



# DESARROLLADOR SOFTWARE – TALLER ANGULAR

FECHA- [] - MAYO 2022

Horario: 15:00 a 20:30 h

Juan Antonio Herrerías Berbel

# Índice

\_\_01 Angular

\_\_02 Structural Directives

\_\_03 Pipes & Methods

\_\_04 Splitting Components

\_\_05 Mocks & Models

\_\_06 Property & Class Binding

\_\_07 Event Binding

\_\_08 Two-way Binding

\_\_09 Service

\_\_10 HTTP

\_\_11 Component Interaction

\_\_12 Basic Routing

\_\_13 Share Data Between Components

\_\_14 Reactive Forms

Herre - jaherrerias@gmail.com





— 09

Service



```
TS item-list.component.ts x
1 import { Component, OnInit } from '@angular/core';
2 import { Item } from '../item.model';
3 import { ITEMS } from '../mocks';
4
5 @Component({
6   selector: 'app-item-list',
7   templateUrl: '../item-list.component.html',
8   styleUrls: ['../item-list.component.css']
9 })
10 export class ItemListComponent implements OnInit {
11   myItems: Item[];
12
13   constructor() { }
14
15   ngOnInit() {
16     this.myItems = ITEMS;
17   }
18
19   totalItems() {
20     return this.myItems.reduce( (prev, current) => prev + current.stock, 0);
21   }
22 }
```

```
TS mocks.ts x
1 import { Item } from '../item.model';
2
3 export const ITEMS: Item[] = [{
4   'id': 1,
5   'name': 'Item name',
6   'description': 'This item is the best one',
7   'stock': 5,
8   'price': 14.99
9 },
10 {
11   'id': 2,
12   'name': 'Another Item name',
13   'description': 'This item is the smallest',
14   'stock': 7,
15   'price': 5
16 },
17 {
18   'id': 3,
19   'name': 'A cheap Item',
20   'description': 'The cheapest item!',
21   'stock': 0,
22   'price': 3.99
23 }];
```

We are loading our item-list by importing our mock file, but this isn't the best solution for working with data.

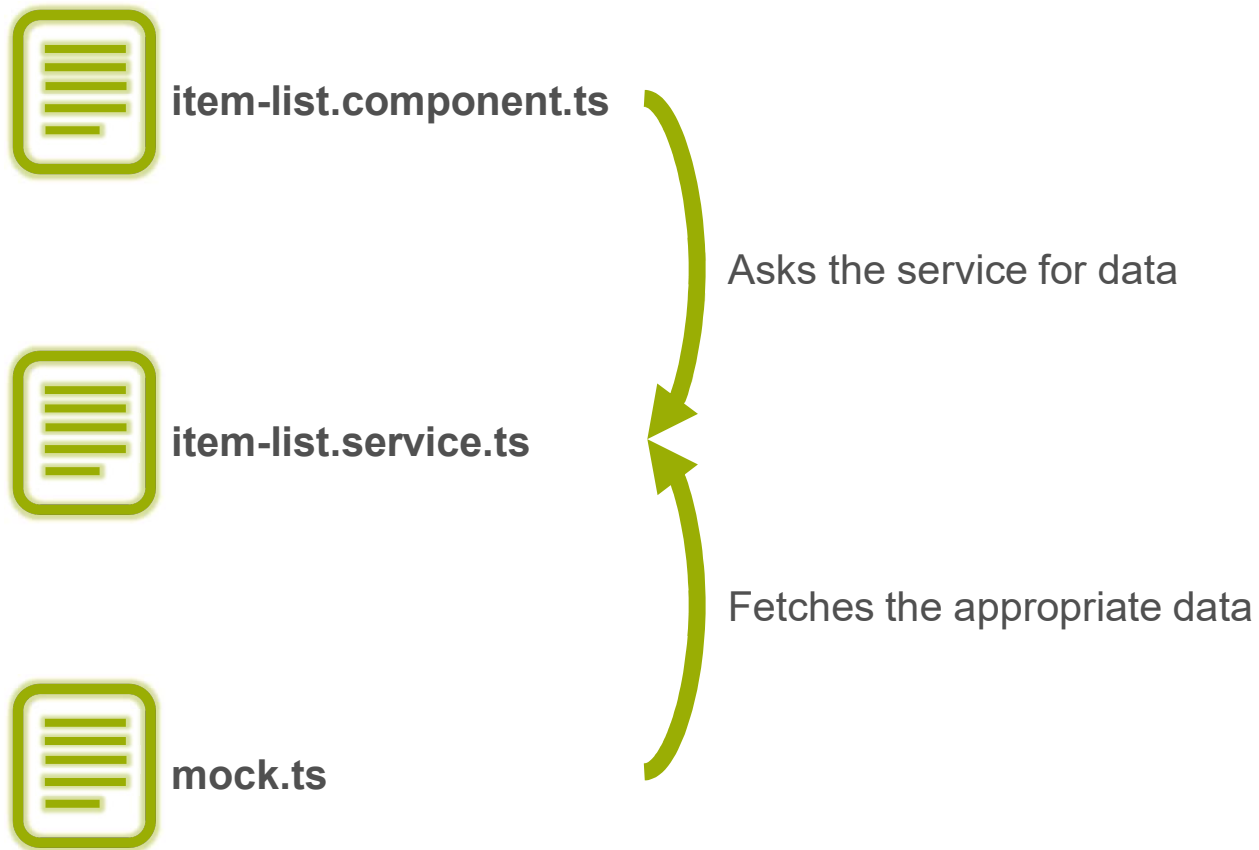


- We'd need to import the mocks on every file that needs the data. If the way we access this data changes, we'd have to change it everywhere.
- It's not easy to switch between real and mock data.
- This sort of data loading is best left to service classes.

```
TS item-list.component.ts x
1  import { Component, OnInit } from '@angular/core';
2  import { Item } from '../item.model';
3  import { ITEMS } from '../mocks';
4
5  @Component({
6    selector: 'app-item-list',
7    templateUrl: '../item-list.component.html',
8    styleUrls: ['../item-list.component.css']
9  })
10 export class ItemListComponent implements OnInit {
11   myItems: Item[];
12
13   constructor() { }
14
15   ngOnInit() {
16     this.myItems = ITEMS;
17   }
18
19   totalItems() {
20     return this.myItems.reduce( (prev, current) => prev + current.stock, 0);
21   }
22 }
```



Services are used to organize and share code across your app, and they're usually where we create our data access methods.



First, let's create the simplest service, and then we'll learn something called **dependency injection** to make it even more powerful.



TS item-list.service.ts ✕

```
1 import { ITEMS } from './mocks';
2 import { Item } from './item.model';
3
4 export class ItemListService {
5   getItemList(): Item[] {
6     return ITEMS;
7   }
8 }
```

- We've decoupled our data.

But .....

