Implementing inter-component communications in Angular 4

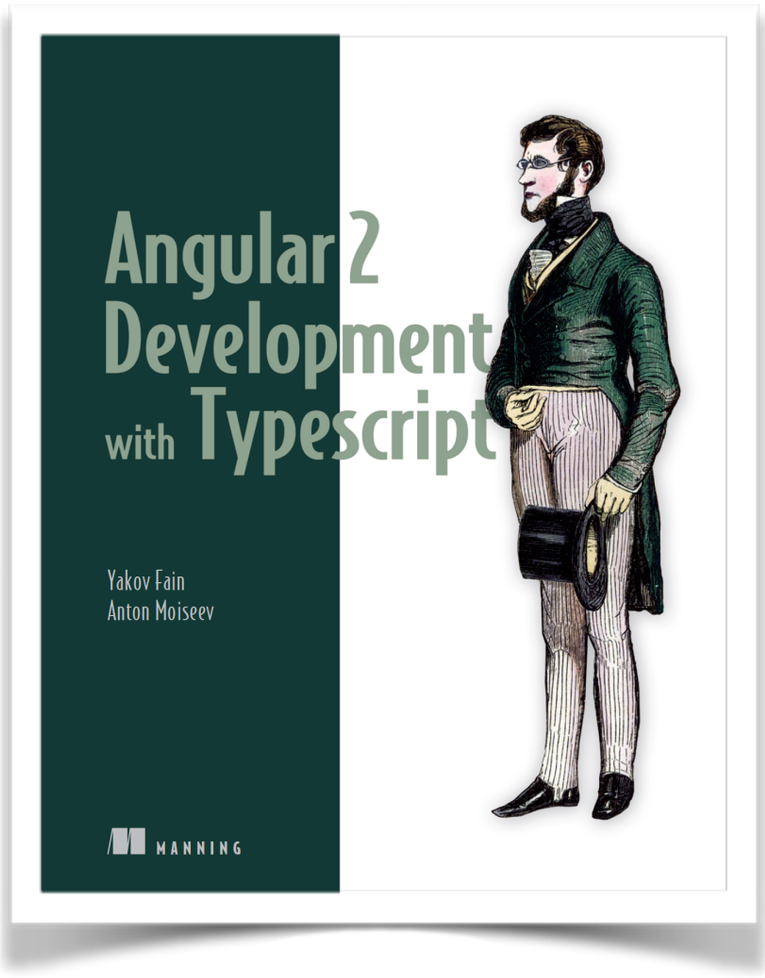
@yfain

About myself

- Work for Farata Systems
  Angular practice lead

- Java Champion

- Latest book:
  “Angular Development with TypeScript”



In this unit

- @Input and @Output properties
- Using a parent component as a mediator
- Using an injectable service as a mediator
- Projection of HTML
- Component lifecycle

Input and Output Properties

- Think of a component as a black box with entry and exit doors

- Properties marked as **`@Input()`** are used for getting data from the parent component

- The parent component can pass data to its child using bindings to input properties

- Properties marked as **`@Output()`** are used for sending events (and data) from a component

Parent binds to input props

```
@Component({
    selector: 'app',
    template: `
    <input type="text" placeholder="Enter stock (e.g. AAPL)"  (change)="onInputEvent($event)">
    <br/>


    <order-processor [stockSymbol]="stock" quantity="100">
    </order-processor>
  `
})
class AppComponent {
    stock:string;

    onInputEvent({target}):void{
        this.stock=target.value;
    }
}
```

Binding

No binding

Input Properties in Child

```
@Component({
    selector: 'order-processor',
    template: `
    Buying {{quantity}} shares of {{stockSymbol}}
    `})
class OrderComponent {

    @Input() quantity: number;

    private _stockSymbol: string;

    @Input()
    set stockSymbol(value: string) {
        this._stockSymbol = value;
        if (this._stockSymbol != undefined) {
          console.log(`Sending a Buy order to NASDAQ: ${this.stockSymbol} ${this.quantity}`);
        }
    }

    get stockSymbol(): string {
        return this._stockSymbol;
    }
}
```

Demo: @Input()

ng serve --app input -o

Output properties in a child

```
@Component({
    selector: 'price-quoter',
    template: `<strong>Inside PriceQuoterComponent:
                        {{stockSymbol}} {{price | currency:'USD':true}}</strong>`,

    styles:[`:host {background: pink;}`]
})
class PriceQuoterComponent {

    @Output() lastPrice: EventEmitter <IPriceQuote> = new EventEmitter();

    stockSymbol: string = "IBM";
    price:number;

    constructor() {
        setInterval(() => {
            let priceQuote: IPriceQuote = {
                stockSymbol: this.stockSymbol,

                lastPrice: 100*Math.random()
            };

            this.price = priceQuote.lastPrice;

            this.lastPrice.emit(priceQuote);

        }, 1000);
    }
}
```

pipe

```
interface IPriceQuote {
    stockSymbol: string;
    lastPrice: number;
}
```

A child emits events via output properties

The parent listens to the lastPrice event

```
@Component({
    selector: 'app',
    template: `
      <price-quoter (lastPrice)="priceQuoteHandler($event)"></price-quoter><br>

      AppComponent received: {{stockSymbol}} {{price | currency:'USD':true}}
    `
})
class AppComponent {

    stockSymbol: string;
    price:number;

    priceQuoteHandler(event:IPriceQuote) {
        this.stockSymbol = event.stockSymbol;
        this.price = event.lastPrice;
    }
}
```

Demo: @Output()

