Reactive streams in RxJS REST API design with Express.js

Michał Jabłoński

About me

Michał Jabłoński

/michaljabi

Full stack developer,

JavaScript / TypeScript trainer

2007 (X)HTML + CSS + JavaScript

2009 Flash + HTML + AS 2.0 + PHP

…

2011 Flex + AS 3.0 + Java EE + GWT + JavaScript

…

2017 Angular + React + JS ES6 + TypeScript + RxJS

Plan

- Intro: REST API design
  - HTTP verbs (methods) / status codes / headers - usage
  - Best API designing practices
- ExpressJS
  - Project architecture
  - ExpressJS middleware concept
  - Request handling (extracting): body / query params / headers,
  - Error handling
  - Setup ExpressJS with TypeScript
  - Development tooling (nodemon, eslint etc.)
  - Environment / production setup -> 12 factor app practices in JavaScript
- Debugging
  - Debugger tool in VSCode: usage in front-end (Angular) and back-end (Express) apps

Plan 2

- Introduction to reactive programming
  - functional programming in JavaScript
  - higher order functions
  - way of working with asynchronous code in JavaScript
- Reactive programming with RxJS library
  - concept of reactive programing in RxJS
  - way of making observables from different sources of data
  - subscriptions nad types of subjects in RxJS API
  - difference between hot and cold observables
  - practical usage and differences between operators methods by operator types (combination, creation, filtering, multicasting, transformation, utility)
  - Best practices and common mistakes using Observables
- Reactive streams in Angular
  - Stateful and Stateless services in application
  - Data exchange between services and components examples (benefits of using Observables in application)

Outline

01

REST API design

02

ExpressJS

03

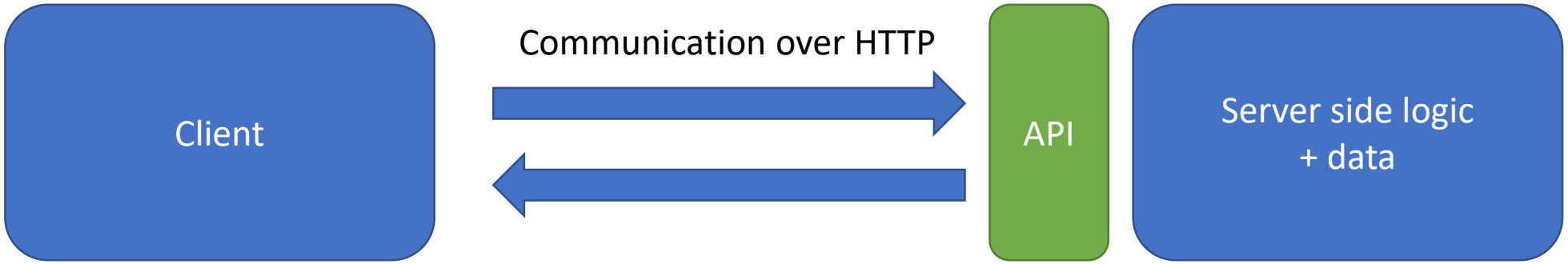
Debugging JavaScript

04

RxJS: Reactive programming

The server-side API

- Some public methods available on the Server
- We can manipulate / use the data from the Server via connected Client



What is REST ?

- REpresentational State Transfer
- Web service / server provides web resources which can be manipulated by stateless operations
- Each request should have all the information required to read / write / update data on the server
- Server does not hold a session for particular user requests

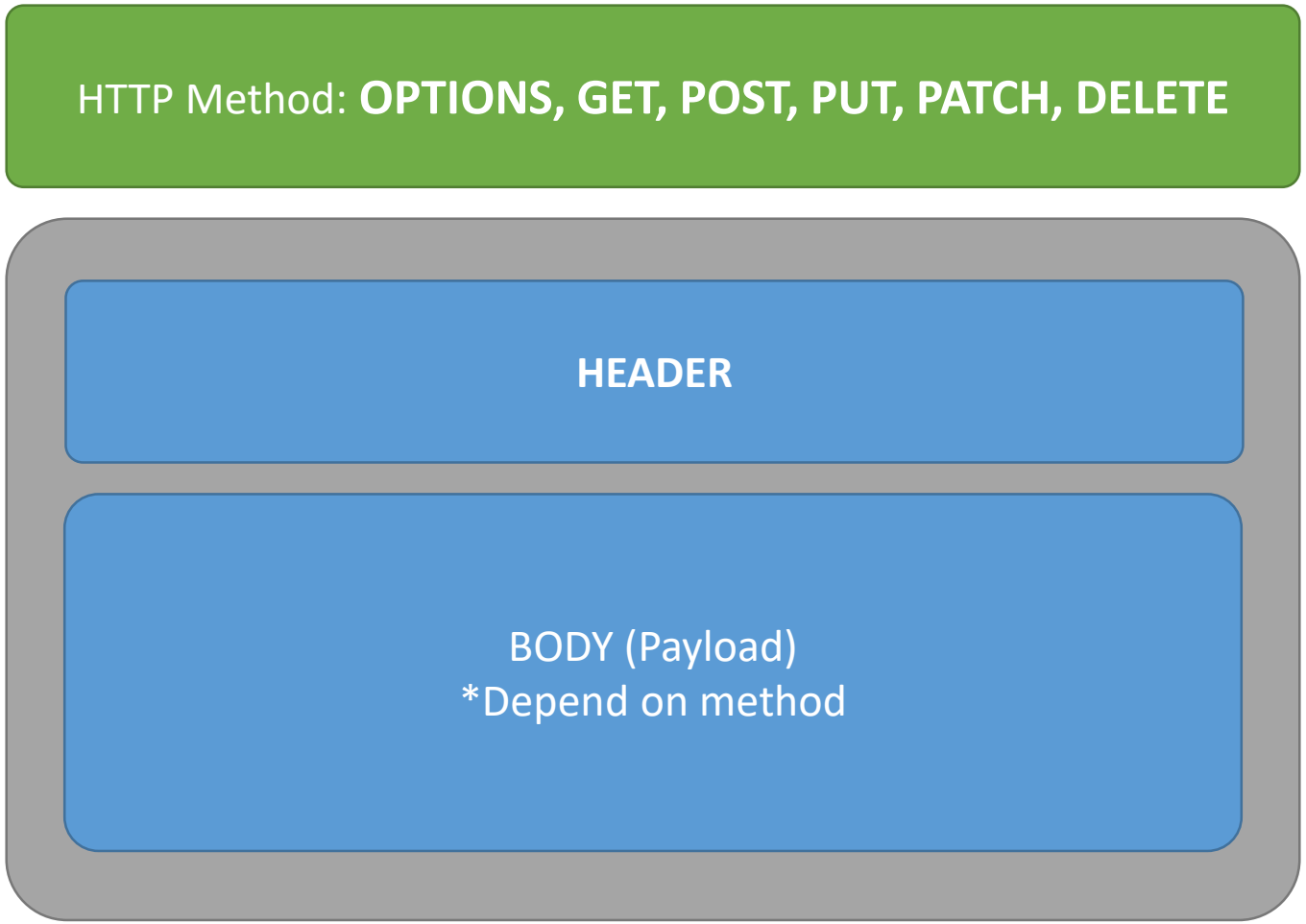
Request – Response model:

