NETWORK COMMUNICATION

LESSON 03

SWAFE-01

NETWORK APPLICATION

OVERVIEW

- Most dynamic web applications gets data from other sources
- Hypertext Transfer Protocol (HTTP) and WebSocket Protocol is commonly used to serve data

CLIENT-SERVER MODEL

> *Client–server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.*

Client-server model—Wikipedia

HYPERTEXT TRANSFER PROTOCOL

THE HTTP/1.1 SPECIFICATION

- Defined in IETF RFC 2616
    - IETF – International Engineering Task Force
    - RFC – Request For Comments
- IETF is the Internet standards body
    - Developing open standards through open processes
    - An international community of network designers, operators, vendors, and researchers
    - Concerned with the evolution of the Internet architecture and the smooth operation of the Internet
- An RFC is a publication format used by IETF
    - Protocols, procedures and programs
    - Concepts
    - Meeting notes

HTTP METHODS

- `POST`—send data to the server
- `GET`—retrieve a resource from the server
- `HEAD`–retrieve resource metadata from the server
- `PUT`—send data to the server and update an existing entity
- `DELETE`—Remove a resource on the server
- `OPTIONS`–Get information about the communication options available

HTTP RESPONSE CODES

- `1xx` Informational–provisional response
- `2xx` Successful—indicates that the client's request was successfully received, understood, and accepted
- `3xx` Redirection—further action needs to be taken by the user agent in order to fulfill the request
- `4xx` Client error–intended for cases in which the client seems to have erred
- `5xx` Server error—indicate cases in which the server is aware that it has erred or is incapable of performing the request

HTTP SPECIFICATIONS

- Initial releases
  - RFC 2068 Hypertext Transfer Protocol -- HTTP/1.1 *January 1997*
  - RFC 2616 Hypertext Transfer Protocol -- HTTP/1.1 *June 1999*
- Updated and split up in June 2014
  - RFC 7230 HTTP/1.1: Message Syntax and Routing
  - RFC 7231 HTTP/1.1: Sematics and Content
  - RFC 7232 HTTP/1.1: Conditional Requests
  - RFC 7233 HTTP/1.1: Range Requests
  - RFC 7234 HTTP/1.1: Caching
  - RFC 7235 HTTP/1.1: Authentication
- Other versions
  - RFC 7540 Hypertext Transfer Protocol Version 2 (HTTP/2) *(May 2015)*
  - draft-ietf-quic-http-34 Hypertext Transfer Protocol Version 3 (HTTP/3) *(February 2021)*

HTTP IN ANGULAR

OVERVIEW

- The ability to request typed response objects
- Streamlined error handling
- Request and response interception
- Testability features

HTTPCLIENTMODULE

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

examples/lesson03-network-communication/projects/server-communication/src/app/app.module.ts

`HttpClient`

- Wraps HTTP requests in `Observable` objects
- Contains methods for some of the most used HTTP requests
  - `get()` returns a configured `GET` HTTP request
  - `post()` returns a configured `POST` HTTP request
  - `delete()` returns a configured `DELETE` HTTP request
  - `request()` returns a generic HTTP request
- Used in combination with RxJS operators to filter values and format to desired output
- Can be initiated with `subscribe()` or `async` pipe

SpaceService

```typescript
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Astronaut, LaunchVehicles } from 'lib-space';
import { Observable } from 'rxjs';
import { environment } from '../environments/environment';

@Injectable({
  providedIn: 'root'
})
export class SpaceService {
  rootUrl = `http://${environment.api.space.host}:${environment.api.space.port

  constructor(private http: HttpClient) { }

  getAstronauts(): Observable<Astronaut[]> {
    return this.http.get<Astronaut[]>(`${this.rootUrl}/astronauts`)
  }

  getLaunchVehicles(): Observable<LaunchVehicles[]> {
```

examples/lesson03-network-communication/projects/server-communication/src/app/space.service.ts

