UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 15

THE REST PARADIGM

Luigi Libero Lucio Starace, PhD

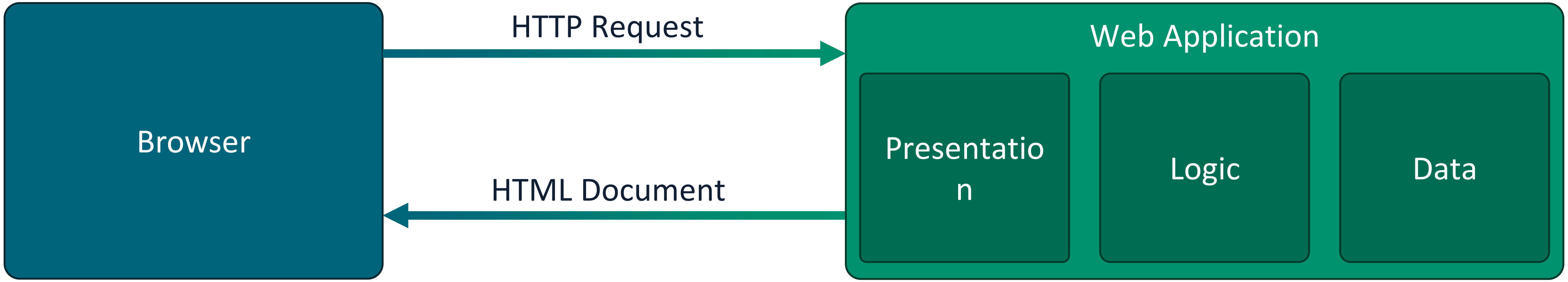
luigiliberolucio.starace@unina.it

https://luistar.github.io

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

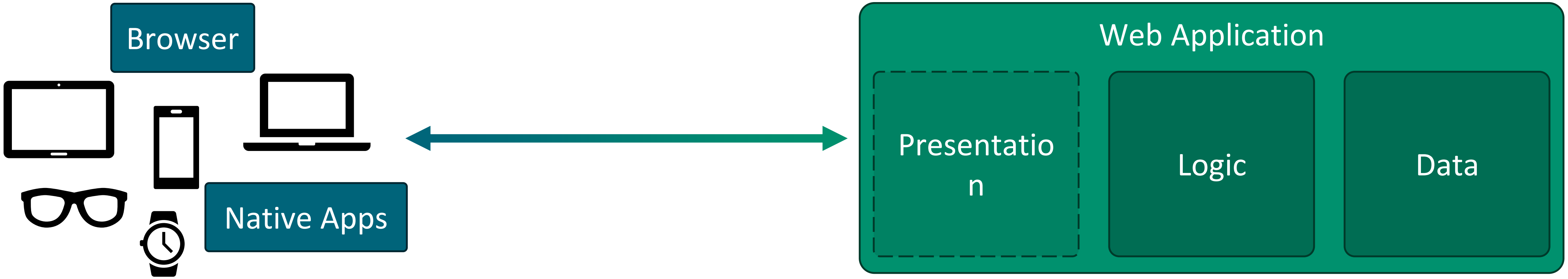
PREVIOUSLY, ON WEB TECHNOLOGIES

- In the last lecture, we developed a full-fledged web app with Express
- Our app took care of everything **on the Server**:
  - It managed **data** (with the Sequelize ORM)
  - It managed **business logic** (with routes, controllers, middlewares, sessions)
  - It managed **presentation** (rendered templates using Pug)
- Browsers were only responsible for visualizing HTML pages



‘TRADITIONAL’ WEB APPS

- This approach is often referred to as «**traditional**» web apps
  - Originated when Browsers had limited capabilities and Web content was generally accessed only via Browsers
- Nowadays, the scenario is more complex
  - Data might need to be accessed also by other software (e.g.: mobile apps)
  - Trend is to exploit the capabilities of modern browsers to deliver a more reactive user experience (i.e., single page apps - we’ll see in a few lectures)



‘TRADITIONAL’ WEB APPS

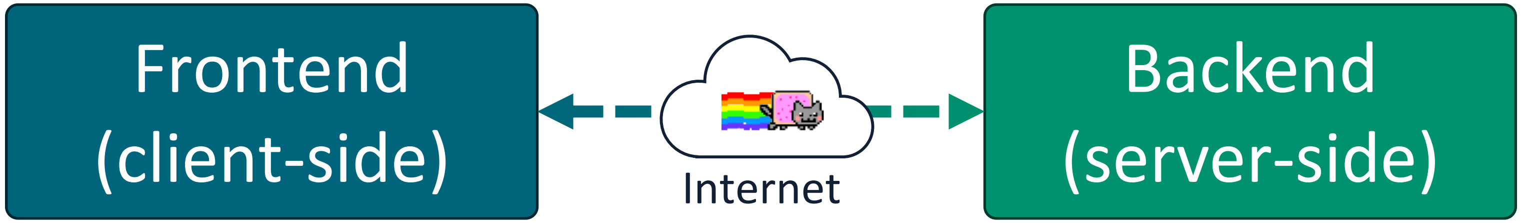
Traditional web apps are **not flexible enough** in some modern scenarios

- Other software willing to access data has to
  1. Download the HTML pages
  2. Parse and analyze them to extract relevant data
- **Not very efficient**: transfer a lot of unneded data
- **Not very robust**: any change in the HTML pages might break the extraction of the data

THE CURRENT TREND

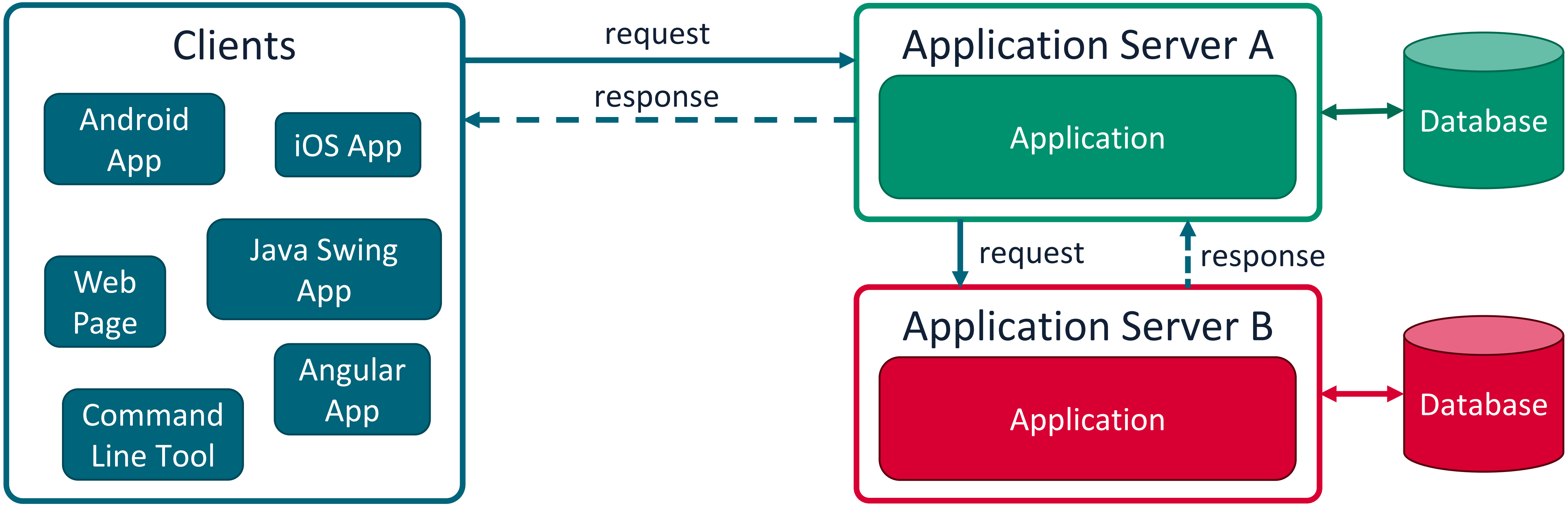
Current trend is to **split** web applications in two components:

