Learn to Build Awesome Apps with Angular 2

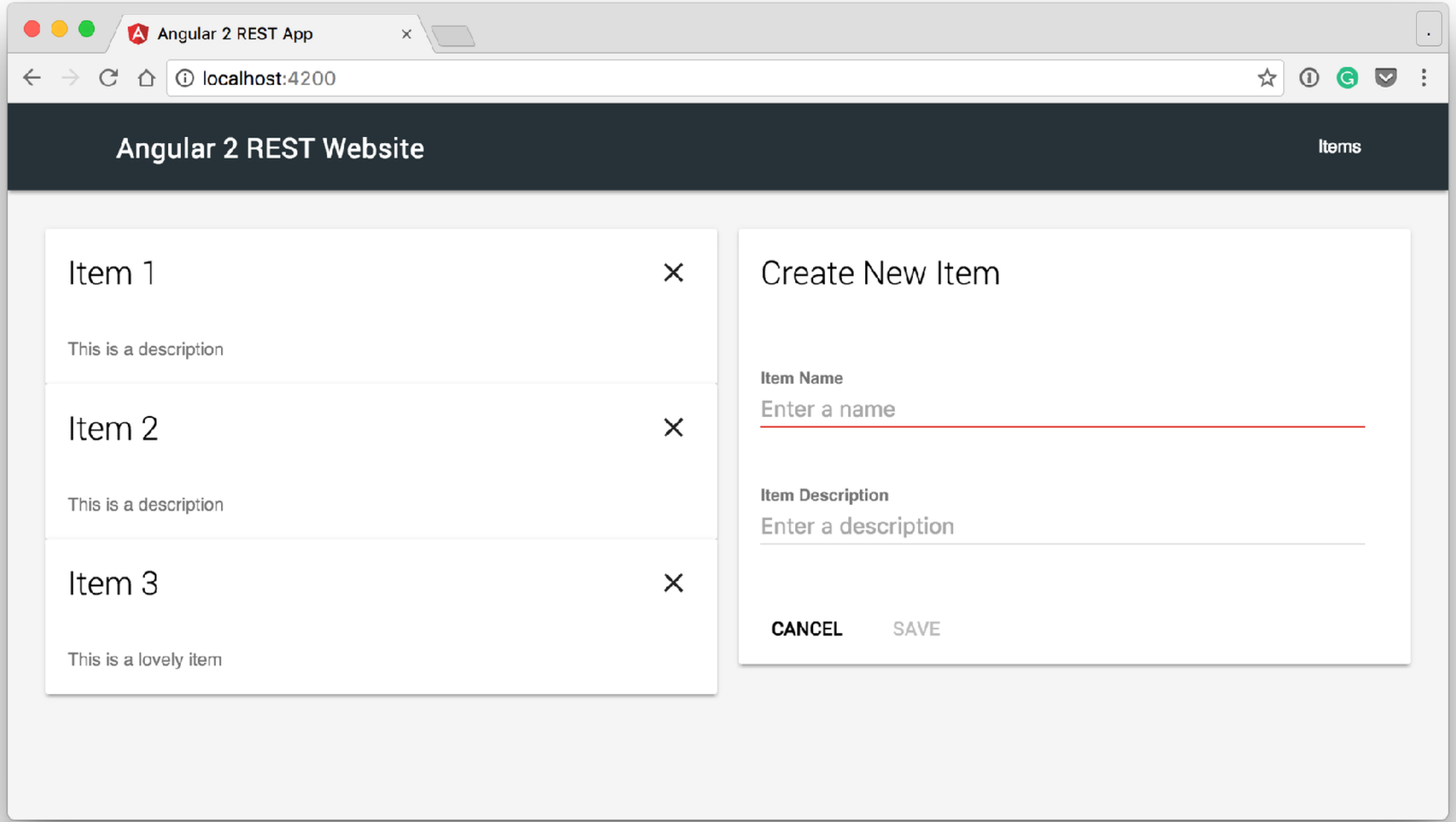
Strong grasp on how to **construct** and **compose** features in Angular 2

Agenda

- Review Challenge
- Component Driven Architecture
- Template Driven Forms
- Server Communication
- Observable Fundamentals

Getting **Started**

https://github.com/onehungrymind/ng2-rest-app


Angular 2 REST App
localhost:4200
Angular 2 REST Website
Items
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

The Demo Application

- A simple RESTful master-detail application built using Angular 2 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 2 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 {}
```

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?

```
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);
```

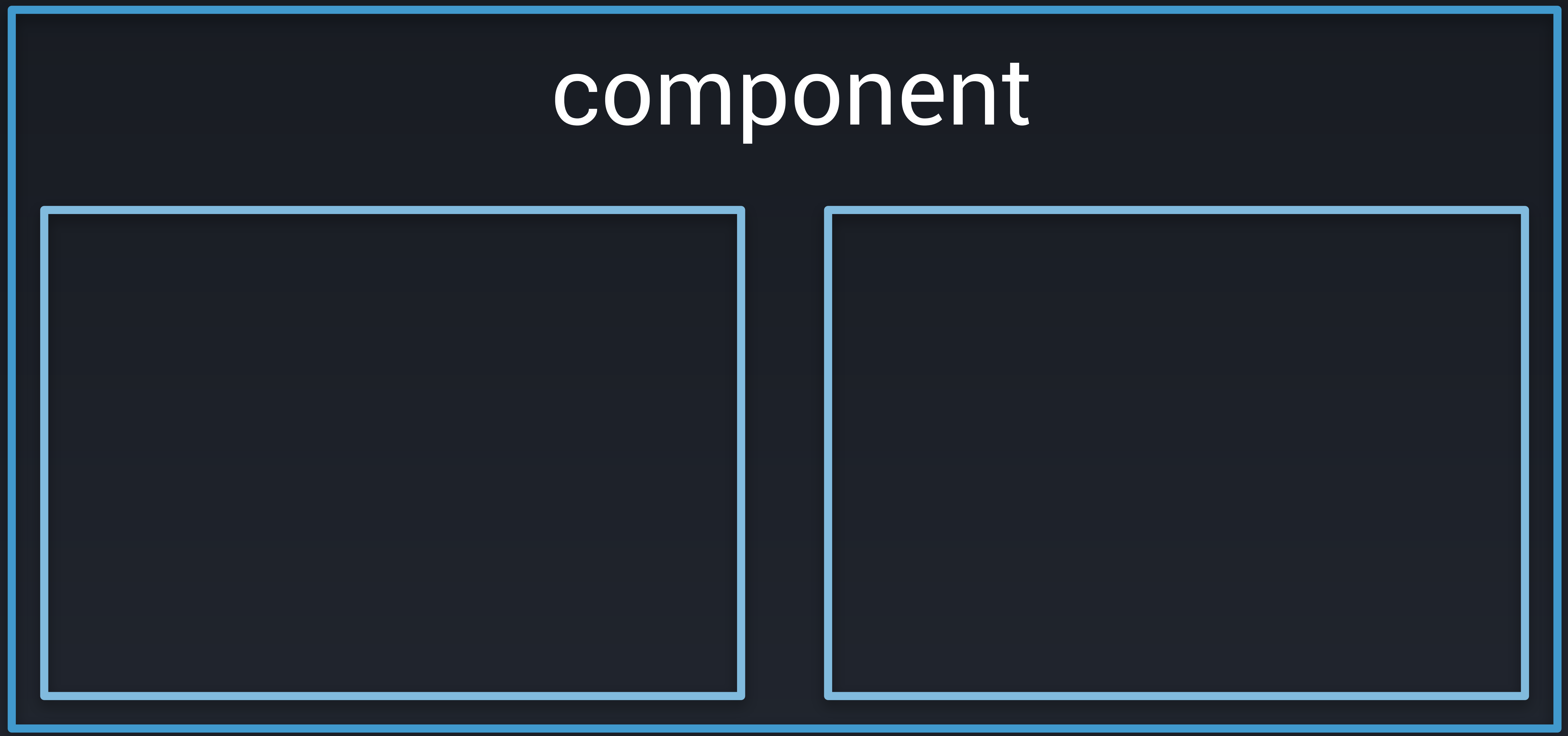
What is the basic structure of a route?

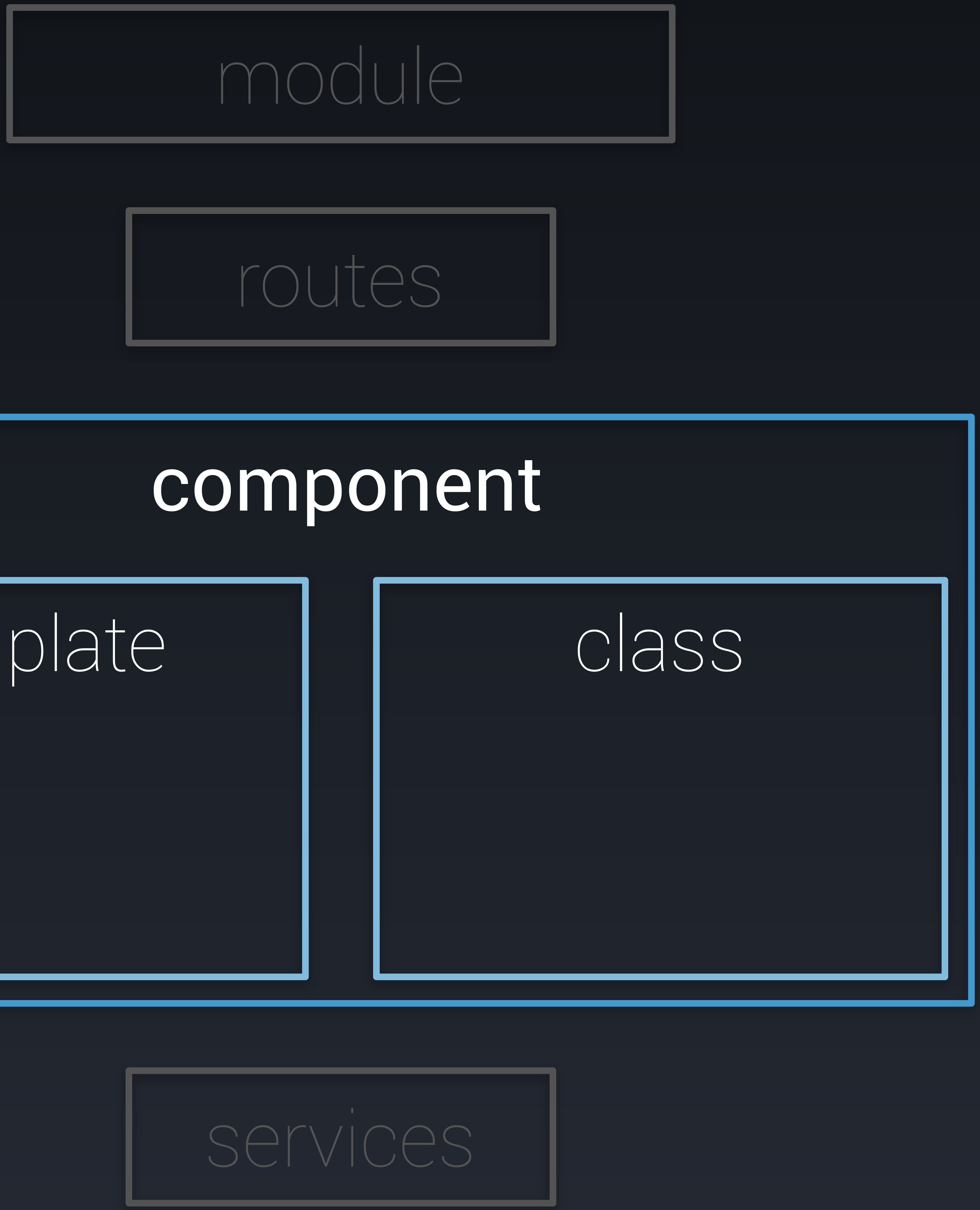
```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ItemsComponent } from './items/items.component';

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

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

What is does ** do?

```
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 { }
```