TS item-list.component.ts ✕

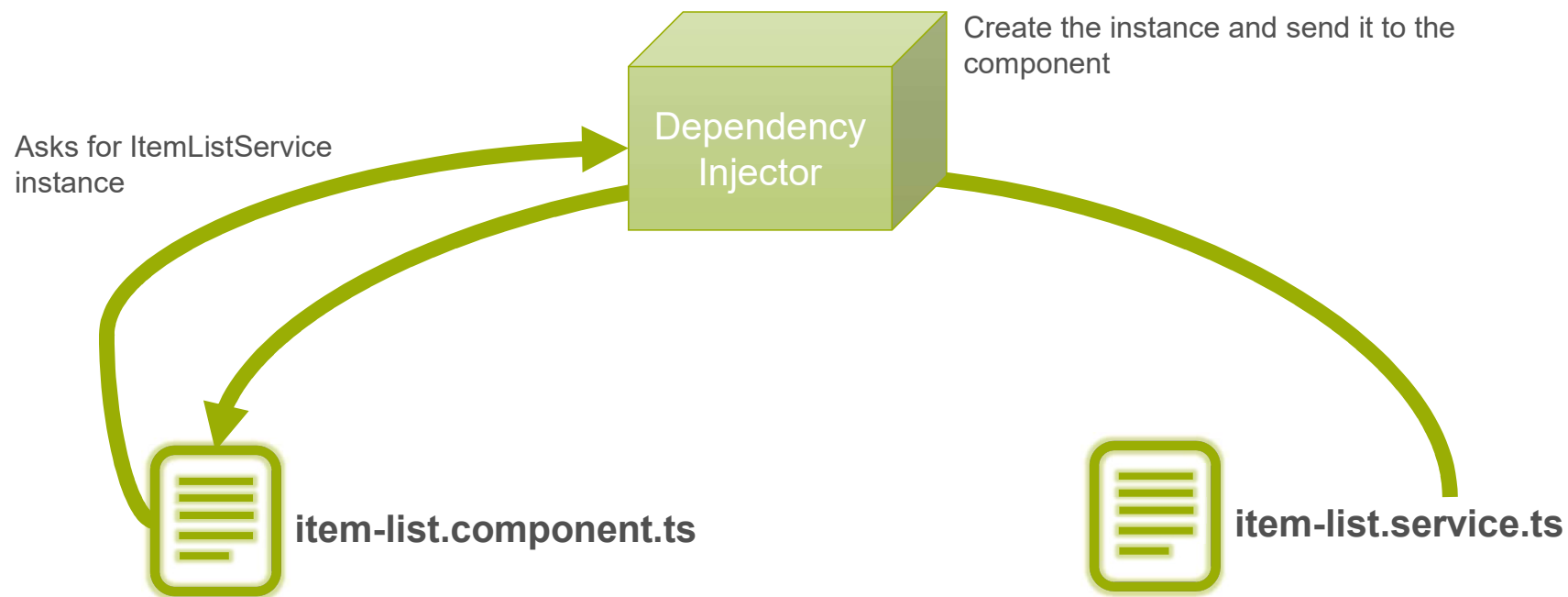
```
1 import { Component, OnInit } from '@angular/core';
2 import { Item } from './item.model';
3 import { ItemListService } from './item-list.service';
4
5 @Component({
6   selector: 'app-item-list',
7   templateUrl: './item-list.component.html',
8   styleUrls: ['./item-list.component.css']
9 })
10 export class ItemListComponent implements OnInit {
11   myItems: Item[];
12
13   constructor() { }
14
15   ngOnInit() {
16     const itemListService = new ItemListService();
17     this.myItems = itemListService.getItemList();
18   }
19 }
```

- Classes using this service must know how to create a ItemListService.
- We'll be creating a new ItemListService every time we need to fetch items.
- It'll be harder to switch between a mock service and a real one.





When you run an Angular application, it creates a dependency injector. An injector is in charge of knowing how to create and send things.

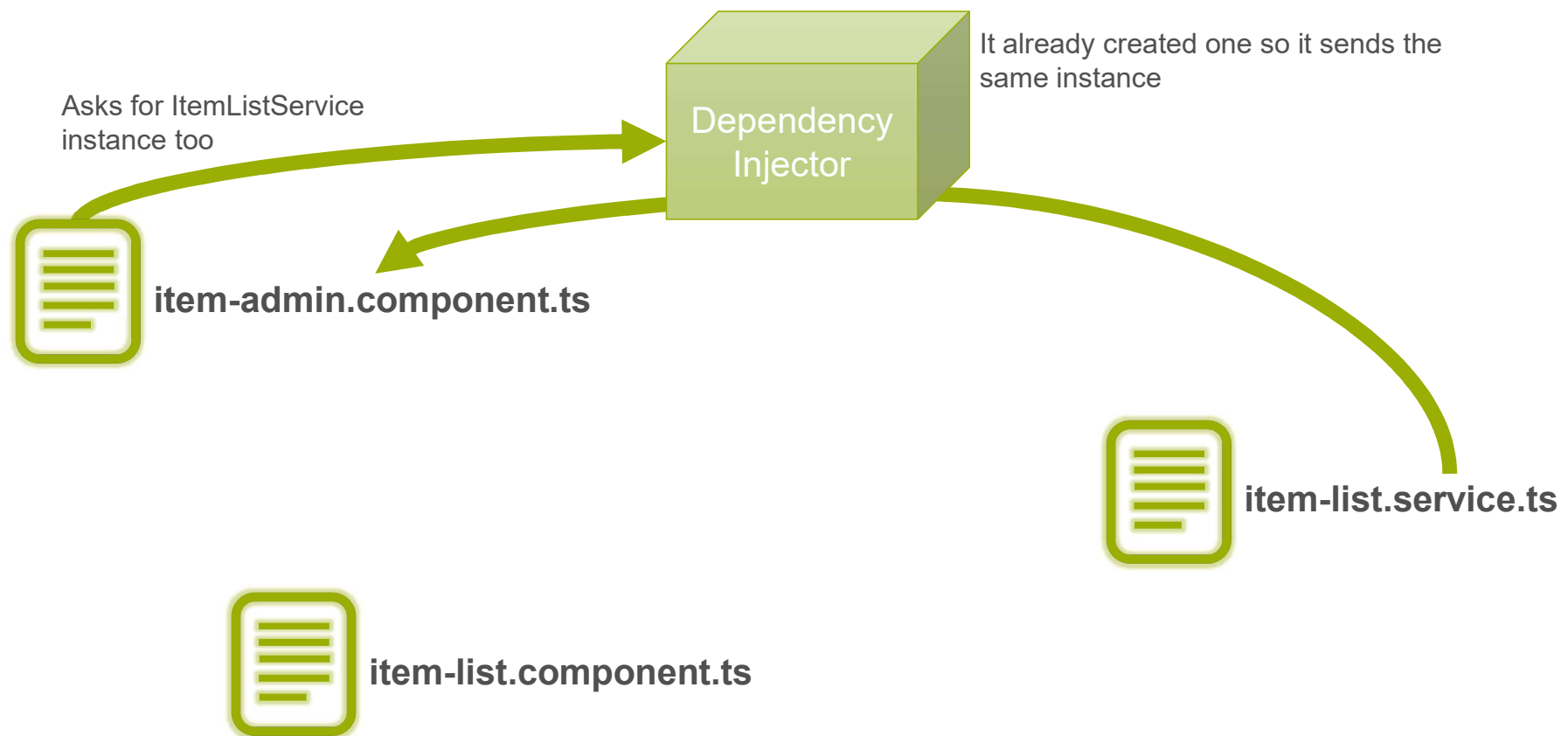


The injector knows how to inject our dependencies.

Create (if needed) and send classes we depend on.

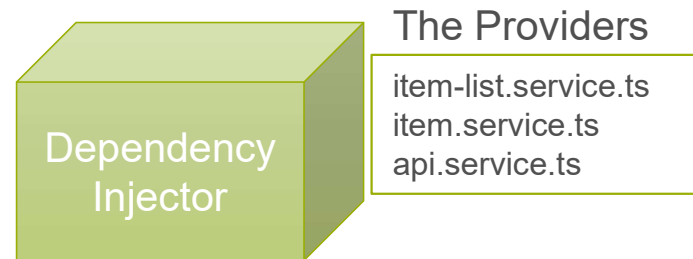


If the injector already created a service, we can have it resend the same service.





