UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – WEB TECHNOLOGIES

THE REST PARADIGM

Luigi Libero Lucio Starace, PhD

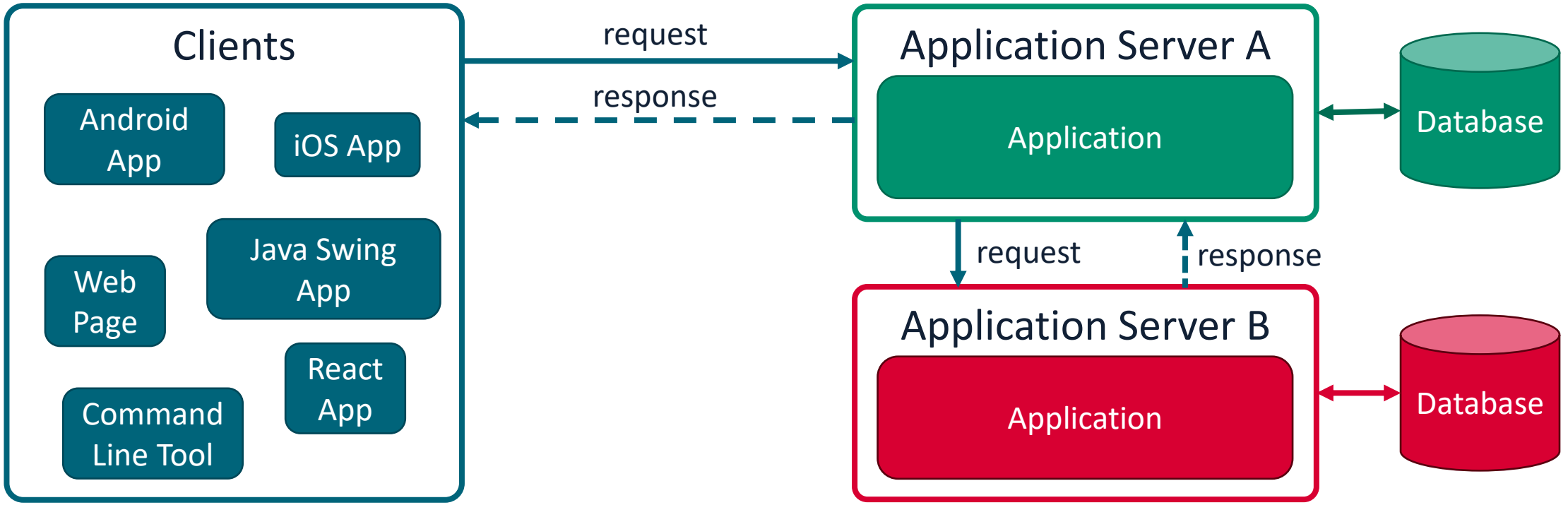
luigiliberolucio.starace@unina.it

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

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)**!

