Week 06: REST API, OpenAPI, Redis

Agenda

```
> GET /info/agenda
> Host: tutor.pb138

< HTTP/1.1 200
< Content-Type: text/markdown;charset=utf-8
< Server: Tutor
<
< # Seminar outline
<
< - Prerequisites
< - HTTP methods
< - HTTP response status codes
< - REST API modelling activity
< - Express.js
< - Runtime validation with Zod
< - OpenAPI
< - Demo
<   - REST, Swagger, Redis repository
```

Prerequisites

- Working Docker/Podman
- A `redis:latest` Docker image (either you already have it pulled as per email, or you can pull it now, **before we start working**)
- As usual, a working Node.js installation

HTTP methods in the context of REST - a quick recap

Method	Use-case
GET	Obtain a resource from the server
POST	Submit data (resource) to the server
PUT	Replace a resource on the server
PATCH	Replace a part of the resource on the server
DELETE	Delete a resource from the server

Requests can be

- Safe: Not altering the state of the server at all (all safe requests are also idempotent).
- Idempotent: Making one request results in the same final effect as making multiple requests of the same kind. (Example: One DELETE of an existing resource results in the resource being deleted, if the client has the credentials. Multiple DELETE requests of the same resource with proper credentials also result in the resource being deleted, even though from the second request onwards the HTTP status code changes)
- Unsafe: Making the request changes the state of the server, or creates some side effects.

HTTP response status codes - an even quicker recap

When designing an API, your API communicates not only with the data you send, but also with HTTP status codes. To ensure the correct behaviour of the apps that consume your API (might be only your app, but also some general scraping tools that rely on status codes), you must use correct status codes to indicate the status in response to the request the client has made.

Status code	Use-case
200	Generic way to say that the request succeeded
201	Resource created
204	The operation was successful and did not retrieve data
301	The resource has moved somewhere else
307, 308	Redirect responses
400	Client has made a bad request
401	Not authorized for this operation
403	Not allowed to proceed with the operation with current authentication
500	Generic error on the server-side (usually don't want to reveal more information than absolutely necessary)

For more information, visit Mozilla documentation of the HTTP response status codes. Also, there is an elaborate list of cat images explaining different HTTP response status codes.

Let's learn by modelling an e-commerce API

Imagine a situation, where you model an API for an e-commerce website, such as alza.cz. What resources does the site have? Focus on the user part. Discuss in groups of three to four people.

- Identify resources
- What methods will the API expose over those resources and what URIs will exist??
  - *Hint:* Imagine the website, think about the main features that the user can do with the web app, transform them into operations on resources
- Let's name the resources according to the methodology presented in the lecture and create routes for them

Počítače a notebooky

Nejnovější články

Více článků »

Let's learn by modelling an e-commerce API

What routes have you created? What HTTP methods have you used? Your tutor(s) might also reveal their solution.

Potential solution

Note: this solution is not complete, it's just a hint at how the API could be designed. **We will work with a slightly modified version of this during the demo.**

Resources

- `product`
  - Products also have some additional resources: `photo`s, `review`s, etc.
- `category`
- `user`
- ...

Routes & Methods

- `/products` - GET with query parameters as filters
- `/product` - POST for creating a new product
- `/product/{productId}` - GET for product detail; PUT, DELETE for product administration
- `/categories` - GET for getting all categories
- `/category` - POST for creating a new category
- `/category/{categoryIdentifier}` - GET subcategories of the category; PUT, DELETE - for category administration

... and we could go on until we have the fully working backend.

Express.js - a Web Application Framework

- A Node.js framework that allows quickly building web applications and REST APIs with Node.js
- Provides a very minimal, precise set of tools necessary for creating web applications
- Used by many other JS/TS frameworks as their backbone

