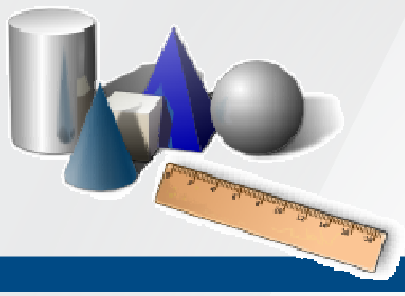
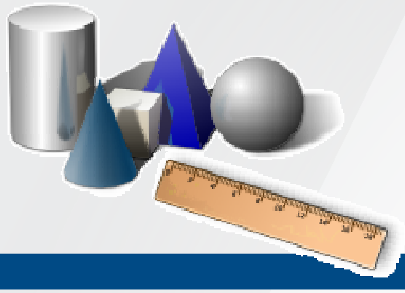


Lab 13

- Take the project from the before folder
- Build and Test the app
- Take a look to the code :
  - our sample application we were retrieving our data using a PeopleService that just accessed a simple array of Persons stored in memory
  - Let’s switch that for a more realistic scenario where we retrieve our data from an actual web service. For that purpose we will use the Star Wars API and Angular 2 Http module. But first things first, we need to enable it.
    - IF we used the Angular cli to boostrap our app the http module and rxjs have been automatically added to our project and are ready to be used

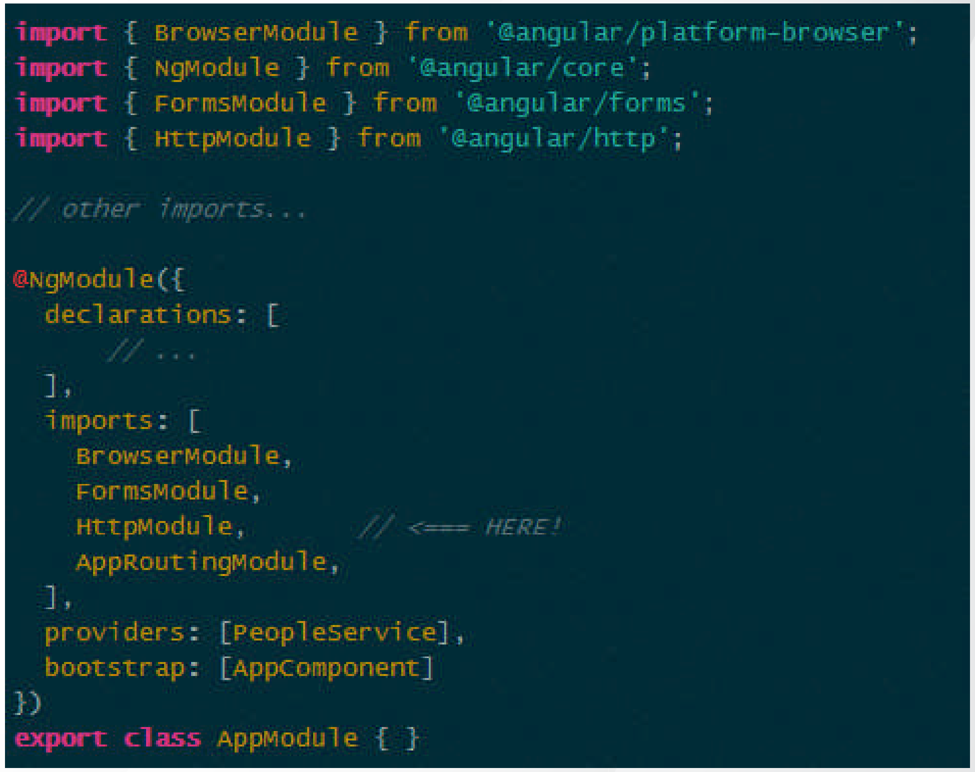
http://swapi.co/api/people/

© Global Knowledge Training LLC. All rights reserved.



Lab 13

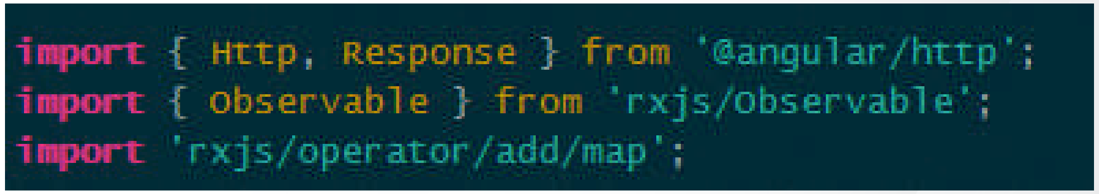
- Next take a look at your app’s main module AppModule in app.module.ts:
- The HttpModule is imported and setup as one of the imports in our app’s main module.



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

// other imports...

