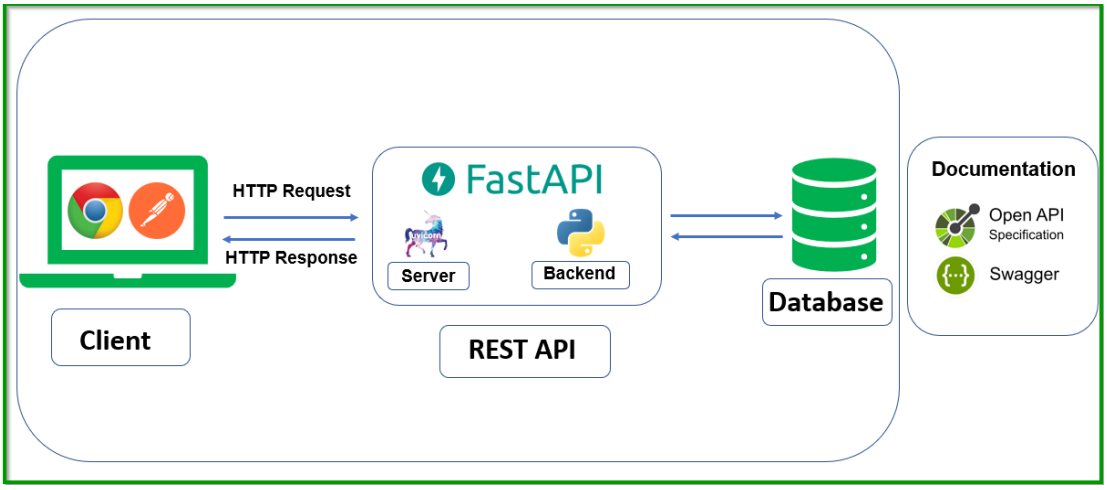
How to build a simple RESTful Car-API with FastAPI and Python?

Part 1: Project Goal & Overview

The goal of this project is to build a RESTful API system using FastAPI, with Uvicorn serving as the server, and Python as the backend programming language. This system allows for the efficient processing of HTTP requests from various clients, such as web browsers and API testing tools



Components

- **Clients**: Chrome browser and **Postman** are clients in this context. **Chrome** can be used to make GET requests directly from the browser's address bar, while Postman allows you to make GET, POST, PUT, and DELETE requests and provides a more comprehensive interface for API testing
- **FastAPI**: Our chosen web framework, FastAPI, will handle endpoint routing, request processing, and response generation. It provides a robust and efficient platform for building RESTful APIs
- **Uvicorn**: Uvicorn is employed as the ASGI server to run our FastAPI application. It listens for incoming HTTP requests, forwards them to the FastAPI application, and sends back the responses
- **Python Backend**: Python serves as the backend programming language. It contains the application logic, data models, and endpoint handlers, ensuring seamless integration with FastAPI
- **Database**: In this project, a simple dictionary is used which acts as a placeholder for a database. It simulates the storage of car data within your API. While it's not a full-fledged database system, it serves the purpose of illustrating API fuctionality.
- **Swagger/OpenAPI**: We use Swagger/OpenAPI to automatically generate interactive API documentation at endpoints like "/docs". This documentation allows users to explore, test, and understand the APIs.

Part 2 : Theoretical Concepts

What is FastAPI?

- modern, high-performance, and easy-to-use web framework for building APIs with Python
- designed to make it simple to create web APIs quickly and efficiently while also providing automatic interactive documentation
- Interactive API Documentation: FastAPI generates interactive API documentation automatically using the OpenAPI and Swagger standards. This documentation is accessible through a web interface and helps to understand and test the API.
- FastAPI is built with asynchronous programming in mind, allowing you to write asynchronous code when needed for I/O-bound tasks
- It supports various authentication and authorization mechanisms, making it suitable for building secure APIs.

What is Pydantic?

- Python library for data validation and parsing
- provides a convenient and declarative way to define data schemas and validate input data
- Automatic Parsing: It can automatically parse and convert data from various sources (e.g., JSON, dictionaries) into Python objects based on your data class definitions.
- Data Modeling: It is often used for modeling and validating data structures, especially when working with APIs, databases, or external data sources.

What is Swagger/OpenAPI?

- provides a clear and standardized way to document APIs, making it easier to understand how to use the API, what endpoints are available, and what data formats to expect
- enable automatic validation of API requests and responses against the defined schema, helping to ensure data consistency and correctness
- integrated in FastAPI

What is Uvicorn?

