# Building RESTful API

Marek Konieczny, Robert Straś

marekko@agh.edu.pl,

Room 4.43, Spring 2025

AGH

Akademia Górniczo-Hutnicza
im. Stanisława Staszica
w Krakowie

# Class Logistics

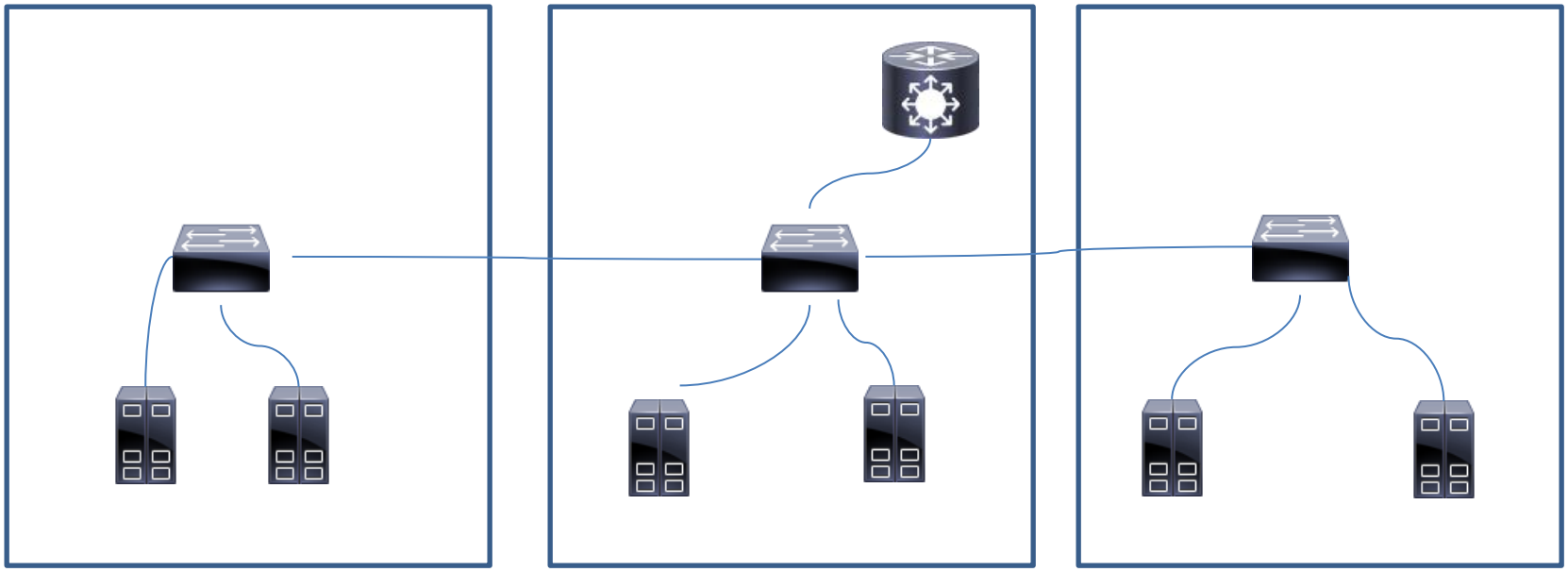
- We will meet on:
  - 10-14.3 laboratory sessions
  - 21-24.3 homework
- All materials on UPEL: including exam part
- Grading:
  - Showing up => +1
  - Hard work => +2
  - Upload results to UPEL => +1
  - Extra activity => +1

# Environment preparation

- You can use your laptops
  - You will need python
- Log in to desktops
  - Select Ubuntu image

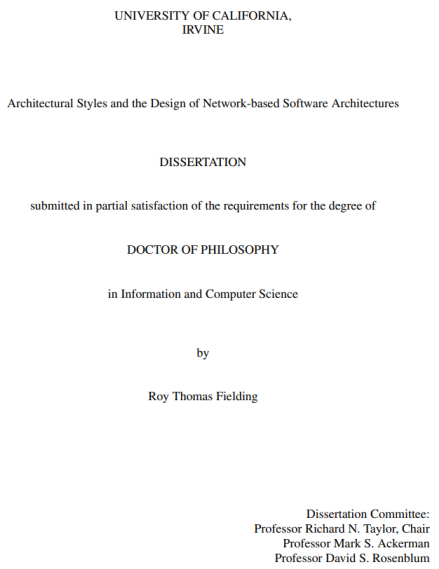
# Environment preparation

- Wire-up all environment to have internet connection



# Origin

- **Re**presentational **S**tate **T**ransfer
- Architectural style
  - not dependent on any specific protocol
- Describes a set of principles derived from analysis of WWW Architecture
  - To make any distributed system scalable
- Originally WWW was created by team led by Timothy Berners-Lee at CERN
  - In 2016 ACM Turing Award



UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

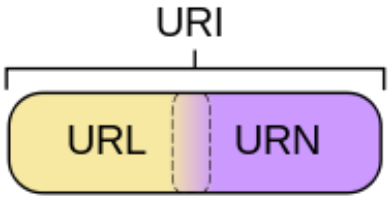
Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

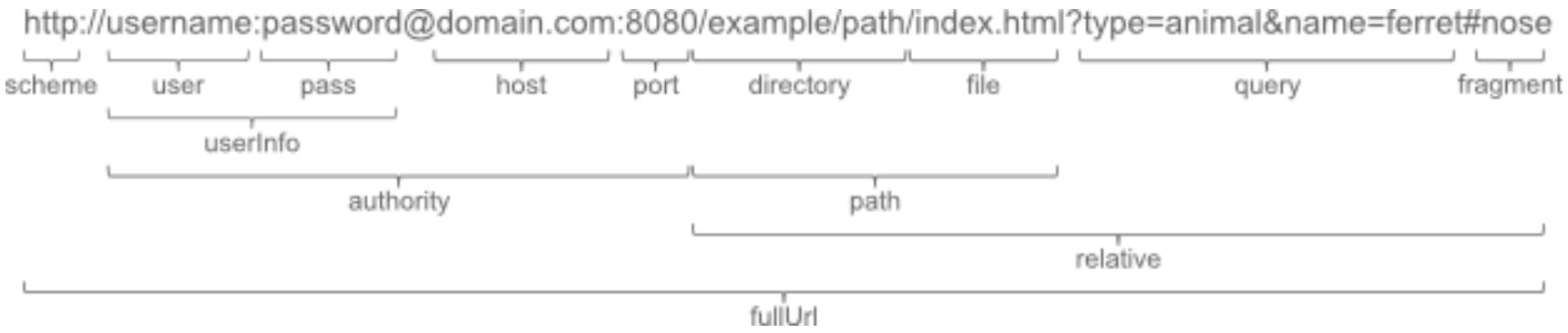
# Basics

- Resource
  - fundamental building block of web-based systems
- Web is often named „resource-oriented”
- Resource is anything with which consumer interacts while achieving some goal
  - document, video, business process, device, spreadsheet, printer
- Exposition of resource to Web:
  - Abstracting out resource information aspects
  - Presenting these aspects to digital world by means of some representation

