UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURES 25-26

The REST Paradigm

Prof. Luigi Libero Lucio Starace

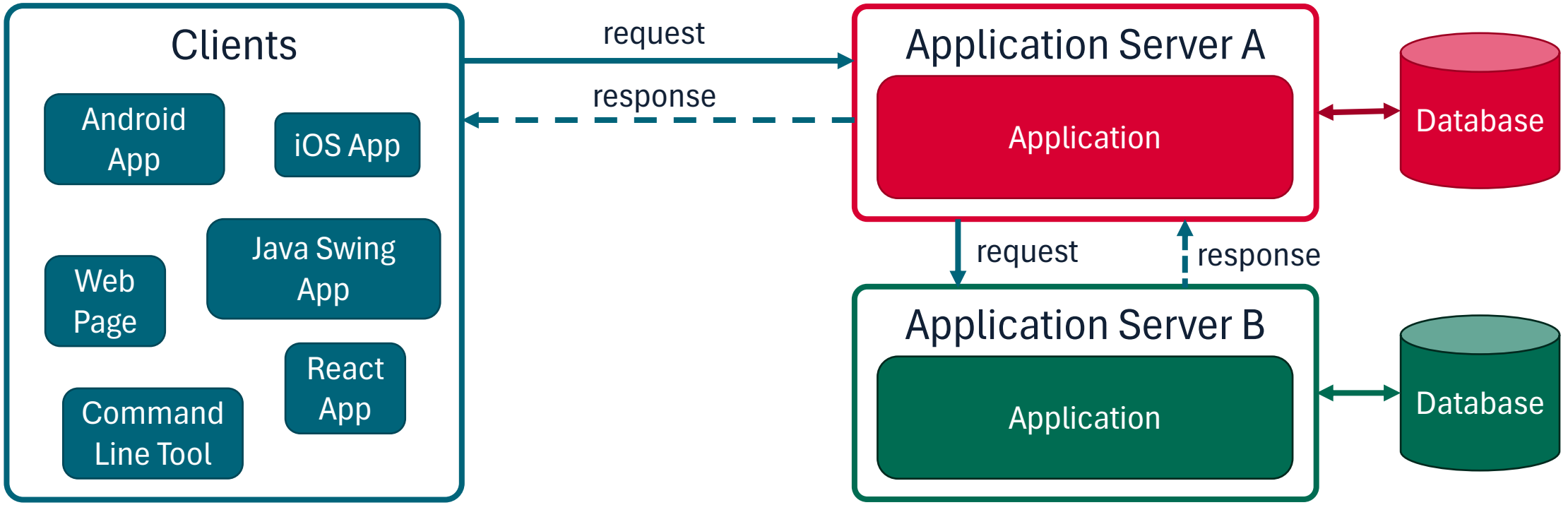
luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

Architectural Context

- **Software** needs to communicate over the **Internet**
- **83% of web traffic is actually API calls**[1]

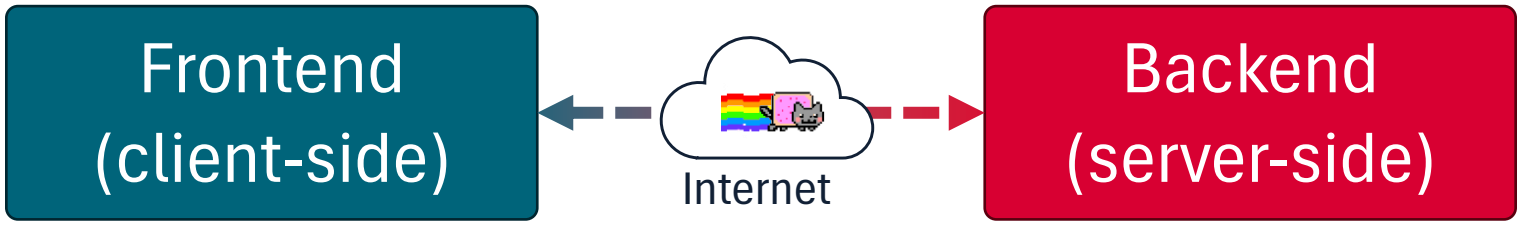


[1] Akamai's State of the Internet Security Report (2019)
https://www.akamai.com/newsroom/press-release/state-of-the-internet-security-retail-attacks-and-api-traffic

Client-Server Architectures

Often, applications are **split** in two components:

- **Backend**: Responsible for data and business logic (on the server-side)
- **Frontend**: Responsible for the UI (on the client-side)
  - Might be a **native mobile** or **desktop app**
  - Or it might still be a «*smarter*» HTML page, that renders an interactive UI leveraging JavaScript (i.e., Single Page Web Apps)
- These two components communicate over the internet



Application Programming Interfaces

Application Programming Interfaces (**API**s) are ways for computer programs to communicate with each other

How can we handle communications over the internet?

- Manually define a protocol over TCP/UDP, open sockets, etc...
  - Not very cost-effective, not very **interoperable**
  - If we want to integrate *n* APIs, we'll need to learn *n* different protocols
- CORBA, Java RMI, SOAP, ...
  - Dedicated protocols/architectures exist(ed), re-inventing an alternative to the web
- Just use **web protocols (HTTP)**!

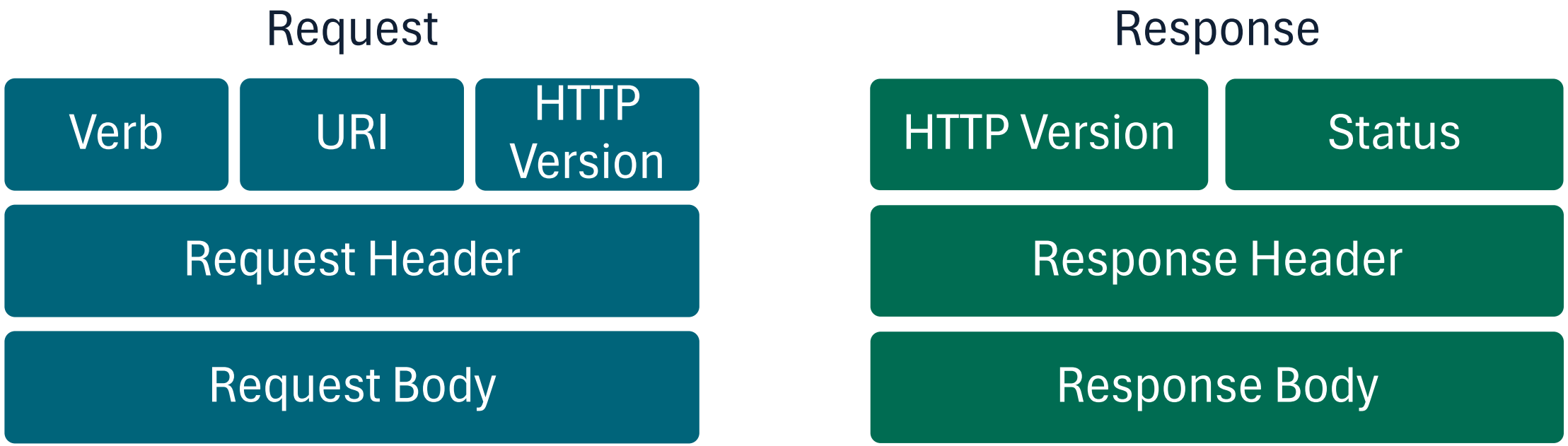
REST

- REST is **not** a standard or a protocol
- REST is an **architectural style**: a set of principles and guidelines that define how web standards should be used in APIs
- Based on HTTP and URIs
- Provides a common and consistent interface based on «proper» use of HTTP
- The prevalent web API architecture style with a **93.4% adoption rate**[1]