- ASGI (Asynchronous Server Gateway Interface) server that is commonly used to run Python web applications, including frameworks like FastAPI
- designed for serving web applications asynchronously, making it well-suited for high-performance, real-time, and asynchronous web applications

- includes a convenient "--reload" option that automatically detects code changes in your application and restarts the server, making it suitable for development and debugging.

Part 3 : Technical Part and Hands on

Requirement

- Build a Car API which allows you to perform CRUD (Create, Read, Update, Delete) operations
- Use Python and FastAPI
- Test your API (for GET-Requests use your Browser & Postman, for POST/PUT/DELETE use Postman)

Import the necessary modules

In [ ]:
```python
# "FastAPI" is our web framework, "HTTPException" for
# HTTP-related exceptions
from fastapi import FastAPI, HTTPException
# "BaseModel" for data validation and parsing,
# "Field" for specifying additional information about fields in data models
from pydantic import BaseModel, Field
```

Instance of the FastAPI class

In [ ]:
```python
app = FastAPI()
```

Create a database/record of Cars in a dictionary format

In [ ]:
```python
cars = {
    1: {"make": "Toyota", "model": "Camry", "year": 2005},
    2: {"make": "Honda", "model": "Civic", "year": 2014},
    3: {"make": "Ford", "model": "Mustang", "year": 2022},
}
```

Pydantic data model class (CarModel)

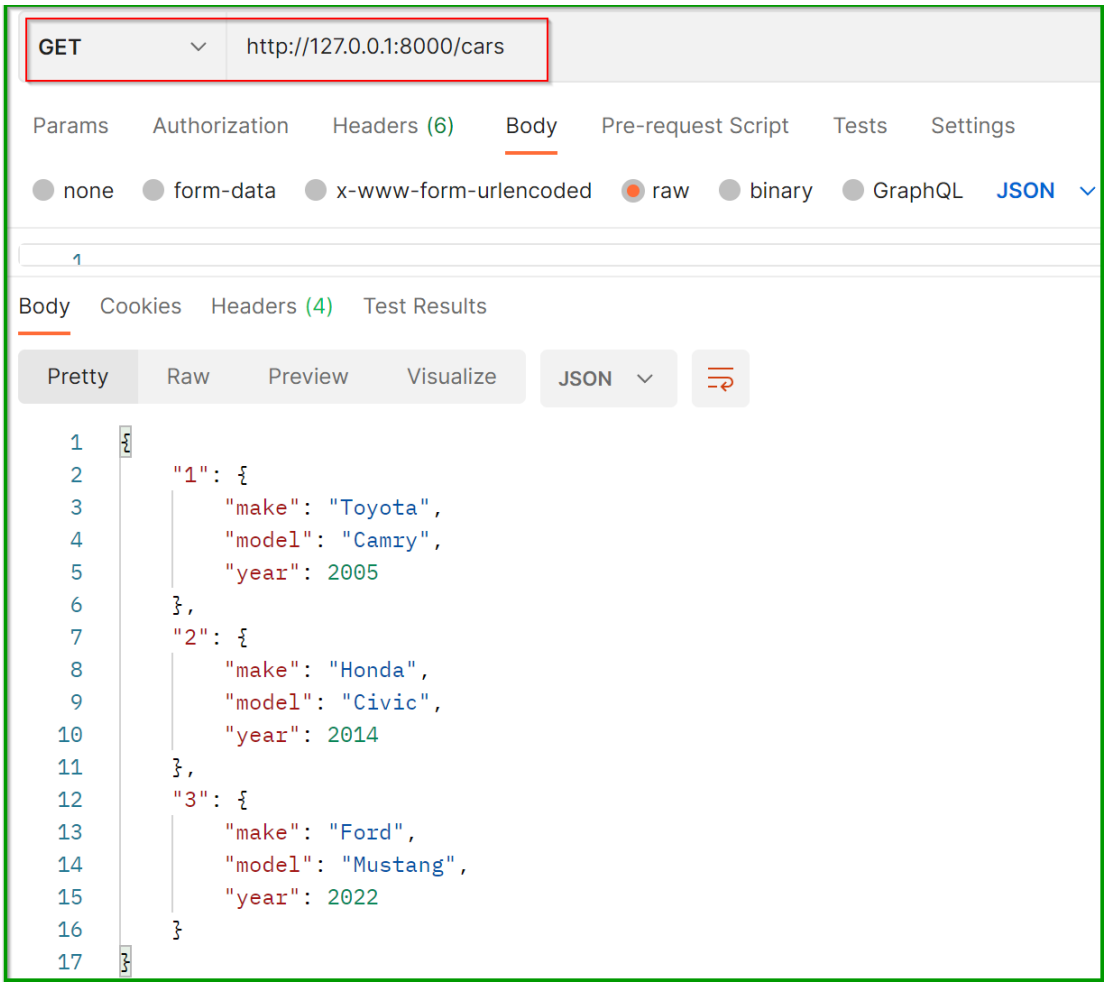
In [ ]:
```python
# CarModel inherits from BaseModel (provided by Pydantic)-
# specifies data structure
# "description" and "example" are helpful for documentation,
# and are not mandatory fields
class CarModel(BaseModel):
    # "make" with the data type String
    make: str = Field(description="Make of the car", example="Toyota")
    # "model" with the data type String
    model: str = Field(description="Model of the car", example="Camry")
    # "year" with the data type Integer
    year: int = Field(description="Year of the car", example=2020)
```