- HTTP protocol allow you to communicate with Web Server

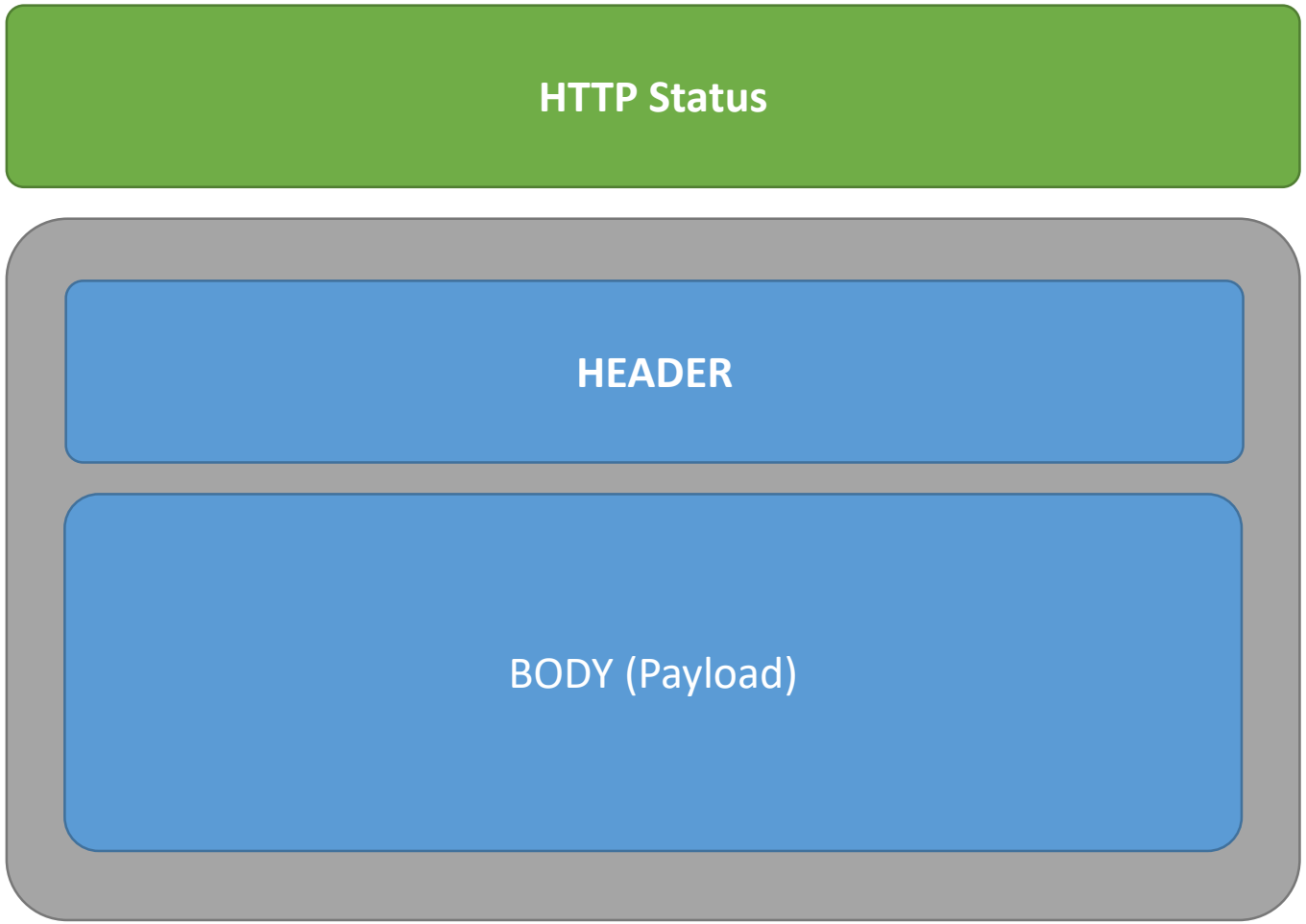
- In REST you will “query” or “command” Server to do something and take feedback from server with data or information about command success or failure
- Request and Response have a specific construction and intention of usage – depend on what kind of operation you want to perform

- That’s why you should follow some rules when crate your own API

The Request


HTTP Method: **OPTIONS, GET, POST, PUT, PATCH, DELETE**
HEADER
BODY (Payload)
*Depend on method

The Response



API design and its structure

- Beside Request information in header and body the Request URL can provide some additional query information used for:
  - Resource identification (ID)
  - Sorting data
  - Filtering data
  - Resource pagination (limit the data)
  - Choosing resource fields

- Response provide a HTTP status code (number) to indicate if data is resolved OK (200) or something went wrong (code from 400 up)

HTTP Request Methods:

Method	
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
HEAD	The HEAD method asks for a response identical to that of a GET request, but without the response body.
POST	The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.
PUT	The PUT method replaces all current representations of the target resource with the request payload.
DELETE	The DELETE method deletes the specified resource.
PATCH	The PATCH method is used to apply partial modifications to a resource.
OPTIONS	The OPTIONS method is used to describe the communication options for the target resource.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

HTTP Response Status Codes:

- 1xx - Informational
- 2xx - Successful
- 3xx - Redirection
- 4xx - Client Error
- 5xx - Server Error

4xx examples:

Code	Information
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
….-	

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

REST API DESIGN

Best practices

Why do we need to talk about good practices?