[1] https://nordicapis.com/20-impressive-api-economy-statistics/

(A look back on) HTTP

- HyperText Transfer Protocol
- Two types of messages: Request and Response



HTTP Requests

- HTTP can request different kinds of operations (using **Verbs**)
- The resource on which to operate is identified by the URI
- Based on idea of CRUD (Create, Retrieve, Update, Delete)

Verb	Operation Description
POST	Here is some new data. Save it and **CREATE** a new resource
GET	**RETRIEVE** information about the resource
PUT	Here is **UPDATE**d info about the resource
DELETE	**DELETE** this resource

HTTP Responses

- Status codes give information about the outcome of the request

Status Code	Meaning
200	Request was successfully handled
201	A resource was successfully created
400	Cannot understand the request
401	Authentication failed, or not authorized.
404	Resource not found
405	Method not supported by resource
500	Application error

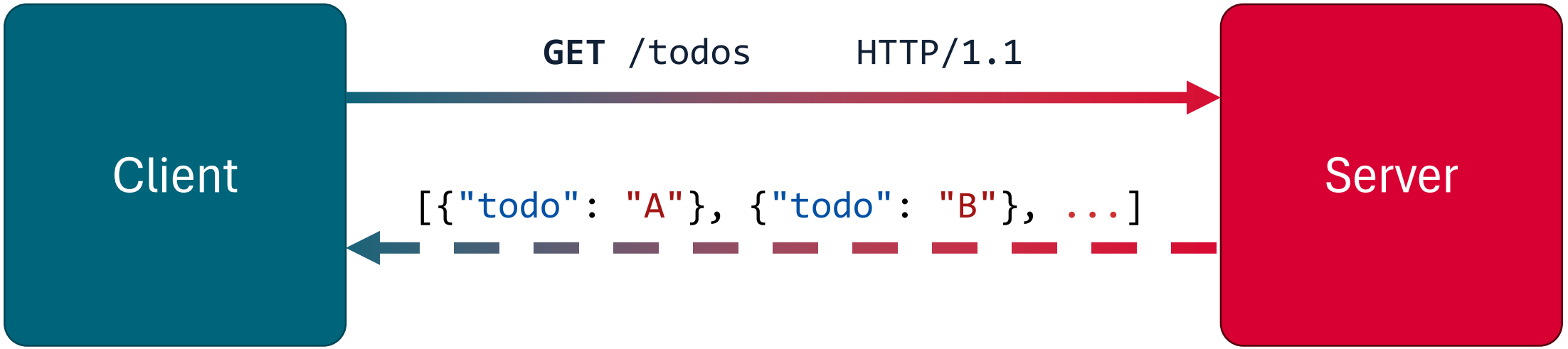
REST Fundamentals

- A REST API allows to interact with **resources** using HTTP
- All resources are associated to a unique URI
- «A resource is anything that's important enough to be referenced as a thing in itself.»[1]
- Resource typically (but not necessarily) correspond to persistent domain objects
- HTTP verbs should be used to retrieve or manipulate resources
- REST API should be **stateless** (e.g.: not use server-side sessions)
  - This ensures better scalabilty! Can you tell why?

[1] Richardson, Leonard, and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", 2008.

Representational State Transfer (REST)

- **Application State** (on the Client)
- **Resource State** (on the Server)
- Trasferred using appropriate representations (e.g.: JSON, XML, ...)



HTTP: Data Interexchange Formats

- Widely used formats include JSON and XML

```
{
  "todos": [{
    "todo": "Learn REST",
    "done": false
  }, {
    "todo": "Learn JavaScript",
    "done": true
  }]
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <todos>
    <todo>
      <text>Learn REST</text>
      <done>false</done>
    </todo>
    <todo>
      <text>Learn JavaScript</text>
      <done>true</done>
    </todo>
  </todos>
</root>
```

REST: Examples

HTTP verb/URI	Meaning
GET /todos	Retrieve a list of all saved To-do items
GET /todos/<ID>	Retrieve only the To-do item whose ID is <ID>
POST /todos	Save a new To-do item. Data of the exam to save are in the request body
PUT /todos/<ID>	Replace (or create) the To-do item whose ID is <ID>, using the data in the request body
DELETE /todos/<ID>	Delete the To-do item having ID <ID>

REST: Nested Resources

- Sometimes, there is a hierarchy among resources
  - A Company has 0..* Departments which have 0..* Employees...
- When designing an API, it is possible to define nested resources
- **`GET /listings/{id}/reviews`** lists all the reviews of a given listing.
- Keep in mind that every resource should have only one URI

REST: Filtering and Ordering

- Often, clients need to retrieve a list of resource with specific filtering criteria or specific sorting in place (e.g.: sort listings by price)
- You may be tempted to add new paths as follows

HTTP verb/URI	Meaning
GET /todos	Retrieve a list of all saved To-do items
GET /todosSortedAsc	Retrieve saved To-do items, sorted alphabetically (ascending order)
GET /todos/filter/key	Retrieve all To-do Items containing the «key» keyword

REST: Filtering and Ordering

HTTP verb/URI	Meaning
GET /todos	Retrieve a list of all saved To-do items
GET /todosSortedAsc	Retrieve saved To-do items, sorted alphabetically (ascending order)
GET /todos/filter/key	Retrieve all To-do Items containing the «key» keyword

- This is not REST compliant!
  - Resources should be accessed via a single path!
- Also, this can lead to complex and hard-to-use APIs
  - Think of all possible sorting and filtering options you may need

REST: Filtering and Ordering

- So, how do we handle these cases?
- You should not add new paths, but you should use **query parameters**

HTTP verb/URI	Meaning
GET /todos	Retrieve a list of all saved To-do items
GET /todos?sort=title&mode=asc	Retrieve saved To-do items, sorted alphabetically by title (ascending order)
GET /todos?q=key	Retrieve all To-do Items containing the «key» keyword

Securing a REST API

The JSON Web Token (JWT) Authentication/Authorization Scheme

API Authentication/Authorization

- REST APIs allow users to manipulate resources
- In most cases, we don’t want everyone to be able to do so
- **Authentication**: We want that only legit users can access the resources
- **Authorization**: We may also want that some users can access only certain resources (e.g.: an employee shouldn’t be able to update its own salary)

