

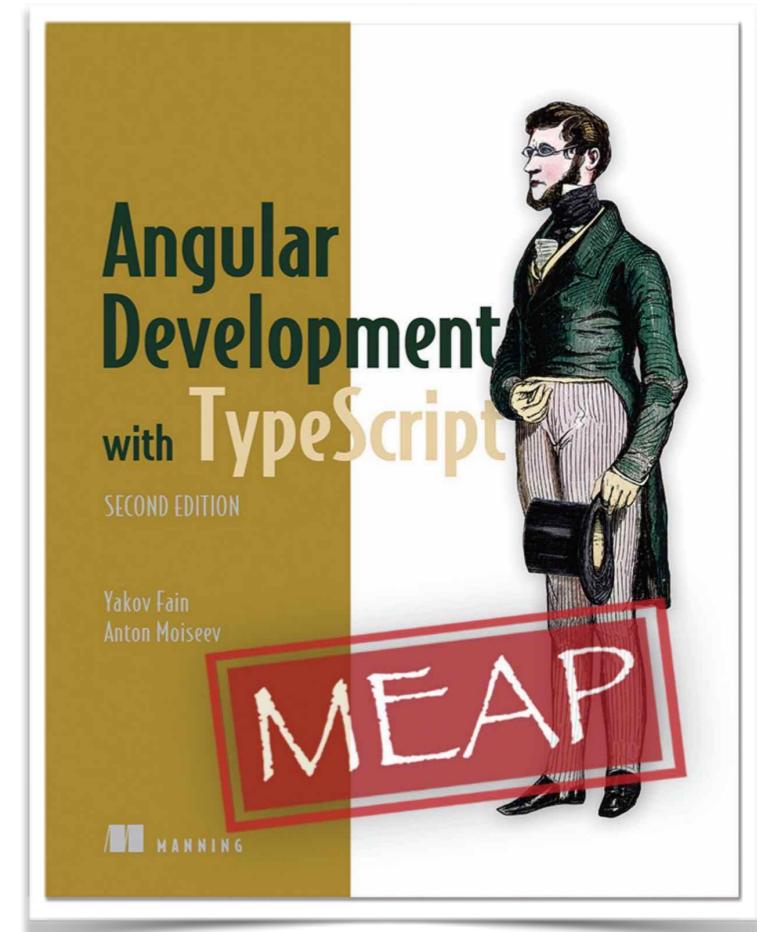
Angular Development with TypeScript

Unit 1: First steps with Angular

Angular docs: <https://angular.io>

About myself

- Work for Farata Systems
- Angular consulting and training
- Java Champion
- Co-authored two editions of the book
“Angular Development with TypeScript”
- Working on a book
“Get Programming with TypeScript”



@yfain

Pre-requisites

- Download and install the current version of Node.js from <https://nodejs.org>
- Install one of these IDEs:
 - Visual Studio Code (<https://code.visualstudio.com>)
 - WebStorm (<https://www.jetbrains.com/webstorm>).
- If you have Angular CLI older than 6.0.0, uninstall it by running the following command:

```
npm uninstall @angular/cli -g
```

- Install the latest Angular CLI:

```
npm install @angular/cli -g
```

- Check that you have Angular CLI 6 installed by running the following command:

```
ng -v
```

Code samples

- Download and unzip the workshop handouts from <https://goo.gl/W3zkbR>
- Install the software listed in the prerequisites slide

In this unit

- Angular modules and components
- npm package manager
- Getting started with Angular CLI
- Intro to TypeScript
- Intro to templates and bindings

Angular

- Component-based framework
- Declarative templates
- Dependency Injection
- Single-pass change detection
- Router
- Integrated RxJS library

Main artifacts

- **Component** - a class that includes UI (a template)
- **Directive** - a class with the code to perform actions on components but has no UI of its own
- **Service** - a class where you put business logic of your app
- **Pipe** - a transformer function that can be used in templates
- **Module** - a class that lists components, directives, service providers, other modules, pipes.

The landing page of an Auction app

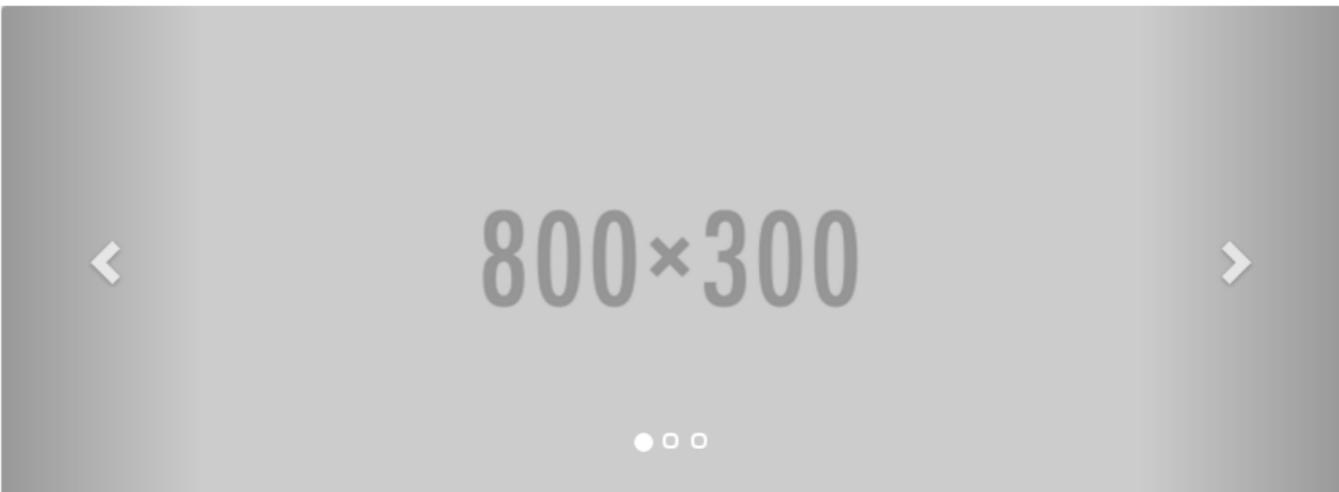
Online Auction About Services Contact

Product title:

Product price:

Product category:

Search



320×150

First Product 24.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 4.3 stars

320×150

Second Product 64.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 3.5 stars

320×150

Third Product 74.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 4.2 stars

320×150

Fourth Product 84.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 3.9 stars

320×150

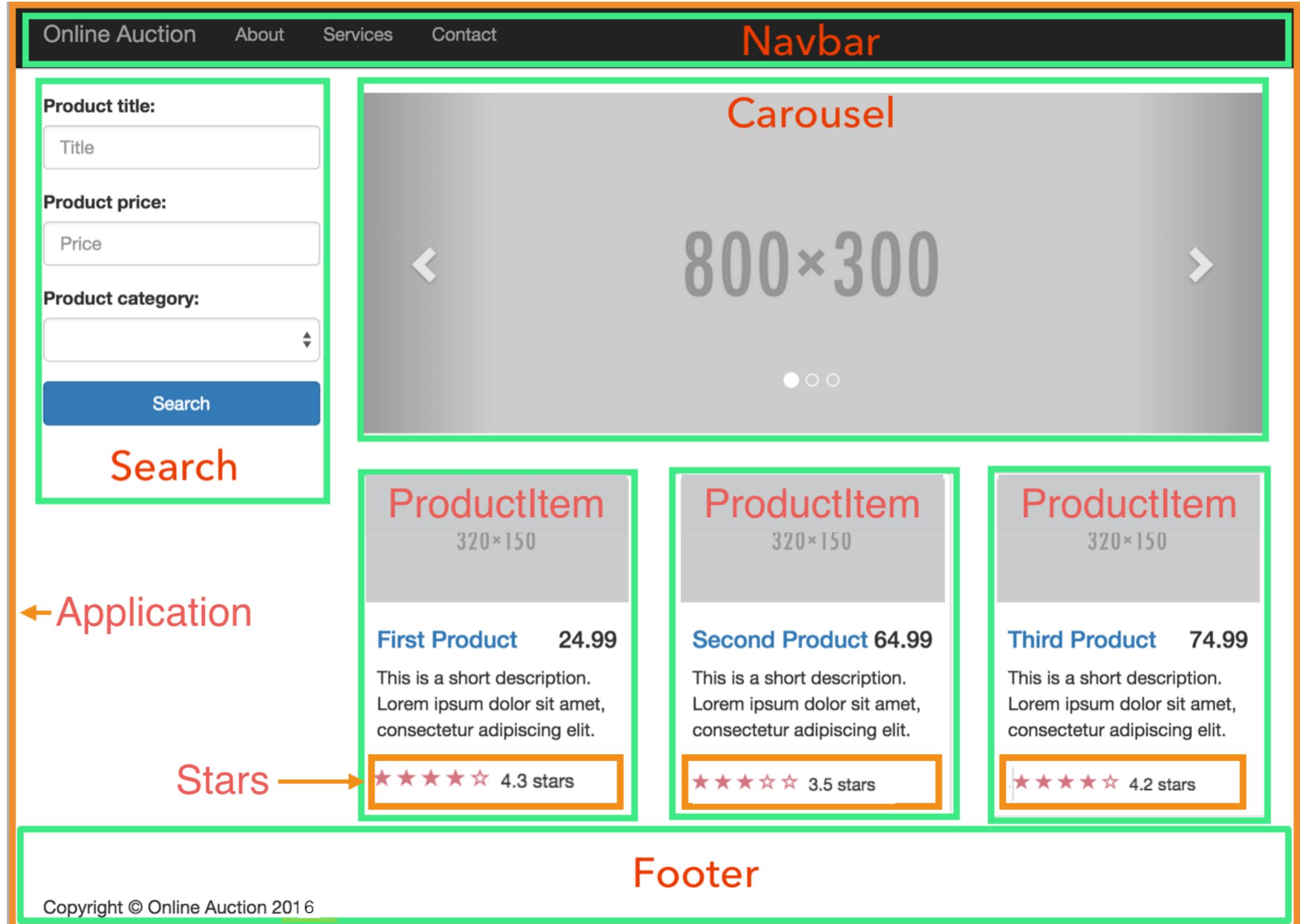
Fifth Product 94.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 5 stars

320×150

Sixth Product 54.99
This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
 4.6 stars

Copyright © Online Auction 2016

An app is a tree of components packaged as one or more modules



Navbar

Carousel

800×300

Product title:

Product price:

Product category:

Search

Product

320×150

First Product 24.99

This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.

4.3 stars

Product

320×150

Second Product 64.99

This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.

3.5 stars

Product

320×150

Third Product 74.99

This is a short description.
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.

4.2 stars

Footer

Copyright © Online Auction 2015

<auction-navbar></auction-navbar>

<div class="container">

<div class="row">

<div class="col-md-3">

<auction-search></auction-search>

</div>

<div class="col-md-9">

<router-outlet></router-outlet>

</div>

</div>

</div>

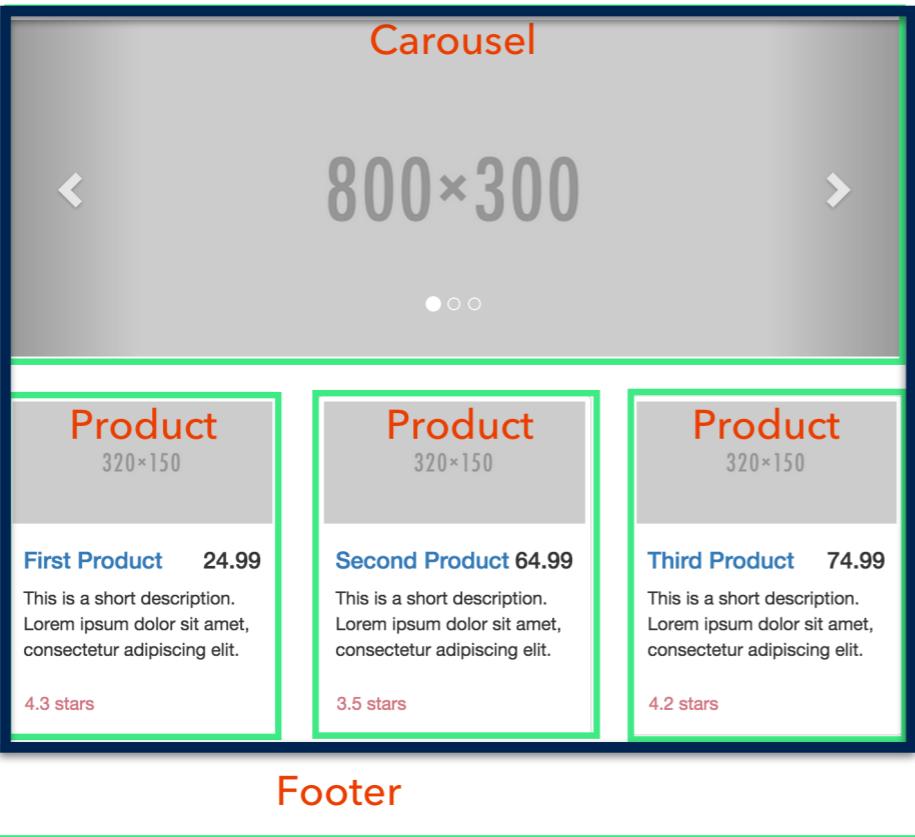
<auction-footer></auction-footer>

template

Product title:

Product price:

Product category:



```
<auction-navbar></auction-navbar>
<div class="container">
  <div class="row">
    <div class="col-md-3">
      <auction-search></auction-search>
    </div>

    <div class="col-md-9">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>

<auction-footer></auction-footer>
```

```
import {Component} from '@angular/core';
import {Product, ProductService} from './product.service';

@Component({
  selector: 'app-root',
  templateUrl: 'application.html',
  styleUrls: ['application.css']
})
export class AppComponent {
  products: Array<Product> = [];

  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }
}
```

template, CSS

TypeScript

An Angular Module

- A module is class annotated with the decorator `@NgModule()`
- Lists all components, services, routes, et al.
- An app has at least one root module
- A typical app has one root and one or more feature modules

Angular modules doc: <http://bit.ly/2cH2kay>

Sample app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

ES6 modules

other modules

List of module components

Root component

```
graph LR; A[ES6 modules] --> B[import statements]; C[other modules] --> D[imports]; E[List of module components] --> F[declarations]; G[Root component] --> H[bootstrap];
```

Installing Angular and dependencies

npm: Node Package Manager

- Install the latest version of Node.js from <https://nodejs.org>
- npm comes with Node.js
- The npmjs.org repository has 400K+ packages
- All packages required by Angular apps are installed from npmjs.org

Semantic Versioning

6.0.3

Major
breaking
changes

Minor
new features,
not breaking

Patch
bug fixes,
not breaking

Installing packages with npm

- Project dependencies are configured in the file `package.json` and are installed locally in the directory `node_modules`
- Installing the package xyz in your project's dir `node_modules`:
`npm install xyz`
- Installing the package xyz globally:
`npm install xyz -g`

package.json

Required for dev and prod

```
"dependencies": {  
    "@angular/animations": "^6.0.0",  
    "@angular/common": "^6.0.0",  
    "@angular/compiler": "^6.0.0",  
    "@angular/core": "^6.0.0",  
    "@angular/forms": "^6.0.0",  
    "@angular/http": "^6.0.0",  
    "@angular/platform-browser": "^6.0.0",  
    "@angular/platform-browser-dynamic": "^6.0.0",  
    "@angular/router": "^6.0.0",  
    "core-js": "^2.5.4",  
    "rxjs": "^6.0.0",  
    "zone.js": "^0.8.26"  
},
```

```
"devDependencies": {
```

```
    "@angular/compiler-cli": "^6.0.0",  
    "@angular-devkit/build-angular": "~0.6.0",  
    "typescript": "~2.7.2",  
    "@angular/cli": "~6.0.0",  
    "@angular/language-service": "^6.0.0",  
    "@types/jasmine": "~2.8.6",  
    "@types/jasminewd2": "~2.0.3",  
    "@types/node": "~8.9.4",  
    "codelyzer": "~4.2.1",  
    "jasmine-core": "~2.99.1",  
    "jasmine-spec-reporter": "~4.2.1",  
    "karma": "~1.7.1",  
    "karma-chrome-launcher": "~2.2.0",  
    "karma-coverage-istanbul-reporter": "~1.4.2",  
    "karma-jasmine": "~1.1.1",  
    "karma-jasmine-html-reporter": "^0.2.2",  
    "protractor": "~5.3.0",  
    "ts-node": "~5.0.1",  
    "tslint": "~5.9.1"  
}
```

Required for dev

Intro to Angular CLI

Docs: <https://github.com/angular/angular-cli/wiki>

Angular CLI features

- Scaffolding the project and creating a basic app
- Generating components, services, modules, etc.
- Dev web server
- Supports dev and prod builds
- Configuring test runners

Getting started with Angular CLI

- Install Angular CLI globally

```
npm i @angular/cli -g
```

- Generate new project using the command `ng new`

Walkthrough 1.1

1. Install Angular CLI using the command window:

```
npm i @angular/cli -g
```

2. Generate a new Angular project **hello-cli**:

```
ng new hello-cli
```

3. **cd hello-cli**

4. Build the app bundles and start the app

```
ng serve -o
```

Some ng new options

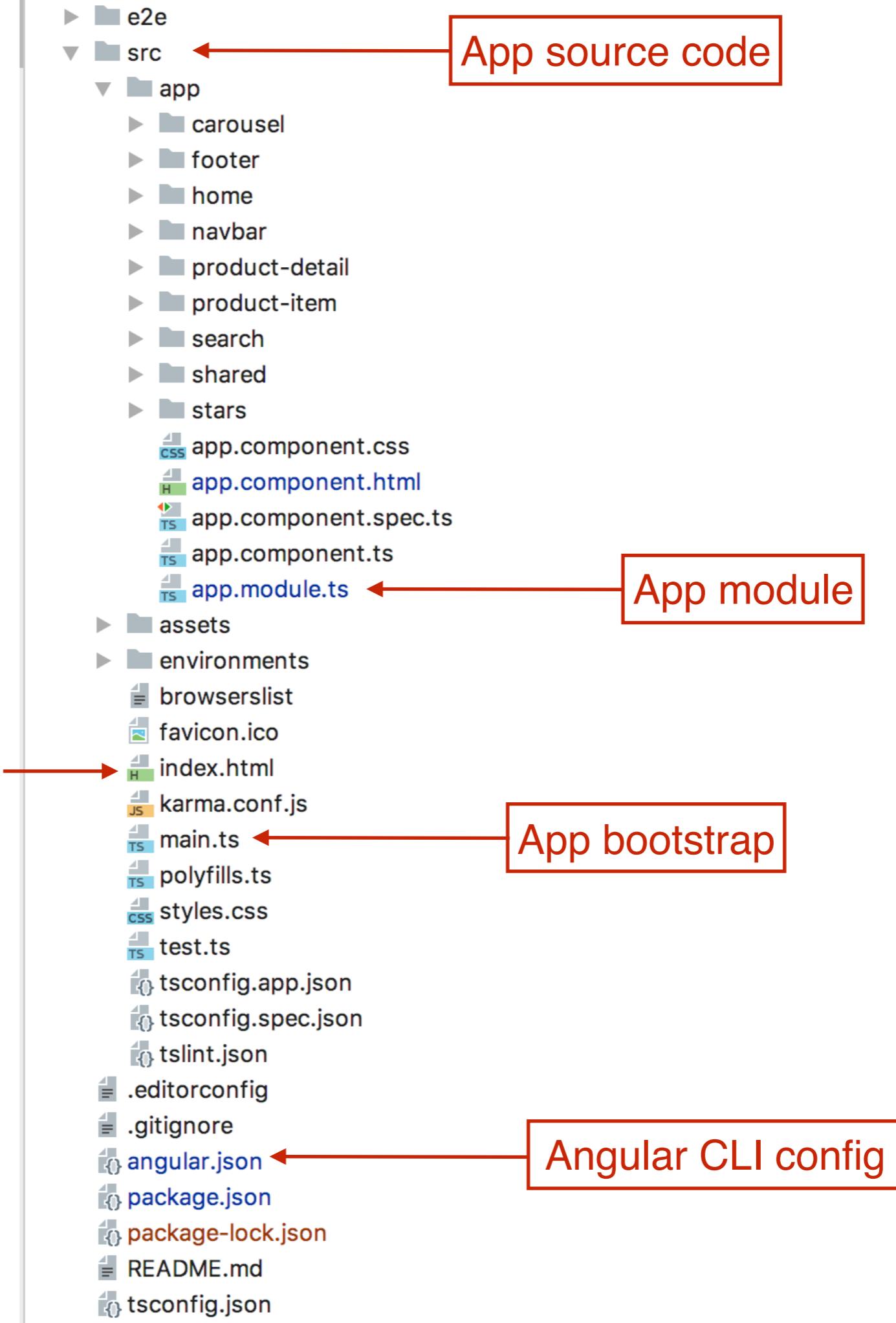
--routing

--inline-style

--inline-template

--skip-tests

Sample project structure



V6 angular.json

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "hello-cli": {  
      "root": "",  
      "sourceRoot": "src",  
      "projectType": "application",  
      "prefix": "app",  
      "schematics": {},  
      "architect": {  
        "build": {  
          "builder": "@angular-devkit/build-angular:browser",  
          "options": {  
            "outputPath": "dist/hello-cli",  
            "index": "src/index.html",  
            "main": "src/main.ts",  
            "polyfills": "src/polyfills.ts",  
            "tsConfig": "src/tsconfig.app.json",  
            "assets": [  
              "src/favicon.ico",  
              "src/assets"  
            ],  
            "styles": [  
              "src/styles.css"  
            ],  
            "scripts": []  
          },  
          "configurations": {  
            "production": {  
              "fileReplacements": [  
                {  
                  "replace": "src/environments/environment.ts",  
                  "with": "src/environments/environment.prod.ts"  
                }  
              ],  
              "optimization": true,  
              "outputHashing": "all",  
              "sourceMap": false,  
              "extractCss": true,  
              "namedChunks": false,  
              "aot": true,  
              "extractLicenses": true,  
              "vendorChunk": false,  
              "buildOptimizer": true  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