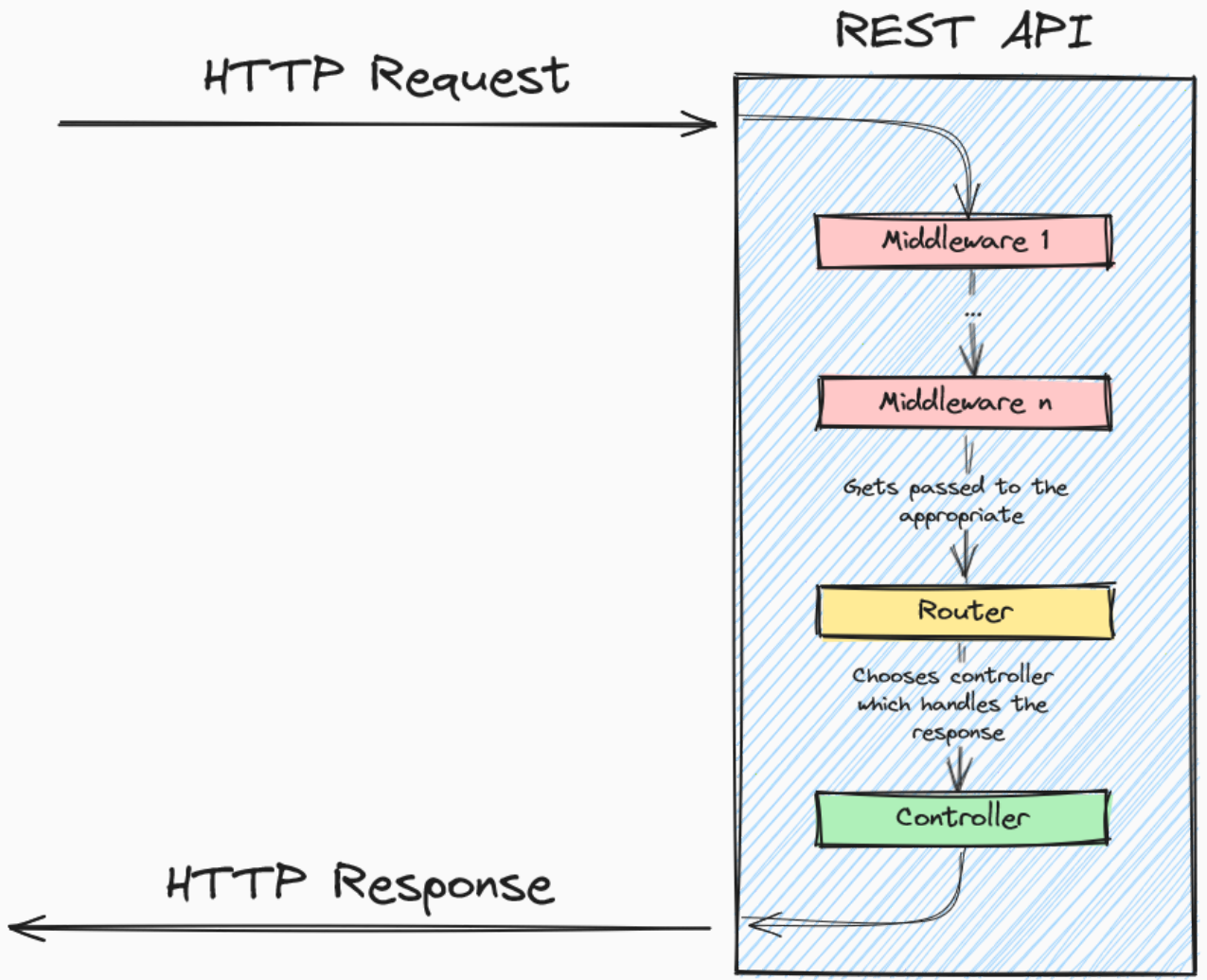
Express.js - Install

```
npm i express
# TypeScript support:
npm i -D @types/express
```

Adding Express to the code

```
import express, { Express, Request, Response } from 'express';

const app: Express = express();
const port = 8080;

app.listen(port, () => {
  console.log(`Server is running at https://localhost:${port}`);
});
```