- **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



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 (**APIs**) 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/

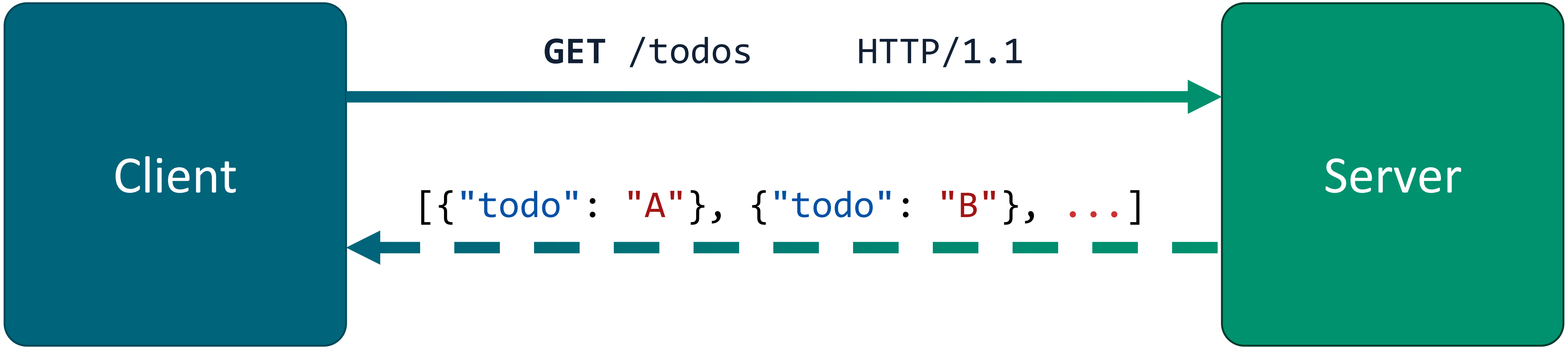
REST FUNDAMENTALS

- 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, …)



```
GET /todos     HTTP/1.1
```

```
[{"todo": "A"}, {"todo": "B"}, ...]
```

HTTP: DATA INTEREXCHANGE FORMATS

- Widely used formats include JSON and XML

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

```xml
<?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 /sellers/{id}/reviews`** lists all the reviews of a given seller.
- Keep in mind that every resource should have only one URI

MORE ON RESOURCES

Four basic resource archetypes can be considered:

- Document
- Collection
- Store
- Controller

RESOURCE ARCHETYPES: DOCUMENTS

- Document is a singular concept
  - Corresponds to an object instance or database record
- Each of the following URIs identifies a document resource:
  - /leagues/serie-a
  - /leagues/serie-a/teams/napoli
  - /leagues/serie-a/teams/napoli/players/mc-tominay

✅ **Rule**: Document names (if not numeric ids) should be **singular**

RESOURCE ARCHETYPES: COLLECTIONS

Collections represent **web-app managed** directories of resources

- Clients may propose to add new resources to a collection
  - Resource decides whether to accept the addition or not
- Each of the following URIs identifies a collection resource:
  - /leagues
  - /leagues/serie-a/teams
  - /leagues/serie-a/teams/napoli/players

✅ **Rule**: Collection names should be **plural**

RESOURCE ARCHETYPES: STORES

Collections represent **client-managed** directories of resources

- Clients can manage resources in a store
  - Resource can be created, retrieved, updated, and deleted
- Each of the following URIs identifies a store resource:
  - /users/1234/favourites

✅ **Rule**: Store names should be **plural**

RESOURCE ARCHETYPES: CONTROLLERS

Controller resources model some **procedural** concept

- You can think of them like **executable functions**
  - Can be invoked with specific parameters, and return a given output or produce a given side-effect
- Each of the following URIs identifies a controller resource:
  - /alerts/1234/resend
  - /login
  - /reset-app

✅ **Rule**: Controller names typically are verbs or verb phrases

URI DESIGN: QUERY AND FILTERING

Often, when retrieving resources, we may want to apply some filtering

✅ **Rule**: The query component of URIs should be used to filter collections/stores

`GET /users`	retrieves all users
`GET /users?role=admin`	retrieves all users with the specified role
`GET /users?age.gt=18&age.lte=40`	

URI DESIGN: QUERY AND FILTERING

Often, when retrieving resources, we may want to apply pagination

✅ **Rule**: The query component of URIs should be used to paginate collection/store results

`GET /todos?size=10&page=2` Get 11th to 20th results (i.e., page 2)

URI DESIGN: QUERY AND FILTERING

When the complexity of pagination/filtering requirements exceed the formatting capabilities of the query string, you may introduce a dedicated **controller resource** in a collection/store