What are the two main pieces of a 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);
  }
}
```

How does **ngOnInit** work?

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

  constructor(private itemsService: ItemsService) {}

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

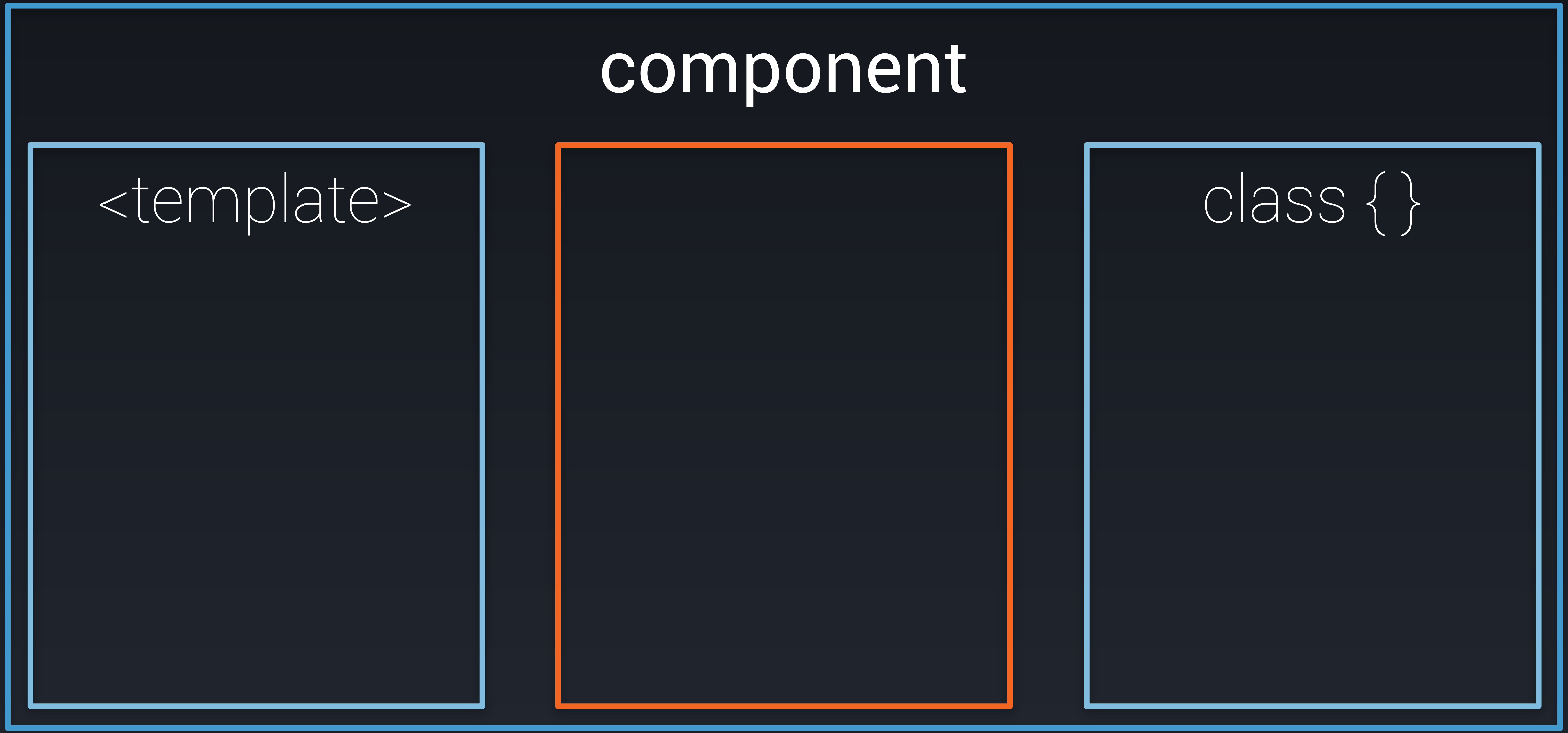
How do we inject a dependency into our component?

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

  constructor(private itemsService: ItemsService) {}

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

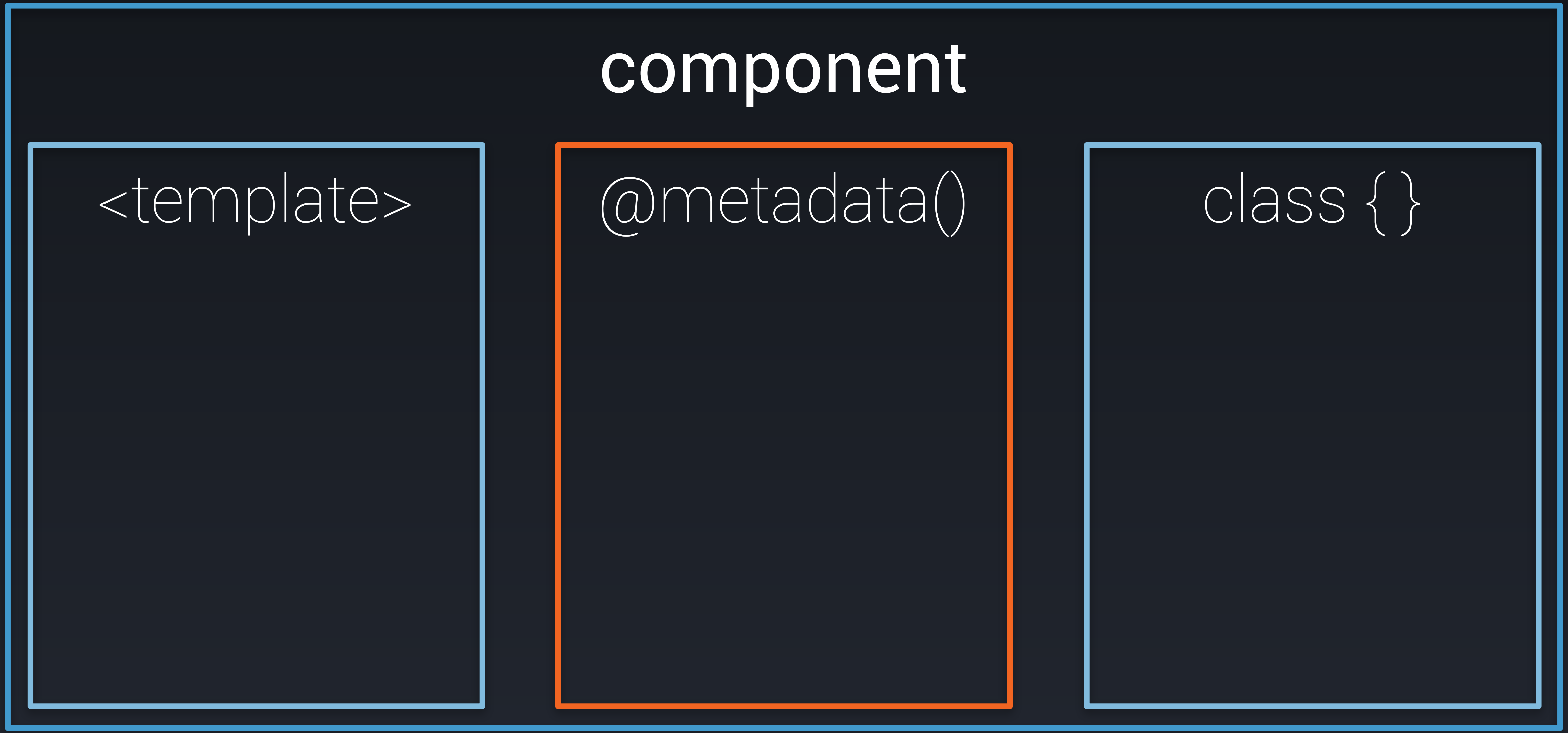
What does the keyword **private** do in the constructor?

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

  constructor(private itemsService: ItemsService) {}

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

How do we connect our template and component class?



Metadata



What are two things a component must have?

```
@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

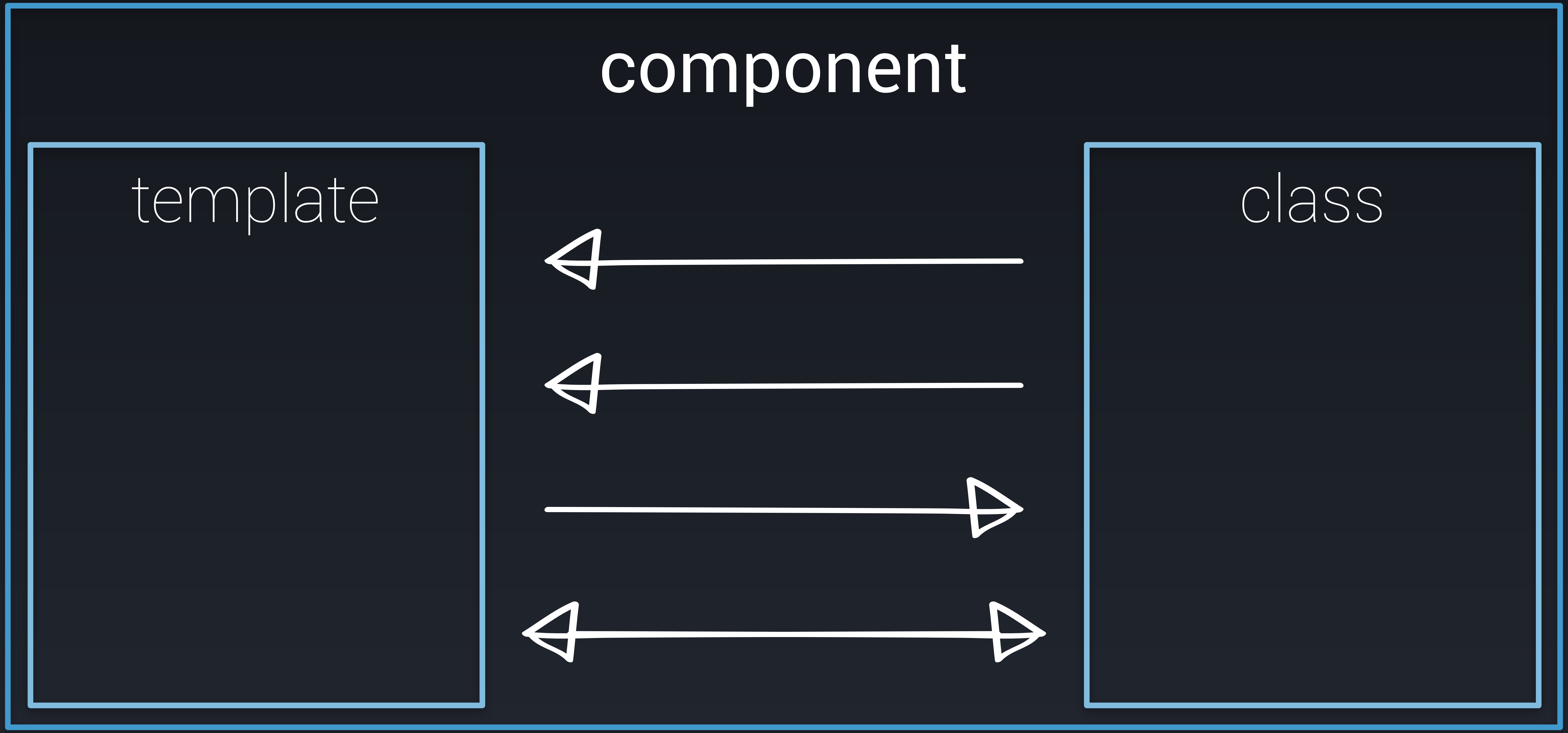
What is another way to define a component template?

```
@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();
}
```

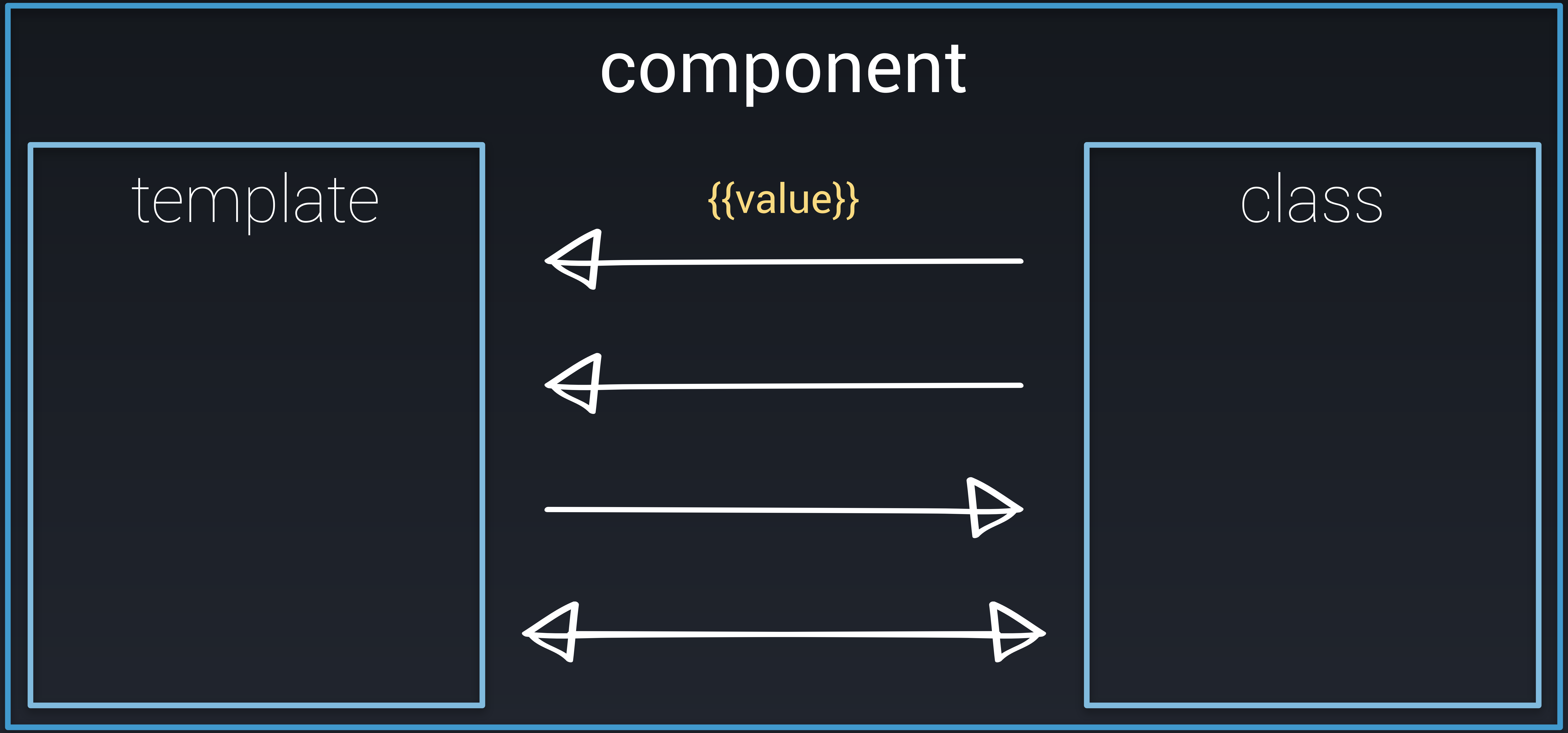
Why might we use one instead of the other?

```
@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']
})
```