# Uniform Resource Identification (URI)

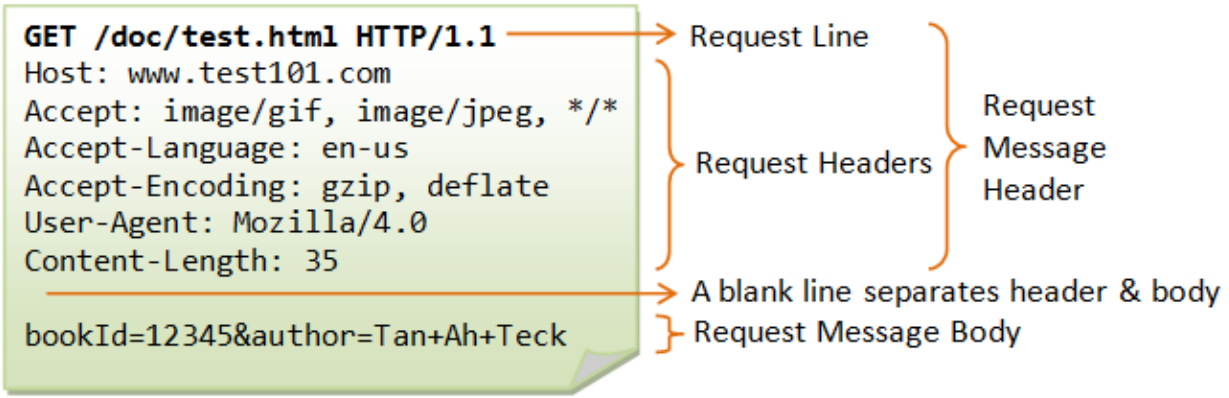



- URI or URL?
  - URI = URL || URN
  - URN is not so popular (urn:oasis:names:specification:docbook:dtd:xml:4.1.2)
  - Usually URI = Uniform Resource Locator (URL)
- Different types : File URL, FTP URL, HTTP URL



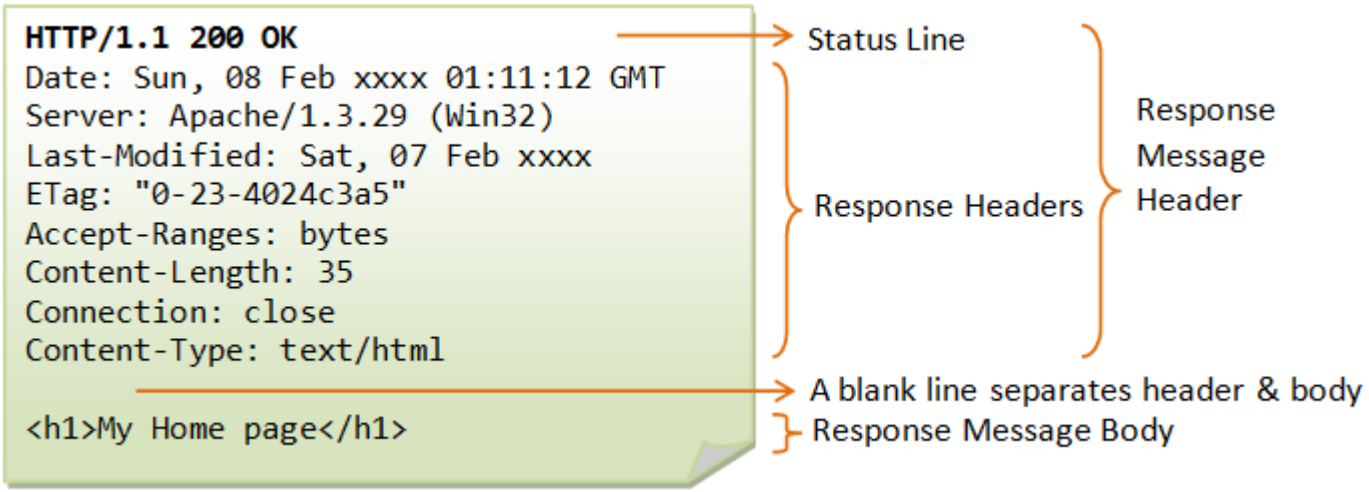
# HyperText Transfer Protocol (HTTP)

- Protocol for distributed systems for sharing media
- Hypermedia: logic extension of hypertext which includes graphics, audio, video which creates new media
- Based on the request-response schema
  - Client send requests to server
- Format
  - Method line
  - Headers
  - Empty line
  - Optional body