Express - middlewares, routes and controllers

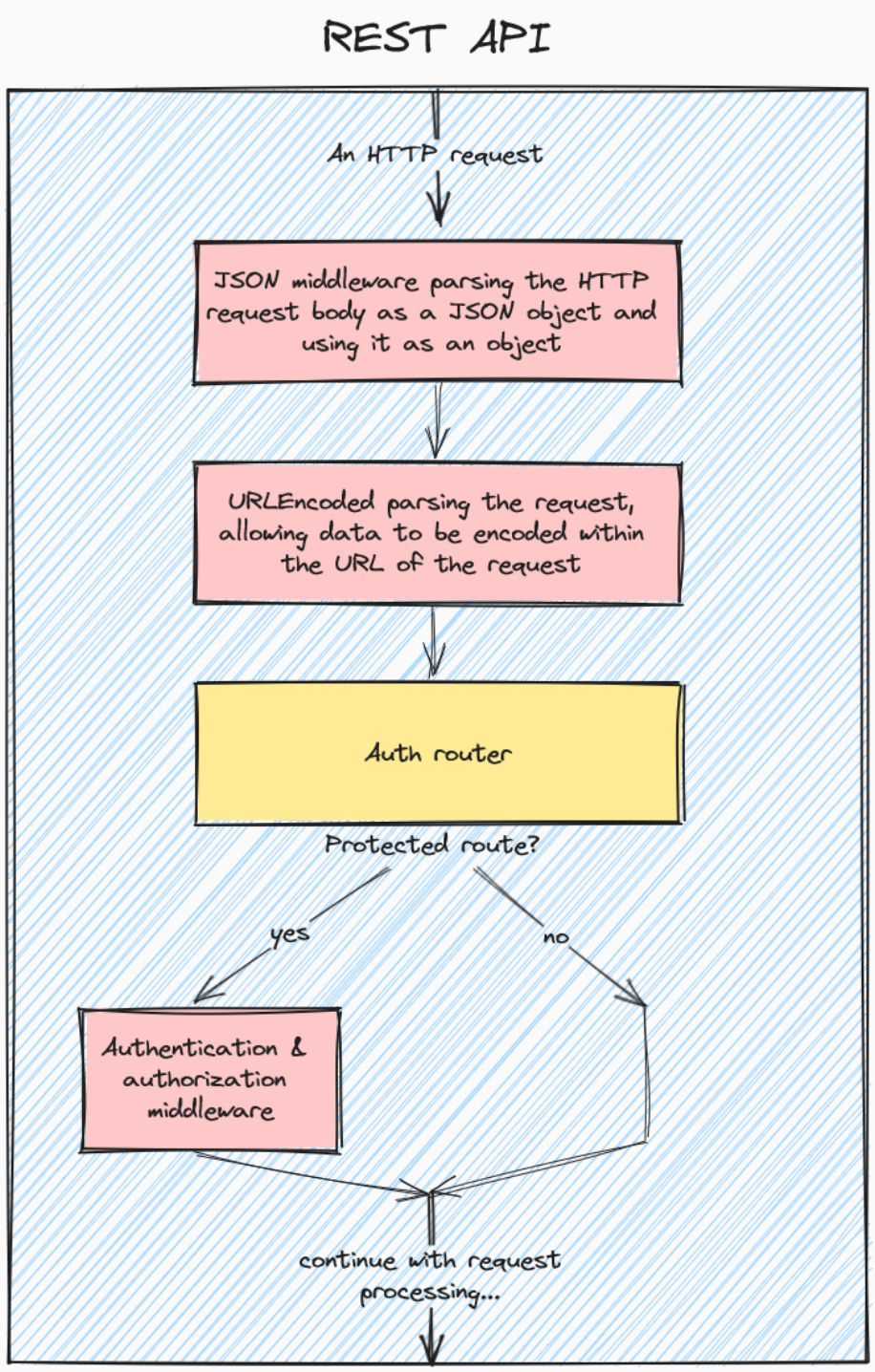


Express - middlewares

- The data is firstly processed by a pipeline of functions called middlewares
- These functions can, for example, check privileges or handle things that need to happen to every request before it is processed individually

```
const app: Express = express();
/* Middlewares: */
api.use(express.json());
api.use(express.urlencoded({ extended: true }));
```