One or more projects

app or library

bundled app goes here

Configuring environments

V5

.angular-cli.json

One or more apps

3rd party lib styles

3rd party lib scripts

bundled app goes here

app bootstrap

prefix for component selectors

put env vars here

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "project": {  
    "name": "auction2"  
  },  
  "apps": [  
    {  
      "root": "src",  
      "outDir": "dist",  
      "assets": [  
        "assets",  
        "favicon.ico"  
      ],  
      "index": "index.html",  
      "main": "main.ts",  
      "polyfills": "polyfills.ts",  
      "test": "test.ts",  
      "tsconfig": "tsconfig.app.json",  
      "testTsconfig": "tsconfig.spec.json",  
      "prefix": "app",  
      "styles": [  
        "styles.css",  
        "./node_modules/bootstrap/dist/css/bootstrap.css"  
      ],  
      "scripts": ["./node_modules/jquery/dist/jquery.js",  
                 "./node_modules/bootstrap/dist/js/bootstrap.js"]  
    },  
    "environmentSource": "environments/environment.ts",  
    "environments": {  
      "dev": "environments/environment.ts",  
      "prod": "environments/environment.prod.ts"  
    }  
  },  
  "e2e": {  
    "protractor": {  
      "config": "./protractor.conf.js"  
    }  
  },  
  "lint": [  
    {  
      "project": "src/tsconfig.app.json"  
    },  
    {  
      "project": "src/tsconfig.spec.json"  
    },  
    {  
      "project": "e2e/tsconfig.e2e.json"  
    }  
  ],  
  "test": {  
    "karma": {  
      "config": "./karma.conf.js"  
    }  
  },  
  "defaults": {  
    "styleExt": "css",  
    "component": {}  
  }  
}
```

Generated app.component.ts: and app.component.html

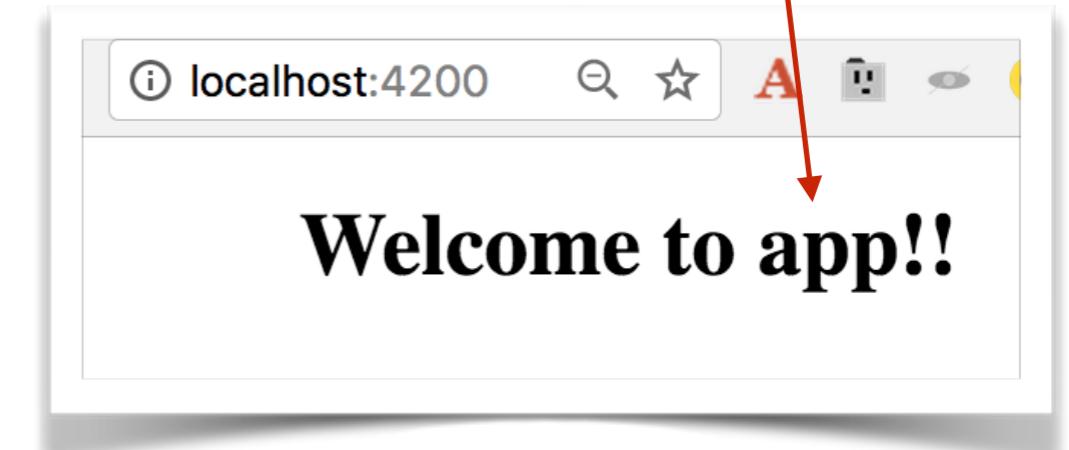
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

app.component.html

```
<div style="text-align:center">
<h1>
  Welcome to {{title}}!!
</h1>
</div>
...
...
```



ng generate (ng g)

- Component: ng g c
- Service: ng g s
- Module: ng g m
- Directive: ng g d
- Class: ng g cl
- Pipe: ng g p
- Help: ng help generate

ng generate samples

- ng g m shipping
- ng g c product
- ng g c product -is -it -spec false
- ng g s product

Inline styles,
inline templates,
no tests

Intro to TypeScript

Watch my video training “TypeScript Essentials” on
[safaribooksonline.com](https://www.safaribooksonline.com)

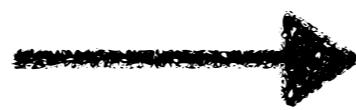
What's TypeScript

- A superset of JavaScript created by Microsoft.
- Increases your productivity in writing JavaScript
- Supports types, classes, interfaces, generics, annotations
- Compiles code into a human-readable JavaScript
- Supports ES6, ES7, and ES8

Transpiling TypeScript Interactively

<http://www.typescriptlang.org/play>

TypeScript



JavaScript (ES5)

The screenshot shows the TypeScript playground interface. On the left, under the 'TypeScript' tab, there is a code editor with the following TypeScript code:

```
1 let foo: string;
2
3 class Bar {
4
5 }
```

Below the code editor are buttons for 'Select...', 'Share', 'Run', and 'JavaScript'. The 'Run' button is highlighted in blue. On the right, under the 'JavaScript' tab, the transpiled ES5 code is displayed:1 var foo;
2 var Bar = (function () {
3 function Bar() {
4 }
5 return Bar;
6 })();
7 |

Classes

TypeScript



JavaScript (ES5)

```
1 class Person {  
2     firstName: string;  
3     lastName: string;  
4     age: number;  
5     ssn: string;  
6 }  
7  
8 var p = new Person();  
9  
10 p.firstName = "John";  
11 p.lastName = "Smith";  
12 p.age = 29;  
13 p.ssn = "123-90-4567";
```

```
1 var Person = (function () {  
2     function Person() {  
3     }  
4     return Person;  
5 })();  
6 var p = new Person();  
7 p.firstName = "John";  
8 p.lastName = "Smith";  
9 p.age = 29;  
10 p.ssn = "123-90-4567";  
11
```

Arrow Function Expressions

```
let getName = () => 'John Smith';
```

```
console.log(`The name is ` + getName());
```

Anonymous
function

TypeScript

Select...

Share

```
1 let getName = () => 'John Smith';
2 console.log(`The name is ` + getName());
```

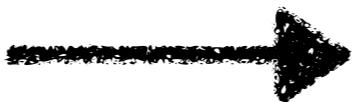
Run

JavaScript

```
1 var getName = function () { return 'John Smith';
2 console.log("The name is " + getName());
3
```

A Class With Constructor

TypeScript



JavaScript (ES5)

The screenshot shows a comparison between TypeScript and JavaScript (ES5) code. On the left, the TypeScript code defines a Person class with a constructor that takes firstName, lastName, age, and _ssn as parameters. It also includes a log statement. On the right, the generated JavaScript code uses a factory function to create instances of the Person class, setting the properties on the this object. Both snippets include a 'Run' button at the top.

TypeScript	JavaScript (ES5)
<pre>1 class Person { 2 constructor(firstName: string, 3 lastName: string, public age: number, private _ssn: string) { 4 } 5 } 6 7 var p = new Person("John", "Smith", 29, "123-90-4567"); 8 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);</pre>	<pre>1 var Person = (function () { 2 function Person(firstName, lastName, age, _ssn) { 3 this.firstName = firstName; 4 this.lastName = lastName; 5 this.age = age; 6 this._ssn = _ssn; 7 } 8 return Person; 9 })(); 10 var p = new Person("John", "Smith", 29, "123-90-4567"); 11 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);</pre>

Inheritance

Classical syntax

Prototypal

TypeScript

Select...

Share

```
1 class Person {  
2  
3     constructor(public firstName: string,  
4                 public lastName: string, public age: number,  
5                 private _ssn: string) {  
6     }  
7 }  
8  
9 class Employee extends Person{  
10 }  
11 }
```

JavaScript

Run

```
1 var __extends = this.__extends || function (d, b) {  
2     for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];  
3     function __() { this.constructor = d; }  
4     __.prototype = b.prototype;  
5     d.prototype = new __();  
6 };  
7 var Person = (function () {  
8     function Person(firstName, lastName, age, _ssn) {  
9         this.firstName = firstName;  
10        this.lastName = lastName;  
11        this.age = age;  
12        this._ssn = _ssn;  
13    }  
14    return Person;  
15 })();  
16 var Employee = (function (_super) {  
17     __extends(Employee, _super);  
18     function Employee() {  
19         _super.apply(this, arguments);  
20     }  
21     return Employee;  
22 })(Person);
```

Generics

TypeScript

Select...

Share

```
1 class Person {  
2     name: string;  
3 }  
4  
5 class Employee extends Person{  
6     department: number;  
7 }  
8  
9 class Animal {  
10    breed: string;  
11 }  
12  
13 var workers: Array<Person> = [];  
14  
15 workers[0] = new Person();  
16 workers[1] = new Employee();  
17 workers[2] = new Animal();  
18
```

Compile time error

Run

JavaScript

```
1 var __extends = (this && this.__extends) || function (d, b) {  
2     for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];  
3     function __() { this.constructor = d; }  
4     d.prototype = b === null ? Object.create(b) : (__.prototype =  
5 );  
6     var Person = (function () {  
7         function Person() {}  
8         return Person;  
10    })();  
11    var Employee = (function (_super) {  
12        __extends(Employee, _super);  
13        function Employee() {  
14            _super.apply(this, arguments);  
15        }  
16        return Employee;  
17    })(Person);  
18    var Animal = (function () {  
19        function Animal() {}  
20        return Animal;  
22    })();  
23    var workers = [];  
24    workers[0] = new Person();  
25    workers[1] = new Employee();  
26    workers[2] = new Animal();  
27
```

No Errors

Interfaces as Custom Types

TypeScript

Select...

Share

Run

JavaScript

```
1 interface IPerson { ←  
2  
3   firstName: string;  
4   lastName: string;  
5   age: number;  
6   ssn?: string;  
7 }  
8  
9 class Person {  
10   constructor(public config: IPerson) {  
11   }  
12 }  
13 }  
14  
15 var aPerson: IPerson = {  
16   firstName: "John",  
17   lastName: "Smith",  
18   age: 29  
19 }  
20  
21 var p = new Person(aPerson);  
22 console.log("Last name: " + p.config.lastName );
```

```
1 var Person = (function () {  
2   function Person(config) {  
3     this.config = config;  
4   }  
5   return Person;  
6 })();  
7 var aPerson = {  
8   firstName: "John",  
9   lastName: "Smith",  
10  age: 29  
11 };  
12 var p = new Person(aPerson);  
13 console.log("Last name: " + p.config.lastName );  
14
```

No interfaces
here

Interfaces and implements

```
1 interface IPayable{  
2  
3     increasePay(percent: number): void  
4 }  
  
5 class Employee implements IPayable{  
6  
7     increasePay(percent: number): void {  
8         // increase salary  
9     }  
10 }  
11 }  
12  
13 class Contractor implements IPayable{  
14  
15     increasePay(percent: number): void {  
16         // increase hourly rate  
17     }  
18 }  
19  
20 var workers: Array<IPayable> = [];  
21 workers[0] = new Employee();  
22 workers[1] = new Contractor();  
23  
24 workers.forEach(worker => worker.increasePay(30));
```

```
1 var Employee = (function () {  
2     function Employee() {  
3     }  
4     Employee.prototype.increasePay = function (percent) {  
5         // increase salary  
6     };  
7     return Employee;  
8 })();  
9 var Contractor = (function () {  
10    function Contractor() {  
11    }  
12    Contractor.prototype.increasePay = function (percent) {  
13        // increase hourly rate  
14    };  
15    return Contractor;  
16 })();  
17 var workers = [];  
18 workers[0] = new Employee();  
19 workers[1] = new Contractor();  
20 workers.forEach(function (worker) { return worker.increasePay(30);  
21 })
```

No interfaces
in JavaScript

Destructuring

```
1 function getStock(){  
2   return {  
3     symbol: "IBM",  
4     price: 100.00,  
5     open: 99.5,  
6     volume:100000  
7   };  
8 }  
9 let {symbol, price} = getStock();  
10  
11 console.log(`The price of ${symbol} is ${price}`);  
12  
13
```

```
1 function getStock() {  
2   return {  
3     symbol: "IBM",  
4     price: 100.00,  
5     open: 99.5,  
6     volume: 100000  
7   };  
8 }  
9 var _a = getStock(), symbol = _a.symbol, price = _a.price;  
10 console.log("The price of " + symbol + " is " + price);  
11  
12  
13  
14
```

TypeScript Compiler: tsc

- Install the typescript compiler tsc globally:

```
npm install typescript -g
```

- To compile main.ts into main.js (ES5 target):

```
tsc --t ES5 main.ts
```

Sample tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "ES5",  
    "module": "commonjs",  
    "experimentalDecorators": true  
  }  
}
```

Compiler's options: <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

Walkthrough 1.2

1. Install tsc globally by running the following command in your command prompt window:

`npm i -g typescript`

2. Switch to the directory `unit1/w2`

3. Review the code in `main.ts`

4. Compile `main.ts` using options from `tsconfig.json`:

`tsc`

5. Open the newly created file `main.js` - it's a valid JavaScript code (ES5).

6. Change the target in `tsconfig.json` to be ES6

7. Compile `main.ts` again:

`tsc`

8. Open the newly created file `main.js` - it's a valid JavaScript code (ES6).

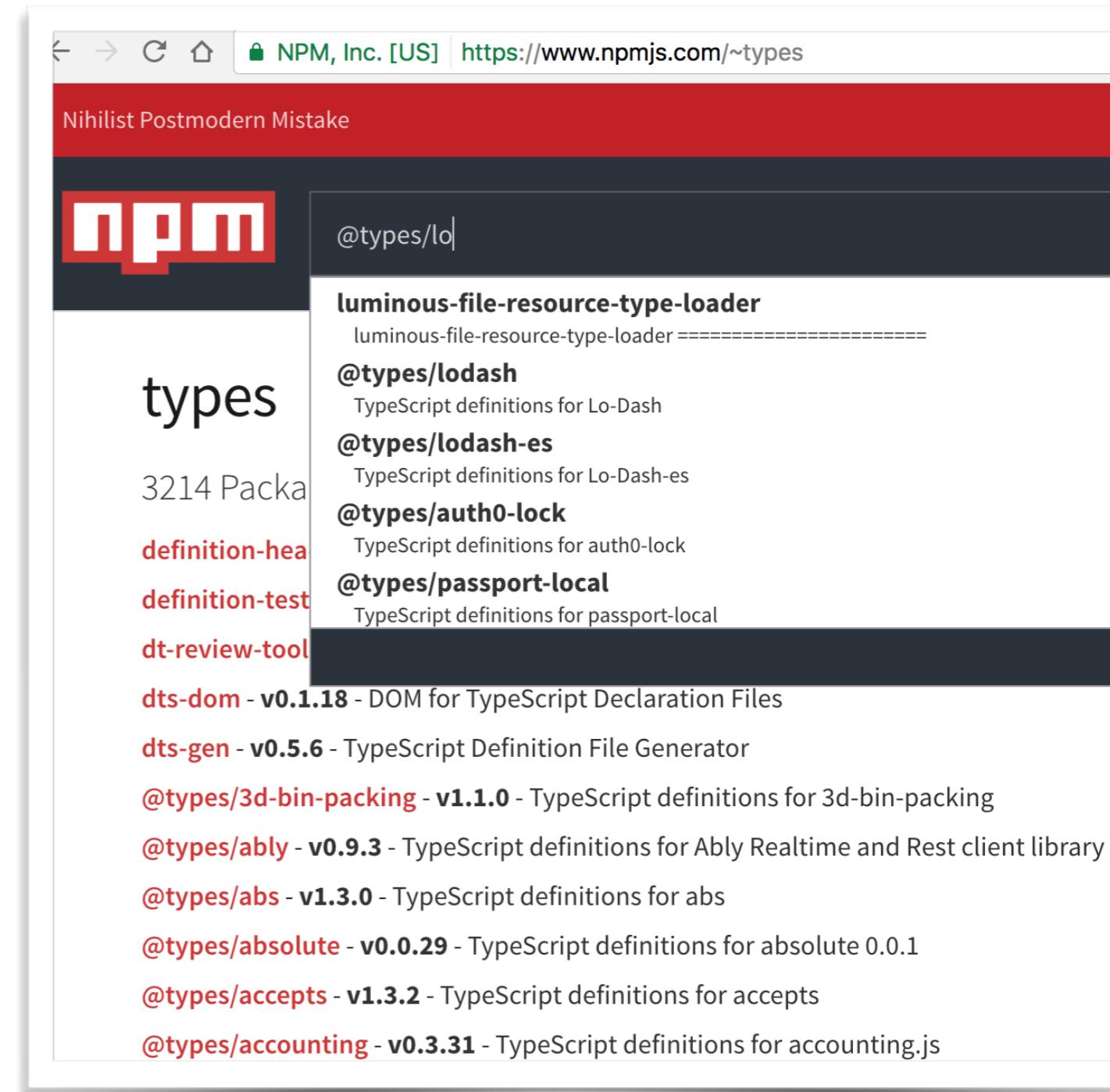
Typescript Type Definitions

Type definition files

- Type definition files (*.d.ts) contain declarations of types for JavaScript libraries
- *.d.ts files help IDE with a type-ahead help
- TypeScript static analyzer uses *.d.ts files to report errors

- npmjs.org has 3K+ of *.d.ts files
- <https://www.npmjs.com/~types>
- Install type definitions, e.g.

`npm i @types/lodash --save`
`npm i @types/jquery --save`



Declarative Templates

Declaring templates and styles

- Component's UI is declared in template or templateURL properties of @Component annotation
- If HTML is short - use template
- If HTML is long - use templateURL and keep the markup in a separate file
- For styles you can use either styles or stylesURL.

```
@Component({
  selector: 'app-root',
  template: '<h1>Hello World!</h1>'
})
```

```
@Component({
  selector: 'app-root',
  stylesURL: '[app.component.css]',
  template: '<h1>Hello World!</h1>'
})
```

Multi-line template strings

Use back ticks to write multi-line templates

```
@Component({
  selector: 'auction-home-page',
  ...
  styleUrls: [ 'home.component.css' ],
  template: `←
    <div class="row carousel-holder">
      <div class="col-md-12">
        <auction-carousel></auction-carousel>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <div class="form-group">
          <input placeholder="Filter products by title" type="text">
        </div>
      </div>
    </div>
  ` →
})
```

Data Binding

- Data binding allows to keep the view and underlying data in sync
- UI elements can be bound to component properties
- Events can be bound to functions or expressions

Unidirectional Binding

From code to template:

Text interpolation
→
<h1>Hello {{ name }}!</h1>

Zip code is not valid
↑
**Property
binding** ↑
**A class
variable**

Unidirectional Binding

From template to code:

```
<button (click)="placeBid()">Place Bid</button>
```



```
<input (input)=“onInputEvent()” />
```

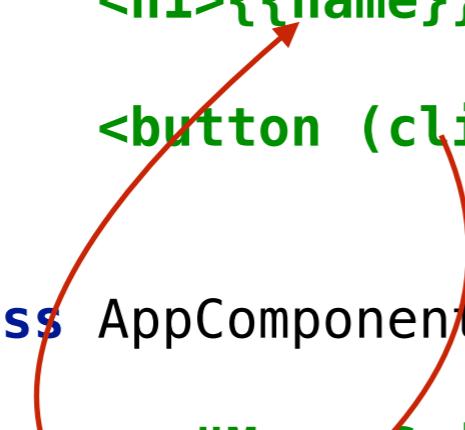
```
Custom  
event
```

A red vertical arrow points downwards from the `<price-quoter (lastPrice)=“priceQuoteHandler($event)”>` element to the `<input (input)=“onInputEvent()” />` element. To the left of the arrow, the text "Custom event" is written in red.

```
<price-quoter (lastPrice)=“priceQuoteHandler($event)”>  
</price-quoter>
```

Property and Event Binding

```
@Component({
  selector: 'one-way-binding',
  template:
    `
      <h1>{{name}}</h1>
      <button (click)="changeName()">Change Name</button>
    `
})
class AppComponent {
  name = "Mary Smith";
  changeName(){
    this.name = "Bill Smart";
  }
}
```



Two-way Binding with ngModel

```
@Component({
  selector: 'two-way-binding',
  template: `<input type='text' placeholder= "Enter shipping address"
    → [(ngModel)] = "shippingAddress">
    <button (click)="shippingAddress='123 Main Street'">Set Default Address</button>
    <p>The shipping address is {{shippingAddress}}</p>
`)
class AppComponent {
  shippingAddress: string;
}
```

Walkthrough 1.3 (start)

- In your IDE open `unit1/w3/bindings` and in its Terminal window run this command:
`npm i`
- Review the code in `src/app/one-way/one.way.component.ts`
- Bundle up and run the app oneway in the browser:
`ng serve oneway -o`
- Press the button **Change name**. One-way binding works.
- In the terminal window stop the web server (Ctrl-C)

Walkthrough 1.3 (end)

- Review the code in `src/app/two-way/app.component.ts`
- Stop the dev server if running (Ctrl-C)
- Build and run the app two way:
`ng serve two way -o`
- Type some text in the input field - the binding from UI to the property `shippingAddress` works
- Click the button to programmatically change property `shippingAddress` - the binding works

What have we learned

- Angular CLI generates the new app and all its dependencies
- TypeScript compiler options are configured in `tsconfig.json`
- `npm install` downloads all dependencies listed in `package.json`
- You can configure multiple apps in the file `angular.json`
- An Angular component is a class annotated with `@Component`
- An app requires at least one class decorated with `@NgModule()`
- A component's HTML is specified in `template` or `templateURL`
- Angular supports both one-way and two-way data binding

Angular Development with TypeScript

Unit 2: App navigation with the router

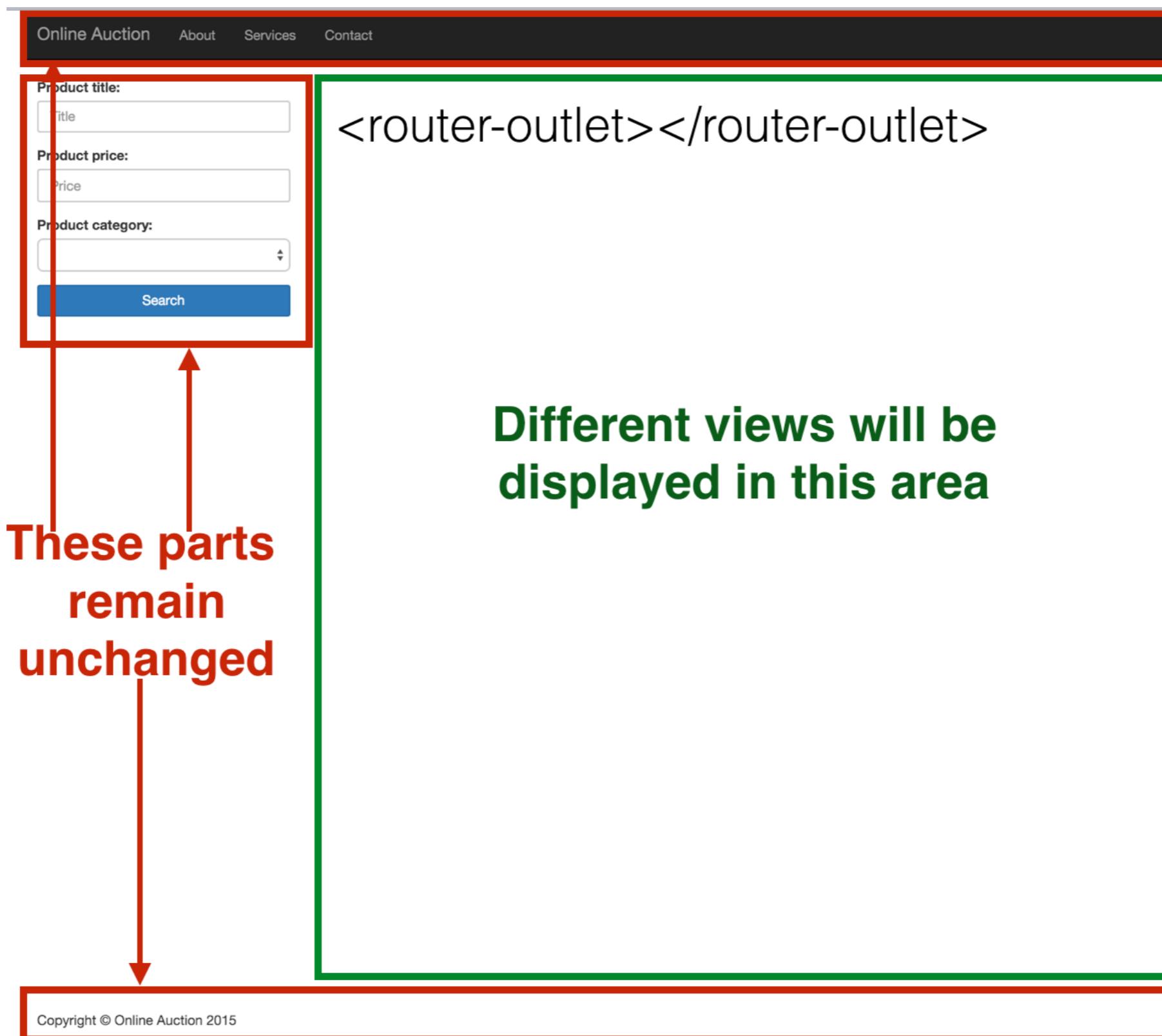
In this unit

- Main router elements
- Configuring routes
- Passing data to routes
- Child routes
- Guarding routes
- Lazy loading of modules