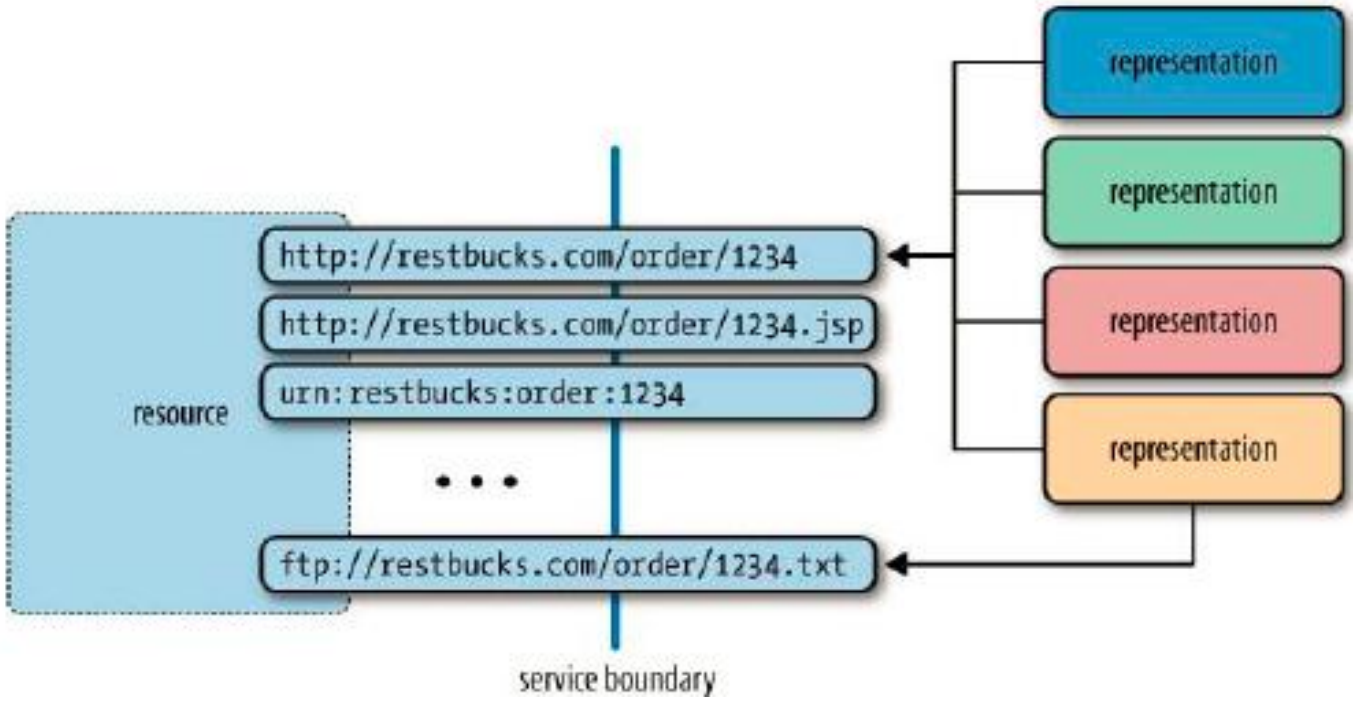
# HyperText Transfer Protocol (HTTP)

- First line contains code of the response
  - Succes: 2xx
  - Redirections: 3xx
  - Error of client: 4xx
  - Server error: 5xx
- Later headers, and body