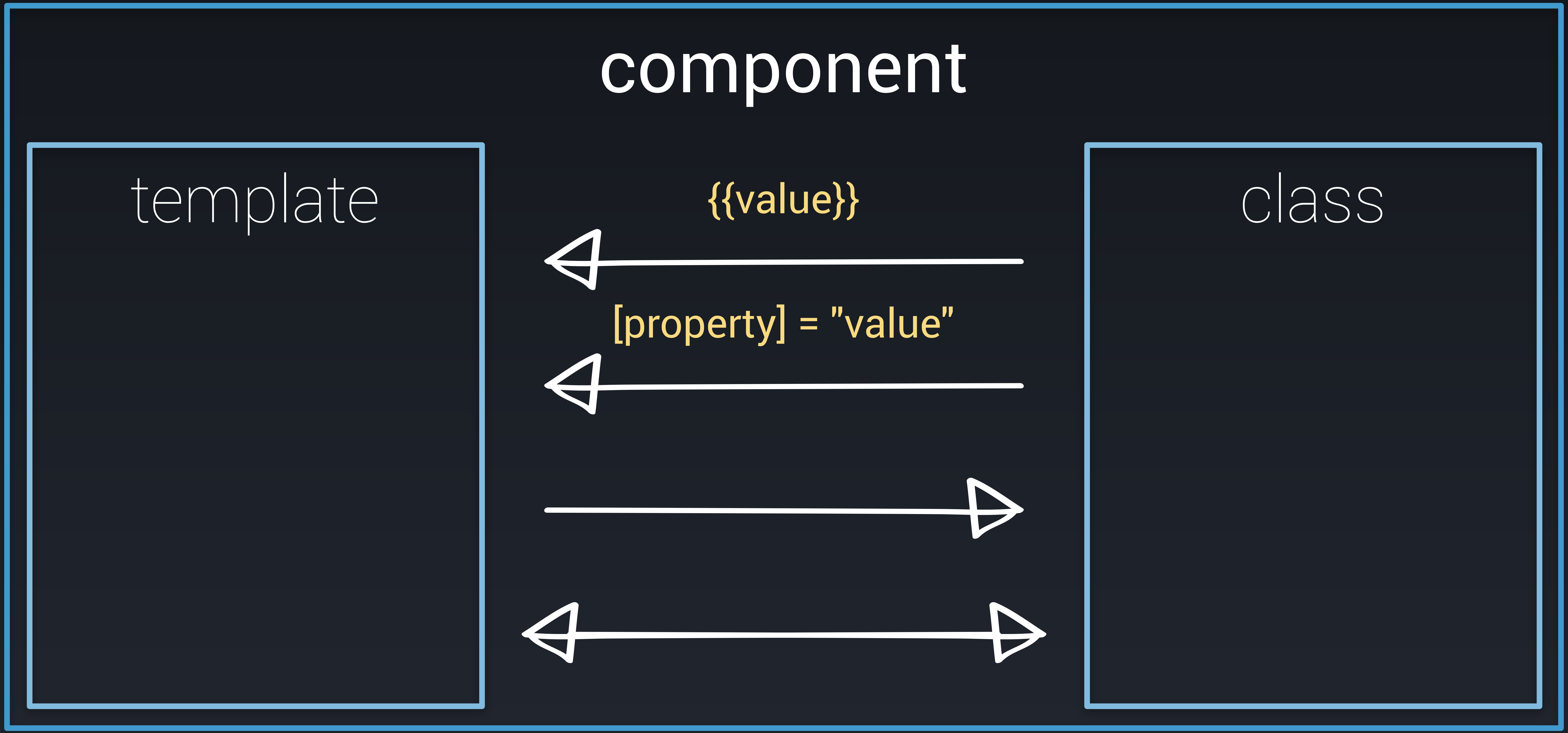
What is the easiest way to bind to a simple text value?



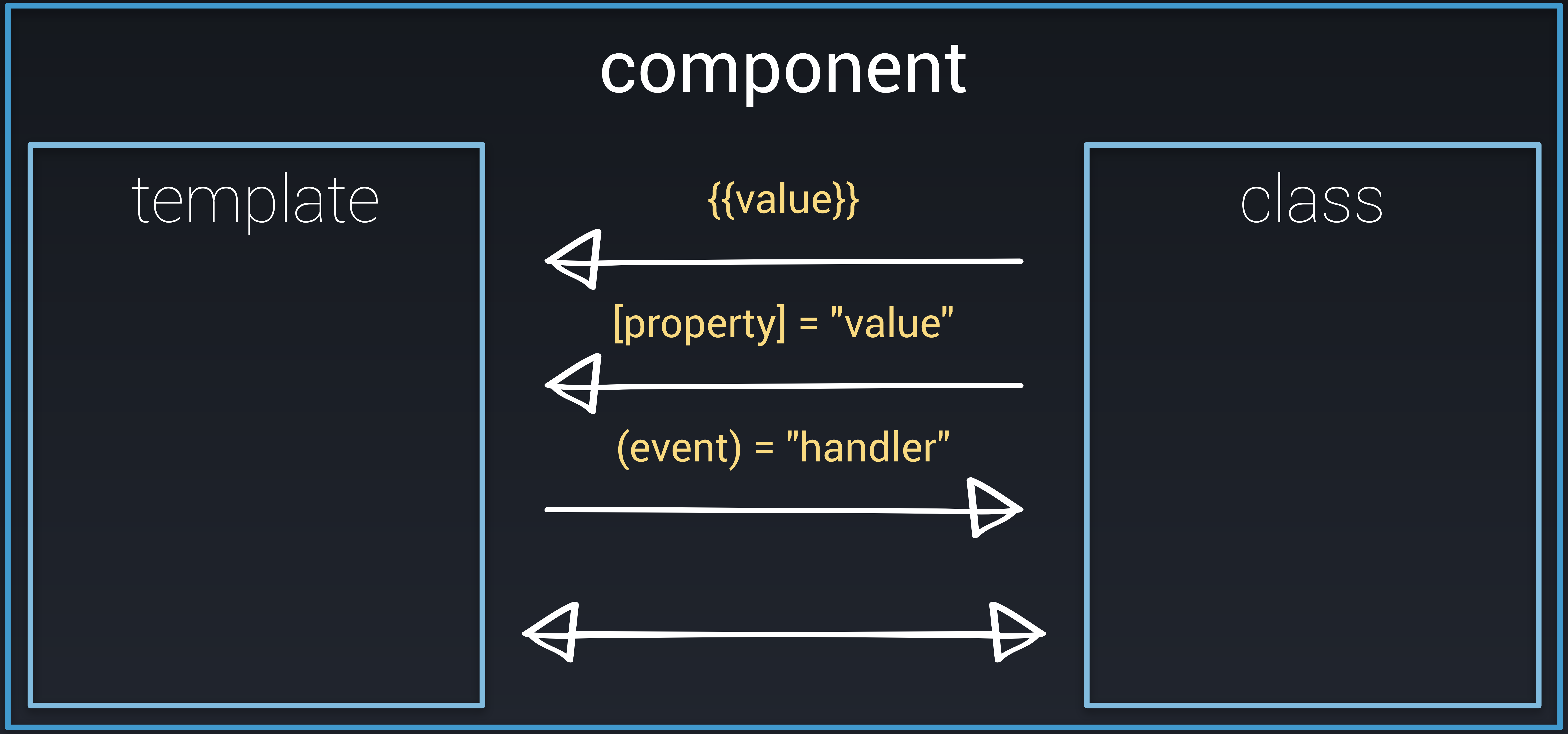
What is the best binding for sending data to the template?



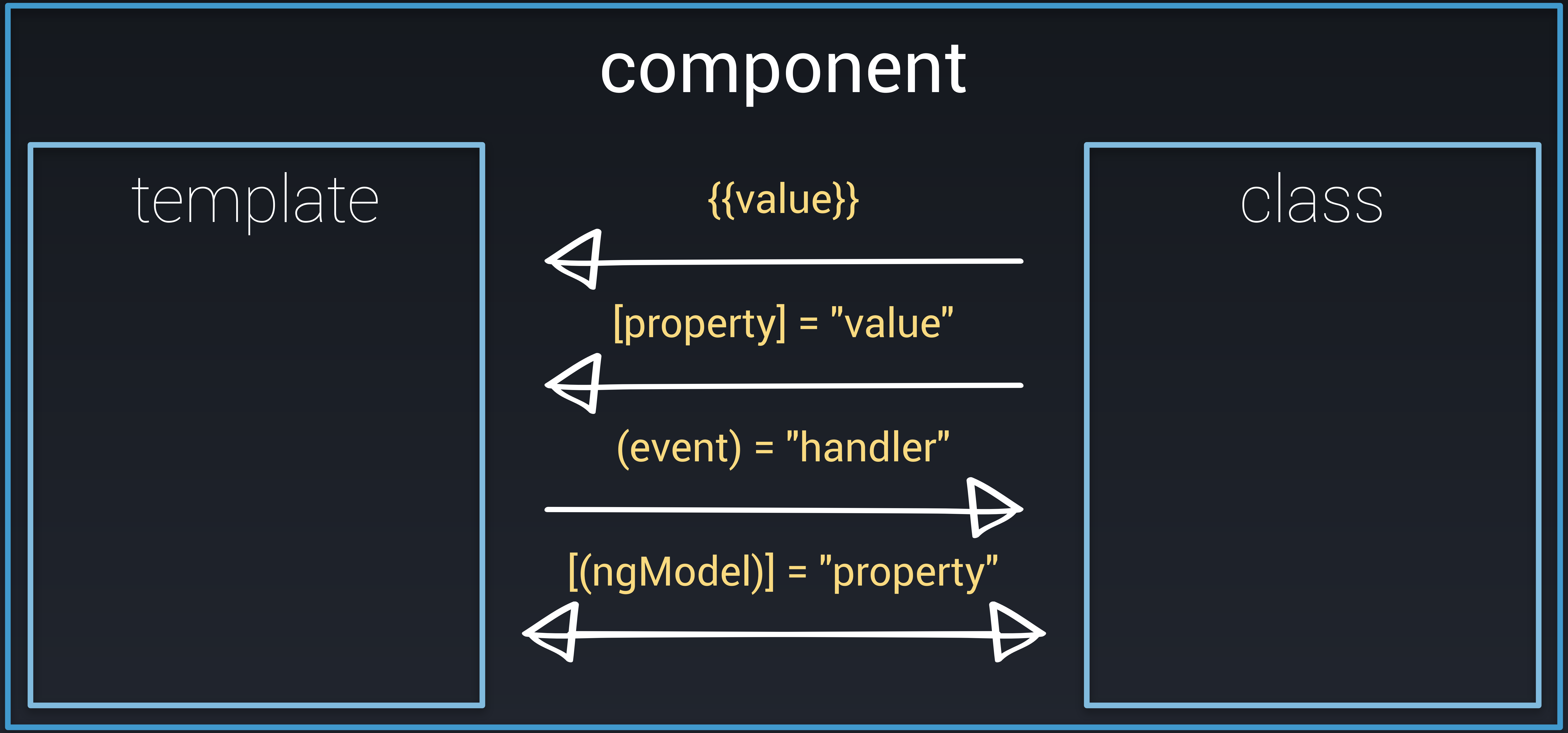
How do we communicate from the template to the class?