TYPED RESPONSE

OVERVIEW

- Consume output more easily and obvious
- Response types can act at type assertion at compile time
  - But is not guranteed that the server will respond with the expected type
  - Be sure to have proper error handling
- RxJS transformation
  - Use the `map` operator to transform response data as needed by the UI
  - Read the transformed data with the `async` pipe in the template

RESPONSE TYPE FOR `Astronaut`

```typescript
export interface Astronaut {
  name: string;
  nationality: string;
  organization: string;
  status?: string;
  born: number;
  died?: number;
  time_in_space?: number;
}
```

examples/lesson03-network-communication/projects/lib-space/src/lib/astronaut.type.ts

RESPONSE TYPE FOR `LaunchVehicle`

```typescript
export interface LaunchVehicles {
  name: string;
  country: string;
  height: number;
  status: string;
  mass: number;
  launch_history: {
    first: number;
    last?: number;
    total: number;
  }
}
```

examples/lesson03-network-communication/projects/lib-space/src/lib/launch-vehicle.type.ts

REQUESTING DATA FROM A SERVER

```
import { Component, OnInit } from '@angular/core';
import { LaunchVehicles } from 'lib-space';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { SpaceService } from './space.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {

  launchVehicles$: Observable<LaunchVehicles[]> | null = null;
  astronauts$: Observable<string[]> | null = null;

  constructor(private spaceService: SpaceService) { }

  ngOnInit() {
```

examples/lesson03-network-communication/projects/server-communication/src/app/app.component.ts

DISPLAY DATA IN TEMPLATES

```html
<h2>Astronauts</h2>
<ul>
  <li *ngFor="let astronaut of astronauts$ | async">
    {{ astronaut }}
  </li>
</ul>
<h2>Launch vehicles</h2>
<ul>
  <li *ngFor="let launchVehicle of launchVehicles$ | async">
    {{ launchVehicle | json }}
  </li>
<ul>
```

examples/lesson03-network-communication/projects/server-communication/src/app/app.component.html

SAME-ORIGIN SECURITY MODEL

&

CROSS-ORIGIN RESOURCE SHARING

OVERVIEW

- The Cross-Origin Resource Sharing (CORS) standard adds new HTTP headers, that let servers describe which origins are premitted to access data from a web browser
- CORS failures results in errors, but for security reasons, specifics about the error are not available to JavaScript

SAME-ORIGIN POLICY

- The same-origin policy is a critical security mechanism that restricts how documents and scripts loaded by one origin can interact with a resource from another origin
- Definition of an origin
  - A origin is a tuple comprised of `[protocol, host, port]`
    - `protocol` —The protocol used: `http://` , `https://` , etc.
    - `host` —The resource location: swafe-01.dk, ece.au.dk, etc.
    - `port` —The port number, if specified
- Implemented by all major browsers (Google Chrome, Firefox, Microsoft Edge, etc.)

CROSS-ORIGIN RESOURCE SHARING (CORS)

- The need to relax the same origin policy
- CORS is a protocol
- Simple and preflighted requests
  - Simple requests (`GET` , `POST` , and `HEAD`) does not trigger a CORS preflight
  - Methods like `PUT` , `DELETE` , and `PATCH` trigger a CORS preflight
- Preflight requests happens when requests that causes side-effects on server data
- Set in the header of HTTP requests and responses

GET

```
GET /cors/astronauts HTTP/1.1
Host: localhost:3000
Origin: http://localhost:4200
Referer: http://localhost:4200/
Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 742
Date: Fri, 10 Sep 2021 07:58:23 GMT

[{"name":"Neil Armstrong","nationality":"US","organization":"NASA","born":1930
```

PREFLIGHT REQUESTS

- A CORS preflight request checks to see if the host supports CORS protocol
- It is sent as an `OPTIONS` using three three HTTP request headers:
  - `Access-Control-Request-Method`—defines what HTTP method(s) will be used for the actual request
  - `Access-Control-Request-Headers`—defines what HTTP headers the client might send with the request
  - `Origin`—indicates where the request originates from
- A preflight response will contain
- Preflight request are automatically issued by the browser
  - In normal cases, this means that front-ends developers do not need to craft such requests themselves