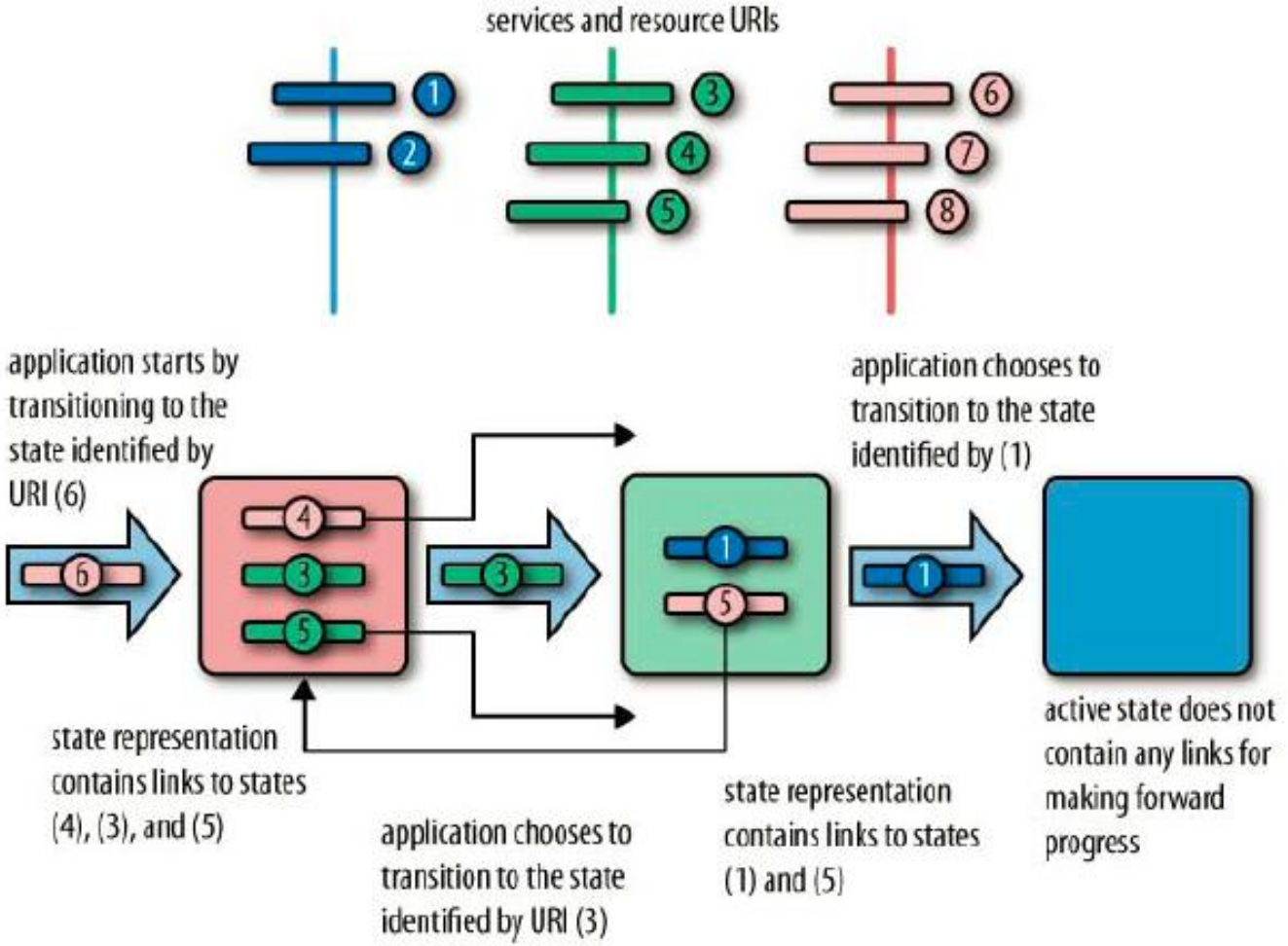



# Services

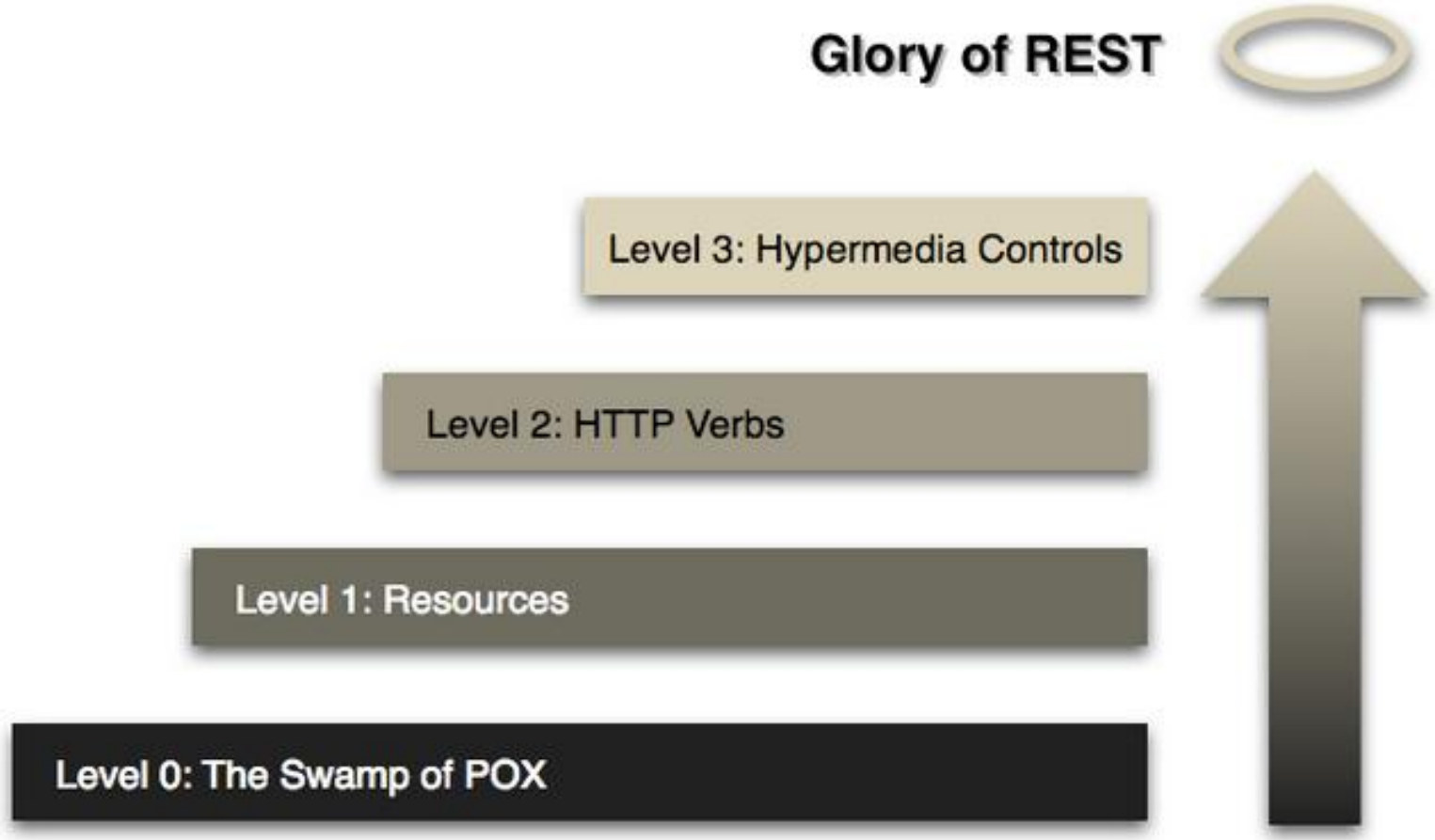
- Entire state of system is exposed as resources



# Services

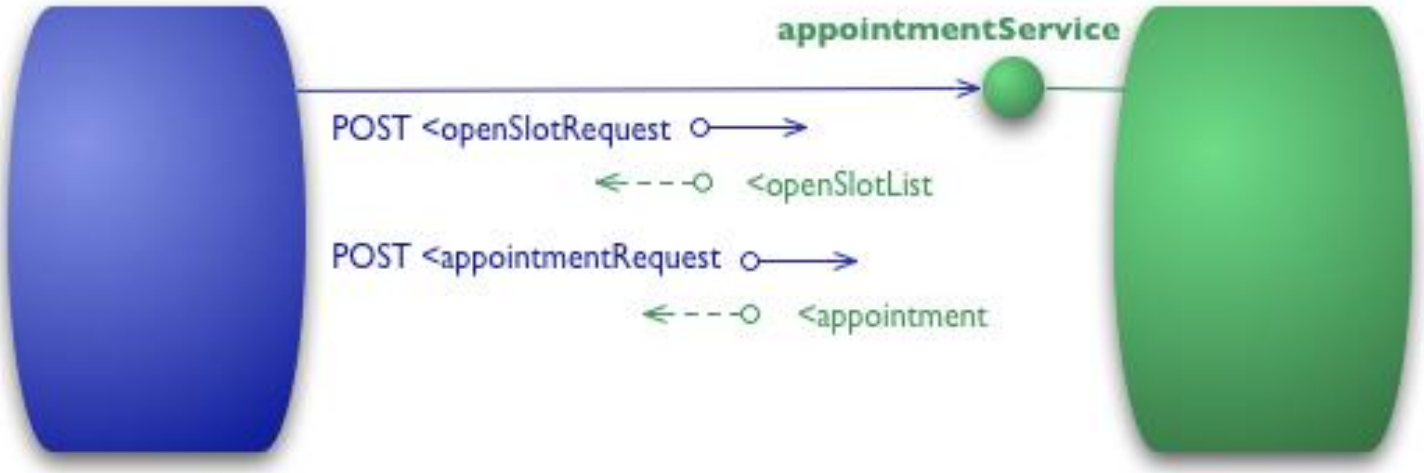

services and resource URIs
1
2
3
4
5
6
7
8
application starts by transitioning to the state identified by URI (6)
application chooses to transition to the state identified by (1)
state representation contains links to states (4), (3), and (5)
application chooses to transition to the state identified by URI (3)
state representation contains links to states (1) and (5)
active state does not contain any links for making forward progress

# RESTful Maturity Model



# Level 0

```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```



# Level 0

```
HTTP/1.1 200 OK
[various headers]