We must tell the injector what it can inject by registering “providers” with it.



There are three steps to make this all work with `ItemListService`:

1. Add the **injectable** decorator to `ItemListService`.
2. Let our **Dependency Injector** know about our service by naming it as a **provider**.
3. **Inject** the dependency into our *item-list.component.ts*.



We need to turn our service into something that can be safely used by our dependency injector.

```
TS item-list.service.ts x
1  import { ITEMS } from './mocks';
2  import { Item } from './item.model';
3  import { Injectable } from '@angular/core';
4
5  @Injectable()
6  export class ItemListService {
7      getItemList(): Item[] {
8          return ITEMS;
9      }
10 }
```

The parentheses!!!



We want all our subcomponents to have access to `ItemListService`. To do this, we register it as a provider at the `AppModule` module.

```
TS app.module.ts x
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4
5  import { AppComponent } from './app.component';
6  import { ItemListComponent } from './item-list/item-list.component';
7  import { ItemListService } from './item-list/item-list.service';
8
9  @NgModule({
10   declarations: [
11     AppComponent,
12     ItemListComponent
13   ],
14   imports: [
15     BrowserModule,
16     FormsModule
17   ],
18   providers: [ItemListService],
19   bootstrap: [AppComponent]
20 })
21 export class AppModule { }
```

Now all subcomponents can ask for (inject) our *ItemListService* when they need it, and an instance of *ItemListService* will either be delivered if it exists, or it will be created and delivered.



TS item-list.component.ts ✕

```
1 import { Component, OnInit } from '@angular/core';
2 import { Item } from '../item.model';
3 import { ItemListService } from '../item-list.service';
4
5 @Component({
6   selector: 'app-item-list',
7   templateUrl: '../item-list.component.html',
8   styleUrls: ['../item-list.component.css']
9 })
10 export class ItemListComponent implements OnInit {
11   myItems: Item[];
12
13   constructor(private itemListService: ItemListService) { }
14
15   ngOnInit() {
16     this.myItems = this.itemListService.getItemList();
17   }
18 }
```

This is what identifies that the ItemListService should be injected into this component.

We don't need to create the instance, Dependency Injector makes it for us

**Now our app is more scalable and testable.**


- Scalable because our dependencies aren't strongly tied to our classes.
- Testable because it'd be easy to mock services when we test the component.



Welcome to

Ever Shop!

There are 12 total items in stock.



ITEM NAME

This item is the best one


-

0

+

€14.99

5 in Stock



ANOTHER ITEM NAME

This item is the smallest


-

0

+

€5.00

7 in Stock



A CHEAP ITEM

The cheapest item!

-

0

+

€3.99

Out of Stock



- Services are used to organize and share code across your app, and they're usually where you create your data access methods.
- We use dependency injection to create and send services to the classes that need them.
- We give our dependency injector providers so that it knows what classes it can create and send for us.
- We ask for dependencies by specifying them in our class constructor.