PREFLIGHT REQUEST

```
OPTIONS /cors/astronauts HTTP/1.1
Host: localhost:3000
Accept: */*
Access-Control-Request-Method: DELETE
Origin: http://localhost:4200
Referer: http://localhost:4200/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,da-DK;q=0.8,da;q=0.7,fr;q=0.6

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Fri, 10 Sep 2021 08:25:35 GMT
```

PROXYING TO A BACKEND SERVER

- Proxying support from `webpack` development server is available
- Follow these steps to set up:
  - Create a file `proxy.conf.json` in the project `src` folder
  - Add the `proxyConfig` option to the `serve target`
  - Check service URLs in the code
- Remember to relaunch the application when changing proxy configuration (rerun `ng serve`)

proxy.conf.json

```
{
  "/api": {
    "target": "http://localhost:3000",
    "secure": false
  }
}
```

examples/lesson03-network-communication/projects/proxying/src/proxy.conf.json

```
... ,
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "production": {
      "browserTarget": "proxying:build:production"
    },
    "development": {
      "browserTarget": "proxying:build:development",
      "proxyConfig": "examples/lesson03-network-communication/projects/proxyin
    }
  },
  "defaultConfiguration": "development"
}, ...
```

examples/lesson03-network-communication/angular.json

ERROR HANDLING

OVERVIEW

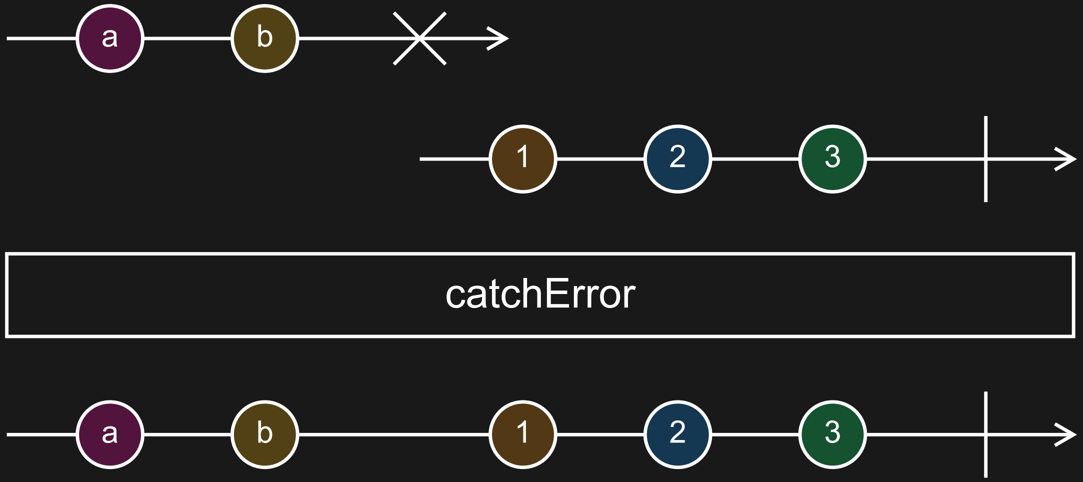
- If the request fails, HttpClient returns an `HttpErrorResponse` error object instead of a successfully response
- Two types of errors can occur:
  - Error responses The backend might reject a request and return a response with status code `4xx` or `5xx`
  - Client-side errors Network errors or unhandled exceptions thrown in RxJS operators
- Client-side error will have `status` set to `0`, error responses will have the HTTP status code (such as `404` Not Found, `500` Internal Server Error, etc.