- Most of the communication via HTTP logic is settle, but still – when you are a person who will program this – final way of working is your decision.

- API have some additional information / way of working with set of HEADERS and way of URLs (API routes) are build. There are some conventions and policies that works !

- Your API need to be predictable at some core concepts

Web Resources

- Accessed via:
- URI (Uniform Resource Identifier)
- Uses URL (L – locator) and URN (N – name) to identify the resource

- Server API often refers to methods and URIs provided by Server to access the web resources
- URI example:

https://my-example-company.org/api/clients/23

CRUD methods in HTTP

- Some of the basic functionalities of REST services is a resource manipulation
- That's why there are some of the core methods to Create Read Update and Delete data (CRUD operations)

Method	Performed action	Rules
GET	**Fetch resources**	Safe (does not alter the state). Server only send the data back to the Client
POST	**Create new resource**	Use body to send new object to server
PUT	**Replace the resource**	Use body to replace some resource. Identified by URI
PATCH	**Update resource data (partial updates)**	Use body to update some resource fields. Identified by URI
DELETE	**Remove resource**	Without body. URI used to match resource for Removal

HTTP headers

- General header

  Headers applying to both requests and responses but with no relation to the data eventually transmitted in the body.

- Request header

  Headers containing more information about the resource to be fetched or about the client itself.

- Response header

  Headers with additional information about the response, like its location or about the server itself (name and version etc.).

- Entity header

  Headers containing more information about the body of the entity, like its content length or its MIME-type.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

API design practices checklist:

- ✓Use plural NOUNS in API URIs
- ✓ HTTP Methods should behave as intended
- ✓ Use headers to inform about: content-type, caching, authorization
- ✓ Use query params for: filtering, sorting, pagination
- ✓ Respect the HTTP status response codes (200, 300, 400, 500…)

GOAL: fetch list of users

method: **GET**

http://example.com/api/**users**

e.g. header:

Authorization: Barer xh72uey….

Content-Type: application/json

RULES:

- We use **noun in plural** for URI name
- Request should have **no body**
- Server **data should not change** (it is only query command !)

POSSIBLE RESPONSES:

- 200 OK status with data in the body as a JSON Array with collection of users (web resources)
- 401 Unauthorized – if Barer token is invalid or expired
- 403 Forbidden – token is valid, but you do not have permissions to see those resources

BAD: fetch list of users

method: **GET**

http://example.com/api/**getUsers**

e.g. header:

My-Auth: Barer xh72uey….

body:

{ “name”: “John”, “role”:”Admin”}

RESPONSE:

- Server respond with arrays of users
- Server adds a new user if present inside a body

PROBLEMS:

- Using a verb inside URI with camelCase instead of plural noun (users)
- Updating the data with doing both to the server – querying and inserting new user; GET request should not have body
- Using custom “My-Auth” header

GOAL: fetch list of admin users

method: **GET**

http://example.com/api/users?**role=admin&name=Jan**&**sort=+name**

FILTER:

- We place filtering inside query params
- Server will respond with the same rules as before – but with different body (filtered array of users)
- If there is no results – we should get an empty array

SORT:

- Special field in query params, despite the Web resource field name to sort by; it gives additional + or - sign to decide about ASC or DESC sorting
- If you want to sort by multiple fields, you can provide coma separated fields:

sort=+name,-role

GOAL: fetch one user with ID: 15

method: **GET**

http://example.com/api/users**/15**

PARAMS:

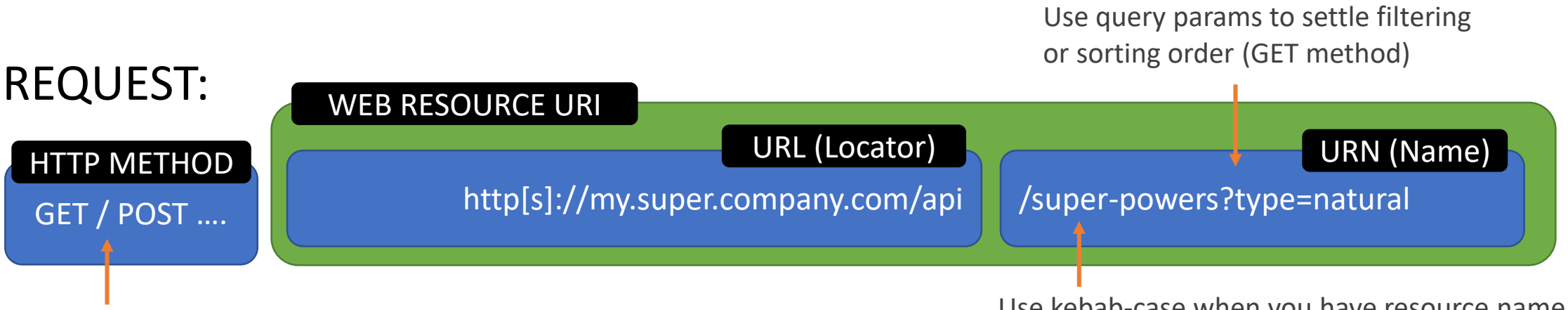
- The server should extract the ID from URI
- Notice that we should not end the path (URI) with additional slash /