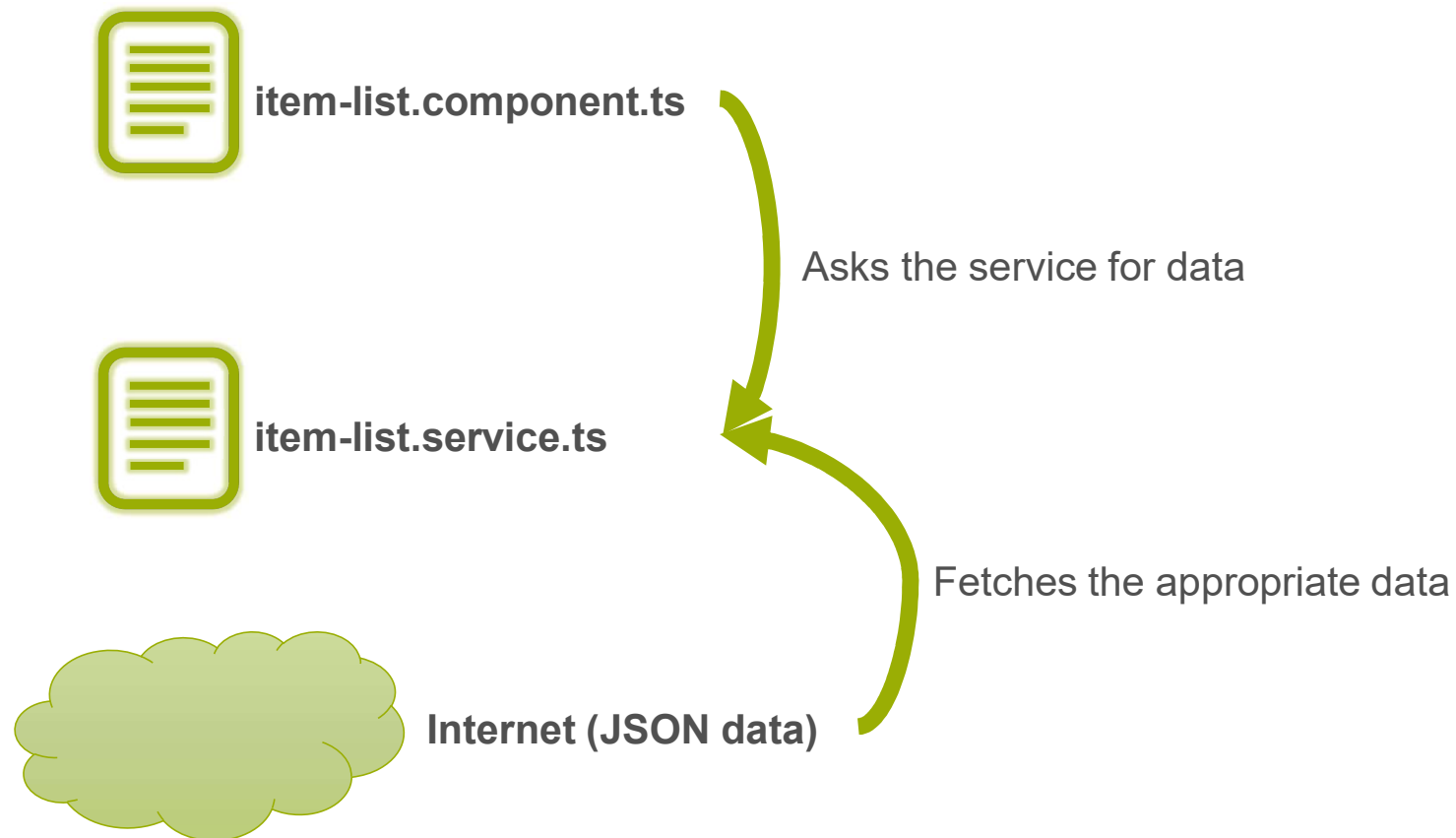


— **10**

# HttpClient

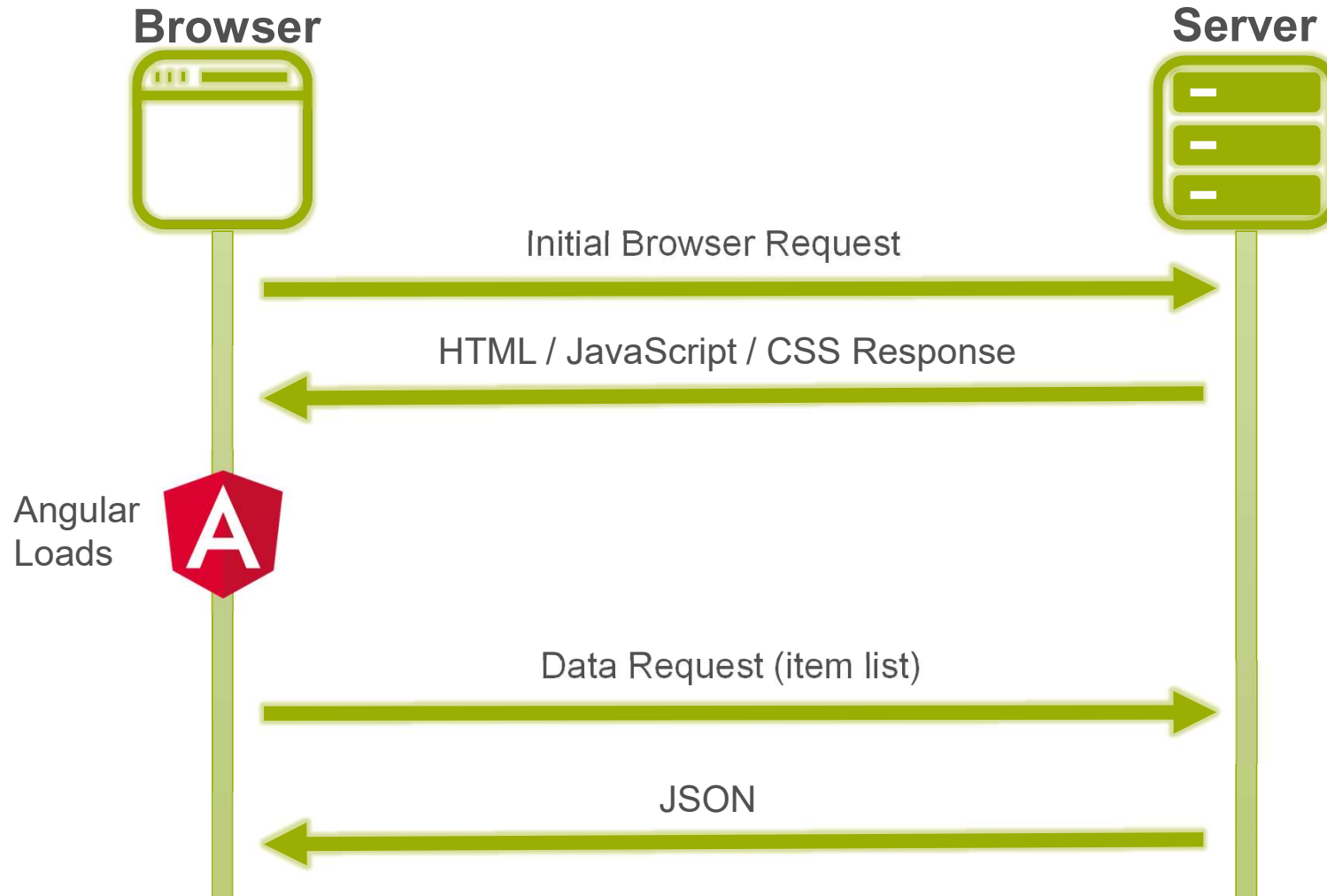


Up until now, we've been seeding our items with mock data. How might we fetch this from the internet instead?










When a user visits our web page, the Angular app loads first in our browser, and then it fetches the needed data.





1. Get a full fake REST API with JSON Server.  `item.json`
2. Ensure our application includes the libraries it needs to do Http calls.  `app.module.ts`
3. Tell our injector about the http provider.  `app.module.ts`
4. Inject the http dependency into our service and make the http get request.  `item-list.service.ts`
5. Listen for data returned by this request.  `Item-list.component.ts`



We have our awesome app and only want to work on its front end like we are making right now. But we need to create an API for it first because without it, we wouldn't be able to proceed with the app.

**json-server** is an open source mock API tool that solves this problem for us. It allows us to create an API with a database within minutes with bare minimum setup. Creating mock CRUD APIs with JSON-server requires zero-coding, we just need to write a configuration file for it.

**Github Project:** <https://github.com/typicode/json-server>

Install json-server globally using NPM:

```
npm install -g json-server
```

We need to create a JSON file in which we'll store item data for JSON Server to use (item-list-mock.json):

```
{ } item-list-mock.json x
1 {
2   "item-list": [
3     {
4       "id": 1,
5       "name": "Item name",
6       "description": "This item is the best one",
```

All we have to do now is start up JSON Server, pointing it to the *item-list-mock.json* file we just created, like so:

```
json-server item-list-mock.json
```



Head over to [localhost:3000](http://localhost:3000) and you'll see the default JSON Server page:

```
MINGW64/c:/Development/GitRepositories/ever-shop/src/app
jherrieri@ALC-7N6CPF2 MINGW64 /c:/Development/GitRepositories/ever-shop/src/app (master)
$ json-server item-list-mock.json

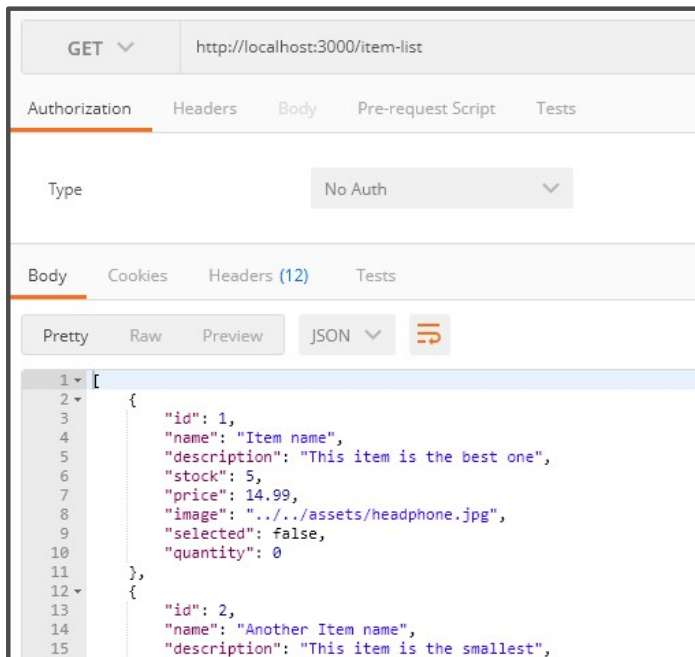
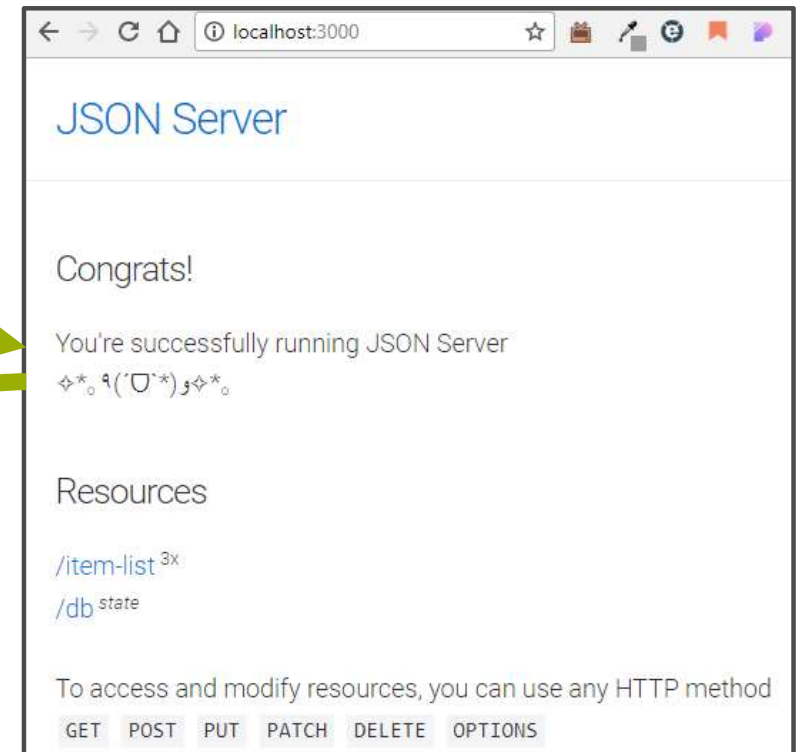
\{\^_\^}/ hi!

Loading item-list-mock.json
Done

Resources
http://localhost:3000/item-list

Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
```