Express - middlewares

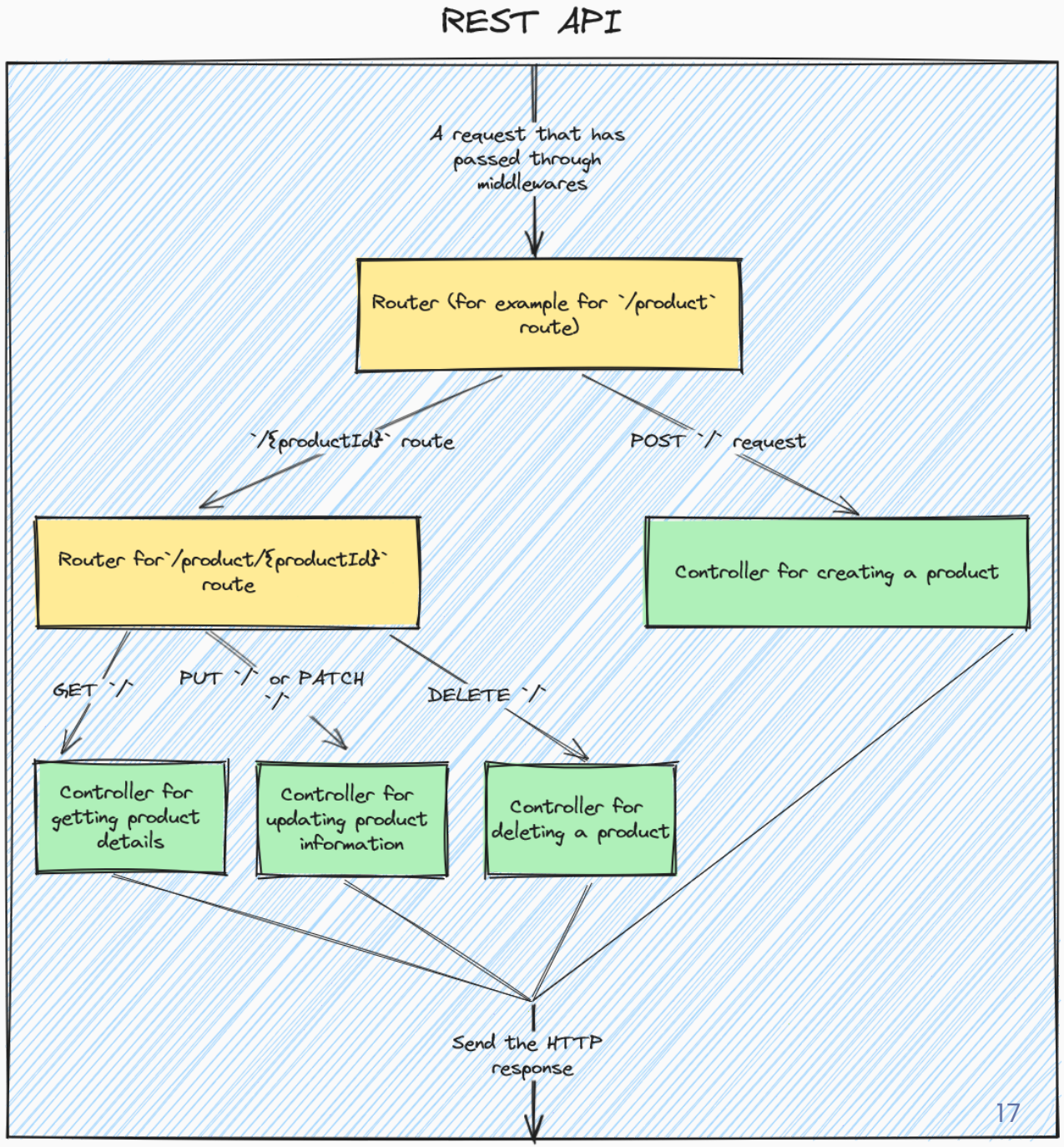


Express - routes (routers)

- The request then gets processed via a router
- Router routes the requests. It defines the flow of individual requests

```
// this import happens at the beginning of our project root file (typically `index.ts`)
import { productRouter } from './product/router';

api.use('/product', productRouter);
```

Express - routes (routers)



Express - routes (routers) and controllers

- Each route has an assigned controller - a function that processes the request individually

```
// './product/router' file where the `productRouter` is defined
import { Router } from 'express';
import { productControllers } from './controllers';

export const productRouter = Router();
const productSpecificRouter = Router();

/* post for `/product` router */
productRouter.post('/', productControllers.post);

/* get, put/patch, delete for `/product/productId` router */
productRouter.use('/:productId', productSpecificRouter);
productSpecificRouter.get('/', productControllers.get);
productSpecificRouter.put('/', productControllers.edit);
productSpecificRouter.patch('/', productControllers.edit);
productSpecificRouter.delete('/', productControllers.delete);
```