Single-Page Apps (SPA)

- SPA don't refresh the entire page to display different views
- SPA is a collection of view states: Home, Product Detail, Shipping etc.
- The Router allows the user to navigate from one view to another within the SPA

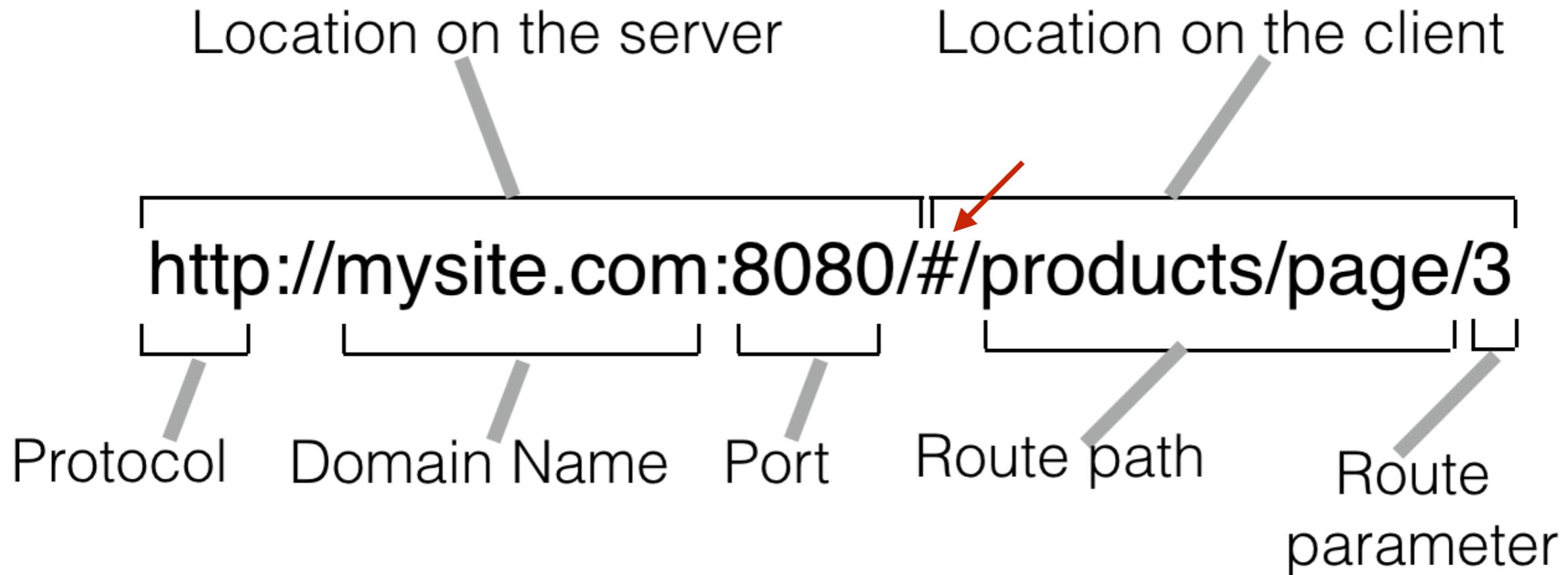
Router Outlet



Location Strategies

- **HashLocationStrategy** - a hash sign (#) is added to the URL, and the fragment after the hash identifies the route on the client.
- **PathLocationStrategy** - a History API based strategy works only in the browsers that support HTML5.

Location Strategies: Hash-based



```
@NgModule({  
  ...  
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

Location Strategies: History API

http://mysite.com:8080/products/page/3



no hash

```
@NgModule({  
  ...  
  providers: [{provide: APP_BASE_HREF, useValue: '/'},  
  bootstrap: [ApplicationComponent]  
})  
export class AppModule { }
```

APP_BASE_HREF affects how the router resolves routerLink and router.navigate() calls within the app

<base href="..."> affects how the browser resolves URLs when loading static resources like <link>, <script>, and .

Sample routes configuration

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent} from "./home";
import {ProductDetailComponent} from "./product";

const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product', component: ProductDetailComponent}
];

export const routing = RouterModule.forRoot(routes); // config for the root module
```

app.routing.ts



The root module

```
...
import {LocationStrategy, HashLocationStrategy} from '@angular/common';

import {routing} from './components/app.routing';

@NgModule({
  imports:      [ BrowserModule, routing ],
  declarations: [ AppComponent, HomeComponent, ProductDetailComponent ],
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap:   [ AppComponent ]
})
class AppModule { }
```



Navigation using template links

```
@Component({  
  selector: 'app',  
  template:  
    <a [routerLink]="'/'>Home</a>  
    <a [routerLink]="/product">Product Details</a>  
  
    <router-outlet></router-outlet>  
})  
export class AppComponent {}
```

The screenshot shows a browser window with the title bar "Home Product Details". Below it, a red box highlights the text "Home Component". To the right, the browser's developer tools are open, specifically the "Elements" tab. The DOM tree is visible, starting with the <!DOCTYPE html> declaration and the <html> element. Inside the <body> element, there is an <app> component. Two <a> tags are present within this component: one with href="#" and another with href="#/product". A callout box with a red border and white text, containing the text "[routerLink] replaced with href", points to the first <a> tag. Red arrows also point from this callout box to the href attributes of both the <a> tags in the DOM tree.

```
<!DOCTYPE html>  
<html>  
  <head>...</head>  
  <body> == $0  
    <app>  
      <a ng-reflect-router-link="/" ng-reflect-href="#" href="#">Home</a>  
      <a ng-reflect-router-link="/product" ng-reflect-href="#/product" href="#">Product Details</a>  
      <router-outlet></router-outlet>  
      <home _ngcontent-qsl-3>...</home>  
    </app>  
    <!-- Code injected by live-server -->  
    <script type="text/javascript">...</script>  
  </body>  
</html>
```

Programmatic navigation

- navigate()
- navigateByURL()

```
template: `  
  <a [routerLink]="'/'>Home</a>  
  <button (click)="navigateToProductDetail()">Product Details</button>  
  <router-outlet></router-outlet>  
`
```

```
class AppComponent {  
  constructor(private _router: Router){}  
  navigateToProductDetail(){  
    this._router.navigate(['/product']);  
  }  
}
```

Walkthrough 2.1

- In your IDE open the dir **unit2/router-samples**
- In the Terminal window run
npm install
- In the src dir, review the code of the files `app.routing`,
`app.component`, `home.component`, and
`product.component`
- Run the **basic** app (configured in `angular.json`):
ng serve basic -o

Passing parameters to a route

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponent}
];

@Component({
  selector: 'app',
  template: `
    <a [routerLink]="/">Home</a>
    <a [routerLink]="/product", productId>Product Details</a>
    <router-outlet></router-outlet>
  `
})
class AppComponent {
  productId = 1234;
}
```

The diagram illustrates the flow of parameters from the AppComponent class to the routerLink in the template. A red arrow points from the `productId` variable in the AppComponent class to the `productId` placeholder in the `[routerLink]` attribute of the second `a` tag in the template. Another red arrow points from the `productId` variable to the `productId` placeholder in the `[routerLink]` attribute.

Receiving params in a route: ActivatedRoute

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'product',
  template: `<h1 class="product">Details for product {{productID}}</h1>`,
  styles: ['.product {background: cyan}']
})
export class ProductDetailComponent {
  productID: string;

  constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.paramMap.get('id');
  }
}
```

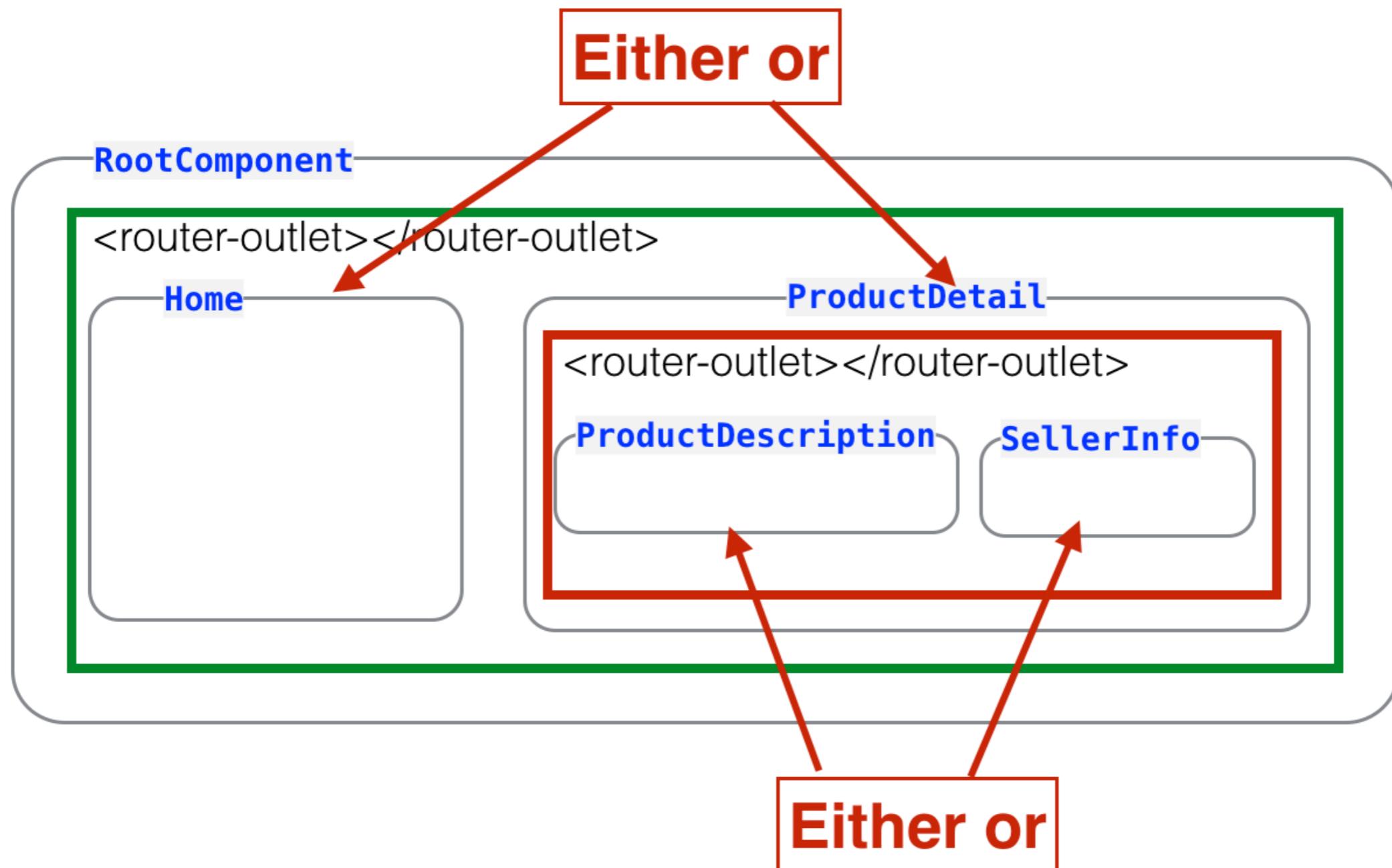
You can also receive params by subscribing to an observable property of the ActivatedRoute

Walkthrough 2.2

- Review the code in the dir params
- In app.component.ts the ProductDetail links uses product/:id and renders ProductDetailComponent with 1234 as product ID.
- Note that app.component.ts includes
productId: **number** = **1234**;
- Note the product/:id configuration in app.routing.ts
- Stop the dev server if running (Ctrl-C)
- Run the params app:
ng serve params -o
- In the browser navigate to Product Detail. It renders “Product Detail for Product 1234”

Child Routes

A child component can have its own routes configured



Configuring child routes

Configure routes for child components using the `children` property of the configuration object

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponent,
   → children: [
      {path: '', component: ProductDescriptionComponent},
      {path: 'seller/:id', component: SellerInfoComponent}
    ]}
];

@Component({
  selector: 'app',
  template: `
    <a [routerLink]="'/'>Home</a>
    <a [routerLink]="'/product', 1234">Product Details</a>
    <router-outlet></router-outlet>
  `
})
class AppComponent {}
```

Walkthrough 2.3

- Review the code of the files in the dir `child`
- `ng serve child -o`
- The main page is displayed. Click on the link [Product with children](#), and it'll navigate to the `ProductDescription` route inside `ProductDetail`
- Click on the [Seller Info](#) link and it'll navigate to the `SellerInfo` route inside `ProductDetail`
- Note the URL in the browser:
<http://localhost:4200/#/product/1234/seller/5678>

parent's route child's route

Guard interfaces

- `CanActivate` mediates the navigation to a route
- `CanActivateChild` mediates the navigation to a child route
- `CanDeactivate` mediates the navigation away from the current route
- `Resolve` performs route data retrieval before route activation
- `CanLoad` mediates the navigation to a feature module loaded asynchronously

canActivate and canActivate

- canActivate - allows navigating to a route if a certain condition is met, e.g. the user is logged in
- canDeactivate - allows navigating from a route if a certain condition is met, e.g. prompting about saving changes

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'login', component: LoginComponent},
  {path: 'product', component: ProductDetailComponent,
    canActivate: [LoginGuard], canDeactivate: [UnsavedChangesGuard]}
];

export const routing = RouterModule.forRoot(routes);
```

Login Guard

```
@Injectable()
export class LoginGuard implements CanActivate{
    constructor(private router: Router){}
    →canActivate() {
        // A call to the actual login service would go here
        // For now we'll just randomly return true or false
        const loggedIn = Math.random() < 0.5;
        if(!loggedIn){
            alert("You're not logged in and will be redirected to Login page");
            this.router.navigate(['/login']);
        }
        return loggedIn;
    }
}
```



Unsaved Changes Guard

```
export class UnsavedChangesGuard implements CanDeactivate<ProductDetailComponent> {  
  constructor(private _router:Router){}
  →canDeactivate(component: ProductDetailComponent){  
    if (component.name.dirty) {
      return window.confirm("You have unsaved changes. Still want to leave?");
    } else {
      return true;
    }
  }
}
```

The `dirty` property is a part of the Forms API

Unsaved Changes Guard

```
export class UnsavedChangesGuard implements CanDeactivate<ProductDetailComponent>{  
  constructor(private _router:Router){}  
  
  canDeactivate(component: ProductDetailComponent){  
  
    if (component.name.dirty) {  
      return window.confirm("You have unsaved changes. Still want to leave?");  
    } else {  
      return true;  
    }  
  }  
}
```

```
@Component({  
  selector: 'product',  
  template: `<h1 class="product">Product Detail Component</h1>  
           <input placeholder="Enter your name"  
                 type="text" [formControl]="name">`,  
  styles: ['.product {background: cyan}']  
})  
export class ProductDetailComponent{  
  
  name: FormControl = new FormControl();  
}
```

Walkthrough 2.4

- Review the code in the files in the dir guards
- `ng serve guards -o`
- Click on the Product Detail link and enter your name
- The app either navigates to ProductDescription route or shows a popup that you're not logged in and navigates to the Login view
- When ProductDetail is shown, click on the Home link, and you'll see the message asking if you want to save changes

Feature modules and lazy loading

Advantages of lazy loading

- The app startup time is shorter
- Splitting the app into modules allows creating reusable chunks of code, e.g. ShippingModule
- During bundling, the lazily loaded module is placed in a separate bundle
- The code is loaded only when the user navigates to the module

Root vs feature modules

```
@NgModule({  
  imports: [ BrowserModule ],  
  declarations: [ AppComponent, HomeComponent,  
    ProductDetailComponent ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

Root module

```
@NgModule({  
  imports: [ CommonModule,  
    RouterModule.forChild([  
      {path: '', component: LuxuryComponent}  
    ]) ],  
  declarations: [ LuxuryComponent ]  
})  
export class LuxuryModule { }
```

Feature module

Creating a lazy-loaded module

- Create a module that imports CommonModule and loads its routes using `forChild()`
- Don't import this module in your root app module
- In router configuration use the property `loadChildren` that has a string value

Routes config for lazy loading

```
@NgModule({
  imports: [ BrowserModule,
    RouterModule.forRoot([
      {path: '', component: HomeComponent},
      {path: 'product', component: ProductDetailComponent},
      {path: 'luxury',
        Lazy load → loadChildren: './luxury.module#LuxuryModule'}
    ])
  ],
  declarations: [ AppComponent, HomeComponent, ProductDetailComponent],
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

LuxuryModule

AppModule

```
chunk {inline} inline.bundle.js, inline.bundle.js.map
chunk {luxury.module} luxury.module.chunk.js, luxury.m
chunk {main} main.bundle.js, main.bundle.js.map (main)
chunk {polyfills} polyfills.bundle.js, polyfills.bundl
chunk {styles} styles.bundle.js, styles.bundle.js.map
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map
```

Preloaders

- During the app launch, it can preload all lazy modules in the background while the user is interacting with your app
- Add `preloadStrategy` to the routes config

```
RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'product', component: ProductDetailComponent},
  {path: 'luxury', loadChildren: './luxury.module#LuxuryModule' }
],
{
  → preloadingStrategy: PreloadAllModules
})
```

- You can create a custom preloading strategy by creating a class that implements the `PreloadingStrategy` interface.

Walkthrough 2.5

- Review the code in the files located in the dir `lazy`
- `ng serve lazy -o`
- Open Chrome Dev Tools on the **Network** tab and refresh
- Note that the browser loads `luxury.module.chunk.js` only when you click on the Luxury Item link
- Stop the dev server
- In `lazy/app.module.ts` uncomment the line with `preloadStrategy`
- The app restarts and the Network tab shows that the file `luxury.module.chunk.js` was preloaded

What have we learned

- The router renders components in the designated area called router outlet
- Navigation to the routes can be done either by clicking on a link or programmatically
- The URL on the address bar of the browser is updated as the user navigates to the routes
- Parent components can pass data to routes
- Guards are used to mediate navigation to/from the route
- A child component can have its own route configuration
- The router can load modules either eagerly or lazily

Angular Development with TypeScript

Unit 3: Dependency injection

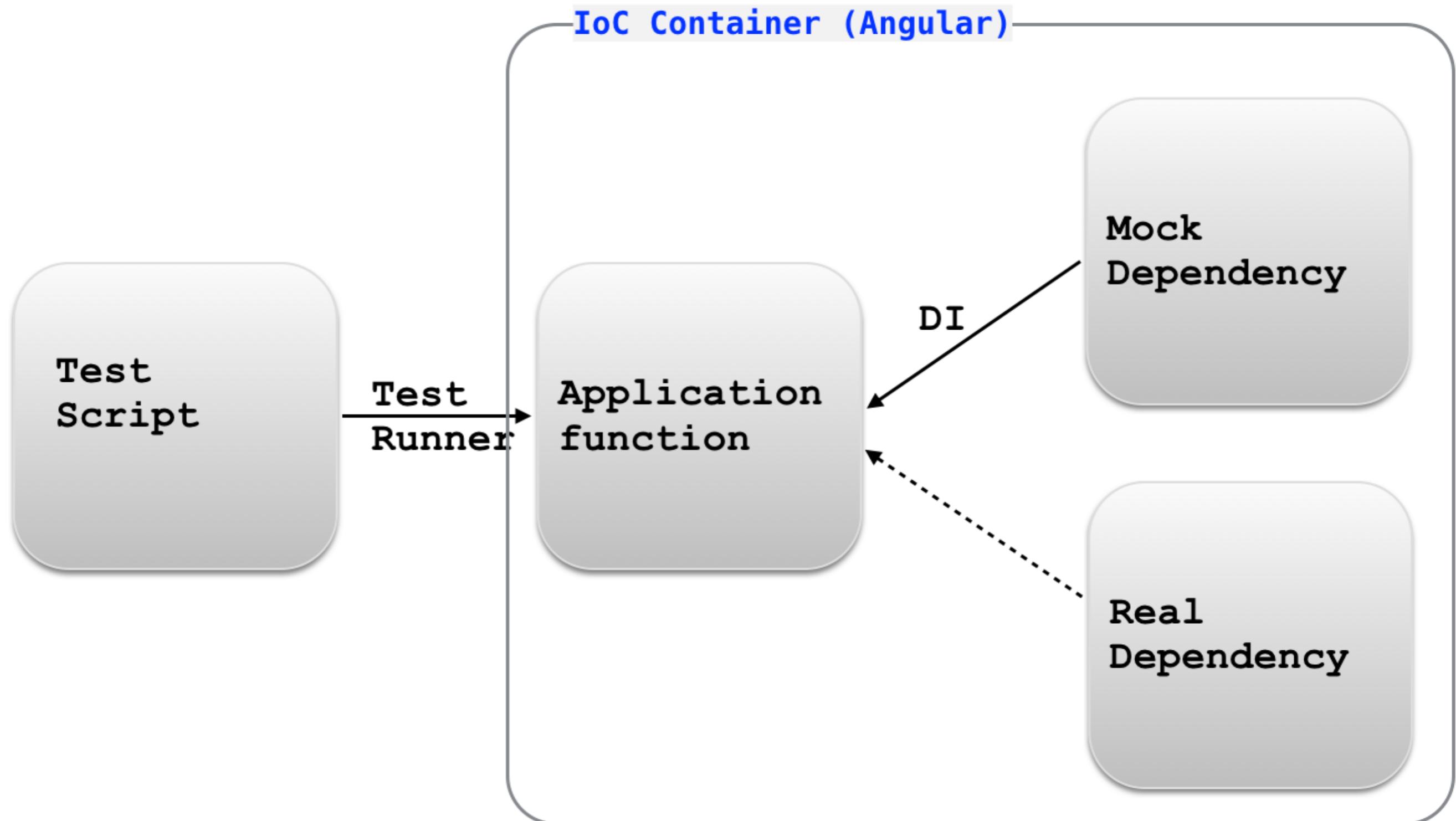
In this unit

- Benefits of dependency injection
- Providers and injectors
- Injecting using factories and values

Benefits of Dependency Injection

- Change the behavior of a component, service, function, etc) without modifying its code.
- Offers an easy way of swapping the objects that your app uses
- Simplifies testability of your app

DI: Simplified Testability



Dependency Injection in Angular

- Don't instantiate classes with the new operator
- Angular creates and injects services into components
- Angular creates and injects values into variables

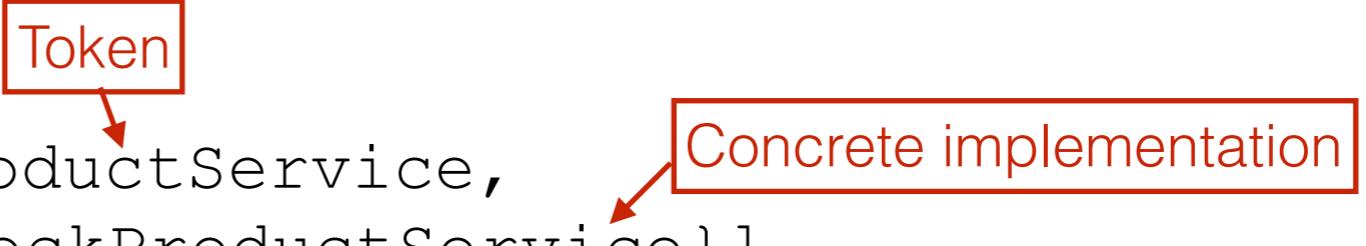
Dependency Injection in Angular

- Angular injects services into only via constructors
- Each component has **its own injector**
- You specify **a provider** so Angular knows **what** to inject

Where to declare providers?