The great part is that you can make GET, POST, PUT, PATCH, and DELETE requests to the server for testing your prototype, straight out-of-the-box.



- The HTTP library provides the get call we will use to call across the internet.
- The **RxJS** library stands for Reactive Extensions and provides some advance tooling for our http calls.



**UOOOOO!!!**



To inject the HttpClient service we need to import the HttpClientModule.

```
import { ItemListService } from './item-list/item-list.service';
import { HttpClientModule } from '@angular/common/http';

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

// import HttpClientModule after BrowserModule.

Now we'll be able to inject the HTTP library when we need it.







Now let's inject http and call out to the internet.

```
import { HttpClient } from "@angular/common/http"; 100.6K (gzipped: 27.3K)
import { Injectable } from "@angular/core"; 218.2K (gzipped: 70.1K)
import { Observable } from "rxjs"; 7.6K (gzipped: 2.7K)
import { Item } from "../item.model";

@Injectable()
export class ItemListService {

  constructor(private http: HttpClient) {}

  getItemList(): Observable<Item[]> {
    return this.http.get<Item[]>('http://localhost:3000/item-list');
  }
}
```




Injecting HTTP as a dependency and using RxJS Observables to call our fake API



Since our service now returns an observable object, we need to subscribe to that data stream and tell our component what to do when our data arrives.

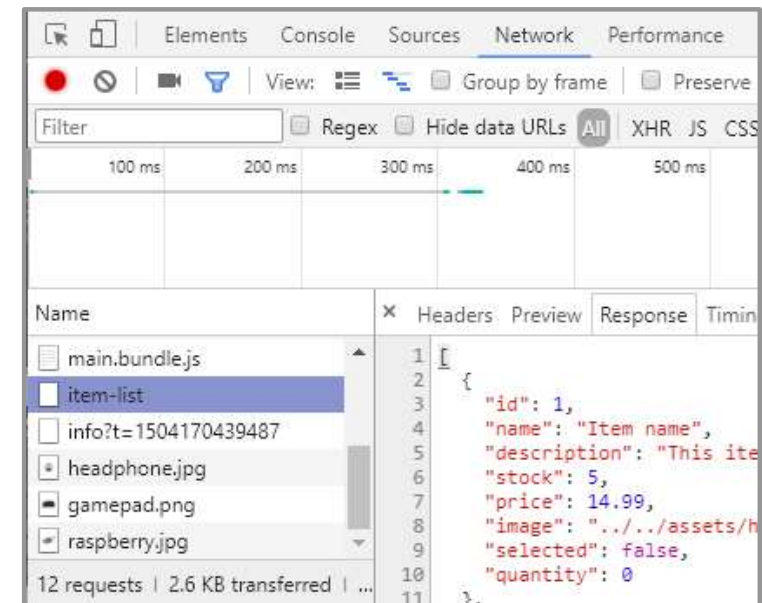
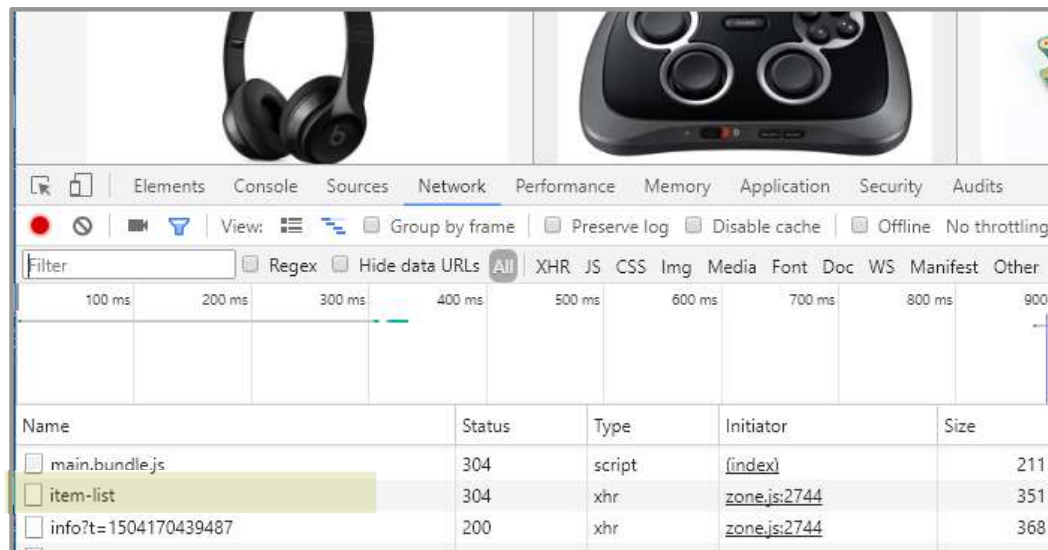
```
loadItems() {  
  this.itemService.getItemList()  
    .subscribe((data: Item[]) => this.myItems = data);  
}
```



When items arrive on our data stream, set it equal to our local items array.



If we open our browser, Chrome for example, and Chrome DevTools (F12) we can check in Network tab that now, there is at least one http call made by our app.




**What happen if we open Console Tab? Is there any error? Why?**



Welcome to

# Ever Shop!

There are 12 total items in stock.



ITEM NAME

This item is the best one


-

0

+

€14.99

5 in Stock



ANOTHER ITEM NAME

This item is the smallest


-

0

+

€5.00

7 in Stock



A CHEAP ITEM

The cheapest item!

-

0

+

€3.99

Out of Stock



## POST

TS *item-list.service.ts* ✕

```
const httpOptions = {  
  headers: new HttpHeaders({  
    'Content-Type': 'application/json',  
    'Authorization': 'my-auth-token'  
  })  
};  
  
addItem (item: Item): Observable<Item> {  
  return this.http.post<Item>(this.URL_BASE, item, httpOptions);  
}
```

TS *item-list.component.ts* ✕

```
addNewItem(item: Item) {  
  this.itemService.addItem(item)  
    .subscribe(data => this.loadItems() );  
}
```



PUT

TS *item-list.service.ts* ✕

```
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': 'my-auth-token'
  })
};

updateItem (item: Item): Observable<Item> {
  const url = `${this.URL_BASE}/${item.id}`;
  return this.http.put<Item>(url, item, httpOptions);
}
```

TS *item-list.component.ts* ✕

```
updateItem(item: Item) {
  this.itemListService.updateItem(item).subscribe();
}
```



## DELETE

TS *item-list.service.ts* ✕