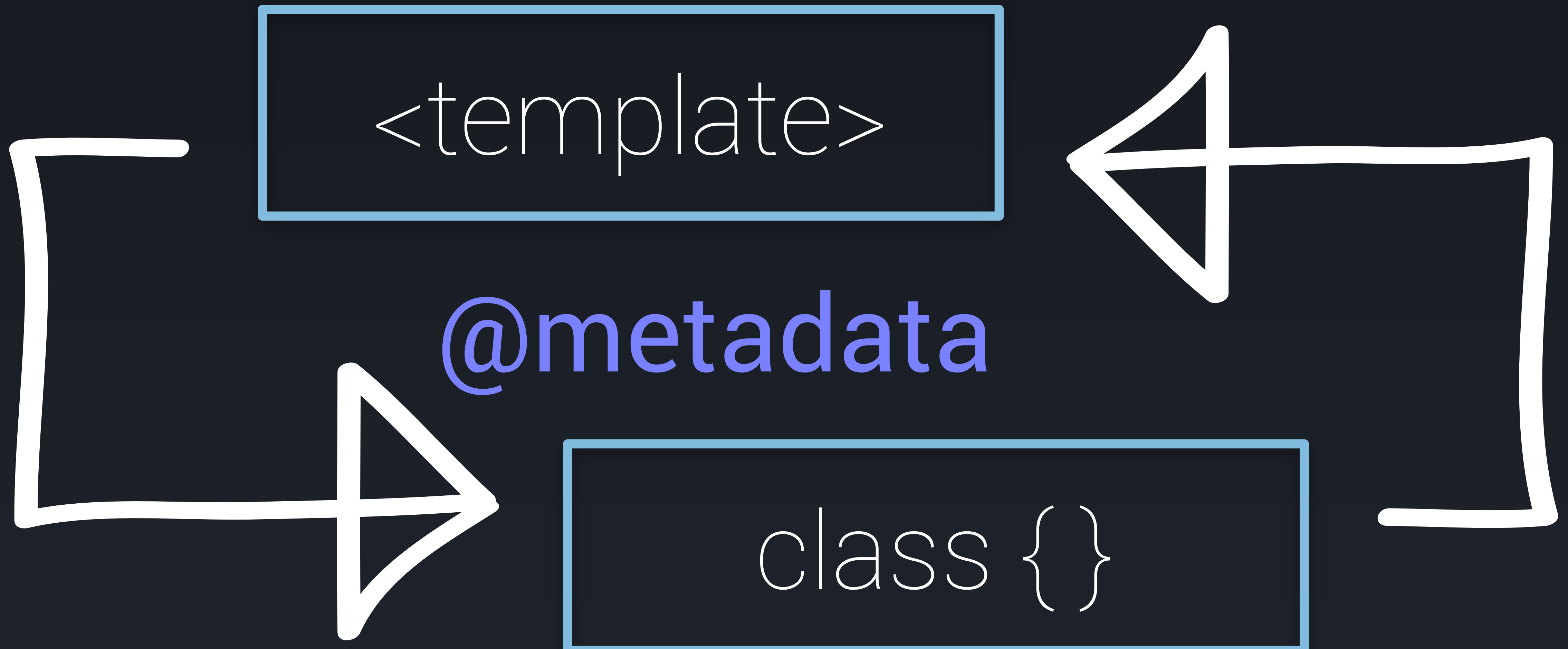
How do we keep a value in sync between both?

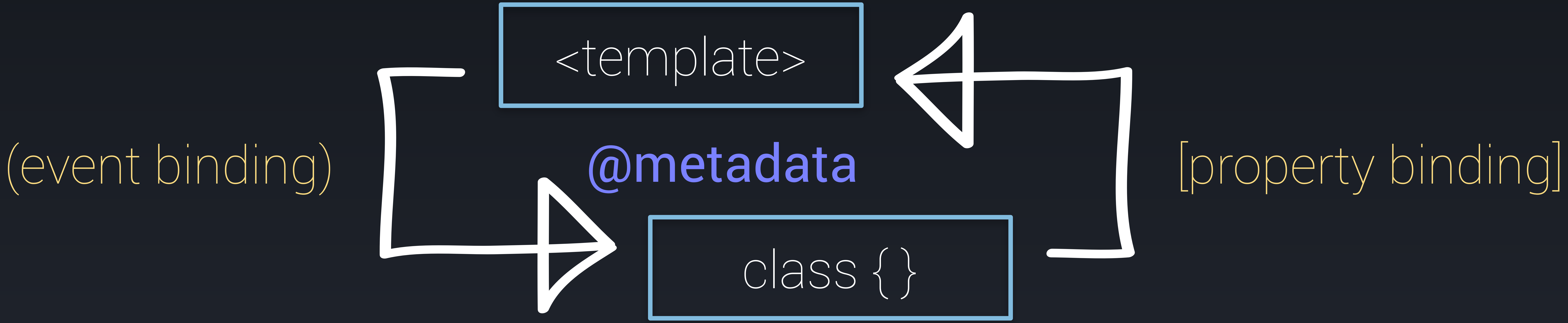


How does two way data binding really work?



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




(event binding)
<template>
@metadata
class { }
[property 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

```
<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

How do components and directives differ?

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

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

Directives

```
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?

```
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();
  }
}
```

Challenges

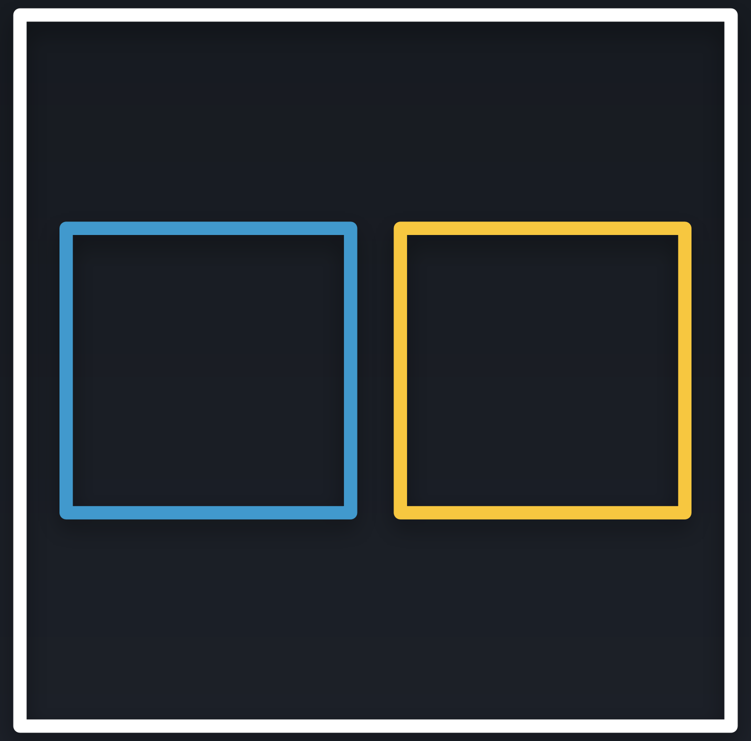
- In your working example, create a new **review** feature including file structure, component class and template
- Make it available to the rest of the application
- Display the **review component** in the application via its HTML **selector**
- Display the **review component** in the application via 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

