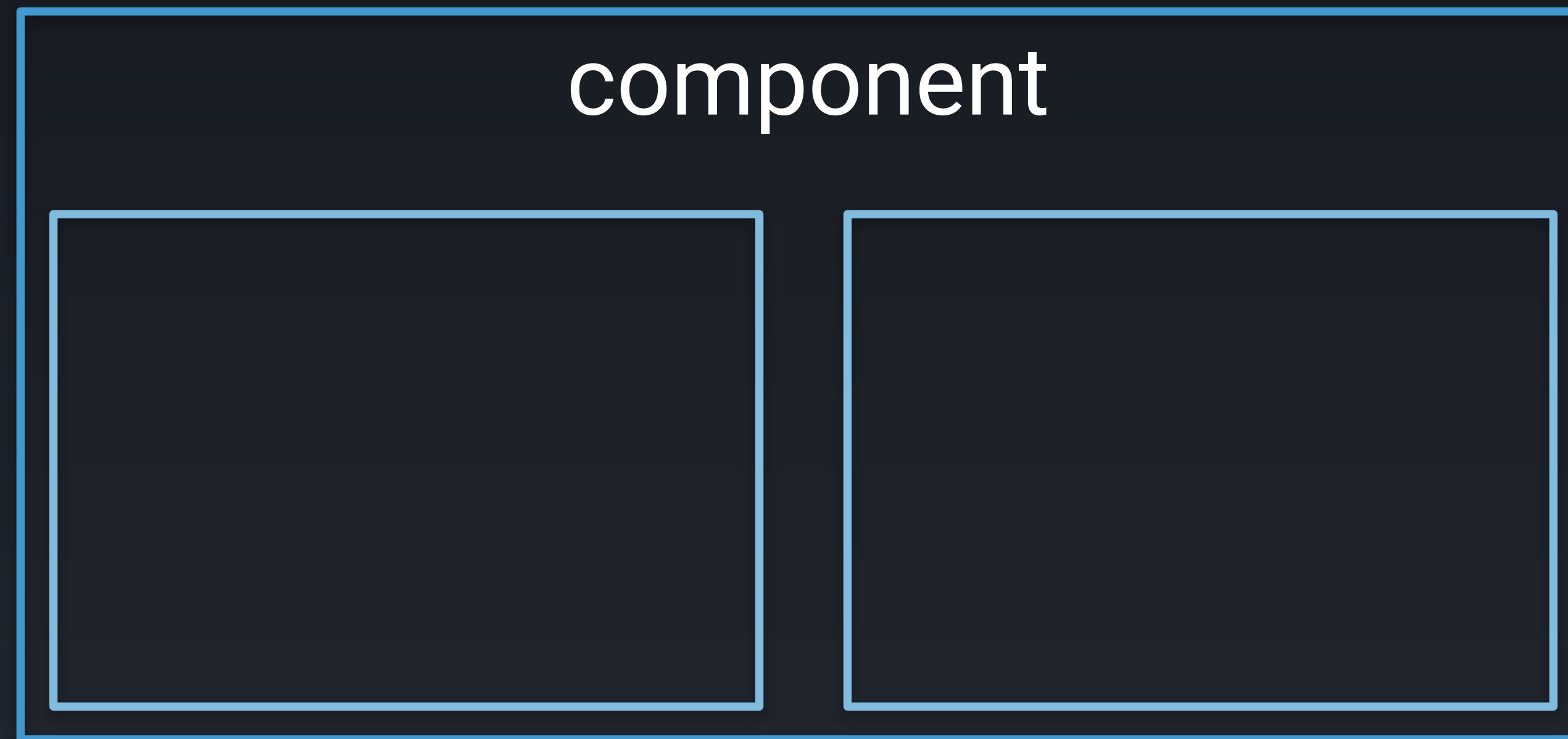
component

module

routes

# component

template

class

services

# What does **implements OnInit** mean?

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

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

**Components**

# How does **ngOnInit** work?

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

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```
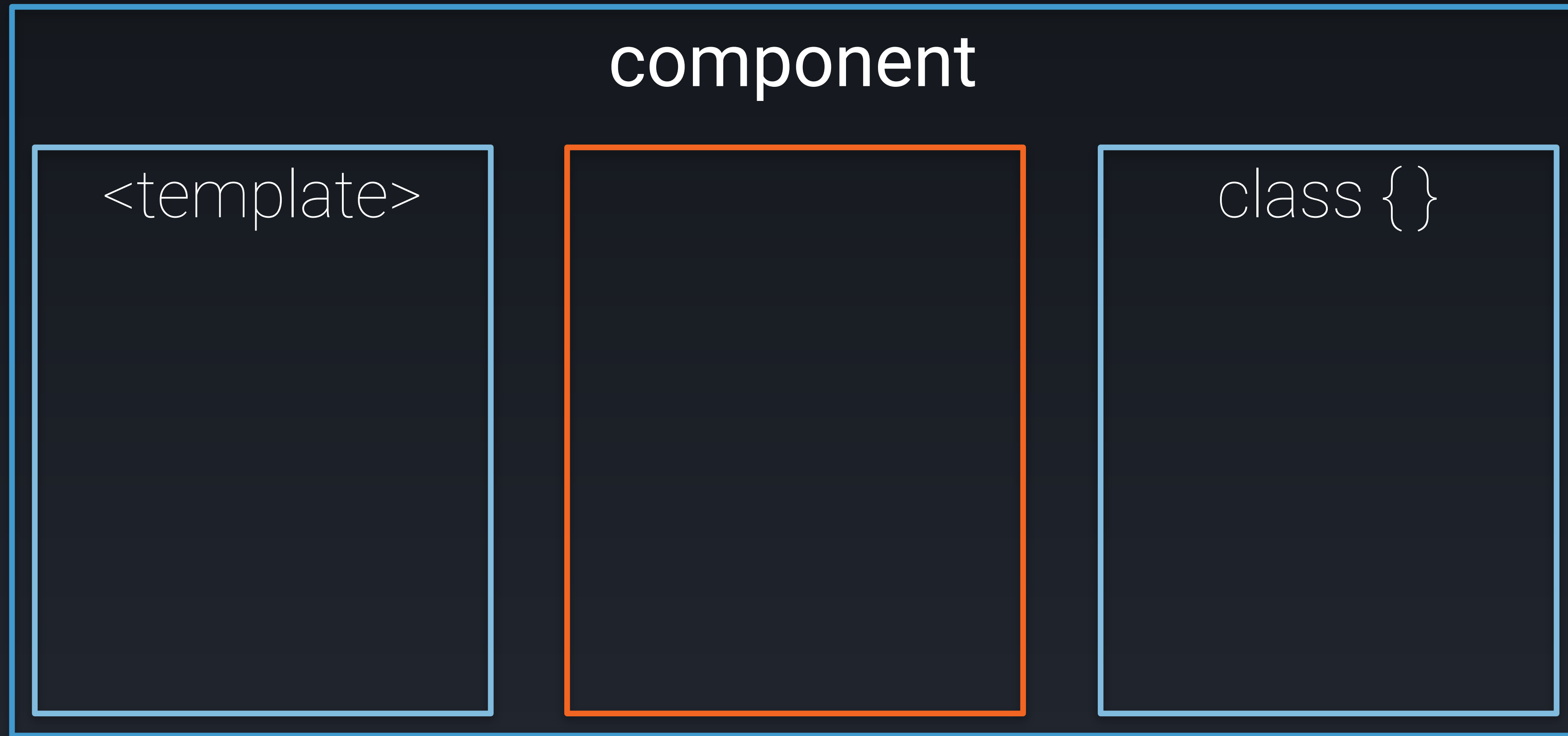
**Components**

# How do we inject a dependency into our component?

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

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

**Components**

# What does the keyword **private** do in the constructor?

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

  constructor(private itemsService: ItemsService) {}

  ngOnInit() {
    this.itemsService.loadItems()
      .then(items => this.items = items);
  }
}
```

**Components**

# How do we connect our template and component class?

**component**

<template>

class {}

# Metadata

component

<template>

@metadata()
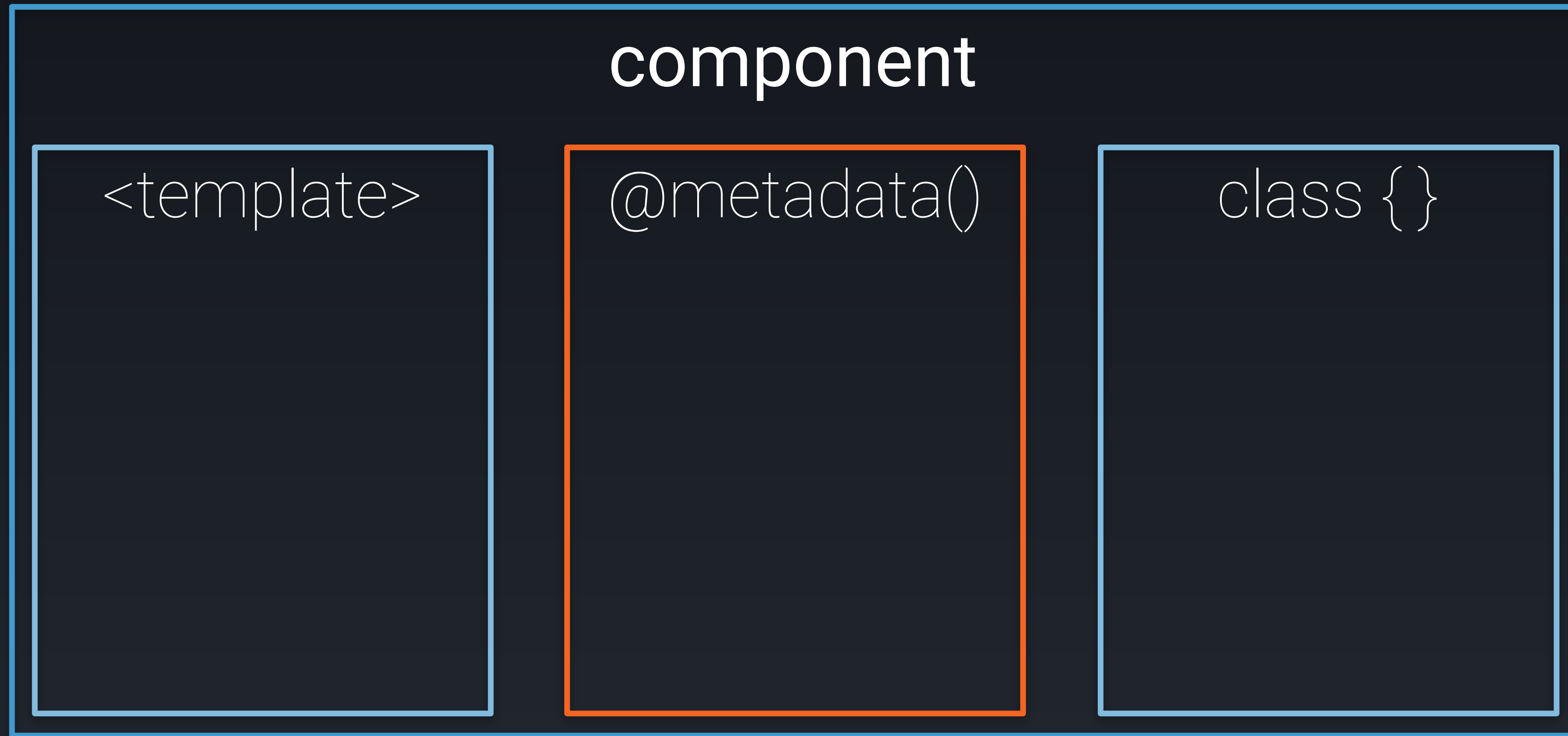
class {}

# What are two things a component must have?

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

**Metadata**

# What is another way to define a component template?

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

**Templates**

# Why might we use one instead of the other?