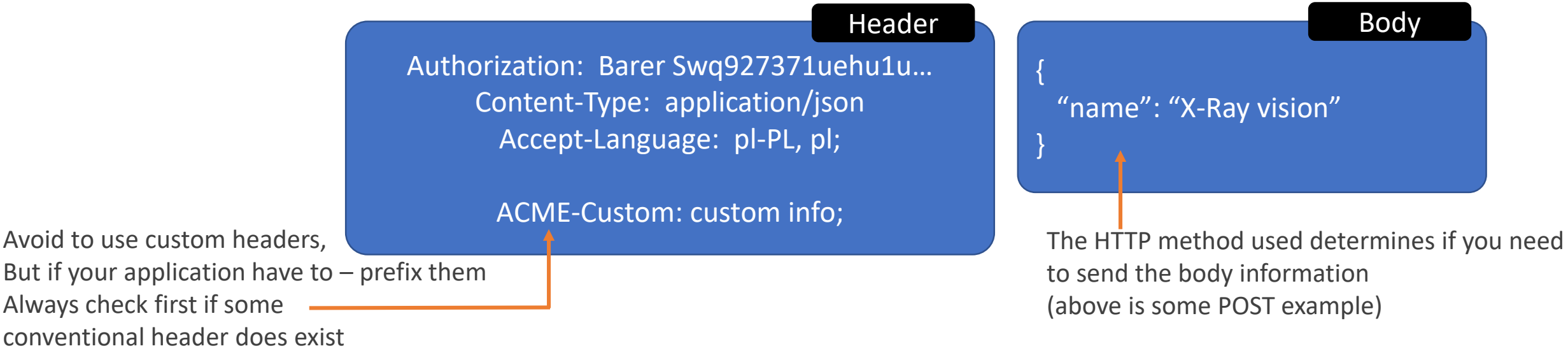
POSSIBLE RESPONSES:

- 200 OK - data in the body as a JSON Object with **one user**
- 401, 403… etc.
- 404 Not Found – often body provide meaningful error token or explanation

To sum up:

REQUEST:





GOAL: add new user

method: **POST**

http://example.com/api/**users**

e.g. header:

Authorization: Barer xh72uey….

Content-Type: application/json

body:

{ “name”: “Ana”, “role”:”Admin”}

RULES:

- Send POST to the whole collection URI
- Provide **body** with Resource - without id (indicate that is a new element)

POSSIBLE RESPONSES:

- 200 OK – body with added resource + its DB id
- 401 , 403 etc.

GOAL: update existing user

method: **PATCH**

http://example.com/api/**users/34**

e.g. header:

Authorization: Barer xh72uey….

Content-Type: application/json

body:

{ “role”:”User”}

RULES:

- Send **PATCH** to update only some of the fields inside the web resource
- Provide **body** with Resource - without id – it is given inside the URI

POSSIBLE RESPONSES:

- 200 OK – body with updated resource

GOAL: replace existing user

method: **PUT**
http://example.com/api/**users/34**

e.g. header:
Authorization: Barer xh72uey….
Content-Type: application/json

body:
{“name”: “Ana”, “role”:”User”}
or:
{“name”:”Olaf”}

RULES:

- Send **PUT** to update whole resource
- Provide **body** with Resource - without id – it is given inside the URI
- If you do not provide some of the fields – they will be *set as nulls*

POSSIBLE RESPONSES:

- 200 OK – body with replaced resource
- 401 , 403 etc.

GOAL: delete existing user

method: **DELETE**

http://example.com/api/**users/34**

e.g. header:

Authorization: Barer xh72uey….

Content-Type: application/json

RULES:

- Delete method should **not have a body**
- Provide some additional authorization inside header

POSSIBLE RESPONSES:

- 200 OK – body with deleted resource
- 204 No Content – delete successful but response does not provide the body

More samples

- Relations when one resource is a sub-collection of another web resource instance:
  - Get all authorizations for user id: 34

    http://example.com/api/**users/34/authorizations**
  - Get one authorization: 7 of user: 22

    http://example.com/api/**users/22/authorizations/7**

Express.js

Fast, unopinionated, minimalist web framework for Node.js

What is Express.js ?

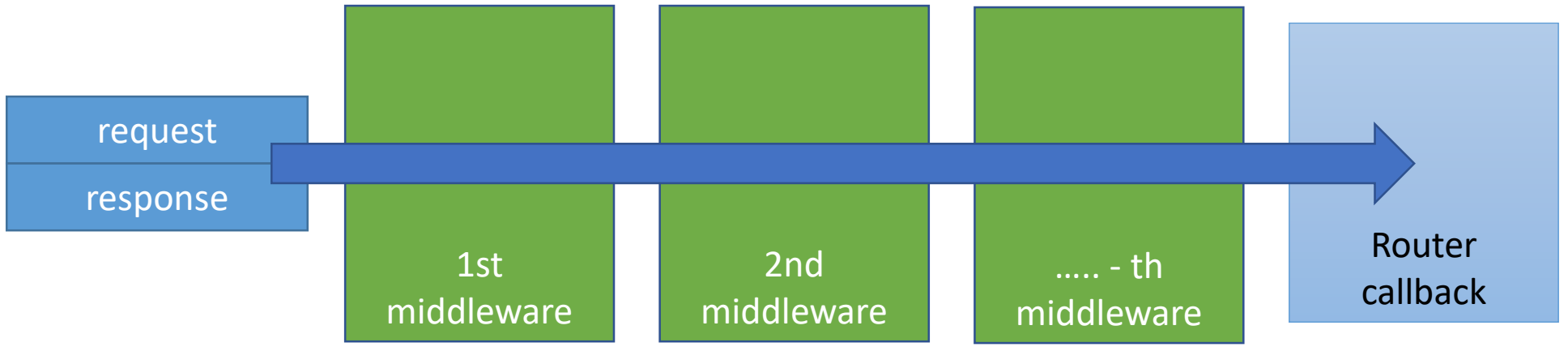
- Back – end framework for Node.js.
- It can create a fully featured HTTP server:
  - with serving static assets; or
  - dynamic web pages “templated html + JS + CSS”; or
  - making an REST API
- It can be empowered with middleware “plug-in” libraries
- Its very flexible and lightweight – because it not enforce you to have some extra complicated boilerplate to “achieve the goal”
- Many other “more complex” back-end frameworks use Express.js as a base library for their logic

What problems Express.js solves?