Component Driven
Architecture

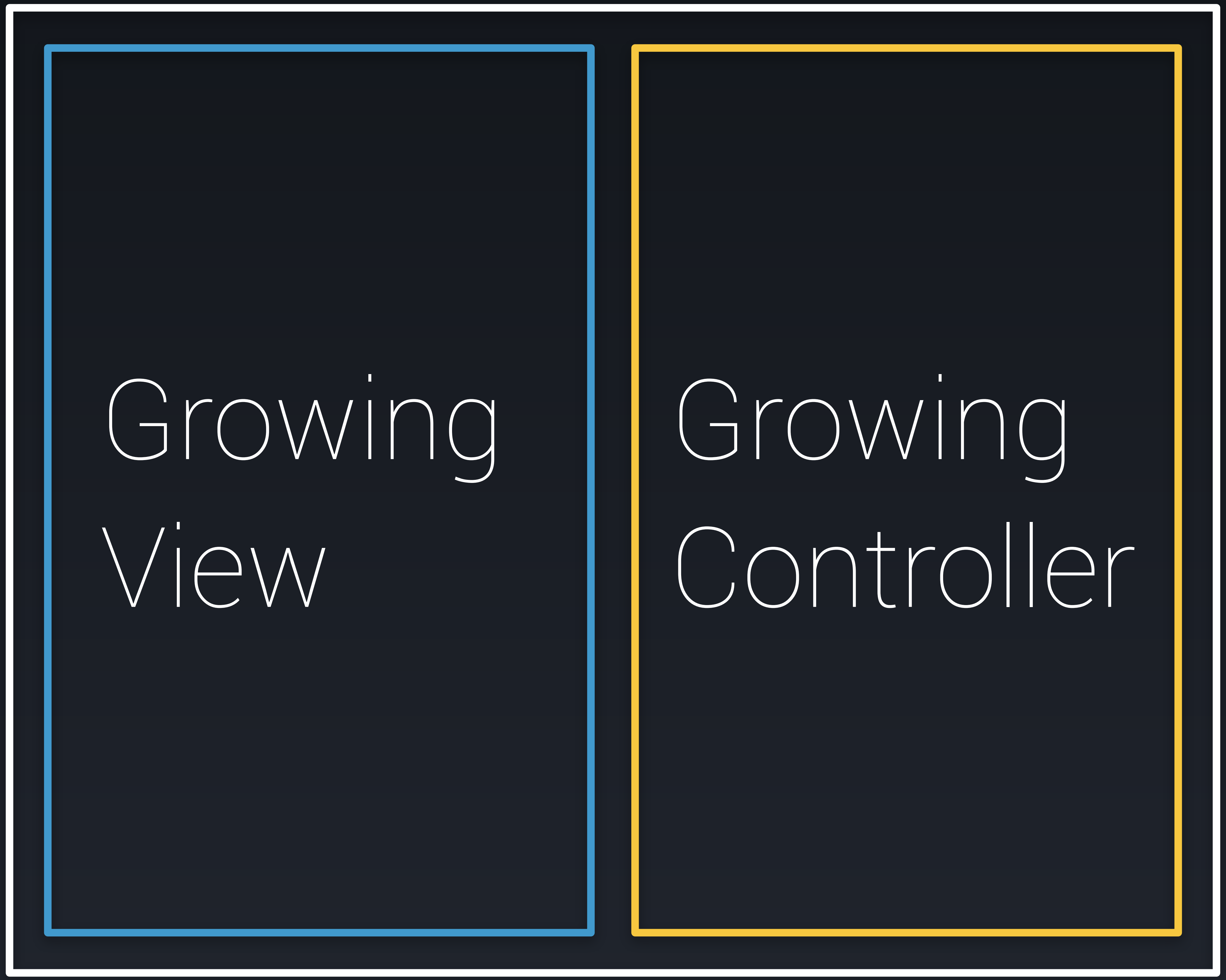
Component Driven Architecture

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

A Brief History of Angular

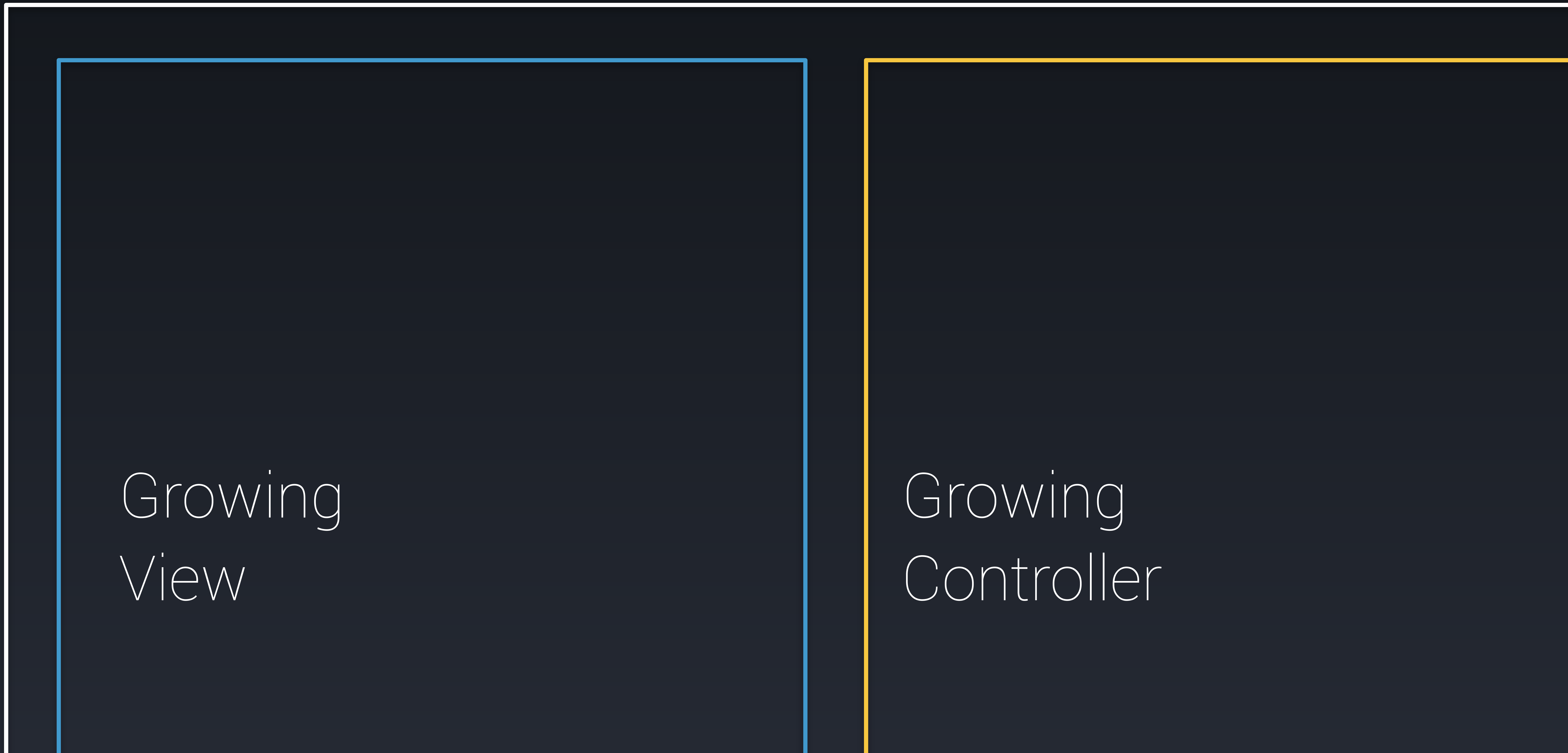


tiny app == tiny view + tiny controller

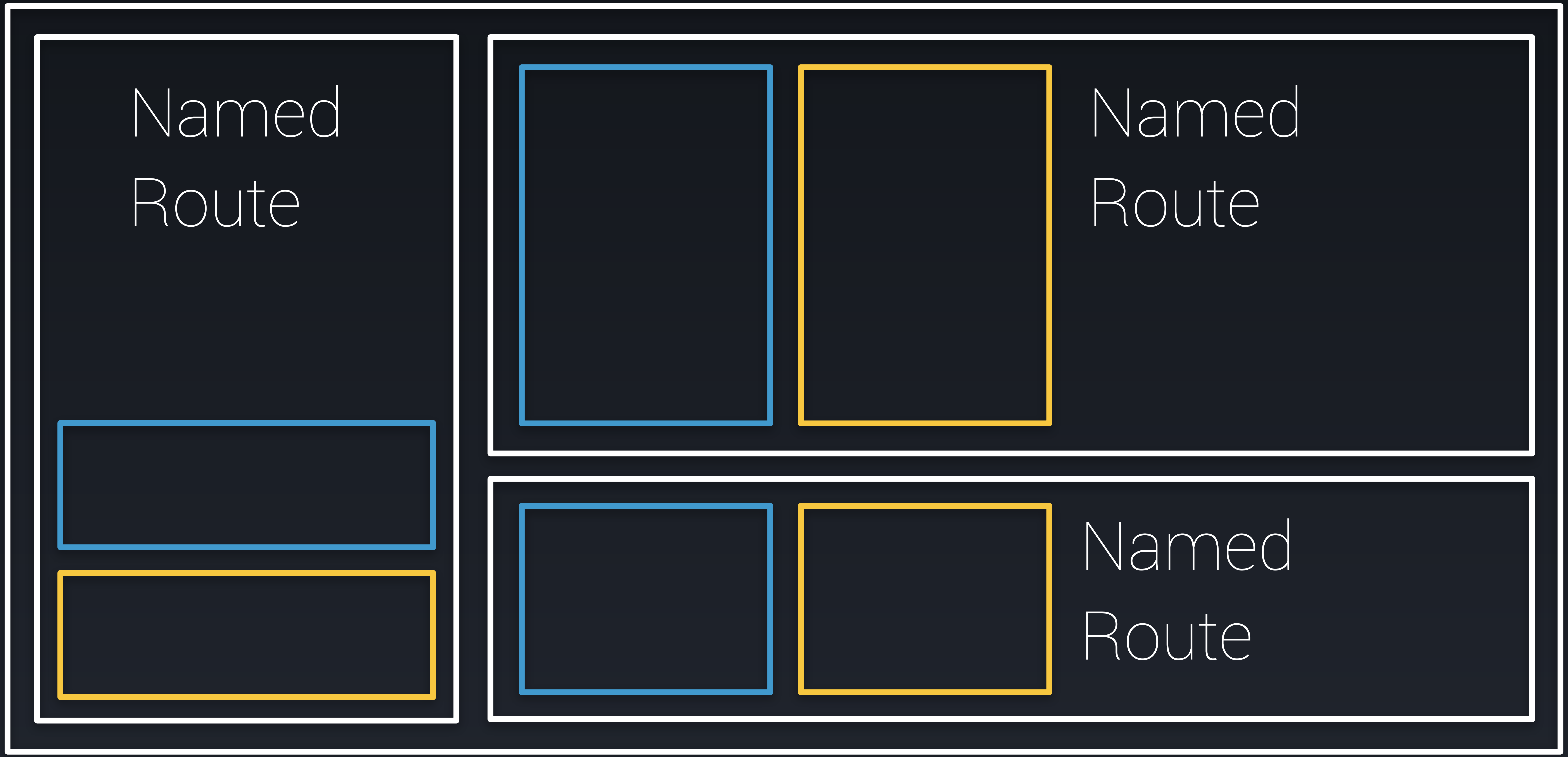


Growing Application

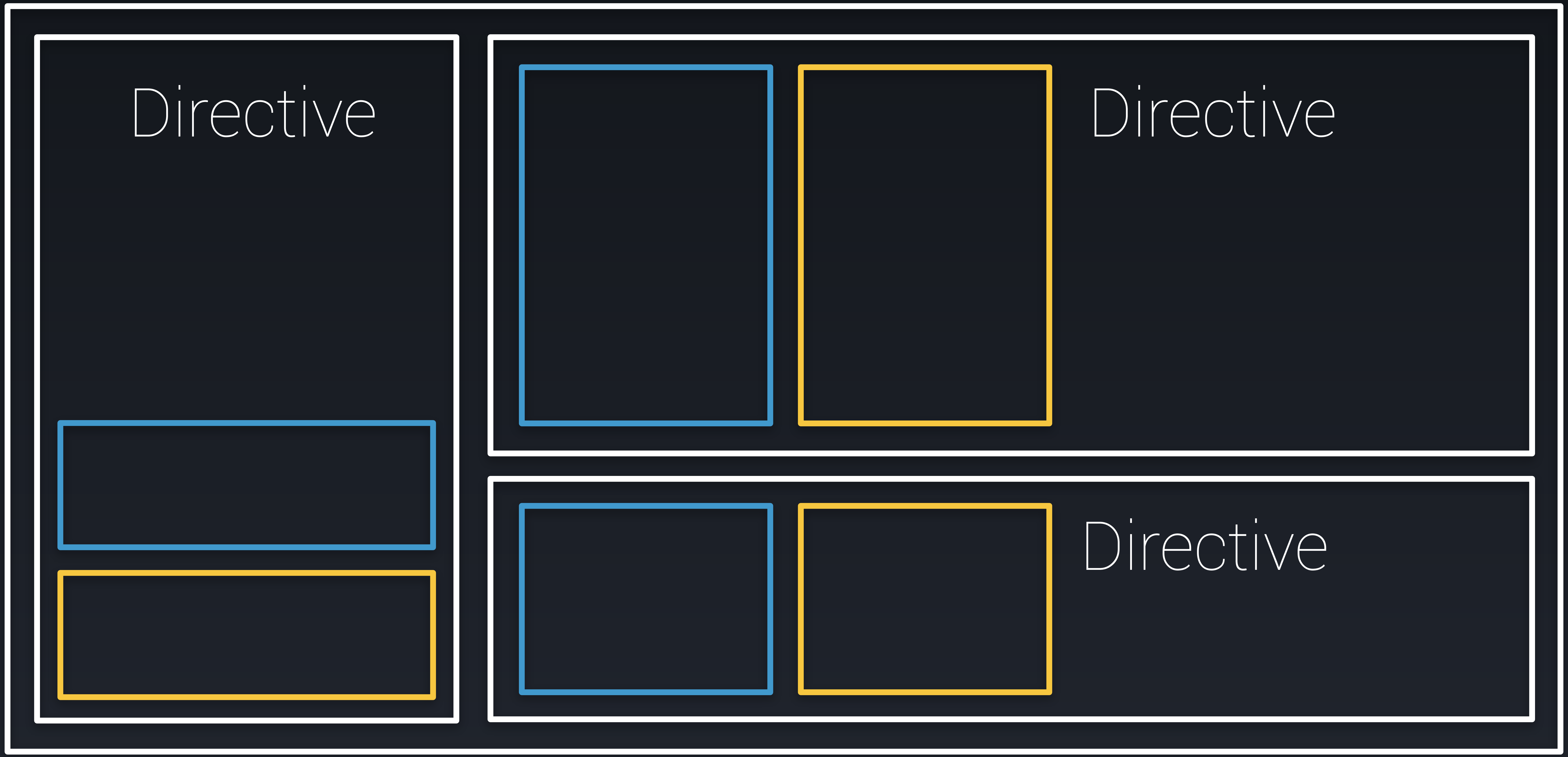
Realistic Application



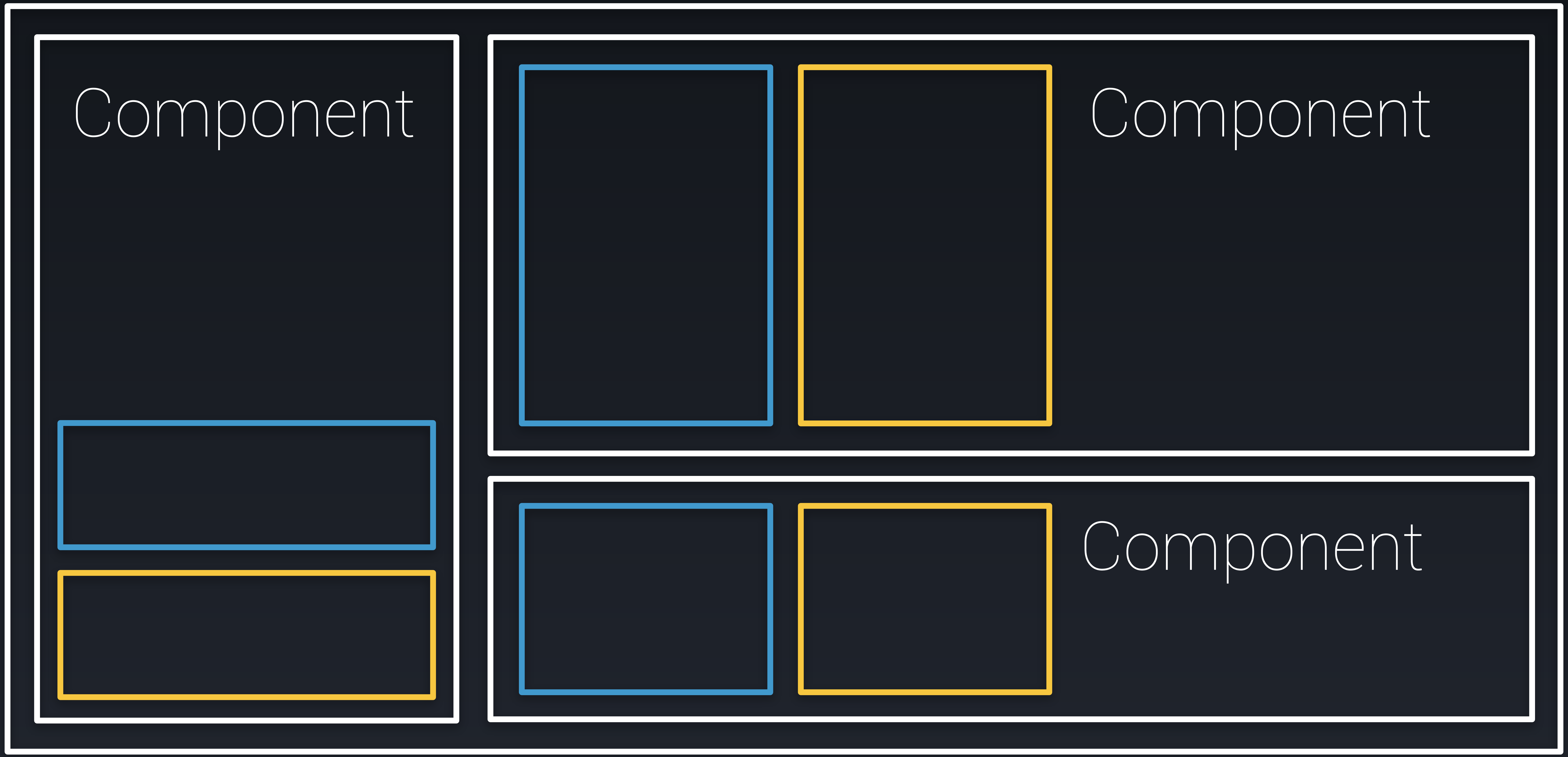
Uh oh!