RXJS OPERATORS

- `catchError` catches errors on the `Observable` to be handled by returning a new observable or throwing an error
- `retry` retries the `Observable` a predefined number of times. Defaults to `Infinity`
- `retryWhen` retries the `Observable` based on custom criteria

`catchError`

Catch errors that happens inside the stream



catchError

Catch errors that happens inside the stream

```
import { HttpErrorResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { Astronaut } from 'lib-space';
import { Observable, of } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { FaultyService } from '../faulty.service';
import { SpaceError } from '../space-error.type';

@Component({
  selector: 'app-catch-error',
  templateUrl: './catch-error.component.html',
  styleUrls: ['./catch-error.component.scss']
})
export class CatchErrorComponent implements OnInit {
  observable$: Observable<Astronaut[] | SpaceError> | null = null
  error = ''

  constructor(private faultyService: FaultyService) { }

```

examples/lesson03-network-communication/projects/error-handling/src/app/catch-error/catch-error.component.ts

retry

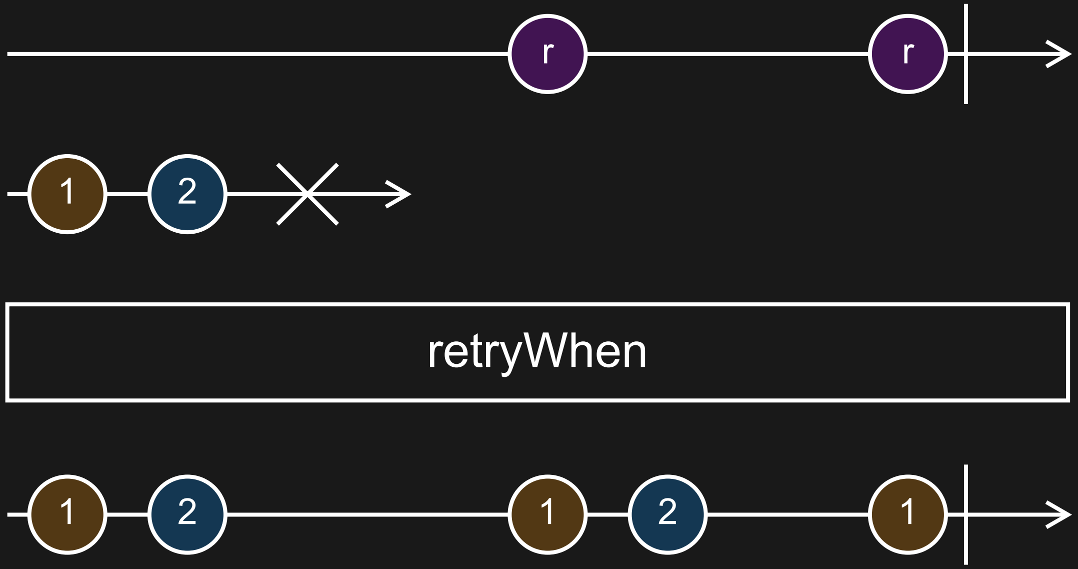
Retry an observable sequence should an error occur

```
import { Component, OnInit } from '@angular/core';
import { Astronaut } from 'lib-space';
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
import { FaultyCall } from '../faulty-call.type';
import { FaultyService } from '../faulty.service';
import { SpaceError } from '../space-error.type';

@Component({
selector: 'app-retry',
templateUrl: './retry.component.html',
styleUrls: ['./retry.component.scss']
})
export class RetryComponent implements OnInit {

  observable$: Observable<Astronaut[] | SpaceError> | null = null
  faults: FaultyCall[] = []

  constructor(private faultyService: FaultyService) { }
```

examples/lesson03-network-communication/projects/error-handling/src/app/retry/retry.component.ts