Express - controllers

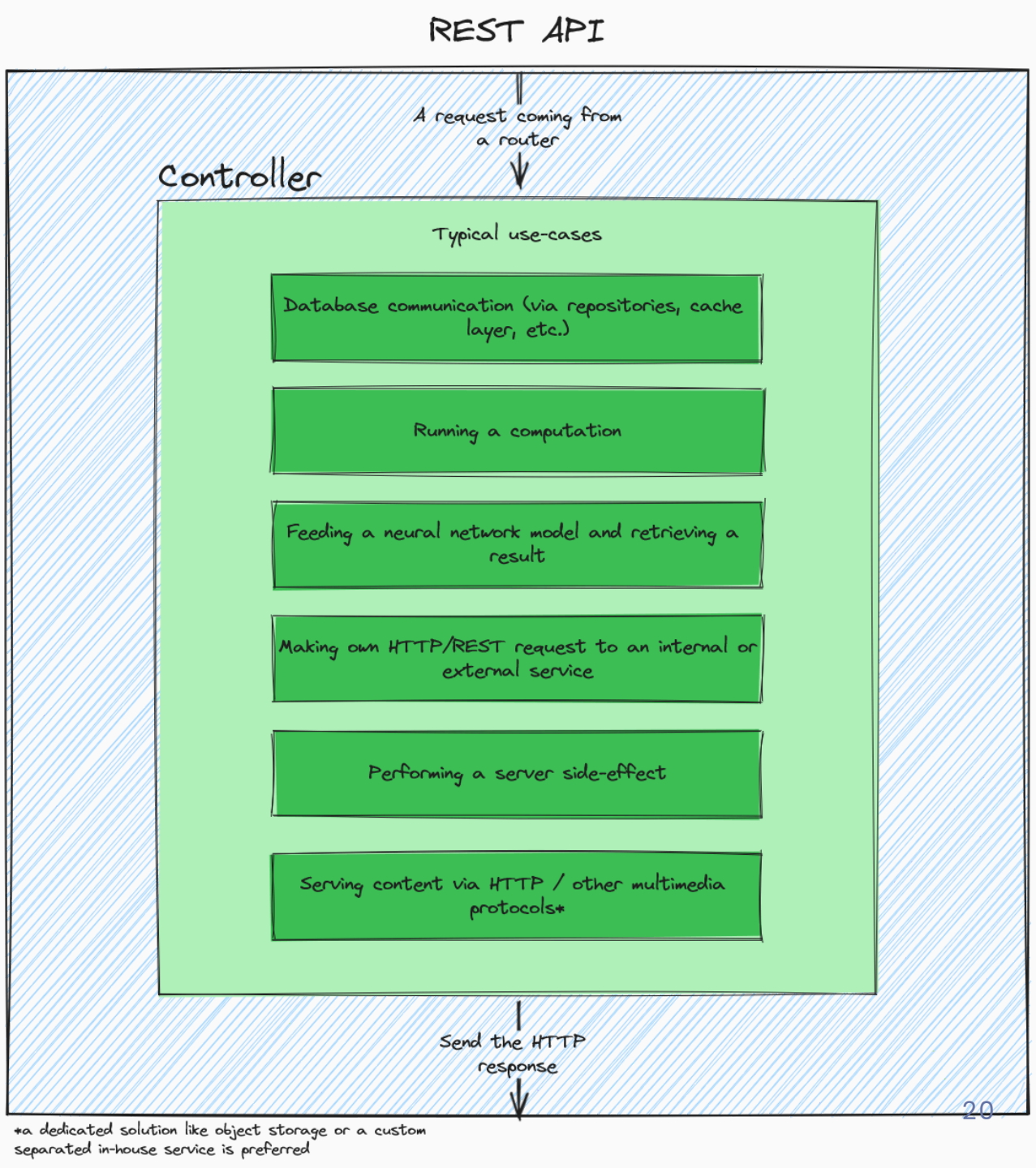
- The controller then provides the logic for handling of the specific request

```
// './product/controllers/' file where `productControllers` is defined
const get = (req: Request, res: Response) => {
  /* The logic is defined here. Calls to the database, parsing query params,
     error handling, sending responses etc. */
};

/* ... rest of the file */

export const productControllers = {
  get,
  set,
  /* ... */
};
```

Express - controllers

Note: The response of a controller should be fast. In case an expensive operation is triggered, the system should create a "session" and the client should then poll the status of the task via periodic requests to the session id endpoint

General rule: The quicker the response, the better the UX of the client is.