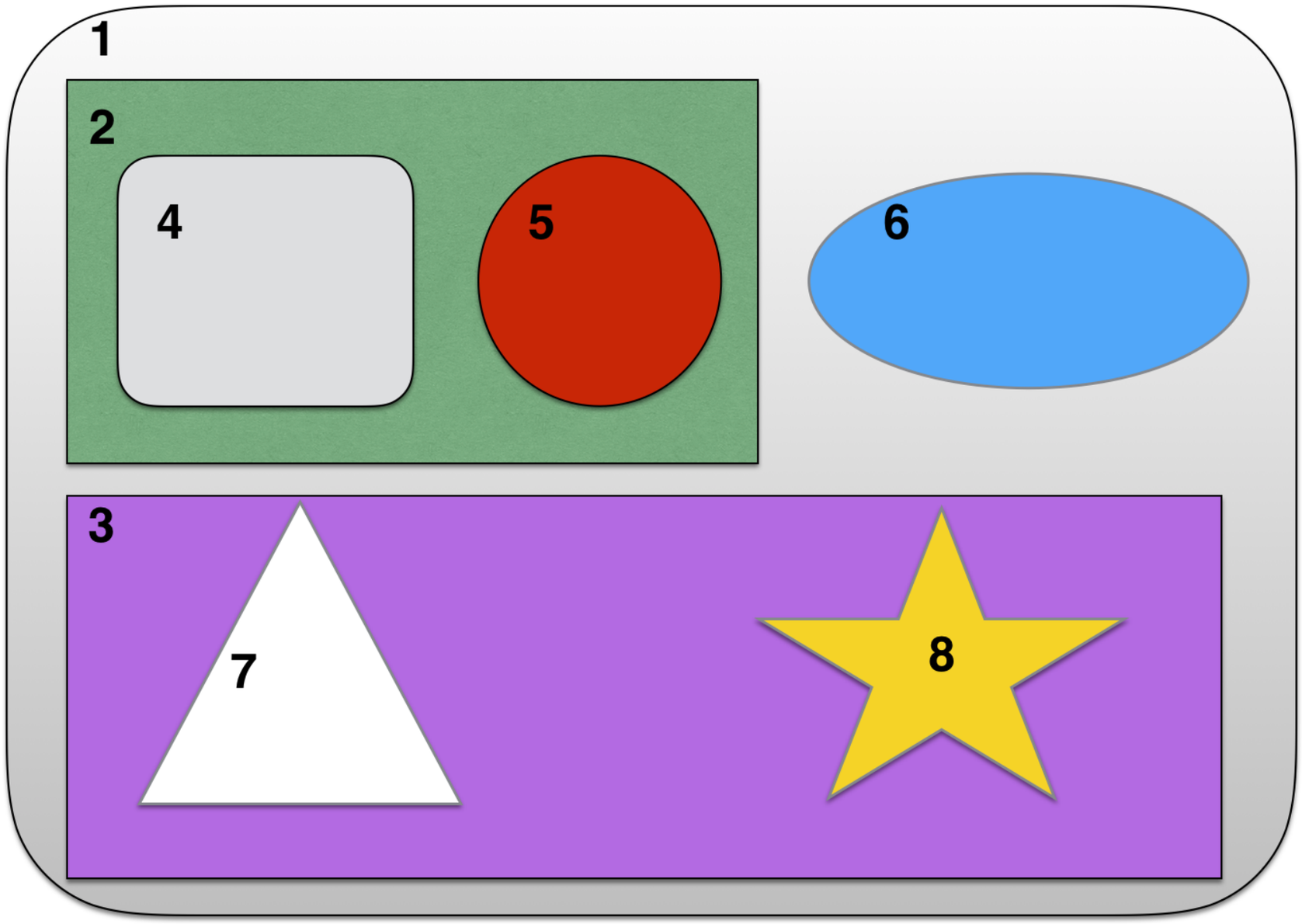
ng serve --app output -o

Implementing the Mediator Design Pattern

Loosely-Coupled Components

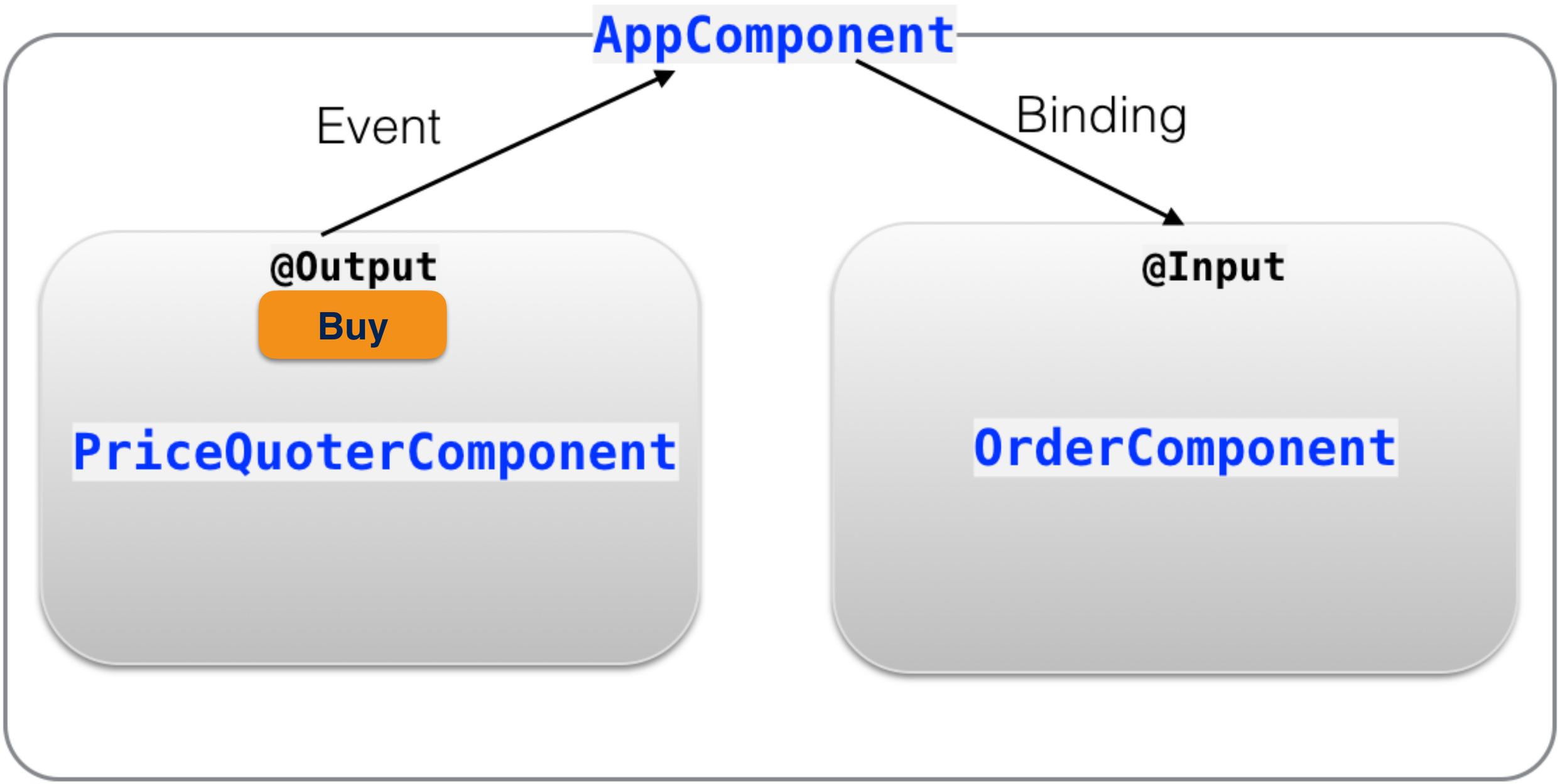
- The Mediator acts as a man in the middle
- A component A sends the data to a mediator, which passes the data to a component B
- Component A and Component B don't know about each other
- **A parent component** can mediate siblings' communications
- **An injectable service** can mediate communication between any components (an IoC design pattern)