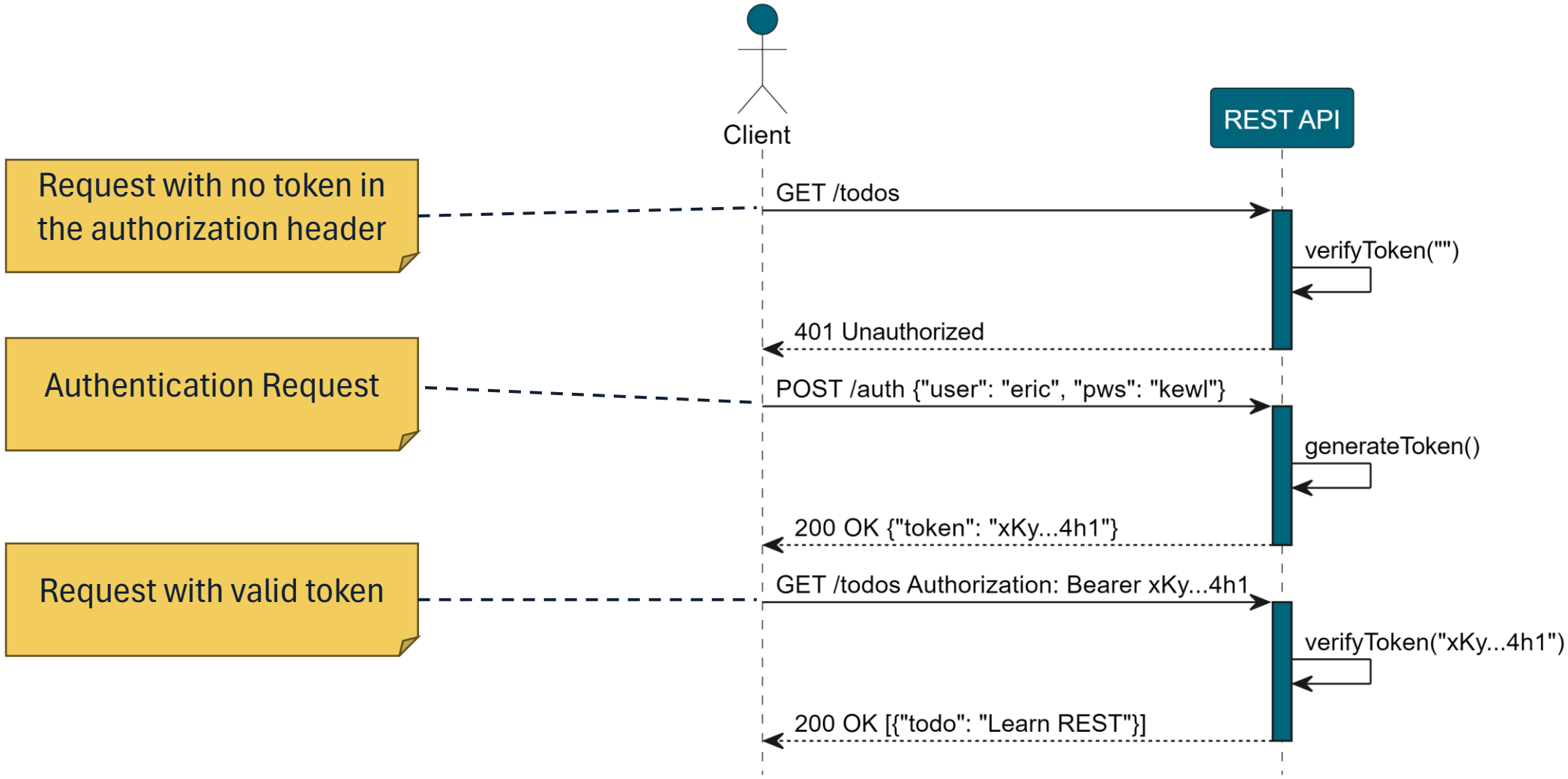
How to secure REST APIs

A widely-used authentication scheme is based on Tokens

💡 Idea:

1. Clients send a request with username and password to the API
2. API validates username and password, and generates a token
3. The token (a string) is returned to the client
4. Client must pass the token back to the API at every subsequent request that requires authentication (in the Authorization Header)
5. API verifies the token before responding

Token-based Authentication Scheme



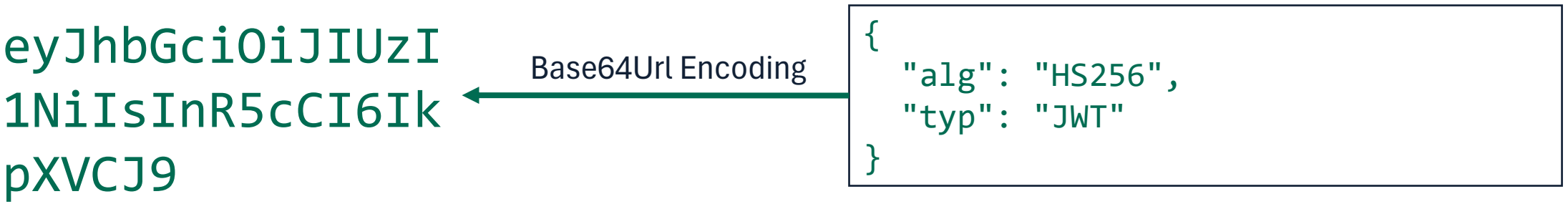
JSON Web Token (JWT)