GET-Endpoint - All Cars

In [ ]:
```python
# Decorator for GET-Requests
@app.get("/cars")
# function that handles GET-Requests
def get_cars():
    # returns cars dictionary
    return cars
```

Test your API in your Browser (e.g. Chrome)

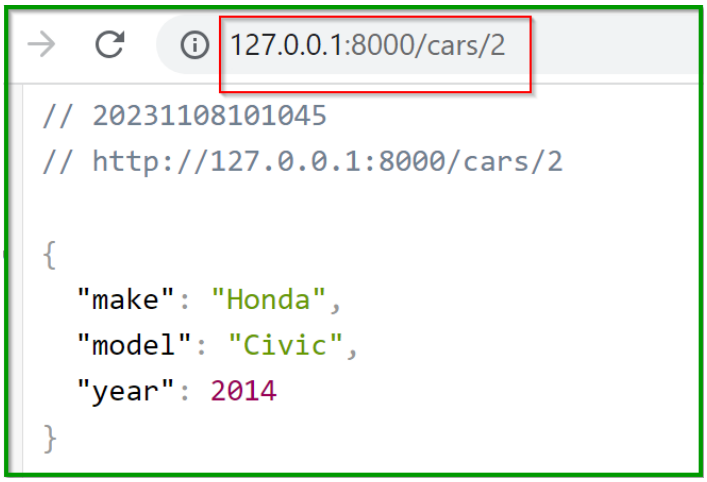
Test your API with Postman



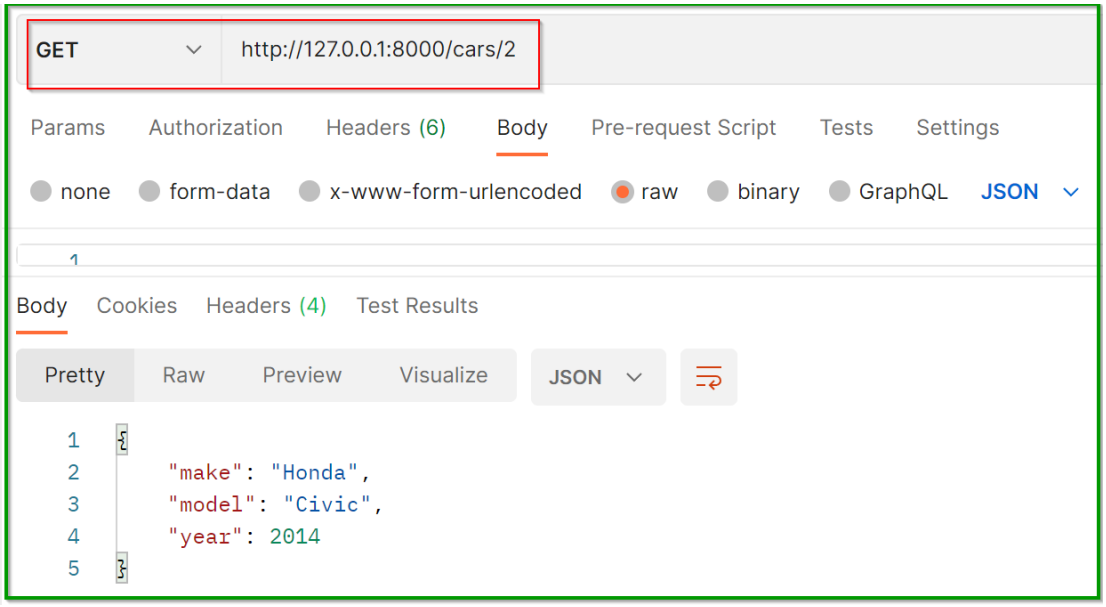
GET-Endpoint - Find a Car by ID

In [ ]:
```python
# Decorator for GET-Requests, "car_id" has to be provided by user
@app.get("/cars/{car_id}")
# function takes one parameter (car_id must be integer)
def get_car(car_id: int):
    # corresponding car information from the cars dictionary
    car = cars.get(car_id)
    # checks whether a car with the specified ID was found in the
    # cars dictionary
    if car:
        # If a car with the specified ID is found, you return the
        # car information as the response
        return car
    # If no car with the specified ID is found = HTTPException
    # + status code of 404 (Not Found)
    else:
        raise HTTPException(status_code=404, detail="Car not found"
```

Test your API in your Browser (e.g. Chrome)