```
POST /users/search
```

This allows client to pass complex filtering criteria as an object in the request body

IMPLEMENTING A REST API

- REST APIs are conceptually simple
- We just need to listen for HTTP requests, and handle them
- Once done handling the request, we send a HTTP response
- That’s the same things we already did with our traditional web app!
  - Instead of responding with an HTML representation to be rendered in a browser, we use a more machine-friendly representation (e.g. JSON)
  - Instead of getting input via form submissions, we get inputs in a machine-friendly representation (e.g.: JSON)
  - We should not rely on sessions, and other authentication schemes should be put in place

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)

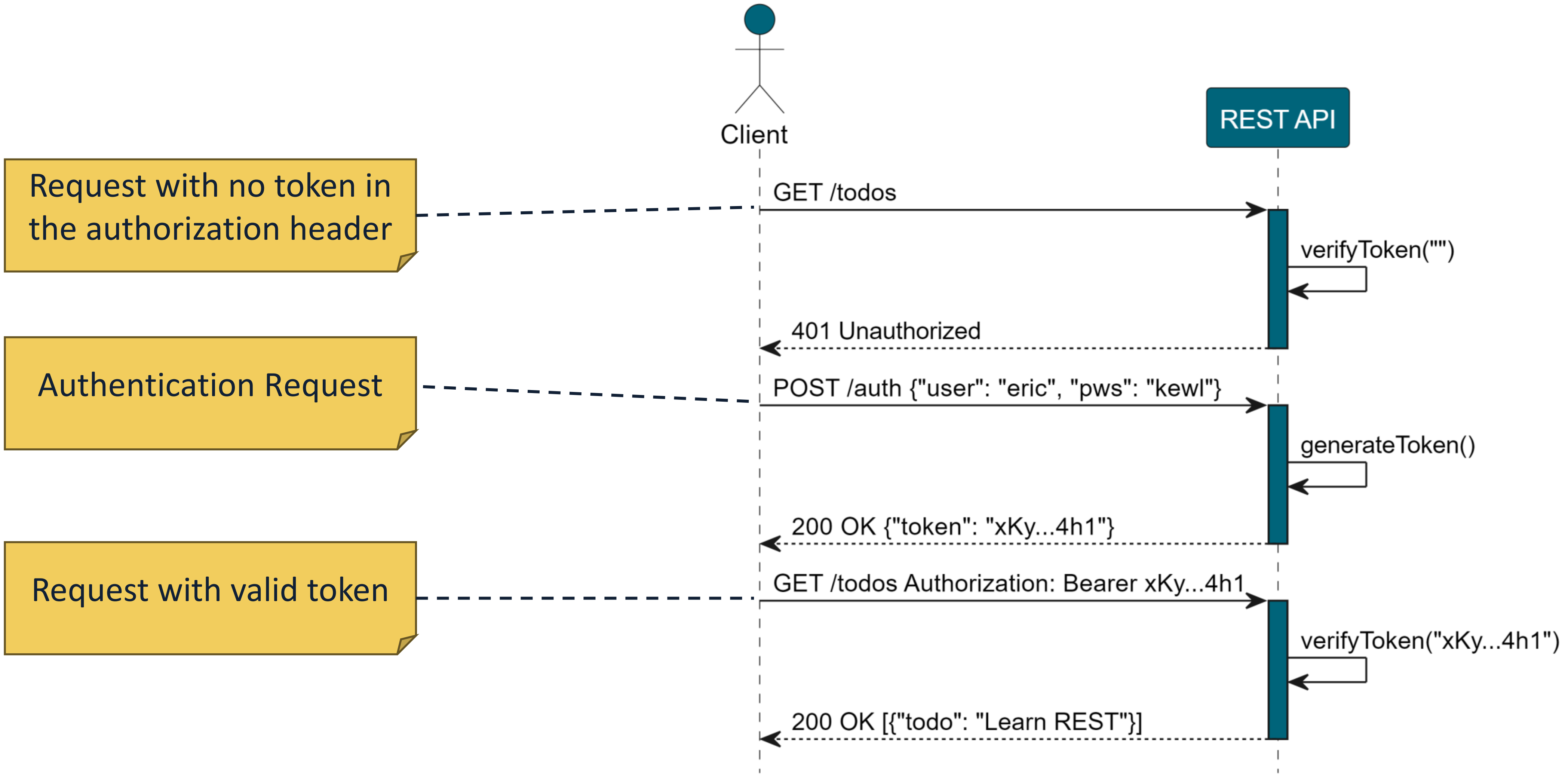
HOW TO SECURE REST APIS

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



Client

REST API

Request with no token in the authorization header

GET /todos

verifyToken("")

401 Unauthorized

Authentication Request

POST /auth {"user": "eric", "pws": "kewl"}

generateToken()

200 OK {"token": "xKy...4h1"}

Request with valid token

GET /todos Authorization: Bearer xKy...4h1

verifyToken("xKy...4h1")

200 OK [{"todo": "Learn REST"}]

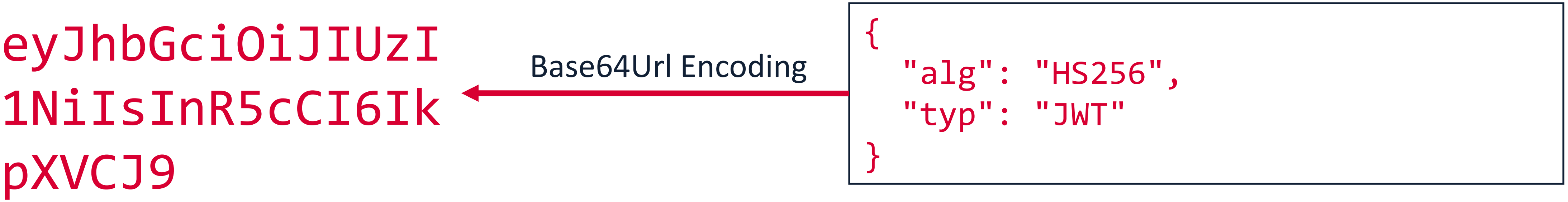
JSON WEB TOKEN (JWT)

- 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!**



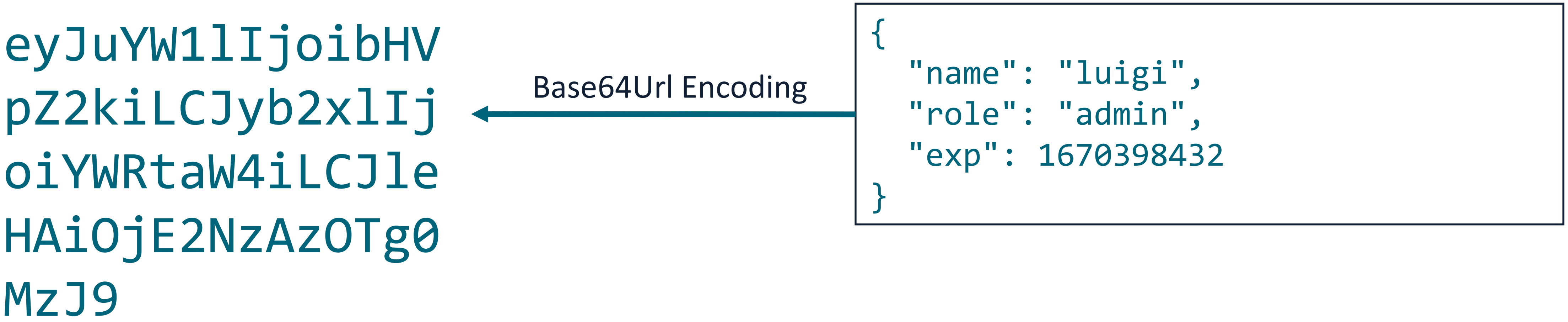
```
eyJhbGciOiJIUzI
1NiIsInR5cCI6Ik
pXVCJ9
```

Base64Url Encoding

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

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



```
eyJuYW1lIjoibHV
pZ2kiLCJyb2xlIj
oiYWRtaW4iLCJle
HAiOjE2NzAzOTg0
MzJ9
```

Base64Url Encoding

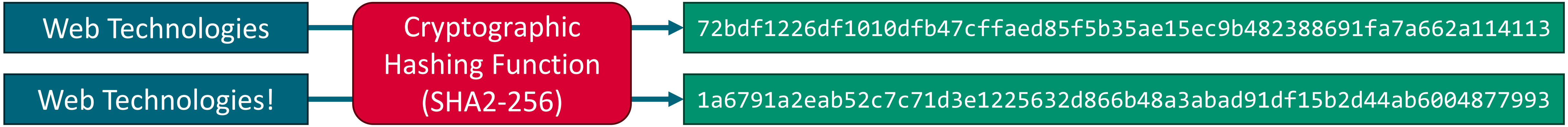
```
{
  "name": "luigi",
  "role": "admin",
  "exp": 1670398432
}
```

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

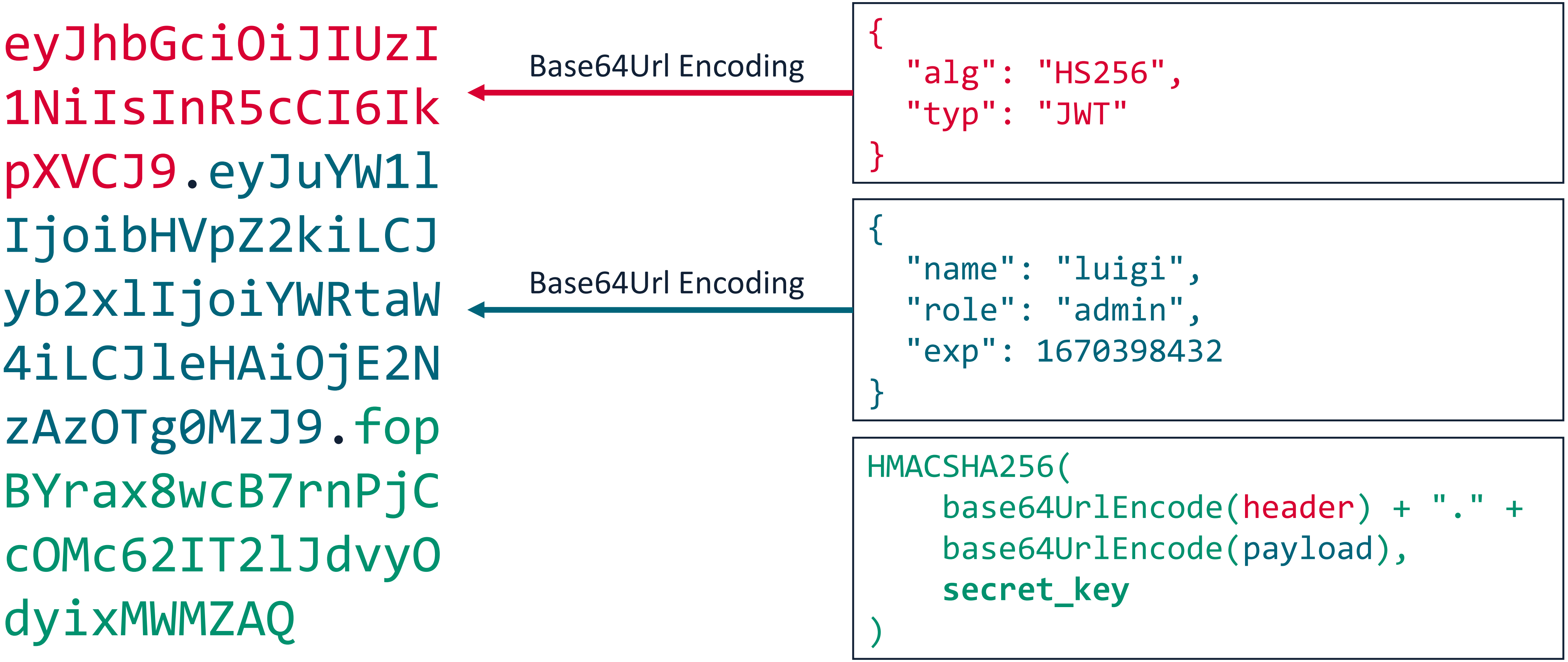
Functions that map an arbitrary binary string (**input**) to a fixed-size binary string (**digest** or **hash**)


Web Technologies
Web Technologies!
Cryptographic Hashing Function (SHA2-256)
72bdf1226df1010dfb47cffaed85f5b35ae15ec9b482388691fa7a662a114113
1a6791a2eab52c7c71d3e1225632d866b48a3abad91df15b2d44ab6004877993

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