- Routing and defining application (REST) endpoints
- Extracting request data (query, payload, headers)
- Handling response (defining response)
- Usage of view engines (MVC model on server side)
- Scaling the logic with different functionalities (split the architecture)

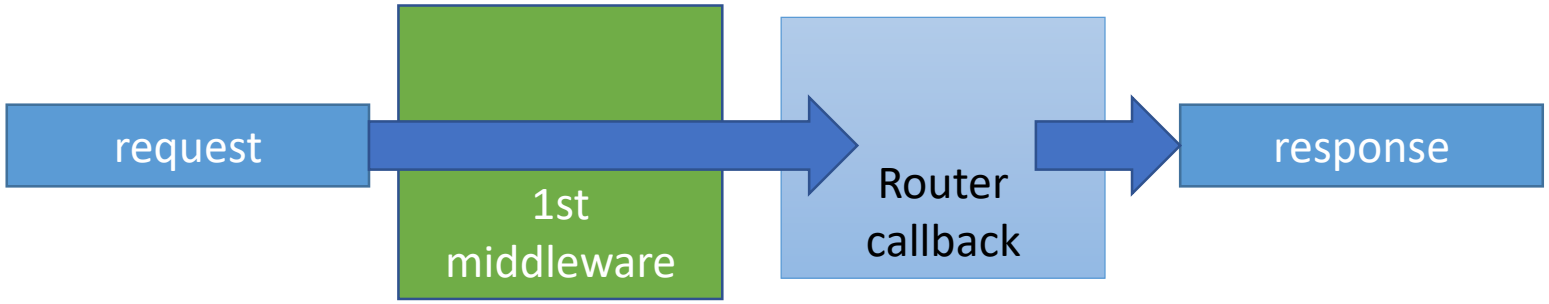
How Express.js works ?

- Before Request will achieve destination point – it will go trough the middleware functions
- Middleware can affect (mutate) the request or response

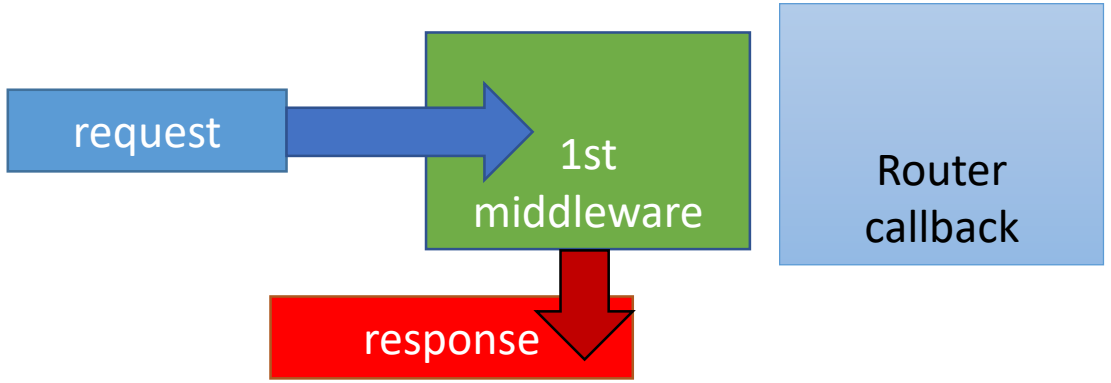


Middleware can:

- Pass the processing



- Stop the processing with Error



- Modify request or response object

Middleware levels

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

- **REMEMBER:**

The order of the middleware MATTERS. Place (level) where you attach the middleware will affect on the moment when it will fire.

Express.js good practices to consider:

- Using protective middleware for production:
  - Helmet
- Setting up CORS policy for REST API
  (Cross-Origin Resource Sharing)
- Prepare logging middleware
- Prepare error handling with dedicated server errors (consistent with HTTP statuses)

Express.js things to avoid:

- Do not write whole application in “one file” (logic separation)
- Do not forget to resolve middleware (next())
- Do not send response more than once (methods res.json() / res.send() – should be called once)

Architecture in express app

- Basically when you attach the routes with flexibility that Router() in express.js application you can see that you can separate this routing to other JavaScript modules.
  - You can make one routing file which will hold all the information about whole routing, or have few those kind of files for each part of application (application domain)
- Then in each file connected with the routing to some Web Resources you can make so called “Controller”.
- Controller will have REST end-points related to Resource/Domain logic

Example: Domain Driven Design

- Your application can be divided in to separate domains
- Each domain will handle its own logic – and has own controllers (REST API endpoints). Logic can be “façade” behind the services – which will provide the public API for the end-points.
- Node application can emit the events. That might be the way for communication between domains. That approach will separate domain logic only for itself – and show the registry “who” and “what for” want to use that domain data.

12 factor app principles

I. Codebase
One codebase tracked in revision control, many deploys
II. Dependencies
Explicitly declare and isolate dependencies
III. Config
Store config in the environment
IV. Backing services
Treat backing services as attached resources
V. Build, release, run
Strictly separate build and run stages
VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding
VIII. Concurrency
Scale out via the process model
IX. Disposability
Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity
Keep development, staging, and production as similar as possible
XI. Logs
Treat logs as event streams
XII. Admin processes
Run admin/management tasks as one-off processes

https://12factor.net/

12 factor app principles

- Many of that factors are obvious for developers, e.g.:
  - You have a **GIT repository** for codebase and many deploys like: production, stage, demo, lab etc.
  - Dependencies for installing and running the app will be held in **package.json**
  - Backing services are attached as **resources** (so our app will have to connect to them – they are independent deployment [e.g. DB, WebServices, 3-rd patry APIs)
  - Build and release stage should be separated (you build your project e.g. – /transpilation process/ and then add a config to release)

12 factor app principles

- Things that you need to consider (not such a obvious decision):
  - **III. Config and VI. Processes**
    - Way to handle the node ENV (environment) variables. In order to separate the dynamic configuration (different config values depend on deployment – production, stage, lab … etc.)
  - **XI. Logs**
    - Using logging mechanism inside your app (console.log – it’s a side effect !)
  - **IX. Disposability and XII. Admin processes**
    - Way to manage startup and shutdown and treat your application as a process on the working server. That is not resolved by node application “out of the box”.