```
@Component({
  selector: 'app-items-list',
  template: `
<div *ngFor="let item of items" (click)="selected.emit(item)">
  <div>
    <h2>{{item.name}}</h2>
  </div>
  <div>
    {{item.description}}
  </div>
  <div>
    <button (click)="deleted.emit(item); $event.stopPropagation();">
      <i class="material-icons">close</i>
    </button>
  </div>
</div>
`,
  styleUrls: ['./items-list.component.css']
})
```
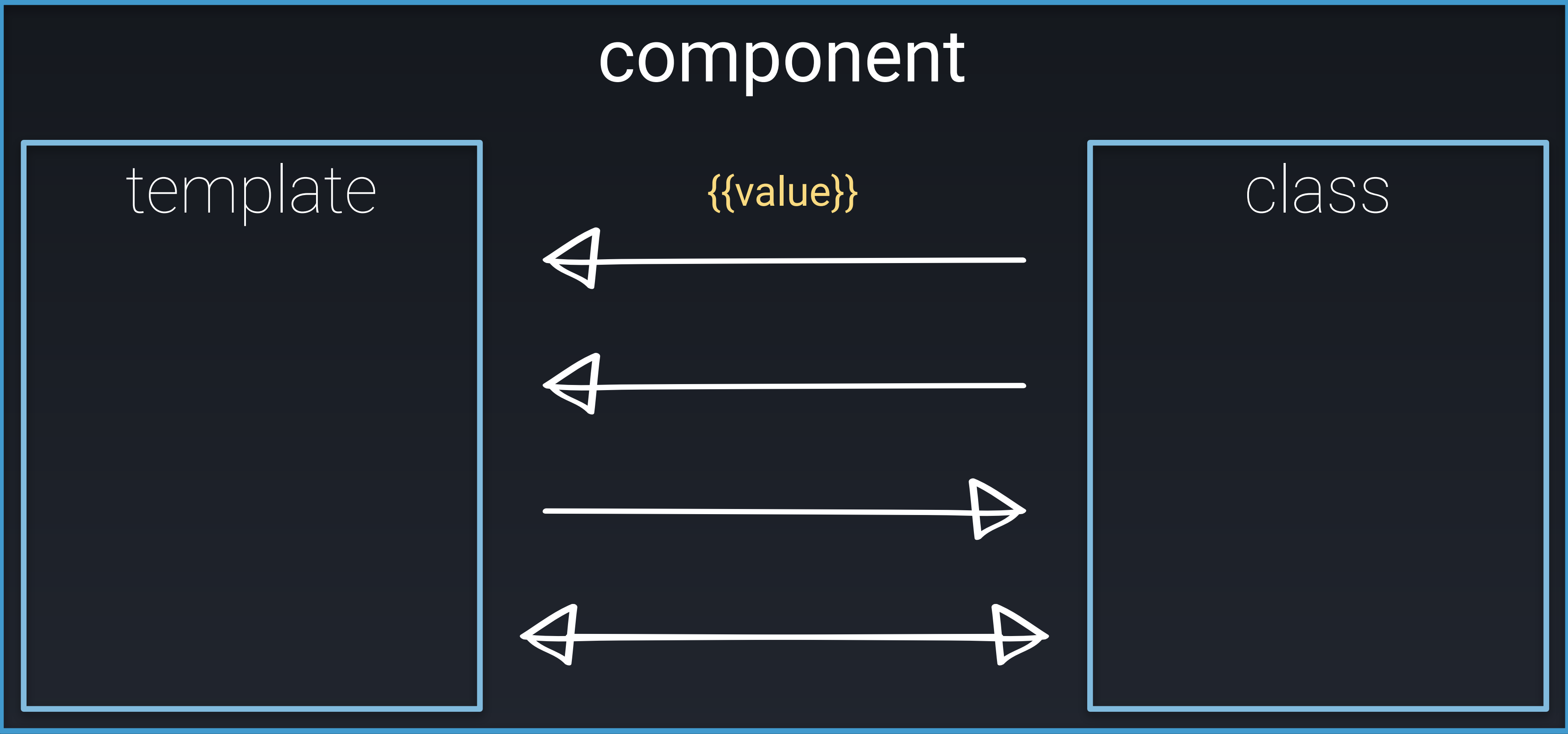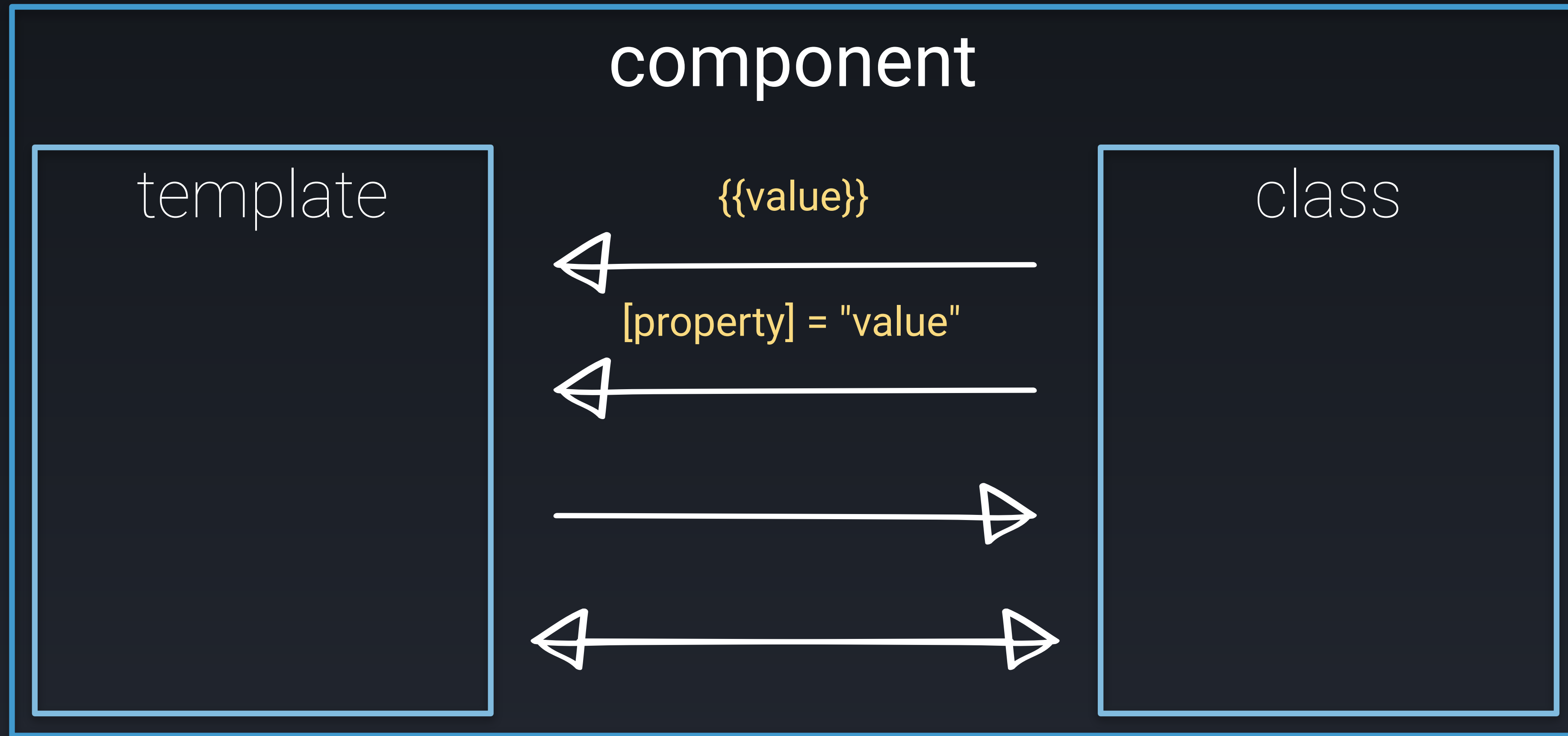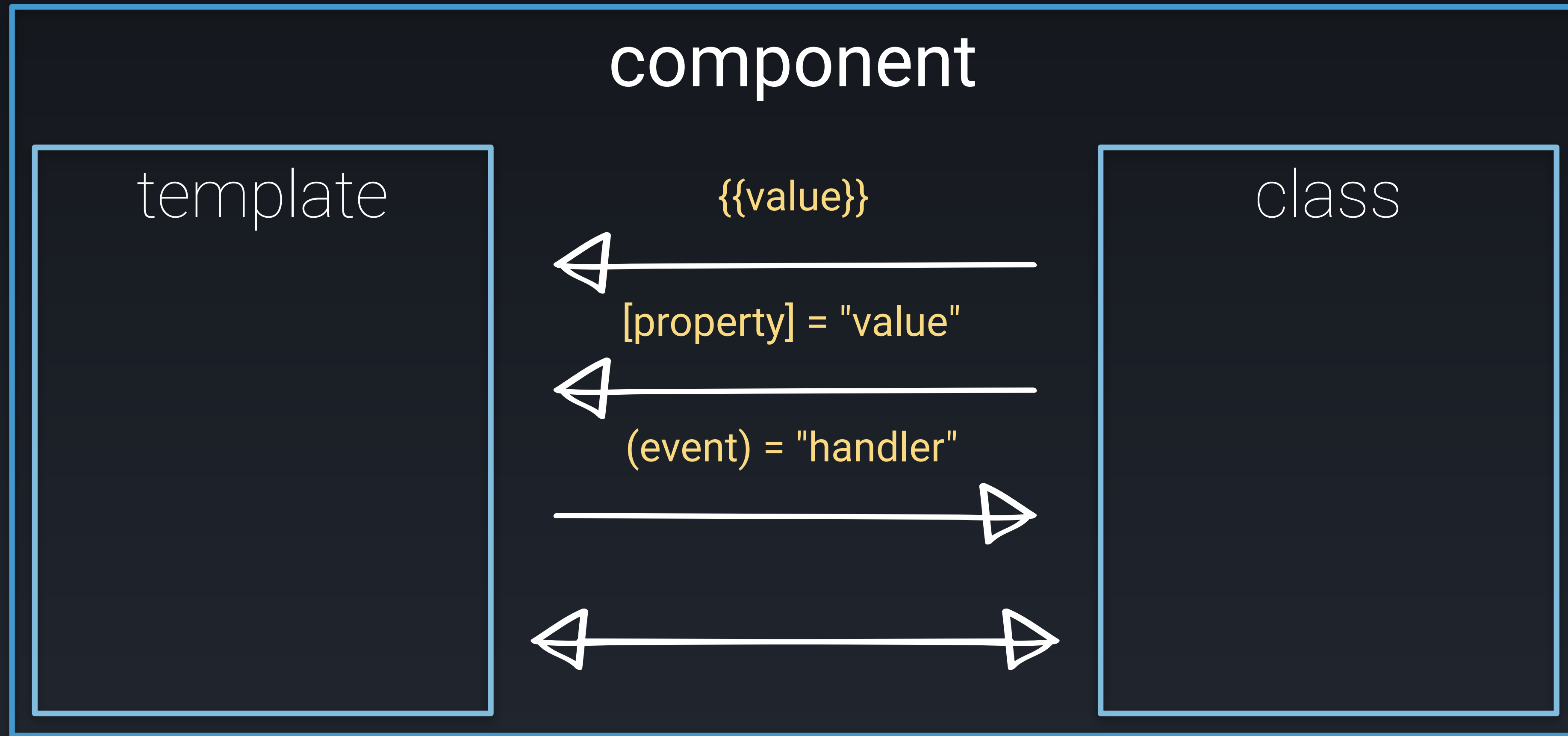
**Templates**

# What is the easiest way to bind to a simple text value?
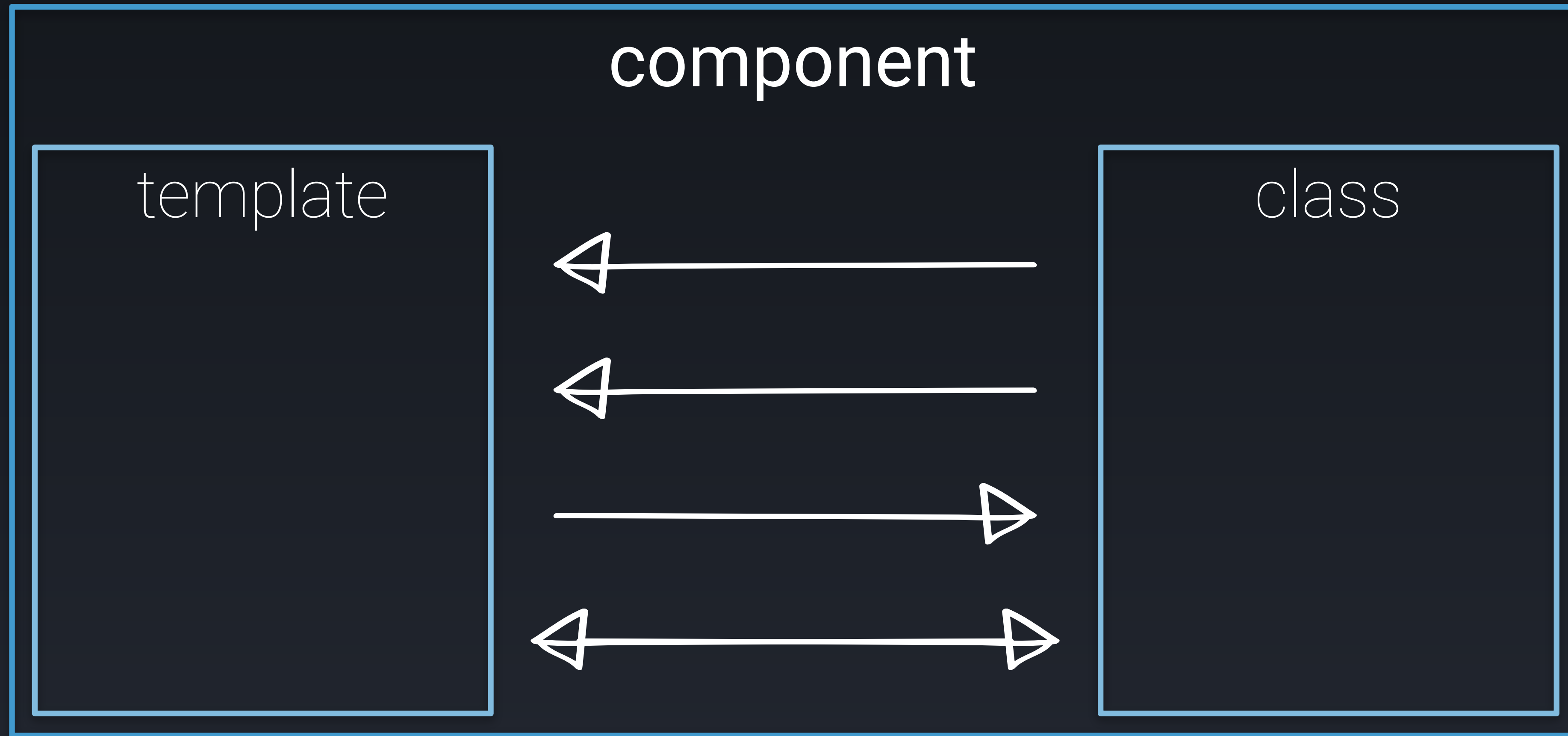
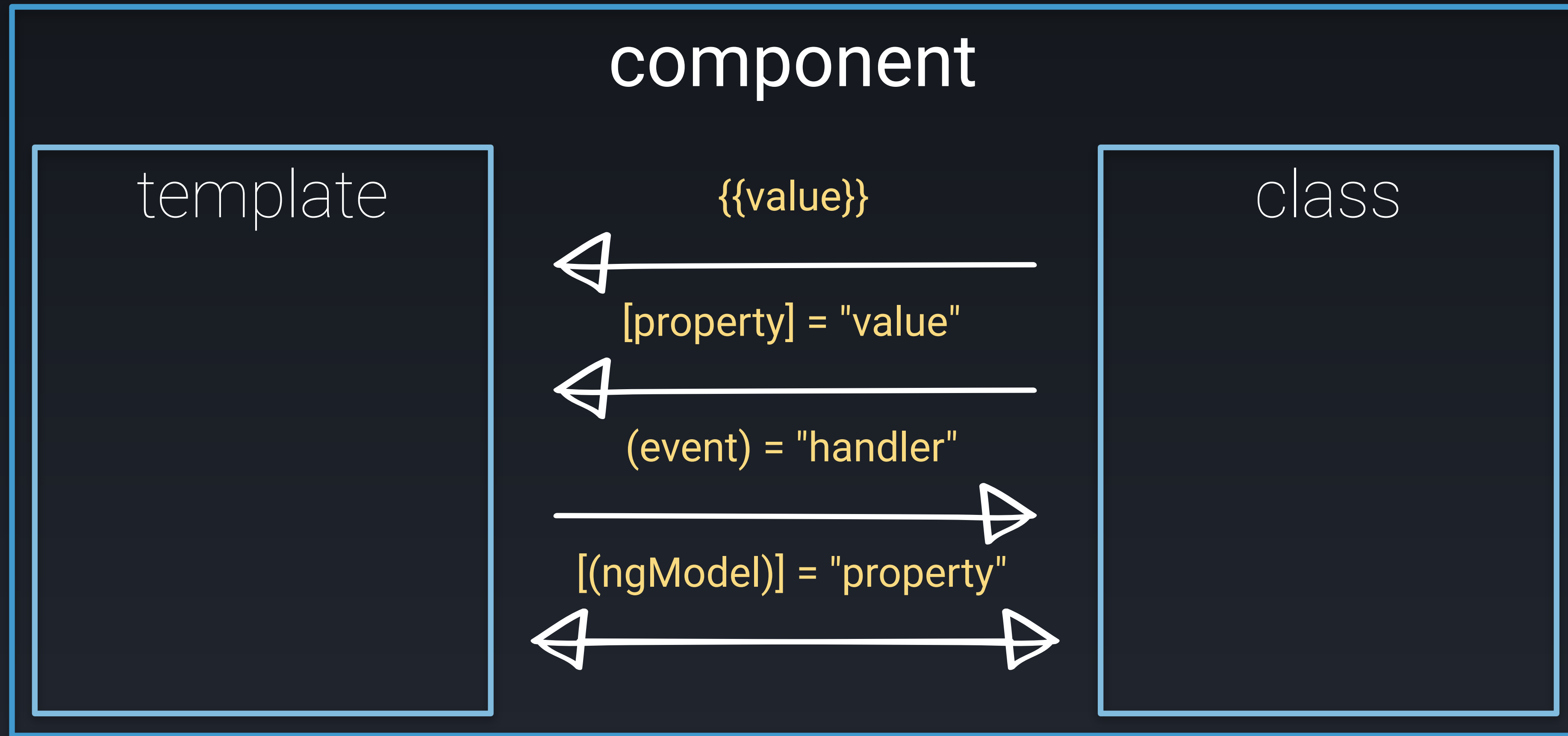# What is the best binding for sending data to the template?

# How do we communicate from the template to the class?

component

template

{{value}}

[property] = "value"

class

# How do we keep a value in sync between both?

component

template

class

{{value}}

[property] = "value"

(event) = "handler"

# How does two way data binding really work?

# What bindings do we use on each side of the diagram?

<template>

@metadata

class {}

(event binding)

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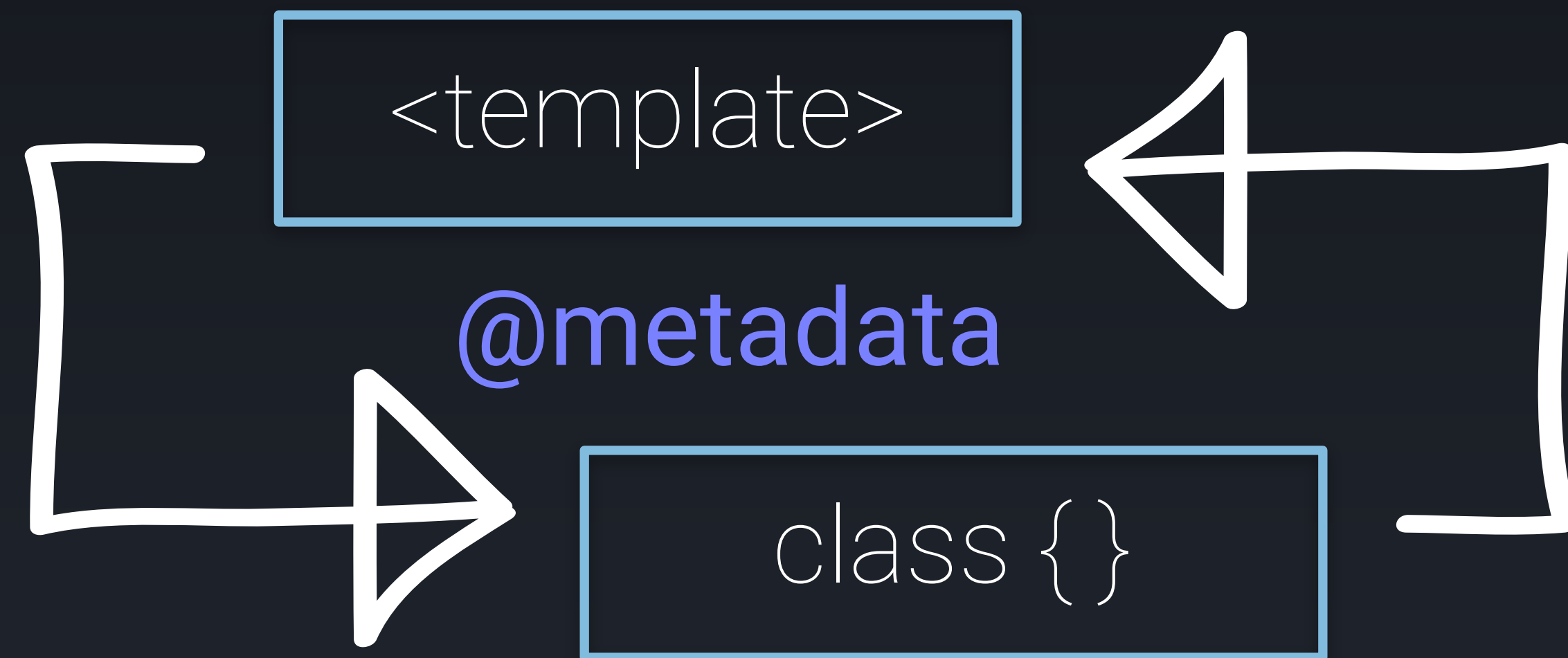
```html
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let experiment of experiments" [experiment]="experiment"></experiment>
<hr/>
<div>
  <h2 class="text-error">Experiments: {{message}}</h2>
    <form class="form-inline">
      <input type="text" [(ngModel)]="message" placeholder="Message">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