Parent components as mediators



QUIZ: How the number 7 can send the data to number 6?

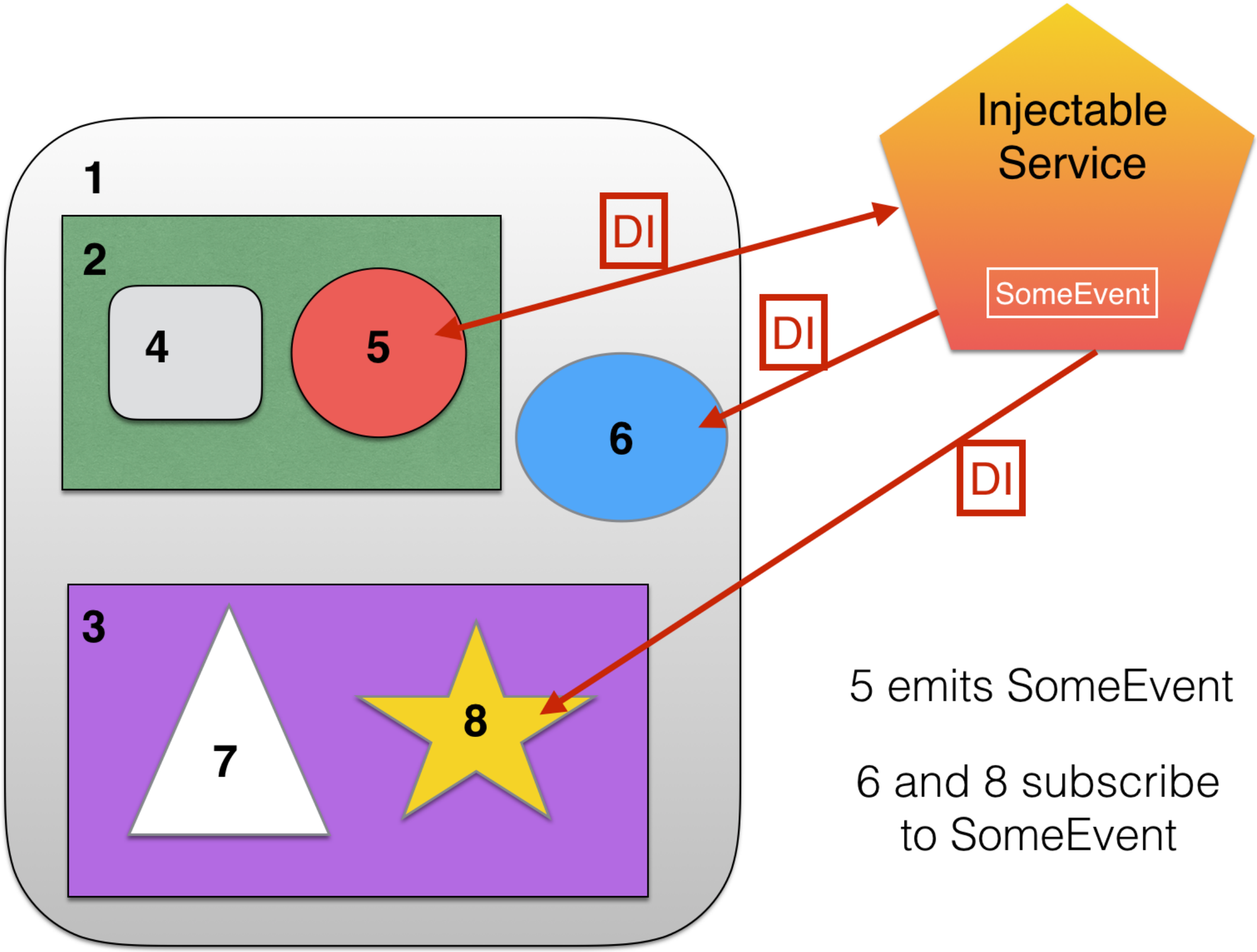
Parent as a mediator



Demo: Mediator parent

```
ng serve --app mediator1 -o
```

Injectable service as a mediator



5 emits SomeEvent

6 and 8 subscribe
to SomeEvent

Subscribing to EventEmitter

`EventEmitter` extends the RxJS `Subject` and includes `emit()` and `subscribe()` methods.

```
myEvent: EventEmitter<string> = new EventEmitter();

myEvent.emit(“Hello World”);

…

myEvent.subscribe(event => console.log(”Received " + event);
```

Demo

ng serve --app mediator2 -o

Projecting HTML fragments to the templates of child components

Projection

- Allows to change a template's content at runtime
- You can pass one or more HTML fragments to a child component
- If a child's template includes **<ng-content>**, it will be replaced by an HTML fragment given by the parent
- In AngularJS it was known as transclusion

Basic projection

Child

```
@Component({
  selector: 'child',
  styles: ['.wrapper {background: lightgreen;}'],
  template: `
    <div class="wrapper">
     <h2>Child</h2>
      <div>This div is defined in the child's template</div>

     <ng-content></ng-content> <---- insertion point

    </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
class ChildComponent {}
```