Express on production

- Consider using .env files to resolve environment variables

- You need to have process manager to run your node application
  - PM2: https://pm2.io/
  - Forever: https://github.com/foreverjs/forever
  - Strong Loop PM: http://strong-pm.io/

- Example:
  - PM2 have a ecosystem files (json configs) where you can define all the NODE_ENV variables

Express on production

- You have some loggers out there, to manage server logging messages:
  - Morgan
    https://www.npmjs.com/package/morgan
  - Winston
    https://www.npmjs.com/package/winston
    https://www.npmjs.com/package/express-winston

Express with TypeScript?

- Since TypeScript is a superset of JavaScript
- It is just an additional step for transpilation process

- Additionally you need to provide .d.ts files for extended Request and Response objects.

- More details inside the typescript example (attached to this course).

Debugging JavaScript

Server- (Express.js) and Client- (Angular) side

Server side (e.g. Express.js) - VSC debugging

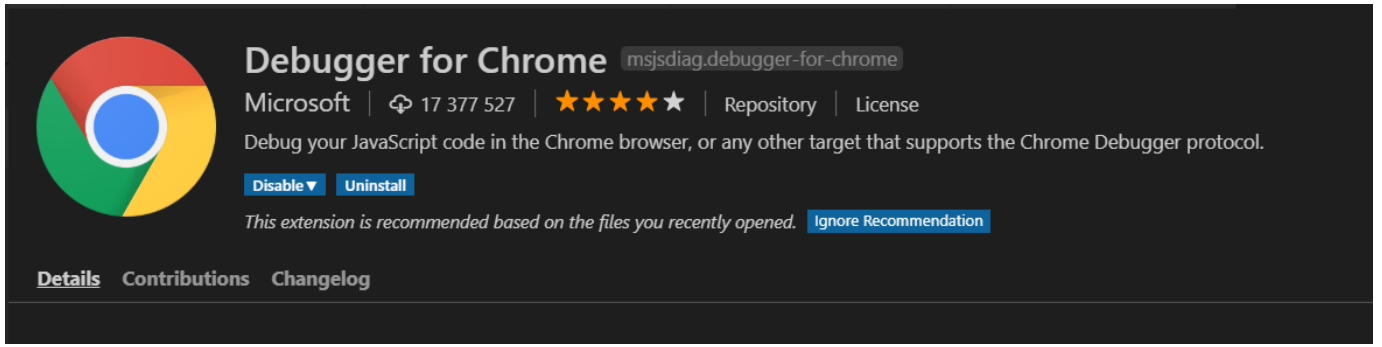
- Tools needed:
  - Debugger in VSC is already prepared for node app.
  - Example TS config:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Debug Project",
            "program": "${workspaceFolder}\\src\\index.ts",
            "preLaunchTask": "npm: ts",
            "sourceMaps": true,
            "smartStep": true,
            "internalConsoleOptions": "openOnSessionStart",
            "outFiles": [
                "${workspaceFolder}/dist/**/*.js"
            ]
        }
    ]
}
```

  - SourceMaps if you transpile code (e.g. TypeScript)
  - Also with TypeScript a “preLaunchTask” need to be defined to transpile code first

Client side (e.g. Angular app) - VSC debugging

- Tools needed:
  - Debugger for Chrome



  - Config:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "chrome",
            "request": "launch",
            "name": "Launch Chrome against localhost",
            "url": "http://localhost:4200",
            "webRoot": "${workspaceFolder}"
        }
    ]
}
```

  - Point to app port (4200)
  - SourceMaps enabled in project

Functional Programming

Introduction to reactive world.

Functional programming

- Use: PURE FUNCTIONS
- Rules:
  - Do not mutate input data!
  - Do not put inner dependencies inside the function
  - Function should aim to do only one task and be shortest possible
  - For the same input data – it should produce the same output data - the behavior is „predictable”
- Usage of Higher Order Functions
  - Function take other function as a parameter and can extend / change the output

Functional code

- Functional programming is commonly used by array methods like: map, filter, reduce, every – etc.

- Higher Order function accepts a function parameter and most commonly it realizes some logic on each element of the array.

- Example:

[1, 2, 3, 4].map(x => x *2) will make the: [2, 4, 6, 8]

Dealing with asynchronous code

Introduction to reactive world.

Callbacks and drawbacks

- If function can accept another function as a parameter, natural way to deal with asynchronous code in JavaScriptare so called "Callbacks"

- We put function Z() to function Y()
- Function Y() is making some asynchronous operation and after that it calls function Z()
- Notice that result of asynchronous operation can be passed to function Z() as an parameter. Z(result)

- More practical examples - inside coding exercises.

- Unfortunately to many callbacks (one async operation after another) lead us to make so called “callback of hell”

I Promise, I will see you .then()

- With problem of nesting complicated constructions of the callbacks and producing "spaghetti code" which is hard to maintain and reason about; Here there comes: Promises

- The Idea is behind flattening of the asynchronous calls by using .then() method attached in "fluent API" manner.

- Function with .then(x) will wrap its output with another Promise, so you can call another .then() method etc.:

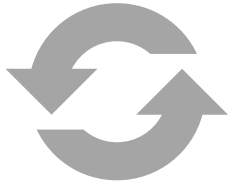
  .then(...).then(...).then(...)

- What is more interesting is concept of having one error handler: .catch() method, which return any error catched despite place where in "then chain" it occurs.