```
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': 'my-auth-token'
  })
};

deleteItem(id: number): Observable<{}> {
  const url = `${this.URL_BASE}/${id}`;
  return this.http.delete(url, httpOptions);
}
```

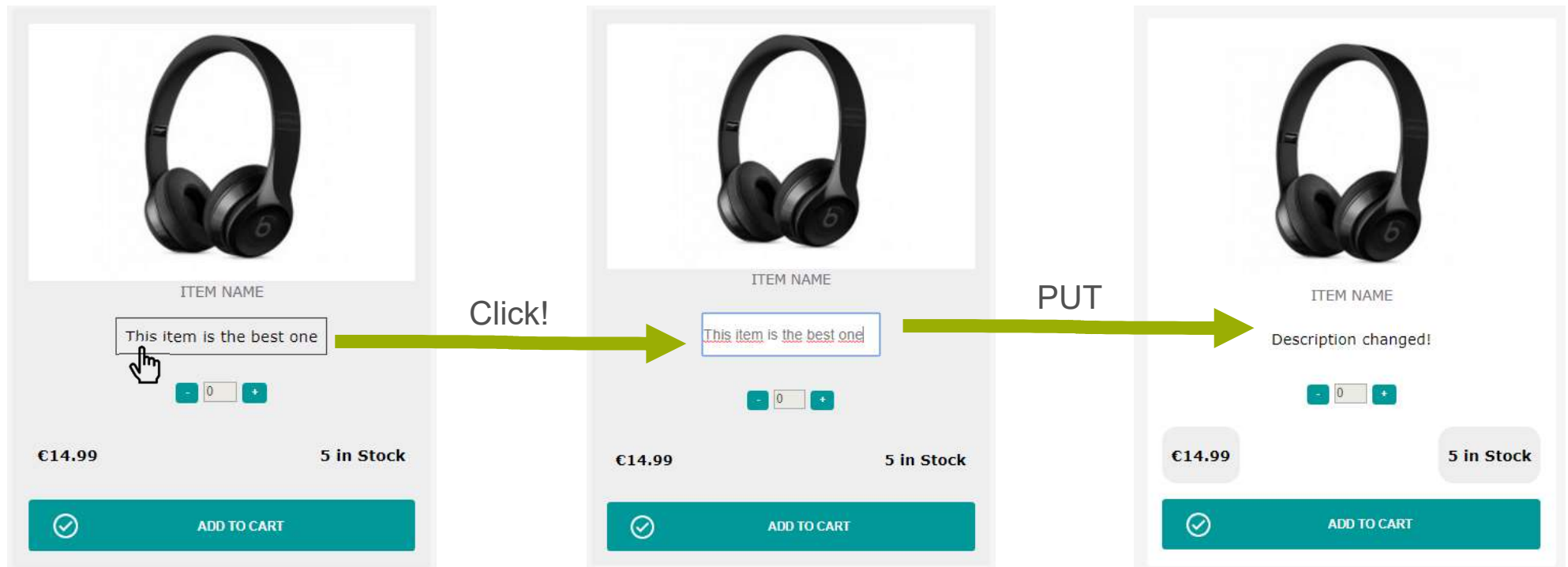
TS *item-list.component.ts* ✕

```
deleteItem(item: Item) {
  this.itemListService.deleteItem(item.id)
    .subscribe(
      data => {
        this.myItems = this.myItems.filter( el => {return el.id !== item.id} );
      });
}
```



- Vamos a añadir nuevas funcionalidades a nuestra tienda.
- Puedes editar la descripción del item. Cuando haces click sobre la descripción, el campo cambia a modo editable y después de cambiar el texto, tienes que actualizar la información.
  - 1. Añadir el servicio necesario a `item-list.service.ts`
  - 2. Añadir el método a `item-list.component.ts` para llamar al método de actualización en `item-list.service.ts`
  - 3. Cambiar la plantilla HTML para añadir la funcionalidad para la descripción del item. Después de confirmar el cambio, tienes que llamar al método creado en `item-list.component.ts` para hacer finalmente la llamada al servicio.





Name	×	Headers	Preview	Response	Timing
1					
1					

2 requests | 1008 B transferred

▼ General

Request URL: http://localhost:3000/item-list/1  
Request Method: PUT  
Status Code: 200 OK  
Remote Address: 127.0.0.1:3000  
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers [view source](#)

Access-Control-Allow-Credentials: true  
Access-Control-Allow-Origin: http://localhost:4200



- El ultimo cambio que hemos hecho, vamos a hacerlo utilizando algo que hemos aprendido de TS: enum
- Creamos una nueva enumeración llamada Mode, dentro del propio modelo de Item.
- Tendremos dos tipos de propiedades del enum: Edit y View
- Esta vez, cuando pasemos el ratón por encima del input, mostraremos un icono de fontawesome de pencil. Cuando hagamos click sobre él, pasaremos al modo Edit del enum. Una vez en modo Edit, pulsaremos sobre otro botón de cierre para pasar el modo View.





- **Nueva funcionalidad: añadir un item.** Tenemos que añadir una nueva funcionalidad a nuestro servicio para que sea capaz de AÑADIR nuevos items.
- Añadir lo necesario para que podamos guardar los datos de nuestro modelo en nuestro nuevo item. Será una capa (div) que se muestre cuando le demos al botón “New Item” y que se oculte cuando guardemos el nuevo item o cancelemos.
- **Nueva funcionalidad: borrar un item.** Necesitamos añadir una nueva funcionalidad a nuestro servicio para que podamos BORRAR items de nuestro sistema.
- En el modo edición, añadir un botón en nuestro item que nos permita al pulsarlo, llamar a nuestro servicio y borrar el item. Después de borrarlo, ya no debería aparecer en la lista que se muestre.



- Angular apps usually load data using service classes after the Angular app is initialized and running.
- We can use the HTTP library through dependency injection.
- Our http calls return an observable, not a promise, which behaves more like an array.



# Component Interaction



Using the same concept we have seen with bindings, we need to tell Angular what is coming into our component.



- We need to be able to pass data into our component
- To do this, we can import the **Input** decorator from the Angular core, and simply decorate the property

```
<div class="order-element">  
  <app-cart [cart]="cart"></app-cart>  
</div>
```

Data

Data

```
@Component({  
  selector: 'app-cart',  
  templateUrl: './cart.component.html',  
  styleUrls: ['./cart.component.css']  
})  
export class CartComponent implements OnInit {  
  @Input() cart: Cart;  
  
  constructor() { }
```

```
@Component({  
  selector: 'app-order',  
  templateUrl: './order.component.html',  
  styleUrls: ['./order.component.css']  
})  
export class OrderComponent implements OnInit {  
  cart: Cart;  
  shippingInformation: ShippingInformation;  
  total = 0;  
  ordered: boolean;  
  
  constructor(private cartService: CartService) { }  
  
  ngOnInit() {  
    this.cart = this.cartService.cart;  
    this.ordered = false;  
  }  
}
```



As we do when we setup an `@Input` decorator to accept an input binding, we can do the same and listen in the parent for when a value changes inside our child component.





