From Class Diagram to API Server

Software Engineering - Tutorial

Antonio Bucchiarone - bucchiarone@fbk.eu

Academic year 2022/2023 - First semester

Overview

> One of the critical steps in that process is to go from a Class Diagram to implement an API server.

> In this lecture we see all the steps to follow from the Class Diagram to the API Specification and implementation

How to Design API with UML?

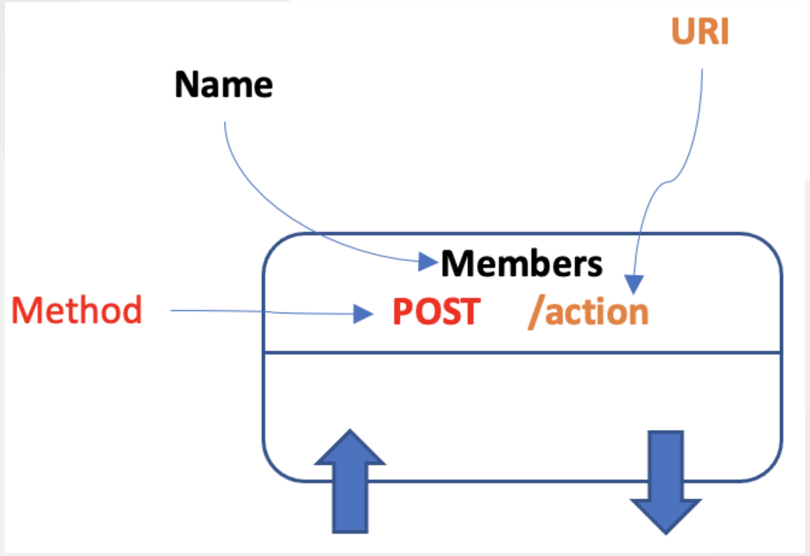
- You can design your API by drawing a **class diagram** that represents your resource, the request and response body.

API Resources

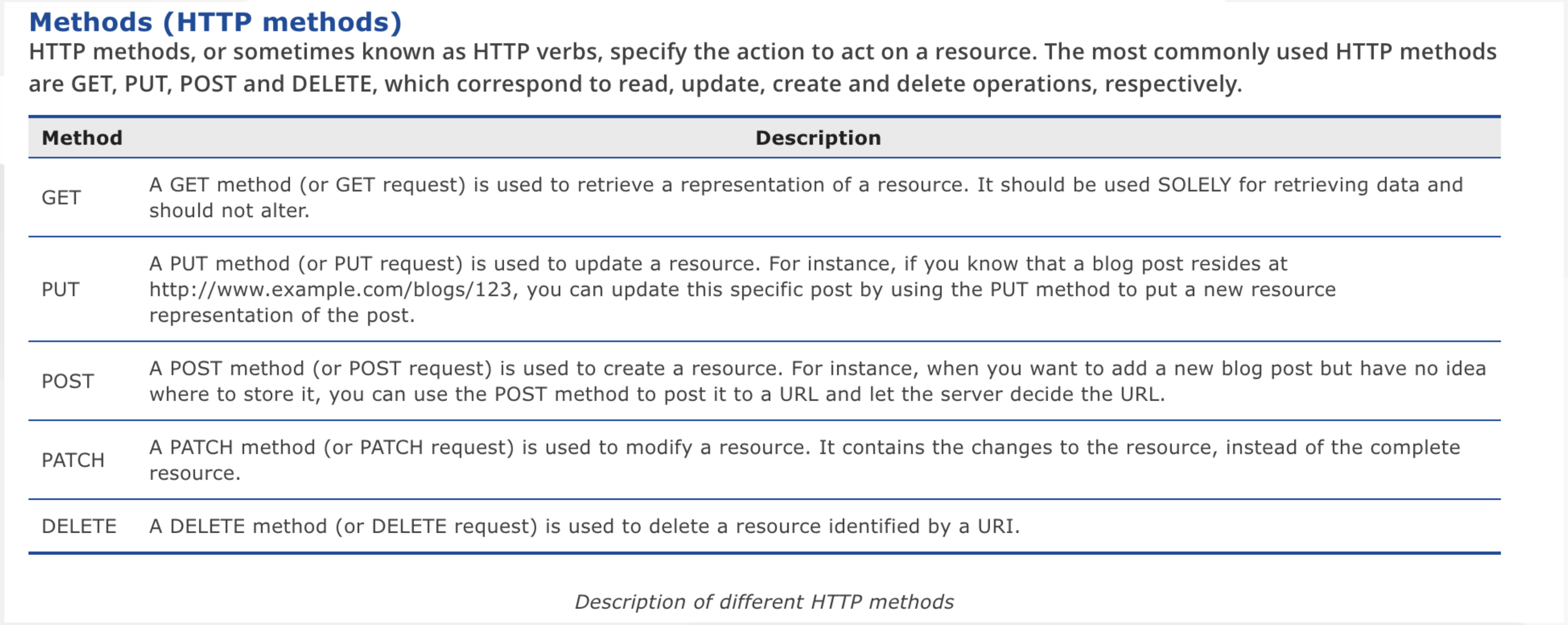
- An API **resource** is the fundamental unit of an API.
- It is an **object** with a URI, the **http** request **method**, **associated parameters** and the **request/response** body.
- Each of the API resources represents a specific service available on the path specified by its URI property.
- Therefore, if you want to model multiple services, please draw multiple API resources.