# Data Binding

```
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let 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>
```
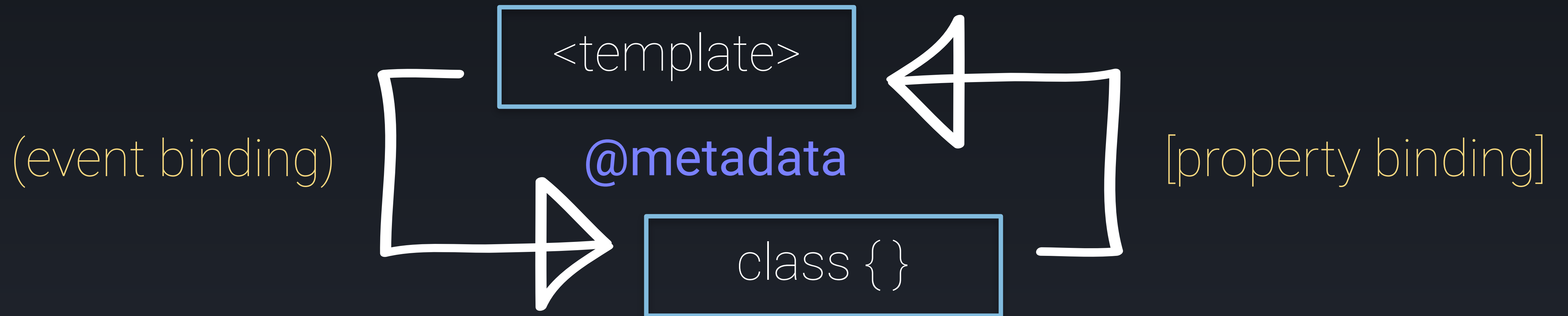
# Data Binding

```html
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let 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

```html
<h1>{{title}}</h1>
<p>{{body}}</p>
<hr/>
<experiment *ngFor="let 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