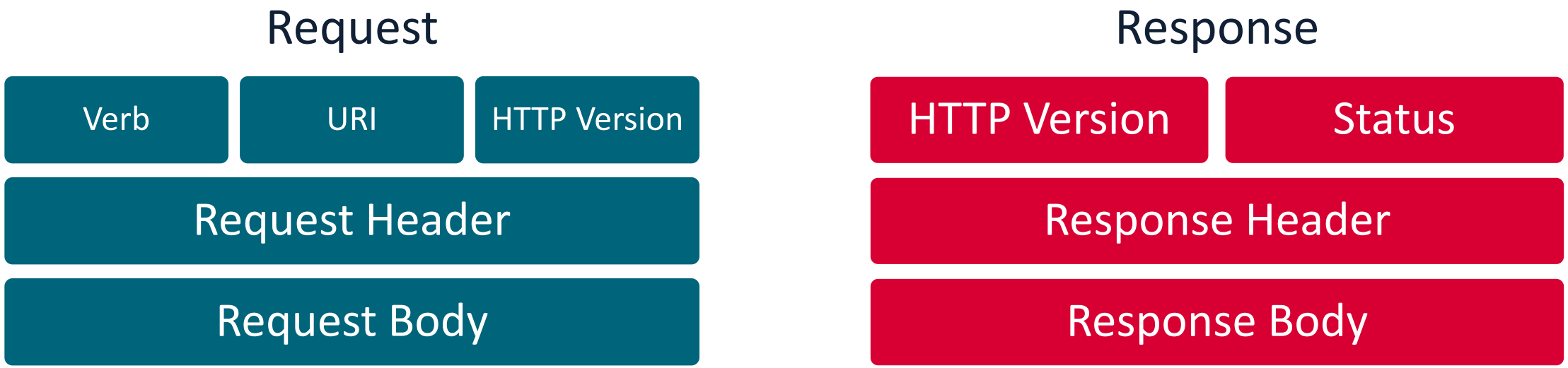
REST

- REST is 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

- HTTP is the 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 (path)
- Based on idea of **CRUD** (**C**reate, **R**etrieve, **U**pdate, **D**elete)

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

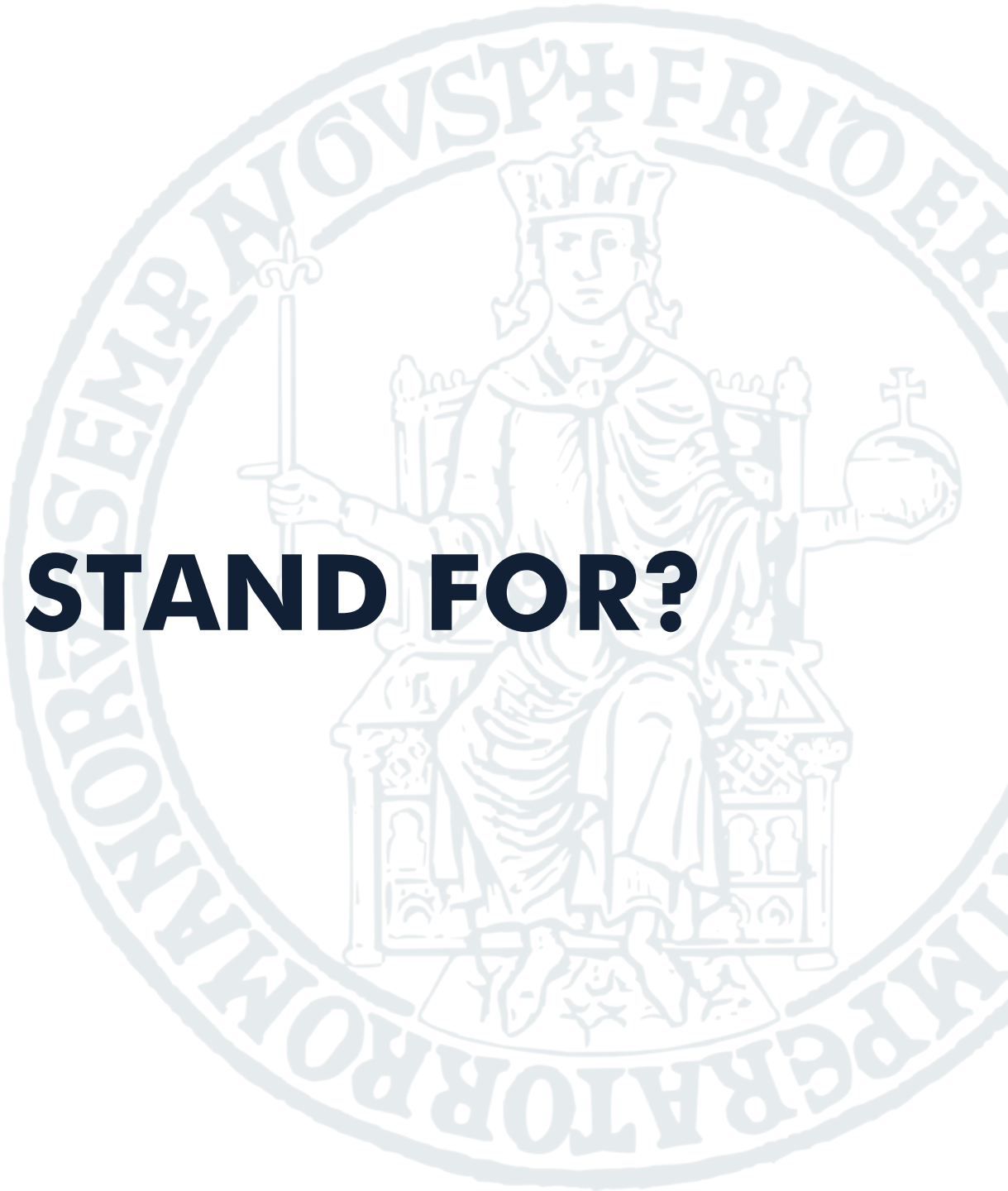
Status Code	Meaning
200	Request was successfully handled
201	A resource was successfully created
400	Cannot understand the request
401	Authentication failed, or not authorized
404	Resource not found
405	Method not supported by resource
500	Application error