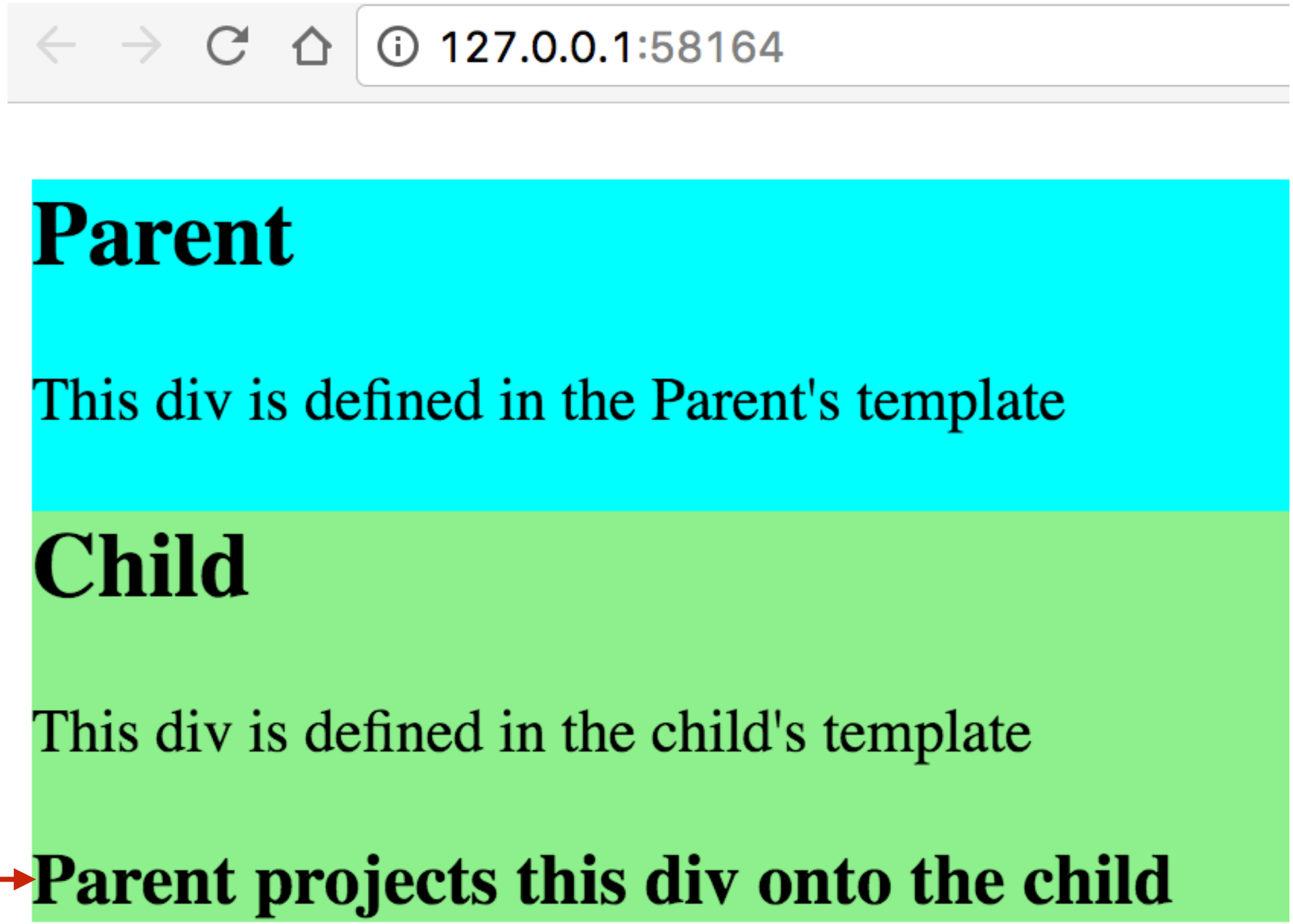
Parent

```
@Component({
  selector: 'app',
  styles: ['.wrapper {background: cyan;}'],
  template: `
    <div class="wrapper">
     <h2>Parent</h2>
      <div>This div is defined in the Parent's template</div>
      <child>

        <div><h3>Parent projects this div onto the child </h3></div>

      </child>
    </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
class AppComponent {}
```

Projecting a from parent to child



Projecting into multiple areas

- A component can have more than one <ng-content> tag in its template
- The attribute `select` allows to distinguish <ng-content> areas

```
<ng-content select=".header" ></ng-content><p>

<div>This content is defined in child</div><p>

<ng-content select=".footer"></ng-content>
```

Parent

```
@Component({
  selector: 'app',
  styles: ['.wrapper {background: cyan;}'],
  template: `
    <div class="app">
     <h2>Parent</h2>
      <div>This div is defined in the Parent's template</div>
      <child>

        <div class="header" ><i>Child got this header from parent {{todaysDate}}</i></div>

        <div class="footer"><i>Child got this footer from parent</i></div>
      </child>
    </div>
  `
})
class AppComponent {
  todaysDate: string = new Date().toLocaleDateString();
}
```

Child

```
@Component({
  selector: 'child',
  styles: ['.wrapper {background: lightgreen;}'],
  template: `
    <div class="child">
     <h2>Child</h2>
      <ng-content select=".header"></ng-content><p>
      <div>This content is defined in child</div><p>
      <ng-content select=".footer"></ng-content>
    </div>
  `
})
class ChildComponent {}
```

Shadow DOM Support

- Every Web page is represented by a tree of DOM objects
- Shadow DOM allows to encapsulate a subtree of the HTML elements to create a boundary between Web components
- Such subtree is rendered as a part of the HTML document, but its elements are not attached to the main DOM tree
- With Shadow DOM the CSS styles of Web components won’t be merged with the main DOM CSS

encapsulation property of @Component

```
encapsulation: ViewEncapsulation.None
```

- `ViewEncapsulation.Emulated` - Emulate encapsulation of the Shadow DOM (default; may be deprecated)

- `ViewEncapsulation.Native` - Use the Shadow DOM natively supported by the browser