```
@Component({
  selector: 'app-order',
  templateUrl: './order.component.html',
  styleUrls: ['./order.component.css']
})
export class OrderComponent implements OnInit {
  cart: Cart;
  shippingInformation: ShippingInformation;
  total = 0;
  ordered: boolean;

  constructor(private cartService: CartService) { }

  ngOnInit() {
    this.cart = this.cartService.cart;
    this.ordered = false;
  }

  updateTotal(total) {
    this.total = total;
  }
}
```

Event Data

```
<div class="order-element" *ngIf="!ordered">
  <app-cart [cart]="cart" (total)="updateTotal($event)"></app-cart>
</div>
```

Event

```
export class CartComponent implements OnInit {
  @Input() cart: Cart;
  @Output() total: EventEmitter<number> = new EventEmitter<number>();

  constructor() { }

  ngOnInit() {
    this.total.emit(this.getTotalCart());
  }

  remove(itemRemovable: number) {
    this.cart.items = this.cart.items.filter(item => item.id !== itemRemovable);
    this.total.emit(this.getTotalCart());
  }
}
```





Use an **input property setter** to intercept and act upon a value from the parent

```
TS item-list.component.ts x
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-name-parent',
5   template: `
6     <h2>Master controls {{names.length}} names</h2>
7     <app-name-child
8       *ngFor="let name of names" [name]="name">
9     </app-name-child>
10   `
11 })
12 export class NameParentComponent {
13   // Displays 'Dr IQ', '<no name set>', 'Bombasto'
14   names = ['Dr IQ', ' ', ' Bombasto '];
15 }
```

```
TS item.component.ts x
1 import { Component, Input } from '@angular/core';
2
3 @Component({
4   selector: 'app-item',
5   template: '<h3>{{name}}</h3>'
6 })
7 export class ItemComponent {
8   private _name = '';
9
10   @Input()
11   set name(name: string) {
12     this._name = (name && name.trim()) || '<no name set>';
13   }
14
15   get name(): string { return this._name; }
16 }
```



- Detect and act upon changes to input property values with the `ngOnChanges()` method of the **OnChanges** lifecycle hook interface
- You may prefer this approach to the property setter when watching multiple, interacting input properties

```
@Component({
  selector: 'app-version-parent',
  template: `
    <h2>Source code version</h2>
    <button (click)="newMinor()">
      New minor version
    </button>
    <button (click)="newMajor()">
      New major version
    </button>
    <app-version-child
      [major]="major"
      [minor]="minor">
    </app-version-child>
  `
})
export class VersionParentComponent {
```

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

@Component({
  selector: 'app-version-child',
  template: `
    <h3>Version {{major}}.{{minor}}</h3>
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changeLog">{{change}}</li>
    </ul>
  `
})
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];

  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    const log: string[] = [];
    // tslint:disable-next-line:forin
    for (const propName in changes) {
      const changedProp = changes[propName];
      const to = JSON.stringify(changedProp.currentValue);
      if (changedProp.isFirstChange()) {
        log.push(`Initial value of ${propName} set to ${to}`);
      } else {
        const from = JSON.stringify(changedProp.previousValue);
        log.push(`${propName} changed from ${from} to ${to}`);
      }
    }
    this.changeLog.push(log.join(', '));
  }
}
```



— **12**

# Basic Routing



- We use routing to separate different parts of our app, generally (but not always) by using the URL to denote our location.
- Angular's router is super easy to use
- The idea is that every URL (/login, /dashboard) is mapped to a component.
- There are three main components that we use to configure routing in Angular:
  - **Routes** describes our application's routes
  - **RouterOutlet** is a "placeholder" component that gets expanded to each route's content
  - **RouterLink** is used to link to routes



- **Routes** is an object that we use on our application component that describes the routes we want to use. For instance, we could write our routes like this:

```
const routes: Routes = [  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'home', component: HomeComponent },  
  { path: 'login', component: LoginComponent },  
  { path: 'dashboard', component: DashboardComponent }  
];
```

We can redirect using the  
'redirectTo' option

We use 'path' to specify the URL.

We specify the component we  
want to route to.





- To install our router into our app we use the `RouterModule.forRoot()` function in the `imports` key of our `app-routing.module.ts`:

```
app-routing.module.ts X
src > app > app-routing.module.ts > ...
1 import { NgModule } from '@angular/core'; 218.5K (gzipped: 70.2K)
2 import { RouterModule, Routes } from '@angular/router'; 96.2K (gzipped: 25.6K)
3 import { ItemEditComponent } from './items/item-edit/item-edit.component';
4 import { ItemsComponent } from './items/items.component';
5 import { OrderComponent } from './order/order.component';
6 import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
7
8 const routes: Routes = [
9   { path: '', redirectTo: 'home', pathMatch: 'full' },
10  { path: 'home', component: ItemsComponent },
11  { path: 'item/:id', component: ItemEditComponent },
12  { path: 'order', component: OrderComponent },
13  { path: '**', component: PageNotFoundComponent }
14 ];
15
16 @NgModule({
17   imports: [RouterModule.forRoot(routes)],
18   exports: [RouterModule]
19 })
20 export class AppRoutingModule { }
21
```

Route Wildcards





- After defining the routes, we need to import them into the **AppModule** to take effect:

```
@NgModule({
  declarations: [
    AppComponent,
    ItemListComponent,
    ItemsComponent,
    ItemComponent,
    CartComponent,
    OrderComponent,
    ShippingInfoComponent,
    ItemDetailComponent,
    PageNotFoundComponent,
    ItemEditComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [ItemService, CartService, {provide: LOCALE_ID, useValue: 'es-ES'}],
  bootstrap: [AppComponent]
})
export class AppModule { }
registerLocaleData(es);
```



- It is also possible, to create routes that take dynamic parameters
- We append the ID of the instance to the route: **'/item/12345'**
- To define a route with a parameter, we add a **':'** to the route

```
TS app.routes.ts x
1  import { Routes, RouterModule } from '@angular/router';
2  import { ItemsComponent } from './items/items.component';
3  import { OrderComponent } from './order/order.component';
4  import { ModuleWithProviders } from '@angular/core';
5  import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
6  import { ItemEditComponent } from './items/item-edit/item-edit.component';
7
8  const routes: Routes = [
9    { path: '', component: ItemsComponent },
10   { path: 'home', component: ItemsComponent },
11   { path: 'item/:id', component: ItemEditComponent },
12   { path: 'order', component: OrderComponent },
13   { path: '**', component: PageNotFoundComponent }
14 ];
15
16 export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Route Wildcards



- To read out the current route parameter in the component, we need the active route
- We subscribe to the params observable of the active route

```
constructor(private service: ItemService, private route: ActivatedRoute, private router: Router) { }
```

```
ngOnInit() {  
  this.route.params.subscribe(params => {  
    const id = <string>params['id'];  
    if (id != null) {  
      // Load the right article  
    }  
  });  
}
```



- From HTML Template: **RouterLink**

```
<div>
  <a routerLink="/home">Cancel</a>
  <button (click)="save()">Save</button>
</div>
```

- Code from component itself

```
constructor(private service: ItemService, private route: ActivatedRoute, private router: Router) { }
```