API Resource Model

- **URI:** Each Resource has its own URI. Consumers access to URL to access for the API Resource.
- **Method:** Specifies the action to act on the resource.
- **Name:** Name of the specified resource



API Resource Methods



Methods (HTTP methods)

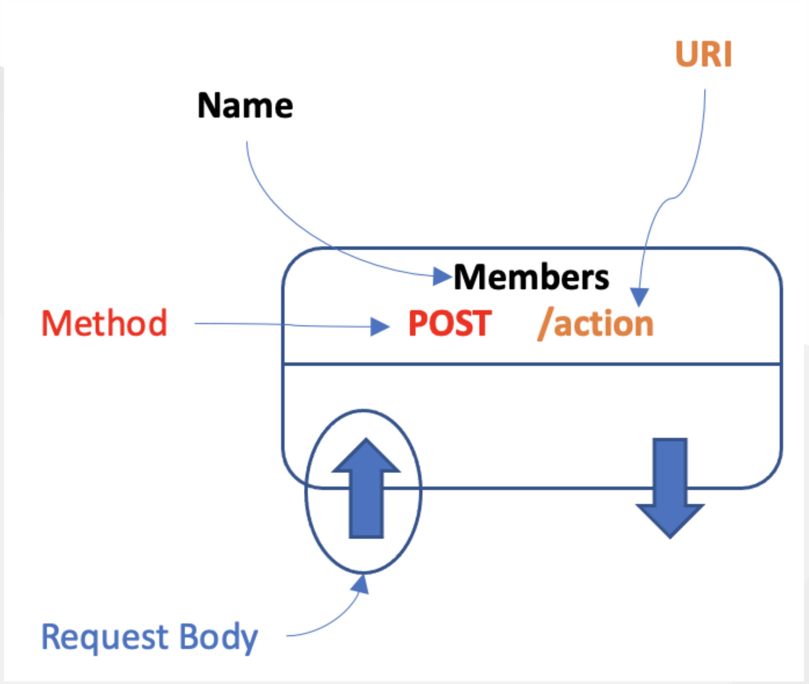
HTTP methods, or sometimes known as HTTP verbs, specify the action to act on a resource. The most commonly used HTTP methods are GET, PUT, POST and DELETE, which correspond to read, update, create and delete operations, respectively.

Method	Description
GET	A GET method (or GET request) is used to retrieve a representation of a resource. It should be used SOLELY for retrieving data and should not alter.
PUT	A PUT method (or PUT request) is used to update a resource. For instance, if you know that a blog post resides at http://www.example.com/blogs/123, you can update this specific post by using the PUT method to put a new resource representation of the post.
POST	A POST method (or POST request) is used to create a resource. For instance, when you want to add a new blog post but have no idea where to store it, you can use the POST method to post it to a URL and let the server decide the URL.
PATCH	A PATCH method (or PATCH request) is used to modify a resource. It contains the changes to the resource, instead of the complete resource.
DELETE	A DELETE method (or DELETE request) is used to delete a resource identified by a URI.

Description of different HTTP methods

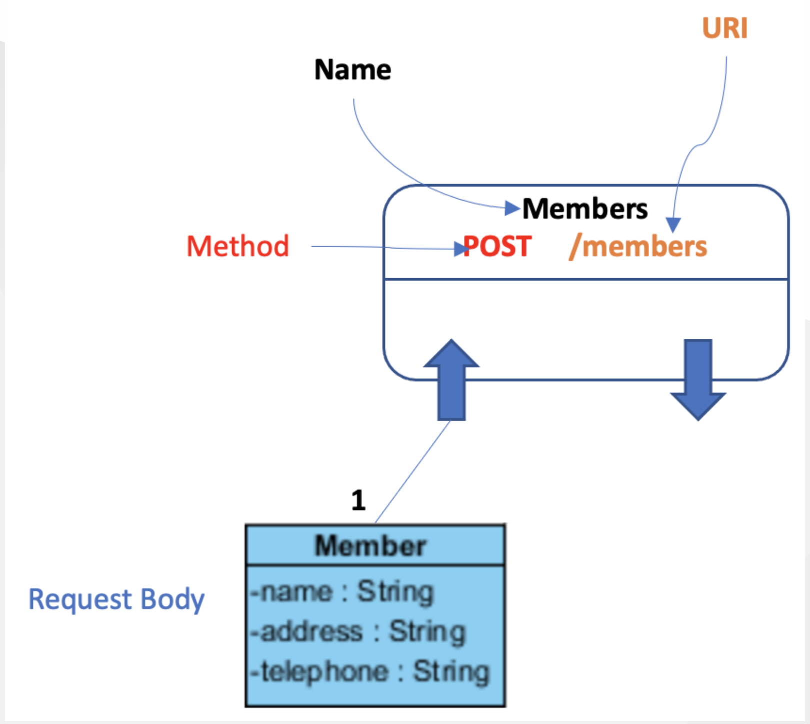
API Request Body

> If the Resource uses a **POST**, **PUT**, **PATCH** or **DELETE** method and if **parameter** is required in using the Resource, model the parameters by drawing UML class(es).