# How do components and directives differ?

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

@Directive({selector: 'blink'})
export class Blinker {
  constructor(element: ElementRef) {
    // All the magic happens!
  }
}
```

**Directives**

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

@Directive({selector: '[blinker]'})
export class Blinker {
  constructor(element: ElementRef) {
    // All the magic happens!
  }
}
```

# Directives

# How is a service like a component? How are they different?

```typescript
import { Injectable } from '@angular/core';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

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

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

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

**Services**

# Challenges

- In your working example, create a new **review** feature including file structure, component class and template
- Make it available to the rest of the application
- Display the **review component** in the application via its HTML **selector**
- Display the **review component** in the application via a **route**
- Bind to a simple property in the template
- Create an array and use a built in directive to display the array in the template

# Template
Driven Forms

# Template Driven Forms

- FormsModule
- Form Controls
- Validation Styles

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

**FormsModule**

# ngModel

- Enables two-way data binding within a form
- Creates a **FormControl** instance from a domain model and binds it to a form element
- We can create a local variable to reference the **ngModel** instance of the element

```
<input [(ngModel)]="selectedItem.name"
       name="name" #nameRef="ngModel"
       placeholder="Enter a name"
       type="text">
```

ngModel

# Form Controls

- **ngControl** binds a DOM element to a **FormControl**
- **FormControl** is responsible for tracking value and validation status of a single form element
- You can group **FormControl**s together with **FormGroup**
- **ngForm** binds an HTML form to a top-level **FormGroup**
- We can create a local variable to reference the **ngForm** instance of a form
- **ngModelGroup** creates and binds a **FormGroup** instance to a DOM element

```html
<form novalidate #formRef="ngForm">
  <div>
    <label>Item Name</label>
    <input [(ngModel)]="selectedItem.name"
      name="name" required
      placeholder="Enter a name" type="text">
  </div>
  <div>
    <label>Item Description</label>
    <input [(ngModel)]="selectedItem.description"
      name="description"
      placeholder="Enter a description" type="text">
  </div>
</form>
```

ngForm

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

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

ngForm

```
<form novalidate #formRef="ngForm">
  <fieldset ngModelGroup="user">
    <label>First Name</label>
    <input [(ngModel)]="user.firstName"
      name="firstName" required
      placeholder="Enter your first name" type="text">
    <label>Last Name</label>
    <input [(ngModel)]="user.lastName"
      name="lastName" required
      placeholder="Enter your last name" type="text">
  </fieldset>
</form>
```

ngModelGroup

```html
<div ngModelGroup="user">
  <label>First Name</label>
  <input [(ngModel)]="firstName"
    name="firstName" required
    placeholder="Enter your first name" type="text">
  <label>Last Name</label>
  <input [(ngModel)]="lastName"
    name="lastName" required
    placeholder="Enter your last name" type="text">
</div>
<pre>{{formRef.value | json}}</pre>