```
this.router.navigate(['/home']);
```

```
editItem(id: number) {
  this.router.navigate(['/item', id]);
}
```

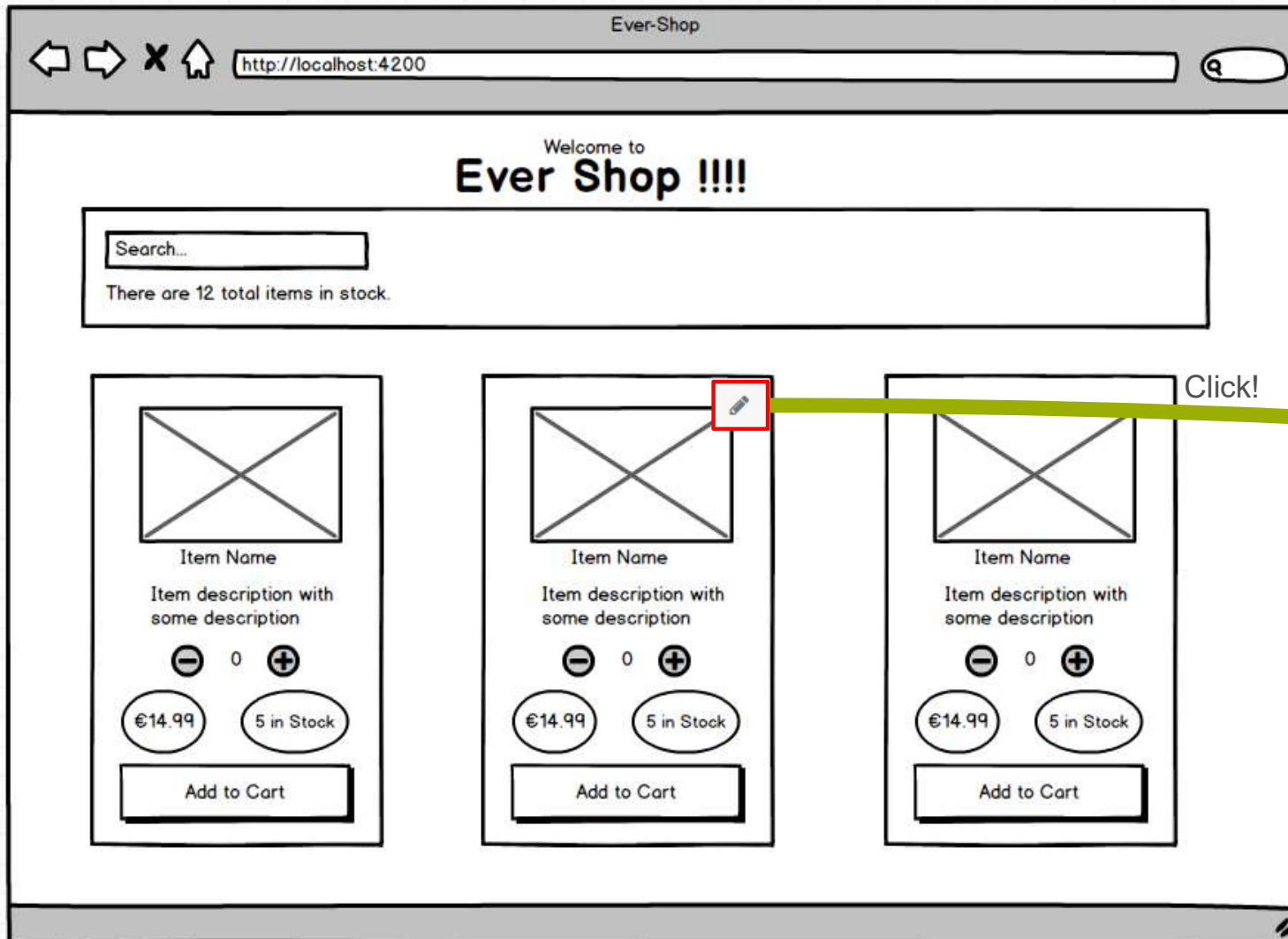


- The **RouterOutlet** directive tells our router where to render the content in the view.

```
@View({  
  template: `  
    <div>  
      Stuff in the outer template here  
      <router-outlet></router-outlet>  
    </div>  
  `,  
})
```

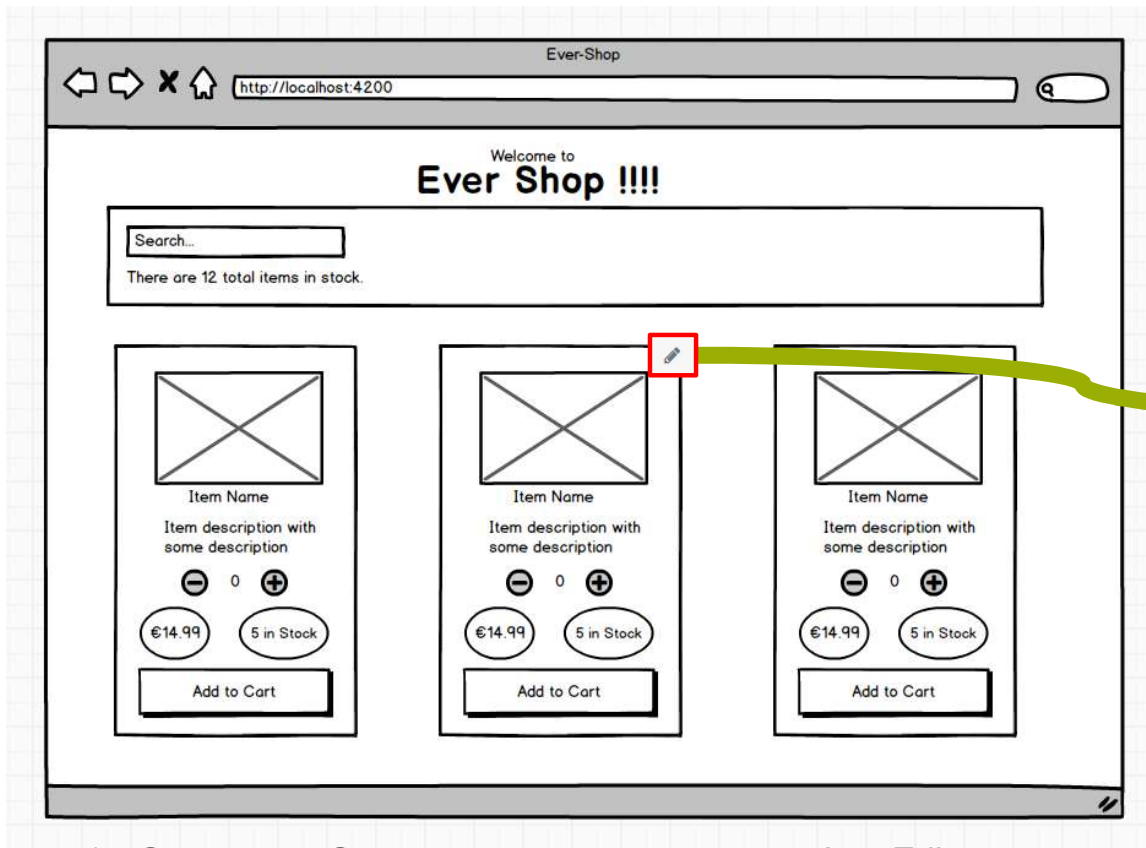
Whenever we switch routes, our content will be **rendered** in place of the **router-outlet** tag.

If we were switching to the **/login** page, the content rendered by the **LoginComponent** would be placed there.



Click!

Item Name	Another Item name
Item Description	These item is the smallest
Stock	7
Price (€)	5
Image Name (*.*)	../assets/pic2.png



Click!

Item Name	<input type="text" value="Another Item name"/>
Item Description	<input type="text" value="These item is the smallest"/>
Stock	<input type="text" value="7"/>
Price (€)	<input type="text" value="5"/>
Image Name (*)	<input type="text" value="../assets/pic2.png"/>

1. Create new Component -> ng g component ItemEdit
2. Add New Route 'item/:id'
3. Change the component to manage an Item and load it from ActivatedRoute
4. We can cancel the view to navigate to home or save the model. After saving we have to update our item and then navigate to home