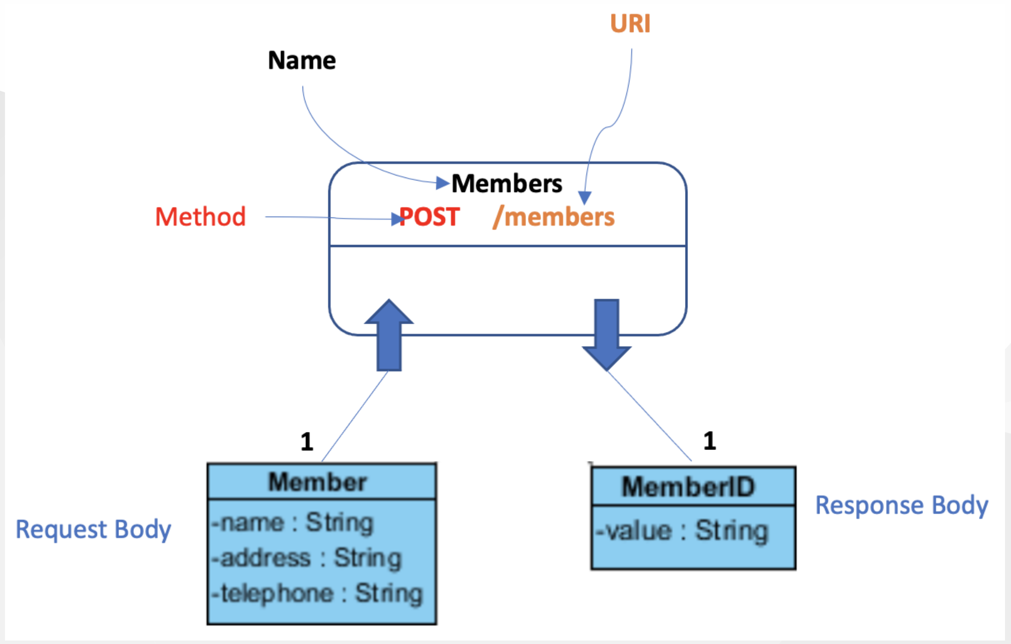
API Request Body Class

> For instance, if we are going to create a member via the **/members** Resource, we need to send the **member's** details to the server for creating a member record. Therefore, name the class **Member** for storing member's details.



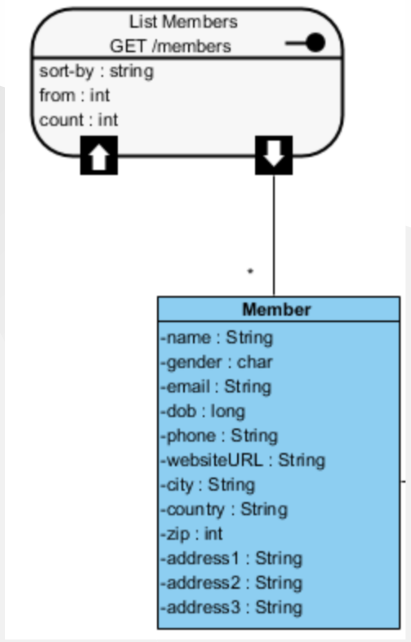
API Response

> If the API will return a value or object, a Class Association must be defined.



API GET Parameters

- **Parameters** are referring to query parameters used for passing data to an API.



Modeling multiple scenarios

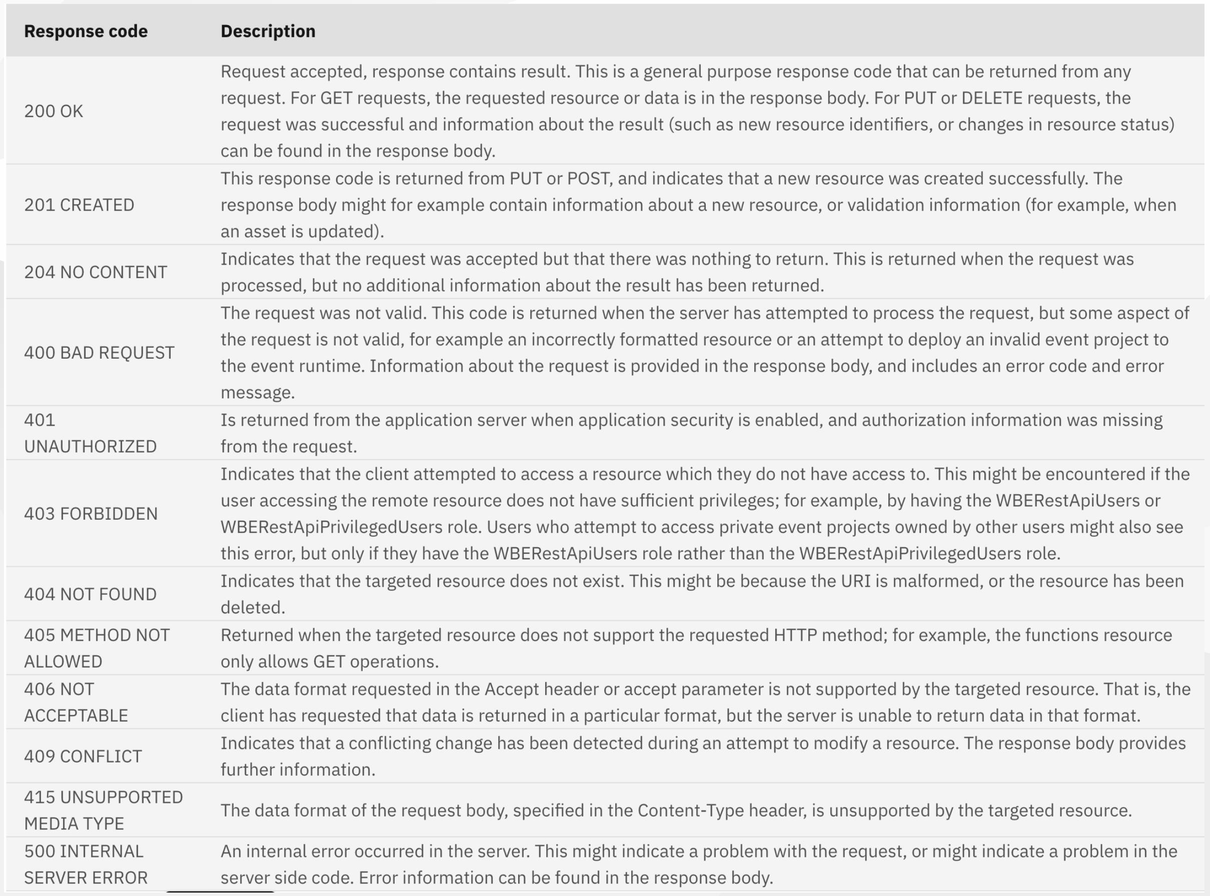
- Sometimes, you may need to model multiple scenarios where there could be **multiple** or **different** Response Bodies.
- For example, you want to define the various **HTTP status codes** that can be returned as well as in some cases you may be returning an Error object embedded within the main response object.