- Providers allows you to map a *token*, to its concrete implementation
- Typically you define a provider on the app level to create a singleton

```
@NgModule ({  
  ...  
  providers: [{provide: ProductService,  
    useClass: MockProductService} ]  
})
```



You can define a provider inside `@Component` if need be.

Short notation for providers

When a token and a type have the same name,
use the short notation:

```
@NgModule ( {  
  ...  
  providers: [ ProductService ]  
} )
```

Injection point

- Angular injects values via the component's constructor:

```
constructor(productService: ProductService) {  
    productService.doSomething();  
}
```

- If a component doesn't declare a provider, Angular checks component's parent, grandparent, etc.

Changing a provider

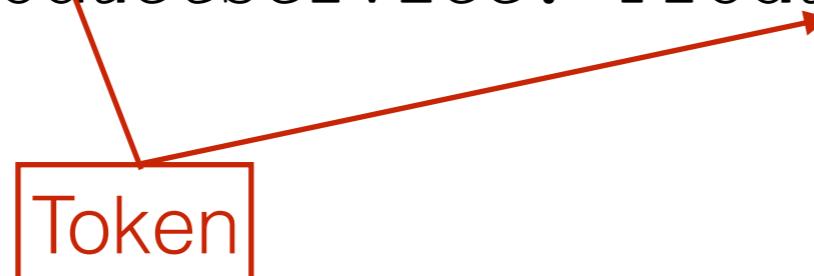
- To inject a **different** implementation of a service, change the provider:

```
{provide: ProductService, useClass: AnotherProductService}
```

- The component's constructor remains the same even when the concrete implementation changes:

```
constructor(productService: ProductService) { ... }
```

Token



Demo: generate a service with Angular CLI

- In IDE open your hello-cli project, and in the Terminal view run this:

ng g s product -m app.module

```
import { Injectable } from '@angular/core';
@Injectable()
export class ProductService {
  constructor() { }

}
```

- CLI generates product.service.ts and adds its provider to app.module.ts

```
... import { ProductService } from './product.service';

@NgModule({
  declarations: [
    AppComponent,
    ProductComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

A simple injectable service

```
import {Injectable} from "@angular/core";

@Injectable()
export class ProductService {
  getProduct(): Product {
    // An HTTP request can go here
    return new Product(0, "iPhone 7", 249.99, "The latest iPhone, 7-inch screen");
  }
}
```

```
export class Product {
  constructor(
    public id: number,
    public title: string,
    public price: number,
    public description: string) {
  }
}
```

Injecting ProductService

```
@Component({
  selector: 'di-product-page',
  template: `<div>
    <h1>Product Details</h1>
    <h2>Title: {{product.title}}</h2>
    <h2>Description: {{product.description}}</h2>
    <h2>Price: \${{product.price}}</h2>
  </div>`
})
export class ProductComponent {
  product: Product;

  constructor( productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

Injection



Walkthrough 3.1

- In your IDE, open the directory unit3/di-samples
- Review the code in the directory basic
- **npm install**
- **ng serve basic -o**
- The app shows the info about iPhone7 provided by the injected ProductService

Angular 6 and providedIn

Generate a service:

```
ng g s product
```

The injectable decorator looks like this

```
@Injectable(  
  providedIn: 'root'  
)
```

No need to specify a provider in @NgModule()

Dependencies of dependencies

**Dependency of
Dependency**

Dependency

Component

HttpClient

Injected
into

ProductService

Injected
into

ProductComponent



Dependencies of dependencies

Http providers

```
import {HttpClientModule}  
       from '@angular/common/http';  
  
...  
  
@NgModule({  
  imports:      [ BrowserModule,  
                 HttpClientModule], ←  
  declarations: [ AppComponent,  
                 ProductComponent],  
  bootstrap:    [ AppComponent ]  
})  
class AppModule { }
```

Dependencies of dependencies

```
import {HttpClientModule}  
       from '@angular/common/http';  
  
...  
  
@NgModule({  
  imports:      [ BrowserModule,  
                 HttpClientModule],  
  declarations: [ AppComponent,  
                 ProductComponent],  
  bootstrap:    [ AppComponent ]  
})  
class AppModule { }
```

Use `@Injectable()` if your service has dependencies.

```
import {HttpClient} from '@angular/common/http';  
  
@Injectable()  
export class ProductService {  
  
  products:Observable;  
  
  constructor(private http: HttpClient){  
  
    this.products = this.http.get('products.json');  
  }  
  // other app code goes here  
}
```

```
export class ProductComponent {  
  
  product: Product;  
  
  constructor( productService: ProductService) {  
  
  }  
}
```

Flavors of providers

The property `provide` can:

- map a token by `useClass`
- map a token by `useFactory`
- map a token by `useValue`

Injecting a value into a factory

```
export function productServiceFactory (isProd: boolean) {← Factory
  if (isProd){
    return new ProductService();
  } else{
    return new MockProductService();
  }
};

@Component({
  selector: 'product2',
  template: '{{product.title}}'
})
export class Product2Component {
  product: Product;

  constructor(productService: ProductService) {
    this.product = productService.getProduct();
  }
}
...
@NgModule({
  ...
  providers: [{provide: ProductService, useFactory: productServiceFactory,
    deps: ['IS_PROD_ENVIRONMENT']},
    {provide: 'IS_PROD_ENVIRONMENT', useValue: environment.production}],
  bootstrap: [ AppComponent ]
})
class AppModule { }
```

String as a token → true or false

Environment variables

- By default, `ng serve` uses dev environment settings from `environment.ts`
- Review the code in the directory `environments` in your project



Walkthrough 3.2

- Review the code in the factory folder. The app.module file has
 - `providers: [{provide: ProductService, useFactory: productServiceFactory, deps: ['IS_PROD_ENVIRONMENT']}]`
- `ng serve factory -o`
- With dev build `'IS_PROD_ENVIRONMENT'` is false; the app shows two Samsungs.
- Serve the prod version of the app:
`ng serve --prod factory -o`
- With prod build `'IS_PROD_ENVIRONMENT'` is true; the app shows two iPhones.
- Check the app size of the prod build in the Network tab. It's ~130KB.

What have we learned

- Having a framework creating instances provide more flexibility in switching services
- Providers specify how to create instances while injectors create them
- If you didn't specify a provider on a component level, Angular will look for one up the hierarchy
- Providers can be classes, factory functions, or values
- If a service requires injection of other services mark it as `@Injectable()`

Angular Development with TypeScript

Unit 4: Inter-component communications

In this unit

- `@Input` and `@Output` properties
- Using a parent component as a mediator
- Using an injectable service as a mediator
- 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

The 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;
  }
}
```



Input properties in a child

```
@Component({
  selector: 'order-processor',
  template: `
    <span *ngIf="!!stockSymbol">Buying {{quantity}} shares of {{stockSymbol}}</span>
  `,
  styles: [`:host {background: cyan;}`]
})
export class OrderProcessorComponent {

  → @Input() quantity: number;

  → @Input() stockSymbol: string;
}
```

Note the use of *ngIf directive

Walkthrough 4.1

- In your IDE open the directory `unit4/inter-component`
- `npm i`
- Review the code in the input directory
- `ng serve input -o`
- Enter a stock symbol and move the focus from the field
- The child component received the entered stock symbol

Output properties in a child

```
@Component({
  selector: 'price-quoter',
  template: `<strong>Inside PriceQuoterComponent:
    {{stockSymbol}} {{price | currency:'USD'}}</strong>`,
  styles:[` :host {background: pink;} `]
})
class PriceQuoterComponent {

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

  stockSymbol = "IBM";
  price:number;

  constructor() {
    setInterval(() => {      // can use Observable.interval() instead
      let priceQuote: IPriceQuote = {
        stockSymbol: this.stockSymbol,
        latestPrice: 100*Math.random()
      };

      this.price = priceQuote.latestPrice;

      this.lastPrice.emit(priceQuote);
    }, 1000);
  }
}
```

pipe

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

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'}}`  

})
class AppComponent {  

  stockSymbol: string;  

  price:number;  

  priceQuoteHandler(event:IPriceQuote) {  

    this.stockSymbol = event.stockSymbol;  

    this.price = event.latestPrice;  

  }
}
```

A diagram illustrating the data flow between the template and the component. A red arrow points from the 'lastPrice' event handler in the template to the 'latestPrice' field in the component's priceQuoteHandler method. Another red arrow points from the 'stockSymbol' binding in the template to the 'stockSymbol' field in the component's class definition.

Walkthrough 4.2

- Review the code in the output directory
- `ng serve output -o`
- Price quotes are generated and displayed in both the child and parent components

Implementing the Mediator Design Pattern

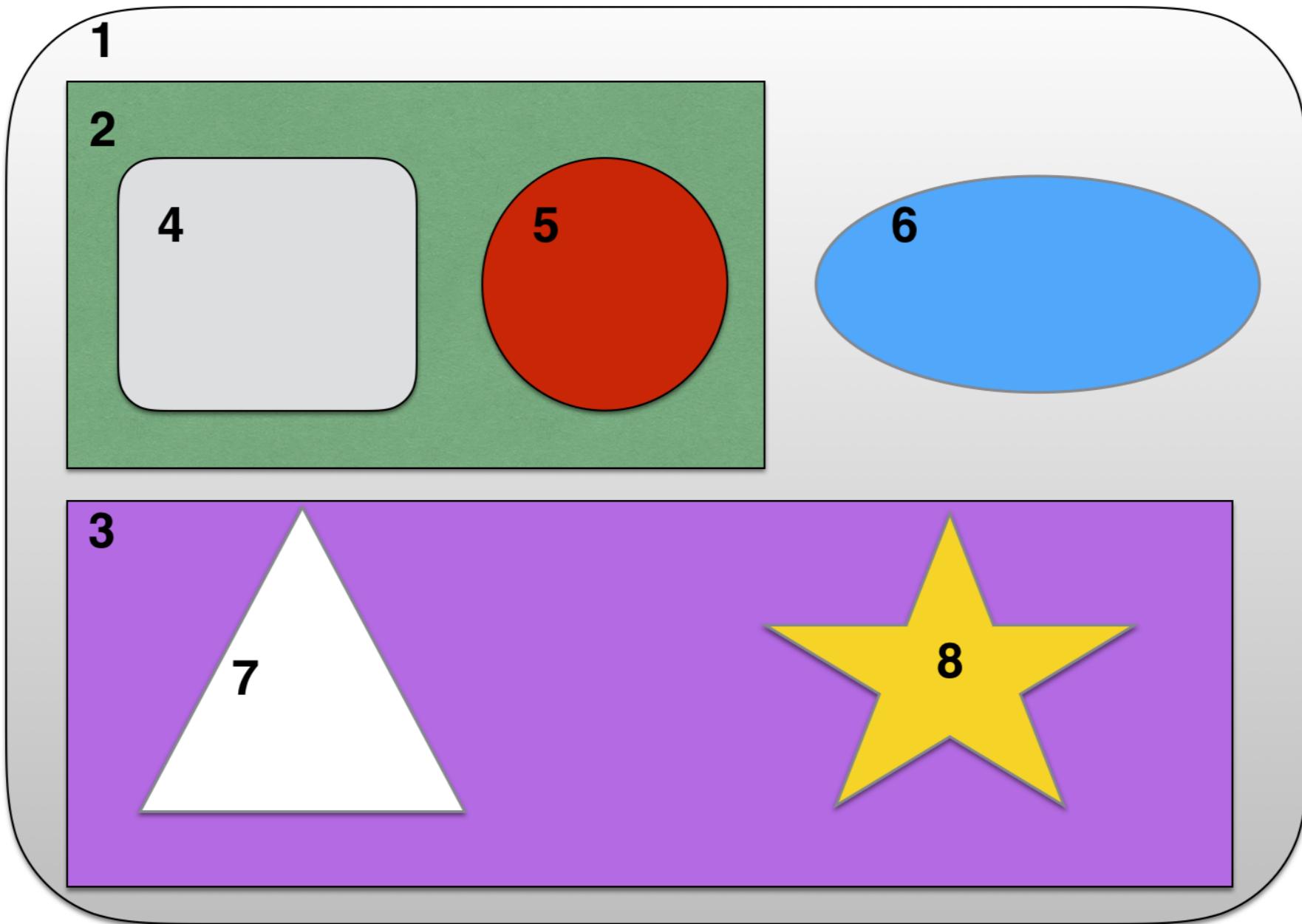
Loosely-Coupled Components

- 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 Angular app as matryoshka dolls

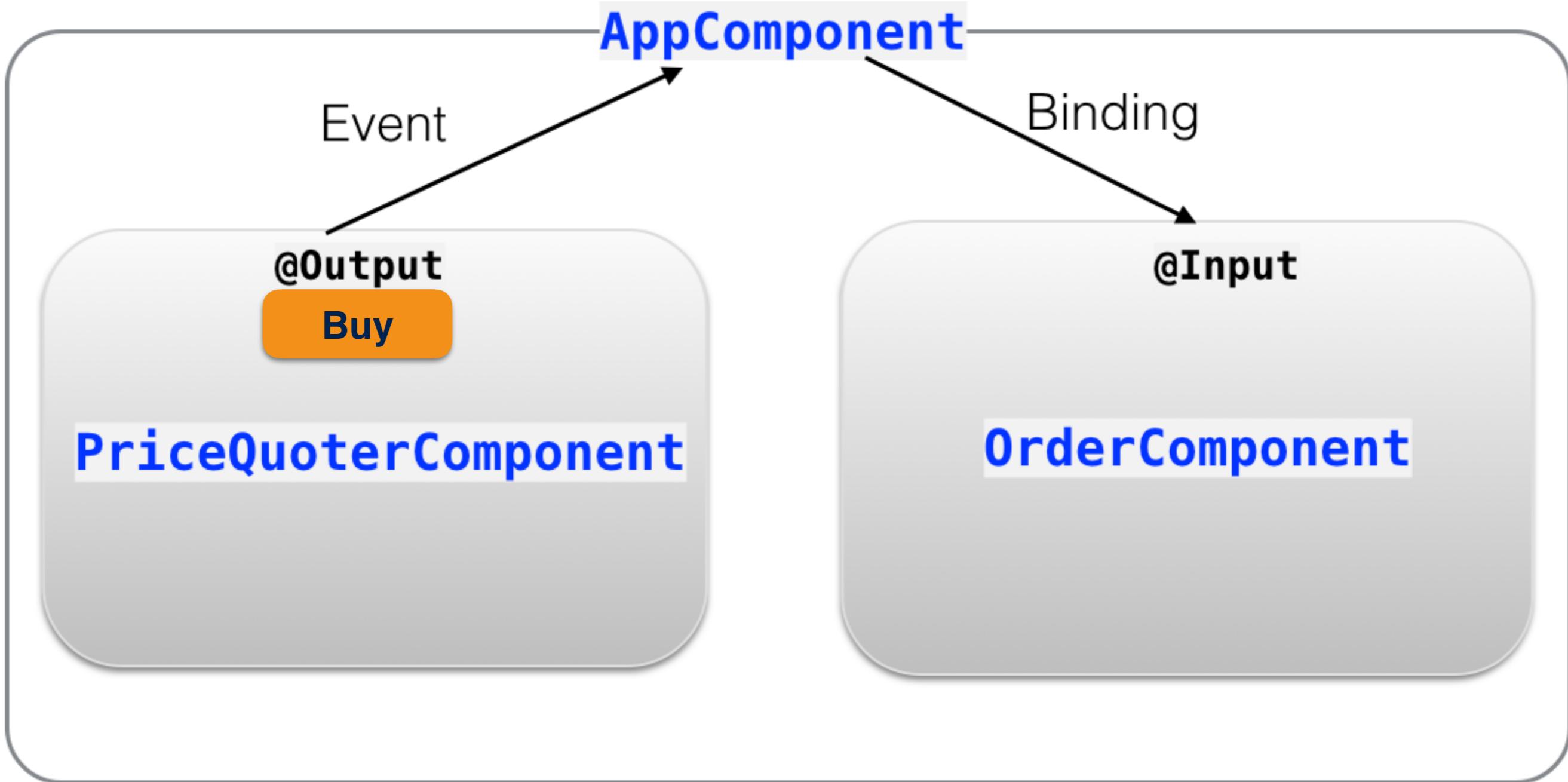


Parent components as mediators



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

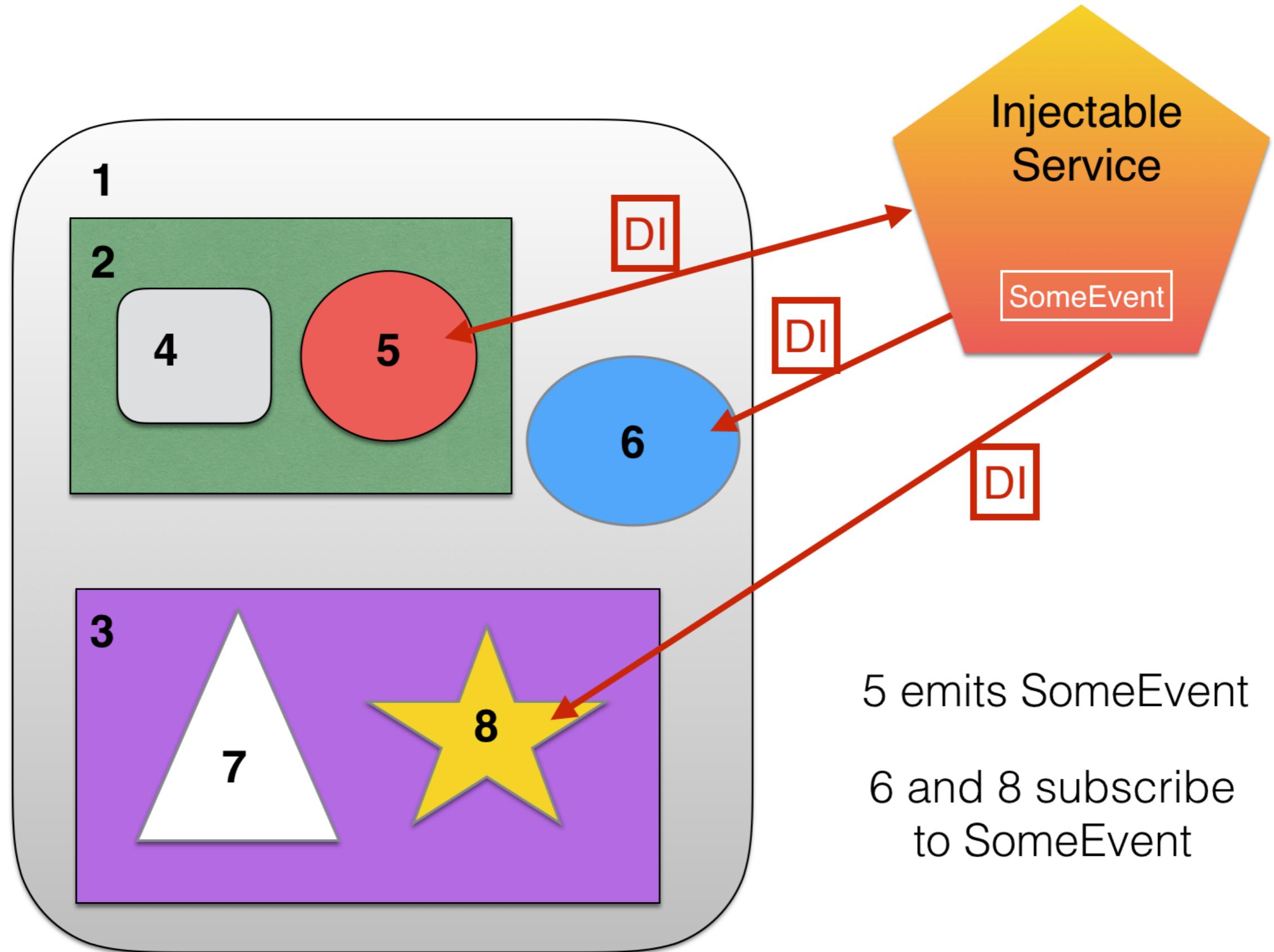
Parent as a mediator



Walkthrough 4.3

- Review the code in the mediator-parent directory
- **ng serve mediator-parent -o**
- Price quotes are generated by the PriceQuoter component
- When the user clicks the Buy button, the data is passed to the OrderComponent via the mediator (parent)