What things do we need to understand before entering Reactive concepts:

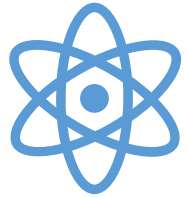
Arrays

(element collections)



Asynchronous Programming (Callbacks, Events)

Functional Programming (mapping, filtering, reducing …)



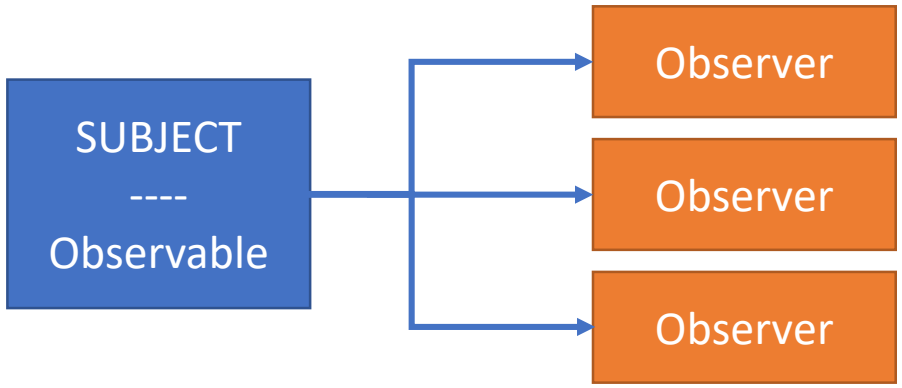
Function composition

Reactive programming

With RxJS library

The Observer pattern

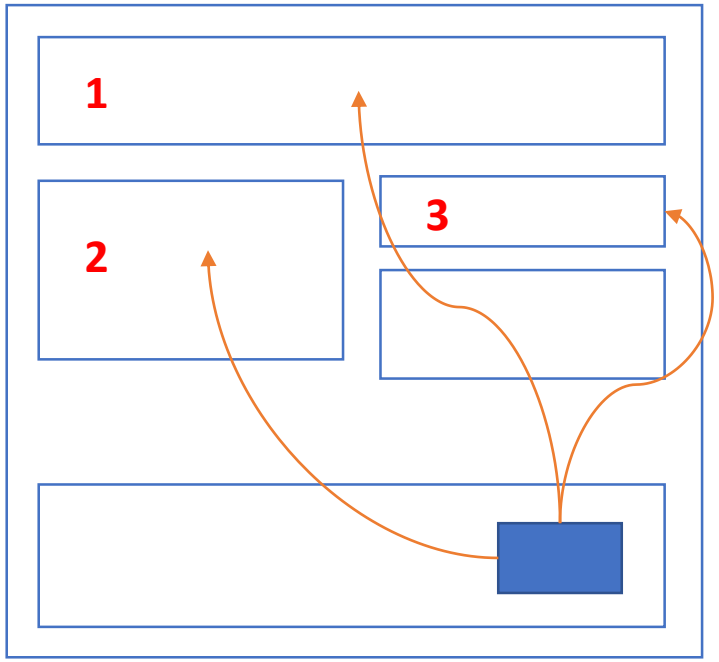
- Sending and receiving values via the streams is based on the "Observer Pattern" concept.



- Subject is emitting values to every Observer
- Each of Observers need to "subscribe" in order to receive values!
- Each of Observers can resign from subscription "unsubscribe"

The Observer pattern

- It works really well on front-end, where e.g. few objects visible on the website shows information based on the same data.
- When the source updates those information, all components can update what they are viewing



Where it can works?

- Its not a framework is a **library** (like underscore or jquery).
- It is not attached to any additional framework which need to be used

- It can be used both:

  front-end or back-end

  (it not depends on Node.js or Browser objects)

Practical RxJS Guide

How RxJS looks in practice

- Lib which can be used nearly everywhere
  - installation via npm : **npm install rxjs -S**
  - include as **<script>** with **src** and link from CDN
- Observable stream can be created from anything
  - You decide how the logic goes (similar to Promises)
- Observables are lazy (opposite to Promises)
- You can use many operators to alter the behavior of the final stream
  - Best way to learn about the operators is by understand the marble diagrams
- You can also alter behavior of the subscription

Hot vs. Cold Observable

- Streams can behave COLD or HOT.

COLD:

- That behavior of Observable cause all Subscribers to receive the same emissions since the sequence starts.
- You can compare this to the movie form YouTube Channel. When me and my friend both click "play" buttons on the movie, but in different time. We both will see the movie from 1st to last frame.

HOT:

- Every subscriber receive actual stream emission
- You can compare this to "live" movies on YT. When situation above is same, but movie is "live" - then me and my friend will recieve move from actual frame
- It works like Event Emitter

Subject

- You can create a hot observable with Subject.
- Subject provide you additional control over emitted data. You have a access to ".next()" method which help you achieve that.
- Subject can be an Observer (you can pass it to subsciption).
- In RxJS you can make it like that:

```
const subject = new Subject();
```

How, what is that, why operators ?

- It is basically what rises the functionality of RxJS. Thanks to use them you can compose functions in order to achieve desired behavior on the stream

- Thanks to operators we can: filter, combine, agregate and transform the streams

- W bibliotece RxJS jesteśmy w stanie napisać własny operator. Realizujący zdaną przez nas logikę.

How to understand operators

- Everyone who understand well how Higher Order Functions works form known array methods (map, filter, reduce, etc.) can also imagine way of operators usage.
- Operator works on observable attached to it and resolves to new observable object
- Thanks to that operators have so called "pipeable" feature.
- You can make a pipeline from them.
- That uses simple mechanism to pass result of one function as an input to another
- Here you must follow the rules from functional programming to not mutate the data.

Can we see it ?