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

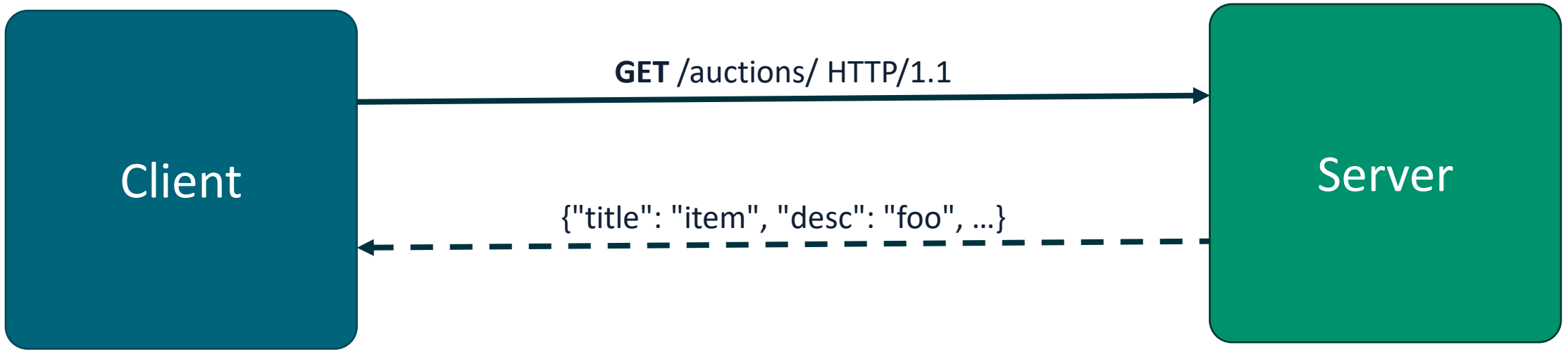
[1] Richardson, Leonard, and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", 2008.

WHAT DOES REST STAND FOR?



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

```
{
   "exams": [{
         "name": "ASD",
         "grade": 30
      },{
         "name": "INGSW",
         "grade": 30
      }
   ]
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <exams>
    <exam>
      <name>ASD</name>
      <grade>30</grade>
    </exam>
    <exam>
      <name>INGSW</name>
      <grade>30</grade>
    </exam>
  </exams>
</root>
```

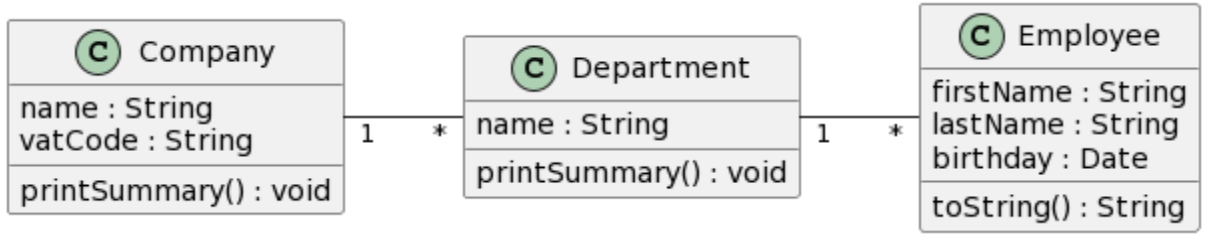
REST: EXAMPLES

We want to write an app that manages a list of Exams a student took

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

REST: NESTED RESOURCES

- Sometimes, there is a hierarchy among resources



- When designing an API, it is possible to define nested resources
- `GET /seller/{id}/reviews` lists all the reviews of a given seller.
- Keep in mind that every resource should have only one URI

DESIGNING REST ENDPOINTS: GUIDELINES

- Each resource should have a unique URI
- Use **nouns** to name URIs
- You should use plural names, except for concepts that are necessarily singular (e.g.: **/version** that returns information about the current version of the software)
- Use only lowercase characters in resource names (URIs are case sensitive!)
- Separate composite words with hyphens
  - e.g.: /users/<id>/best-friend
- Avoid special characters in resource URIs (e.g.: «{}\$@€…»)