Large 1.x Application



Large 1.x Application



Any Angular 2 Application

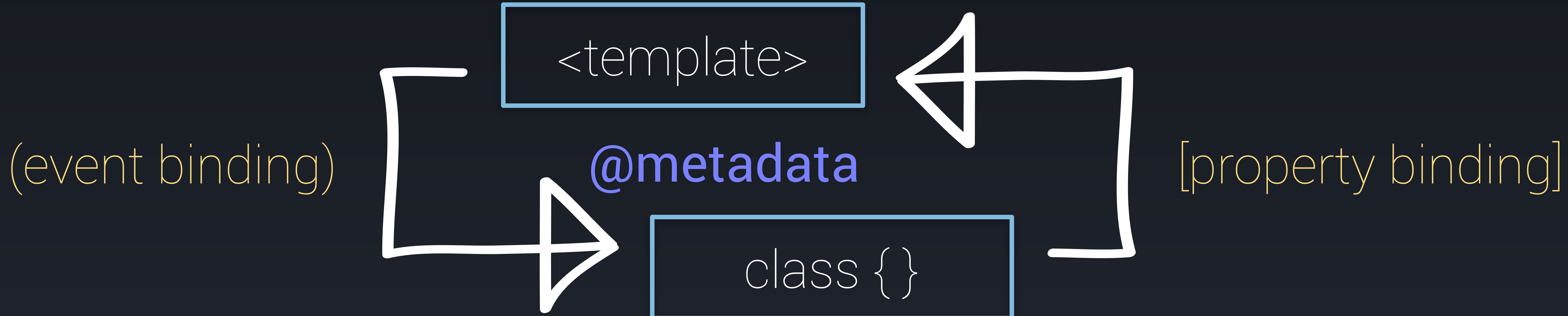
Structure

Communication

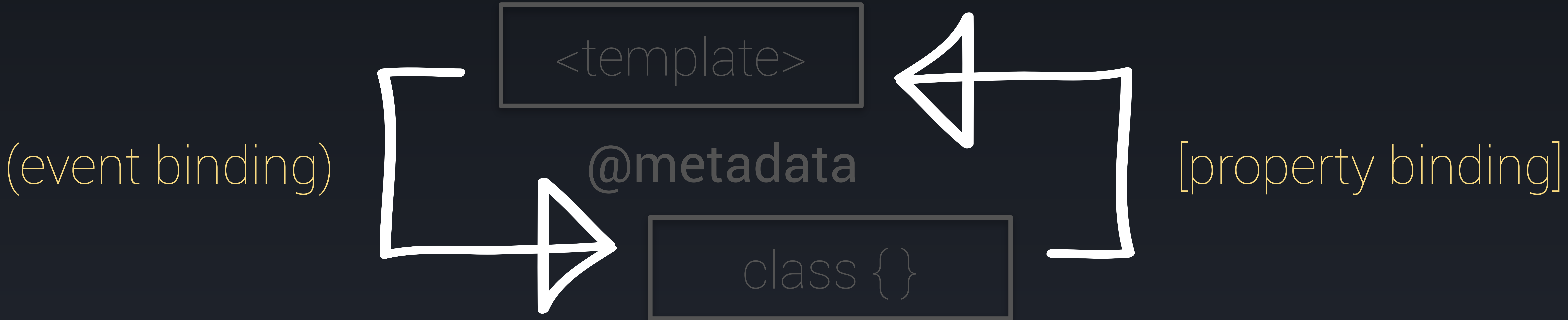
Component Driven Architecture

- 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 self encapsulated building blocks that contain their own templates, styles, and logic so that they can easily be ported elsewhere