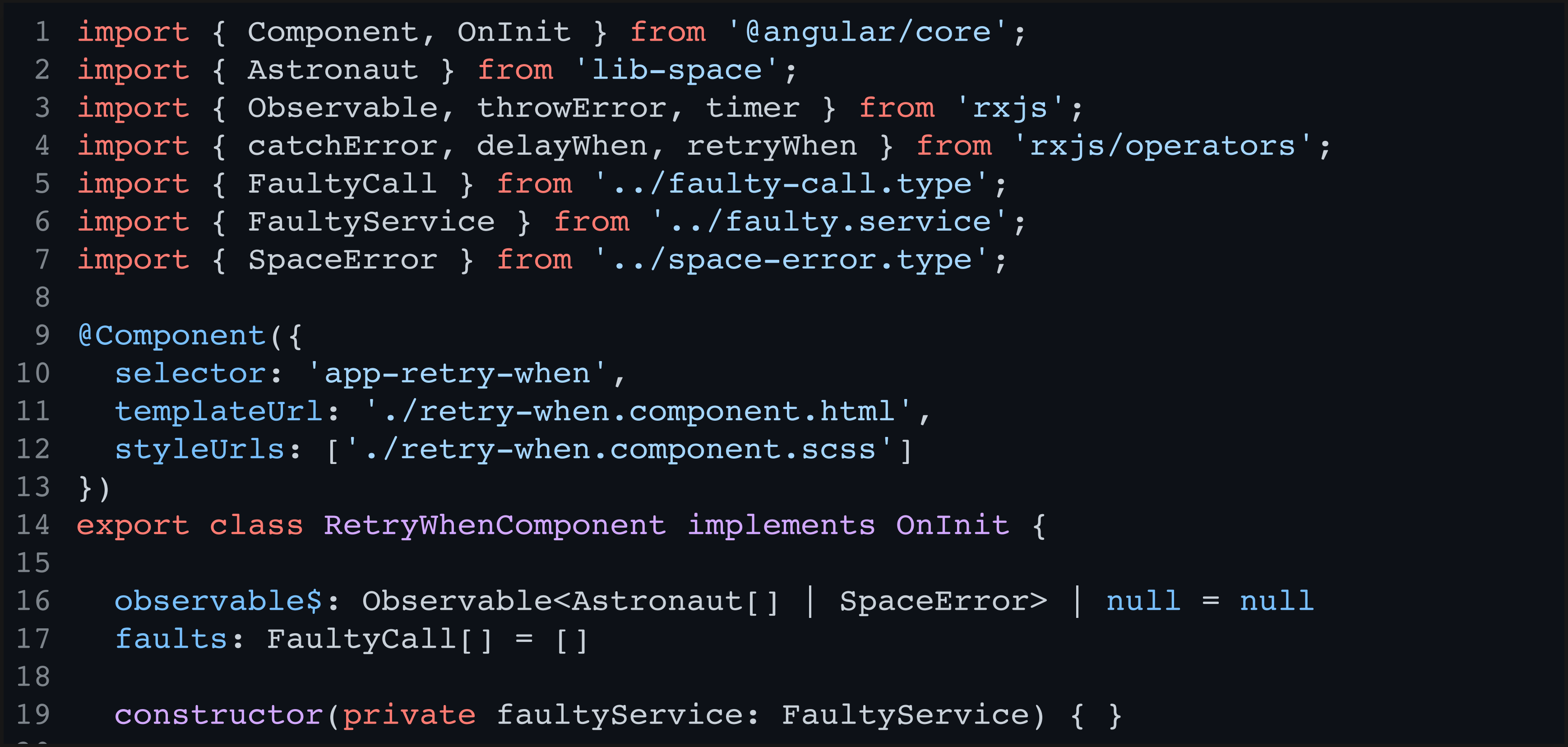
`retryWhen`

Retry an observable sequence on error based on custom criteria



retryWhen

Retry an observable sequence on error based on custom criteria



```
import { Component, OnInit } from '@angular/core';
import { Astronaut } from 'lib-space';
import { Observable, throwError, timer } from 'rxjs';
import { catchError, delayWhen, retryWhen } from 'rxjs/operators';
import { FaultyCall } from '../faulty-call.type';
import { FaultyService } from '../faulty.service';
import { SpaceError } from '../space-error.type';

@Component({
  selector: 'app-retry-when',
  templateUrl: './retry-when.component.html',
  styleUrls: ['./retry-when.component.scss']
})
export class RetryWhenComponent implements OnInit {

  observable$: Observable<Astronaut[] | SpaceError> | null = null
  faults: FaultyCall[] = []

  constructor(private faultyService: FaultyService) { }
```

examples/lesson03-network-communication/projects/error-handling/src/app/retry-when/retry-when.component.ts