Examples

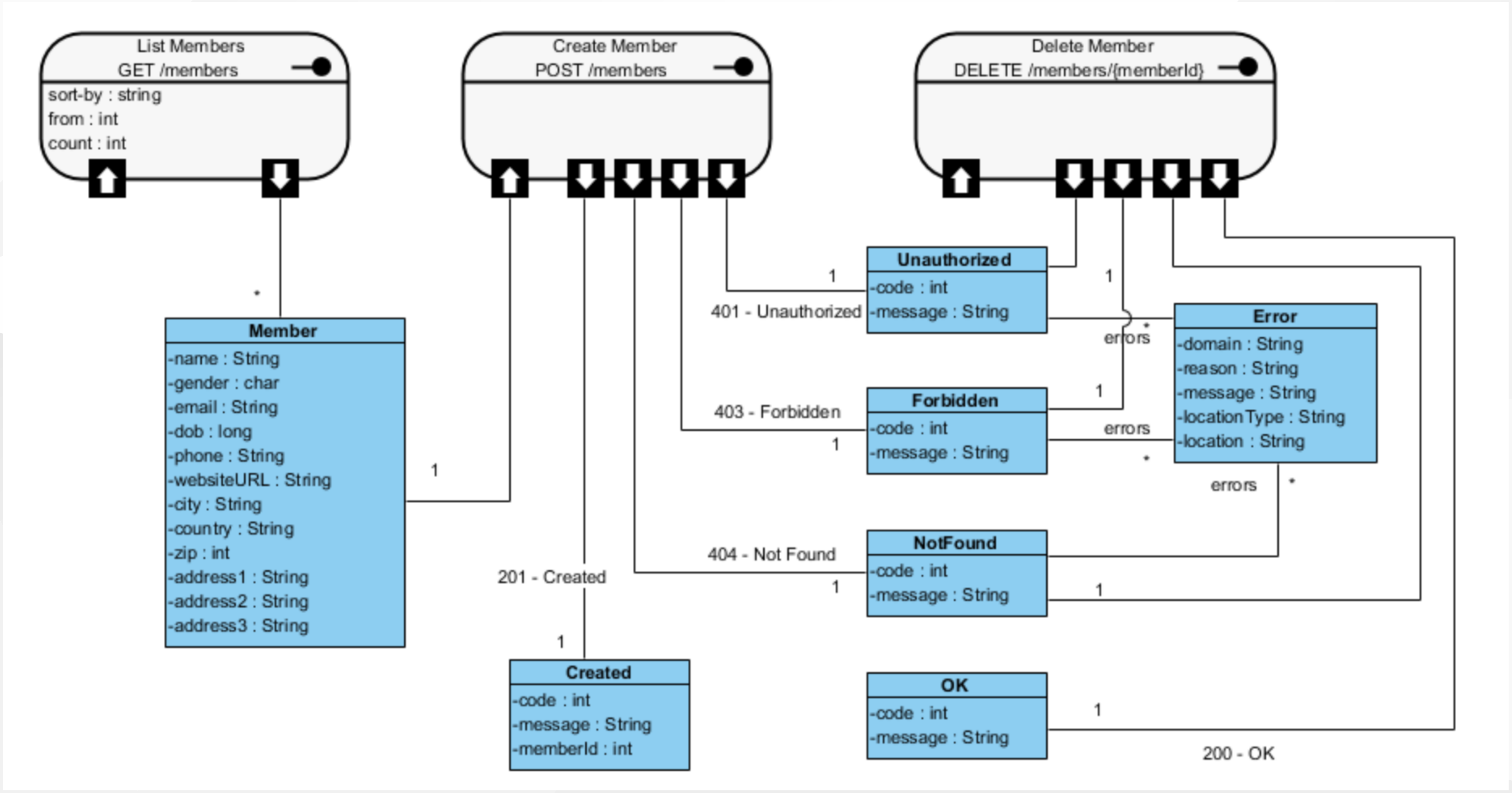
```
Case 1:

Response Header:
status : 200 OK

Response Body:
{
   "customer" : {
   "name" : "Peter",
   }
}
```

```
Case 2:

Response Header:
status : 400 Bad Request

Response Body:
{
   "customer": {
      "error" : {
      "text" : "Invalid customer name."
   }
}
```

> The HTTP standard RFC 2616 is a primary source of information for the meaning of error codes.