IMPLEMENTING A REST API

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
- We need to manage request **routing**
- We need to manage request **filtering** and **authorization**
- 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

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 **framework** is a pre-defined set of software components, tools, and best practices that serve as a **foundation** for developing software
- **Web frameworks** are specifically designed to simplify and streamline the development of web applications
- They provide a (structured) way to build and organize web applications, reducing the need for developers to start from scratch and re-invent the wheel

SOFTWARE LIBRARY

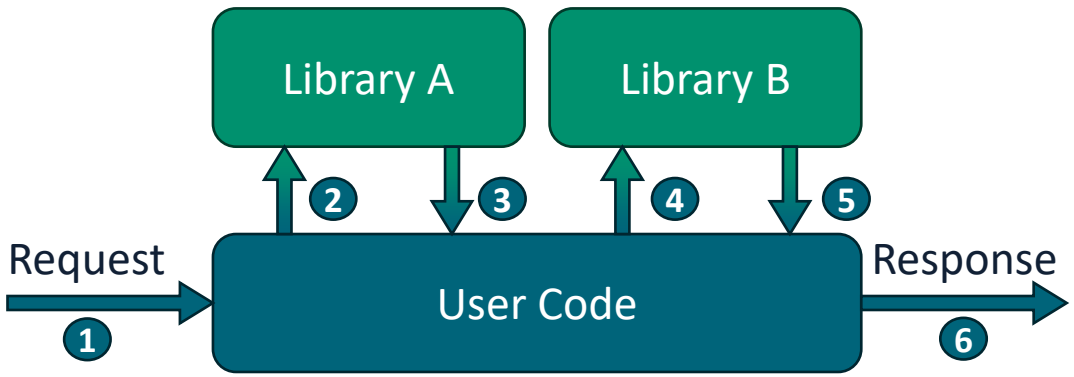
- Essentially a set of functions we can call to get some job done
- Each call does some work and then returns control to the caller
- User code (e.g.: our **main** method) is in control of the execution flow

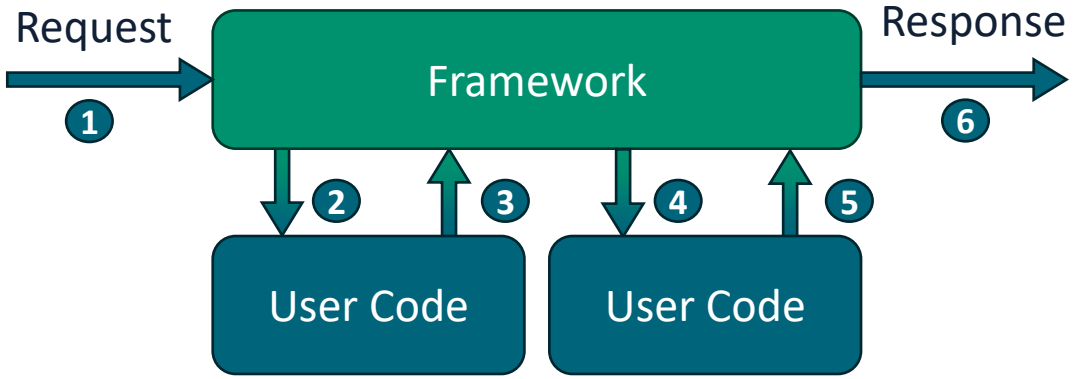
FRAMEWORK

- 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)