@NgModule({
  declarations: [
    // ...
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,          // <=== HERE!
    AppRoutingModule,
  ],
  providers: [PeopleService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

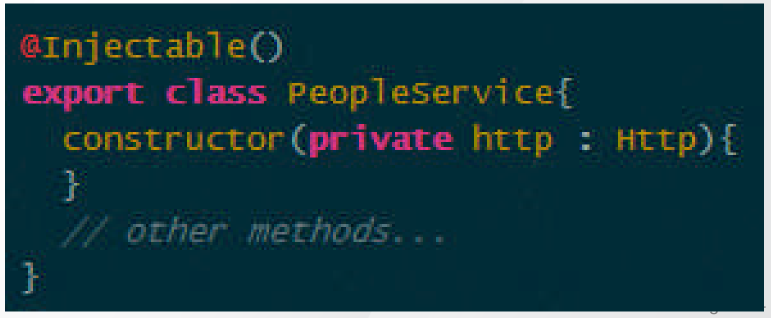
Lab 13

- **Let’s Get Started Using The Http Module**
  - The Angular 2 http module @angular/http exposes a Http service that our application can use to access web services over HTTP.
  - We’ll use this marvellous utility in our **PeopleService** service. We start by importing it



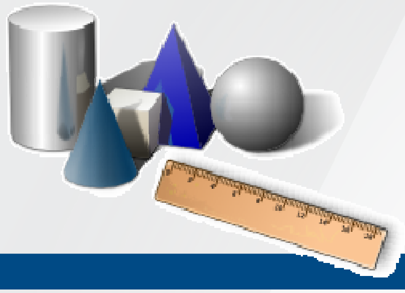
```
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/operator/add/map';
```

  - After importing the necessary items we can inject the Http service inside PeopleService through its constructor:



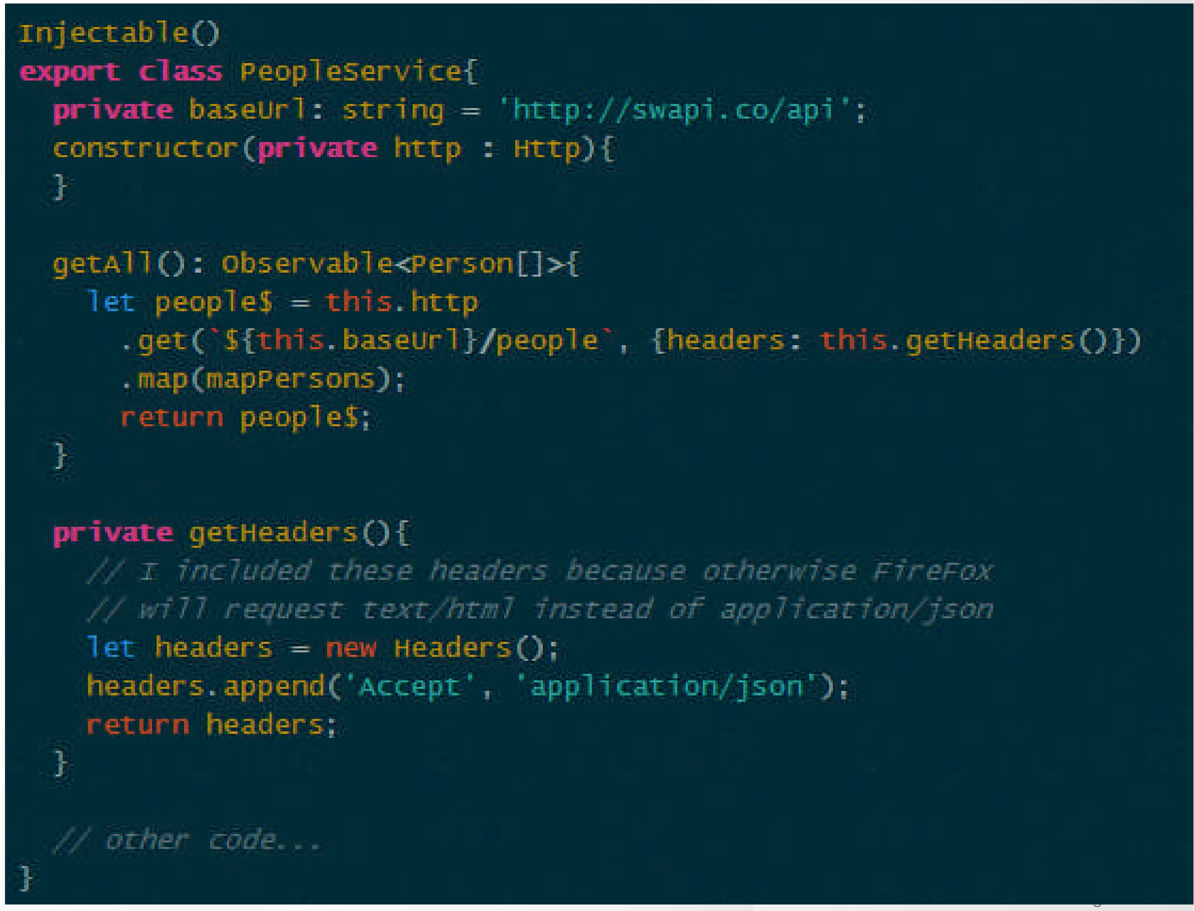
```
@Injectable()
export class PeopleService{
  constructor(private http : Http){
  }
  // other methods...
}
```

© Global Knowledge Training LLC. All rights reserved.