```
eyJhbGciOiJIUzI
1NiIsInR5cCI6Ik
pXVCJ9.eyJuYW1l
IjoibHVpZ2kiLCJ
yb2xlIjoiYWRtaW
4iLCJleHAiOjE2N
zAzOTg0MzJ9.fop
BYrax8wcB7rnPjC
cOMc62IT2lJdvyO
dyixMWMZAQ
```

Base64Url Encoding

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Base64Url Encoding

```
{
  "name": "luigi",
  "role": "admin",
  "exp": 1670398432
}
```

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret_key
)
```

TO–DO LIST REST API WITH EXPRESS

A REST API USING EXPRESS

- We will implement a REST API for our To-do list app
- We will use Express, and re-use most of the code from our Express To-do List web app
- Let’s take a look at the code!
- Code available in Course Materials
- The following slides show some highlights from the demo.

CONFIGURING MIDDLEWARES

```
const app = express();
const PORT = 3000;

// Register the morgan logging middleware, use the 'dev' format
app.use(morgan('dev'));

app.use(cors()); //API will be accessible from anywhere. We'll talk about this in
Lecture 23!

// Parse incoming requests with a JSON payload
app.use(express.json());
```

ERROR HANDLING

```
//error handler
app.use( (err, req, res, next) => {
  console.log(err.stack);
  res.status(err.status || 500).json({
    code: err.status || 500,
    description: err.message || "An error occurred"
  });
});
```

AUTHENTICATION: ROUTES

```
authenticationRouter.post("/auth", async (req, res) => {
  let isAuthenticated = await AuthController.checkCredentials(req, res);
  if(isAuthenticated){
    res.json(AuthController.issueToken(req.body.usr));
  } else {
    res.status(401);
    res.json( {error: "Invalid credentials. Try again."});
  }
});

authenticationRouter.post("/signup", (req, res, next) => {
  AuthController.saveUser(req, res).then((user) => {
    res.json(user);
  }).catch((err) => {
    next({status: 500, message: "Could not save user"});
  })
});
```

AUTHENTICATION: JWT

```
import Jwt from "jsonwebtoken";

static issueToken(username){
  return Jwt.sign({user:username}, process.env.TOKEN_SECRET, {
     expiresIn: `${24*60*60}s`
  });
}

static isTokenValid(token, callback){
  Jwt.verify(token, process.env.TOKEN_SECRET, callback);
}
```

AUTHENTICATION: MIDDLEWARE

```
export function enforceAuthentication(req, res, next){
  const authHeader = req.headers['authorization']
  const token = authHeader?.split(' ')[1];
  if(!token){
    next({status: 401, message: "Unauthorized"});
    return;
  }
  AuthController.isTokenValid(token, (err, decodedToken) => {
    if(err){
      next({status: 401, message: "Unauthorized"});
    } else {
      req.username = decodedToken.user;
      next();
    }
  });
}
```

ENFORCING AUTHORIZATION

- In our API, we want users to be able to modify and visualize only their own To-do items!
- What if an user tries to send a DELETE request for a To-do item that does not belong to them?
- We should also check that the client actually has permission to interact with a resource!

AUTHORIZATION MIDDLEWARE

- Example of sensible route protected with the authorization middleware

```
todoRouter.get("/todos/:id", ensureUsersModifyOnlyOwnTodos, (req, res, next) => {
  TodoController.findById(req).then( (item) => {
    if(item)
      res.json(item);
    else
      next({status: 404, message: "Todo not found"});
  }).catch( err => {
    next(err);
  })
});
```

AUTHORIZATION MIDDLEWARE

```
export async function ensureUsersModifyOnlyOwnTodos(req, res, next){
  const user = req.username;
  const todoId = req.params.id;
  const userHasPermission = await AuthController.canUserModifyTodo(user, todoId);
  if(userHasPermission){
    next();
  } else {
    next({
      status: 403,
      message: "You are forbidden to view or modify this resource"
    });
  }
}
```