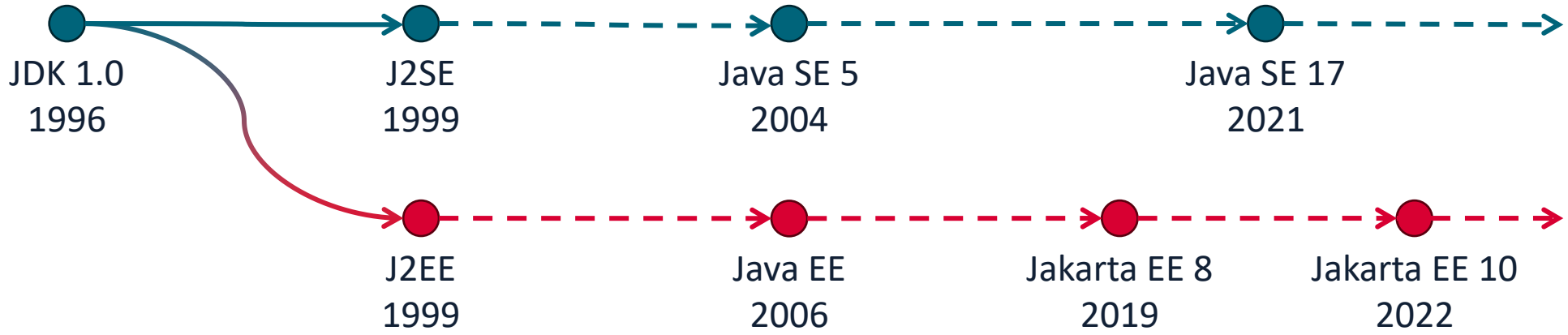
- Many web frameworks to choose from (e.g.: see Wikipedia)

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

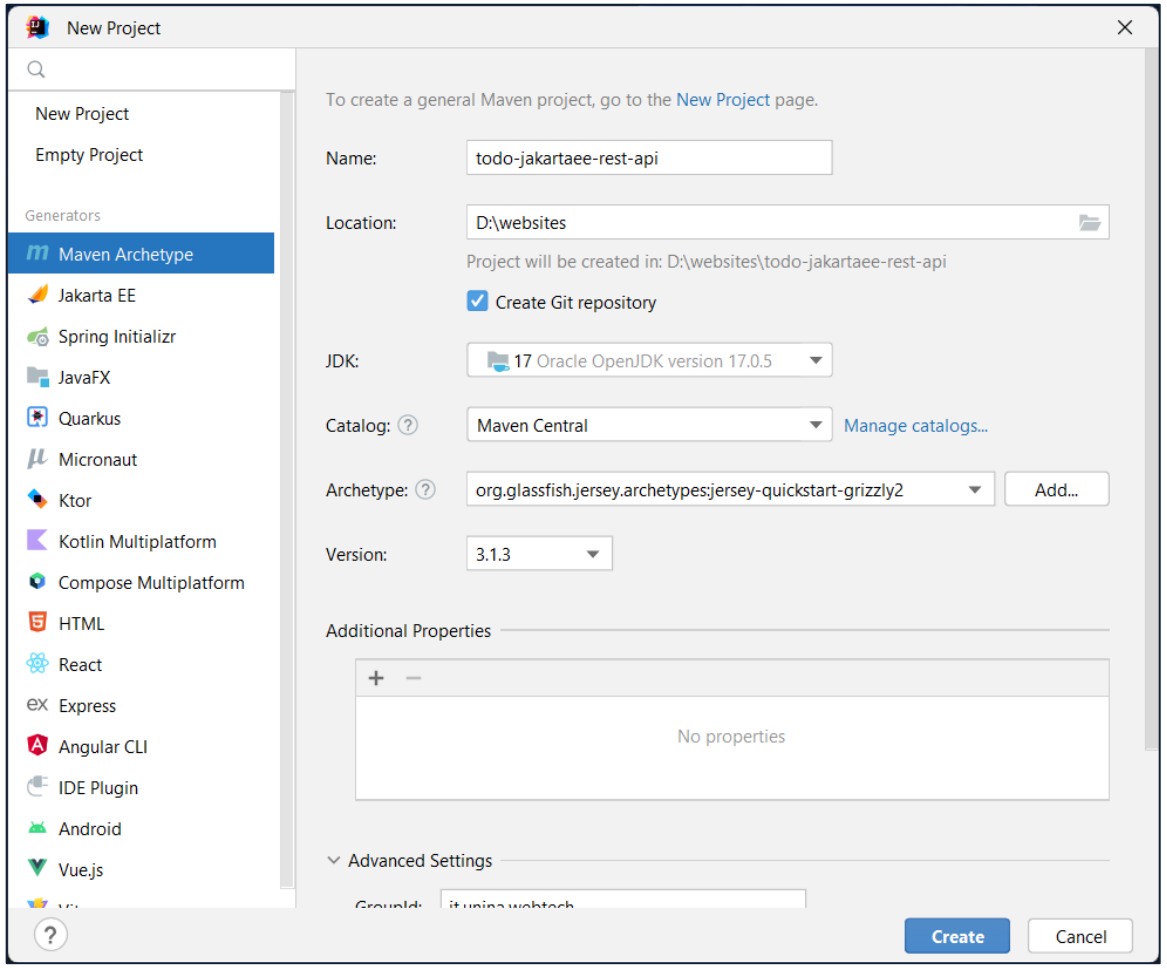
- JAX-RS is just a set of specifications
- The most well-known implementations are **frameworks** such as
  - Eclipse Jersey (we’re gonna use this in the demo!)
  - JBoss RESTEasy
- Annotation-based: specific Java annotations should be used to «tell» the frameworks when to run the code we write