<!--
{

  "user": {
    "firstName": "Test",
    "lastName": "Test"
  }
}
-->
```

ngModelGroup

# Validation Styles

- Angular will automatically attach styles to a form element depending on its state
- For instance, if it is in a valid state then **ng-valid** is attached
- If the element is in an invalid state, then **ng-invalid** is attached
- There are additional styles such as **ng-pristine** and **ng-untouched**

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

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

**Validation Styles**

# Demonstration

# Challenges

- Create a form to display the currently selected **widget**
- Create a button to **save** the edited **widget**
- Create a button to **cancel** editing the **widget**
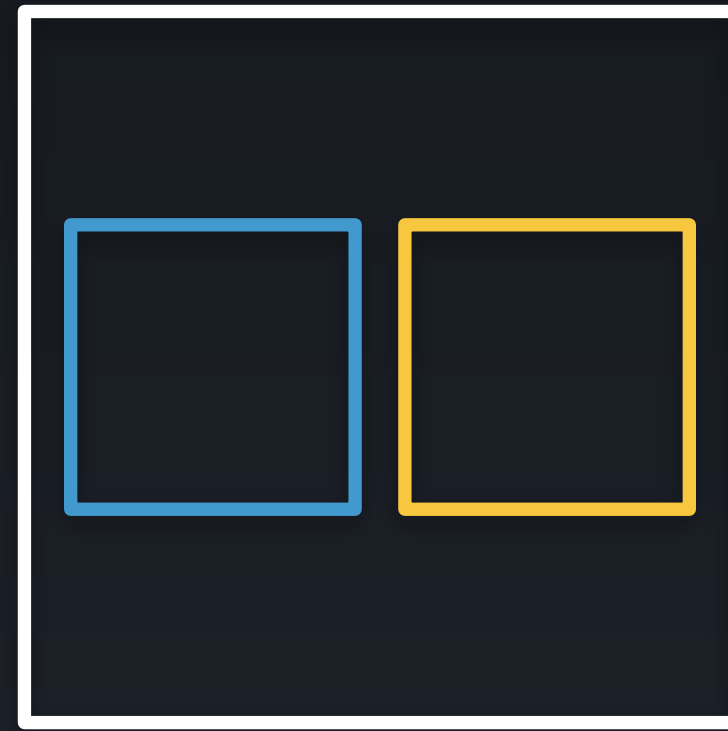- Using **ngForm**, add in some validation for editing the **widget** component

# Component Driven
# Architecture
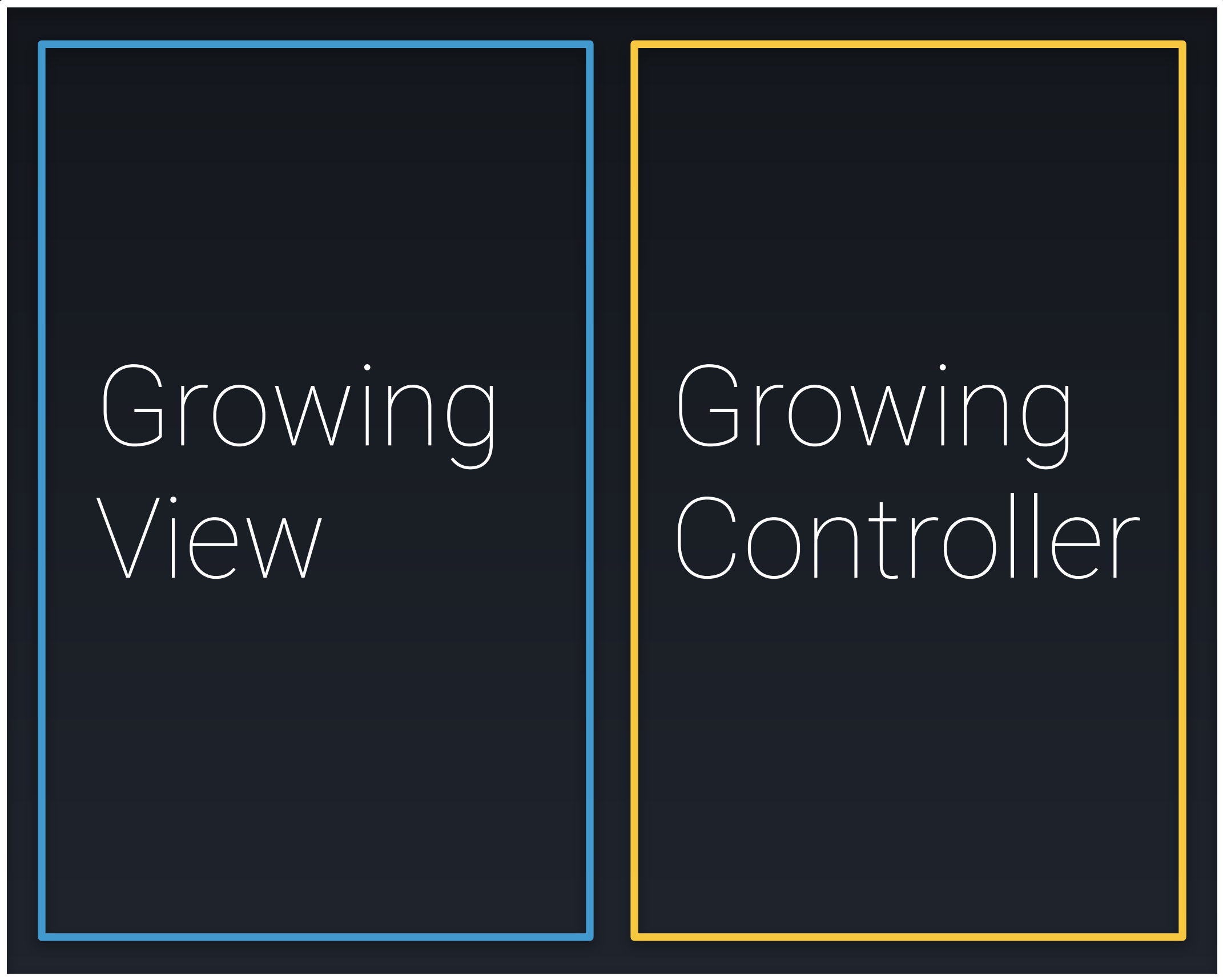
# Component Driven Architecture

- Component Driven Architecture
- Clear contract with @Input and @Output
- @Input
- @Output
- Container and Presentational Components

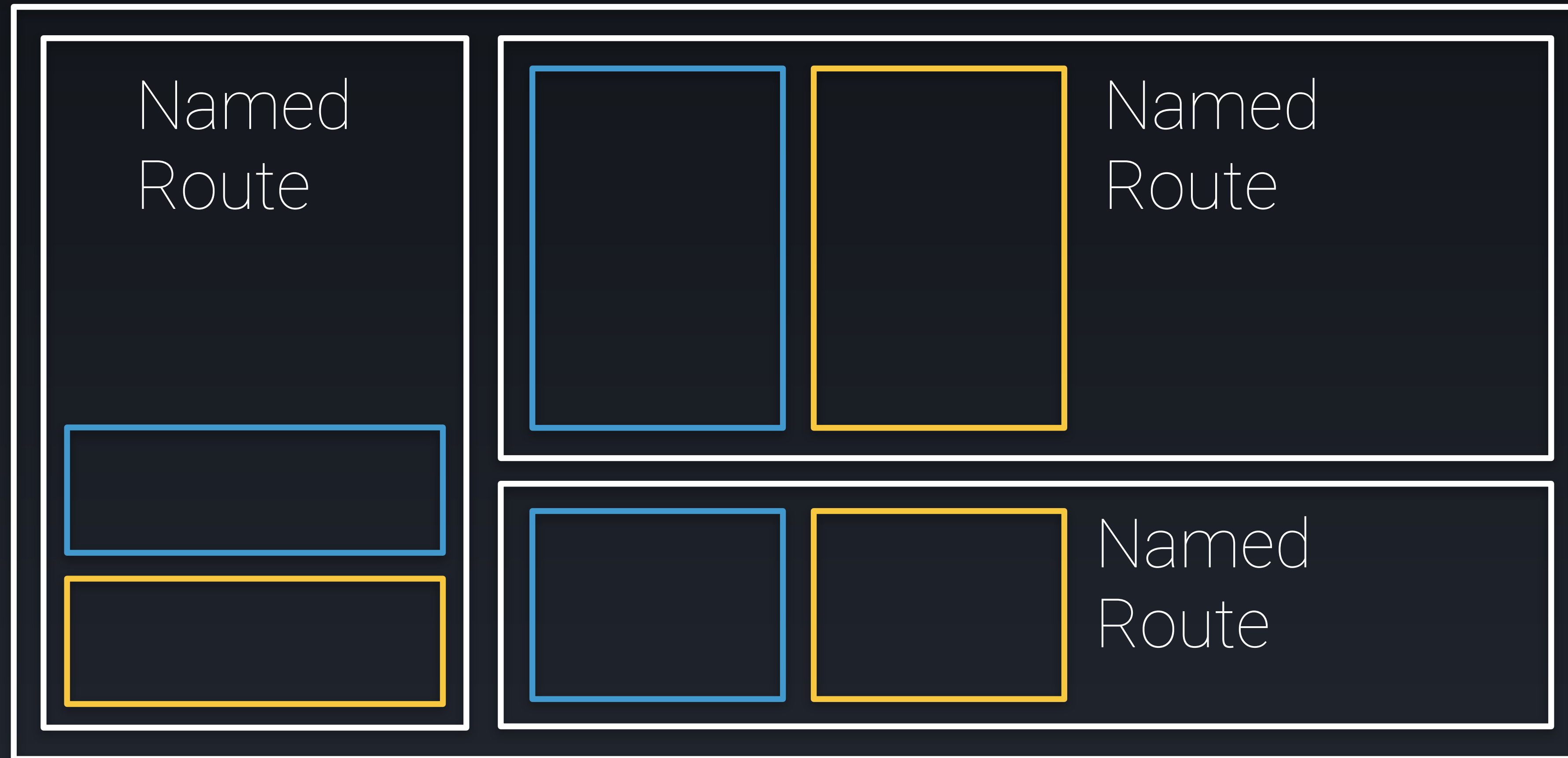# A Brief History of Angular

tiny app == tiny view + tiny controller

# Realistic Application

Growing
View

Growing
Controller

Uh oh!

Large 1.x Application

Large 1.x Application
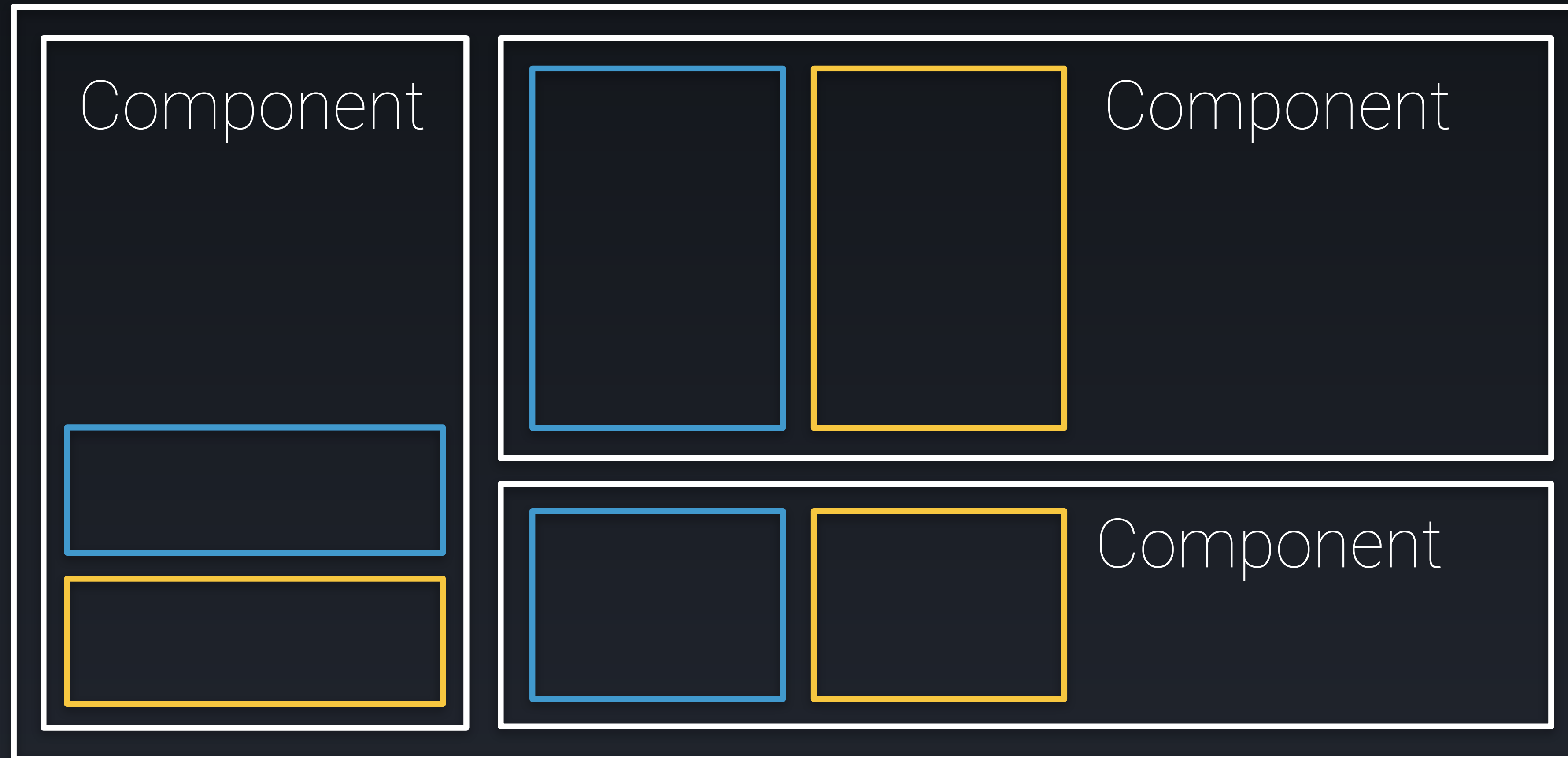
Any Angular Application

# Structure

# Communication

# Component Driven Architecture

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

# Data Binding
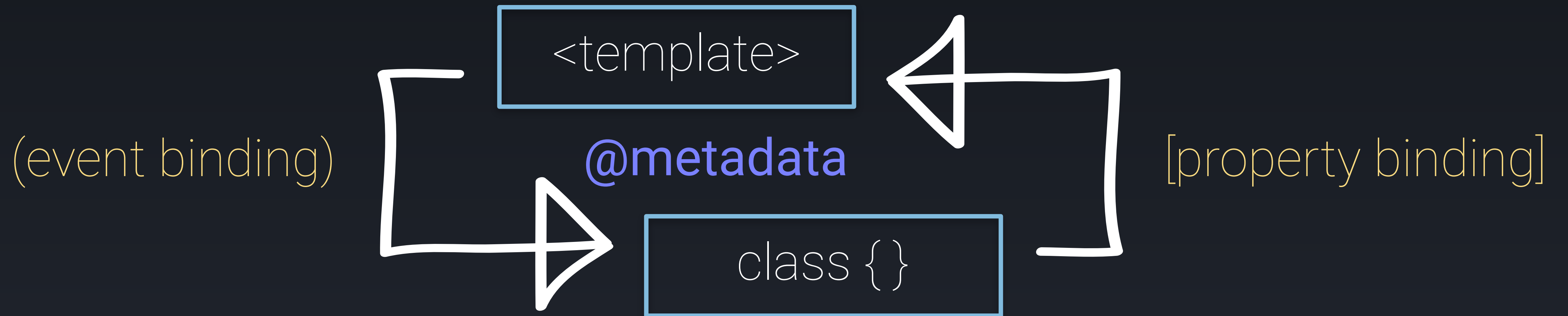


<template>

@metadata

class {}

(event binding)

[property binding]

# Custom Data Binding
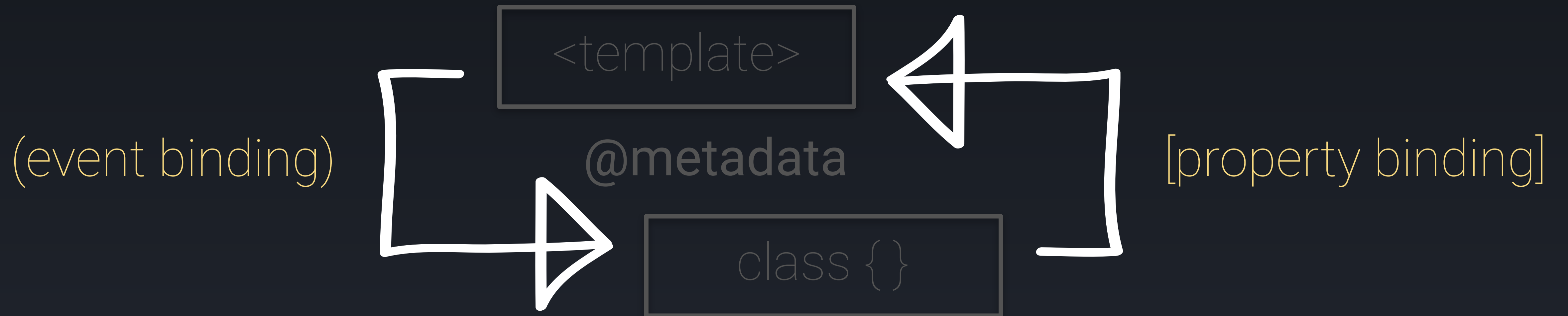
<template>

@metadata

(event binding)                    [property binding]

class {}

# Component Contract

<template>

@Output          @metadata          @Input

class {}

# Parent and Child

Child

@Output      class { }      @Input

@metadata

<template>

Parent

(event)      class { }      [property]

# Parent and Child

Child            @Output    class { }    @Input

<template>

@metadata

Parent    (event)    class { }    [property]