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

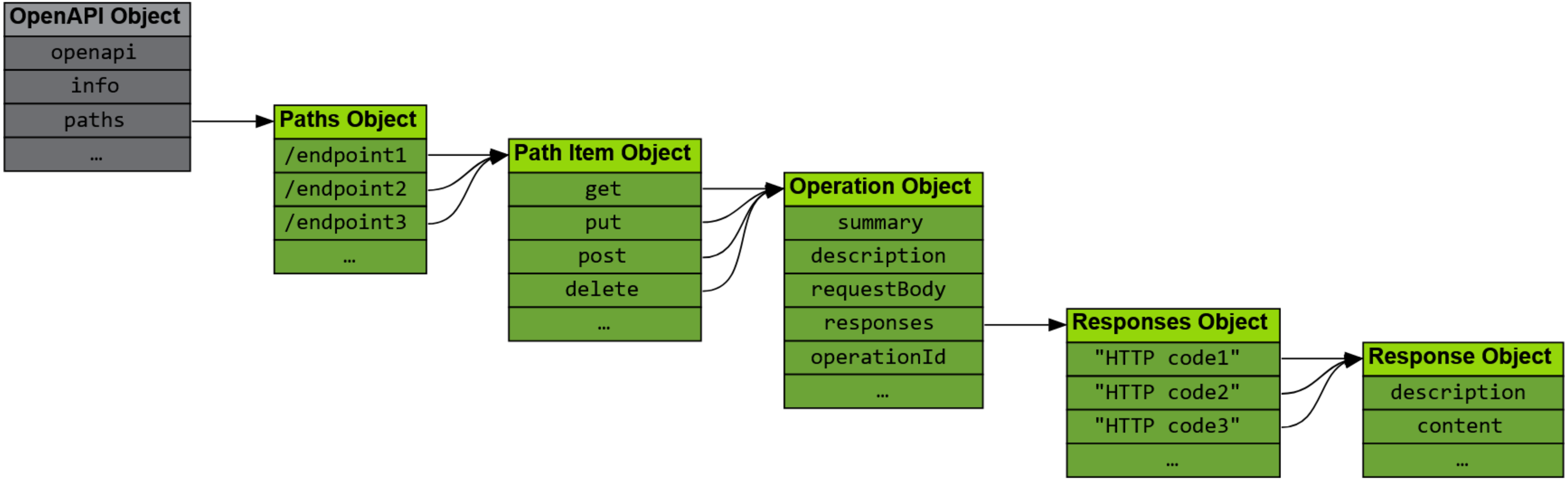


Image from https://learn.openapis.org/specification/paths

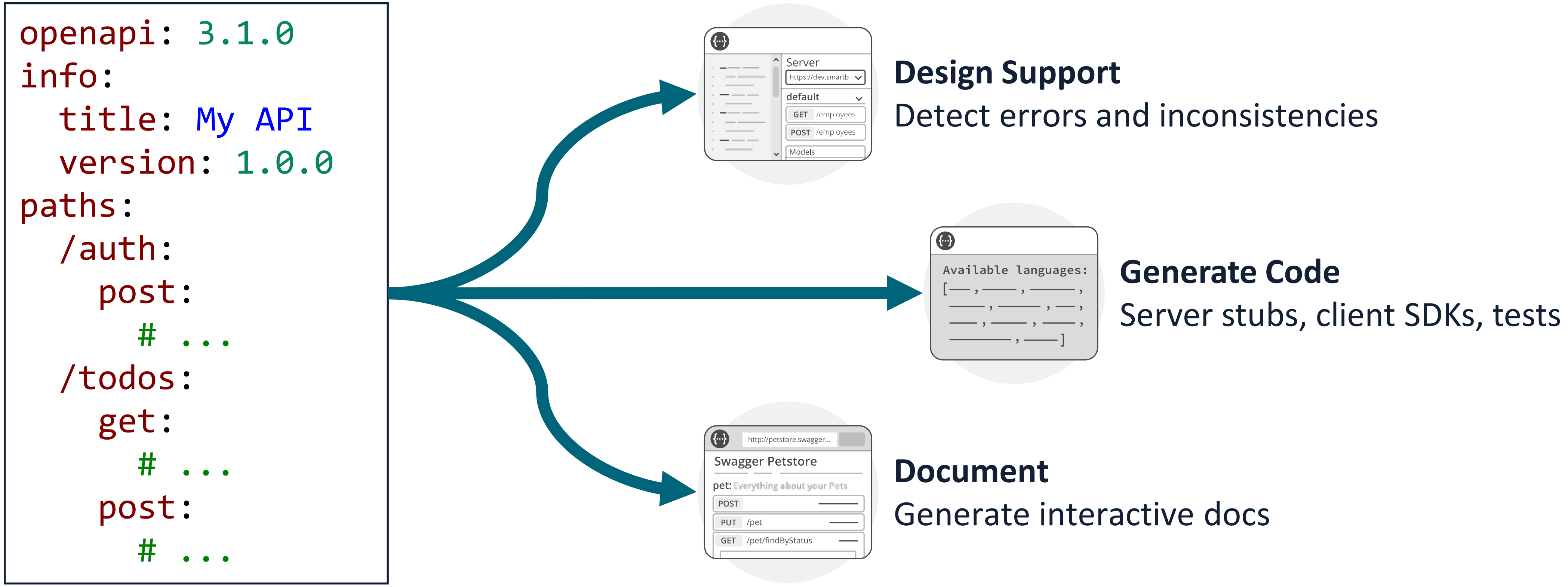
PATH EXAMPLE

- /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

```
/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

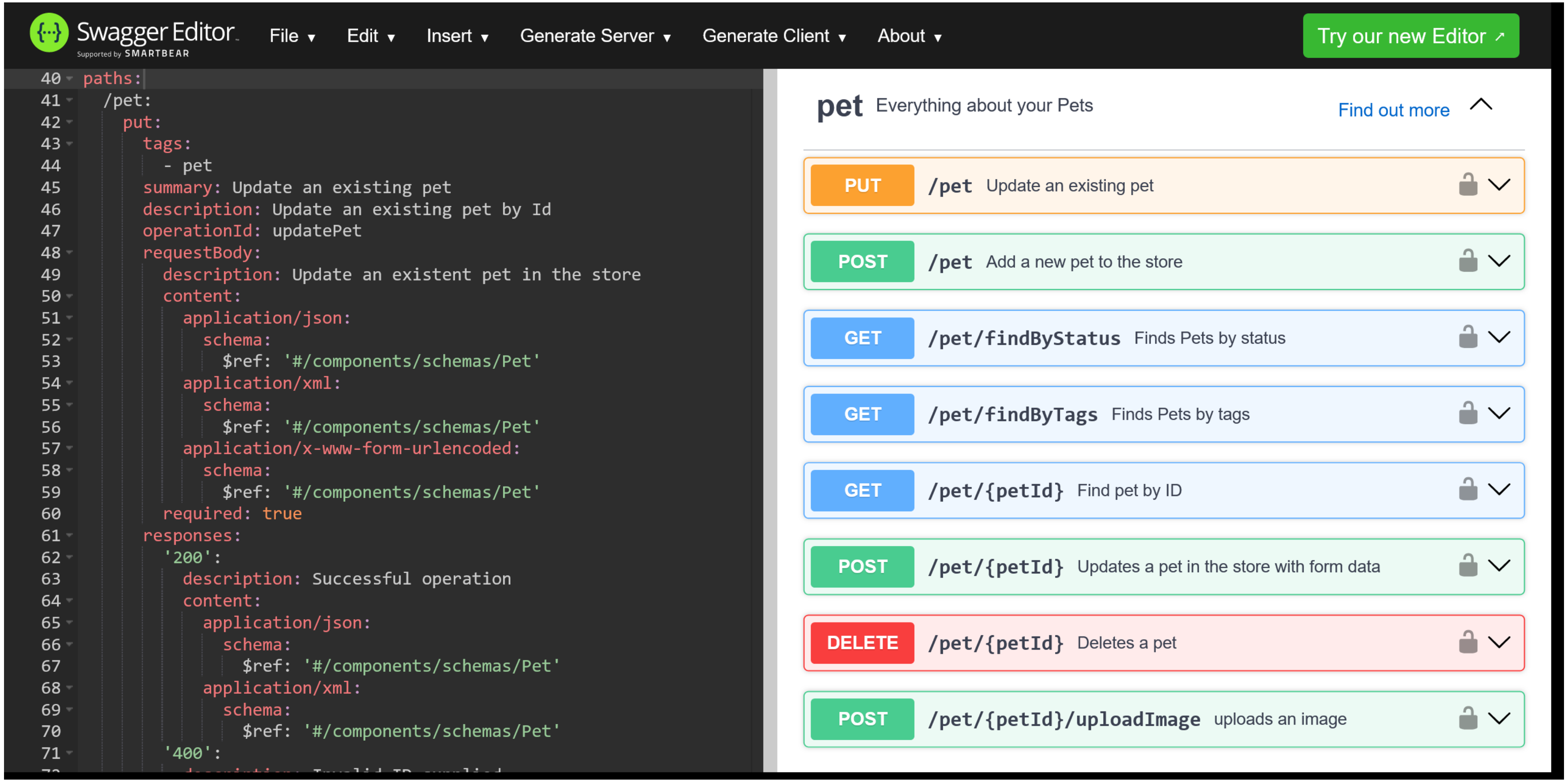
Document

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



```yaml
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/Pet'
        required: true
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
            application/xml:
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid ID supplied
```

pet Everything about your Pets — Find out more

- PUT `/pet` Update an existing pet
- POST `/pet` Add a new pet to the store
- GET `/pet/findByStatus` Finds Pets by status
- GET `/pet/findByTags` Finds Pets by tags
- GET `/pet/{petId}` Find pet by ID
- POST `/pet/{petId}` Updates a pet in the store with form data
- DELETE `/pet/{petId}` Deletes a pet
- POST `/pet/{petId}/uploadImage` uploads an image

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

GENERATING OPENAPI SPECS FOR OUR API

We will use two Node packages

- swagger-jsdoc: automatically generates OpenAPI specifications based on annotations in our source code.
- swagger-ui-express: serves automatically generated Swagger UI documentation from Express, using an existing OpenAPI specification file (we’ll use the one generated by swagger-jsdoc.