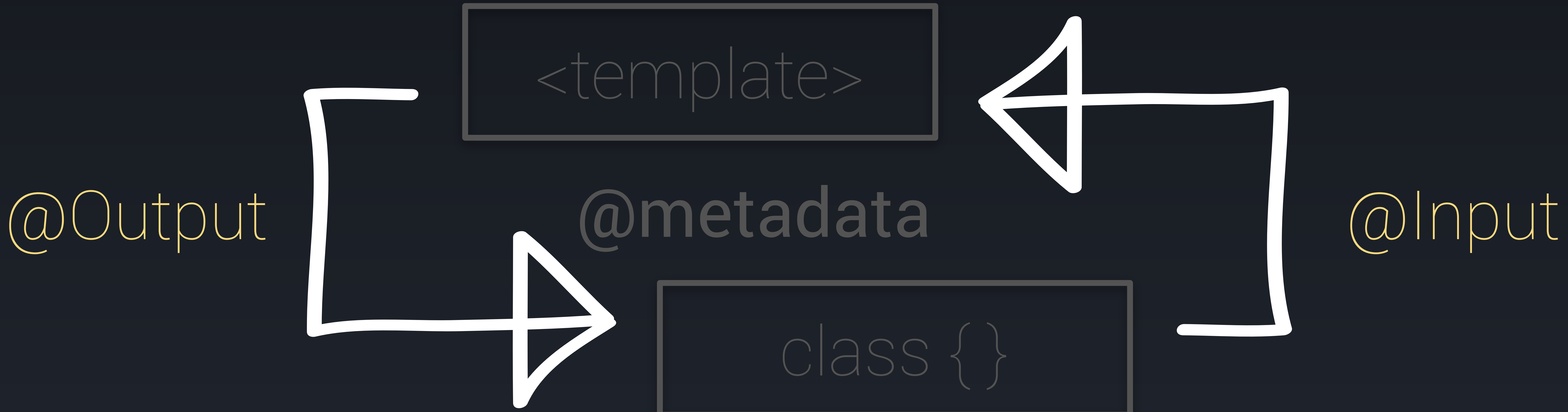
Data Binding



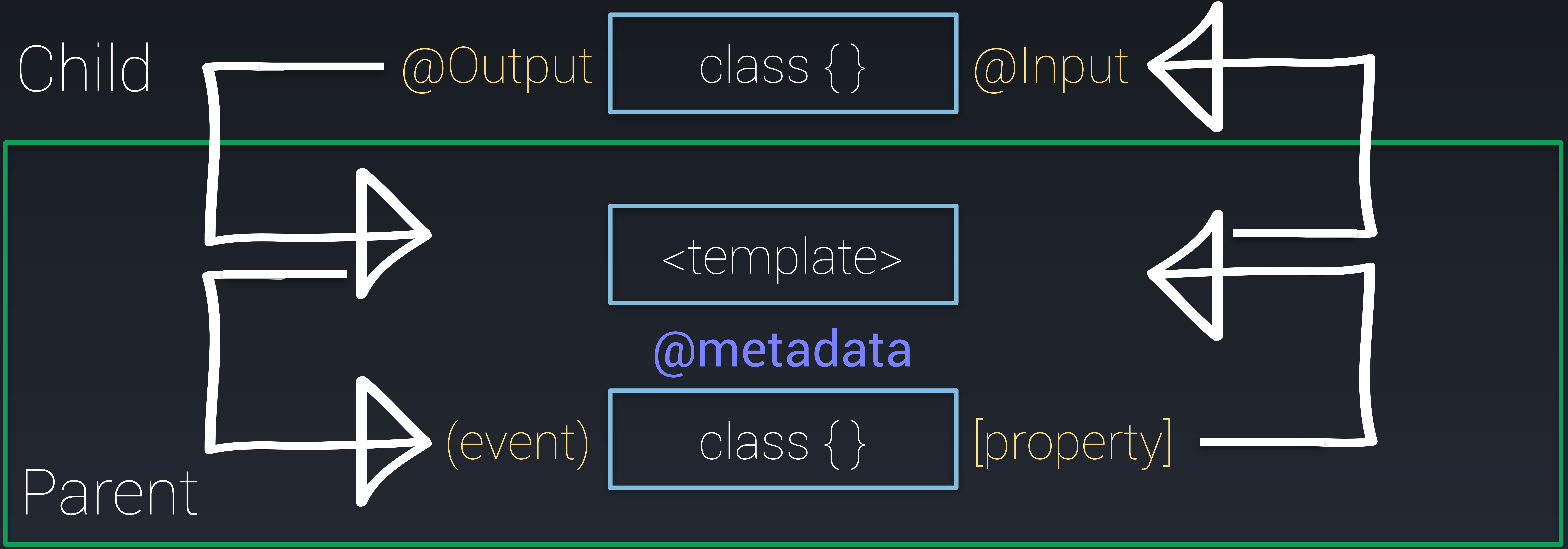
Custom Data Binding



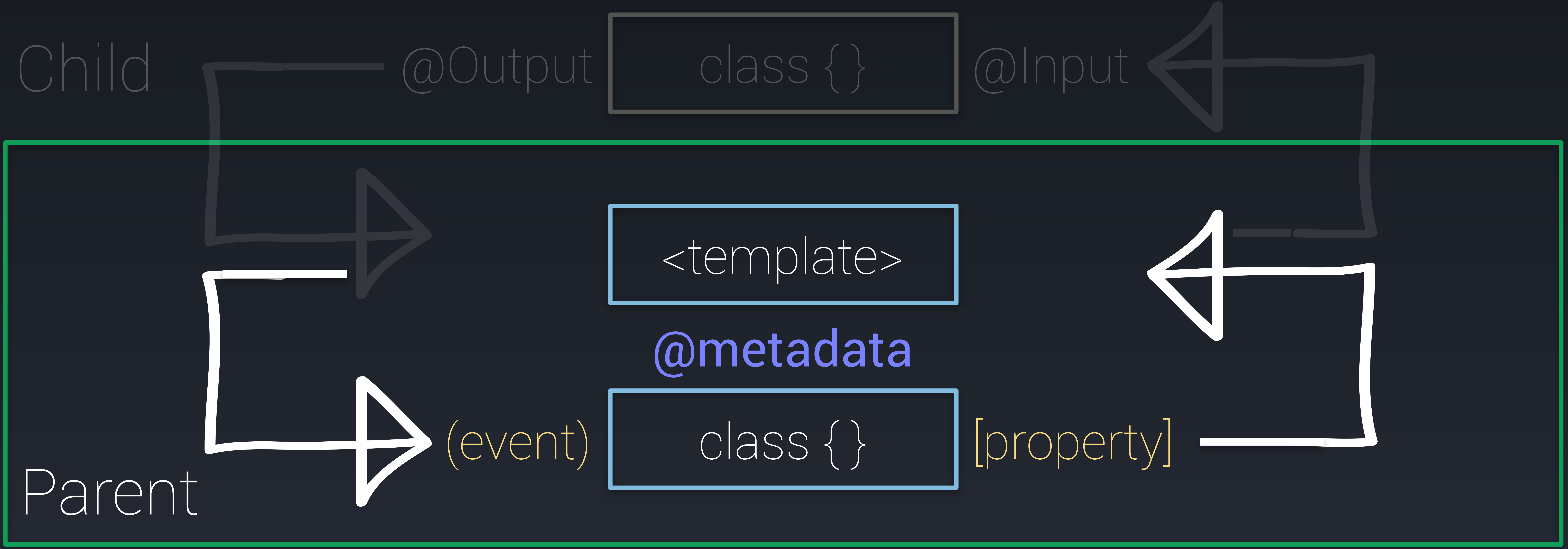
Component Contract



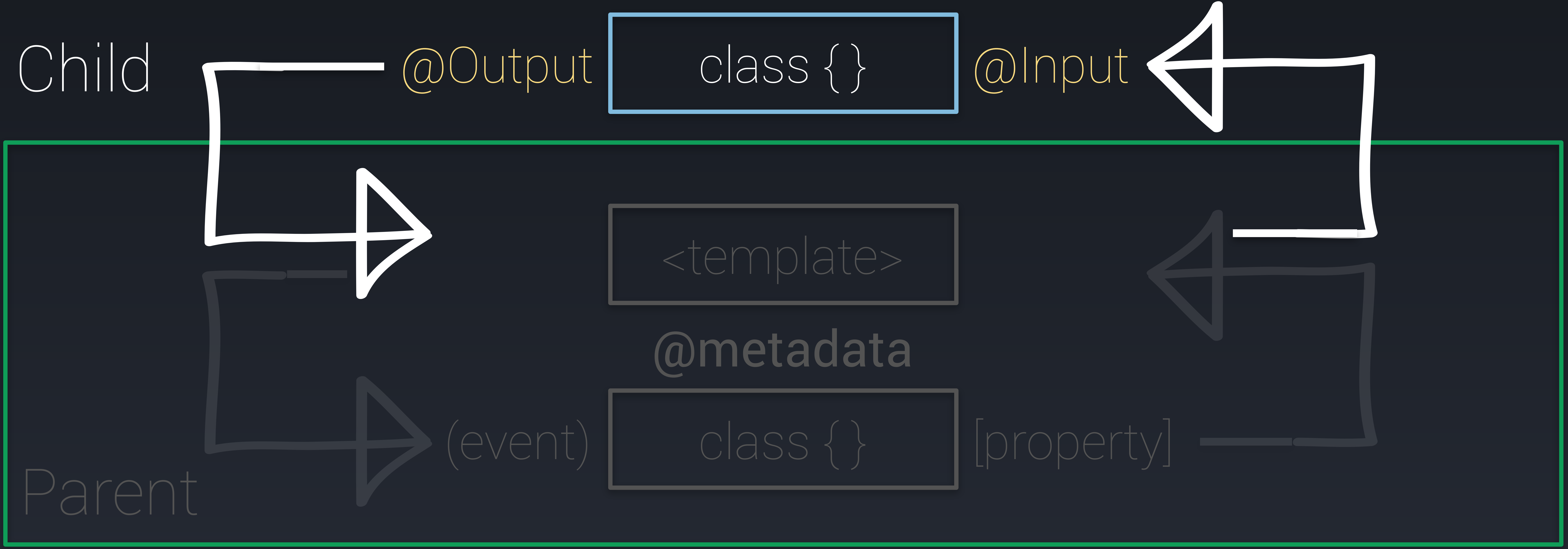
Parent and Child



Parent and Child



Parent and **Child**



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;**

```
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

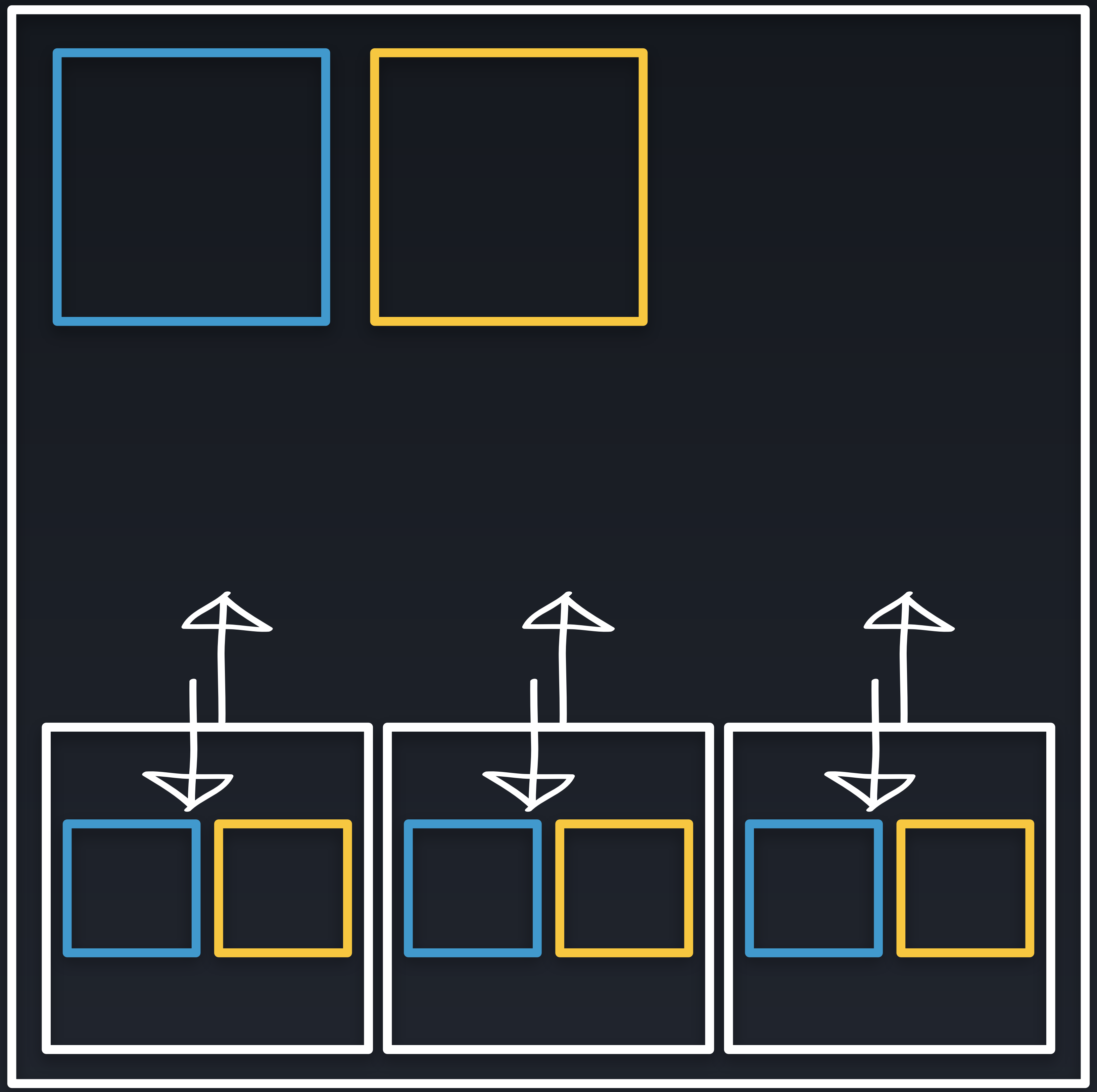
- Exposes an **EventEmitter** property that emits events to the parent component
- Defined inside a component via the @Output decorator: **@Output() showValue = new EventEmitter();**
- Bind in parent template: **<cmp (someValue)="handleValue()"></cmp>**

```
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

```
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 **delete** output event in the **widgets-list** component
- Create a **save** output event in the **widget-details** component
- Create a **cancel** output event in the **widget-details** component

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

```html
<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 **FormControls** 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

```html
<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**
- Use a **lifecycle hook** to isolate the **widget** mutation
- Create a button to **save** the edited **widget** to the parent component
- Create a button to **cancel** editing the **widget** to the parent component
- Using **ngForm**, add in some validation for editing the **widget** component

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

```
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**

```
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