# Parent and **Child**

Child

@Output     class { }     @Input

<template>

@metadata

Parent     (event)     class { }     [property]
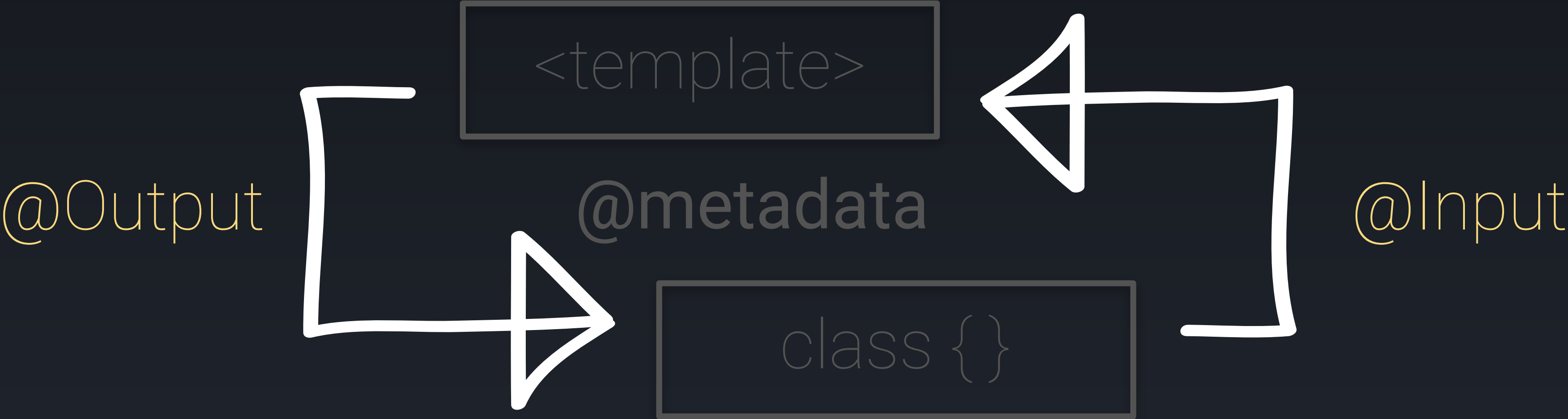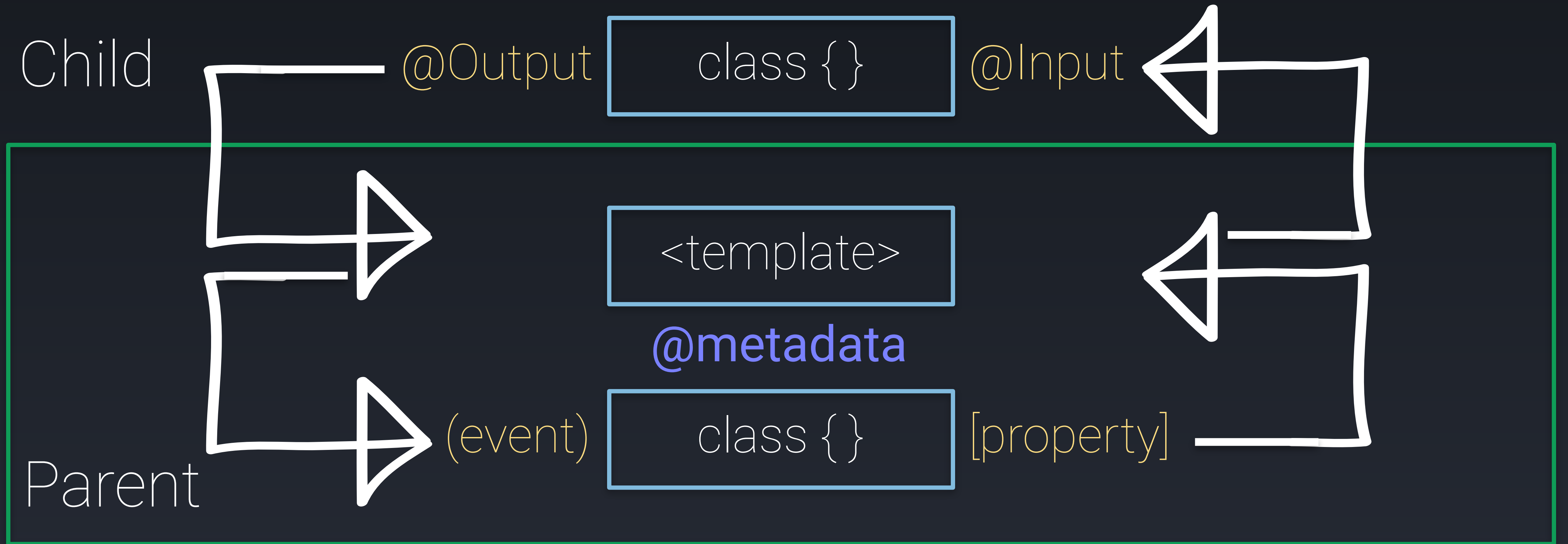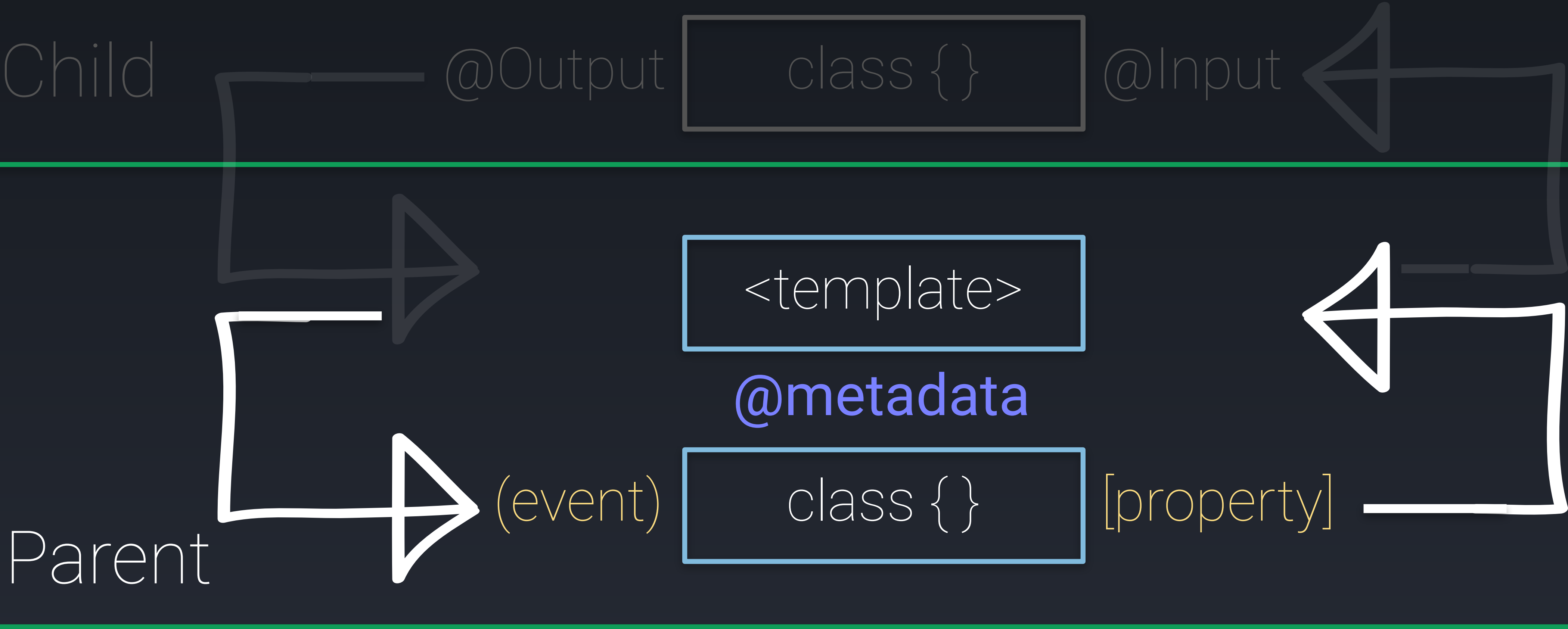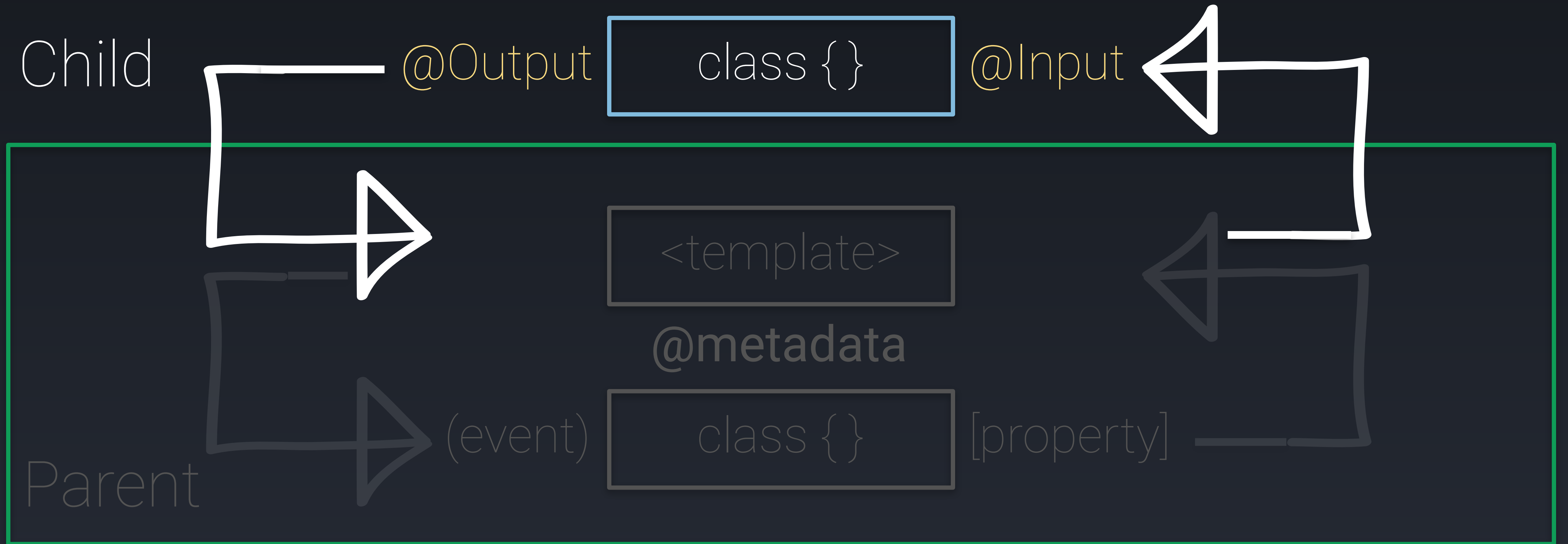
# Component Contracts

- 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

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

# 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;**

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

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

@Input

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

**Parent Component**

# @Output

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

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

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

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

@Output

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

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

**Parent Component**
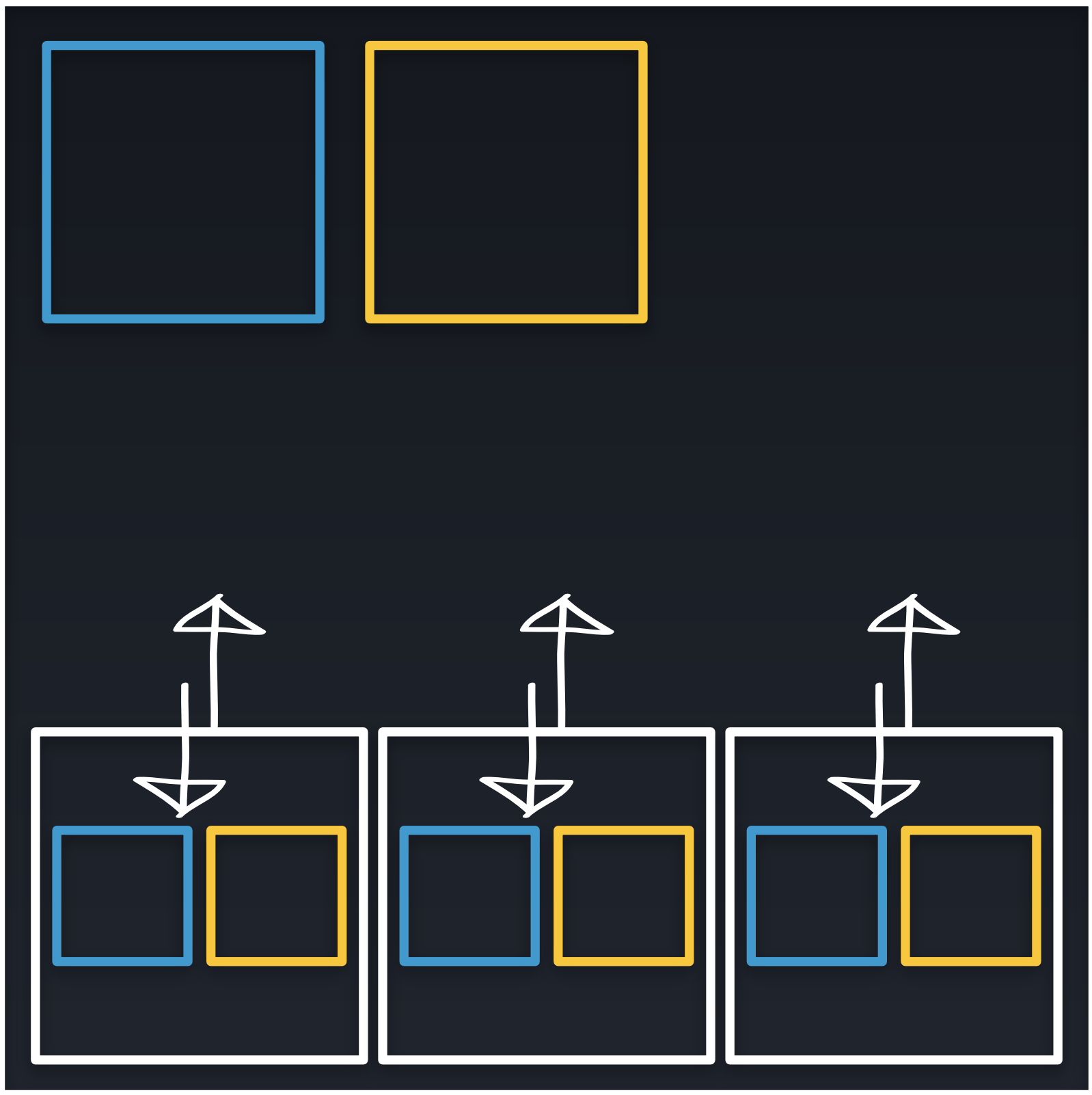
# Container and Presentational Components

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

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

**Presentational Component**

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

  constructor(private itemsService: ItemsService) {}

  ngOnInit() { }

  resetItem() { }

  selectItem(item: Item) { }

  saveItem(item: Item) { }

  replaceItem(item: Item) { }

  pushItem(item: Item) { }

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

**Container Component**

# Demonstration

# Challenges

- Create a presentational **widgets-list** and **widget-details** component using **@Input** and **@Output**
- Pass the **widgets** collection to the **widgets-list** component
- Capture a **selected** output event from the **widgets-list** component
- Display the selected **widget** in the **widget-details** component
- Create a **deleted** output event in the **widgets-list** component
- Create a **saved** output event in the **widget-details** component
- Create a **cancelled** output event in the **widget-details** component