<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

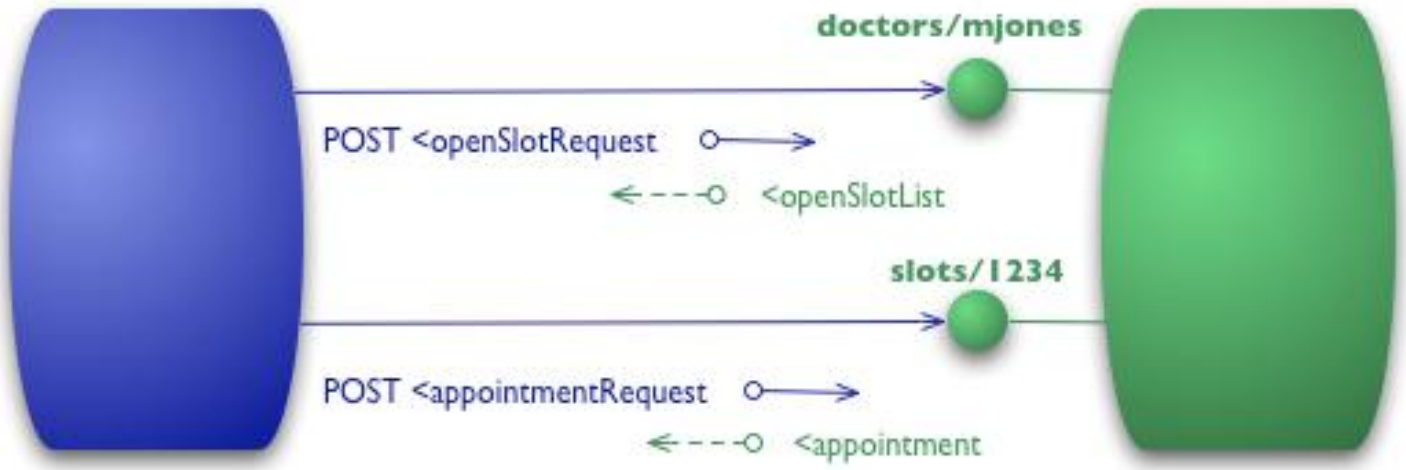
```
HTTP/1.1 200 OK
[various headers]

<appointmentRequestFailure>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>
```

# Level 1

```
POST /doctors/mjones HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04"/>
```

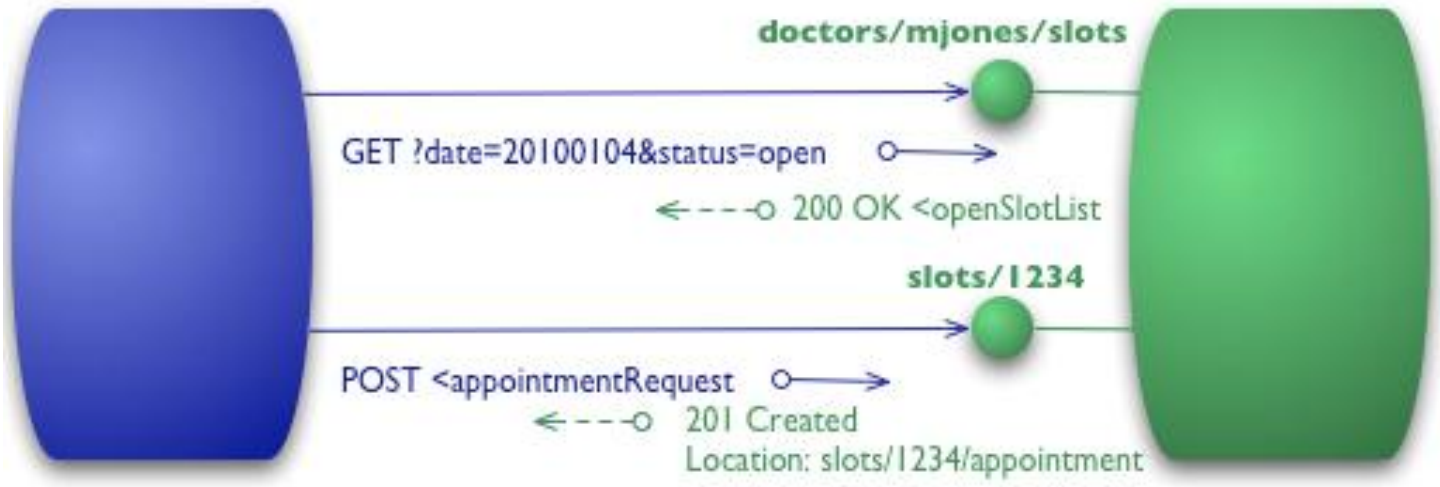
```
HTTP/1.1 200 OK
[various headers]
```