```
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

```
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

- We should **always** handle errors
- 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

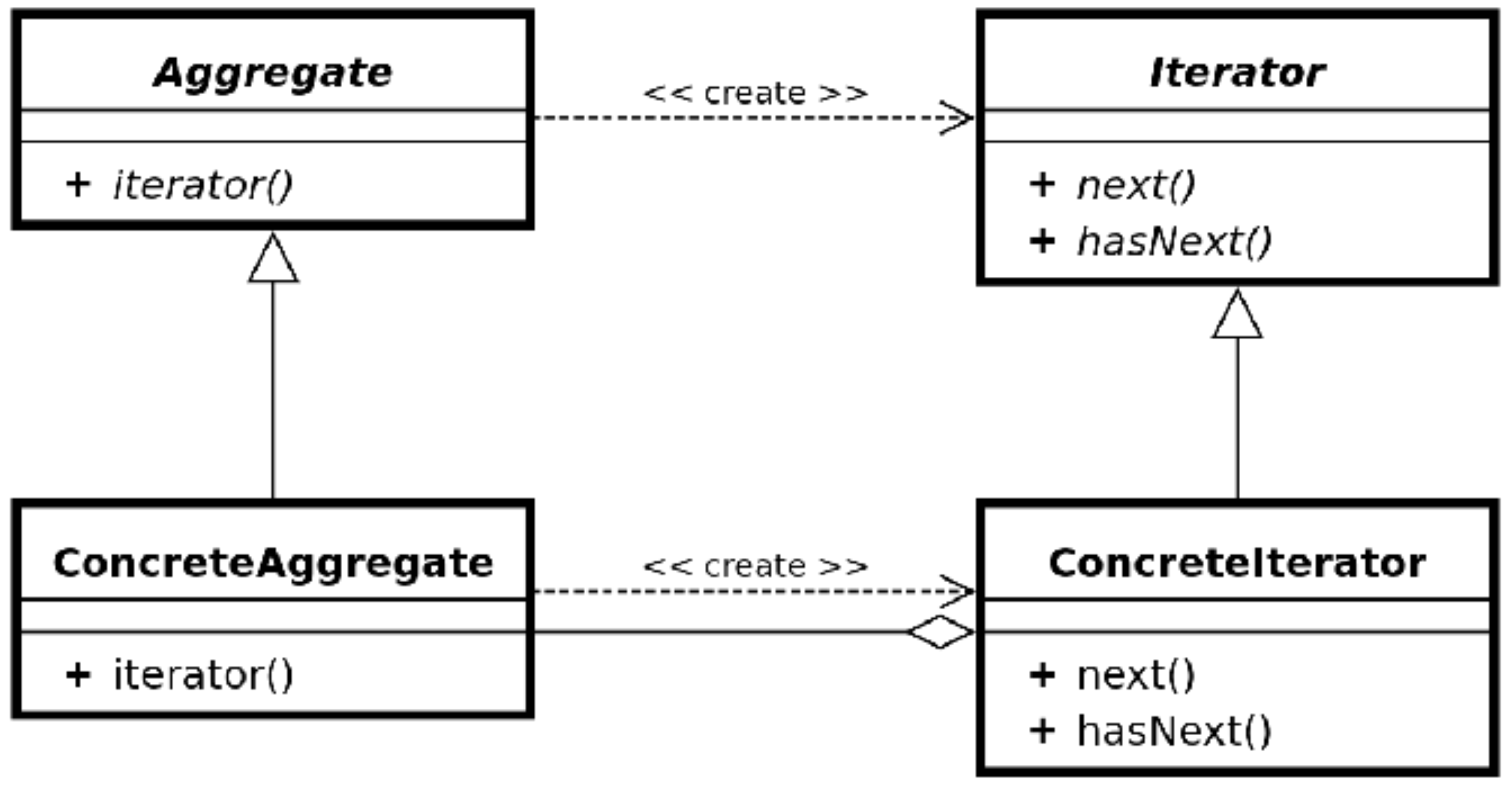
- Replace the local **widgets** collection with a call to the **widgets** endpoint
- Update the **widgets** component to handle the async call
- Flesh out the rest of the **CRUD** functionality using **ItemsService** as reference
- **BONUS Try to use Observable.subscribe**

Observable
Fundamentals

Observable Fundamentals

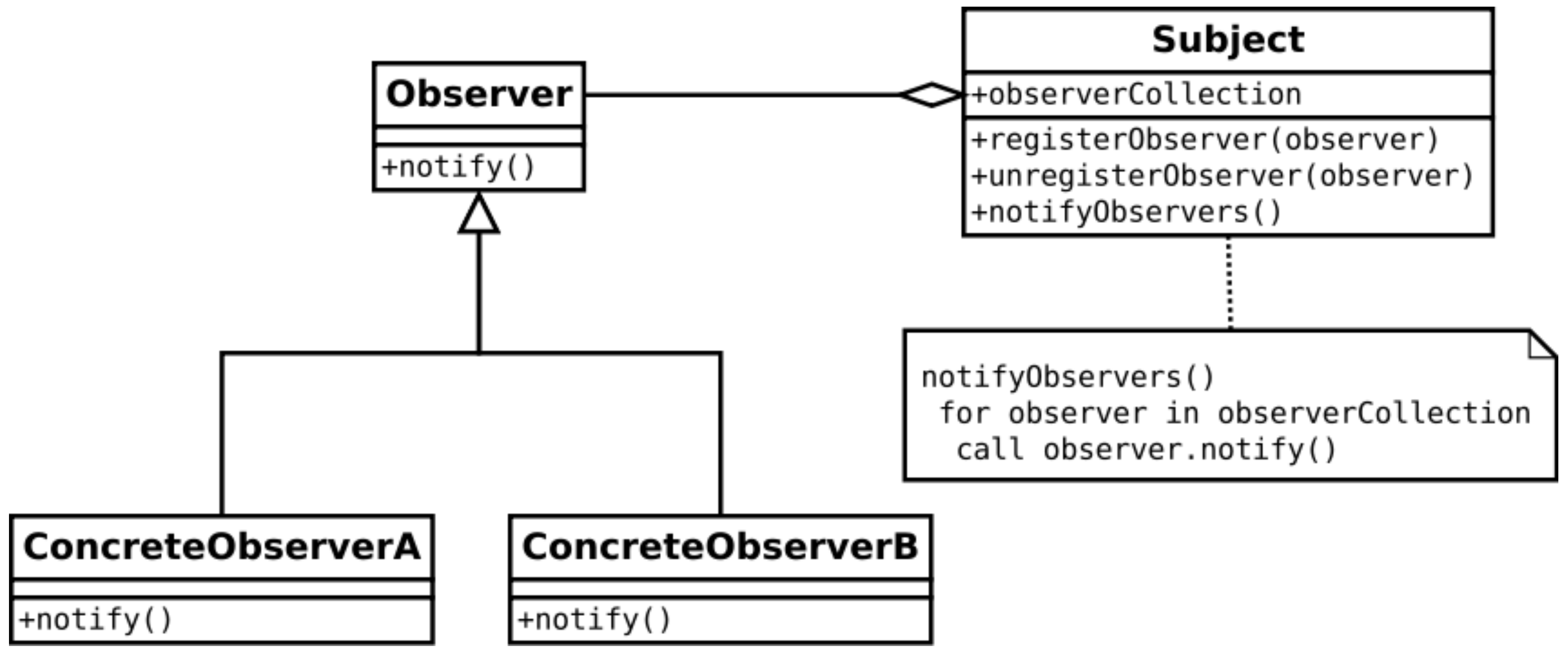
- Basic Observable Sequence
- Observable.map
- Observable.filter

Iterator Pattern



State

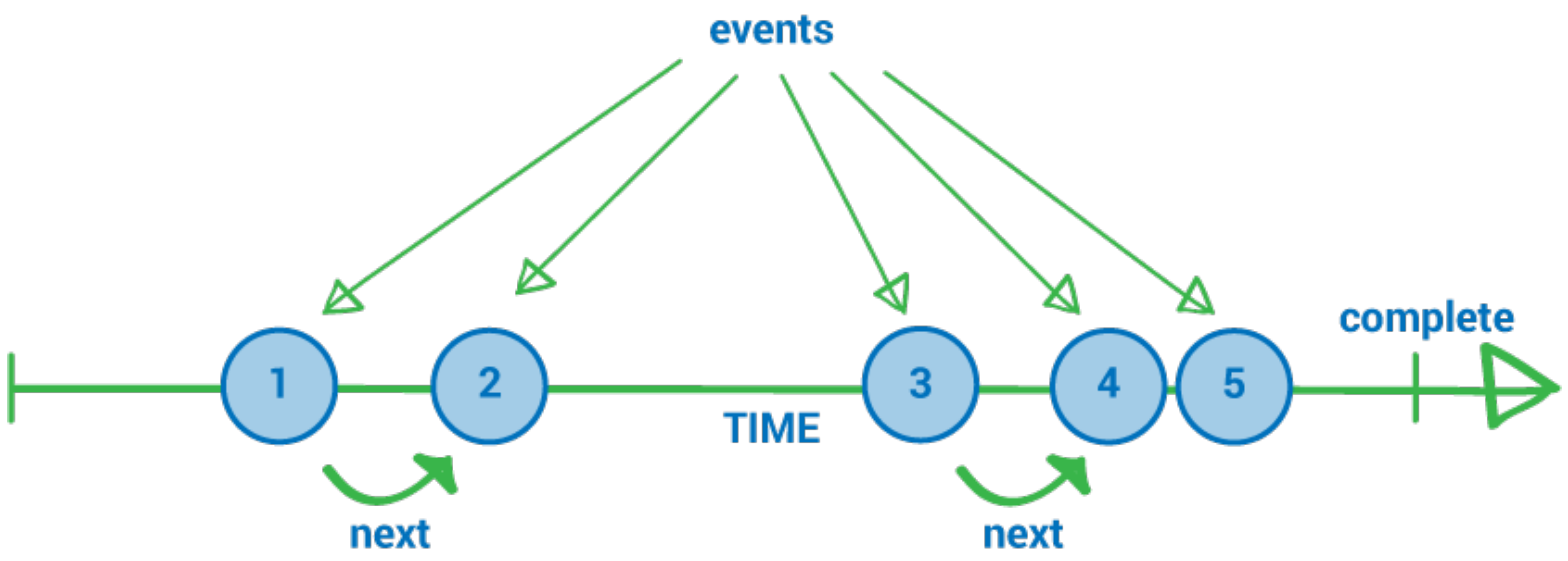
Observer Pattern



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

Communication

Communicate
state over **time**



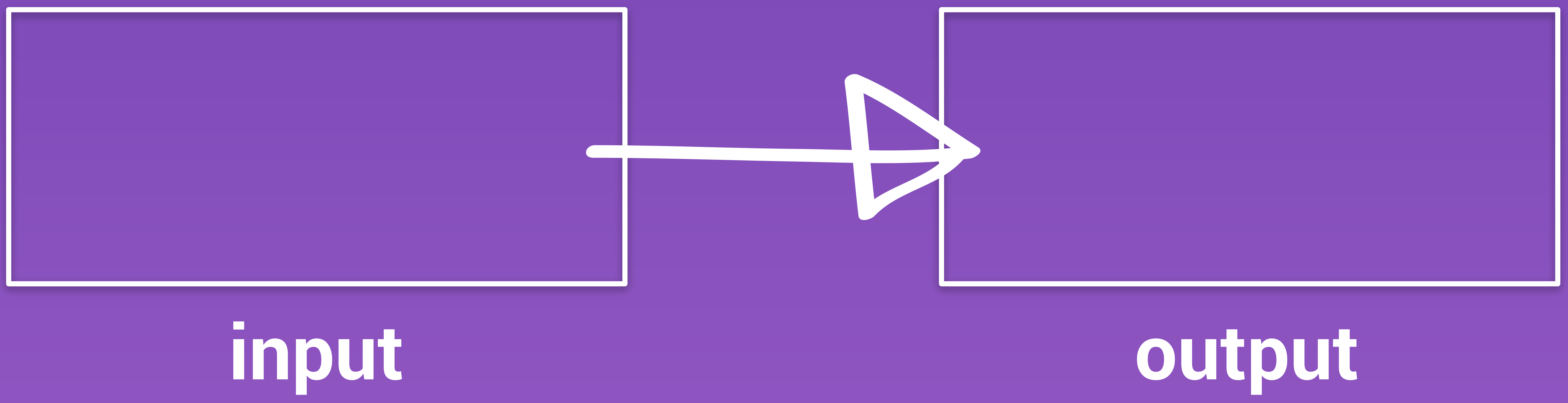
Observable stream

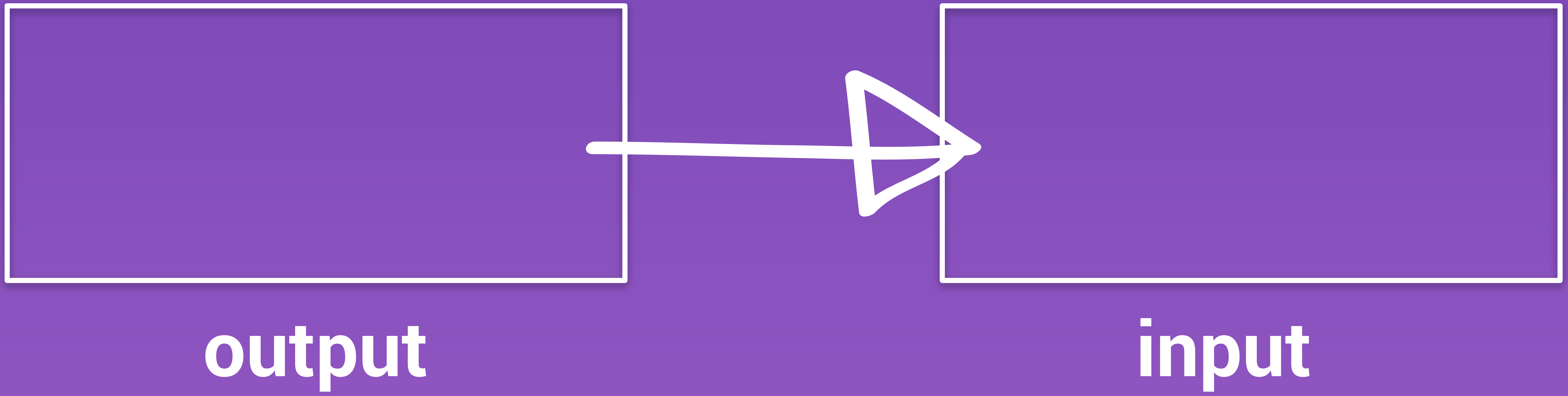
	SINGLE	MULTIPLE
SYNCHRONOUS	Function	Enumerable
ASYNCHRONOUS	Promise	**Observable**

Values over time

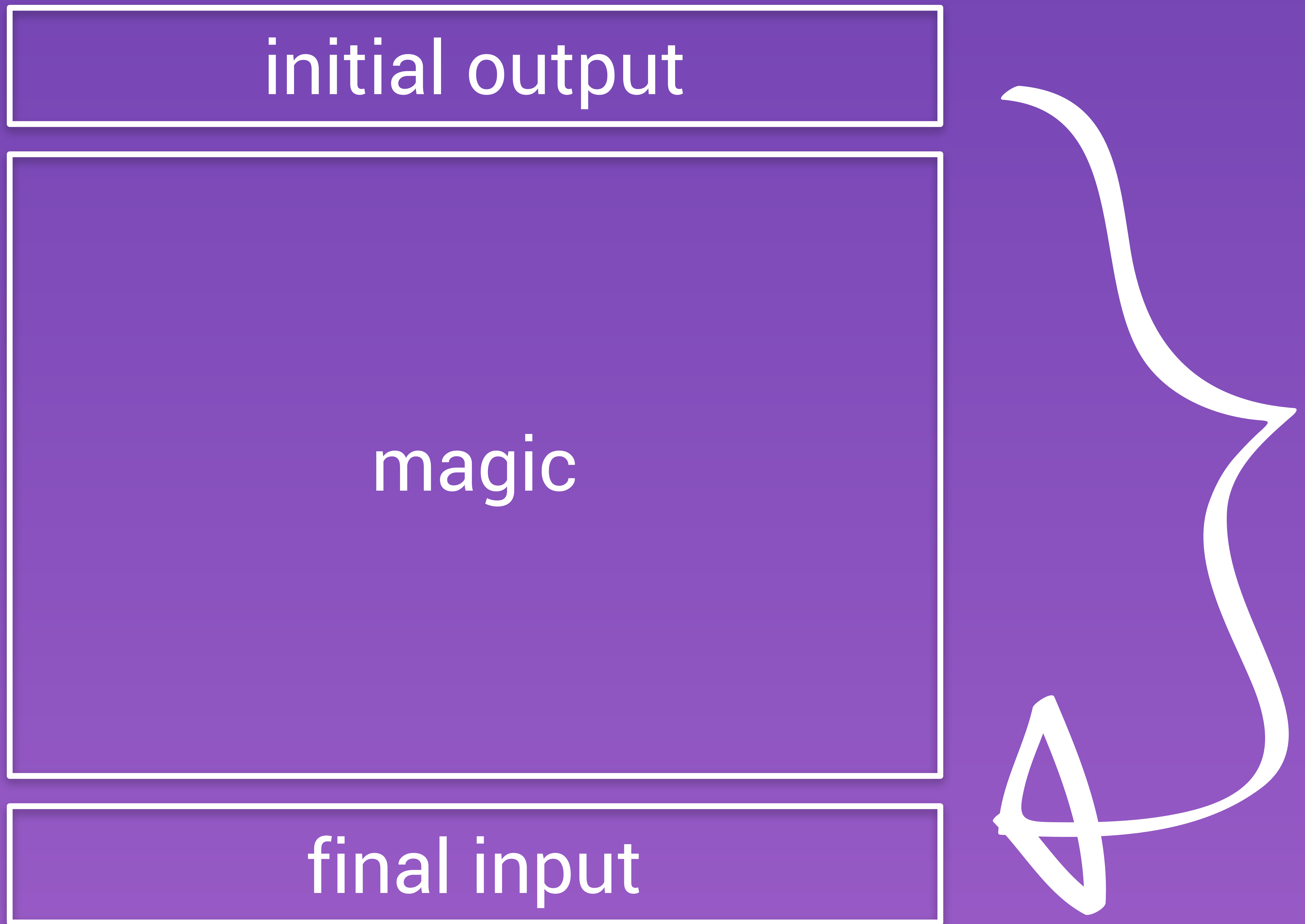
	SINGLE	MULTIPLE
PULL	Function	Enumerable
PUSH	Promise	**Observable**

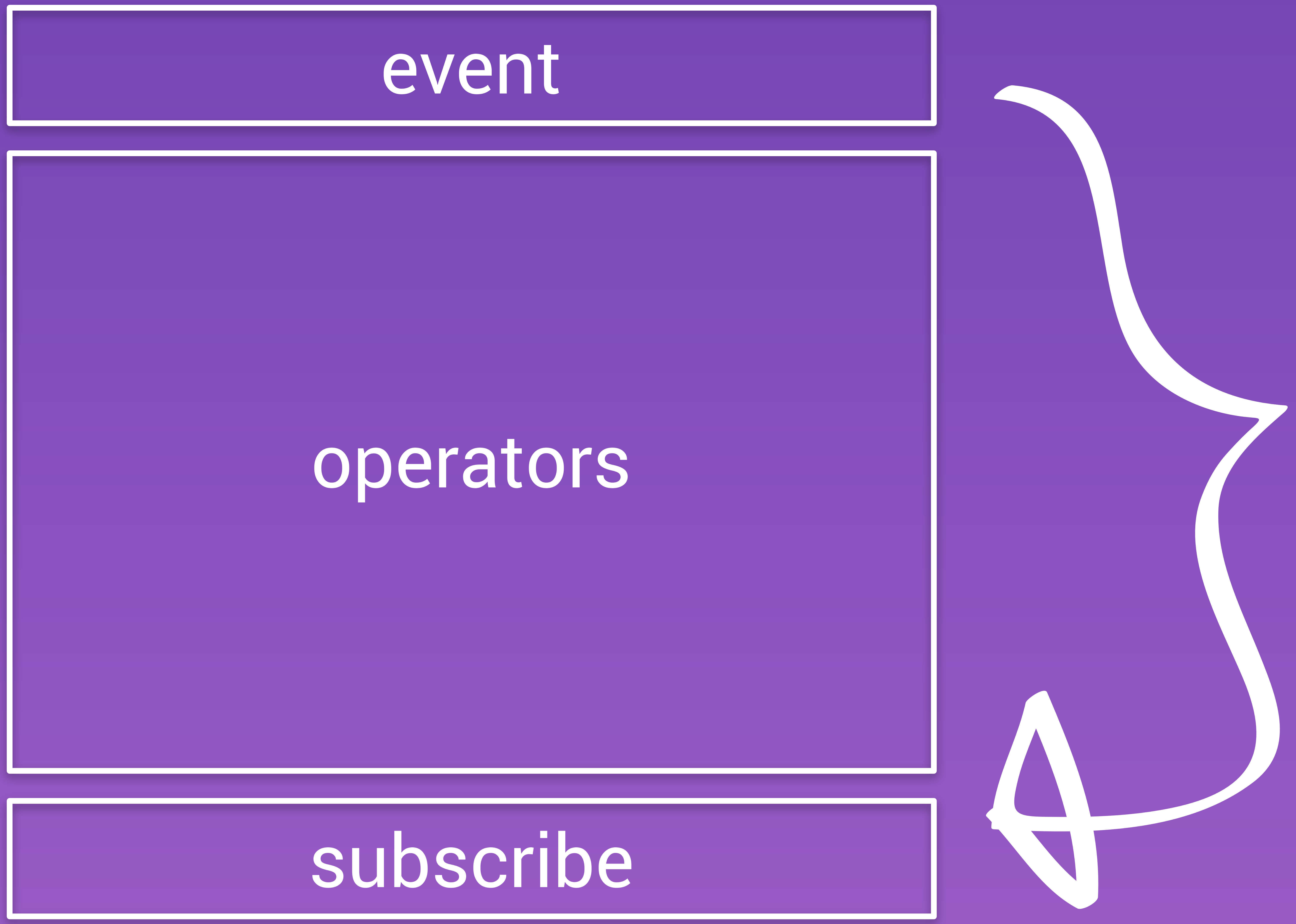
Value consumption


input
output


output
input

The Basic **Sequence**


initial output
magic
final input


event
operators
subscribe

```
@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

- Convert the http calls in the **widgets** service to use **Observable.subscribe**
- Use **Observable.map** to map the response to something the **widgets** component can understand
- Use **Observable.filter** to filter out **widgets** that do not match some criteria
- Use **Observable.map** to perform some additional data transformation to the widgets collection

@simpulton

Thanks!