```
ng g c ./widgets/widgets-list -m ../app.module.ts
```

# Server Communication

# Server Communication

- The HTTP Module
- Methods
- Observable.toPromise
- Observable.subscribe
- Headers
- Error Handling

# The HTTP Module

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

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

# HttpModule

# 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();
}


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

# 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

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

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

http.get

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

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

**Observable.toPromise**

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

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

**Observable.map**

```
constructor(private itemsService: ItemsService) {}

ngOnInit() {
  this.itemsService.loadItems()
    .then(items => this.items = items);
}
```

**Resolving the promise**

# Observable.subscribe

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

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

http.get

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

**Observable.map**

```typescript
constructor(private itemsService: ItemsService) {}

ngOnInit() {
  this.itemsService.loadItems()
    .subscribe(items => this.items = items);
}
```

Observable.subscribe

# Headers

- Http module methods have an optional second parameter which is a **RequestOptions** object
- The **RequestOptions** object has a **headers** property which is a **Headers** object
- We can use the **Headers** object to set additional parameters like **Content-Type**

```typescript
import { Http, Headers } from '@angular/http';
import { Injectable } from '@angular/core';
import { Item } from './item.model';
import 'rxjs/add/operator/map';

const BASE_URL = 'http://localhost:3000/items/';
const HEADER = { headers: new Headers({ 'Content-Type': 'application/json' }) };

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

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

# Headers

```typescript
import { Http, Headers } from '@angular/http';
import { Injectable } from '@angular/core';
import { Item } from './item.model';
import 'rxjs/add/operator/map';


const BASE_URL = 'http://localhost:3000/items/';
const HEADER = { headers: new Headers({ 'Content-Type': 'application/json' }) };


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


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

# Headers

```typescript
import { Http, Headers, RequestOptions } from '@angular/http';
import { Injectable } from '@angular/core';
import { Item } from './item.model';
import 'rxjs/add/operator/map';


const BASE_URL = 'http://localhost:3000/items/';
const headers = new Headers({ 'Content-Type': 'application/json' });
const options = new RequestOptions({ headers: headers });


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

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

# RequestOptions

# Error Handling

- Use **Observable.catch** to process the error at the service level
- Use **Observable.throw** to force an error further down the stream
- Use the **error** handler in the **subscribe** method to respond to the error at the component level

```
loadItem(id) {
  return this.http.get(`${BASE_URL}${id}`)
    .map(res => res.json())
    .catch(error =>
      Observable.throw(error.json().error || 'Server error'));
}
```

Observable.catch

```
ngOnInit() {
  this.itemsService.loadItems()
    .map(items => this.items = items)
    .subscribe(
      this.diffFeaturedItems.bind(this),
      this.handleError.bind(this)
    );
}
```

# Handling the Error

# Demonstration

# Challenges

- Replace the local **widgets** collection with a call to the **widgets** endpoint
- Update the **widgets** component to handle the async call
- Flesh out the rest of the **CRUD** functionality using **ItemsService** as reference

# Observable
# Fundamentals

# Pre Challenge

- Create an **ItemsSearch** component
  
  `ng g c ./items/items-search -m ../app.module.ts`
- Add the **ItemsSearch** component to the **HomeComponent** template
- Go to **http://bit.ly/items-search-snippets** and copy the snippets to the component

# Observable Fundamentals

- Basic Observable Sequence
- Observable.map
- Observable.filter
- Etc.

http://reactivex.io/rxjs/

# Iterator Pattern

**Aggregate**

+ *iterator()*

<< create >>

**Iterator**

+ *next()*
+ *hasNext()*

**ConcreteAggregate**

+ iterator()

<< create >>

**ConcreteIterator**

+ next()
+ hasNext()

## State

# Observer Pattern

**Observer**

+notify()

**Subject**

+observerCollection

+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

**ConcreteObserverA**

+notify()

**ConcreteObserverB**

+notify()

notifyObservers()
 for observer in observerCollection
 call observer.notify()
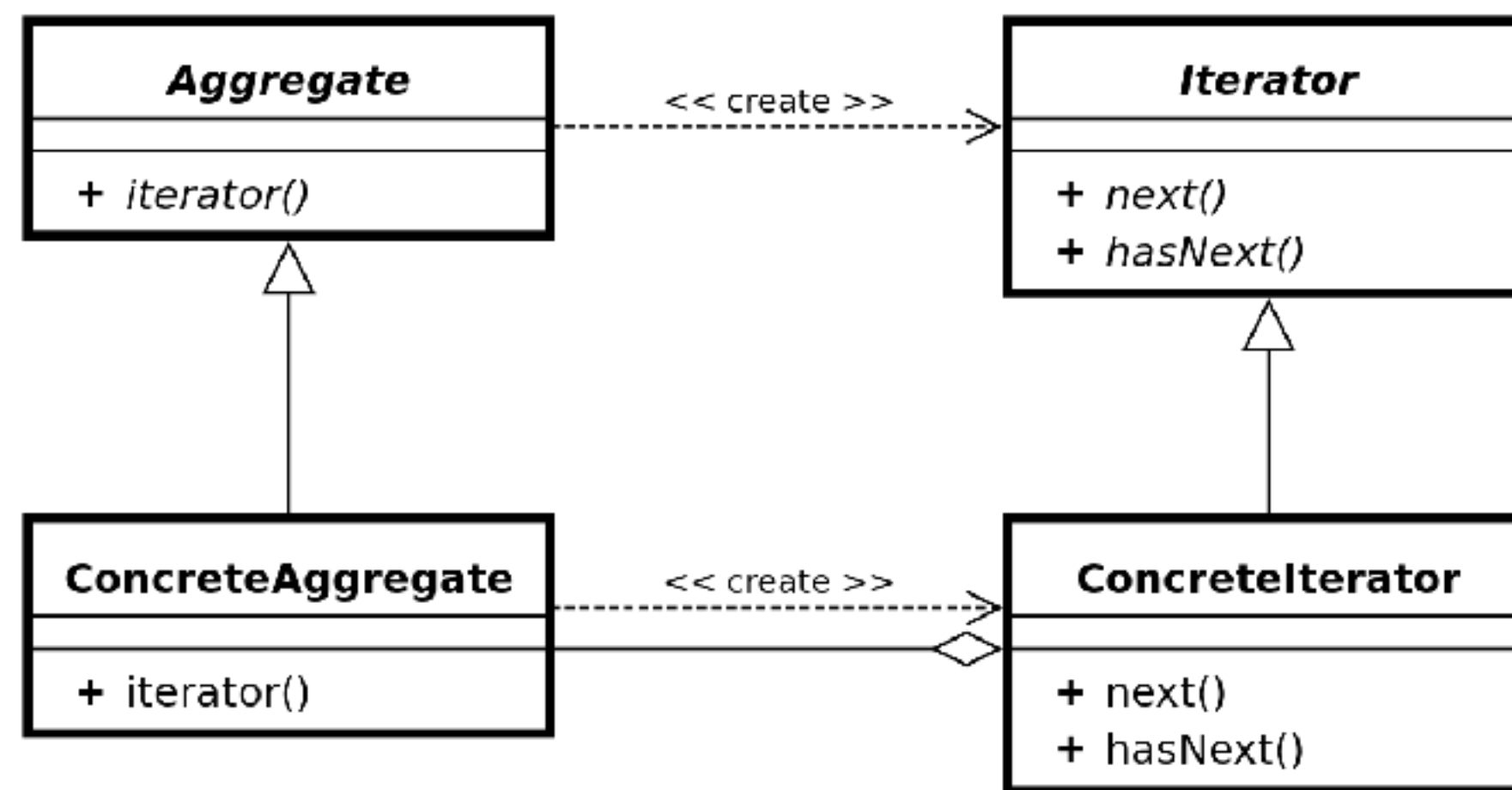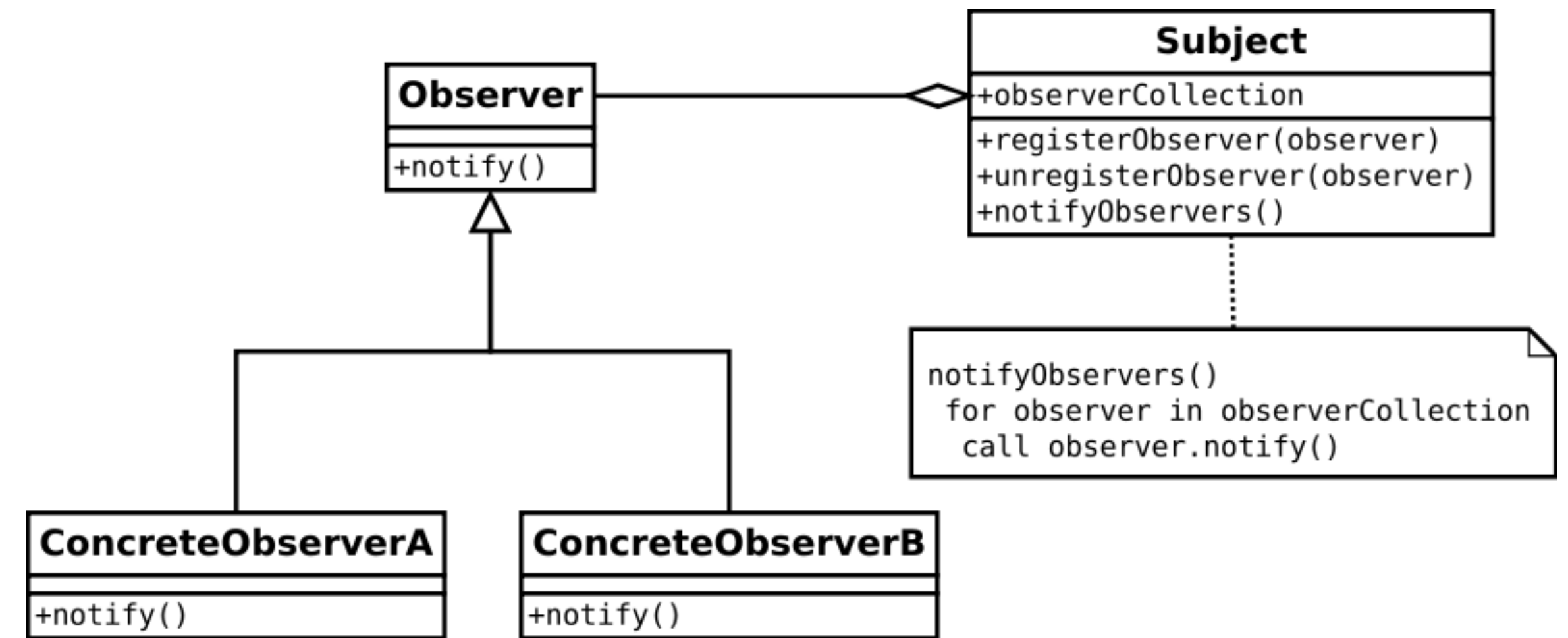
## Communication