Response code	Description
200 OK	Request accepted, response contains result. This is a general purpose response code that can be returned from any request. For GET requests, the requested resource or data is in the response body. For PUT or DELETE requests, the request was successful and information about the result (such as new resource identifiers, or changes in resource status) can be found in the response body.
201 CREATED	This response code is returned from PUT or POST, and indicates that a new resource was created successfully. The response body might for example contain information about a new resource, or validation information (for example, when an asset is updated).
204 NO CONTENT	Indicates that the request was accepted but that there was nothing to return. This is returned when the request was processed, but no additional information about the result has been returned.
400 BAD REQUEST	The request was not valid. This code is returned when the server has attempted to process the request, but some aspect of the request is not valid, for example an incorrectly formatted resource or an attempt to deploy an invalid event project to the event runtime. Information about the request is provided in the response body, and includes an error code and error message.
401 UNAUTHORIZED	Is returned from the application server when application security is enabled, and authorization information was missing from the request.
403 FORBIDDEN	Indicates that the client attempted to access a resource which they do not have access to. This might be encountered if the user accessing the remote resource does not have sufficient privileges; for example, by having the WBERestApiUsers or WBERestApiPrivilegedUsers role. Users who attempt to access private event projects owned by other users might also see this error, but only if they have the WBERestApiUsers role rather than the WBERestApiPrivilegedUsers role.
404 NOT FOUND	Indicates that the targeted resource does not exist. This might be because the URI is malformed, or the resource has been deleted.
405 METHOD NOT ALLOWED	Returned when the targeted resource does not support the requested HTTP method; for example, the functions resource only allows GET operations.
406 NOT ACCEPTABLE	The data format requested in the Accept header or accept parameter is not supported by the targeted resource. That is, the client has requested that data is returned in a particular format, but the server is unable to return data in that format.
409 CONFLICT	Indicates that a conflicting change has been detected during an attempt to modify a resource. The response body provides further information.
415 UNSUPPORTED MEDIA TYPE	The data format of the request body, specified in the Content-Type header, is unsupported by the targeted resource.
500 INTERNAL SERVER ERROR	An internal error occurred in the server. This might indicate a problem with the request, or might indicate a problem in the server side code. Error information can be found in the response body.


List Members
GET /members
sort-by : string
from : int
count : int
Create Member
POST /members
Delete Member
DELETE /members/{memberId}
Member
-name : String
-gender : char
-email : String
-dob : long
-phone : String
-websiteURL : String
-city : String
-country : String
-zip : int
-address1 : String
-address2 : String
-address3 : String
*
1
201 - Created
1
Created
-code : int
-message : String
-memberId : int
401 - Unauthorized
1
Unauthorized
-code : int
-message : String
403 - Forbidden
1
Forbidden
-code : int
-message : String
404 - Not Found
1
NotFound
-code : int
-message : String
OK
-code : int
-message : String
1
200 - OK
Error
-domain : String
-reason : String
-message : String
-locationType : String
-location : String
errors
*
errors
*
errors
*
1
1
1