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```



# Level 2

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```



# Level 2

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

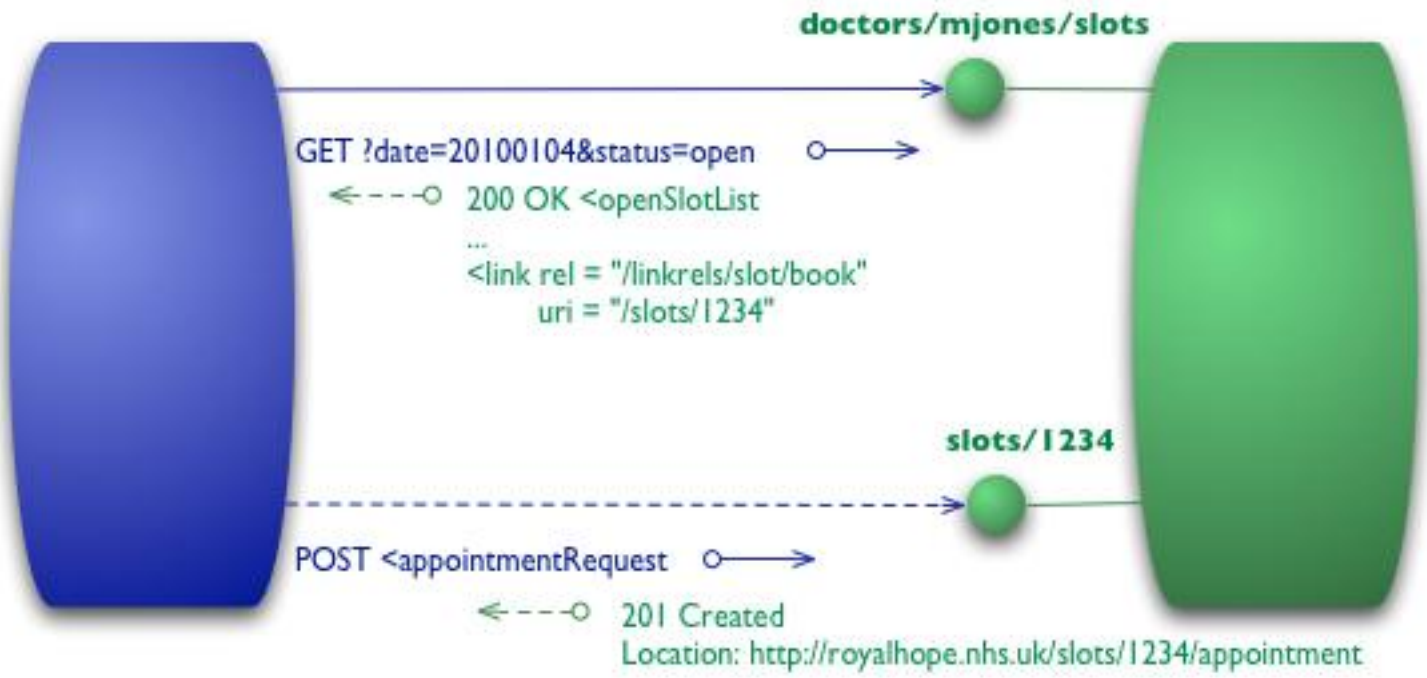
```
HTTP/1.1 409 Conflict
[various headers]

<openSlotList>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

# HTTP Methods as Verbs

| HTTP Verb | Common Meaning | Safe | Idempotent |
|---|---|---|---|
| GET | Retrieve the current state of the resource | YES | YES |
| POST | Create a sub resource | NO | NO |
| PUT | Initialize or update the state of a resource at given URI | NO | YES |
| DELETE | Clear a resource | NO | YES |
| HEAD | Retrieve the current metadata of the resource | YES | YES |

# Level 3

- Hypermedia as the Engine of Application State (HATEOAS)



# Level 3

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104@status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

# REST Design Methodology

1. Identify resources to be exposed as services
2. Model relationships between resources with hyperlinks
3. Define URIs to address the resources
4. Understand what it means to do a GET, POST, PUT, DELETE for each resource
5. Design and document resource representation
6. Implement and deploy on Web server
7. Test with a Web browser

| | GET | PUT | POST | DELETE |
|---|---|---|---|---|
| /loan | ✓ | ✓ | ✓ | ✓ |
| /balance | ✓ | ✗ | ✗ | ✗ |
| /client | ✓ | ✓ | ✓ | ✗ |
| /book | ✓ | ✓ | ✓ | ✓ |
| /order | ✓ | ? | ✓ | ✗ |
| | | | | |
| /soap | ✗ | ✗ | ✓ | ✗ |

# Task 0 – environment setup

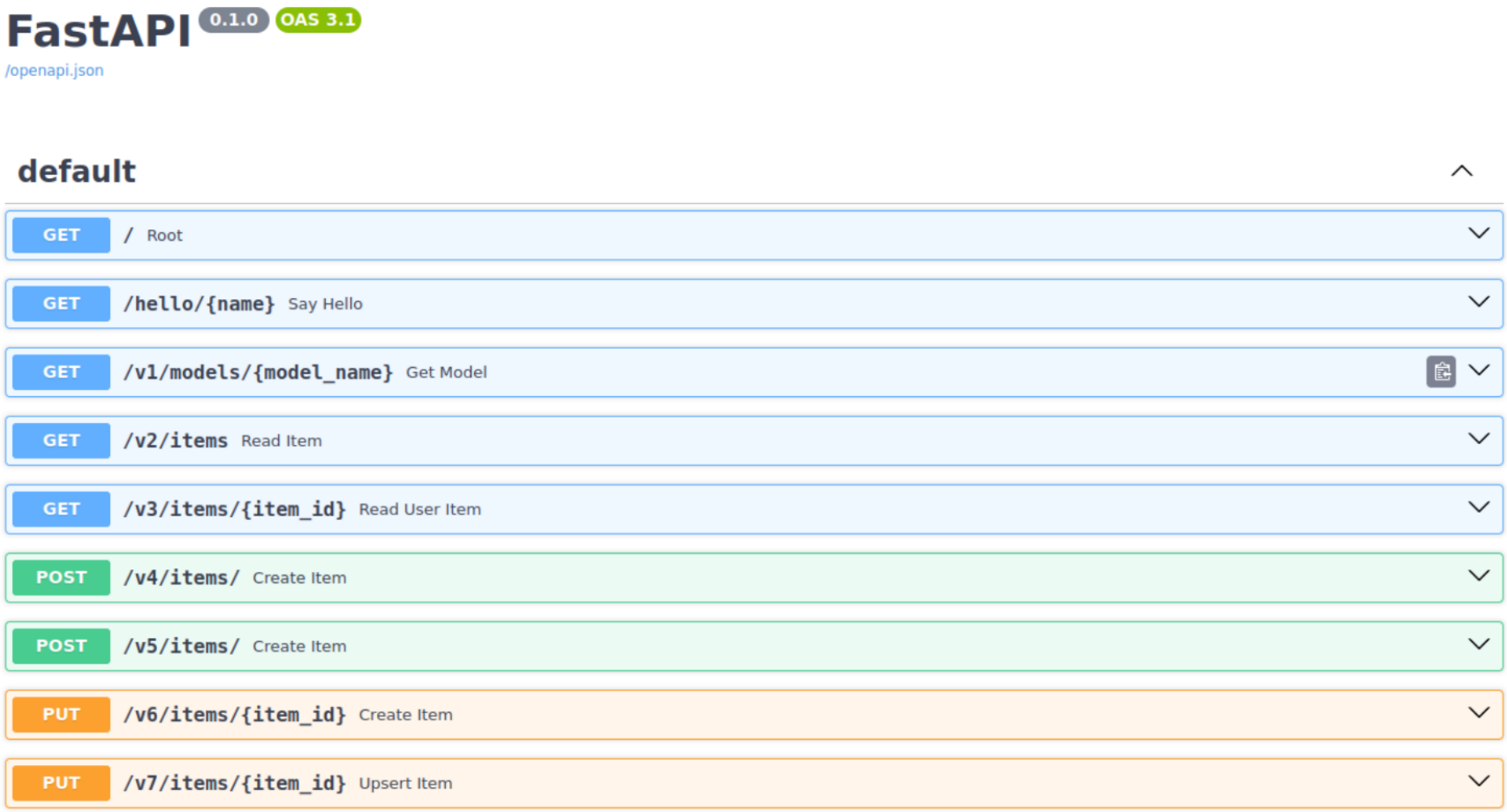
- Open attached distributed.py file
- You will need following packages:
  - pip3 install fastapi
  - pip3 install uvicorn
- Start web server inside the file location
  - uvicorn distributed:app --reload

# Task 0 – environment setup

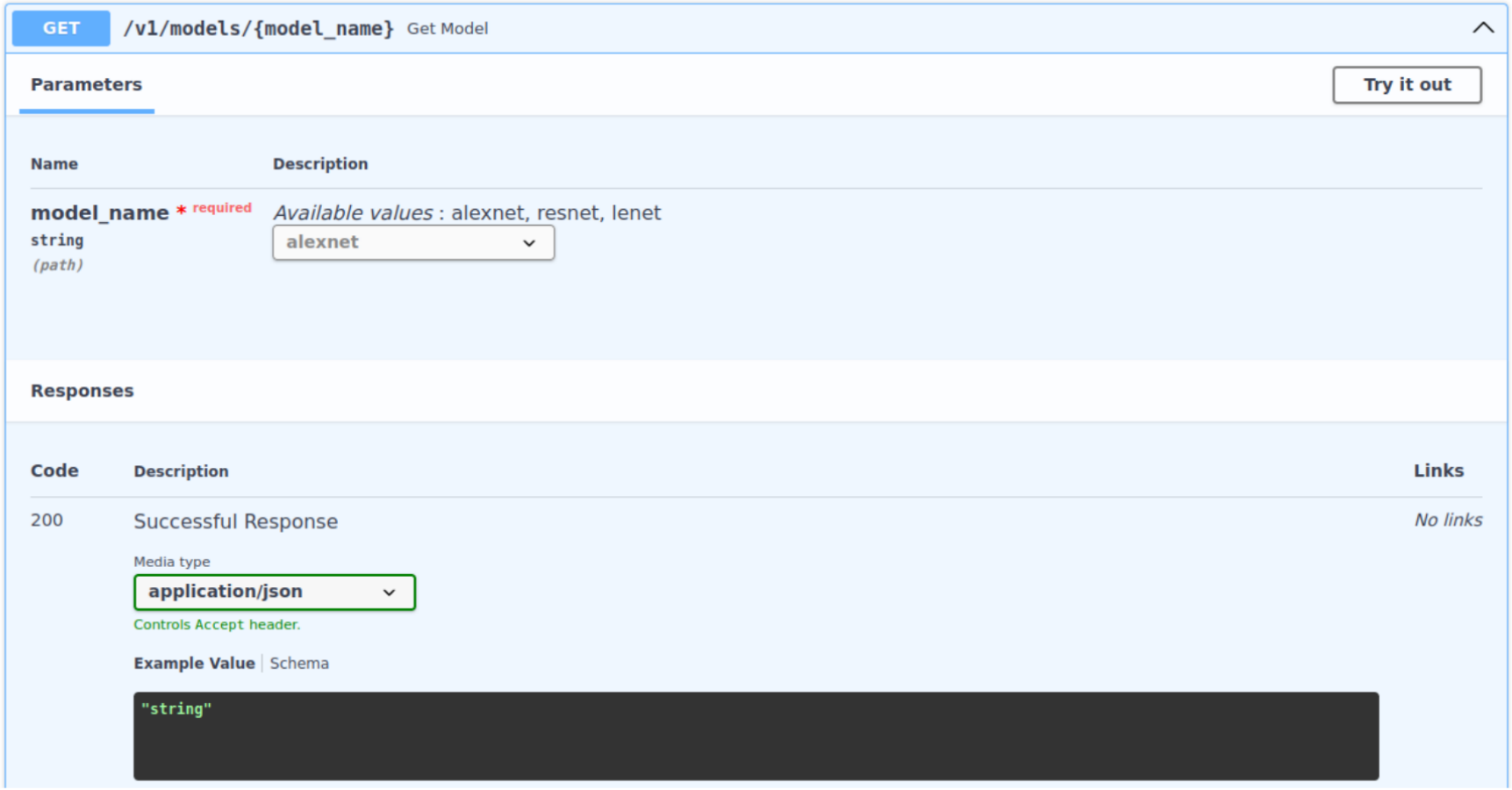
- Navigate to Swagger UI:
  - http://localhost:8000/docs
- Open API specification
  - http://localhost:8000/openapi.json

- Based on tutorials:
  - https://fastapi.tiangolo.com/tutorial/

# Task 0 – environment setup



# Task 0 – environment setup



# Task 0 – environment setup