- `ViewEncapsulation.None` - Don’t use the Shadow DOM encapsulation

ViewEncapsulation.Native

Parent

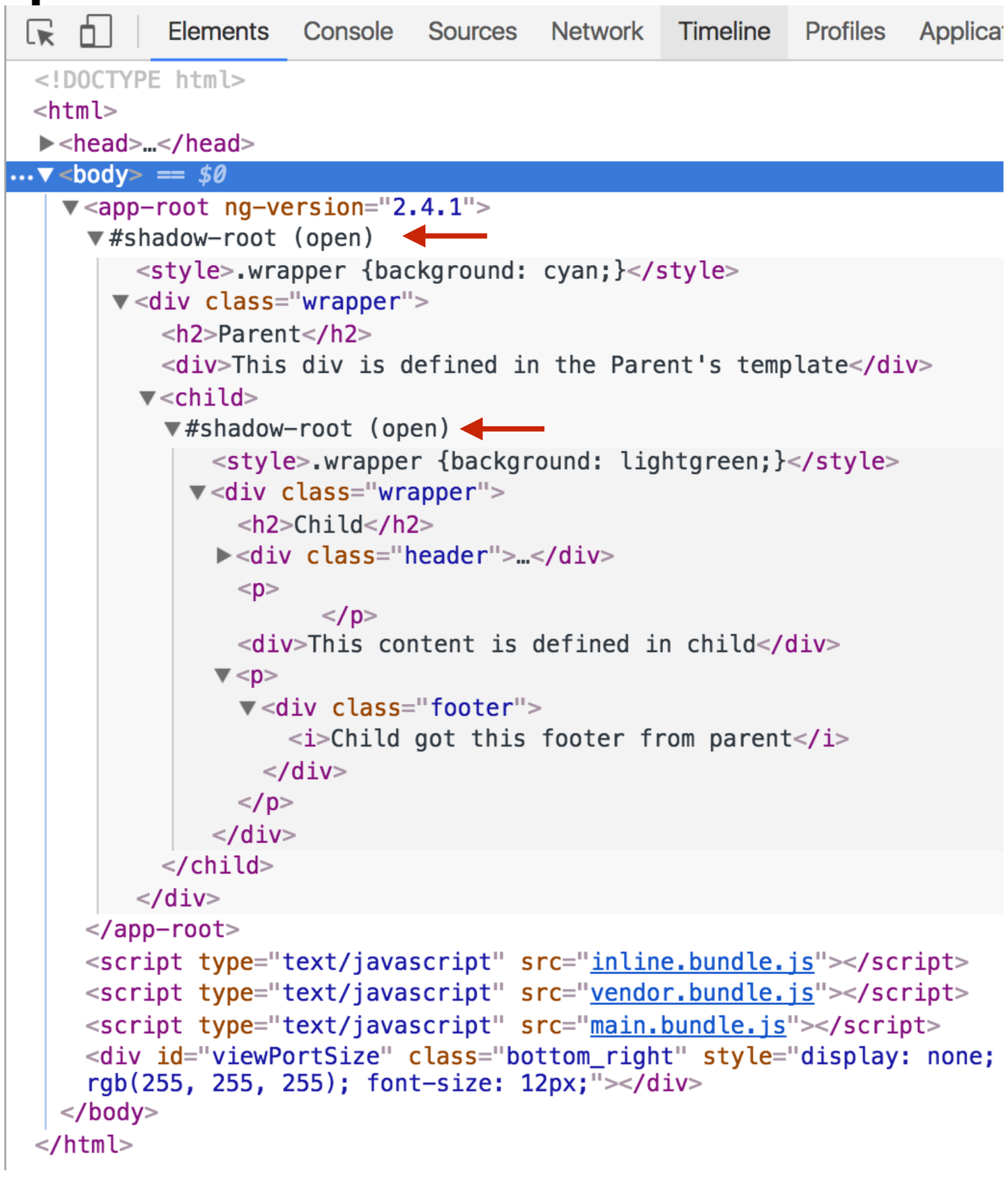
This div is defined in the Parent's template

Child

Child got this header from parent 1/3/2017

This content is defined in child

Child got this footer from parent



ViewEncapsulation.Emulated

Parent

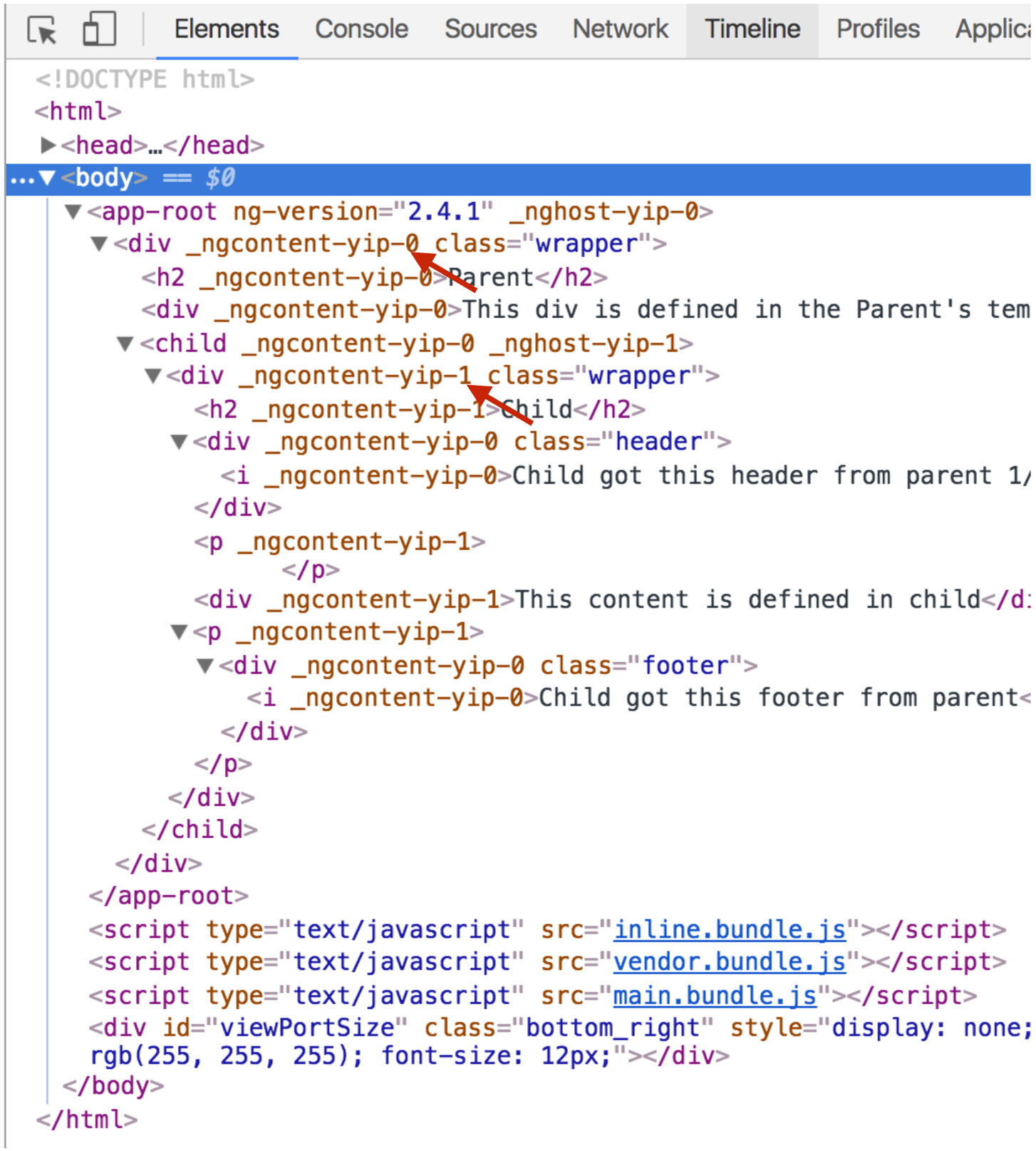
This div is defined in the Parent's template