- JWT is a widely-adopted open standard (RFC 7519)
- Allows to securely share claims between two parties
- A JWT token is a string consisting of three parts, separated by “.”
- Structure: Header.Payload.Signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX
VCJ9.eyJuYW1lIjoibHVpZ2kiLCJyb2x
lIjoiYWRtaW4iLCJleHAiOjE2NzAzOTg
0MzJ9.fopBYrax8wcB7rnPjCcOMc62IT
2lJdvyOdyixMWMZAQ
```

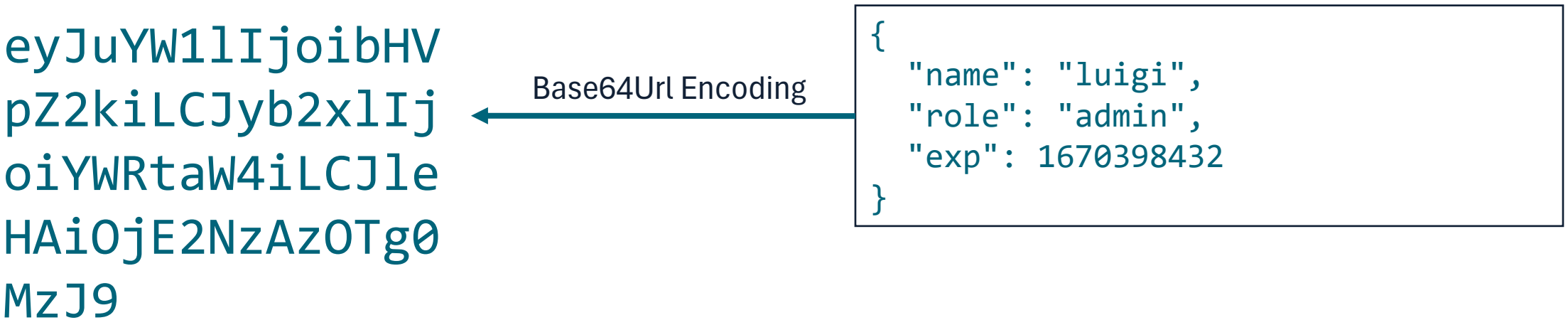
JWT: Header

- The JWT header contains information about the **type of token** and the type of hashing function used to **sign it**, in JSON format
- It is **encoded** using Base64Url Encoding
  - Note that this is merely an encoding, not an encryption! **It can be easily be inverted!**



JWT: Payload

- The payload is a JSON object containing **claims**
- Some registered claims are recommended (e.g. «exp» indicates an expiration date for the token)
- Claims are customizable (we can write any claim in the payload)
- It is **encoded** using Base64Url Encoding



JWT: Signature

- To ensure that tokens are not tampered or forged, a signature is included
- The signature is obtained by applying a **cryptographic hashing function** to the string obtained by concatenating:
  - The **header** of the token
  - The **payload** of the token
  - A **secret key** known only by the server that issues the token
- Actually, JWT signatures are a little bit more complex than that
  - Check out HS256 or RS256 if you want to know more: https://auth0.com/blog/rs256-vs-hs256-whats-the-difference/

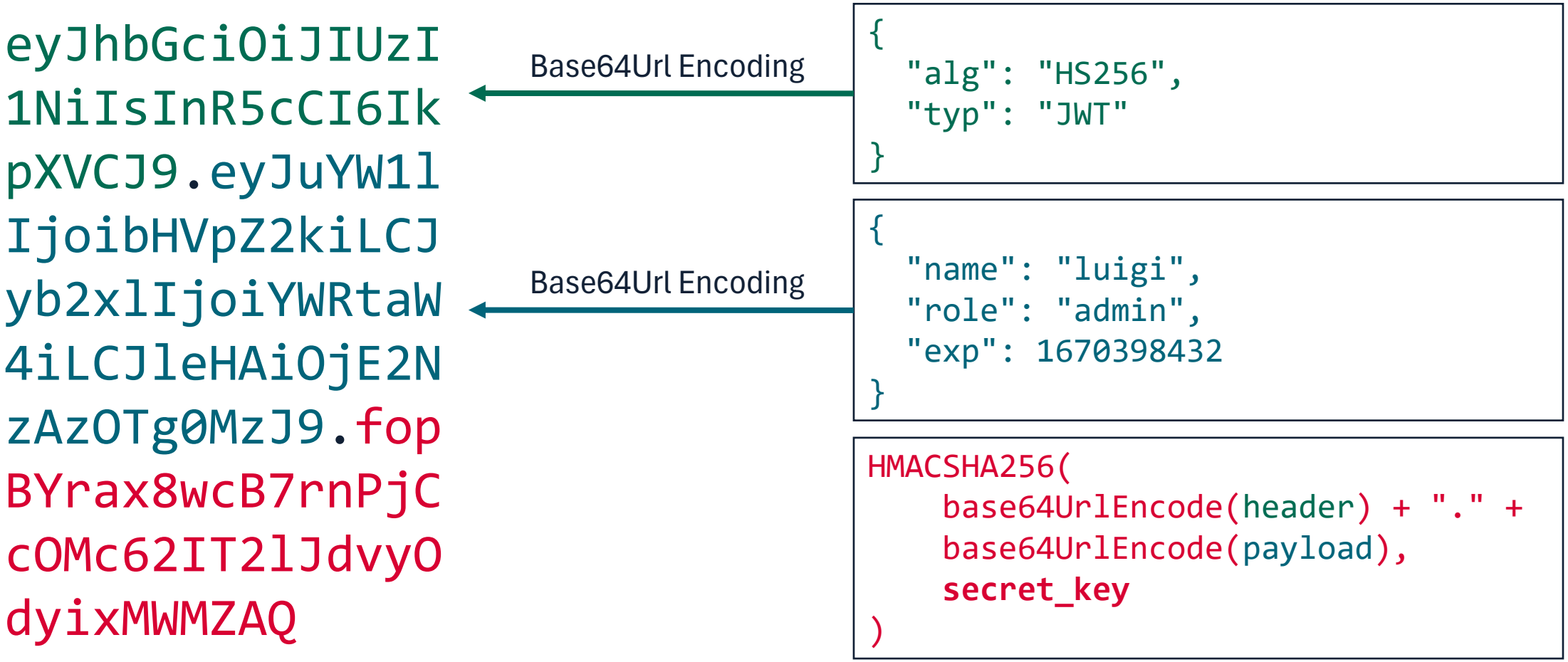
Cryptographic Hashing Functions

Functions that map an arbitrary binary string (**input**) to a fixed-size binary string (**digest** or **hash**)

Secure cryptographic hashing funcs have some interesting properties:

- They are virtually **collision free** (basically it's impossible to find two different inputs that lead to the same digest)
- They can be computed easily, but **cannot be inverted**
  - Given an input, it is easy to compute its digest. But given a digest, it's unfeasible to compute the input that generated it

JSON Web Token: Overview



Implementing a REST API

Practical demonstration with JAX-RS

Implementing a REST API

- REST APIs are conceptually simple
- We just need to listen for HTTP requests, and handle them
  - Typically, in prevalently synchronous languages such as Java, a new thread is created to handle the request
  - Once done handling the request, we send a HTTP response accordingly
- You should already know how to implement a REST API using low-level languages and sockets from the Computer Networks/Operating Systems Lab courses.

Implementing a REST API from scratch

Implementing a REST API from scratch is feasible and within our reach

There’s quite a lot of efforts involved though…

- We need to parse the HTTP requests
- Possibly, we need to deserialize objects received as JSON/XML in requests
- We need to prepare HTTP responses
- We need to serialize objects to encode them as JSON/XML in responses
- We need to manage request routing
- We need to manage request filtering and authorization

Luckily, we’re software engineers! No need to re-invent the wheel!

(Web) Frameworks

Web Frameworks can help us implement REST APIs without dealing with these repetitive, «common» core tasks

What's a Framework?

- A collection of reusable components, tools, libraries, and practices designed to simplify software development and maintenance