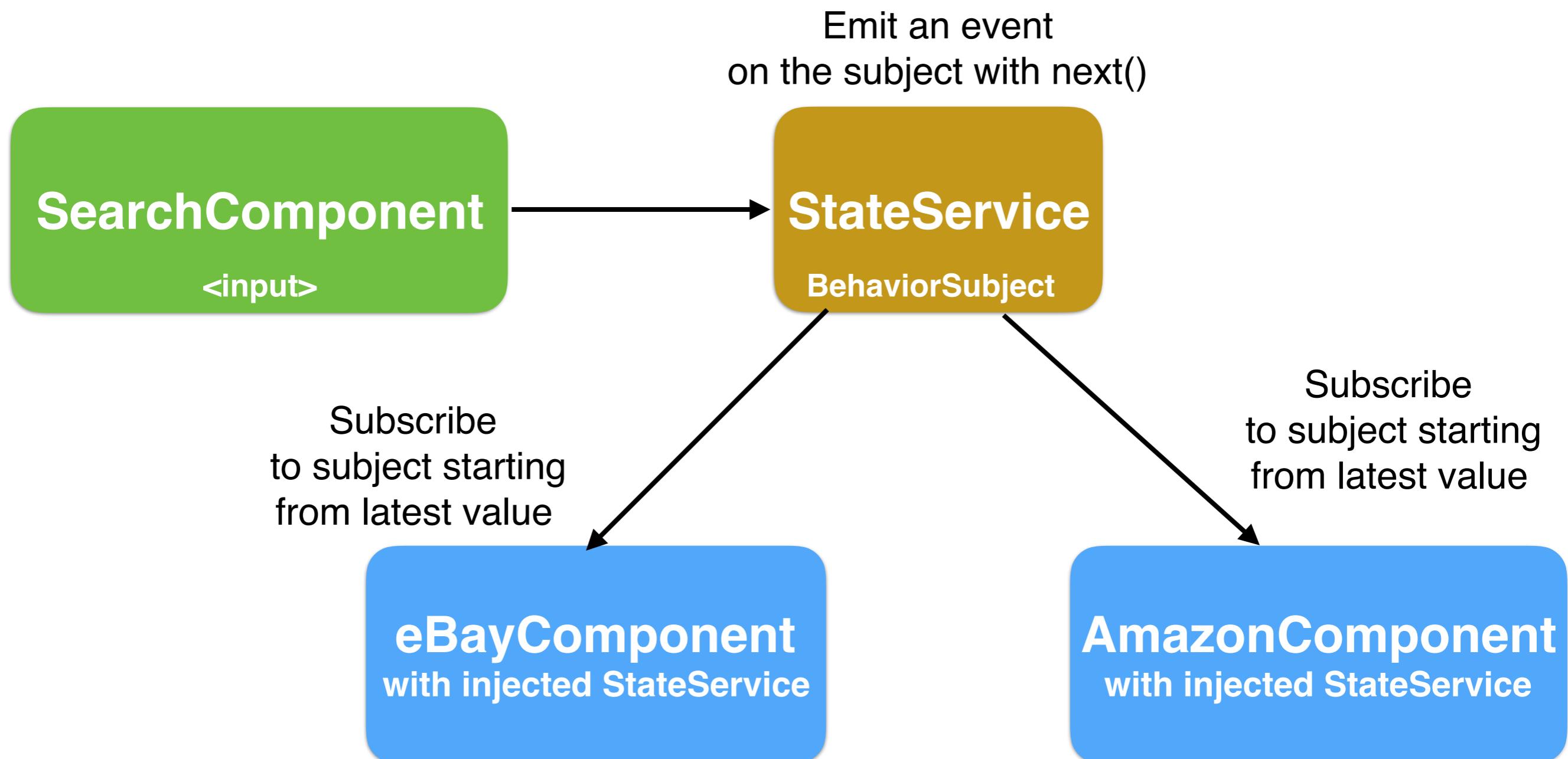
Injectable service as a mediator



BehaviorSubject

- Includes an observable and observer(s)
- Remembers the last emitted value
- Every new subscriber gets the initial or last emitted value
- The last emitted value could represent your app state

Inter-component communications with BehaviorSubject



```
export class StateService{  
  
    stateEvent: BehaviorSubject<string> = new BehaviorSubject('');  
  
    set searchCriteria(value: string) {  
  
        this.stateEvent.next(value);  
    }  
}
```

```
export class SearchComponent {  
  
    searchInput: FormControl;  
  
    constructor(private state: StateService){  
        this.searchInput = new FormControl('');  
  
        this.searchInput.valueChanges  
            .debounceTime(300)  
            .subscribe(searchValue =>  
                this.state.searchCriteria = searchValue);  
    }  
}
```

```
export class AmazonComponent implements OnDestroy{  
  
    searchFor: string;  
    subscription: Subscription;  
  
    constructor(private state: StateService){  
  
        this.subscription = state.getState()  
            .subscribe(event => this.searchFor = event);  
    }  
  
    ngOnDestroy() {  
        this.subscription.unsubscribe(); // a must  
    }  
}
```

EbayComponent uses async pipe instead of explicit subscription

Demo

- Review the code in the mediator-service directory
- `ng serve mediator-service -o`
- Enter any text in the search field of the top level component. The eBay component gets it from the state service via subscription
- Navigate to amazon. The AmazonComponent gets the value entered in the SearchComponent
- Add more text in the SearchComponent. AmazonComponent gets it via subscription
- Navigate back to eBay. The EbayComponent gets the latest text (i.e. state) from the service.

What have we learned

- Components receive data from their parents via `@Input` properties
- Components send data to their parents by emitting events via `@Output` properties
- 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 Development with TypeScript

Unit 6: Reactive programming and Observable Streams

In this unit

- Intro to reactive programming
- Events as observables
- Observables in the router and forms
- Observable HTTP requests
- Shared subscriptions

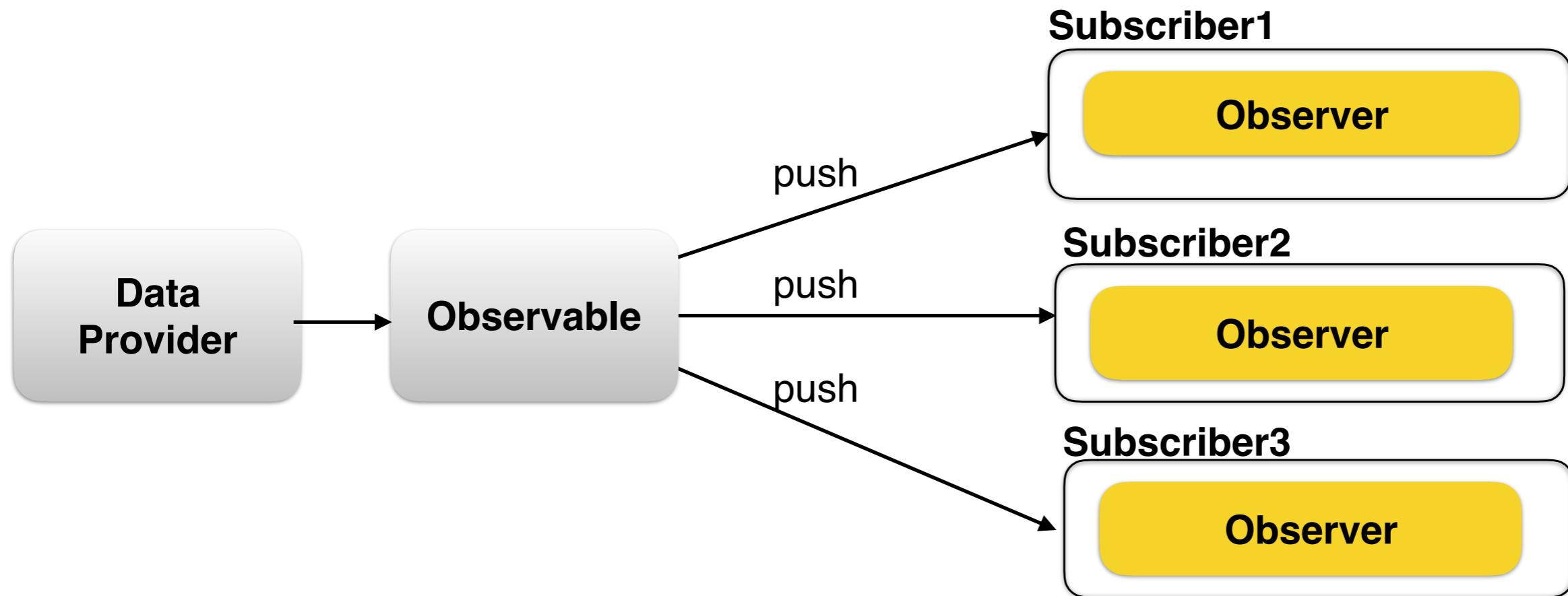
RxJS Essentials

Watch my video training “RxJS Essentials” on
[safaribooksonline.com](https://www.safaribooksonline.com)

Main RxJS players

- **Observable** - data stream that pushes data over time
- **Observer** - consumer of an observable stream
- **Subscriber** - connects observer with observable
- **Operator** - en-route data transformation

Pushing data to subscribers



Subscribe to messages from Observable and handle them in the Observer

An Observable can:

- Subscribe/unsubscribe to its data stream
- Emit the next value to the observer
- Notify the observer about errors
- Inform the observer about the stream completion

An Observer can provide:

- A function to handle the next value from the stream
- A function to handle errors
- A function to handle end-of-stream

Creating an Observable

- **from(myArray)** - converts an array or Iterable into Observable
- **Observable.create(myObserver)** - returns an Observable that can invoke methods on myObserver
- **of(1,2,3)** - turns a sequence of values into an Observable
- **fromEvent(myInput, 'keyup')** - converts an event into an Observable
- **interval(1000)** - returns emits an integer every second

Demo Observable.create()

[https://codepen.io/yfain/pen/NMjgyp?
editors=1111](https://codepen.io/yfain/pen/NMjgyp?editors=1111)

An Operator



<http://reactivex.io/rxjs/manual/overview.html#categories-of-operators>

Filter



```
observableProducts  
.filter(product => product.price < 8)
```

dot-chainable
operators
are deprecated

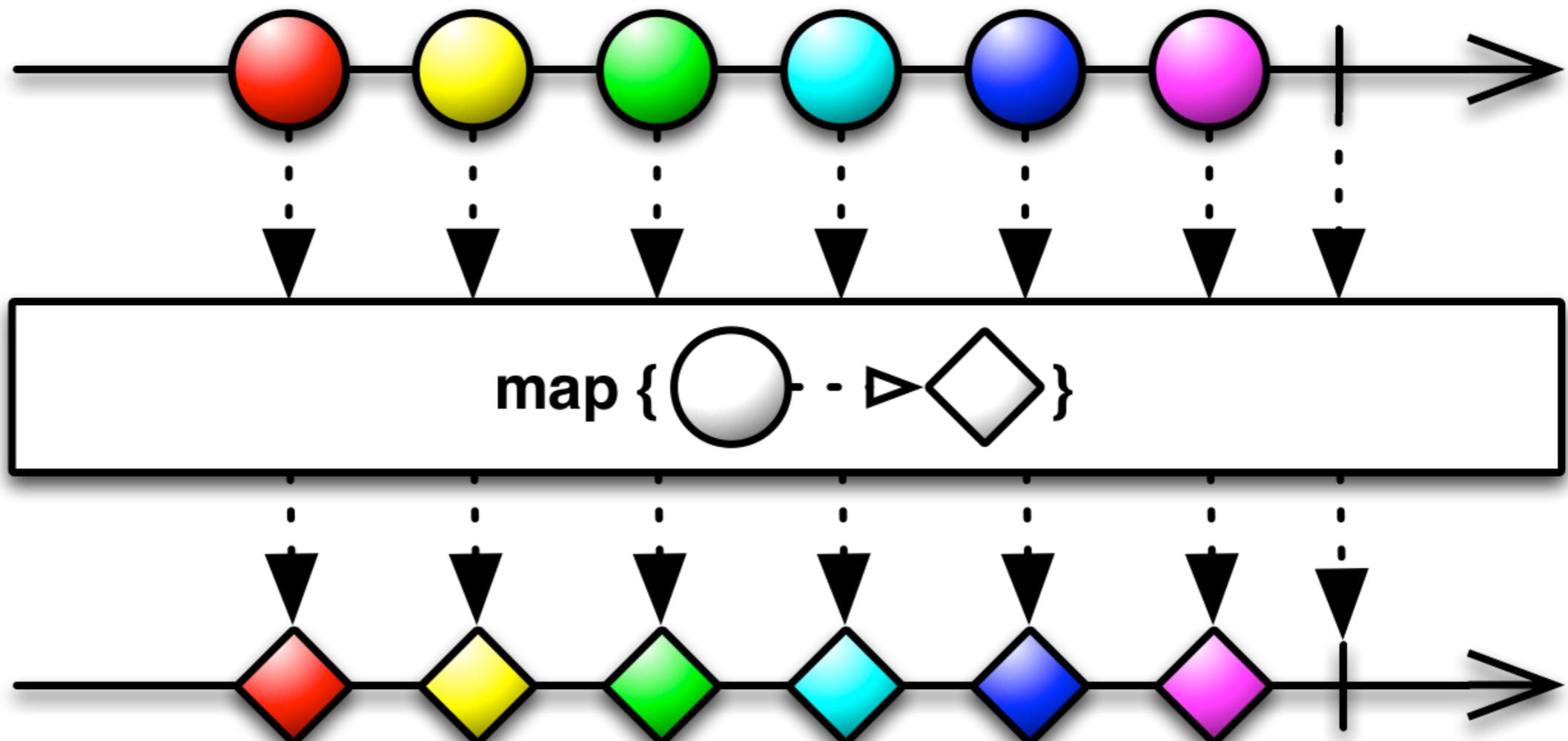
higher-order function

pure function

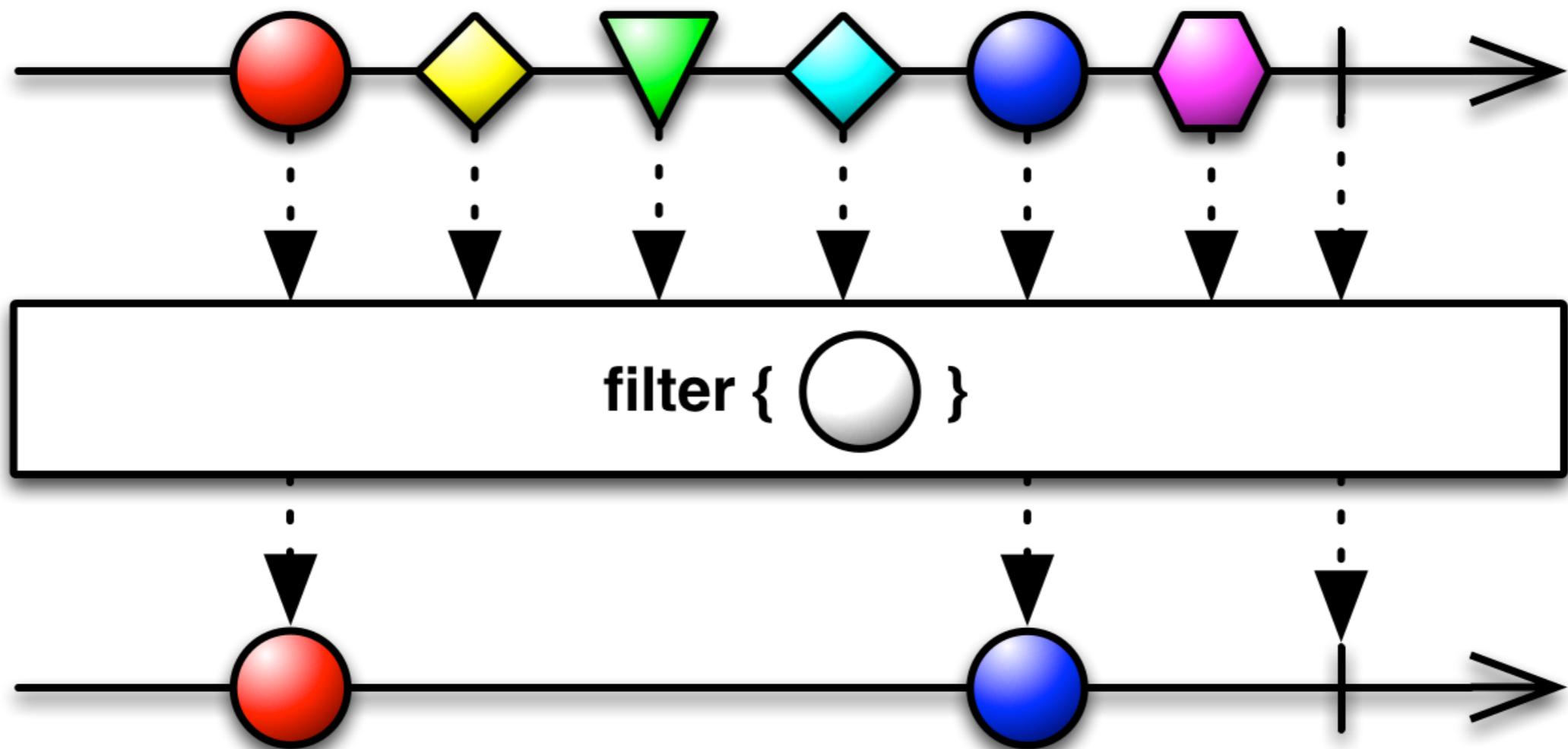
Marble Diagrams

<http://rxmarbles.com>

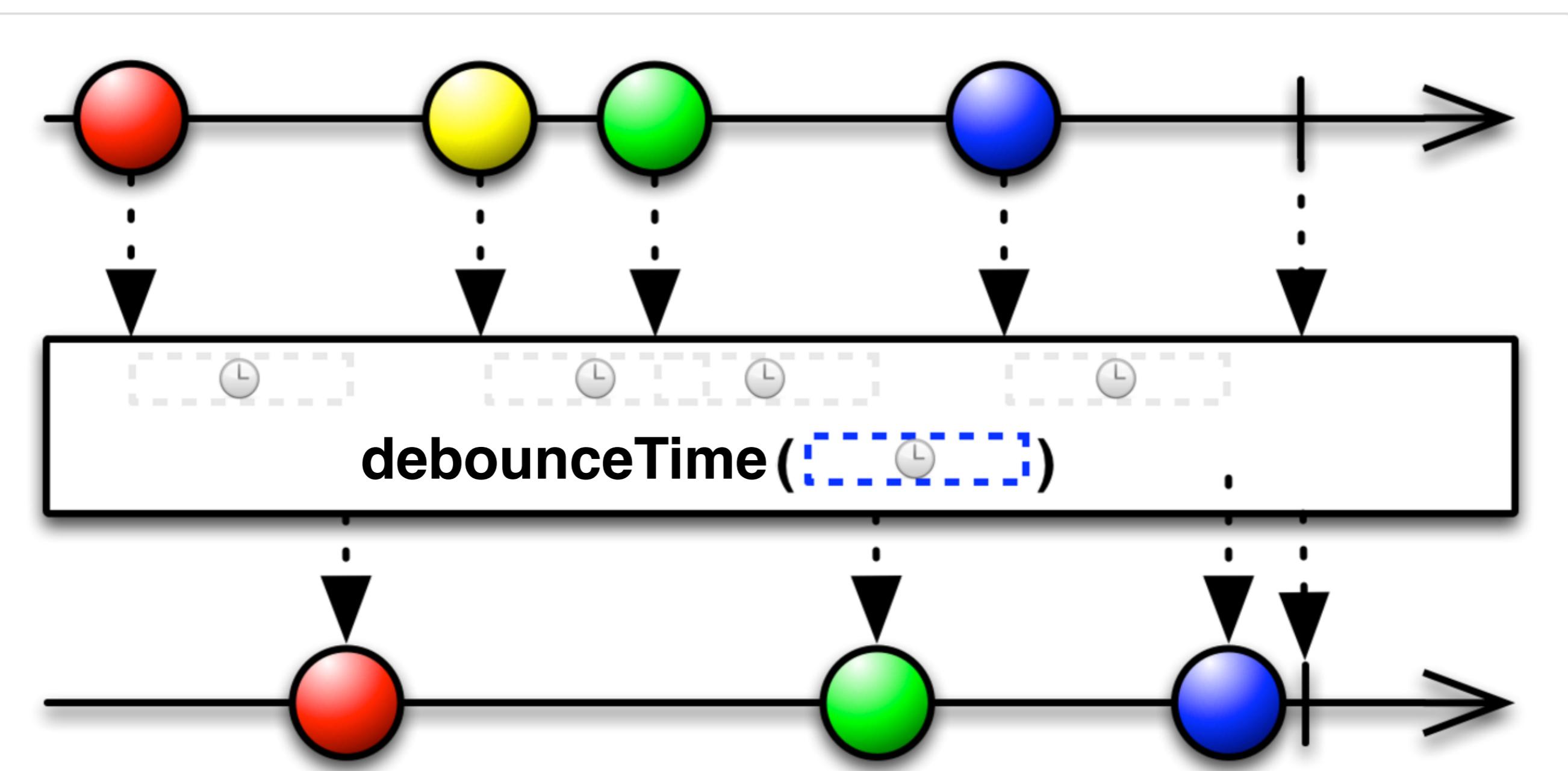
map



filter



debounceTime



Importing RxJS operators

- RxJS 5: Importing from rxjs/add/operator patches
Observable.prototype

```
import 'rxjs/add/operator/debounceTime';

this.searchInput.valueChanges
  .debounceTime(500)
  .subscribe(stock => this.getStockQuoteFromServer(stock));
```



Importing RxJS operators

- RxJS 5: Importing from rxjs/add/operator patches
Observable.prototype

```
import 'rxjs/add/operator/debounceTime';

this.searchInput.valueChanges
  .debounceTime(500)
  .subscribe(stock => this.getStockQuoteFromServer(stock));
```

- RxJS 6: Pipeable operators are pure functions and are tree-shakeable

```
import { debounceTime } from 'rxjs/operators';

this.searchInput.valueChanges
  .pipe(debounceTime(500))
  .subscribe(stock => this.getStockQuoteFromServer(stock));
}
```

Walkthrough 6.1

- Open the directory `unit6/observables` in your IDE
- `npm i`
- This app subscribes to observable events from the FormControl
Review the code in the directory `formcontrol`.
- `ng serve formcontrol -o`
- Open the browser console
- Enter any stock symbol to see the generated price on the console
- Because of `debounceTime()`, the FormControl's observable emits when the interval between entering characters is more than 500 milliseconds

Let's have a beer

```
let beers = [  
    {name: "Stella", country: "Belgium", price: 9.50},  
    {name: "Sam Adams", country: "USA", price: 8.50},  
    {name: "Bud Light", country: "USA", price: 6.50},  
    {name: "Brooklyn Lager", country: "USA", price: 8.00},  
    {name: "Sapporo", country: "Japan", price: 7.50}  
];
```