A REST API USING JAX-RS

- We'll implement a basic To-do List REST API
- Using JAX-RS (with Eclipse Jersey)
- Dockerfile provided as well
- Let's take a look at the code!
- No worries, the code is available in a public GitHub repository (link at the end of these slides)

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)!

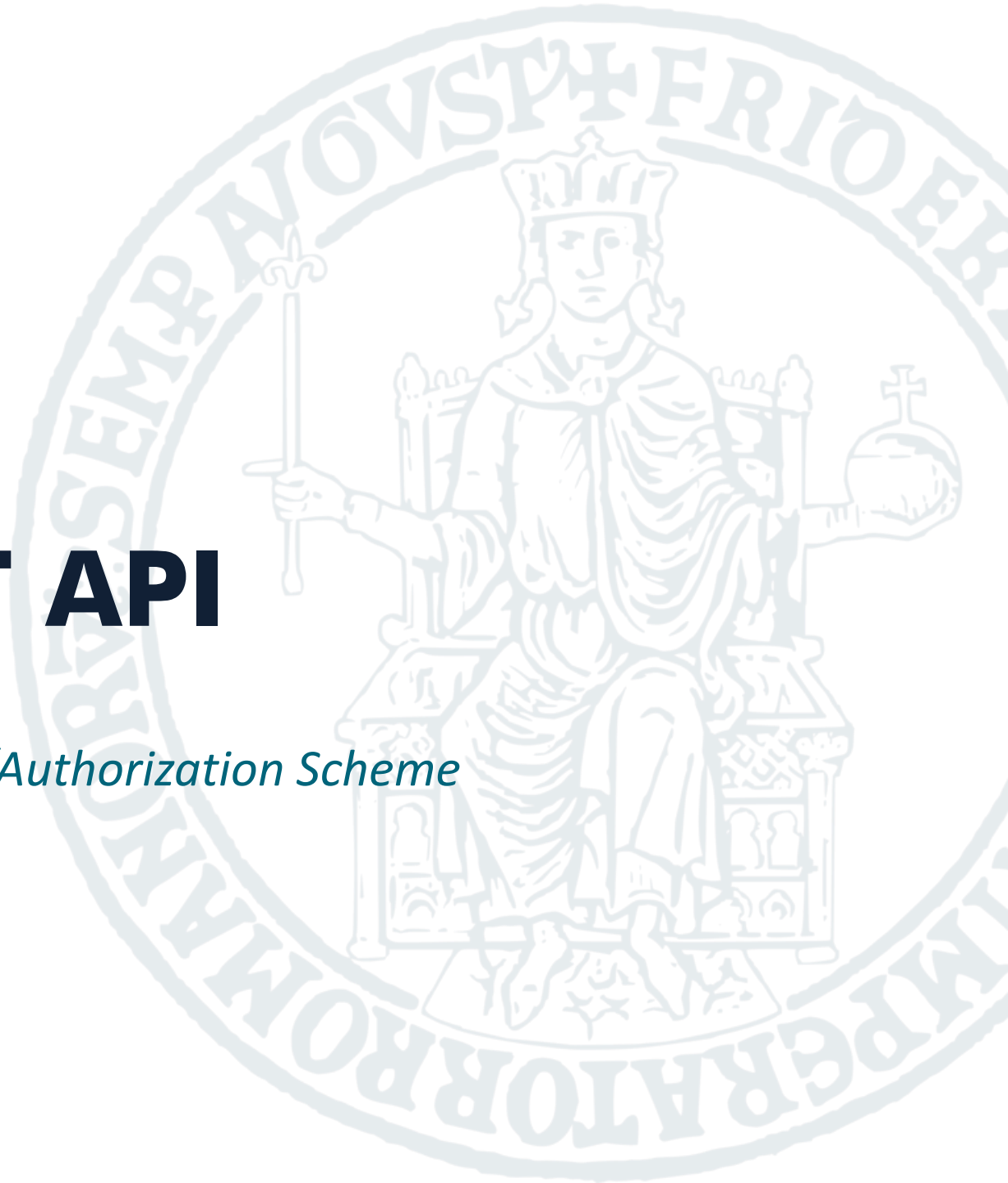