- Yes, to understand how particular operator works on stream, you can use so called "Marble Diagrams“

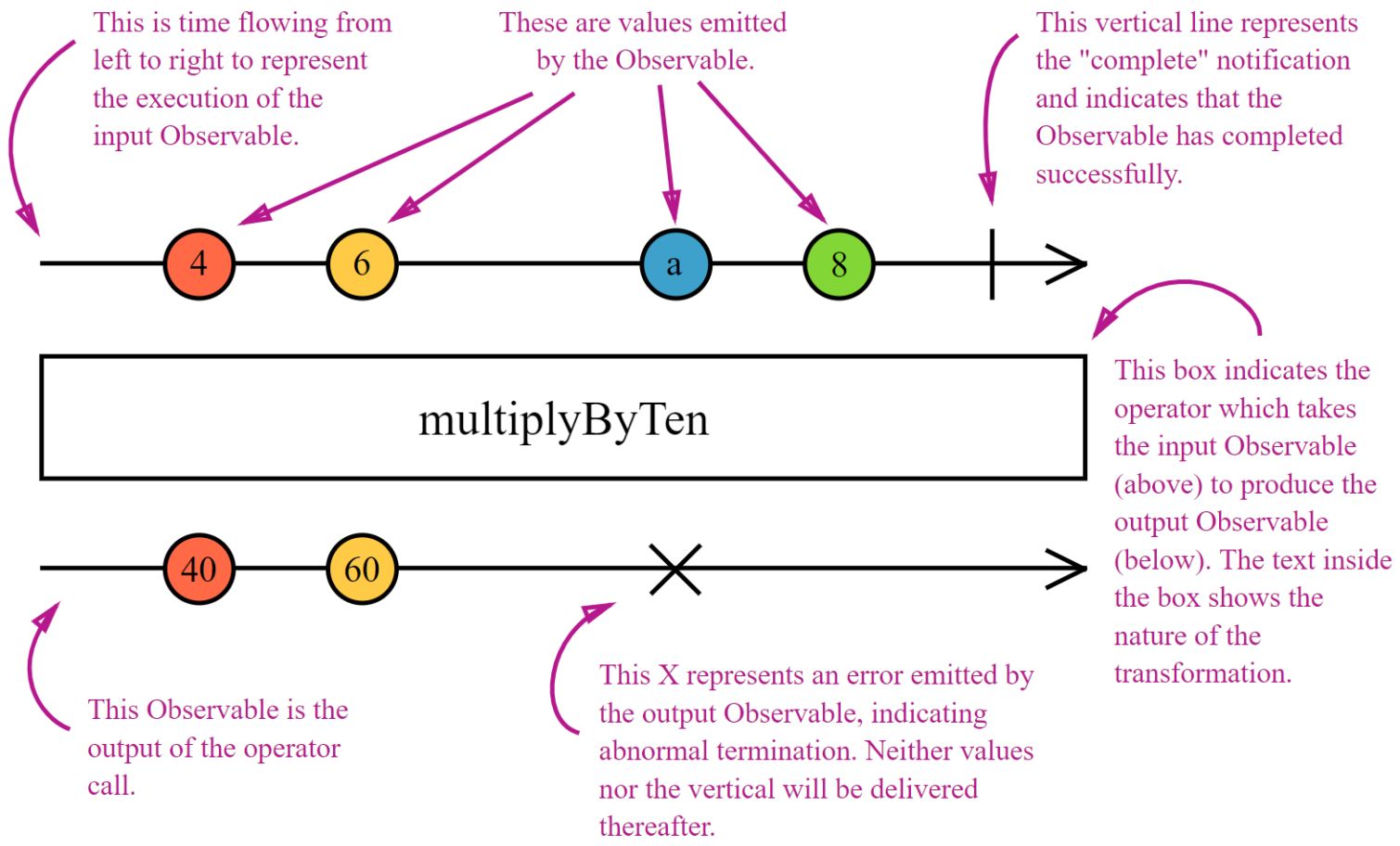
http://rxmarbles.com/

- You can see those things interactively. The streams with marbles - you can move them on the axis to see how result stream behave afterwards

- Good examples and explanations of operators in RxJS you can find on:

https://www.learnrxjs.io/operators/

The RxJS Marbles idea:



https://github.com/ReactiveX/rxjs/blob/master/doc/operators.md

Stream creation

- There are so called "static operators". Some of them are working like "helpers" for Observable creation.

- Example with "fromEvent":

```
const click$ = fromEvent(document, ’click’)
```

- click$ is now a stream of „click” events from „document” object on page

- You can distinguish two types of operators:

static operators - used for stream creation or combination

instance operators - the “pipeable ones“ , to work with existing streams

Stream creation

- **create**
- **empty**
- **from**
- **fromEvent**
- **interval**
- **of**
- **range**

- List of all: http://reactivex.io/rxjs/manual/overview.html#creation-operators

Combination

The combination operators allow the joining of information from multiple observables.

- **combineLatest**
- **concat**
- **forkJoin**
- **merge**
- **mergeAll**
- **startWith**
- **withLatestFrom**
- **zip**

- List:

http://reactivex.io/rxjs/manual/overview.html#combination-operators

Transformation & filtering

- **concatMap**
- **map**
- **mapTo**
- **mergeMap / flatMap**
- **pluck**
- **reduce**
- **scan**
- **switchMap**

- **debounceTime**
- **distinctUntilChanged**
- **filter**
- **skipWhile**
- **take**
- **takeUntil**

- List :

http://reactivex.io/rxjs/manual/overview.html#transformation-operators
http://reactivex.io/rxjs/manual/overview.html#filtering-operators

Utility

- delay
- delayWhen
- finalize
- tap

- Complete list:

http://reactivex.io/rxjs/manual/overview.html#utility-operators

Common mistakes, things to avoid

- Nesting subscriptions

  (that lead us to have "observable of hell")
- Data mutation (data from stream is not treated as read-only)
- Instances of Subject available with public access
  (many sources of truth)

- Subject overuse (using subject where normal Observables
  can be used)

Reactive streams in Angular

Why and How we can use RxJS in Angular front-end applications

Angular separates the business logic

- Components have mainly a “presentation” function in Angular.
- Logic behind component should be connected with its view
  - It is more “local” logic to the component itself
- Business logic and application state should be separated via services
- Services are communicating between each other and components with RxJS