Runtime validation

Problem: During runtime of our application, we expect the HTTP bodies (within the REST context) to contain objects with well-defined structure. We might help ourselves by defining types to work with these objects in TypeScript. This works, if the shape of the data we receive matches the type we have defined. If it does not, our application crashes on accessing undefined properties... How can we solve this issue?

```
type ProductCreateData = {
  name: string,
  description: string,
};

// within a controller:
const data: ProductCreate = req.body;

// we want to use the object properties for something, pass it to the DB repository, etc.
console.log(data.description);
//               ^ Our app might crash here as we cannot be sure this property exists!!!
```

Solution: We import a well written runtime validation library with full TypeScript support like zod(preferred) or yup, and we create schemas for the objects we want to use!

Runtime validation - zod schema + validation

If the parsing returns success, the data we put to the validator was **checked during runtime** and passed the validation. Now, **TypeScript also knows it can rely on the exact shape of the data** the schema specified. We can now safely pass this data to the rest of the application, for example to the database repository, or a function which executes some form of calculation. Amazing, isn't it?

```
import z from 'zod';

// define object schema, possibly with additional restrictions on the types
const ProductCreateSchema = z.object({
  name: z.string().min(4),
  description: z.string().min(20),
}).strict();

export type ProductCreateData = z.infer<typeof ProductCreateSchema>;

// usage within a controller (async version was used, as this can be computationally heavy and we don't want a blocking operation):
const validationResult = await ProductCreateSchema.safeParseAsync(req.body);

if (!validationResult.success) {
  // handle the error and use early return, you now have the `.error` property available
  return res.status(400).send({ error: validationResult.error });
}

// handle the success where `validationResult.data` exists, which is now typesafe (has type `ProductCreateData`)!
const { data } = validationResult;

await productRepository.create(data);
// and continue with the business logic of your controller
```

Open API - documentation, documentation, documentation

Documentation is the key for every API consumer. What is not documented is unknown, especially once the project grows in scale. OpenAPI documentation solves this by:

- Providing a standardized way of writing documentation for APIs
- Allowing developers to easily document their API:
  - Routes & Methods
  - Expected data within request bodies/headers
  - API consumer input conditions (what goes into request bodies, query parameters, etc.)
  - Potential responses from the API
- Allowing consumers of the API to understand it without seeing the internals

Most of the time the implementation of the API you consume is private -> government APIs, corporate APIs, closed-source services. You might not have the permission to look into the source code of a product even within your company.

Adding Swagger documentation into your project

Adding the OpenAPI documentation tools is as easy as adding these few dependencies for your express project: `swagger-ui-express`, `yaml`

```
npm i swagger-ui-express yaml
```

And adding them to your Express application

```
import express, { Express } from 'express';
import swaggerUi from 'swagger-ui-express';
import fs from 'fs';
import yaml from 'yaml';

const api: Express = express();
const documentationFile = fs.readFileSync('../docs/swagger.yml');
const swaggerDocument = yaml.parse(documentationFile);

/* Add the api-documentation endpoint */
api.use('/api-documentation', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```

Get inspired by the demo!

The demo application already has the documentation complete and exposed for you. You can see how to write the documentation, and how to embed swagger within your express.js application.

Run the demo application - start Redis

Our demo application connects to a Redis service. The application uses a database repository*, which allows us to persistently store data. We will use the repository as a "blackbox" - we input data and retrieve the data from the repository. The repository hides all the implementation details of communication with the Redis service.

```
docker run -d --name redis -p 6379:6379 redis
```

* *You will learn how to create database repositories (and the whole idea behind this design pattern) later during the course. For now, we will learn how to consume the repositories to load and store data within a database solution of choice.*

Run the demo application

We're now able to start the application.