JSON WEB TOKENS

OVERVIEW

- JSON Web Token (JWT) is a compact claims representation format
- A JWT consists of three partitions
  - `Header` contains type and signing information
  - `Payload` contains claims and other user information
  - `Signature` contains a hash of `header` and `payload`
- Some relevant IETF standards
  - RFC7519 JSON Web Token
  - RFC8725 JSON Web Token Best Current Practices

JWT CLAIMS

- `aud` Audience—what resource is this token intended for?
- `iss` Issuer—who issued the token?
- `iat` Issued At—identifies the time the claim was issued
- `nbf` Not Before–identifies the time before which the JWT must not be accepted
- `exp` Expires—identifies the time on or after which the JWT must not be accepted

BEST PRACTICES

- Validate JWTs on the client before using them
  - Use JSON Set URL (`jku`) or JSON Web Key (`jwk`) in the `header` for verification
- Always verify the issuer (`iss`) and audience (`sud`)
- Always add an expiration time (`exp`)

INTERCEPTION

OVERVIEW

- Request and response transformation
  - Before sending a request to the server
  - Before receiving a response in the application
- Perform implicit tasks
  - Such as setting an authentication token for all requests sent to the server
  - Alternatively implemented explicitly for every method call
- Multiple interceptors forms a forward-and-backwards chain of request/response handlers

`HttpInterceptor`

- All interceptors is managed by Angular's DI system (just like services)
- Interceptors are dependencies of `HttpClient`
  - Therefore, they must be provided in the same injector (or a parent of the injector) that provides `HttpClient`
  - If added after `HttpClient` is created, they are ignored
- Create a "barrel" file in `/app` to keep track of injectors
  - Contains an array of `HttpInterceptor` objects
  - Use `HTTP_INTERCEPTORS` multi-provider token when providing interceptors
- `HttpRequest` and `HttpResponse` are immutable
  - Cannot be modified after it is created
  - Use the `clone` method
- Check context with `HttpContext`
  - If only some requests/responses should be handled or passed on without modification

INTERCEPTOR

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
  HttpContextToken,
  HttpContext
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from '../service/auth.service';
import { switchMap } from 'rxjs/operators';

const AUTH = new HttpContextToken<boolean>(() => false)

export function auth() {
  return new HttpContext().set(AUTH, true)
}

@Injectable()
```

examples/lesson03-network-communication/projects/interception/src/app/http-interceptors/auth-token.interceptor.ts

SERVICE

```typescript
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Astronaut } from 'lib-space';
import { Observable } from 'rxjs';
import { auth } from '../http-interceptors/auth-token.interceptor';

@Injectable({
  providedIn: 'root'
})
export class SpaceService {

  rootUrl = `http://localhost:3000/auth`

  constructor(private http: HttpClient) { }

  astronauts(): Observable<Astronaut[]> {
    return this.http.get<Astronaut[]>(`${this.rootUrl}/astronauts`, {
      context: auth()
    })
  }
```

examples/lesson03-network-communication/projects/interception/src/app/service/space.service.ts

CALL FROM CLASS

```typescript
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { SpaceService } from './service/space.service';
import { Astronaut } from 'lib-space';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  token$: Observable<string> | null = null;
  astronauts$: Observable<Astronaut[]> | null = null;

  constructor(private spaceService: SpaceService) { }

  ngOnInit() {
    this.astronauts$ = this.spaceService.astronauts()
  }
```

examples/lesson03-network-communication/projects/interception/src/app/app.component.ts