Lab 13

- So! Now that we have injected the Http service in our PeopleService we can use it to get those pesky Star Wars personages. We will update
  our getAll method like this:

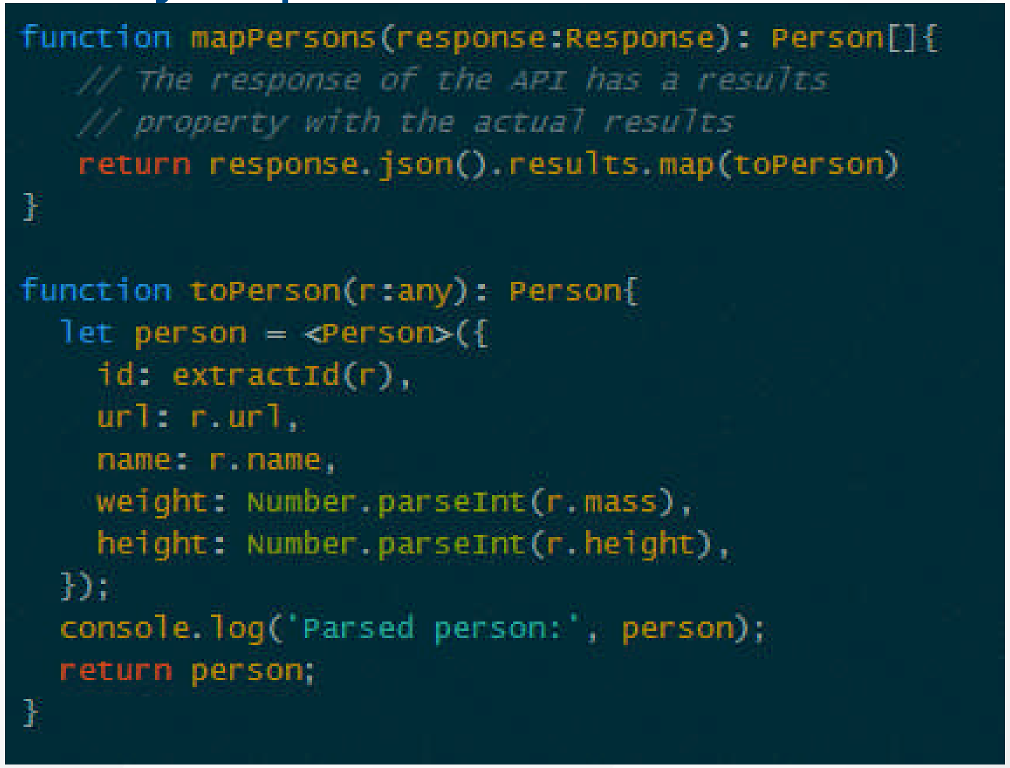


```
Injectable()
export class PeopleService{
  private baseUrl: string = 'http://swapi.co/api';
  constructor(private http : Http){
  }

  getAll(): Observable<Person[]>{
    let people$ = this.http
      .get(`${this.baseUrl}/people`, {headers: this.getHeaders()})
      .map(mapPersons);
      return people$;
  }

  private getHeaders(){
    // I included these headers because otherwise FireFox
    // will request text/html instead of application/json
    let headers = new Headers();
    headers.append('Accept', 'application/json');
    return headers;
  }

  // other code...
}
```

© Global Knowledge Training LLC. All rights reserved.

Lab 13

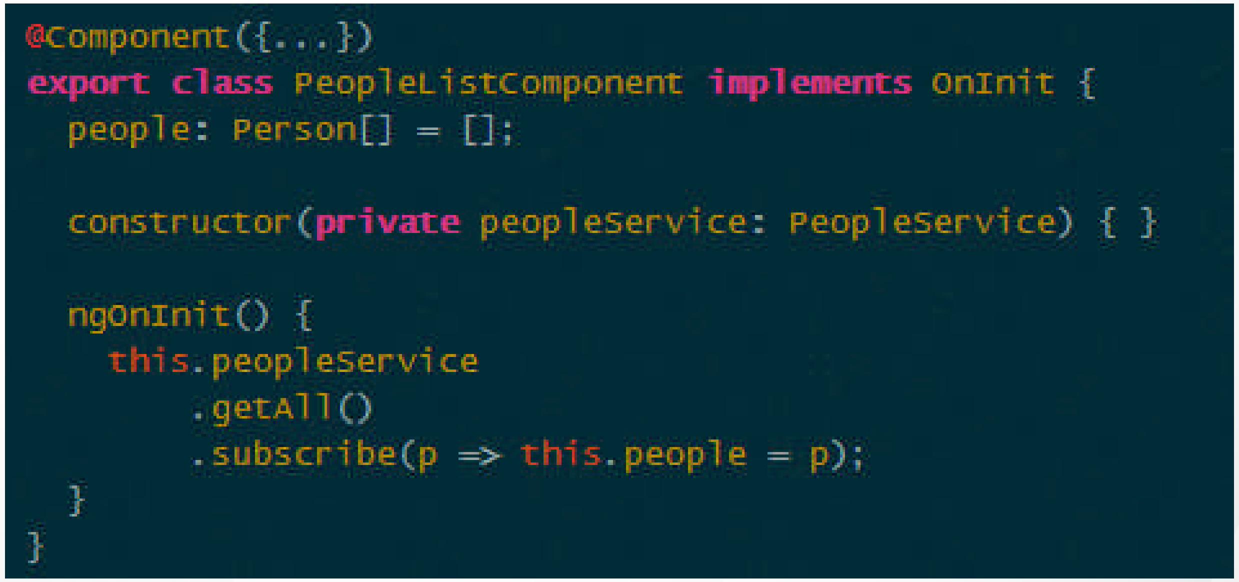
- we can define that mapPersons function to transform a Response into an array of persons as follows:



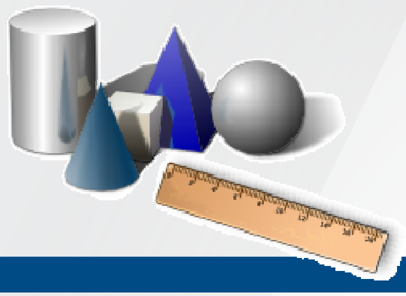
```
function mapPersons(response:Response): Person[]{
   // The response of the API has a results
   // property with the actual results
   return response.json().results.map(toPerson)
}

function toPerson(r:any): Person{
  let person = <Person>({
    id: extractId(r),
    url: r.url,
    name: r.name,
    weight: Number.parseInt(r.mass),
    height: Number.parseInt(r.height),
  });
  console.log('Parsed person:', person);
  return person;
}
```

Lab 13

- How can you unpack that array of persons from the Observable<Person[]>?
  - You subscribe to the observable like this:
-

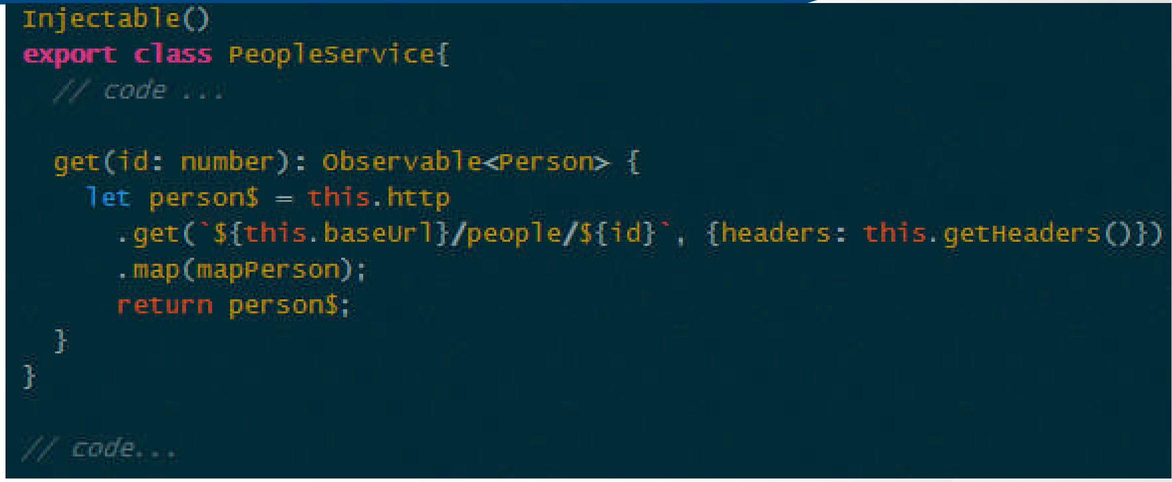


```
@Component({...})
export class PeopleListComponent implements OnInit {
  people: Person[] = [];

  constructor(private peopleService: PeopleService) { }

  ngOnInit() {
    this.peopleService
        .getAll()
        .subscribe(p => this.people = p);
  }
}
```



Lab 13

- We can continue consuming the Star Wars web service by following the same process with the get method:

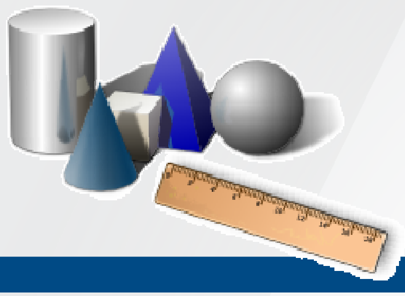


```
Injectable()
export class PeopleService{
  // code ...

  get(id: number): Observable<Person> {
    let person$ = this.http
      .get(`${this.baseUrl}/people/${id}`, {headers: this.getHeaders()})
      .map(mapPerson);
      return person$;
  }
}

// code...
```

And the save method:

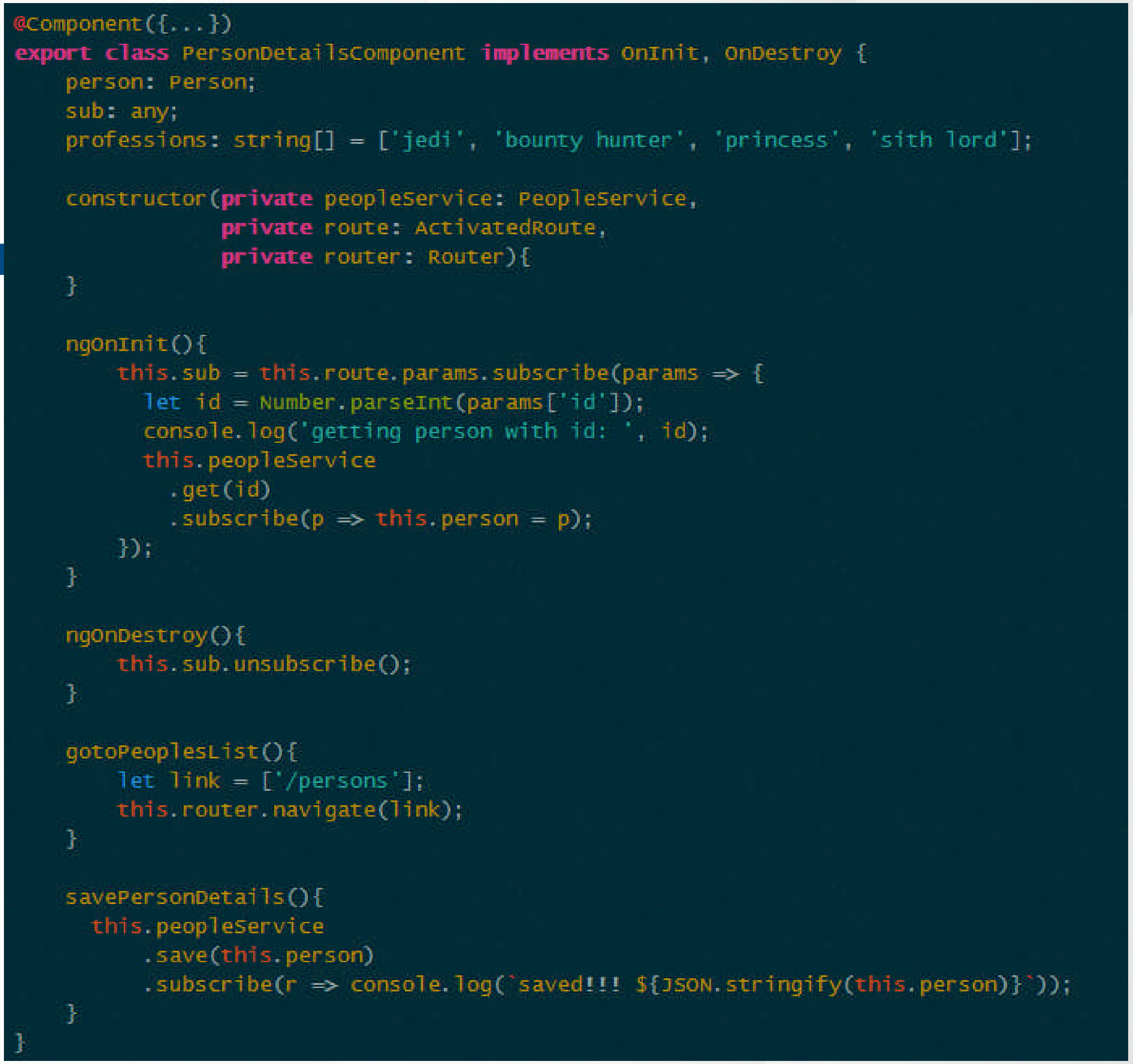
```
// code ...
save(person: Person) : Observable<Response>{
   // this won't actually work because the StarWars API doesn’t
   // is read-only. But it would look like this:
  return this
   .http
   .put(`${this.baseUrl}/people/${person.id}`, JSON.stringify(person), {headers: this.getHeaders()});
}
```

© Global Knowledge Training LLC. All rights reserved

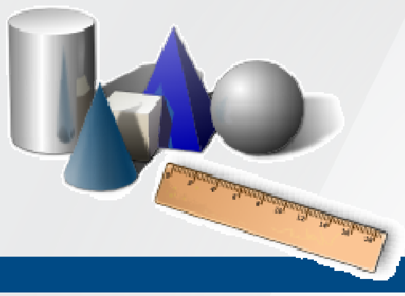


Lab 13

- Since we have changed the public API of the PeopleService service again, we'll need to update the PersonDetailsComponent that uses it. Once again, we subscribe to the observables being returned:
- Finally! We have our whole application consuming a real web service and getting real data to display. Yey!

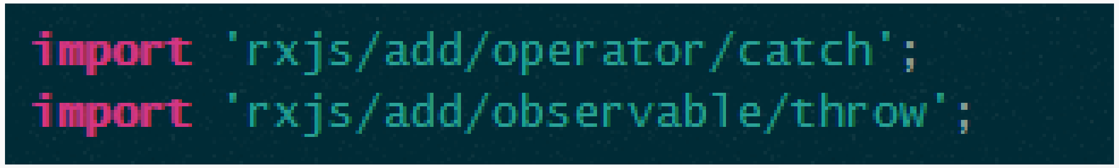


```
@Component({...})
export class PersonDetailsComponent implements OnInit, OnDestroy {
    person: Person;
    sub: any;
    professions: string[] = ['jedi', 'bounty hunter', 'princess', 'sith lord'];

    constructor(private peopleService: PeopleService,
                private route: ActivatedRoute,
                private router: Router){
    }

    ngOnInit(){
        this.sub = this.route.params.subscribe(params => {
          let id = Number.parseInt(params['id']);
          console.log('getting person with id: ', id);
          this.peopleService
            .get(id)
            .subscribe(p => this.person = p);
        });
    }

    ngOnDestroy(){
        this.sub.unsubscribe();
    }

    gotoPeoplesList(){
        let link = ['/persons'];
        this.router.navigate(link);
    }

    savePersonDetails(){
      this.peopleService
          .save(this.person)
          .subscribe(r => console.log(`saved!!! ${JSON.stringify(this.person)}`));
    }
}
```

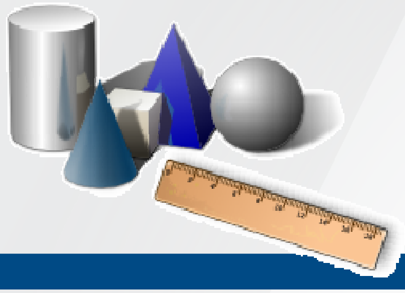


Lab 13

- **Error Handling with Observables (obtional)**
- We have two different levels of abstraction (one deals with HTTP requests and responses and the other one with domain model objects), we are going to have two levels of error handling.
  - The first level of error handling happens at the service level and deals with problems that can happen with HTTP requests
  - We start by importing the catch operator and the throw method from rxjs as follows in PeopleService class