CODE EXAMPLES

```
@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)

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)

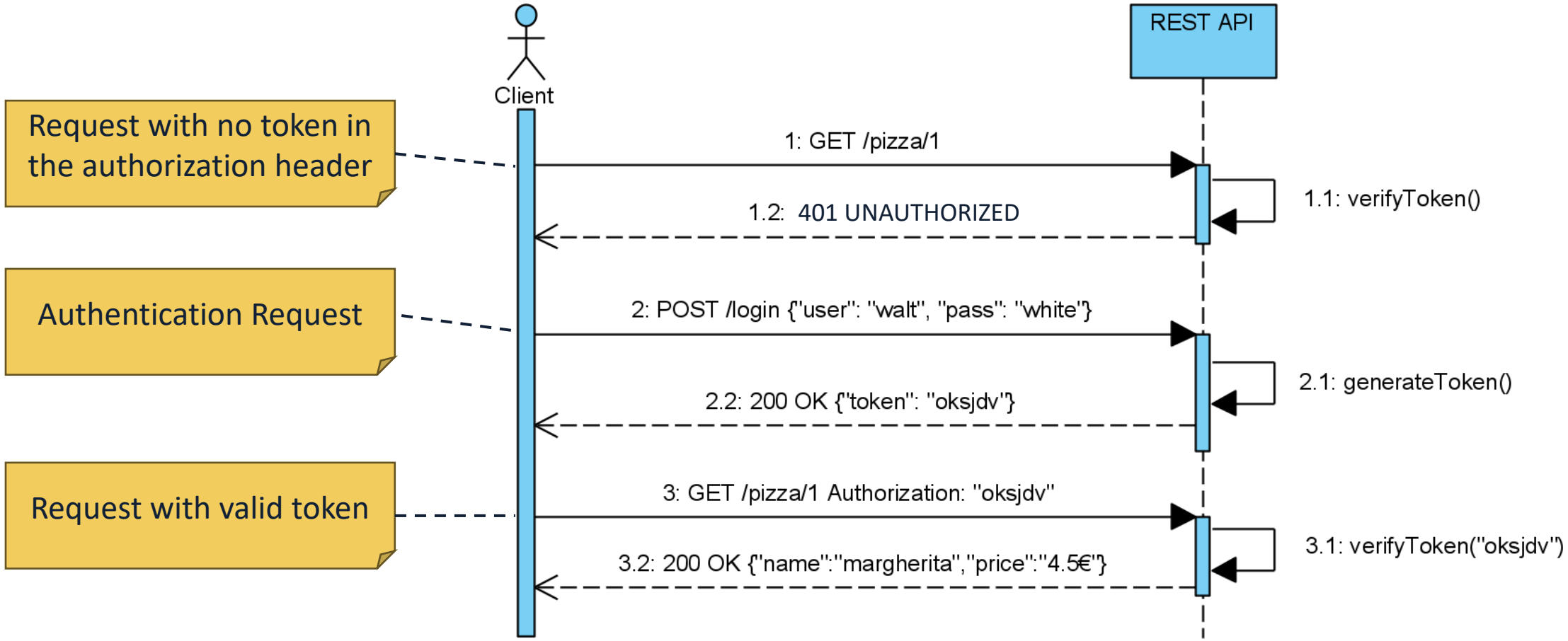
HOW TO SECURE REST APIS

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

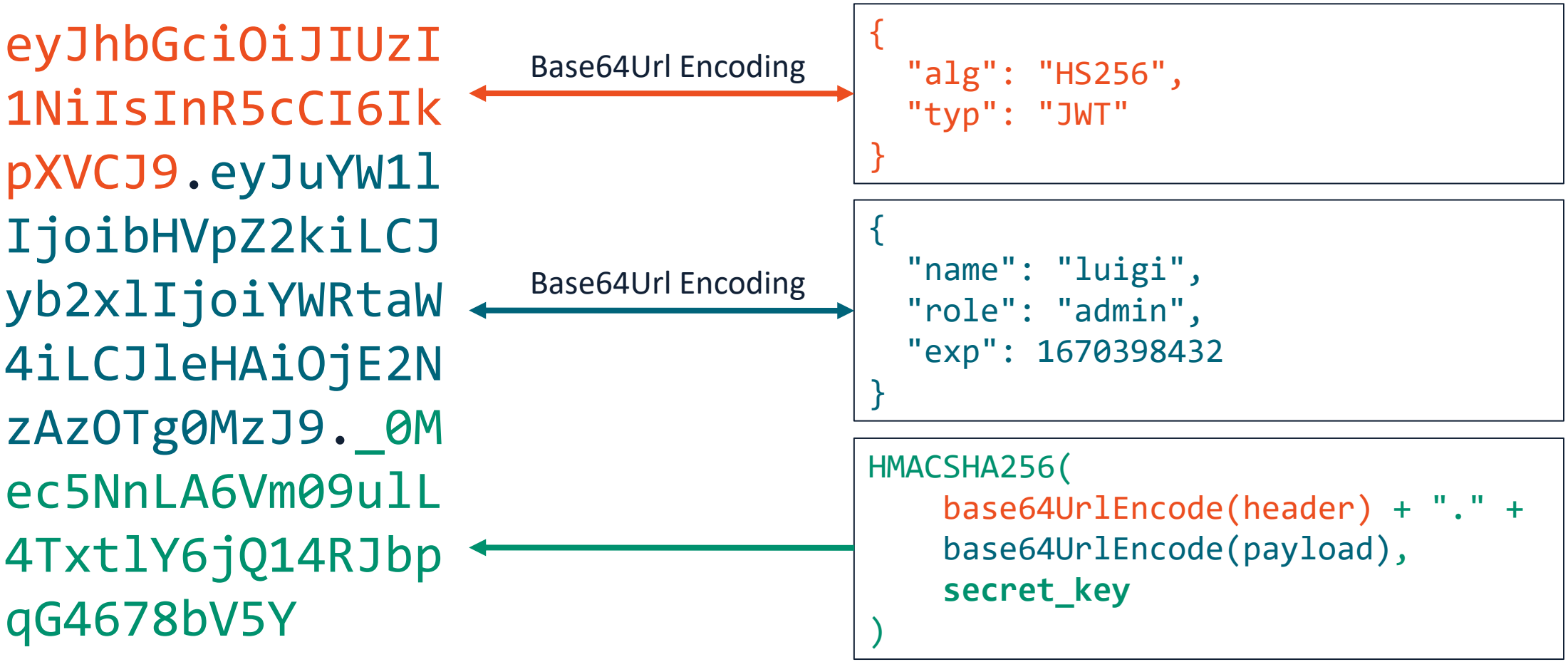


JSON WEB TOKEN (JWT)

- 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
```