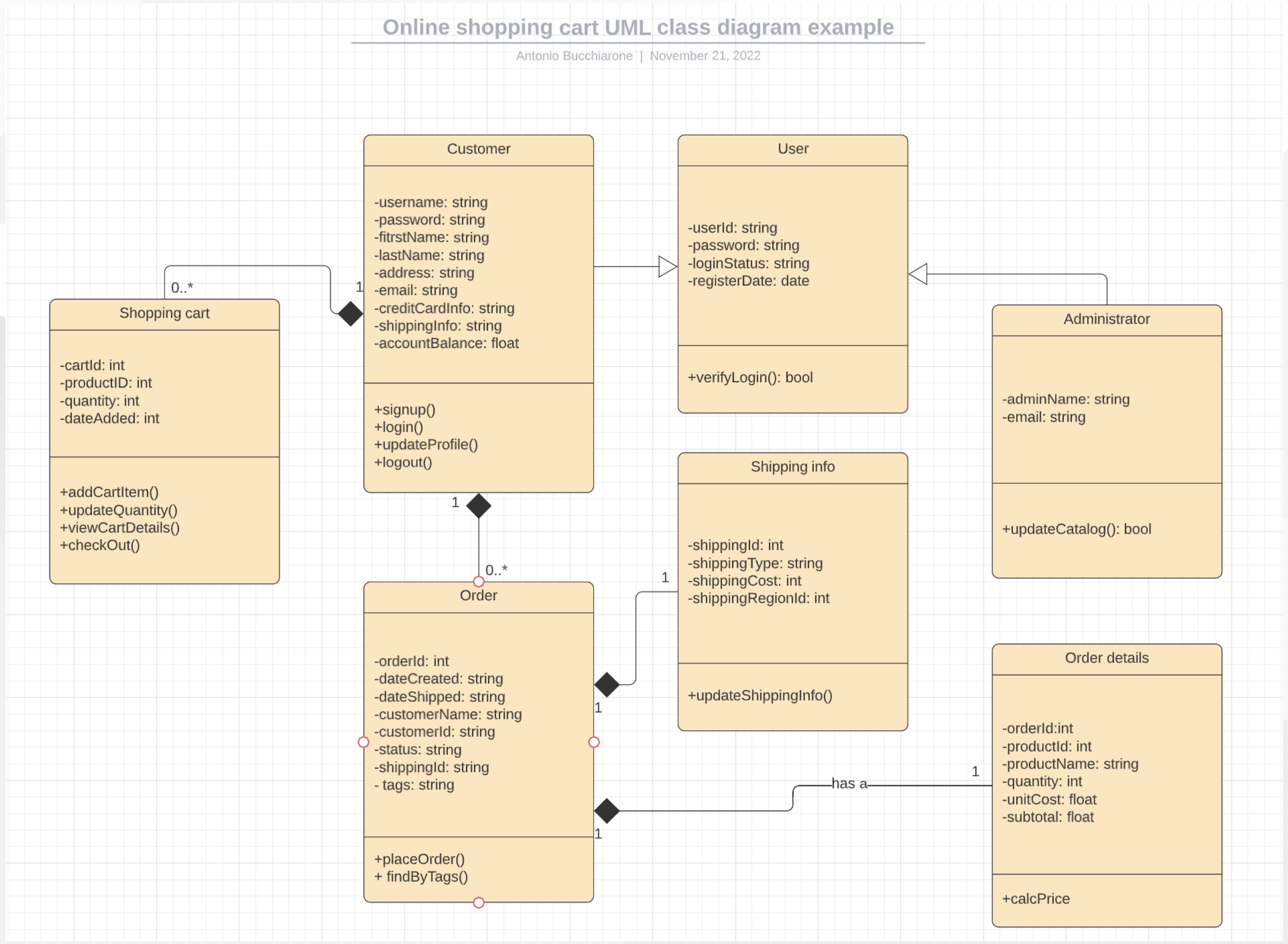
For Deliverable D4

1. APIs Specification from Class Diagram
2. APIs Implementation and Documentation
3. APIs Testing
4. FrontEnd Implementation
5. Application Deployment