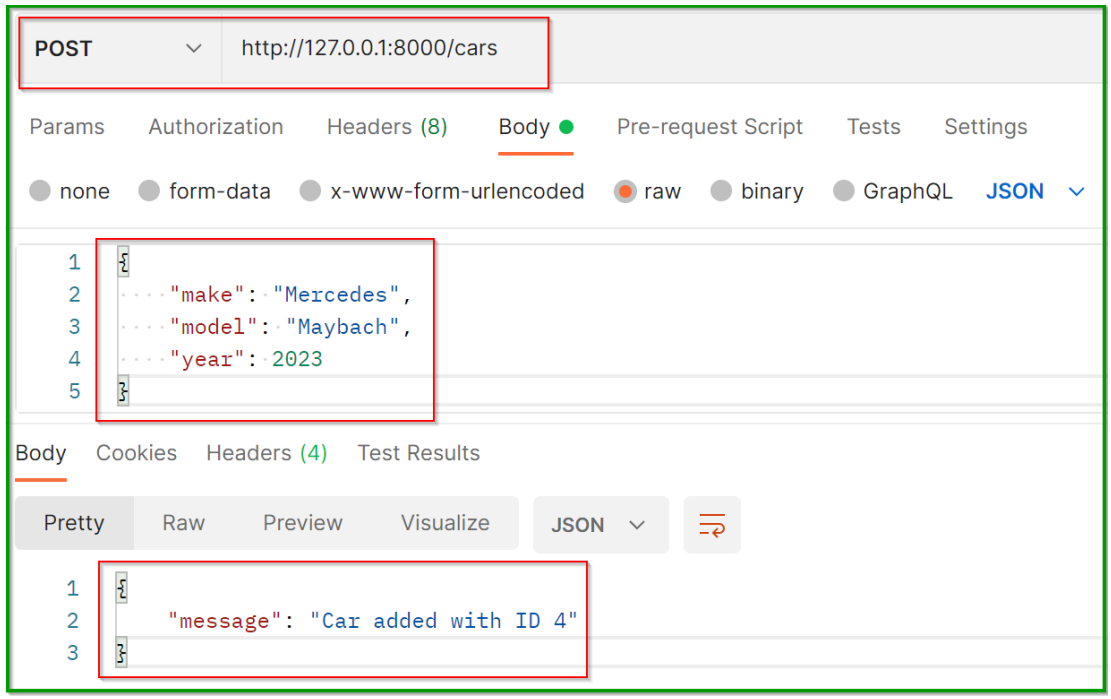
Test your API with Postman

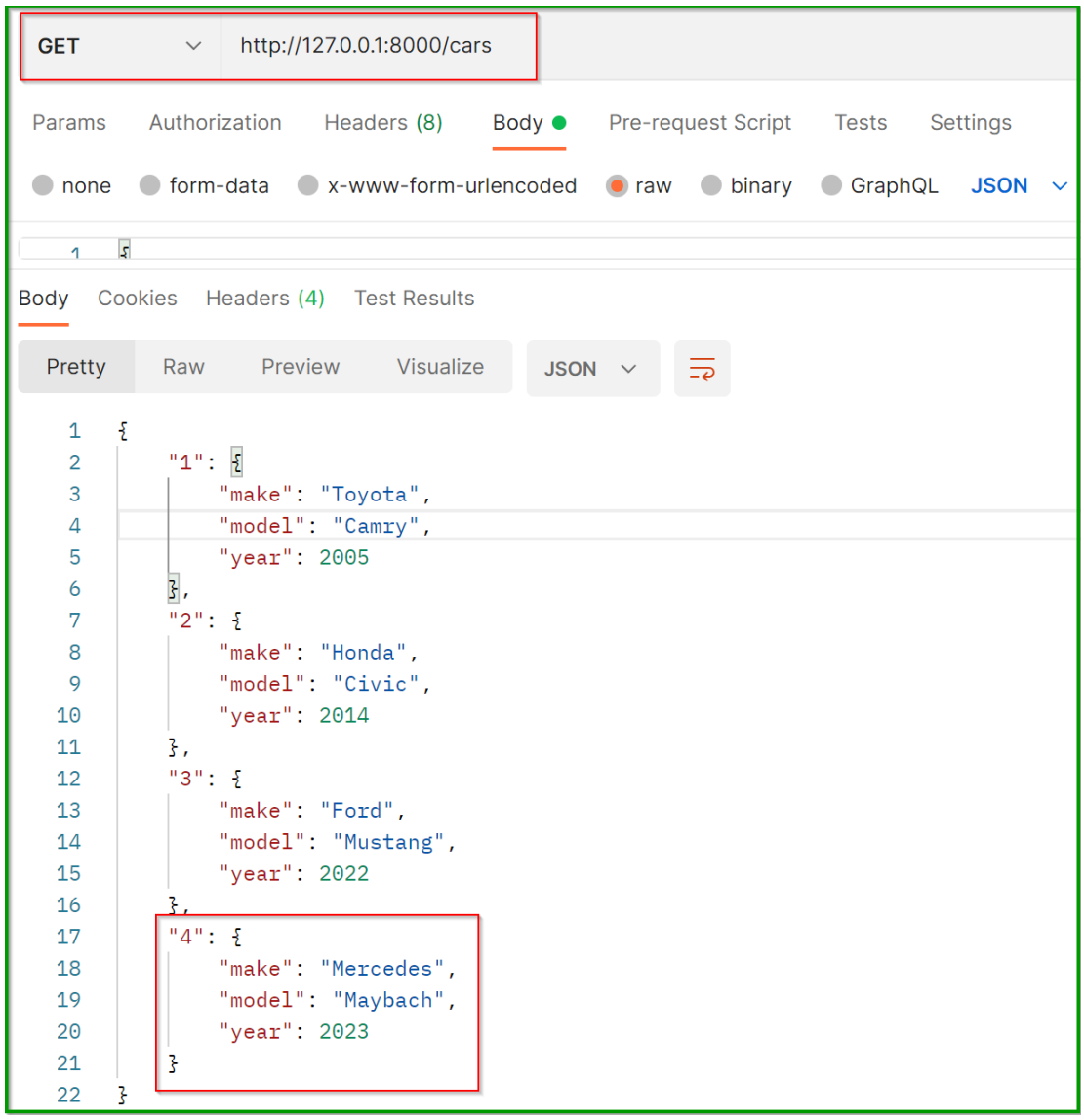


POST-Endpoint - Add a new Car

In [ ]:
```python
# Decorator for POST-Requests for a specific endpoint ("/cars")
@app.post("/cars")
# function to execute the POST-Request
# takes one parameter, car, of type CarModel (see above)
def add_car(car: CarModel):
    # generates a new unique ID (key) for the car to be added
    # If there are existing cars, it calculates the maximum key (ID)
    # and increments it by 1 to generate a new ID
    # If there are no existing cars, it sets the ID to 1
    car_id = max(cars.keys()) + 1 if cars else 1
    # adds the new car's data to the cars dictionary
    cars[car_id] = car.dict()
    # car has been added with its unique ID
    return {"message": f"Car added with ID {car_id}"}
```