- Frameworks do not directly implement any user feature, but can be used to simplify the development of the actual features
- Their purpose is to provide a foundation for software development by offering ready-made solutions for common problems, enforcing best practices, and ensuring consistency across applications.

Software Library

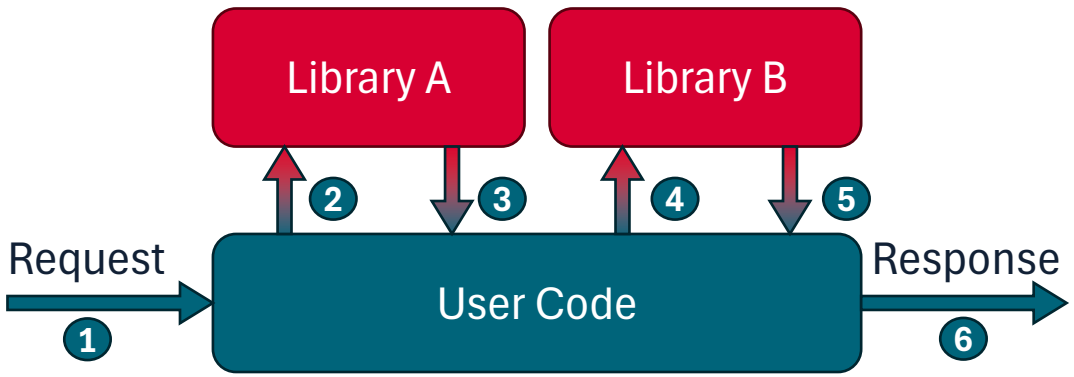
- Essentially a set of functions we can call to get some job done
- Each call does some work and then returns control to the caller
- User code (e.g.: our **main** method) is in control of the execution flow

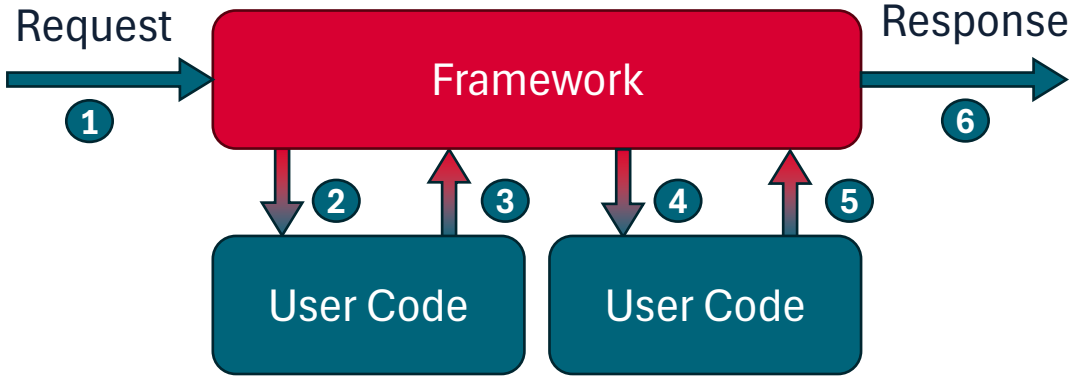
Framework

- Embody some abstract design, with more **behaviour** built-in
- User-written code can be **plugged-in** into the framework
- The framework is in control of the execution flow and calls user code when appropriate

Inversion of Control (IoC) Principle

- **IoC** is a defining feature of frameworks
- The control flow is **inversed** from traditional imperative programming
  - The responsibility of managing the control flow is shifted from the developer to a framework
- A.k.a. Hollywood's Law: "*Don't call us, we'll call you*".





Opinionated Frameworks

- Frameworks can be more or less **opinionated**
- A (strongly) **opinionated** framework comes with a rigid set of pre-defined conventions, best practices, and decisions made by the framework's creators. It **enforces** a specific way of doing things, and leaves less wiggle room to devs.
- Unopinionated frameworks do not enforce a specific way of doing things

Keep in mind, this framework has really strong opinions...

Artwork generated using DALL-E 3

Opinionated Frameworks

Pros

- Ensure higher levels of consistency across different projects
- Can speed up development, reducing decision fatigue

Cons

- May have a steeper learning curve
- May not be flexible enough for a certain project

Web Frameworks (Server-side)

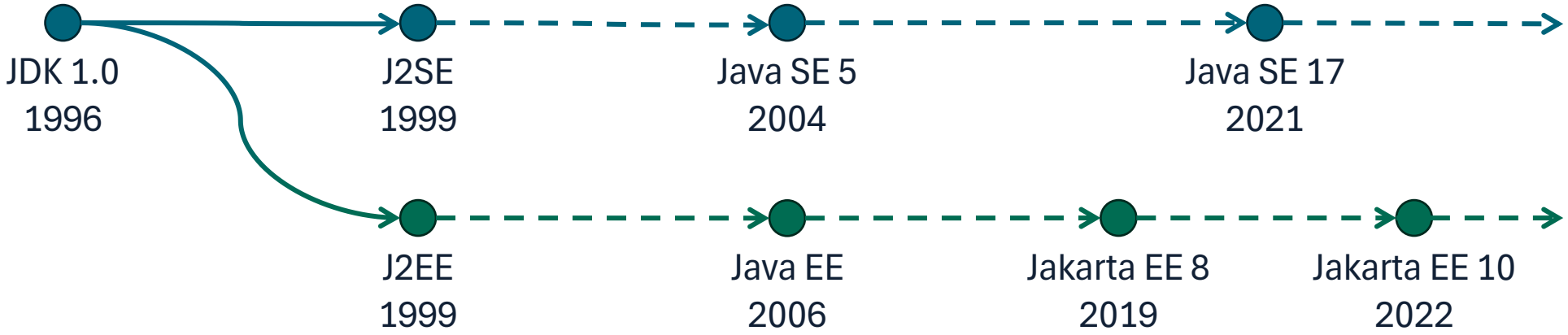
- Many web frameworks to choose from (e.g.: see Wikipedia)

One of the above is not a real framework, but a Pokemon. Can you tell which one?

Practical Demonstration

Implementing a REST API with Jakarta EE (JAX-RS)

JAKARTA ENTERPRISE EDITION (EE)



- Jakarta EE is evolution of Java EE
- It's a set of specification, including standards such as Servlets, Jakarta Server Pages, **JAX-RS**...
- **JAX-RS** is a specification for implementing **REST API**

JAX-RS AND ITS IMPLEMENTATIONS

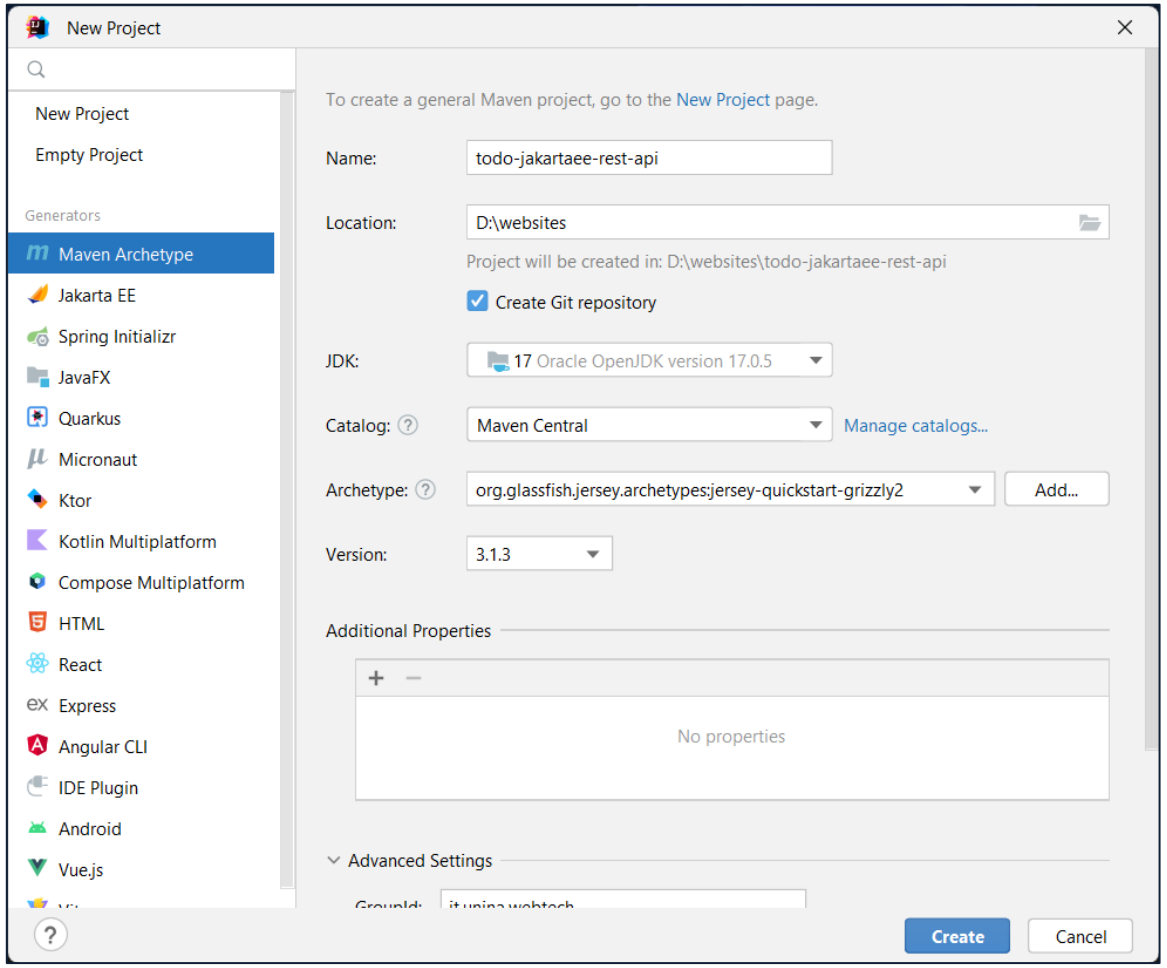
- JAX-RS is just a set of specifications
- The most well-known implementations are **frameworks** such as
  - Eclipse Jersey (we're gonna use this in the demo!)
  - JBoss RESTEasy
- Annotation-based: specific Java annotations should be used to «tell» the frameworks when to run the code we write

A REST API USING JAX-RS

- Let’s take a look at the code

MAVEN ARCHETYPE

- The easiest way to get started is to use a **Maven Archetype**
- Archetypes are basically templates, including dependecies and boilerplate code
- We'll use the **jersey-quickstart-grizzly2** archetype from Maven Central



ADD SUPPORT FOR JSON

- After creating your project from the archetype, remember to include JSON support by uncommenting the following dependency in the **pom.xml**

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-binding</artifactId>
</dependency>
```