JSON WEB TOKEN



JWT: VERIFICATION


```
eyJhbGciOiJIUzI
1NiIsInR5cCI6Ik
pXVCJ9.eyJuYW1l
IjoibHVpZ2kiLCJ
yb2xlIjoiYWRtaW
4iLCJleHAiOjE2N
zAzOTg0MzJ9._0M
ec5NnLA6Vm09ulL
4TxtlY6jQ14RJbp
qG4678bV5Y
```


When a server receives a **JWT**

1. Extracts "**headers.payload**"
2. Signs "**headers.payload**" with its secret key
   - Using the algorithm specified in the header
3. Checks that the **signature** is equal to the one attached to the JWT
   1. If equals, the JWT is **legit** and the claims contained in the payload were issued by the server
   2. If not, the JWT is **invalid**!

Demo: https://jwt.io/ (secret: `software_engineering`)

JWT AUTHENTICATION IN OUR JAX–RS API

- Let’s take a look at the code again!
- No worries, the JWT authentication code is available in a public GitHub repository (link at the end of these slides)

REPOSITORY

- The complete source code, Dockerfile, and instructions are available at https://github.com/luistar/jakartaee-rest-example
- An example using the Spring framework is available at https://github.com/luistar/rest-service-pizzashop

REFERENCES

- https://developer.mozilla.org/en-US/docs/Learn/Server-side