Test your API with Postman

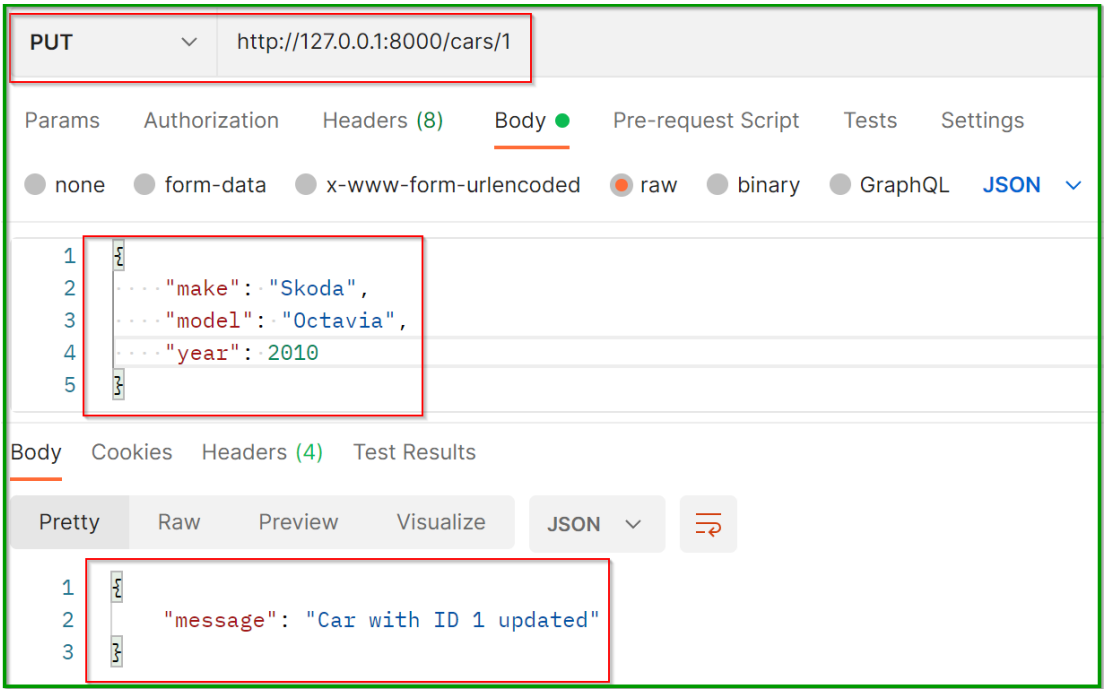


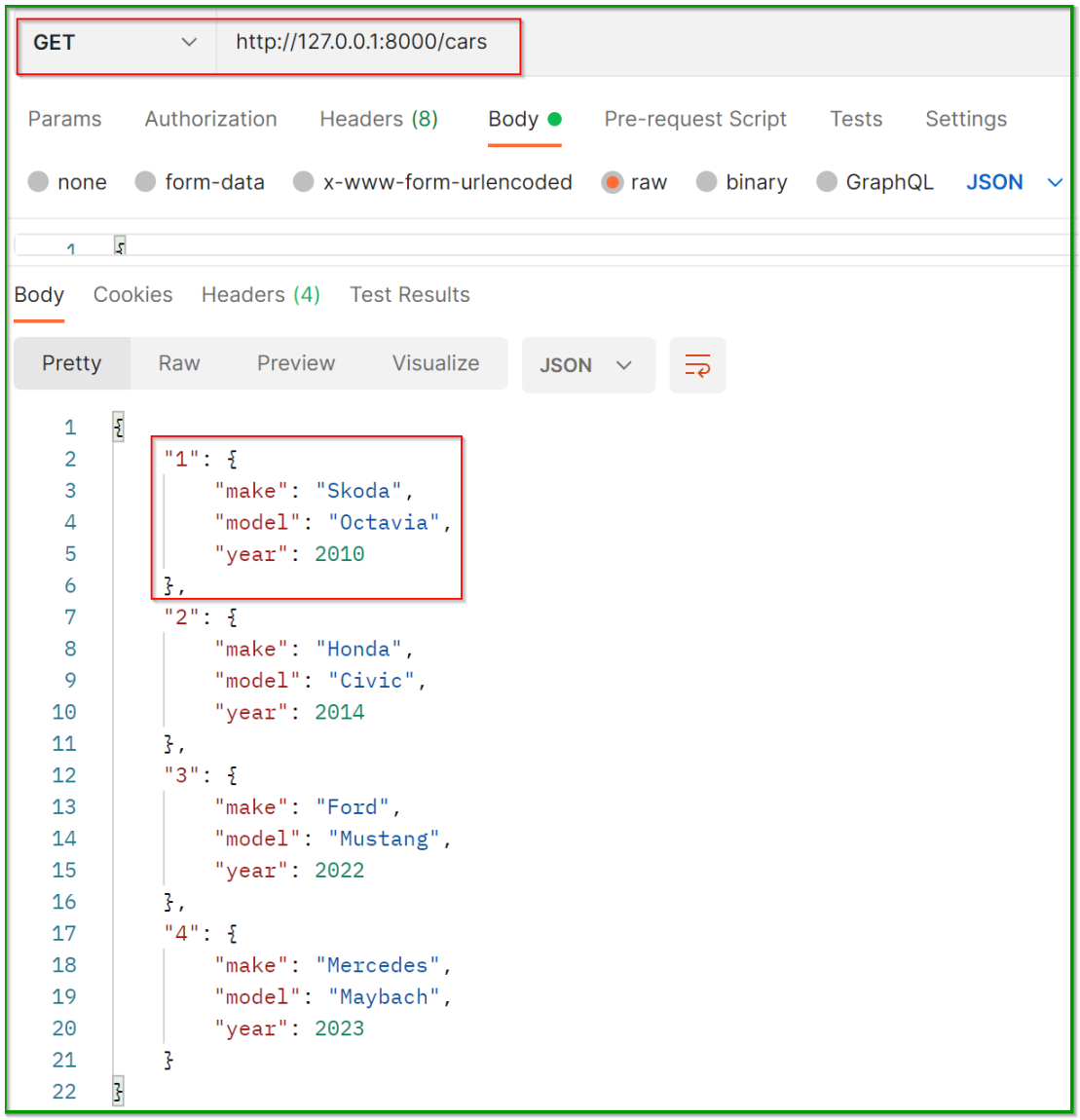


PUT-Endpoint - Update Car Information

In [ ]:
```python
# Decorator for PUT-Requests
# Endpoint has a dynamic parameter "car_id" - replace with actual ID
@app.put("/cars/{car_id}")
# function handles PUT-Request
# 2 parameters (ID and "updated_car" in request-
# type of CarModel (see above))
def update_car(car_id: int, updated_car: CarModel):
    # checks whether the specified car_id exists
    # in the cars dictionary
    # If the car is not found,code raises an HTTPException
    # with a status code of 404 (Not Found)
    if car_id not in cars:
        raise HTTPException(status_code=404, detail="Car not found")
    # If the specified car_id is found in the cars dictionary,
    # code proceeds to the else block
    else:
        # Update the car's attributes
        car = cars[car_id]
        car["make"] = updated_car.make
        car["model"] = updated_car.model
        car["year"] = updated_car.year
        # After successfully updating the car's information,
        # function sends success message
        return {"message": f"Car with ID {car_id} updated"}
```