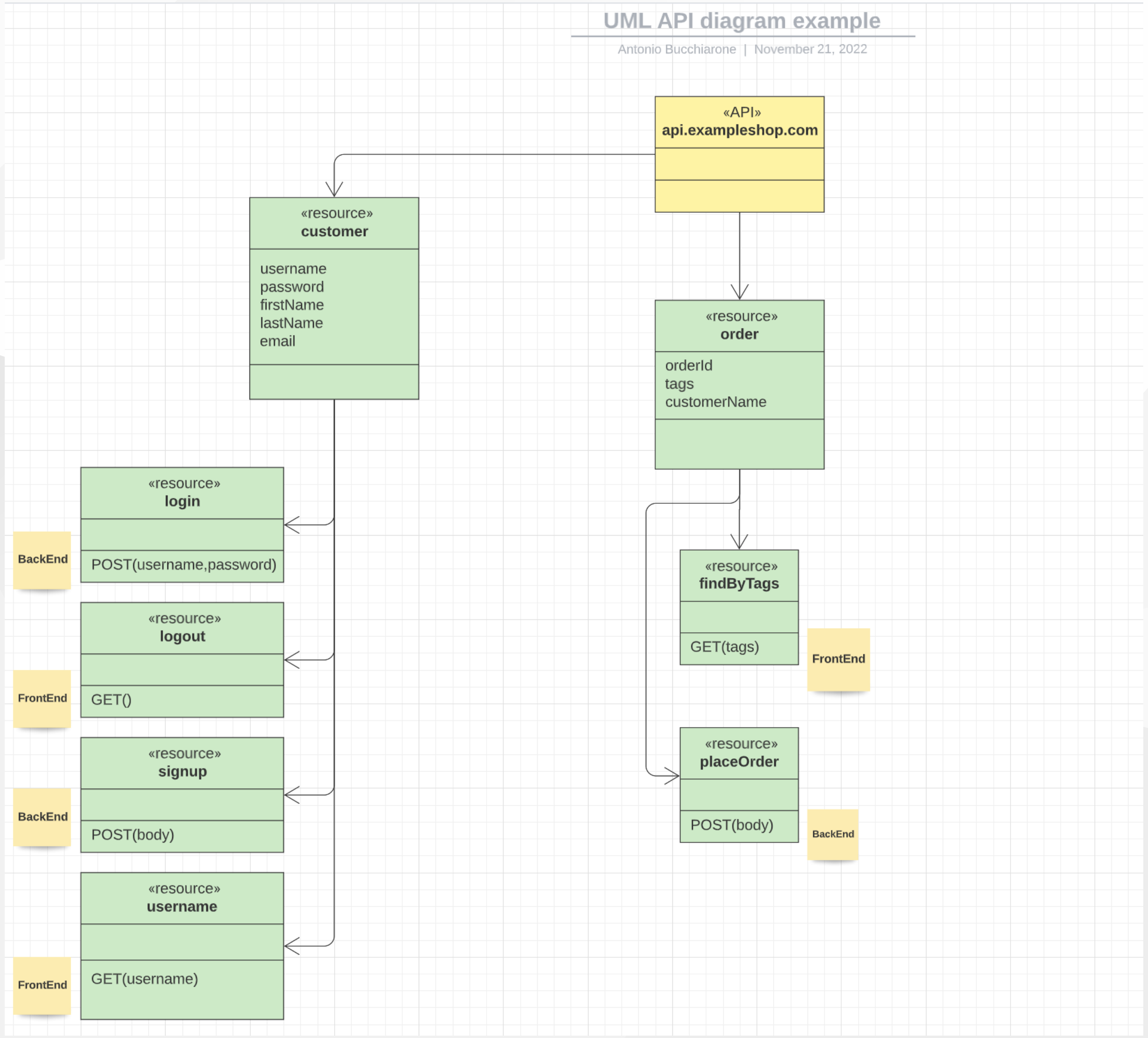
APIs Specification from Class Diagram

1. Resources Extraction from the Application Class Diagrams
2. Resources Models

Class Diagram Example



Resources Extraction

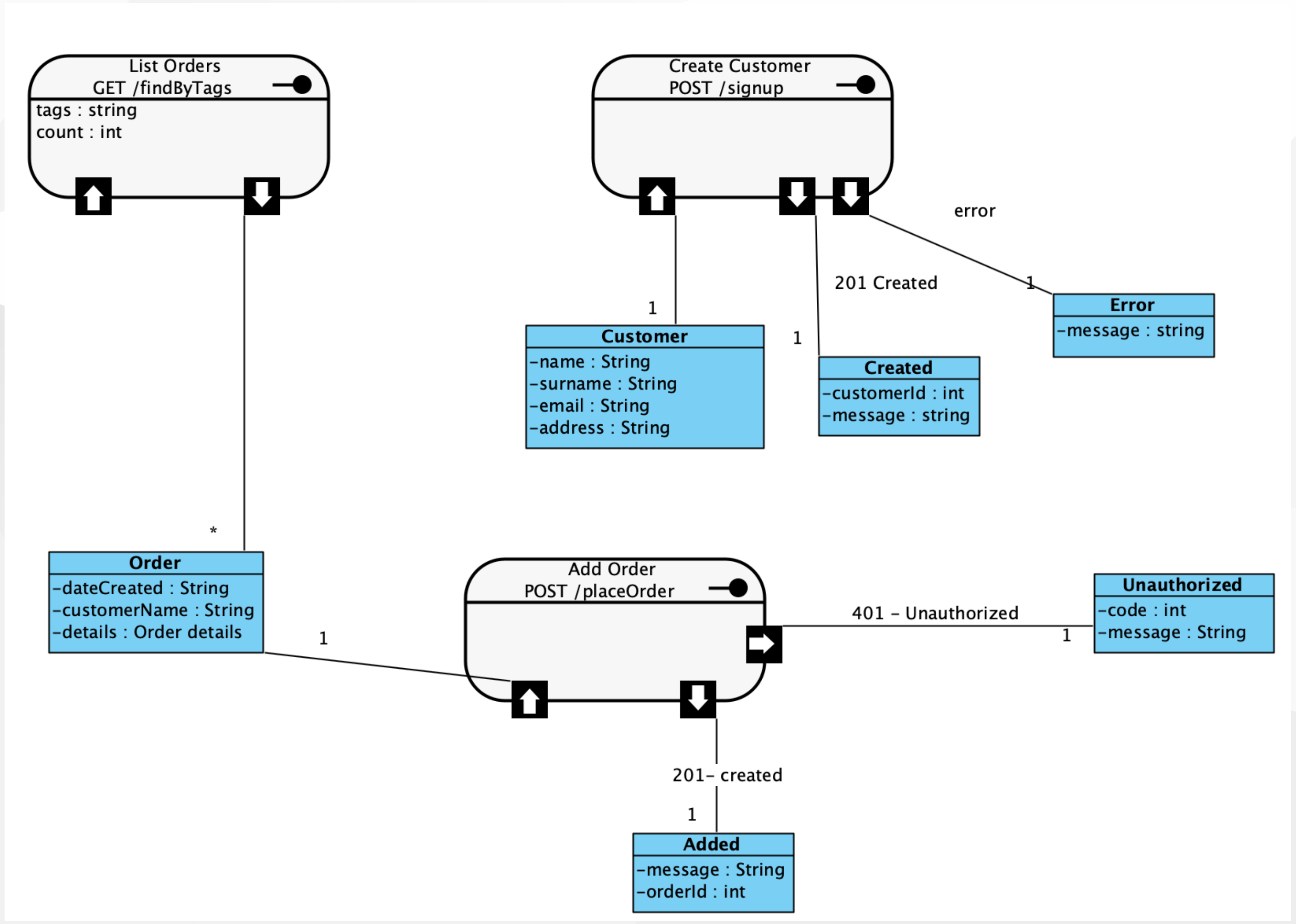


Exercise (15-20 min)

> Is your turn. Starting from your class diagram try to extract:

```
1. At least 2 main resources (i.e., customer, order) and their attributes
2. For each main resource at least one POST and one GET API
3. For each API define which has an effect in the Backend and in the Frontend
```

Resources Models



Exercise (20 min)

> Is your turn. Derive your resource model with at least one GET and one POST API

```
1. For each API identify the **Request** and **Response** bodies
2. Identify the needed parameters for the GET APIs
3. Identify the needed messages
```

API "signup" Implementation

1. Create the **Customer** Model for the DB
2. Create the **Customer** Route with the API endpoint
3. Create the **Customer** Controller for the API implementation