```
@luigi → express-hello-world $ npm install swagger-jsdoc
@luigi → express-hello-world $ npm install swagger-ui-express
```

GENERATING OPENAPI SPECS FOR OUR API

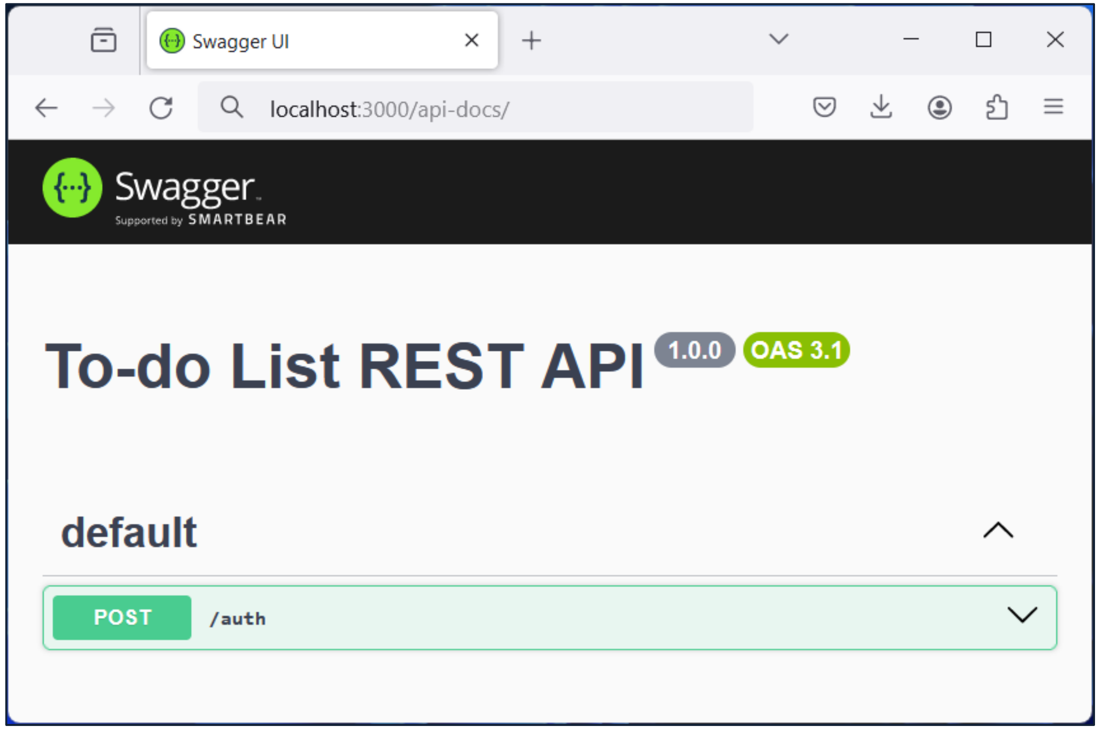
```
// generate OpenAPI spec and show Swagger UI

// Initialize swagger-jsdoc -> returns validated Swagger spec in json format
const swaggerSpec = swaggerJSDoc({
  definition: {
    openapi: '3.1.0',
    info: {
      title: 'To-do List REST API',
      version: '1.0.0',
    },
  },
  apis: ['./routes/*Router.js'], // files containing annotations
});

//Swagger UI will be available at /api-docs
app.use('/api-docs', swaggerUI.serve, swaggerUI.setup(swaggerSpec));
```

GENERATING OPENAPI SPECS FOR OUR API

```
/**
 * @swagger
 *  /auth:
 *    post:
 *      description: Authenticate user
 *      requestBody:
 *        content:
 *          application/json:
 *            schema:
 *              type: object
 *              properties:
 *                usr:
 *                  type: string
 *                pwd:
 *                  type: string
 */
authenticationRouter.post("/auth", async (req, res) => {/*...*/});
```

Note: Some parts of the Path specification were omitted for the sake of brevity

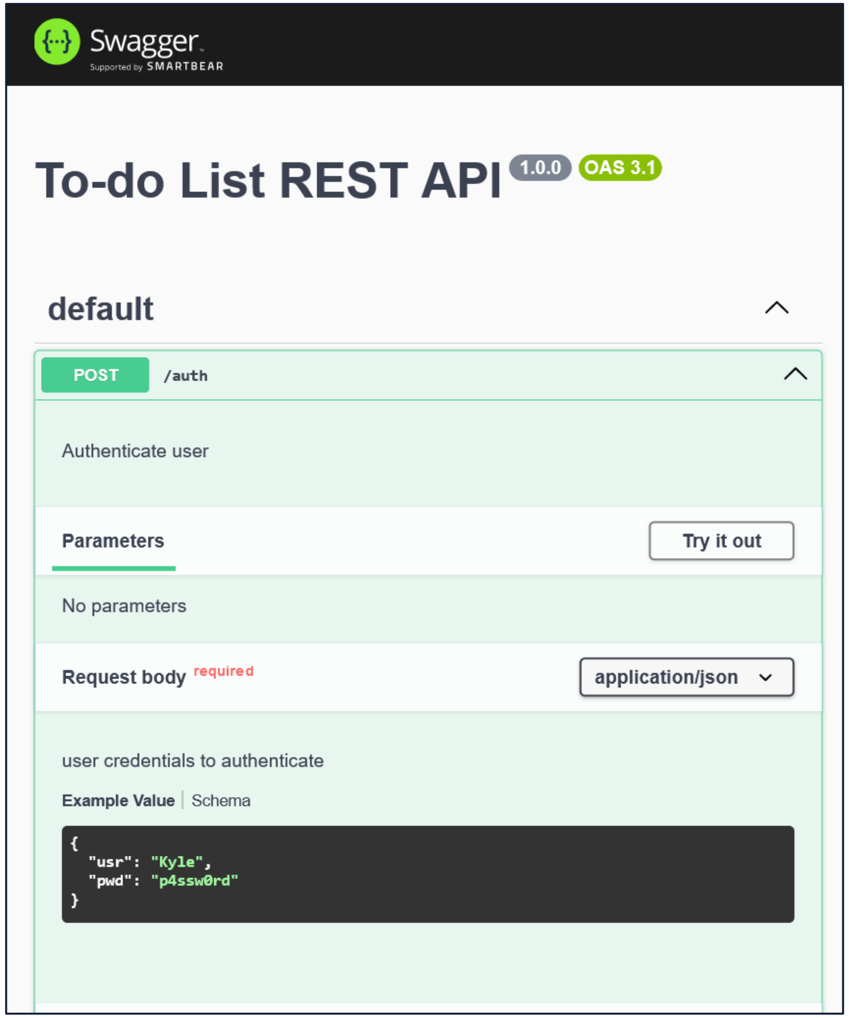
GENERATING OPENAPI SPECS FOR OUR API



To-do List REST API 1.0.0 OAS 3.1

default

POST /auth



To-do List REST API 1.0.0 OAS 3.1

default

POST /auth

Authenticate user

Parameters

No parameters

Request body required

application/json

user credentials to authenticate

Example Value | Schema