```
# install dependencies if you have not already
npm i

# run the api in watch mode - the express app will keep refreshing on changes
npm start
```

Go to http://localhost:6001/api-documentation to see the documentation.

Live coding: Implement the REST API an e-commerce site

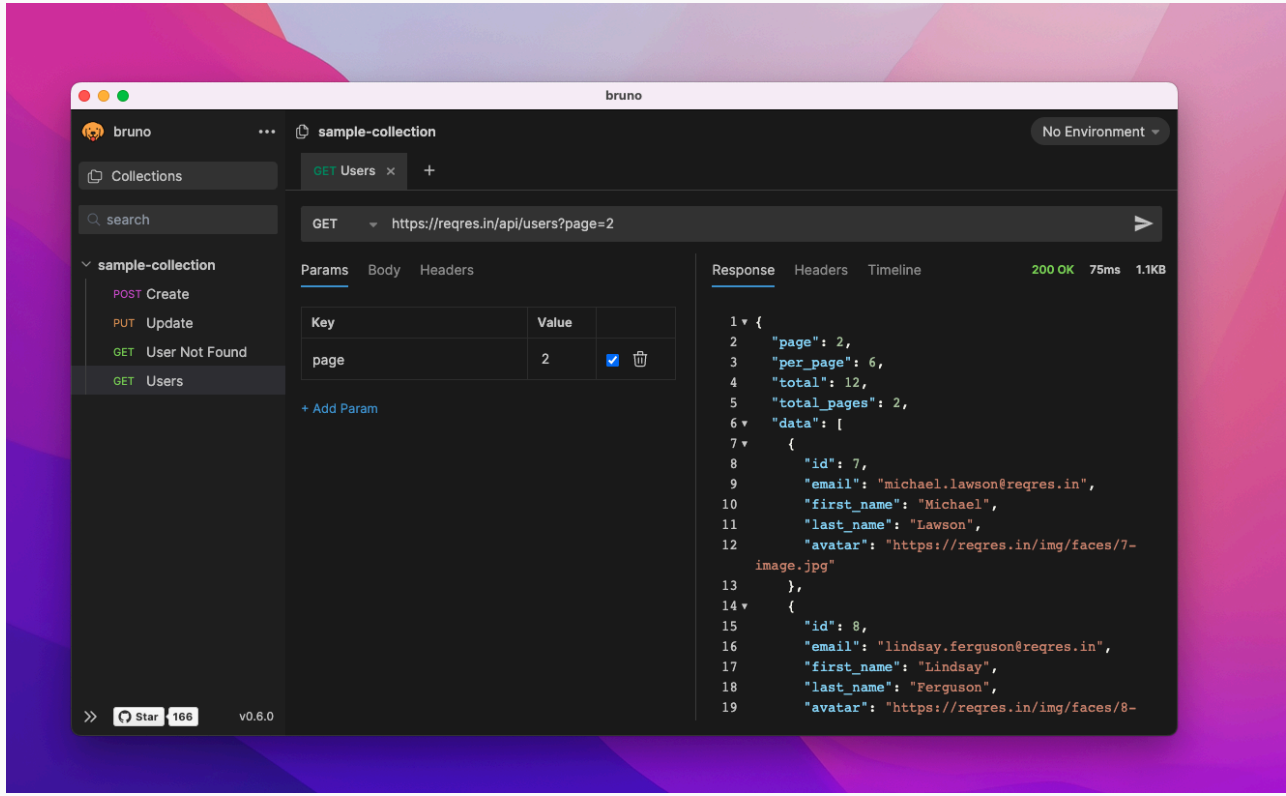
The OpenAPI docs we made provide you with all information necessary to create the e-commerce API. **Let's start!**

(The demo is available in the interactive syllabus)

Testing the API during development - Bruno

There are many tools that you can use to test your APIs, even your web browser is a very powerful one. For a more robust development experience, you might want to look into Bruno, which is an open-source tool for testing REST APIs. The Free version covers everything you'll need in the context of this course (and more). It even has a VSCode extension!

> You might have heard about Insomnia or Postman. These solutions transitioned into paid services (SaaS) in recent years and the user base has started to shift away from them. A version of Insomnia that some people started to use instead is Insomnium - a privacy-oriented fork of Insomnia.



Questions?