Observable Beer

No streaming yet



```
const { from } = rxjs;
const { filter, map } = rxjs.operators;

observableBeers$ = from(beers)
  .pipe(
    filter(beer => beer.price < 8),
    map(beer => beer.name + ": $" + beer.price)
  );
```

Streaming begins



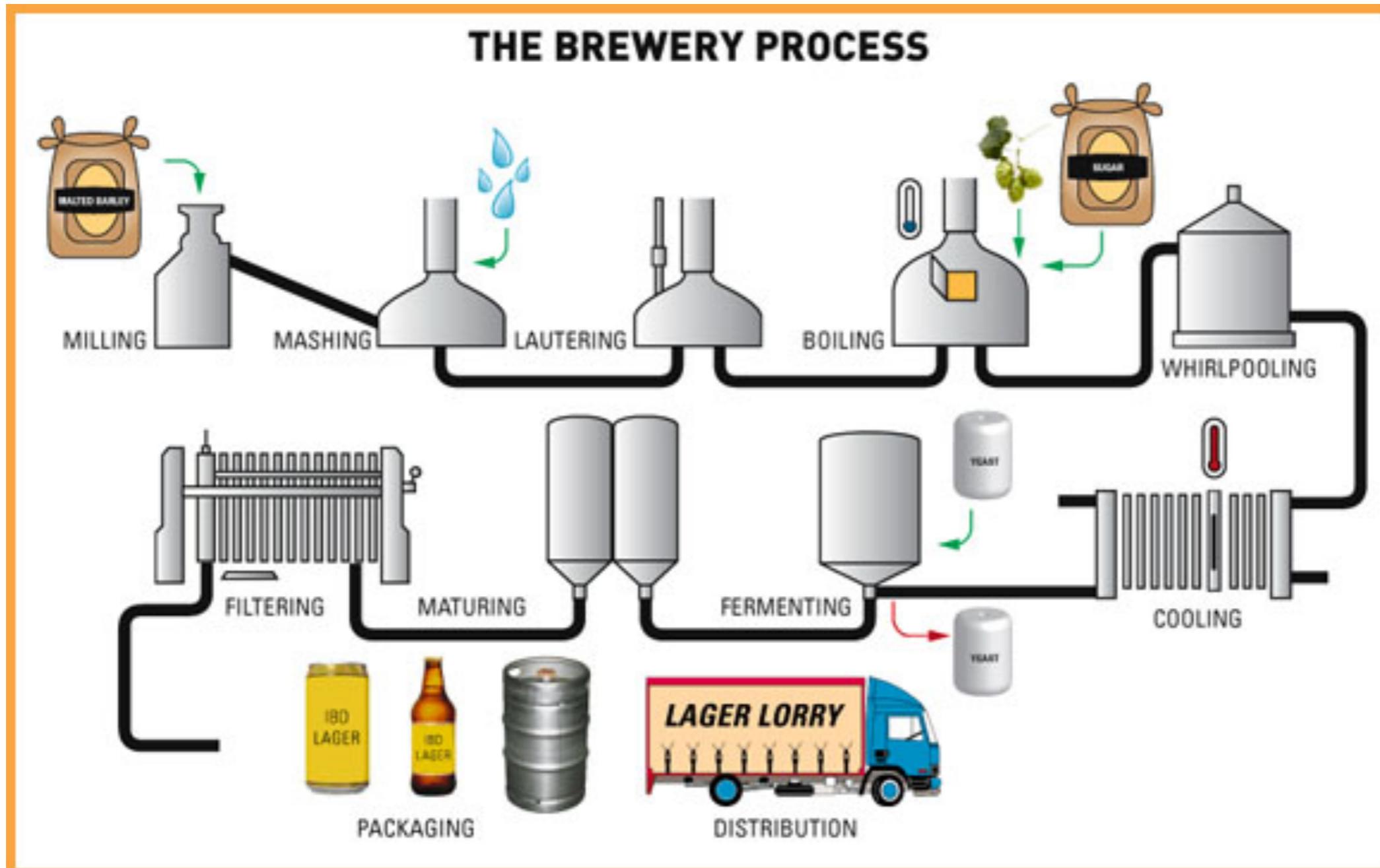
```
observableBeers$
  .subscribe(
    beer => console.log(beer),
    err  => console.error(err),
    ()   => console.log("Streaming is over")
  );
```

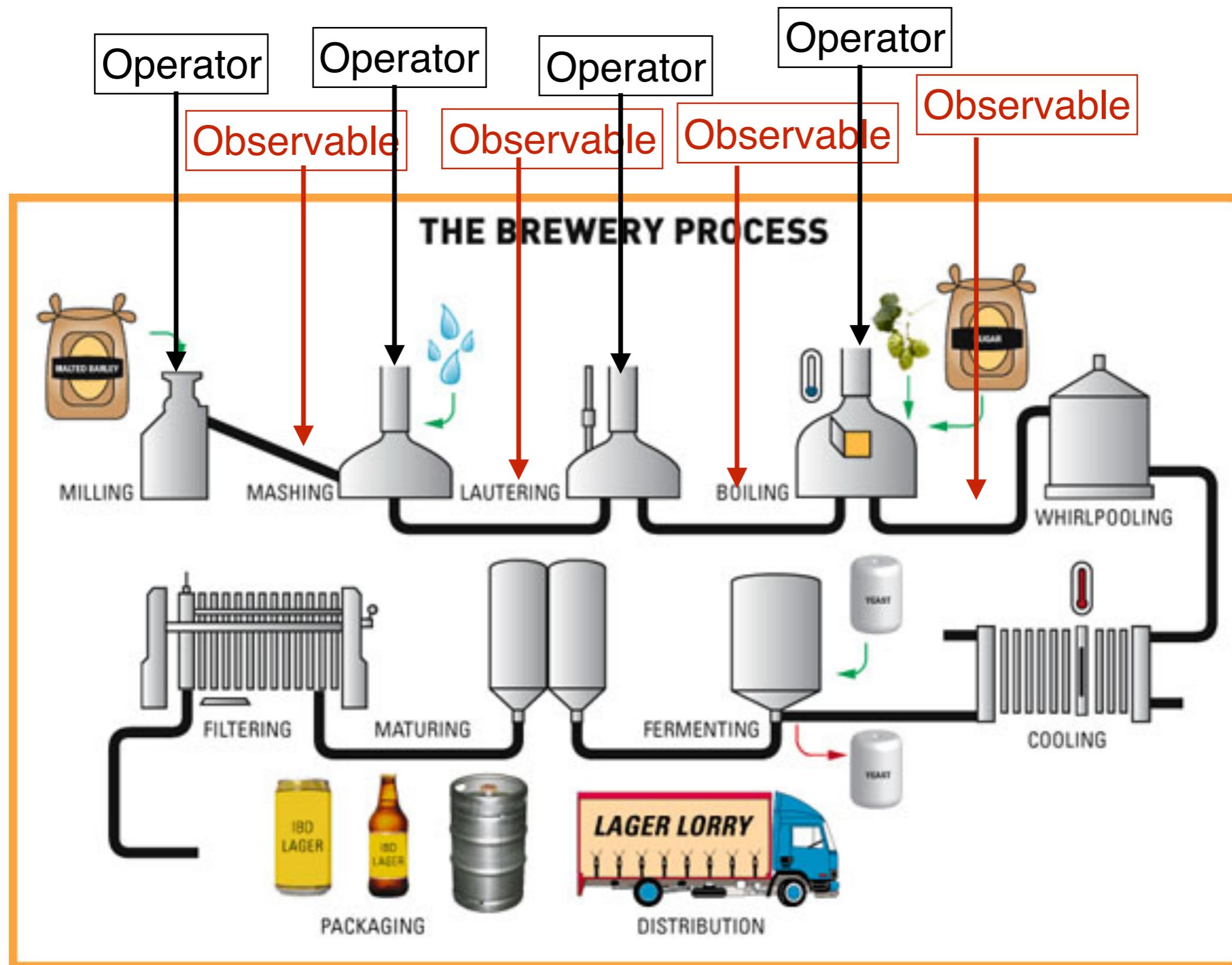
Demo

from() + filter + map()

[https://codepen.io/yfain/pen/OZmgmM?
editors=1012](https://codepen.io/yfain/pen/OZmgmM?editors=1012)

RX: the data moves across your algorithm





Error-handling operators

- `error()` is invoked by the Observable on the Observer.
- `catchError()` - intercepts the error in the subscriber before the observer gets it. Replaces RxJS 5 `catch()`.
- `retry(n)` - retry immediately up to n times
- `retryWhen(fn)` - retries as prescribed by the argument function

Failover with catchError()

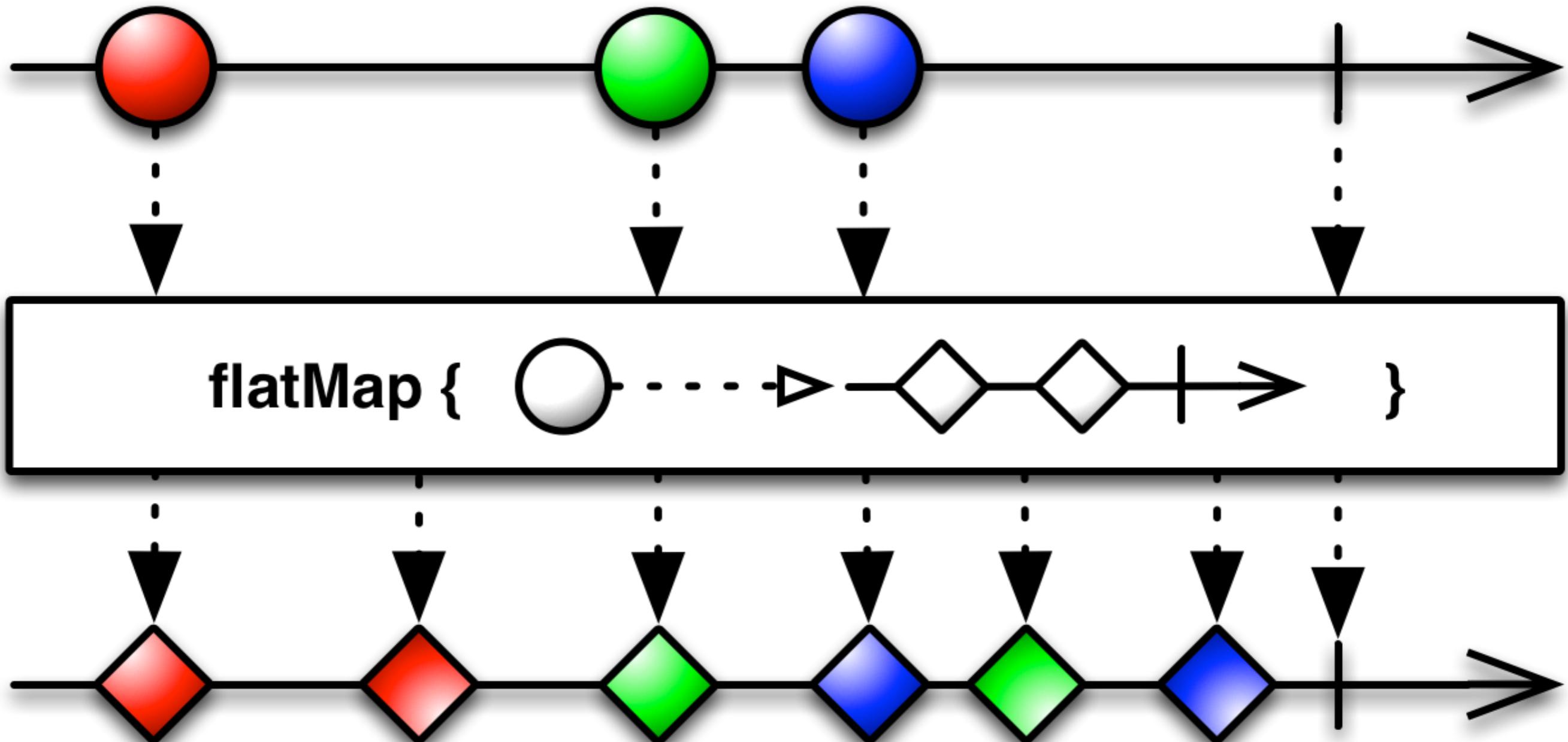
F
a
i
l
o
v
e
r

```
function getData(): Observable {...} // data source 1
function getCachedData(): Observable {...} // data source 2
function getDataFromAnotherService(): Observable {...} // data source 3

getData()
  .pipe(
    .catchError(err => {
      if (err.status === 500){
        console.error("Switching to streaming cached beer data");
        return getCachedData();
      } else{
        console.error("Switching to another beer service");
        return getDataFromAnotherService();
      }
    }),
    map(beer => beer.name + ", " + beer.country)
  )
  .subscribe(
    beer => console.log("Subscriber got " + beer)
  );
```

See it in CodePen: <https://codepen.io/yfain/pen/OZgpZx?editors=1011>

flatMap



Operator flatMap

- Handles every value emitted by an observable as another observable
- Auto-subscribes to the internal observable and unwraps it

```
function getDrinks() {  
  
    let beers = from([  
        {name: "Stella", country: "Belgium", price: 9.50},  
        {name: "Sam Adams", country: "USA", price: 8.50},  
        {name: "Bud Light", country: "USA", price: 6.50}  
    ]);  
  
    let softDrinks = from([  
        {name: "Coca Cola", country: "USA", price: 1.50},  
        {name: "Fanta", country: "USA", price: 1.50},  
        {name: "Lemonade", country: "France", price: 2.50}  
    ]);  
  
    return Observable.create( observer => {  
        observer.next(beers);           // pushing the beer pallet (observable)  
        observer.next(softDrinks);     // pushing the soft drinks pallet (observable)  
    }  
);  
  
getDrinks()  
    .pipe(  
        .flatMap(drinks => drinks)      // unloading drinks from pallets  
    )  
    .subscribe(  
        drink => console.log("Subscriber got " + drink.name + ": " + drink.price )  
    );  
}
```

See it in CodePen: <https://codepen.io/yfain/pen/MGopzx?editors=1011>

@yfain

The side-effect operator tap()

- Can perform a side effect for every emitted value
- Returns an observable that is identical to the source
- Replaces the deprecated do () operator

```
myObservable$  
  .pipe(  
    tap(beer => console.log(`Before: ${beer}`)),  
    map(beer => `${beer.name}, ${beer.country}`),  
    tap(beer => console.log(`After: ${beer}`))  
  )  
  .subscribe(...);
```

Observables in Angular

- Router
- Reactive Forms
- Handling HTTP responses
- AsyncPipe

Observables in Forms

FormControl

- valueChanges - the value of the form control changes

```
this.searchInput.valueChanges.subscribe(...);
```

- statusChanges - status of the form control (valid/invalid)

```
this.password.statusChanges.subscribe(...);
```

Observable Events

```
@Component({
  selector: "app",
  template: `
    <h2>Observable events demo</h2>
    <input type="text" placeholder="Enter stock" [FormControl]="searchInput">
  `
})
class AppComponent implements OnInit {

  searchInput: FormControl;

  constructor(){
    this.searchInput = new FormControl('');
  }

  ngOnInit(){
    this.searchInput.valueChanges
      .pipe(
        debounceTime(500)
      )
      .subscribe(stock => this.getStockQuoteFromServer(stock));
  }

  getStockQuoteFromServer(stock) {
    console.log(`The price of ${stock} is ${100*Math.random().toFixed(4)}`);
  }
}
```

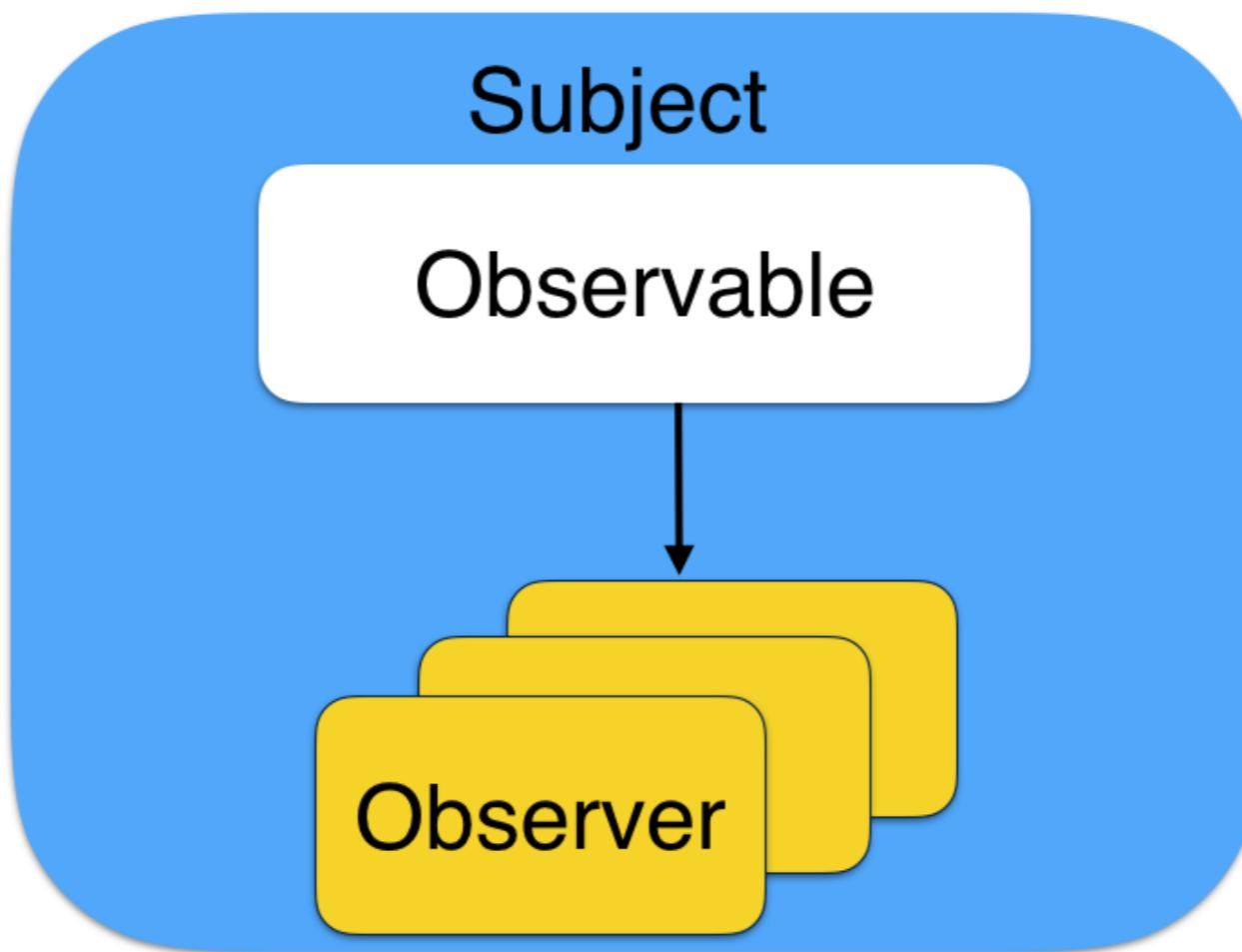
The diagram illustrates the flow of observable events. A red arrow points from the declaration of `this.searchInput` in the template to its definition in the constructor. Another red arrow points from the `valueChanges` method in the `ngOnInit()` function to a red-bordered box containing the word **Observable**, indicating that the `valueChanges` method returns an observable object.

Demo

- **ng serve formcontrol -o**
- Open the browser console
- Enter any stock symbol to see the generated price on the console
- Because of debounceTime() , the FormControl's observable emits when the interval between entering characters is more than 500 milliseconds

Multi-casting with Subject

Rx.Subject is both an observable and observer



```
const mySubject = new Subject();
...
subscription1 = mySubject.subscribe(...);
subscription2 = mySubject.subscribe(...);
...
mySubject.next(123); // each subscriber gets 123
```

BehaviorSubject

- Includes an observable and observer(s)
- Remembers the last emitted value
- Every new subscriber gets the initial or last emitted value
- The last emitted value could represent your app state

Revisit the mediator service example from unit 4

HttpClient and Observables

```
@Component({
  selector: 'http-client',
  template: `<h1>All Products</h1>
<ul>
  <li *ngFor="let product of products">
    {{product.title}}
  </li>
</ul>
`)
class AppComponent implements OnInit{

  products: Array = [];

  constructor(private http: HttpClient) {}

  ngOnInit() {
    this.http.get<Product[]>('/products')
      .subscribe(
        data => {
          this.products=data;
        },
        err =>
          console.log("Can't get products. Error code: %s, URL: %s ",
                     err.status, err.url),
        () => console.log('Product(s) are retrieved')
      );
  }
}
```

The diagram illustrates the flow of data from the template to the component's logic. A red arrow points from the template's `ngFor` loop to the `product.title` binding in the `ngFor` expression. Another red arrow points from the `products` array declaration in the component to the `data` parameter in the `subscribe` method. A vertical red box labeled "Observer" on its left side has three red arrows pointing to the `err`, `data`, and `complete` handlers in the `subscribe` method.