```
{
  "usr": "Kyle",
  "pwd": "p4ssw0rd"
}
```

REST: RULES AND GUIDELINES

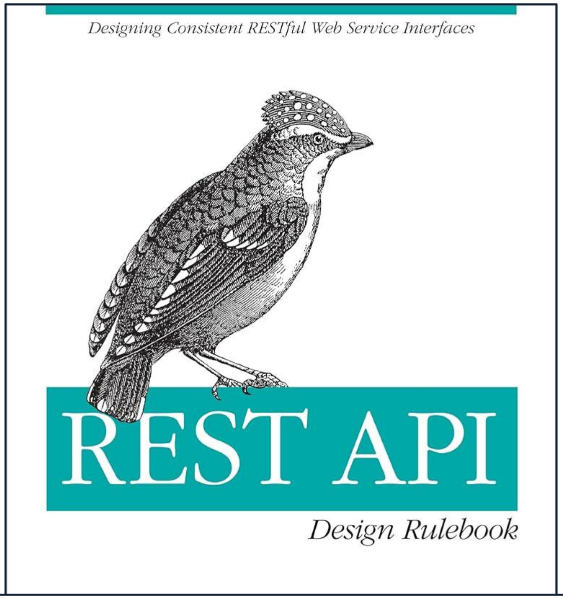
SOME RULES TO CONSIDER

- 401 Unauthorized must be used when problems with credentials arise
- Plural nouns should be used for collection/store names
- Singular nouns should be used for document names
- Controller names should include a verb or phrase
- Trailing slash should not be included in URIs
- CRUD function names should not be used in URIs
- Forward slash (/) must be used to indicated hierarchical relationships
- GET must be used only to retrieve a representation of a resource
- URIs should be lowercase and not contain underscores (_)
- Hyphens (-) should be used to make URIs more readable (e.g.: in place of CamelCase)

REST: RULES AND GUIDELINES

A good set of basic rules/guidelines (including the ones in the previous slide) was defined in

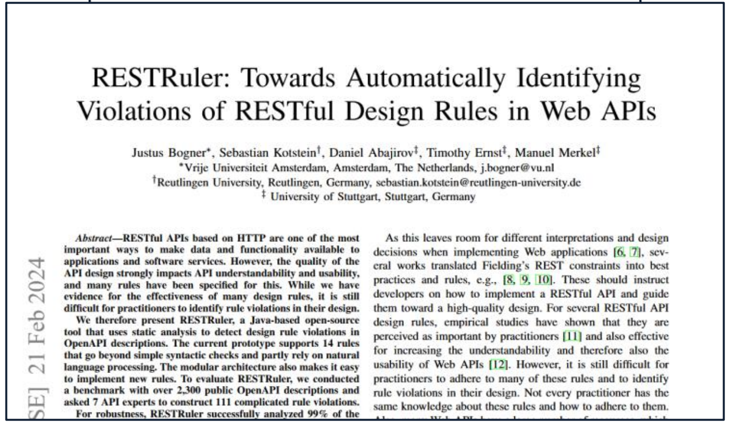
- Masse, Mark. *REST API design rulebook: designing consistent RESTful web service interfaces*. O'Reilly Media, Inc.", 2011.
- Researchers have developed tool to automatically detect violations of these rules (REST anti-patterns)
  - Bogner, J., et al. (2024, June). RESTRuler: towards automatically identifying violations of RESTful design rules in web APIs. In *2024 IEEE 21st International Conference on Software Architecture (ICSA).* https://arxiv.org/abs/2402.13710



Designing Consistent RESTful Web Service Interfaces

REST API

Design Rulebook



RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs

Justus Bogner*, Sebastian Kotstein†, Daniel Abajirov‡, Timothy Ernst‡, Manuel Merkel‡
*Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, j.bogner@vu.nl
†Reutlingen University, Reutlingen, Germany, sebastian.kotstein@reutlingen-university.de
‡ University of Stuttgart, Stuttgart, Germany

SE] 21 Feb 2024

***Abstract*—RESTful APIs based on HTTP are one of the most important ways to make data and functionality available to applications and software services. However, the quality of the API design strongly impacts API understandability and usability, and many rules have been specified for this. While we have evidence for the effectiveness of many design rules, it is still difficult for practitioners to identify rule violations in their design.**

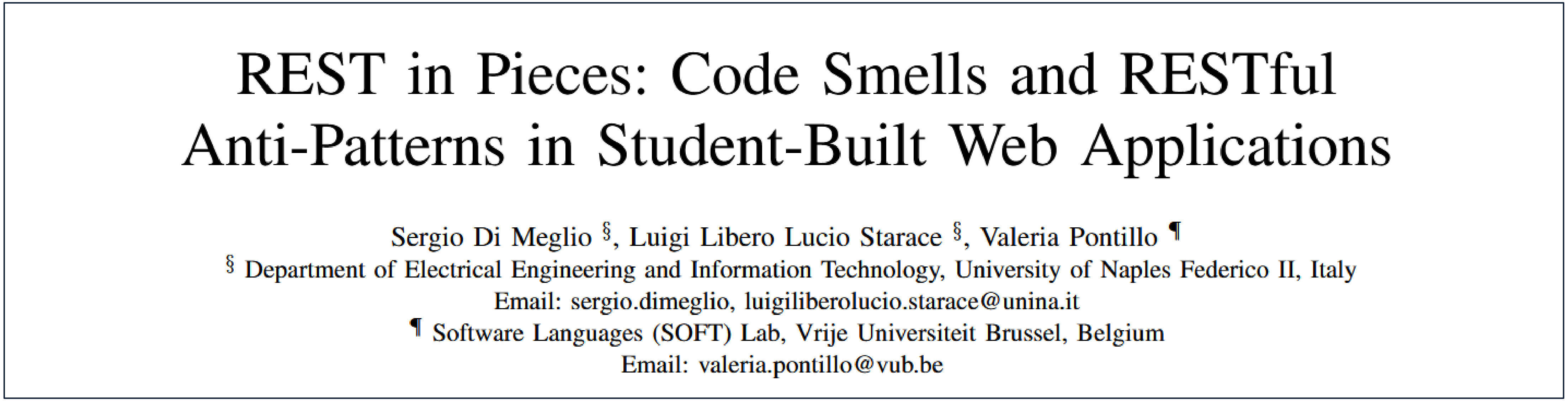
We therefore present RESTRuler, a Java-based open-source tool that uses static analysis to detect design rule violations in OpenAPI descriptions. The current prototype supports 14 rules that go beyond simple syntactic checks and partly rely on natural language processing. The modular architecture also makes it easy to implement new rules. To evaluate RESTRuler, we conducted a benchmark with over 2,300 public OpenAPI descriptions and asked 7 API experts to construct 111 complicated rule violations.

For robustness, RESTRuler successfully analyzed 99% of the

As this leaves room for different interpretations and design decisions when implementing Web applications [6, 7], several works translated Fielding's REST constraints into best practices and rules, e.g., [8, 9, 10]. These should instruct developers on how to implement a RESTful API and guide them toward a high-quality design. For several RESTful API design rules, empirical studies have shown that they are perceived as important by practitioners [11] and also effective for increasing the understandability and therefore also the usability of Web APIs [12]. However, it is still difficult for practitioners to adhere to many of these rules and to identify rule violations in their design. Not every practitioner has the same knowledge about these rules and how to adhere to them.