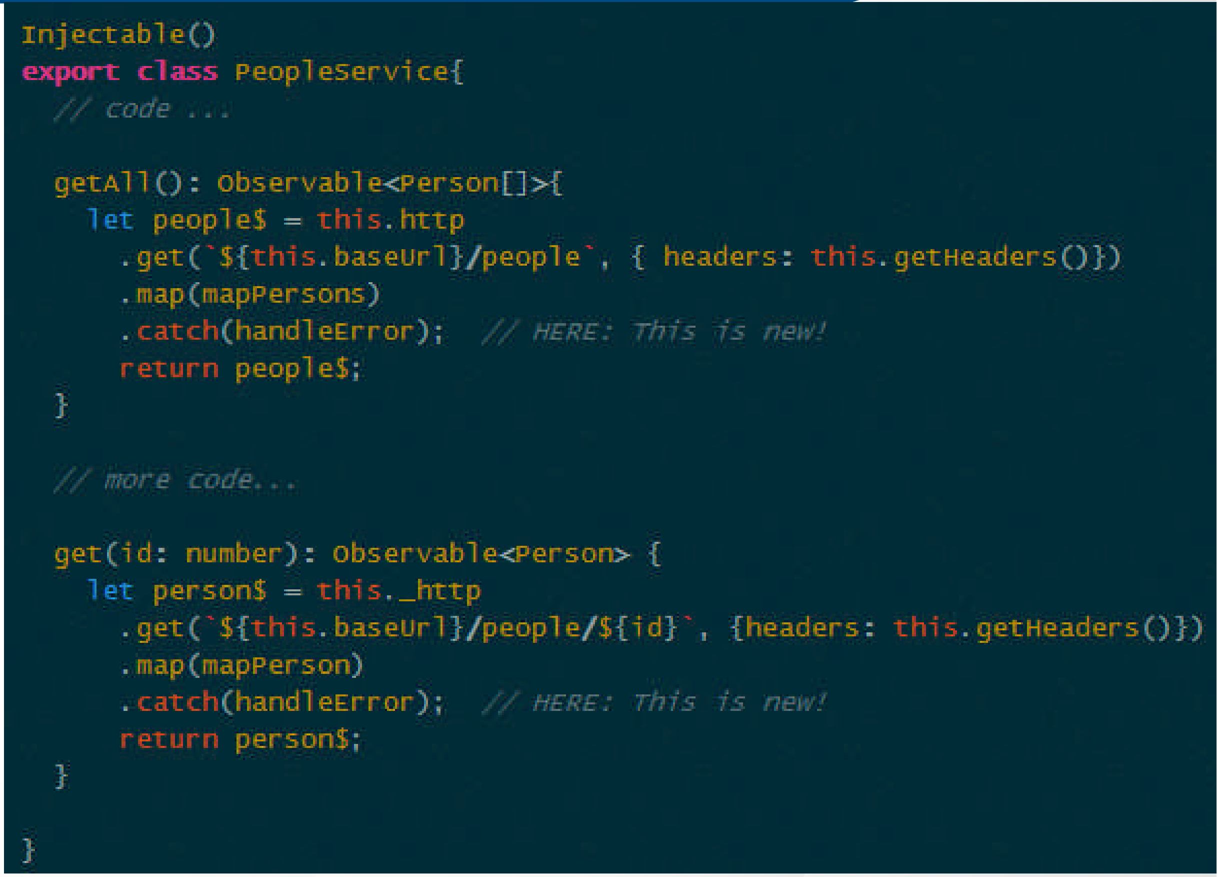


```
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';
```

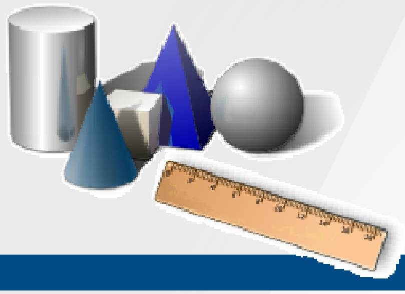


Lab 13

- And now we use these in our rxjs streams. Specifically:
  - We use catch to handle errors within a stream
  - We use throw to throw a new error at a higher level of abstraction



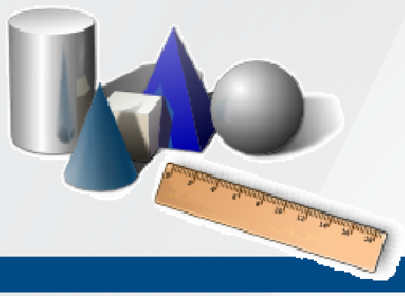
```
Injectable()
export class PeopleService{
  // code ...

  getAll(): Observable<Person[]>{
    let people$ = this.http
      .get(`${this.baseUrl}/people`, { headers: this.getHeaders()})
      .map(mapPersons)
      .catch(handleError);  // HERE: This is new!
      return people$;
  }

  // more code...

  get(id: number): Observable<Person> {
    let person$ = this._http
      .get(`${this.baseUrl}/people/${id}`, {headers: this.getHeaders()})
      .map(mapPerson)
      .catch(handleError);  // HERE: This is new!
      return person$;
  }

}
```