Child

Child got this header from parent 1/3/2017

This content is defined in child

Child got this footer from parent



ViewEncapsulation.None

Parent

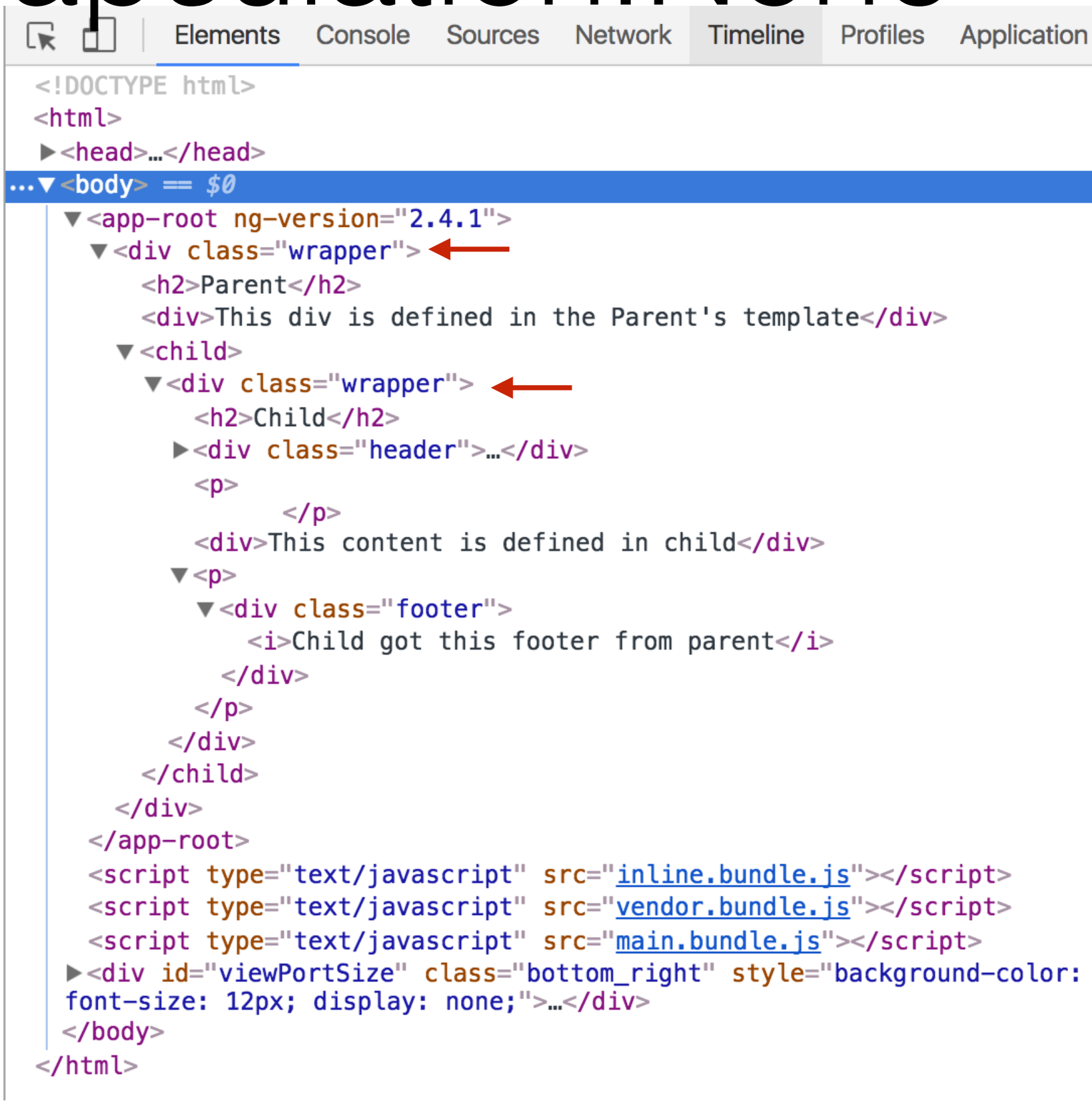
This div is defined in the Parent's template

Child

Child got this header from parent 1/3/2017

This content is defined in child

Child got this footer from parent



Demo: Projection

ng serve --app projection -o

Exposing child API with @ViewChild

A parent component can use the child's API from both the template and the TypeScript code.

```
@Component({
    selector: 'app',
    template: `
    <h1>Parent</h1>
    <child #child1></child>

    <child #child2></child>

    <button (click)="child2.greet('Child 2')">Invoke greet() on child 2</button>
  `
})
class AppComponent implements AfterViewInit {
    @ViewChild('child1')
    firstChild: ChildComponent;

    ngAfterViewInit() {
        this.firstChild.greet('Child 1');
    }
}
```

Demo

ng serve --app childapi -o

Passing params to routes

Receiving params in ActivatedRoute

- Inject `ActivatedRoute` into a component to receive the route params during navigation