CODE EXAMPLES

```
@Path("greet") //this class handles requests on the /greet path
public class GreeterController {
    @GET //when a request is made using the GET verb, invoke this method
    @Produces(MediaType.TEXT_PLAIN) //map return value to plaintext
    public String greet(){
        return "Hello User!";
    }
}
```

CODE EXAMPLES

```
@Path("/todos")
public class TodoController {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<TodoItem> getAll() {
        return TodoItemDAO.getAll();
    }
}
```

- The returned `List<TodoItem>` is automatically converted to a JSON representation (thanks to the `@Produces` annotation)!

CODE EXAMPLES

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response addTodoItem(TodoItem todo){
    try {
        TodoItemDAO.add(todo);
        return Response.status(Response.Status.CREATED).build();
    } catch (Exception e) {
        return Response.status(Response.Status.BAD_REQUEST)
                       .entity(e.getMessage()).build();
    }
}
```

- The JSON object received in the body of the `@POST` request is automatically serialized in the `TodoItem` argument (thanks to the `@Consumes` annotation)

OpenAPI

OPENAPI: DESCRIBING A REST API

- OpenAPI (formerly Swagger) is an open, formal standard to describe HTTP APIs
- Why?
  - **Standardization**: ensures consistent documentation practices
  - **Documentation**: can be used to automatically generate comprehensive docs
  - **Code generation**: Allows for automatic generation of client libraries and server stubs
  - **Machine and human-readable:** enables automated discovery and more
  - **Most broadly adopted industry standard**

THE OPENAPI SPECIFICATION

- **OpenAPI Descriptions** are written as structured text documents
- Each document represents a JSON object, in JSON or YAML format
- These specification files describe version of the OpenAPI specification (`openapi`), the API (`info`) and the endpoints (`paths`)

```yaml
# YAML format
openapi: 3.1.0
info:
  title: A minimal specification
  version: 0.0.1
paths: {} # No endpoints defined
```