Lab 13

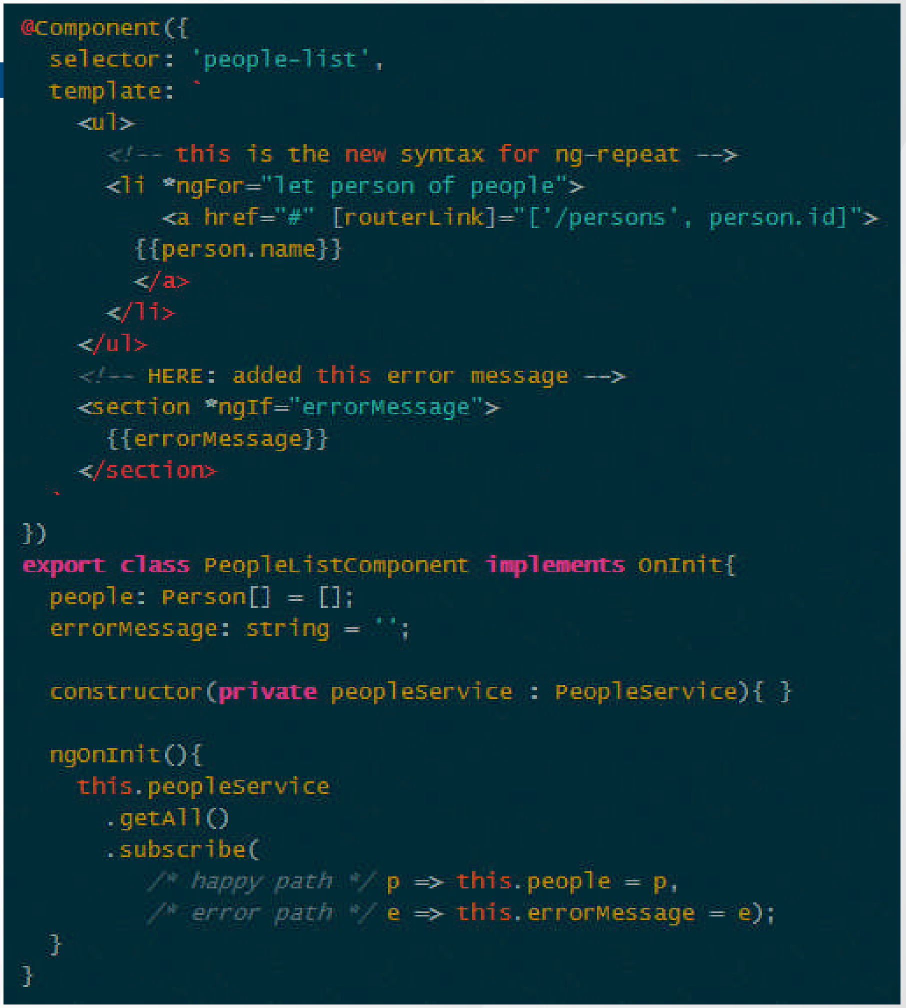
And throw let’s us push a new error upwards into our application:

```
// this could also be a private method of the component class
function handleError (error: any) {
  // log error
  // could be something more sofisticated
  let errorMsg = error.message ||
 `Yikes! There was a problem with our hyperdrive device and we couldn't retrieve your data!`
 console.error(errorMsg);
  // throw an application level error
  return Observable.throw(errorMsg);
}
```

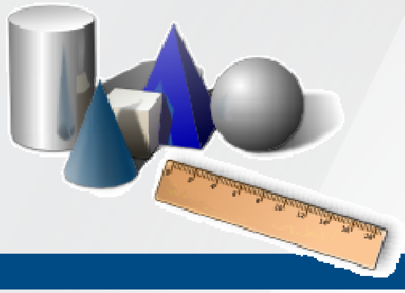


Lab 13

- The second level error handling happens at the application level inside the component:
- Where we use the second argument of the subscribe method to subscribe to errors and display error messages to the user.
-