- Use `ActivatedRoute.snapshot` to get params once

- Use `ActivatedRoute.param.subscribe()` for receiving multiple params over time

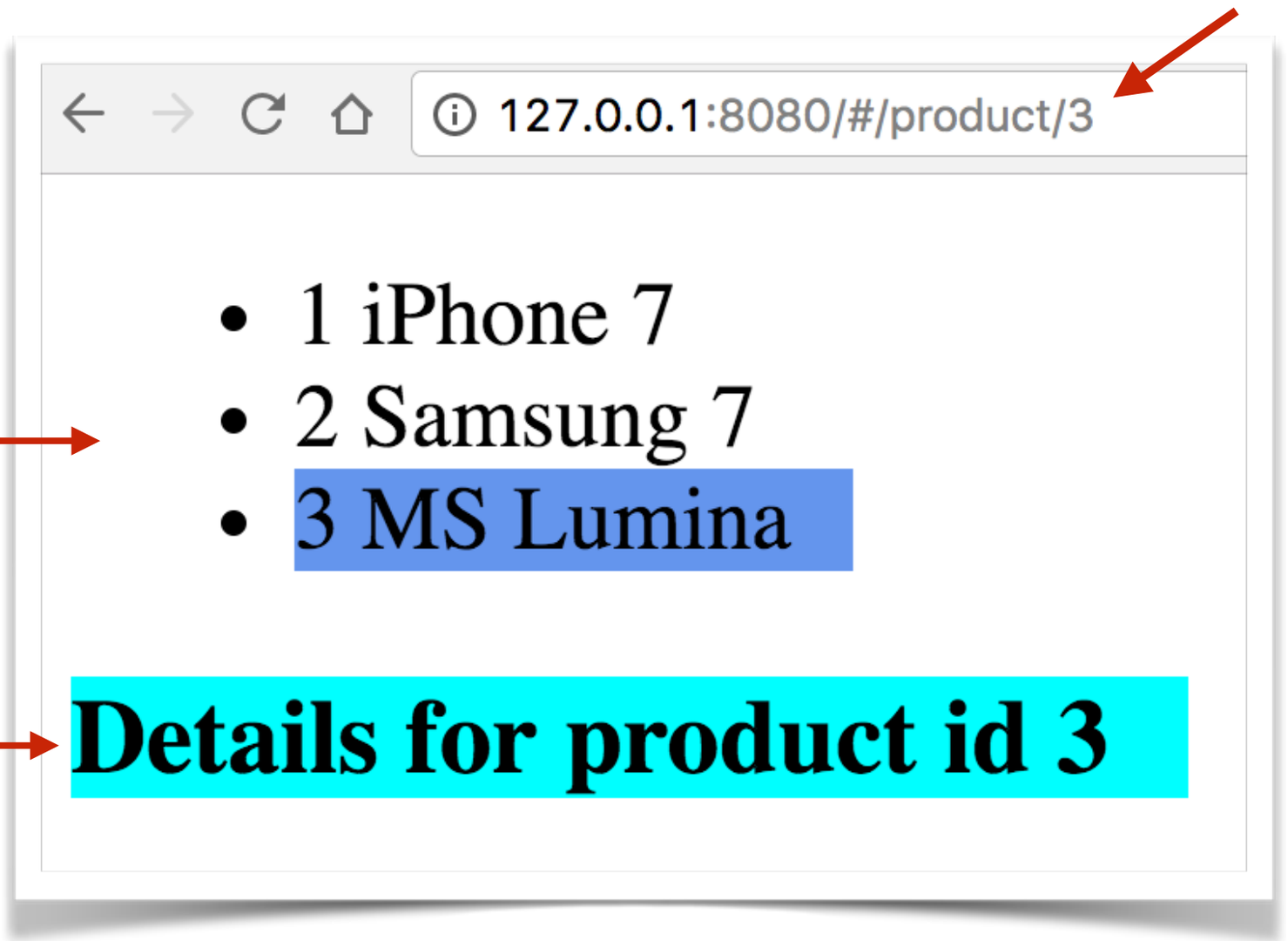
ActivatedRoute (fragment)

```
export declare class ActivatedRoute {
    /** An observable of the URL segments matched by this route */
    url: Observable<UrlSegment[]>;
    /** An observable of the matrix parameters scoped to this route */
    params: Observable<Params>;
    /** An observable of the query parameters shared by all the routes */
    queryParams: Observable<Params>;
    /** An observable of the URL fragment shared by all the routes */
    fragment: Observable<string>;
    /** An observable of the static and resolved data of this route. */
    data: Observable<Data>;
    /** The outlet name of the route. It's a constant */
    readonly paramMap: Observable<ParamMap>;
    readonly queryParamMap: Observable<ParamMap>;
    toString(): string;
    ...
}
```

Master-Detail with Router

Parent component

<router-outlet>



```
const routes: Routes = [
    {path: 'productDetail/:id', component: ProductDetailComponentParam}
];
…

class AppComponent {
  …
    constructor(private _router: Router){}

    onSelect(prod: Product): void {
         this._router.navigate(["/productDetail", prod.id]);
    }
}
```

```
export class ProductDetailComponentParam {
  productID: number;

  constructor(private route: ActivatedRoute) {

  this.route.paramMap
    .subscribe(
    params => this.productID = params.get('id')
  );
}

}
```