```python
from fastapi import FastAPI
from enum import Enum

app=FastAPI( )

# sample requests and queries
@app.get("/")
async def root() :
    return {"message" : "Hello World"}

# sample path paramters => entries in URL
@app.get("/hello/{name}")
async def say_hello(name: str) :
    return {"message" : f"Hello {name}"}

# Path parameters predefined values
# https://fastapi.tiangolo.com/tutorial/path-params/
class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

@app.get("/v1/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}
    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}
    return {"model_name": model_name, "message": "Have some residuals"}
```

```python
# query parametres are added as elements to the url e.g.
items?skip=10&limit=3
# https://fastapi.tiangolo.com/tutorial/query-params/
fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"},
{"item_name": "Baz"}]

@app.get("/v2/items")
async def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]

# Optional parameters added to query, one of the element in Union
from typing import Union

#In this case, there are 3 query parameters:
# needy, a required str.
# skip, an int with a default value of 0.
# limit, an optional int.

@app.get("/v3/items/{item_id}")
async def read_user_item(
    item_id: str, needy: str, skip: int = 0, limit: Union[int, None] = None
):
    item = {"item_id": item_id, "needy": needy, "skip": skip, "limit": limit}
    return item
```

# Task 0 – environment setup

```python
# if you want to send it as a request body you have to define the class inheritet from pydantic base model
# Request Body
# https://fastapi.tiangolo.com/tutorial/body/
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
# create model
@app.post("/v4/items/")
async def create_item(item: Item):
    return item
# using model

@app.post("/v5/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict

# all together

@app.put("/v6/items/{item_id}")
async def create_item(item_id: int, item: Item, q: Union[str, None] = None):
    result = {"item_id": item_id, **item.dict()}
    if q:
        result.update({"q": q})
    return result
```

```python
# If the parameter is also declared in the path, it will be used as a path parameter.
# If the parameter is of a singular type (like int, float, str, bool, etc) it will be interpreted as a query parameter.
# If the parameter is declared to be of the type of a Pydantic model, it will be interpreted as a request body.