Async pipe

Auto-subscribes/unsubscribes and unwraps the Observable in the component's template

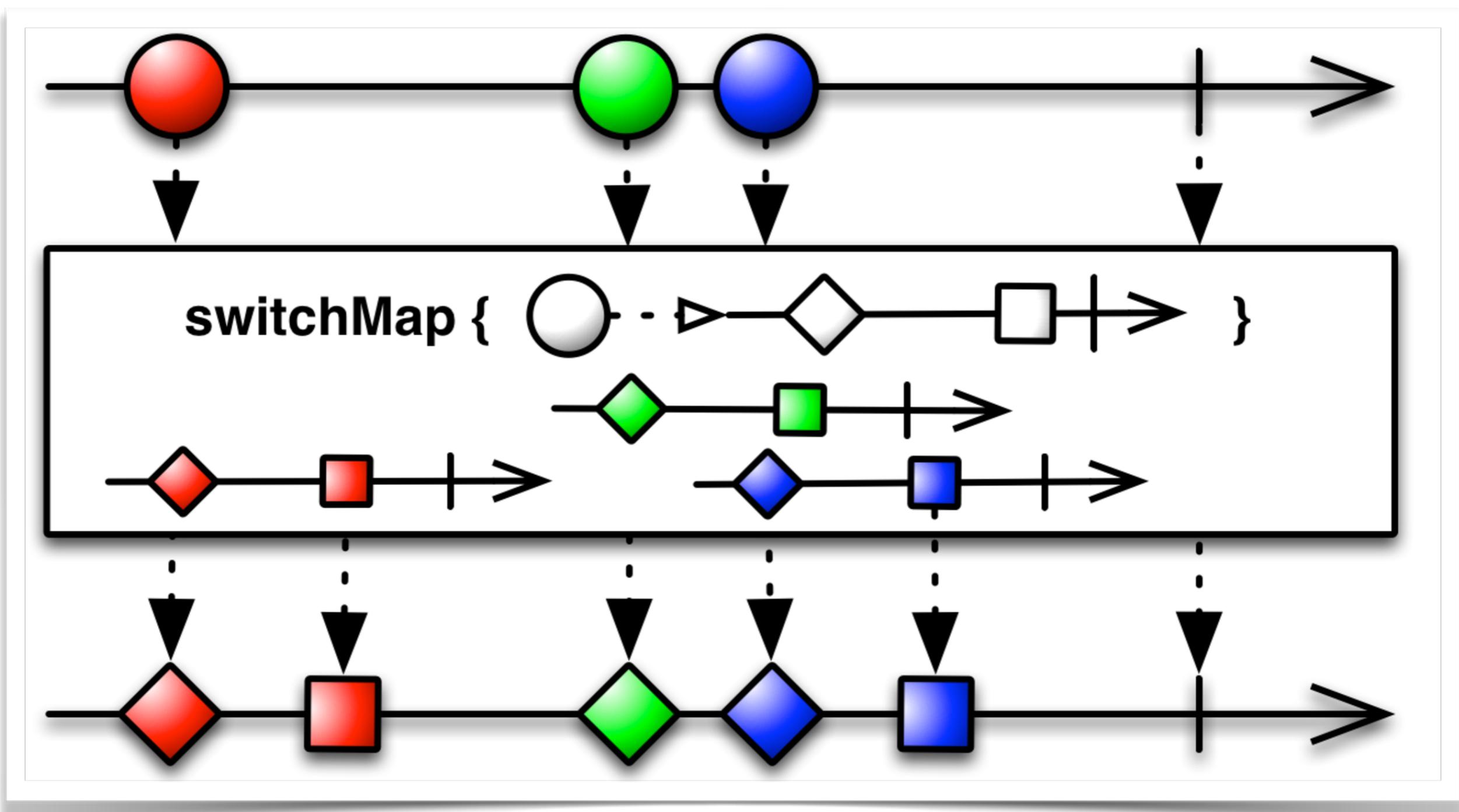
```
@Component({
  selector: 'http-client',
  template: `<h1>All Products</h1>
<ul>
  <li *ngFor="let product of products$ | async">
    {{product.title}}
  </li>
</ul>
{{errorMessage}}
`})
class AppComponent implements OnInit{
  products$: Observable<Product[]>;
  errorMessage: string;

  constructor(private http: HttpClient) {}

  ngOnInit() {
    this.products$ = this.http.get<Product[]>('/products')
      .pipe(
        catchError( err => {
          this.errorMessage = `Can't get product details from ${err.url},
                           error ${err.status}`;
          return Observable.empty();
        })
      )
  }
}
```

```
export interface Product {
  id: number;
  title: string;
  price: number
}
```

The switchMap operator



<http://reactivex.io/documentation/operators/images/switchMap.png>

@yfain

Killing HTTP requests with switchMap

```
@Component({
  selector: "app-root",
  template: `
    <h2>Observable weather</h2>
    <input type="text" placeholder="Enter city" [formControl]="searchInput">
    <h3>{{temperature}}</h3>
  `
})
export class AppComponent {
  private baseWeatherURL: string = 'http://api.openweathermap.org/data/2.5/weather?q=';
  private urlSuffix: string = "&units=imperial&appid=12345"; // use the real appid

  searchInput: FormControl = new FormControl();
  temperature: string;

  constructor(private http: HttpClient) {
    this.searchInput.valueChanges
      .pipe(
        switchMap(city => this.getWeather(city))
      )
      .subscribe(
        res => {
          this.temperature =
            `Current temperature is ${res['main'].temp}F, +
             humidity: ${res['main'].humidity}%`;
        },
        err => console.log(`Can't get weather. Error code: ${err.status}, URL: ${err.url}`)
      );
  }

  getWeather(city: string): Observable<Array<string>> {
    return this.http.get(this.baseWeatherURL + city + this.urlSuffix)
      .pipe(catchError( err => {
        if (err.status === 404) {
          console.log(`City ${city} not found`);
          return Observable.empty(); // empty observable
        }
      }));
  }
}
```

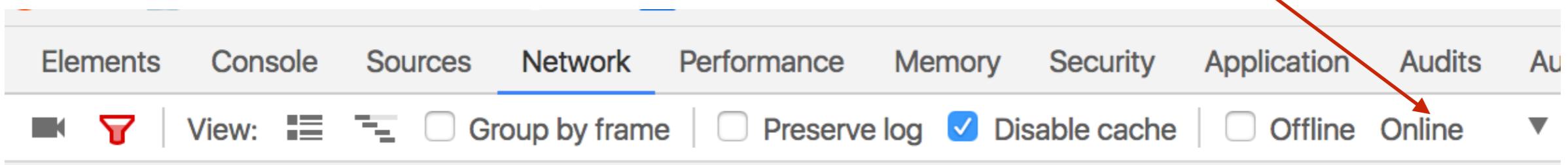
The diagram illustrates the execution flow of the `switchMap` operator. It shows two boxes: "Outer Obs." containing the observable `this.searchInput.valueChanges`, and "Inner Obs." containing the observable `this.getWeather(city)`. A red arrow points from the "Outer Obs." box to the `switchMap` call, and another red arrow points from the `switchMap` call to the "Inner Obs." box. A third red arrow points from the "Inner Obs." box back to the "Outer Obs." box, indicating that each value emitted by the inner observable becomes the next value for the outer observable.

Walkthrough 6.2 (start)

- `ng serve weather -o`
- Open the Network tab in Chrome Dev Tools
- Enter the name of a city and the app renders the weather
- The Network tab shows multiple HTTP requests/responses as you type
- Open the Console tab - if the city was not found, the 404 error was handled by `catch()`

Walkthrough 6.2 (end)

- Open Developer Tools in the Network tabs and emulate slow connection by changing the Online mode to Slow 3G



- Start typing the name of the city
- Note how the app sends several HTTP requests and all of them were cancelled except the last one.

Observables in the Router

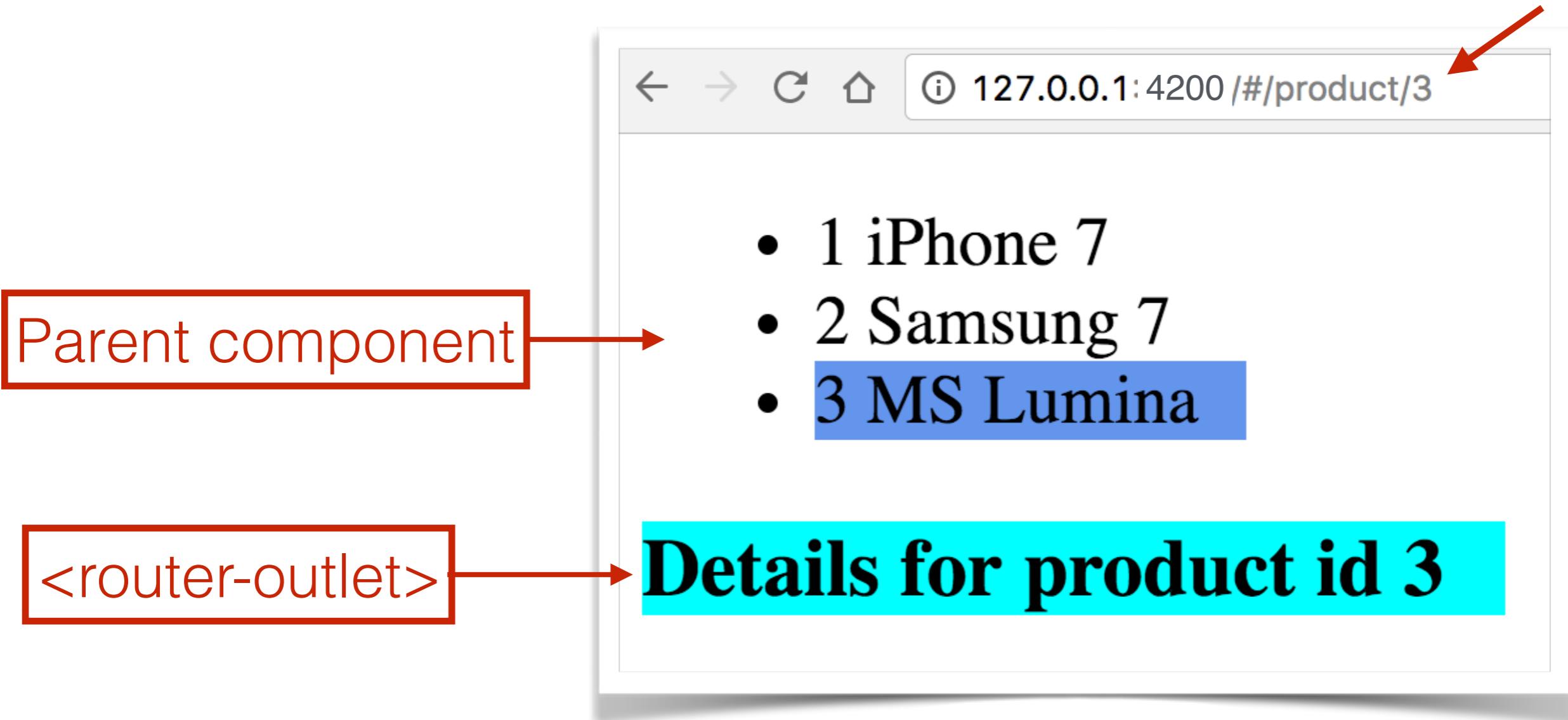
ActivatedRoute (fragment)

```
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;  
    ...  
}
```

Receiving params in ActivatedRoute

- Inject ActivatedRoute into a component to receive the route params
- Use ActivatedRoute.snapshot to get params once
- Use ActivatedRoute.paramMap.subscribe() for receiving changing params over time

Master-Detail with Router



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

class AppComponent {
  ...
  constructor(private _router: Router){}
  onSelect(prod: Product): void {
    this._router.navigate(['/productDetail', prod.id]);
  }
}
```



```
export class ProductDetailComponent {
  productID: string;
  constructor(private route: ActivatedRoute) {
    this.route.paramMap
      .subscribe(params => this.productID = params.get('id'));
  }
}
```

What have we learned

- An observable emits data over time
- An observer contains code to handle data
- RxJS has lots of pipeable operators that can transform observable data en route
- Angular offers observable streams in Forms, Router, HttpClient
- The switchMap operator allows an app to abandon unwanted HTTP responses

Angular Development with TypeScript

Unit 7: Component lifecycle and change detection

In this unit

- Change detection
- Component lifecycle

Change Detection

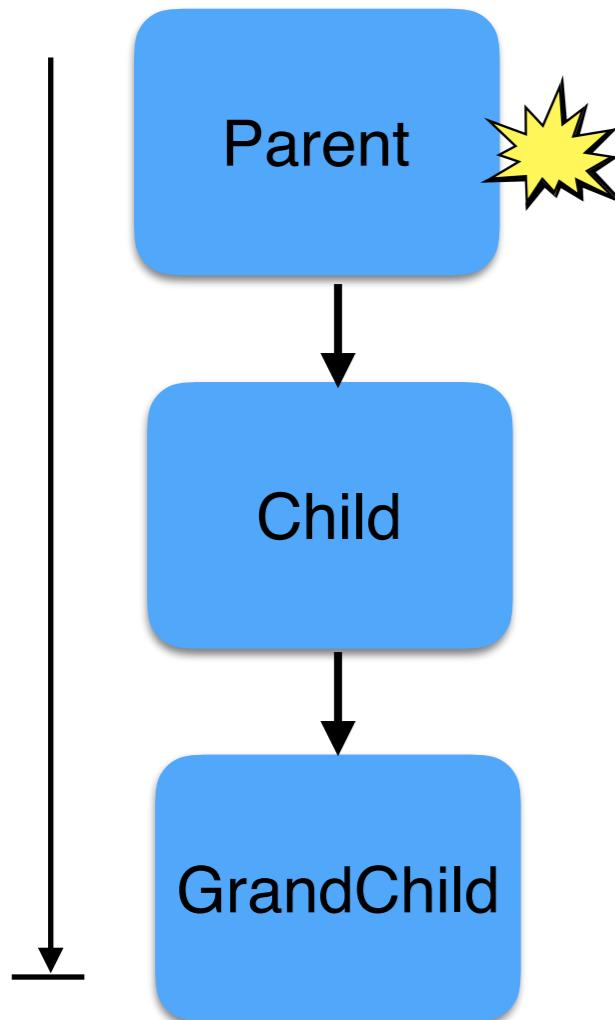
Change Detection (CD)

- CD is about updating the view whenever the underlying model changes
- The CD mechanism patches asynchronous browser's actions:
 - `addEventListener()`
 - `setTimeout()`, `setInterval()`
 - HTTP and WebSocket requests
- CD is implemented in the library `zone.js`

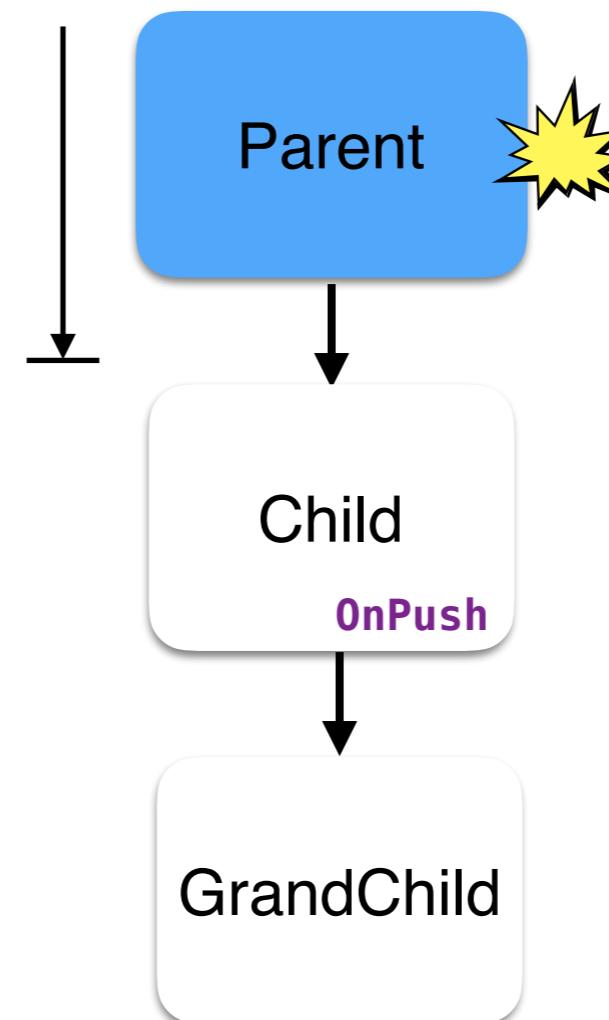
CD strategies: Default vs OnPush

changeDetection: ChangeDetectionStrategy.**OnPush**

Default

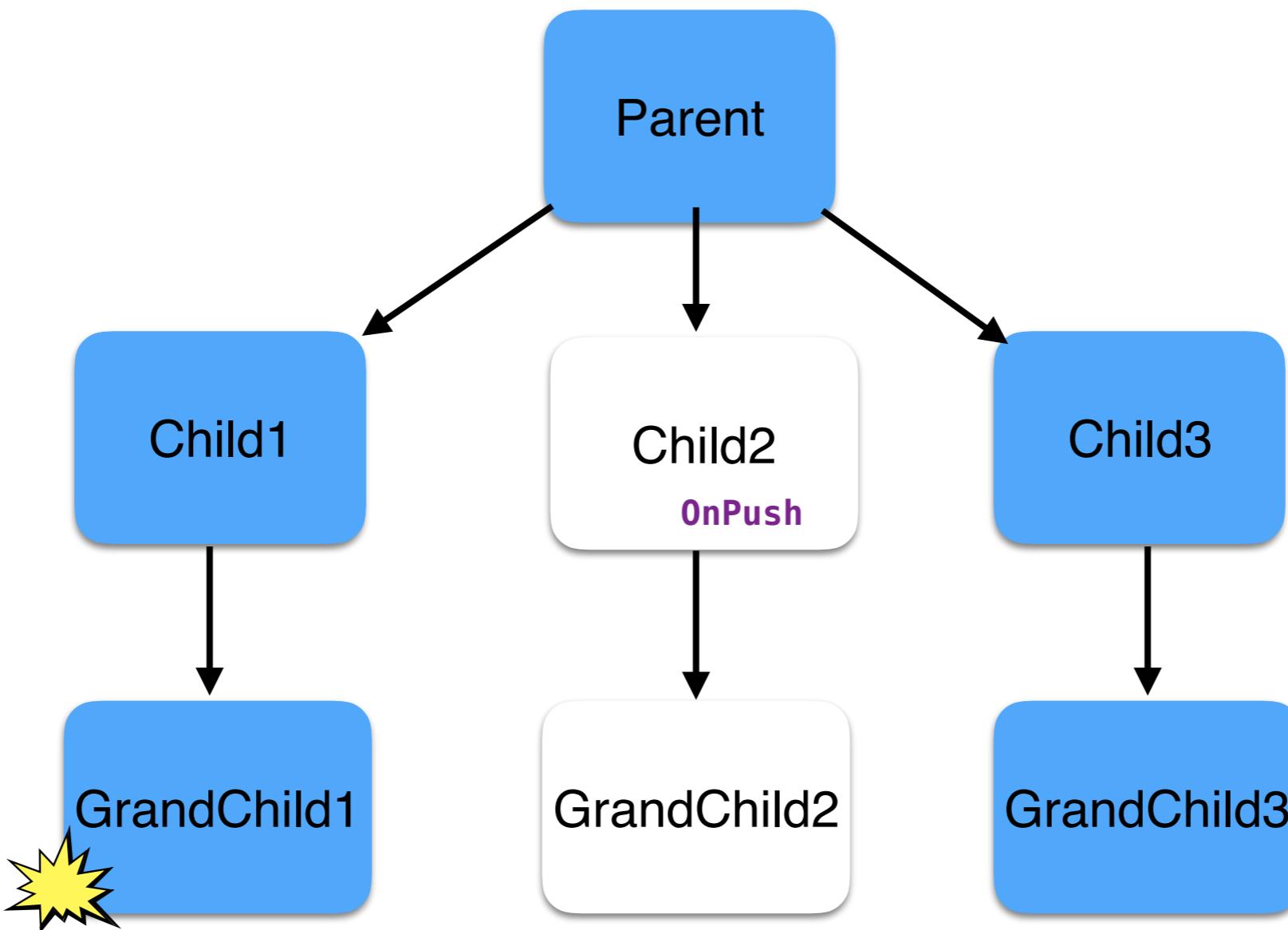


OnPush



With **OnPush** strategy the Zone won't check a component unless the reference of the value bound to its @Input properties change.

CD strategies: default vs OnPush



A change in the GrandChild1 will start the change detection pass in all blue components.

Prod vs Dev Mode

- In dev mode:
 - Change detector runs twice
 - Additional debug info is included in the compiled code
 - Additional warnings may be printed on the console
- Enable prod mode before bootstrapping the app:

```
import {enableProdMode} from '@angular/core';
...
enableProdMode();

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

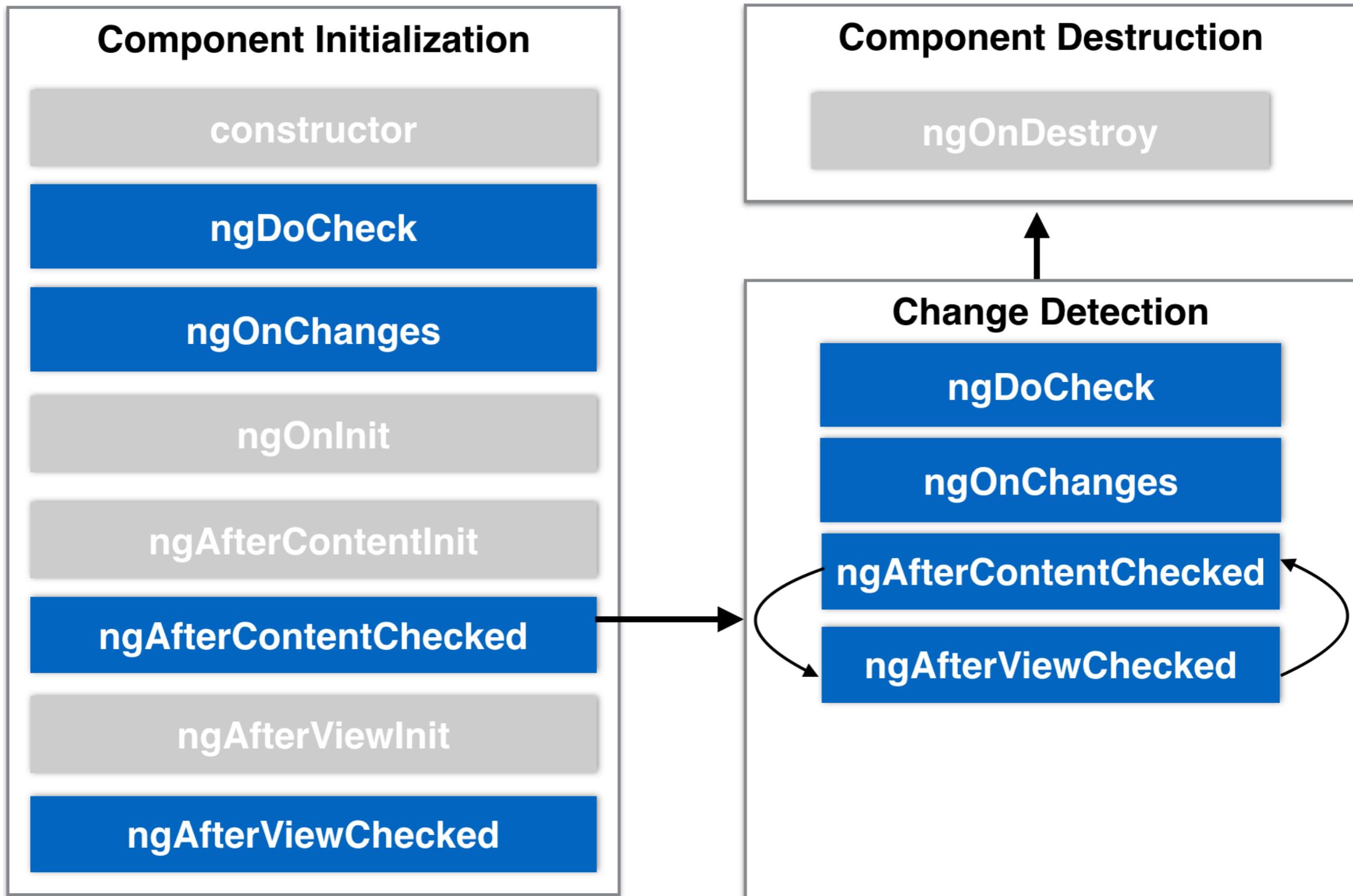
Component Lifecycle

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

Component 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 ngOnInit () is called, component properties are initialized
- ngOnInit () is often used for fetching data.
- Place the code that uses component's properties in ngOnInit () .

```
@Input() productId: number;  
  
constructor(private productService: ProductService) {}  
  
ngOnInit(){  
    this.product = this.productService.getProductById(this.productId);  
}
```

ngOnChanges() in child intercept input properties changes

```
@Component({
  selector: 'app',
  styles: ['.parent {background: lightblue}'],
  template:
    <div class="parent">
      <h2>Parent</h2>
      <div>Greeting: <input type="text" [(ngModel)]="myGreeting"></div>
      <div>User name: <input type="text" [(ngModel)]="myUser.name"></div>

      <child [greeting]="myGreeting" [user]="myUser"></child>
    </div>
    `

})
class AppComponent {
  myGreeting = 'Hello';
  myUser: {name: string} = {name: 'John'};
}
```