REST: RULES AND GUIDELINES

- I’m also doing some research on the topic…
  - Still a WIP

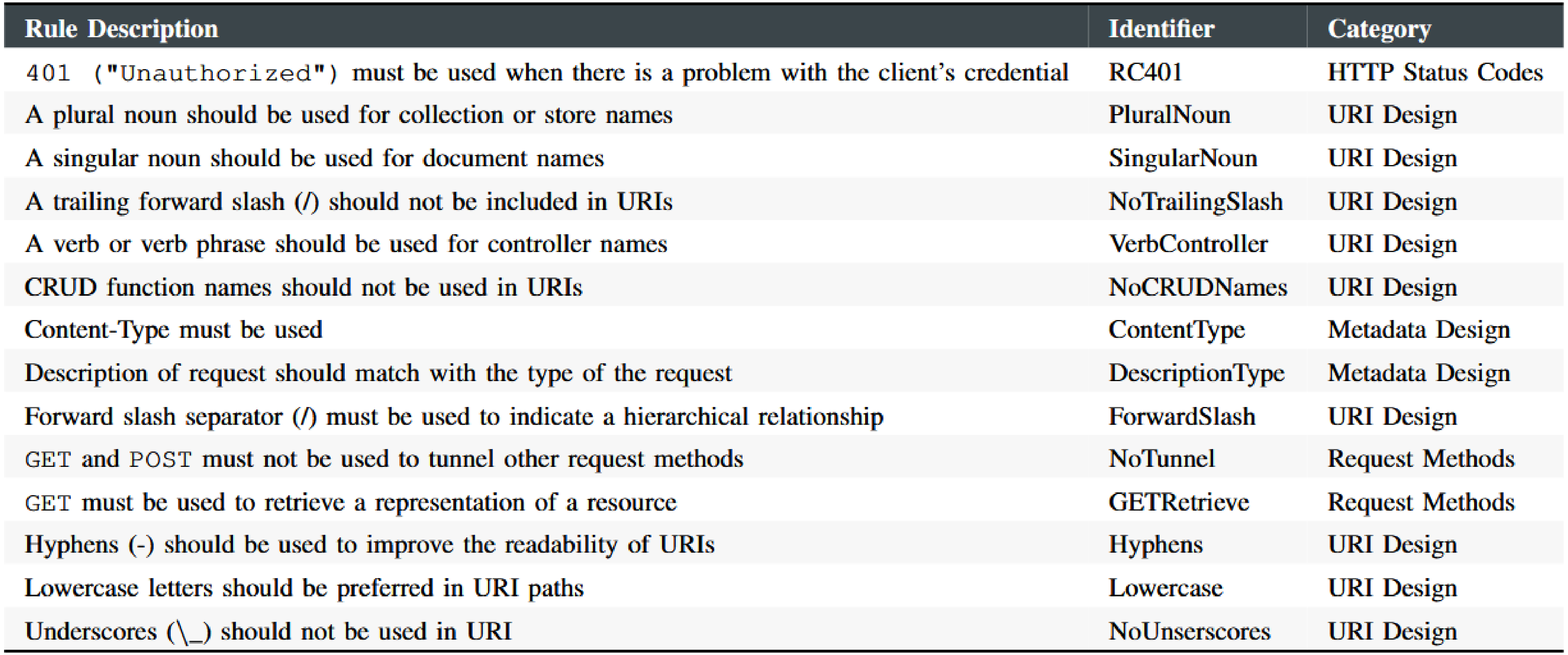


REST in Pieces: Code Smells and RESTful Anti-Patterns in Student-Built Web Applications

Sergio Di Meglio [§], Luigi Libero Lucio Starace [§], Valeria Pontillo [¶]
[§] Department of Electrical Engineering and Information Technology, University of Naples Federico II, Italy
Email: sergio.dimeglio, luigiliberolucio.starace@unina.it
[¶] Software Languages (SOFT) Lab, Vrije Universiteit Brussel, Belgium
Email: valeria.pontillo@vub.be

REST IN PIECES

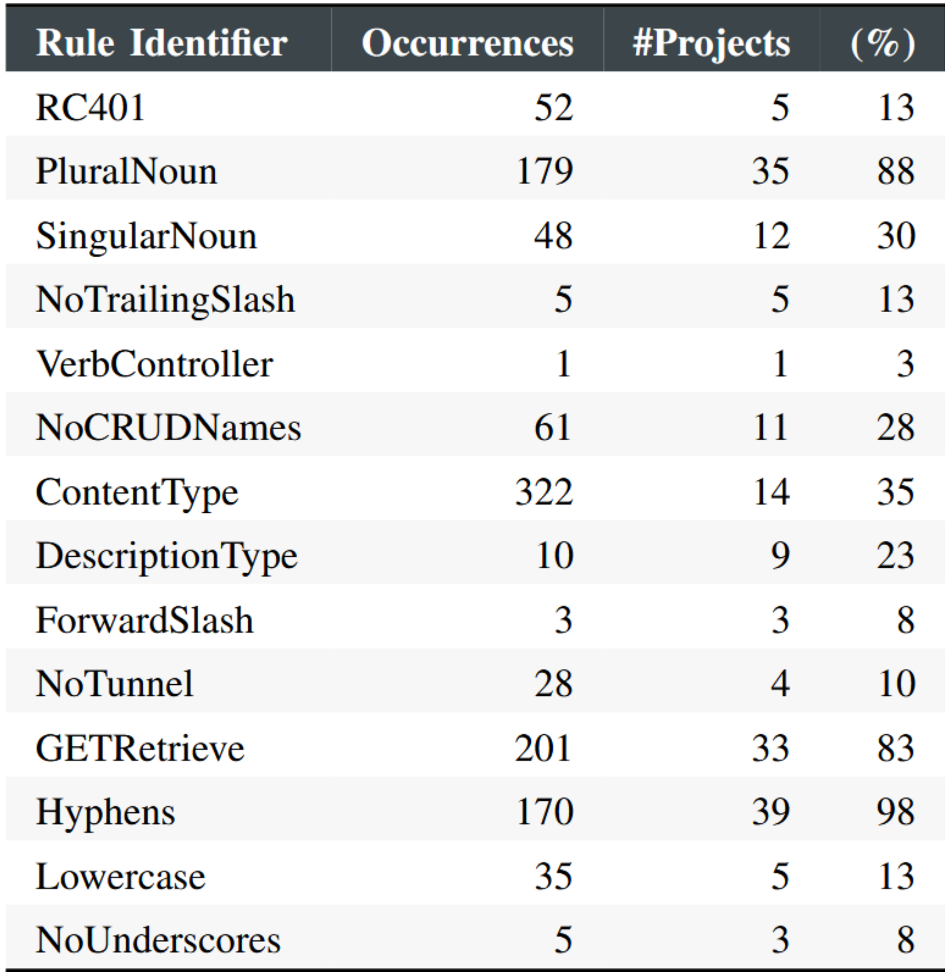
- Automatically searched for violations of the following rules in last year’s web tech projects



Rule Description	Identifier	Category
`401 ("Unauthorized")` must be used when there is a problem with the client’s credential	RC401	HTTP Status Codes
A plural noun should be used for collection or store names	PluralNoun	URI Design
A singular noun should be used for document names	SingularNoun	URI Design
A trailing forward slash (/) should not be included in URIs	NoTrailingSlash	URI Design
A verb or verb phrase should be used for controller names	VerbController	URI Design
CRUD function names should not be used in URIs	NoCRUDNames	URI Design
Content-Type must be used	ContentType	Metadata Design
Description of request should match with the type of the request	DescriptionType	Metadata Design
Forward slash separator (/) must be used to indicate a hierarchical relationship	ForwardSlash	URI Design
`GET` and `POST` must not be used to tunnel other request methods	NoTunnel	Request Methods
`GET` must be used to retrieve a representation of a resource	GETRetrieve	Request Methods
Hyphens (-) should be used to improve the readability of URIs	Hyphens	URI Design
Lowercase letters should be preferred in URI paths	Lowercase	URI Design
Underscores (_) should not be used in URI	NoUnserscores	URI Design

REST IN PIECES (LAST YEAR)

- Preliminary results highlighted a significant number of violations
- Some violations more prevalent than others
  - Hyphens rarely used
  - Plural/singular nouns rules often ignored
  - HTTP methods used incorrectly
  - CRUD names in URIs…
- **Let’s do better this year!** 💪



Rule Identifier	Occurrences	#Projects	(%)
RC401	52	5	13
PluralNoun	179	35	88
SingularNoun	48	12	30
NoTrailingSlash	5	5	13
VerbController	1	1	3
NoCRUDNames	61	11	28
ContentType	322	14	35
DescriptionType	10	9	23
ForwardSlash	3	3	8
NoTunnel	28	4	10
GETRetrieve	201	33	83
Hyphens	170	39	98
Lowercase	35	5	13
NoUnderscores	5	3	8

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/
- **REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces**
  By Mark Massé

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

⚠️ You are **not** required to learn JakartaEE, Spring, or FastAPI for the Web Technologies course. It might be educationally worthwhile to take a look at some web frameworks other than Express. JakartaEE and Spring, for example, use an annotation-based approach. So, feel free to look at the code or try to run the above linked REST API, but be aware that it's not required for this course.