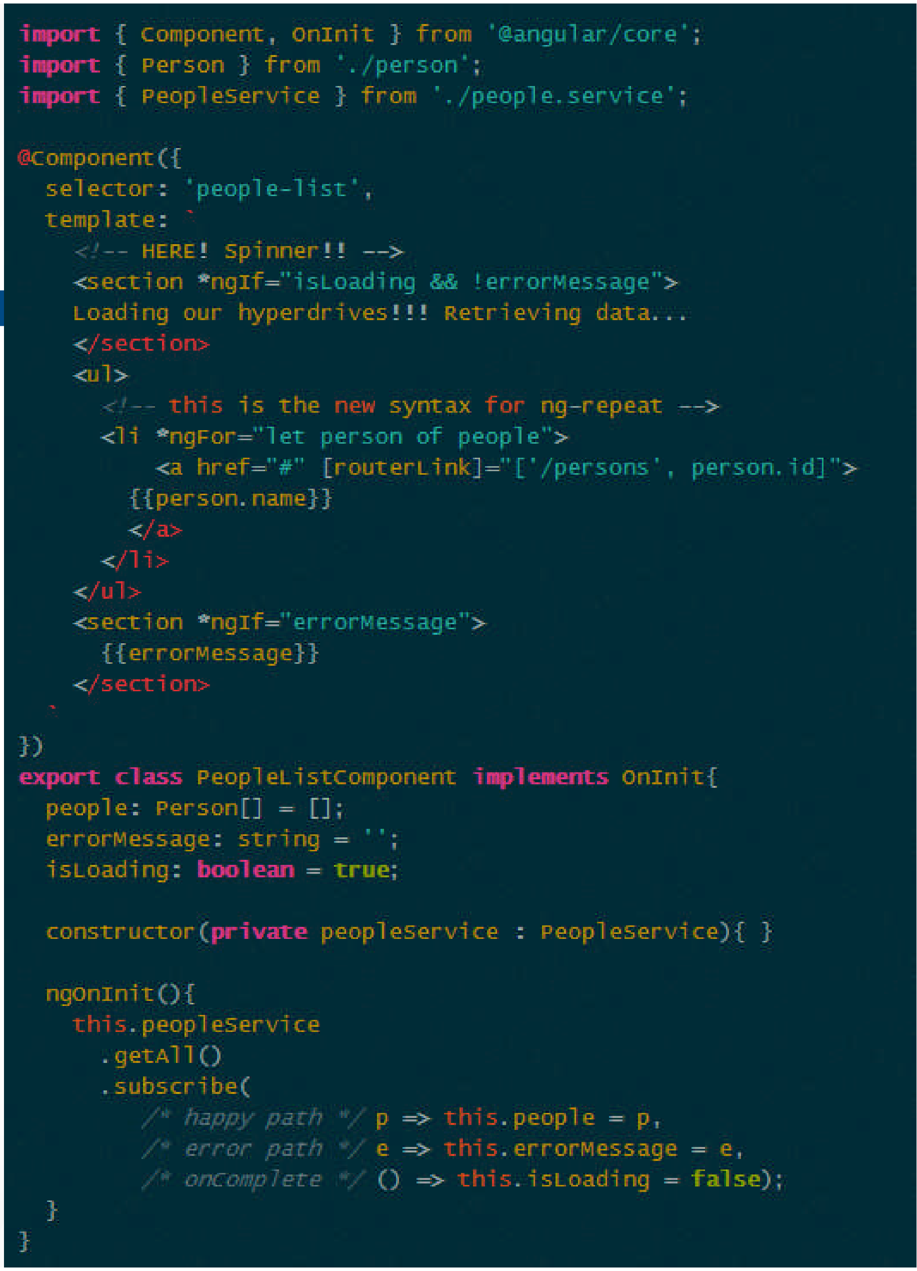


```
@Component({
  selector: 'people-list',
  template: `
    <ul>
      <!-- this is the new syntax for ng-repeat -->
      <li *ngFor="let person of people">
          <a href="#" [routerLink]="['/persons', person.id]">
        {{person.name}}
        </a>
      </li>
    </ul>
    <!-- HERE: added this error message -->
    <section *ngIf="errorMessage">
      {{errorMessage}}
    </section>
  `
})
export class PeopleListComponent implements OnInit{
  people: Person[] = [];
  errorMessage: string = '';

  constructor(private peopleService : PeopleService){ }

  ngOnInit(){
    this.peopleService
      .getAll()
      .subscribe(
         /* happy path */ p => this.people = p,
         /* error path */ e => this.errorMessage = e);
  }
}
```



Lab 13

- Additionally, the subscribe method lets you define a third argument with an onComplete function to execute when a request is gracefully completed. We can use it to, for instance, toggle a progress spinner or a *loading* message:

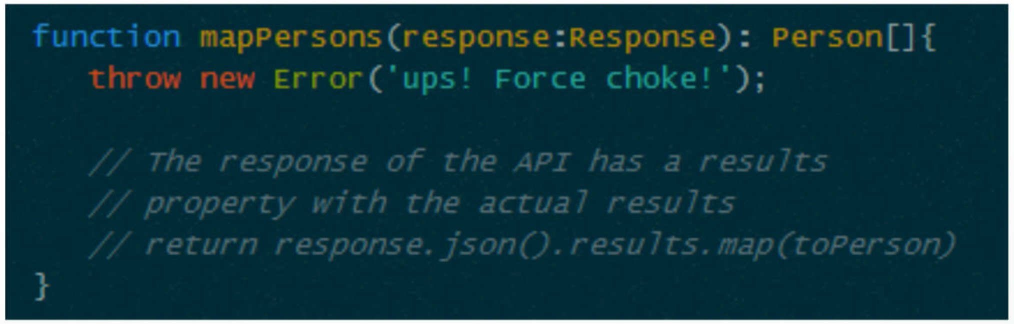


```
import { Component, OnInit } from '@angular/core';
import { Person } from './person';
import { PeopleService } from './people.service';

@Component({
  selector: 'people-list',
  template: `
    <!-- HERE! Spinner!! -->
    <section *ngIf="isLoading && !errorMessage">
    Loading our hyperdrives!!! Retrieving data...
    </section>
    <ul>
      <!-- this is the new syntax for ng-repeat -->
      <li *ngFor="let person of people">
          <a href="#" [routerLink]="['/persons', person.id]">
        {{person.name}}
          </a>
      </li>
    </ul>
    <section *ngIf="errorMessage">
      {{errorMessage}}
    </section>
  `
})
export class PeopleListComponent implements OnInit{
  people: Person[] = [];
  errorMessage: string = '';
  isLoading: boolean = true;

  constructor(private peopleService : PeopleService){ }

  ngOnInit(){
    this.peopleService
      .getAll()
      .subscribe(
         /* happy path */ p => this.people = p,
         /* error path */ e => this.errorMessage = e,
         /* onComplete */ () => this.isLoading = false);
  }
}
```

© Global Knowledge Training LLC. All rights reserved.

Lab 13

- We can test how our error handling is going to work by simulating a problem in the PeopleService
- Let’s throw an error within the mapPersons function (You can also misspell the API resource name to potato to get a HTTP 404 not found error but I thought the force choke was way cooler):



```
function mapPersons(response:Response): Person[]{
   throw new Error('ups! Force choke!');

   // The response of the API has a results
   // property with the actual results
   // return response.json().results.map(toPerson)
}
```

- Excellent! Now you can remove the fake error and continue forward