Parent

ngOnChanges() in child intercept input properties changes

```
@Component({
  selector: 'app',
  styles: ['.parent {background: lightblue}'],
  template:
    <div class="parent">
      <h2>Parent</h2>
      <div>Greeting: <input type="text" [(ngModel)]="myGreeting"></div>
      <div>User name: <input type="text" [(ngModel)]="myUser.name"></div>

      <child [greeting]="myGreeting" [user]="myUser"></child>
    </div>
    `

})
class AppComponent {
  myGreeting = 'Hello';
  myUser: {name: string} = {name: 'John'};
}
```

Parent

```
@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>
    `

})
class ChildComponent implements OnChanges {
  @Input() greeting: string;
  @Input() user: {name: string};

  ngOnChanges(changes: {[key: string]: SimpleChange}) {
    console.log(JSON.stringify(changes, null, 2));
  }
}
```

Child

Mutable vs Immutable

- JavaScript primitives are immutable

```
let greeting = "Hello";    // allocates an address in memory for greeting  
greeting = "Hello Mary"; // allocates another address in memory
```

- JavaScript objects are mutable

```
let user = {name: "John"}; // allocates an address for the var user  
user.name = "Mary";       // the address of the var user remains the same
```

Walkthrough 7.1

- Open the project `unit7/lifecycle`
- Review the code in the directory `comp-lifecycle`. In the `ChildComponent` note the argument of `ngOnChanges()`
- `ng serve lifecycle -o`
- In the Dev Tools console, note the params received by `ngOnChanges()`
- Change the value in the field **Greeting** - `ngOnChanges()` is invoked (`greeting` is immutable)
- Change the value in the field **User name** - `ngOnChanges()` is not invoked (`user` is mutable)

What have we learned

- Lifecycle hooks methods are invoked at specific moments of the component life span
- The code for initial data fetch is typically placed in `ngOnInit()`
- Change detection is controlled by zone.js and is performed automatically

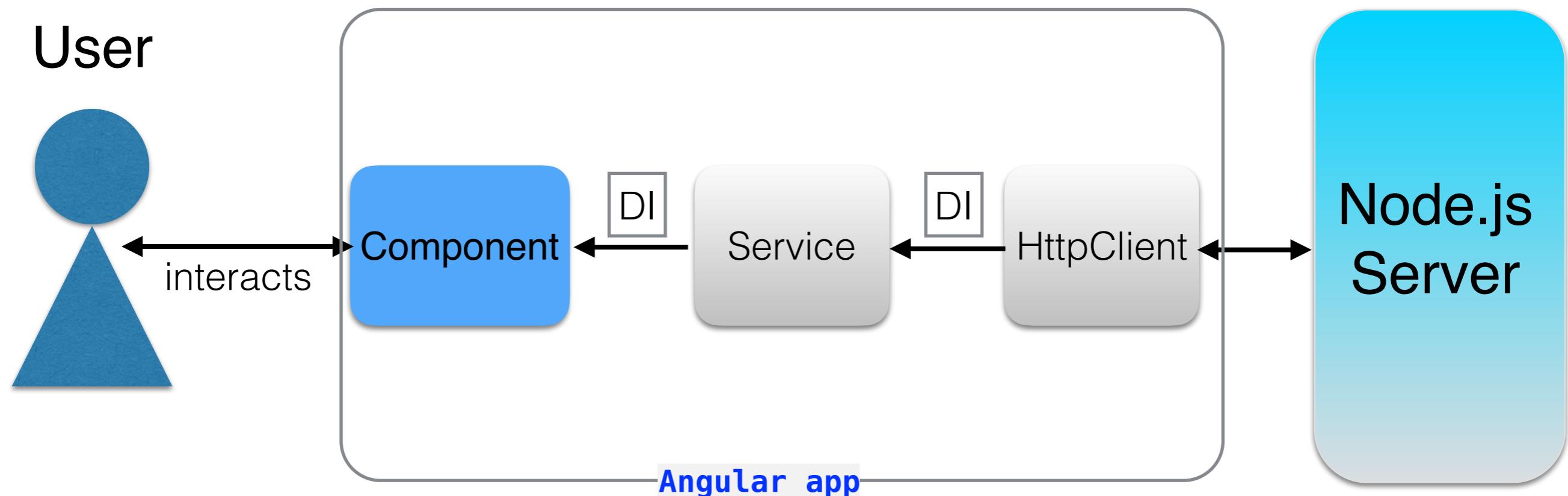
Angular Development with TypeScript

Unit 9: Communicating with a web server using HTTP

In this unit

- Creating a simple HTTP server with Node.js
- Working with the Angular HttpClient object
- Deploying apps on Node server with Angular CLI
- Bundling the auction and deploying it on the Node.js server

Let's develop an app



NodeJS can be replaced with Java, .Net, or other technology.

Angular and Node.js

Angular app

```
data: Observable<Product[]>;  
  
constructor(httpClient: HttpClient) {  
  this.data = httpClient  
    .get<Product[]>('/products');  
}
```

Node.js server

```
app.get('/products', (req, res) =>  
  res.send('Got a request for products');
```



The server

Node.js

package.json (server)

```
{  
  "name": "node-servers",  
  "description": "Node and Express samples",  
  "private": true,  
  "scripts": {  
    "devRest": "node build/my-rest-server"  
  },  
  "dependencies": {  
    "express": "^4.16.2"  
  },  
  "devDependencies": {  
    "@types/express": "^4.0.39",  
    "@types/node": "^6.0.57",  
    "typescript": "2.6.2"  
  }  
}
```

tsconfig.json for the server

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "outDir": "build",  
    "target": "es5"  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

import * as http from 'http';
becomes
var http = require('http');

Creating a RESTful API to the server

```
import * as express from "express";                                rest-server.ts

const app = express();

class Product {
    constructor(public id: number, public title: string, public price: number){}
}

const products = [
    new Product(0, "First Product", 24.99),
    new Product(1, "Second Product", 64.99),
    new Product(2, "Third Product", 74.99) ];

function getProducts(): Product[] {
    return products;
}

app.get('/', (req, res) => {
    res.send('The URL for products is http://localhost:8000/api/products');
});

app.get('/api/products', (req, res) => {
    res.json(getProducts());
});

function getProductById(productId: number): Product {
    return products.find(p => p.id === productId);
}

app.get('/api/products/:id', (req, res) => {
    res.json(getProductById(parseInt(req.params.id)));
});

const server = app.listen(8000, "localhost", () => {
    const {address, port} = server.address();
    console.log(`Listening on ${address}:${port}`);
});
```

Walkthrough 9.1

1.In your IDE open dir unit9/server

2.`npm i`

3.Compile the code into the directory **build**:

`tsc`

4.`node build/rest-server`

5.Open your browser at <http://localhost:8000/api/products>
You'll see 3 products in JSON format.

6.Change the URL to <http://localhost:8000/api/products/1>
You'll see the JSON data for the second product

Angular client

Working with HttpClient

Angular HttpClient

- Offers API for all standard HTTP methods,
e.g. `get()`, `post()`, `put()`, `delete()`, `request()`
- All these methods return `Observable`
- Interceptors - the code that's invoked on each HTTP request or response
- Supports progress events for request uploads and response downloads

Using HttpClient

Import HttpClientModule in @NgModule and inject an HttpClient object into the constructor of a service (or component)

```
import {HttpClientModule}  
       from '@angular/common/http';  
...  
@NgModule({  
  imports:      [ BrowserModule,  
                 HttpClientModule],  
  declarations: [ AppComponent],  
  bootstrap:    [ AppComponent ]  
})  
class AppModule { }
```

```
import {HttpClient} from '@angular/common/http';  
...  
class ProductService {  
  
  constructor(private httpClient: HttpClient){}  
  
  getProducts(){  
    this.httpClient.get<Product[]>('/api/products')  
      .subscribe(...);  
  }  
}
```

Server
endpoint

Explicit subscription to HttpClient

```
@Component({
  selector: 'app-root',
  template: `<h1>All Products</h1>
<ul>
  <li *ngFor="let product of products" > ←3
    {{product.title}} {{product.price}}
  </li>
</ul>
{{error}}
`)}
export class AppComponent implements OnInit, OnDestroy{

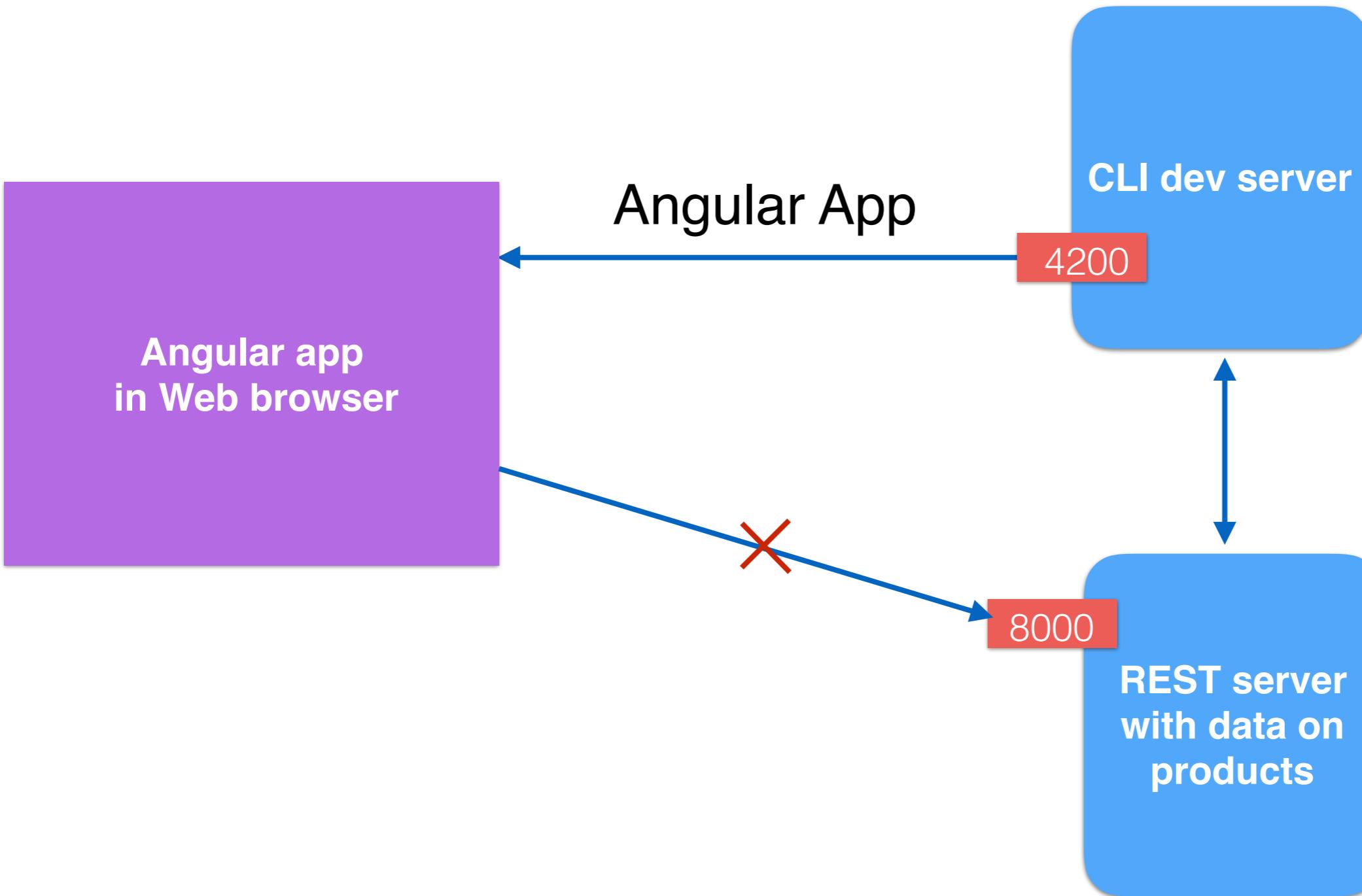
  products: Product[] = [];
  theDataSource: Observable<Product[]>;
  productSubscription: Subscription;
  error:string;

  constructor(private httpClient: HttpClient) {
    this.theDataSource = this.httpClient.get<Product[]>('/api/products'); ←1
  }

  ngOnInit(){
    this.productSubscription = this.theDataSource
      .subscribe(
        data => {
          this.products=data; ←2
        },
        err =>
          this.error = `Can't get products. Got ${err.status} from ${err.url}`
      );
  }

  ngOnDestroy(){
    this.productSubscription.unsubscribe(); ←4
  }
}
```

Configuring a proxy for dev mode



Same origin error

- In dev mode you can continue running the dev server for the client on port 4200 with `ng serve`
- But our REST server runs on port 8000
- If the client will do `httpClient.get('http://localhost:8000/api/products')`, it'll get this:


✖ XMLHttpRequest cannot load http://localhost:8000/api/products. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:4200' is therefore not allowed access.

Due to the **same-origin policy** we can configure a proxy on the client or add the header `Access-Control-Allow-Origin: *` on the server

Configuring proxy for Angular client

```
proxy-conf.json
{
  "/api": {
    "target": "http://localhost:8000",
    "secure": false
  }
}
```

The REST server
runs here

```
Angular client: http.get('/api/products');
```

goes to 4200
and gets redirected

```
ng serve restclient --proxy-config proxy-conf.json
```

Walkthrough 9.2

1. Keep the Node server from walkthrough 1 running on port 8000
2. In your IDE, open the directory **unit9/client** and run **npm i**
3. Review the code in the dir **app/restclient**
4. **ng serve restclient -o**
5. Browser's console shows 404 for `localhost:4200/api/products` because the server that runs on port 4200 doesn't have the endpoint `/api/products`.
6. Stop the dev server that runs on port 4200 (Ctrl-C)
7. Review the content of the file **proxy-conf.json**
8. Start the client with proxy
ng serve restclient --proxy-config proxy-conf.json -o
9. The browser renders three products received from our rest server

Subscribing to HttpClient with async pipe

```
@Component({
  selector: 'app-root',
  template: `<h1>All Products</h1>
<ul>
  <li *ngFor="let product of products$ | async">
    {{product.title}} {{product.price}}
  </li>
</ul>
{{error}}
`})
export class AppComponentAsync{  
  
  products$: Observable<Product[]>;
  error:string;
  constructor(private httpClient: HttpClient) {
    this.products$ = this.httpClient.get<Product[]>('/api/products') ←1
      .catchError( err => {
        this.error = `Can't get products. Got ${err.status} from ${err.url}`;
        return EMPTY;      // empty observable
      });
  }
}
```



Best practice

To run, modify restclient/app.module.ts to boot the AppComponentAsync:
bootstrap: [AppComponentAsync]

Best practice

**Dependency of
Dependency**



Dependency



Component



Angular
Injects

HttpClient → ProductService

Angular
Injects

ProductService → ProductComponent

Encapsulate all communications with the server
in an Angular service

```
@Injectable()
export class ProductService{

  constructor( private http: HttpClient){}

  getProductByID(productId: string): Observable<Product[]>{
    return this.http.get<Product[]>(`api/products/${productId}`);
  }
}
```

Service

```
@Injectable()
export class ProductService{

  constructor( private http: HttpClient){}

  getProductByID(productId: string): Observable<Product[]>{
    return this.http.get<Product[]>(`api/products/${productId}`);
  }
}
```

Service

```
export class AppComponent {

  products$: Observable<Product[]>; // subscribe with async pipe in the template

  constructor(private productService: ProductService) {}

  getProductByID(formValue){
    this.products$ = this.productService.getProductByID(formValue.productID);
  }
}
```

Component

Deploying Angular apps on the server with npm scripts

Serving Angular app from the Node server

- From the Node's perspective, Angular client's code is “static resources”
- Use Node's `path.join()`:

```
const app = express();

app.use('/', express.static(path.join(__dirname, 'public')));
```

server: rest-server-angular.ts

```
let express = require("express");
let path = require("path");
let compression = require("compression");

const app = express();
app.use(compression()); // serve gzipped files

app.use('/', express.static(path.join(__dirname, 'public')));

class Product {
    constructor(
        public id: number,
        public title: string,
        public price: number){}
}

const products = [
    new Product(0, "First Product", 24.99),
    new Product(1, "Second Product", 64.99),
    new Product(2, "Third Product", 74.99)
];

function getProducts(): Product[] {
    return products;
}

app.get('/api/products', (req, res) => {
    res.json(getProducts());
});

function getProductById(productId: number): Product {
    return products.find(p => p.id === productId);
}

app.get('/api/products/:id', (req, res) => {
    res.json(getProductById(parseInt(req.params.id)));
});

const server = app.listen(8000, "localhost", () => {
    const {address, port} = server.address();
    console.log('Listening on %s %s', address, port);
});
```

Angular app
will be
deployed here

all products

product by id

JiT vs AoT compilation

- Just-in-Time compilation: your app includes Angular's compiler and is dynamically compiled in the browser.
- Ahead-of-Time compilation: Angular components and templates are precompiled into JS with the `ngc` compiler.
- The AoT-compiled apps don't include the Angular compiler

AOT doesn't always make smaller bundles, but they load faster in the browser.

ng build for dev

the output goes into the **dist** dir

- ng build



```
136B Jan 29 12:46 data
5.3K Jan 29 12:46 favicon.ico
609B Jan 29 12:46 index.html
5.7K Jan 29 12:46 inline.bundle.js
5.8K Jan 29 12:46 inline.bundle.js.map
9.0K Jan 29 12:46 main.bundle.js
6.2K Jan 29 12:46 main.bundle.js.map
212K Jan 29 12:46 polyfills.bundle.js
259K Jan 29 12:46 polyfills.bundle.js.map
11K Jan 29 12:46 styles.bundle.js
15K Jan 29 12:46 styles.bundle.js.map
2.4M Jan 29 12:46 vendor.bundle.js
2.9M Jan 29 12:46 vendor.bundle.js.map
```



ng build for dev and prod

the output goes into the **dist** dir

- ng build



```
136B Jan 29 12:46 data
5.3K Jan 29 12:46 favicon.ico
609B Jan 29 12:46 index.html
5.7K Jan 29 12:46 inline.bundle.js
5.8K Jan 29 12:46 inline.bundle.js.map
9.0K Jan 29 12:46 main.bundle.js
6.2K Jan 29 12:46 main.bundle.js.map
212K Jan 29 12:46 polyfills.bundle.js
259K Jan 29 12:46 polyfills.bundle.js.map
11K Jan 29 12:46 styles.bundle.js
15K Jan 29 12:46 styles.bundle.js.map
2.4M Jan 29 12:46 vendor.bundle.js
2.9M Jan 29 12:46 vendor.bundle.js.map
```

- ng build --prod

performs AoT
by default



```
3.2K Jan 29 12:51 3rdpartylicenses.txt
136B Jan 29 12:51 data
5.3K Jan 29 12:51 favicon.ico
589B Jan 29 12:51 index.html
1.4K Jan 29 12:51 inline.d483d84aa7d8440978f5.bundle.js
175K Jan 29 12:51 main.8522776bac4edaecdaad.bundle.js
64K Jan 29 12:51 polyfills.47853ebf6acf9efe05b4.bundle.js
79B Jan 29 12:51 styles.9c0ad738f18adc3d19ed.bundle.css
```

Demo: ng build

1. Go to Terminal window in the directory unit9/client. After running each of the following commands check the content of the **dist** dir
2. `ng build` ← **JIT, not optimized**
3. `ng build --prod` ← **AoT, optimized**

Adding custom npm scripts



Walkthrough 9.3

1. In the Terminal window go to dir unit9/client
2. Review the scripts section in package.json
3. **npm run build** *<- It'll run build, postbuild, predeploy, and deploy.*
4. Switch to the server project and review the content of **server/build/public**. The Angular app was bundled and deployed there
5. Stop the server if running
6. Start the server with deployed Angular client:
node build/rest-server-angular
7. Open your browser at localhost:8000. The Angular app shows three products.

HTTP interceptors

@yfain

Why intercepting?

- For pre- and post-processing of all HTTP requests and responses, e.g. adding certain headers
- For implementing cross-cutting concerns, e.g. logging, global error handling, authentication

The original `HttpRequest` is immutable, but you can clone it and update the clone

Intercepting HTTP requests

```
@Injectable()
export class MyInterceptor implements HttpInterceptor {

  constructor (private auth: MyAuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const modifiedRequest =
      req.clone({ req.headers.set('Authorization', this.auth.getToken()) });

    return next.handle(modifiedRequest);
  }
}
```

```
@NgModule({
  ...
  providers: [[ { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true } ]],
})
```

Intercepting HTTP responses

- `next.handle()` returns an observable of `HttpEvent`'s
- Use RxJS operators to handle the response
- No need to subscribe as the original `HttpClient` has a subscriber

```
@Injectable()
export class MyInterceptor implements HttpInterceptor {
  ...
  return next.handle(modifiedRequest).pipe(
    tap(event => {
      if (event instanceof HttpResponse) {
        // clone and change event here if need be
        console.log(event);
      }
      return event;
    });
}
```

Demo: interceptors

An interceptor for logging errors

1. Start the server

```
node build/rest-server-angular-post-errors
```

2. Start the client

```
ng serve interceptor --proxy-config proxy-conf.json
```

3. Open the browser's console and start adding products.
The server will throw random errors that will be logged by the interceptor.

Demo

- **Progress events**
ng serve progressevents

What have we learned

- HTTP requests are treated as observable data streams
- How to create a simple Web server with NodeJS
- How to build and deploy an Angular app on the server
- How to automate your build process with npm scripts
- What are interceptors and progress events