Demo

ng serve --app router -o

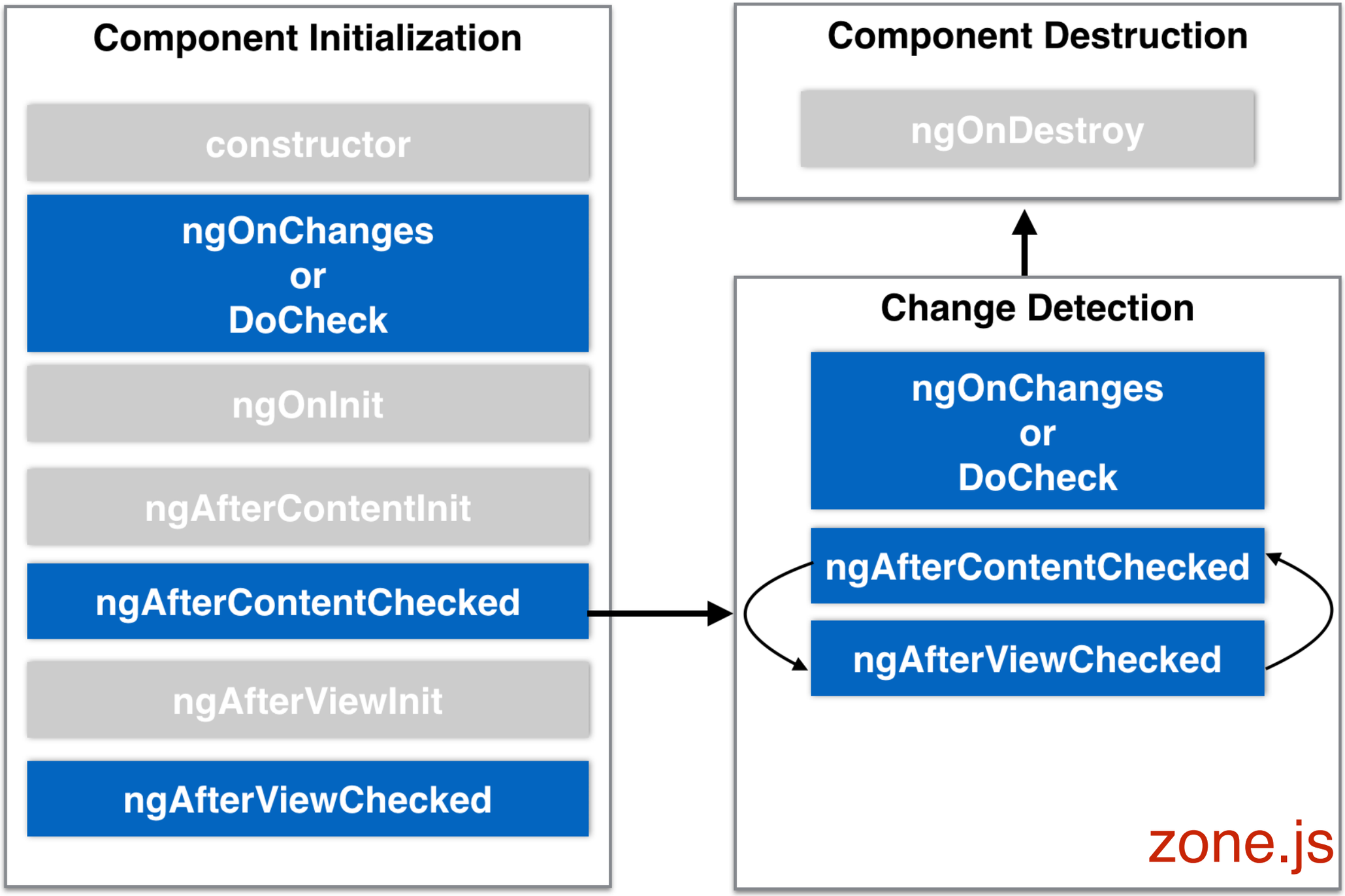
Component Lifecycle

Documentation: https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

Component's Lifecycle

- Component gets created and initialized
- Change-detection mechanism starts monitoring the component
- You can intercept component lifecycle's key moments by implementing lifecycle hook interfaces
- Angular calls the hooks only if you implemented them
- Component gets destroyed

Lifecycle hooks



Implementing lifecycle hooks

`OnInit, OnChanges, AfterViewInit, AfterViewChecked etc.`

```
@Component({
  selector: 'my-component',
  template: '…'
})
class MyComponent implements OnInit, OnChanges {

…

ngOnInit() {
    console.log(`In ngOnInit`);
  }

 ngOnChanges() {
    console.log(`In ngOnChanges`);
  }


}
```

ngOnInit()

By the time it's called, component properties are initialized

Often used for fetching data. Place the code that uses component's properties in `ngOnInit()`.

```
@Input() productId: number;

constructor(productService: ProductService) {}

ngOnInit(){
  this.product = this.productService.getProductById(this.productId);
}
```

Parent changes input properties of child

Parent

```
@Component({
  selector: 'app',
  styles: ['.parent {background: lightblue}'],
  template: `
     <div class="parent">
       <h2>Parent</h2>
      <div>Greeting: <input type="text" [value]="greeting" (change)="greeting = $event.target.value"></div>
      <div>User name: <input type="text" [value]="user.name" (change)="user.name = $event.target.value"></div>
      <child [greeting]="greeting" [user]="user"></child>
    </div>
  `
})
class AppComponent {
  greeting: string = 'Hello';
  user: {name: string} = {name: 'John'};
}
```

ngOnChanges() in child intercept input properties changes

Parent

```
@Component({
  selector: 'app',
  styles: ['.parent {background: lightblue}'],
  template: `
     <div class="parent">
       <h2>Parent</h2>
      <div>Greeting: <input type="text" [value]="greeting" (change)="greeting = $event.target.value"></div>
      <div>User name: <input type="text" [value]="user.name" (change)="user.name = $event.target.value"></div>
      <child [greeting]="greeting" [user]="user"></child>
    </div>
  `
})
class AppComponent {
  greeting: string = 'Hello';
  user: {name: string} = {name: 'John'};
}
```

Child

```
@Component({
  selector: 'child',
  styles: ['.child{background:lightgreen}'],
  template: `
     <div class="child">
      <h2>Child</h2>
      <div>Greeting: {{greeting}}</div>
      <div>User name: {{user.name}}</div>
      <div>Message: <input [(ngModel)]="message"></div>
    </div>
  `
})
class ChildComponent implements OnChanges {
  @Input() greeting: string;
  @Input() user: {name: string};
  message: string = 'Initial message';

  ngOnChanges(changes: IChanges) {
    console.log(JSON.stringify(changes, null, 2));
  }
}
```

Mutable vs Immutable

- JavaScript primitives are immutable

```
var greeting = "Hello";   // greeting's address is @287652
greeting = "Hello Mary";  // greeting's address is @287774
```

- JavaScript objects are mutable

```
var user = {name: "John"};  // user's address is @287000
user.name = "Mary";         // user's address is @287000
```

Demo: Component Lifecycle

```
ng serve --app lifecycle -o
```

What have we learned

- Component receive data from their parents via `@Input` properties
- Components send data to their parents by emitting events via `@Output` properties
- Components shouldn't directly reach out to the internals of other components
- The mediator design pattern is used for arranging inter-component communication in a loosely-coupled manner
- Using injectable services offers the most flexible way of inter-component communications
- Angular offers the Shadow DOM support for all browsers
- Angular offers a number of component lifecycle hooks where you can place code that will be invoked at specific moment of the component life span

Thank you!

- Code samples:
  http://bit.ly/2p2Ol5E

- Training inquiries:
  training@faratasystems.com

- My blog:
  yakovfain.com

- Twitter: @yfain