```json
// JSON format
{
  "openapi": "3.1.0",
  "info": {
    "title": "A minimal specification",
    "version": "0.0.1"
  },
  "paths": {} // No endpoints defined
}
```

THE OPENAPI SPECIFICATION: PATHS

- The API Endpoints are called **Paths** in the OpenAPI specification
- **Path** objects allow to specify a sequence of endpoints
  - For each endpoint, its supported methods
  - For each method,
    - Expected input parameters (if any) (e.g.: path parameters, headers, etc...)
    - Request body (if any)
    - Possible responses
    - And more!

THE OPENAPI SPECIFICATION: PATHS

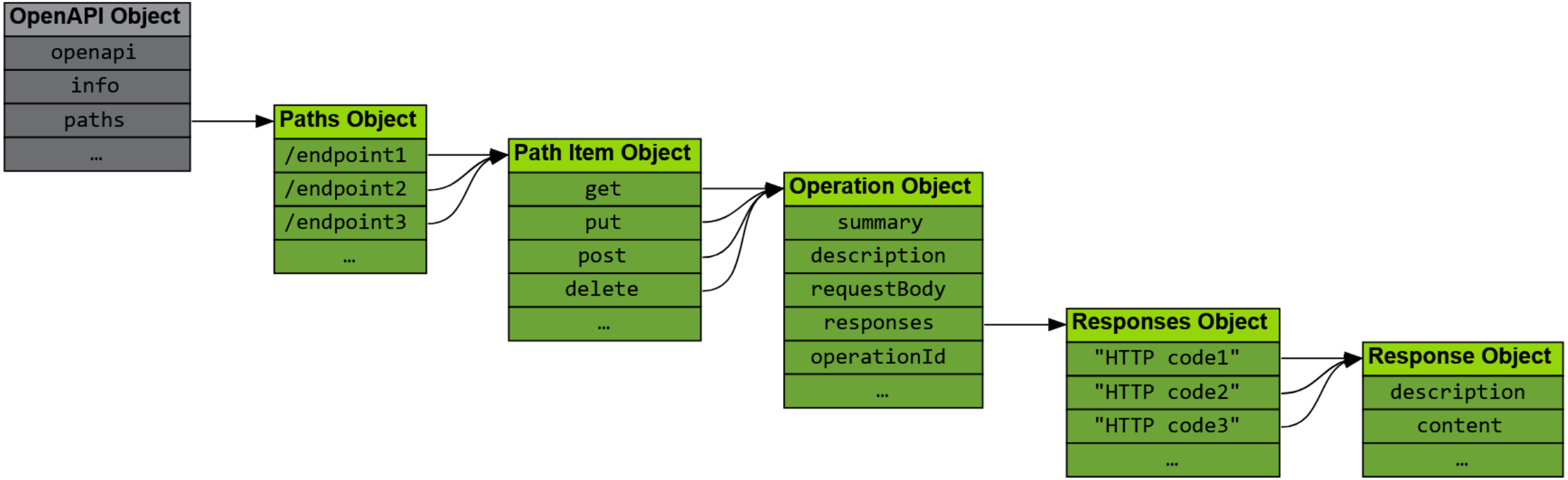


Image from https://learn.openapis.org/specification/paths

PATH EXAMPLE

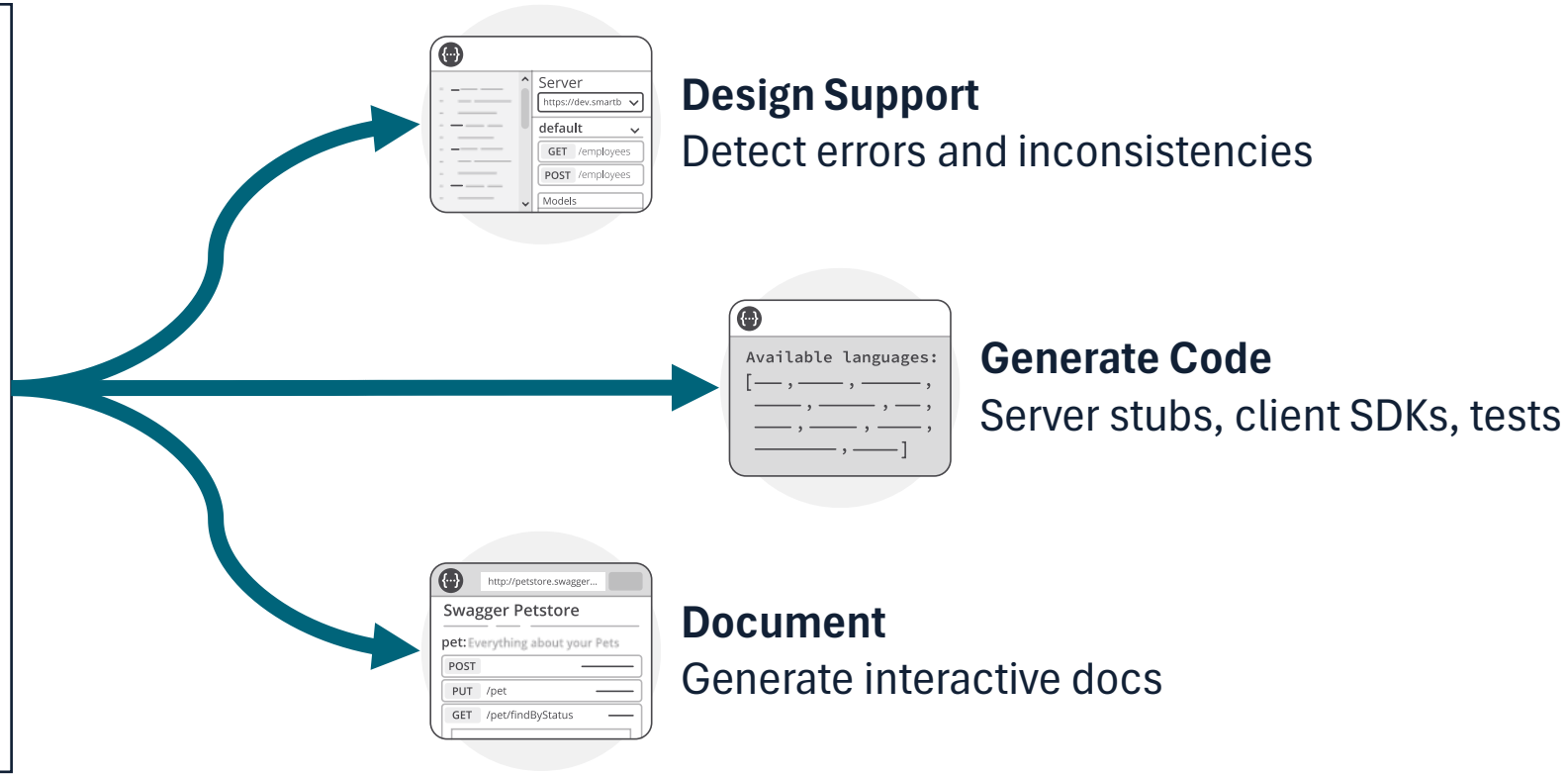
- /auth endpoint, POST method
- Expects Request Body containing a JSON object with type {usr: string, pwd: string}
- Returns 200 OK on success, 401 Unauthorized when credentials are invalid

```yaml
/auth:
  post:
    description: Authenticate user
    produces:
      - application/json
    requestBody:
      description: user to authenticate
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              usr:
                type: string
                example: Kyle
              pwd:
                type: string
                example: p4ssw0rd
    responses:
      200:
        description: User authenticated
      401:
        description: Invalid credentials
```

WORKING WITH OPENAPI SPECIFICATIONS

- Writing an OpenAPI specification for an API is quite a lot of work
- What can we do with an OpenAPI specification?