Customer Data Model - API Data Model

- exploiting the **Class Attributes** of the **customer** Class

```
const mongoose = require("mongoose"); //import mongoose

// customer schema
const CustomerSchema = new mongoose.Schema({
    name: String,
    surname: String,
    email: String,
    address: String
});

const Customer = mongoose.model('Customer', CustomerSchema); //convert to model named Customer
module.exports = Customer; //export for controller use
```

Customer Router - API Endpoint Definition

```
const express = require('express'); //import express

// 1.
const router = express.Router();

const authController = require('../controllers/auth.controller');


router.post('/auth/signup', authController.signup);

// 4.
module.exports = router; // export to use in server.js
```

Customer Controller - API behavior implementation

> Create Controllers

- Controller for Registration

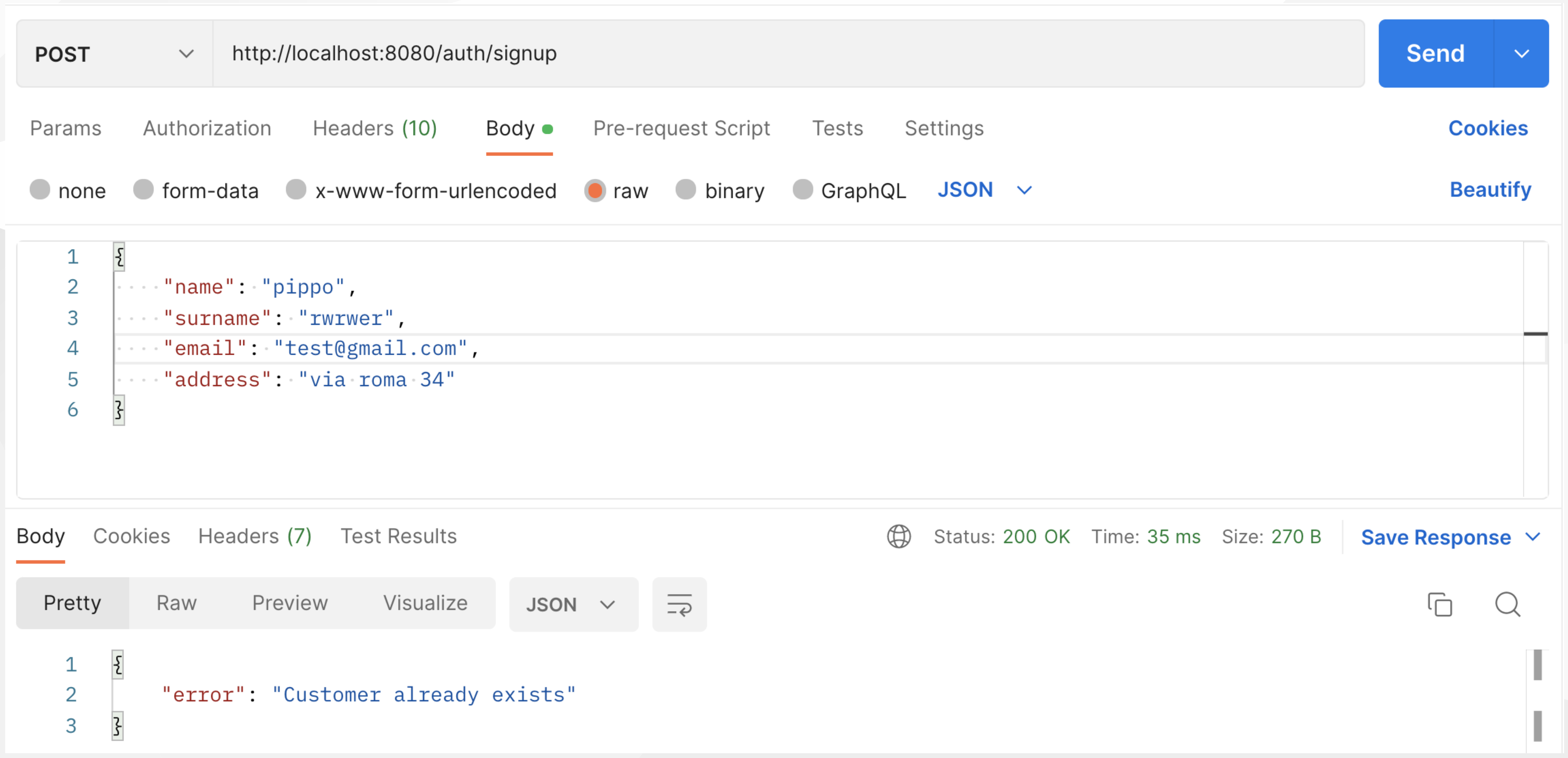
There is 1 main function for the registration:

- signup: create new Customer in database

API code to add a new Customer

```
if (!customer) {
        const newCustomer = new Customer({
            name: req.body.name,
            surname: req.body.surname,
            email: req.body.email,
            address: req.body.address,
        })


        // save this object to database
        newCustomer.save((err, data) => {
            if (err) return res.json({ Error: err });
            return res.json({ message: "Customer Created", data });
        })

    }
```

API Testing via Postman



http://localhost:8080/auth/signup

```
{
    "name": "pippo",
    "surname": "rwrwer",
    "email": "test@gmail.com",
    "address": "via roma 34"
}
```

Status: 200 OK Time: 35 ms Size: 270 B

```
{
    "error": "Customer already exists"
}
```

Questions?

bucchiarone@fbk.eu