# additional status code:
# https://fastapi.tiangolo.com/advanced/additional-status-codes/

from fastapi import Body, FastAPI, status
from fastapi.responses import JSONResponse

items = {"foo": {"name": "Fighters", "size": 6}, "bar": {"name": "Tenders", "size": 3}}

@app.put("/v7/items/{item_id}")
async def upsert_item(
    item_id: str,
    name: Union[str, None] = Body(default=None),
    size: Union[int, None] = Body(default=None),
):
    if item_id in items:
        item = items[item_id]
        item["name"] = name
        item["size"] = size
        return item
    else:
        item = {"name": name, "size": size}
        items[item_id] = item
        return JSONResponse(status_code=status.HTTP_201_CREATED, content=item)
```

# Task 1 – Doodle API example

- Create simple Doodle API
- Small API for voting
  - User can create poll (see what is insider poll)
  - User can cast a vote inside this polls
  - User can add, update and delete all information he provides
  - User can see the results of votes
- Construct API and build the system
  - Test it with the Swagger UI

# Homeworks

**Termin na wysłanie zadania w systemie UPEL:**

**poniedziałek, 24 marca 2025, godz. 13:00 (wszystkie grupy).**

**Webex link: https://agh-mche.webex.com/meet/marekko**

### Temat

Celem zadania jest **napisanie prostego serwisu udostępniającego dane zgodne z REST w oparciu o otwarte serwisy udostępniające REST API**. Stworzyć macie Państwo serwis, który:

1. udostępni klientowi ***statyczną stronę HTML*** z formularzem do zebrania parametrów żądania,
2. strona serwerowa udostępnia REST API, odbierze zapytanie od klienta, i następnie je przetworzy,
3. odpyta serwis publiczny (różne endpointy), a lepiej kilka serwisów o dane potrzebne do skonstruowania odpowiedzi,
4. dokona odróbki otrzymanych odpowiedzi (np.: wyciągnięcie średniej, znalezienie ekstremów, porównanie wartości z różnych serwisów itp.),
5. wygeneruje i wyśle odpowiedź do klienta.

Wybranie realizowanej funkcjonalności i używanych serwisów pozostawiam Państwa wyobraźni, zainteresowaniom i rozsądkowi. Przykładowo:
*Klient podaje miasto i okres czasu ('daty od/do' lub 'ostatnie n dni'), serwer odpytuje ogólnodostępny serwis pogodowy o temperatury w poszczególne dni, oblicza średnią i znajduje ekstrema i wysyła do klienta wszystkie wartości (tzn. prostą stronę z tymi danymi). Ewentualnie serwer odpytuje kilka serwisów pogodowych i podaje różnice w podawanych prognozach.*
Z reguły wygrywa prognoza pogody lub kursy walut, ale **liczę na wykazanie się większą kreatywnością** ;-)

Listę różnych publicznych API można znaleźć np.: na https://publicapis.dev/

Wymagania (czyli jeszcze raz i bardziej szczegółowo)

# Homeworks

**Punktacja (0-6)**

- Strona serwerowa ***udostępnia REST API***, klient może komunikować się z udostępnionym API
- Premiowane będą rozwiązanie inne niż prognoza pogody i kursy walut
- Klient (przeglądarka) ma wysyłać żądanie w oparciu o dane z formularza (statyczny HTML) i otrzymać odpowiedź w formie prostej strony www, wygenerowanej przez tworzony serwis. Wystarczy użyć czystego HTML, bez stylizacji, bez dodatkowych bibliotek frontendowych (nie jest to elementem oceny). Nie musi być piękne, ma działać
- Tworzony **serwis powinien wykonać kilka zapytań** (np.: o różne dane, do kilku serwisów itp.). Niech Państwa rozwiązanie nie będzie tylko takim proxy do jednego istniejącego serwisu i niech zapewnia dodatkową logikę (to będzie elementem oceny, najlepiej 2 lub więcej)
- Odpowiedź dla klienta musi być generowana przez serwer na podstawie: 1) żądań REST do publicznych serwisów i 2) lokalnej obróbki uzyskanych odpowiedzi.
- Serwer ma być uruchomiony **na własnym serwerze aplikacyjnym działającym poza IDE** (lub analogicznej technologii)
- Dodatkowym (ale nieobowiązkowym) atutem jest wystawienie serwisu w chmurze (np.: Heroku). To jest część dla zainteresowanych i nie podlega podstawowej ocenie
- Dopuszczalna jest **realizacja zadania w dowolnym wybranym języku/technologii** (oczywiście sugerowany jest Python i FastAPI). Proszę jednak o zachowanie analogicznego poziomu abstrakcji (operowanie bezpośrednio na żądaniach/odpowiedziach HTTP, kontrola generowania/odbierania danych)
- Wybieramy **serwisy otwarte lub dające dostęp ograniczony, lecz darmowy**, np.: używające kodów deweloperskich

**Punktacja (0-9)**

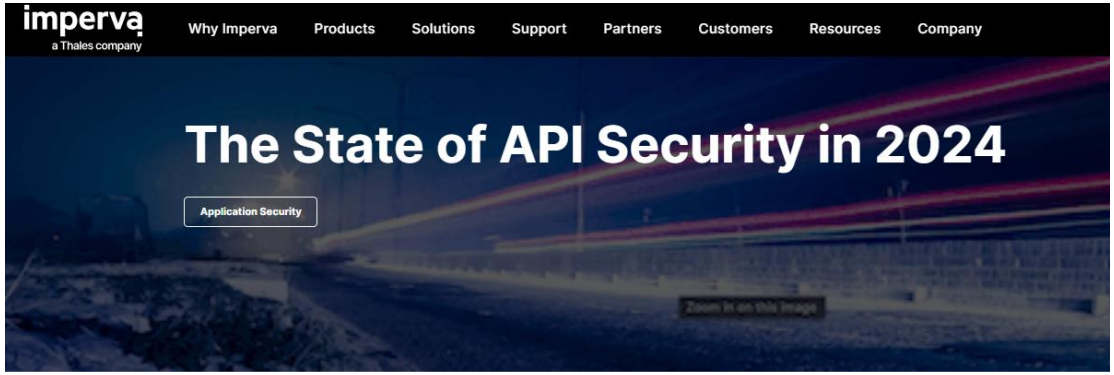
- Wymagania z funkcjonalności powyżej
- Przygotowujemy **test zapytań HTTP z wykorzystaniem POSTMANa/SwaggerUI** (klient-serwer i serwer-serwis_publiczny). Do oddania proszę mieć je już zapisane i gotowe do demo
- **obsługę (a)synchroniczności zapytań** serwera do serwisów zewnętrznych (np.: *promises*)
- walidację danych wprowadzanych przez klienta/przyjmowanych przez REST API
- Warto zwrócić uwagę na **obsługę błędów i odpowiedzi z serwisów** (np.: jeśli pojawi się błąd komunikacji z serwisem zewnętrznym, to generujemy odpowiedni komunikat do klienta, a nie *501 Internal server error*)

**Punktacja (0-10)**

- Wymagania z funkcjonalności powyżej
- Implementacja elementów bezpieczeństwa i zabezpieczania API

# Security

- The most common security issues:
  - Shadow API
  - Deprecated API
  - Unauthorized API




**API calls make up a massive 71% of all web traffic**

Widespread API usage is expanding the attack surface for bad actors. These threats make it critical for organizations to understand the risks and complexities of APIs and the importance of API Security.