```yaml
openapi: 3.1.0
info:
  title: My API
  version: 1.0.0
paths:
  /auth:
    post:
      # ...
  /todos:
    get:
      # ...
    post:
      # ...
```



Design Support
Detect errors and inconsistencies

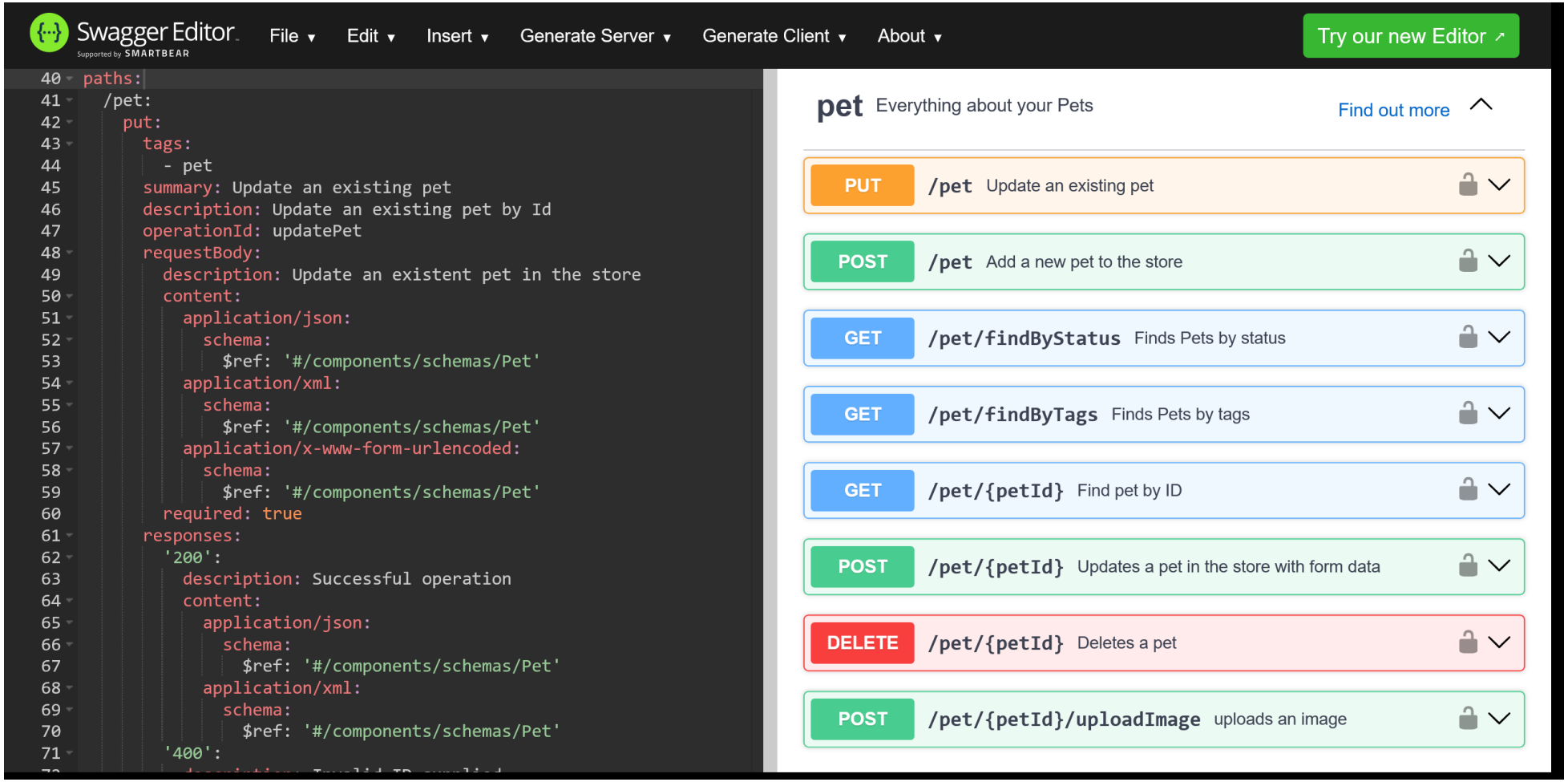
Generate Code
Server stubs, client SDKs, tests

Document
Generate interactive docs

OPENAPI: OPEN SOURCE TOOLS

- OpenAPI is supported by a number of mature, open-source tools
- Some widely used open-source tools are reported as follows:
  - **Swagger Editor**: https://swagger.io/tools/swagger-editor/
  - **Swagger Codegen**: https://swagger.io/tools/swagger-codegen/
  - **Swagger UI**: https://swagger.io/tools/swagger-ui/
- You can also check them out in a web browser, without downloading anything: https://editor.swagger.io/

OPENAPI: SWAGGER EDITOR



Specification

Documentation (Swagger UI)

OPENAPI: CODEGEN

- From the editor toolbar, we can also generate Server and Client code
- Server-side code generation can obviously only provide stubs
  - Supported languages/frameworks include: ASP.NET, Go, Java, JaxRS, Micronaut, Kotlin, Node.js, Python (Flask), Scala, Spring.
- Client-side code generation is a very handy way to make SDKs for our APIs available in tens of different languages
  - Supported languages include: C#, Dart, Go, Java, JavaScript, Kotlin, PHP, Python, R, Ruby, Scala, Swift, TypeScript

OPENAPI-DRIVEN DEVELOPMENT

- Often, developers start working on an API by defining its specification before writing any actual code
- This approach is also referred to as **OpenAPI-driven development**
- This way, they can exploit code generation capabilities to the fullest
- The approach also promotes **independence** between the different teams involved in a project (front-end, back-end, QA). The API definition keeps all these stakeholders aligned

REFERENCES (1/2)

- **The Little Book on REST Services**
  By Kenneth Lange
  Freely available at https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf
- **API design guide**
  Google Cloud
  https://cloud.google.com/apis/design
  Relevant parts: Resource-oriented design, Resource names.
- **REST API Checklist**
  By Kenneth Lange
  https://kennethlange.com/rest-api-checklist/

REFERENCES (2/2)

- **OpenAPI**
  Official Website
  https://www.openapis.org/
  https://learn.openapis.org/ (Recommended reads from this link: Getting started, Introduction)
  https://spec.openapis.org/oas/v3.1.0 (Full specification for OpenAPI 3.1.0)
  ⚠️ For the Web Technologies course, you need to know what the OpenAPI specification standard is, why it has been introduced, and what it can be used for. You are **not** required to learn how to write OpenAPI specification files from scratch.
- **API Documentation: The Secret to a Great API Developer Experience**
  Ebook by Smartbear Software
  Available at https://swagger.io/resources/ebooks/api-documentation-the-secret-to-a-great-api-developer-experience and archived here.
  ⚠️ Interesting read. **Not** required for the Web Technologies course, but it can be useful to better understand the challenges in designing and documenting an API.

EXAMPLES WITH DIFFERENT FRAMEWORKS

- **To-do List REST API using JakartaEE (Java)**
  By Luigi Libero Lucio Starace
  Includes source code, Dockerfile and instructions.
  https://github.com/luistar/jakartaee-rest-example
- **Pizza Shop REST API using Spring (Java)**
  By Luigi Libero Lucio Starace
  https://github.com/luistar/rest-service-pizzashop
- **Quiz REST API using FastAPI (Python)**
  By Márcio Lemos
  https://github.com/marciovrl/fastapi