# Communicate
**state** over **time**
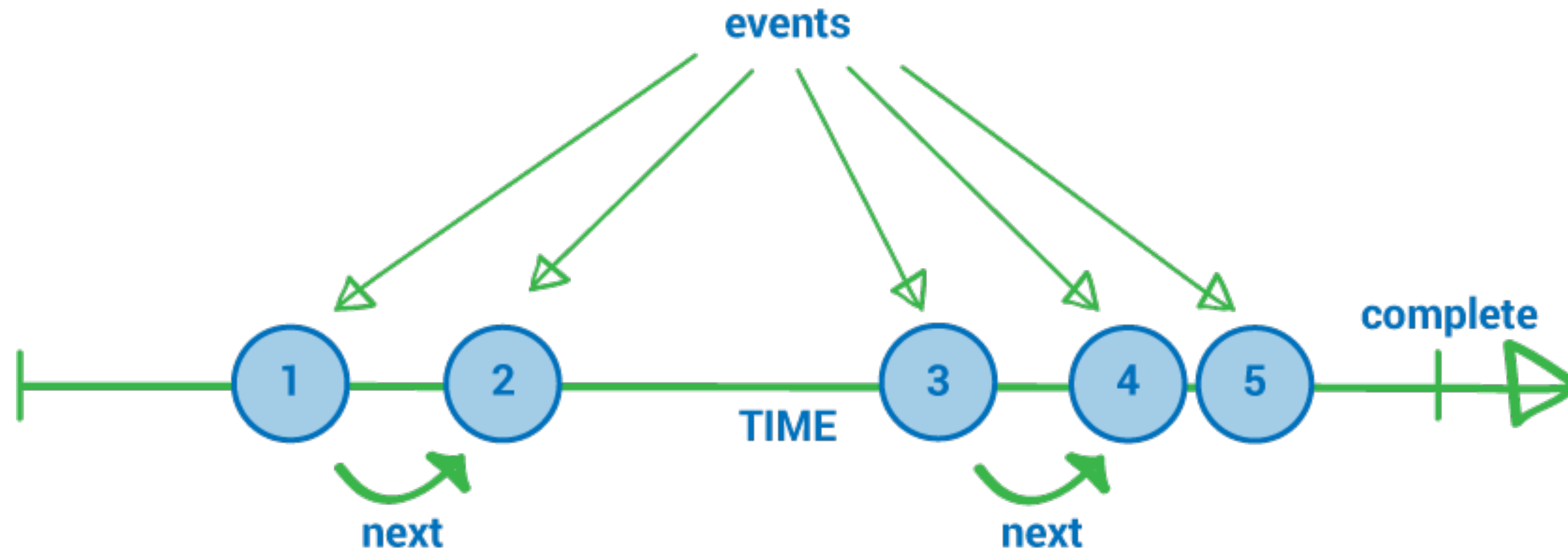
**Observable stream**

|  | SINGLE | MULTIPLE |
|---|---|---|
| SYNCHRONOUS | Function | Enumerable |
| ASYNCHRONOUS | Promise | **Observable** |

# Values over time

|  | SINGLE | MULTIPLE |
|---|---|---|
| **PULL** | Function | Enumerable |
| **PUSH** | Promise | **Observable** |

# Value consumption
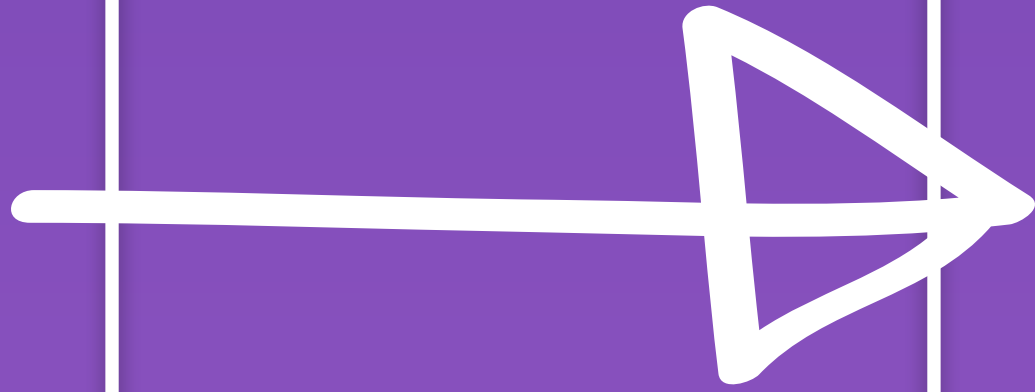
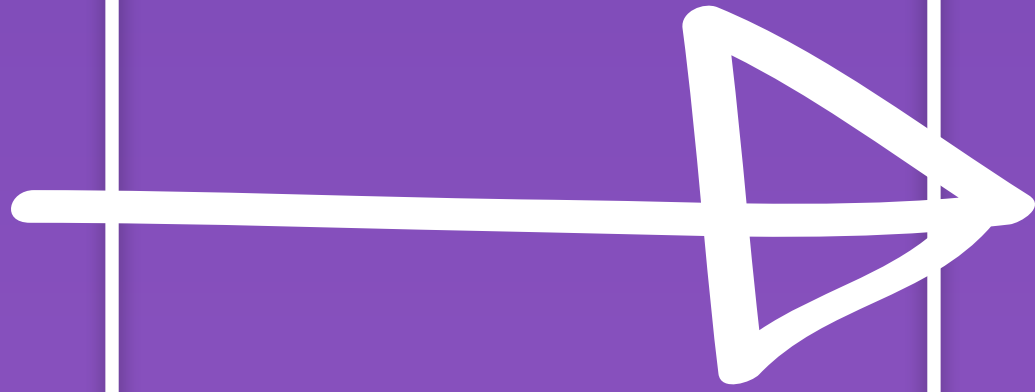input → output

output → input

# The
# Basic
# **Sequence**

initial output

magic

final input

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(result => this.message = 'Beast Mode Activated!');
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(result => this.message = 'Beast Mode Activated!');
}


getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

**Initial output**

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(event => this.message = 'Beast Mode Activated!');
}


getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

**Final input**

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .map(event => 'Beast Mode Activated!')
    .subscribe(result => this.message = result);
}


getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

**Everything in between**

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .filter(event => event.shiftKey)
    .map(event => 'Beast Mode Activated!')
    .subscribe(result => this.message = result);
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

**Everything in between**

# Demonstration

# Challenges

- Capture the input from the **itemsSearch** field
- Use **map** to transform the input into a query value we can use
- Use **map** to transform the query into something else like uppercase
- User **filter** to only allow input when the **shift** key is pressed
- Implement **bufferTime**, **distinctUntilChanged** and **switchMap**

# Reactive Forms

# Pre Challenge

- Create an **Newsletter** component

  `ng g c newsletter -m app.module.ts`
- Add the **Newsletter** component to the **HomeComponent** template
- Go to **http://bit.ly/newsletter-snippets** and copy the snippets to the component

# Reactive Driven Forms

- **FormControl** tracks the value and validation status of an individual form control
- **FormGroup** tracks the value and validity state of a group of FormControl instances.
- **formGroup** directive binds an existing FormGroup to a DOM element.
- **formControlName** directive syncs a FormControl in an existing FormGroup to a form control element by name.
- **FormBuilder** is essentially syntactic sugar that shortens the new FormGroup(), new FormControl(), and new FormArray() boilerplate that can build up in larger forms.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
```

**ReactiveFormsModule**

```
form = new FormGroup({
  first: new FormControl('Nancy', Validators.minLength(2)),
  last: new FormControl('Drew'),
});
```

**FormGroup**

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  <input formControlName="first" placeholder="First name">
  <input formControlName="last" placeholder="Last name">

  <button type="submit">Submit</button>
</form>
```

**FormGroup Directive**

```typescript
export class SimpleFormGroup {
  form = new FormGroup({
    first: new FormControl('Nancy', Validators.minLength(2)),
    last: new FormControl('Drew'),
  });

  get first(): any { return this.form.get('first'); }

  onSubmit(): void {
    console.log(this.form.value);  // {first: 'Nancy', last: 'Drew'}
  }

  setValue() { this.form.setValue({first: 'Carson', last: 'Drew'}); }
}
```

# Simple'ish Version

```typescript
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-newsletter',
  templateUrl: './newsletter.component.html',
  styleUrls: ['./newsletter.component.css']
})
export class NewsletterComponent implements OnInit {
  subscriber: FormGroup;

  constructor(private fb: FormBuilder) { }
}
```

**FormBuilder**

```typescript
subscriber: FormGroup;

constructor(private fb: FormBuilder) { }

ngOnInit() {
  this.subscriber = this.fb.group({
    name: ['', Validators.required],
    email: ['', Validators.required]
  });
}
```

**FormBuilder**

```html
<form novalidate [formGroup]="subscriber" (submit)="subscribe(subscriber)">
  <md-card-content>
    <md-input-container class="full-width">
      <input type="text"  mdInput placeholder="Name" formControlName="name">
    </md-input-container>
    <md-input-container class="full-width">
      <input type="text" mdInput placeholder="Email" formControlName="email">
    </md-input-container>
  </md-card-content>
  <md-card-actions>
    <button type="submit" md-button>Subscribe!</button>
    <button type="button" md-button (click)="reset()">Cancel</button>
  </md-card-actions>
</form>
```

```html
<form novalidate [formGroup]="subscriber" (submit)="subscribe(subscriber)">
  <md-card-content>
    <md-input-container class="full-width">
      <input type="text"  mdInput placeholder="Name" formControlName="name">
    </md-input-container>
    <md-input-container class="full-width">
      <input type="text" mdInput placeholder="Email" formControlName="email">
    </md-input-container>
  </md-card-content>
  <md-card-actions>
    <button type="submit" md-button>Subscribe!</button>
    <button type="button" md-button (click)="reset()">Cancel</button>
  </md-card-actions>
</form>
```

# formGroup Directive

```html
<form novalidate [formGroup]="subscriber" (submit)="subscribe(subscriber)">
  <md-card-content>
    <md-input-container class="full-width">
      <input type="text"  mdInput placeholder="Name" formControlName="name">
    </md-input-container>
    <md-input-container class="full-width">
      <input type="text" mdInput placeholder="Email" formControlName="email">
    </md-input-container>
  </md-card-content>
  <md-card-actions>
    <button type="submit" md-button>Subscribe!</button>
    <button type="button" md-button (click)="reset()">Cancel</button>
  </md-card-actions>
</form>
```

**formControlName Directive**

```typescript
subscribe({ value, valid }: { value: Subscriber, valid: boolean }) {
  console.log(value, valid);
  this.reset();
}

reset() {
  this.subscriber.reset({
    name: '',
    email: ''
  });
}
```

# Demonstration

# Challenges

- Create a **subscriber FormGroup** using **FormBuilder**
- Connect **subscriber** to the template form using the **formGroup** and **formControlName** directives
- Submit **subscriber** and log its **value** to the console and if it is **valid**
- Reset the **subscriber**

WAT.

https://www.yearofmoo.com/

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
```

**BrowserAnimationsModule**

```typescript
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
import { Item } from '../../shared';
import {
  animate, group, query, stagger, style, transition, trigger
} from '@angular/animations';

@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.css'],
  animations: [ ]
})
export class ItemsListComponent implements OnInit {
}
```

@angular/animations

```
animations: [
  trigger('listAnimation', [
    transition(':enter, :leave, * => pending', []),
    transition('* => *', [
      // animate both the newly entered & removed items on the page at the same time
      group([
        query(':enter', [
          style({ opacity: 0, height: '0px' }),
          stagger('50ms', [
            animate('500ms cubic-bezier(.35,0,.25,1)', style('*'))
          ])
        ], { optional: true }),
        query(':leave', [
          stagger('50ms', [
            animate('500ms cubic-bezier(.35,0,.25,1)',
              style({ opacity: 0, height: '0px', borderTop: 0, borderBottom: 0 }))
          ])
        ], { optional: true })
      ]),
    ]),
  ]),
]
```

```
animations: [
  trigger('listAnimation', [
    transition(':enter, :leave, * => pending', []),
    transition('* => *', [
      // animate both the newly entered & removed items on the page at the same time
      group([
        query(':enter', [
          style({ opacity: 0, height: '0px' }),
          stagger('50ms', [
            animate('500ms cubic-bezier(.35,0,.25,1)', style('*'))
          ])
        ], { optional: true }),
        query(':leave', [
          stagger('50ms', [
            animate('500ms cubic-bezier(.35,0,.25,1)',
              style({ opacity: 0, height: '0px', borderTop: 0, borderBottom: 0 }))
          ])
        ], { optional: true })
      ]),
    ]),
  ]),
]
```

```html
<md-list
  [@listAnimation]="prepareListState()"
  [@.disabled]="animationsDisabled">
  <md-list-item *ngFor="let item of items; trackBy: trackItem">
    <h3 md-line>{{item.name}}</h3>
    <p md-line>
      {{item.description}}
    </p>
  </md-list-item>
</md-list>
```

```html
<md-list
  [@listAnimation]="prepareListState()"
  [@.disabled]="animationsDisabled">
  <md-list-item *ngFor="let item of items; trackBy: trackItem">
    <h3 md-line>{{item.name}}</h3>
    <p md-line>
      {{item.description}}
    </p>
  </md-list-item>
</md-list>
```

```
export class ItemsListComponent implements OnInit {
  animationsDisabled = true;

  trackItem(index, item) {
    return item.id;
  }

  ngOnInit() {
    setTimeout(() => {
      this.animationsDisabled = false;
    }, 500)
  }

  prepareListState() {
    return this.items ? this.items.length : 'pending';
  }
}
```

@simpulton

Thanks!

WAT.