Test your API with Postman

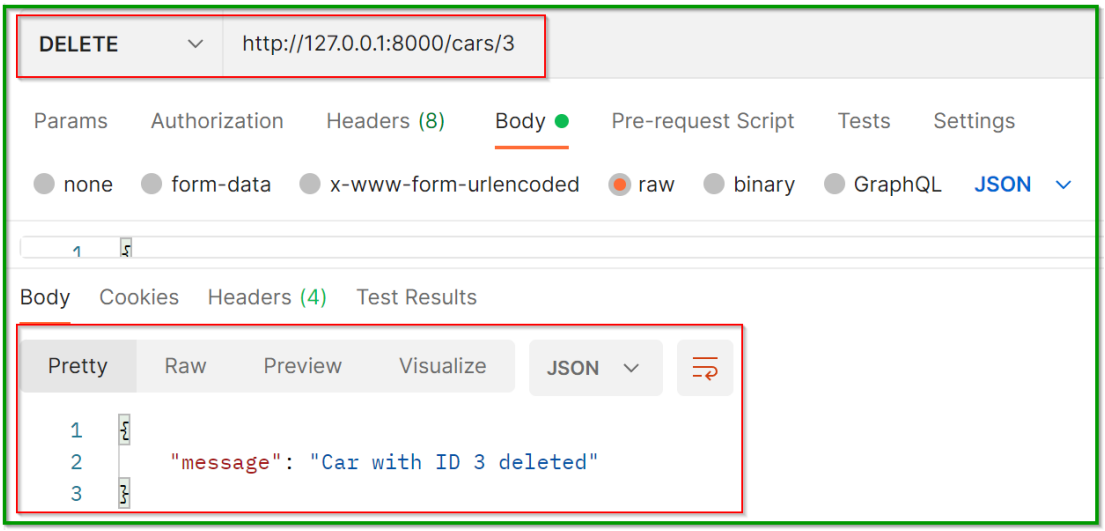


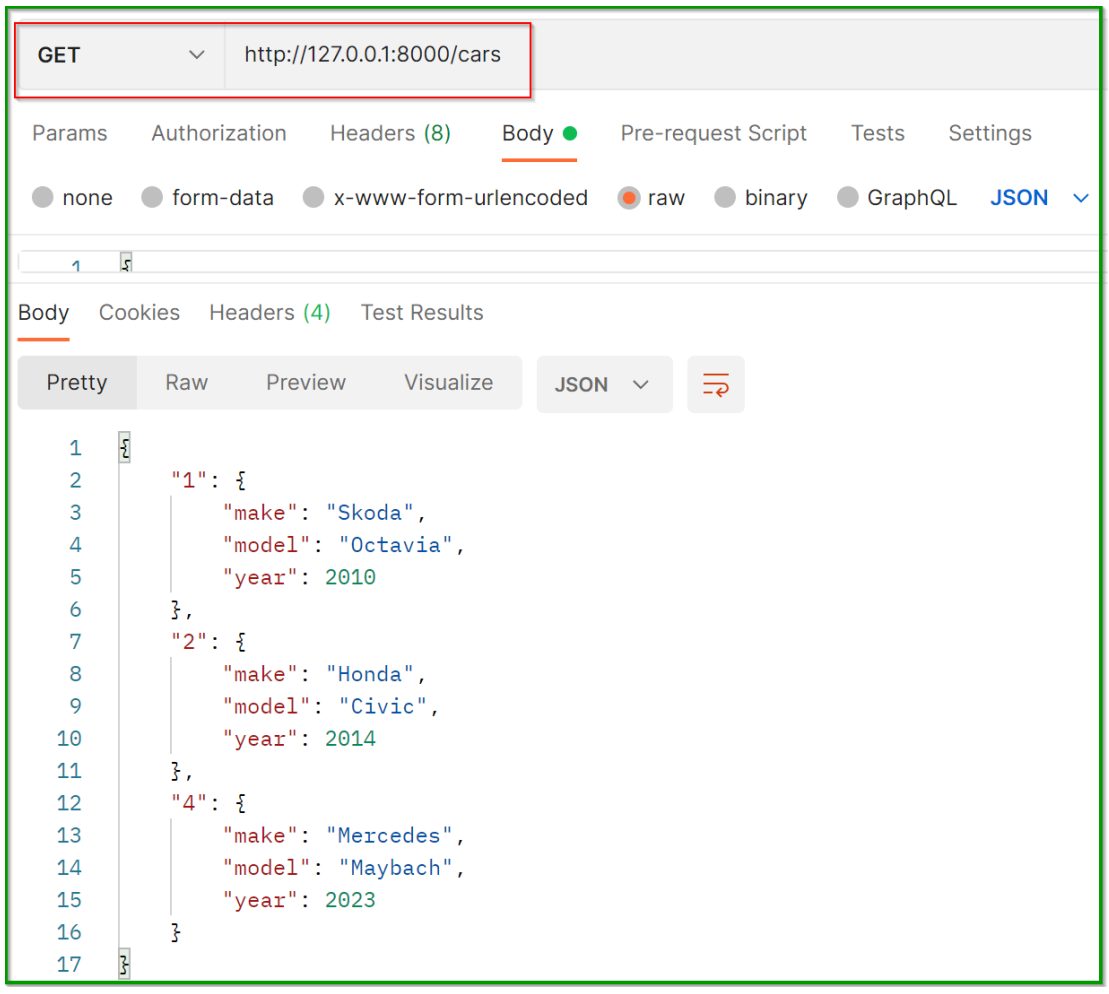


Delete-Request - Delete a Car (by ID)

In [ ]:
```python
# Decorator for DELETE-Requests - dynamic parameter "car_id"
# - client provides ID
@app.delete("/cars/{car_id}")
# functions handles DELETE-Request
def delete_car(car_id: int):
    #  checks whether the specified car_id exists in the
    # cars dictionary
    # If the car is not found, code raises an HTTPException with
    # a status code 404 (Not Found)
    if car_id not in cars:
        raise HTTPException(status_code=404, detail="Car not found")
    # If the specified car_id is found in the cars dictionary,
    # the code proceeds to the else block
    else:
        # deletes the car with the specified car_id from the
        # cars dictionary
        del cars[car_id]
        # JSON message indicating that the car with the specified
        # ID has been deleted
        return {"message": f"Car with ID {car_id} deleted"}
```

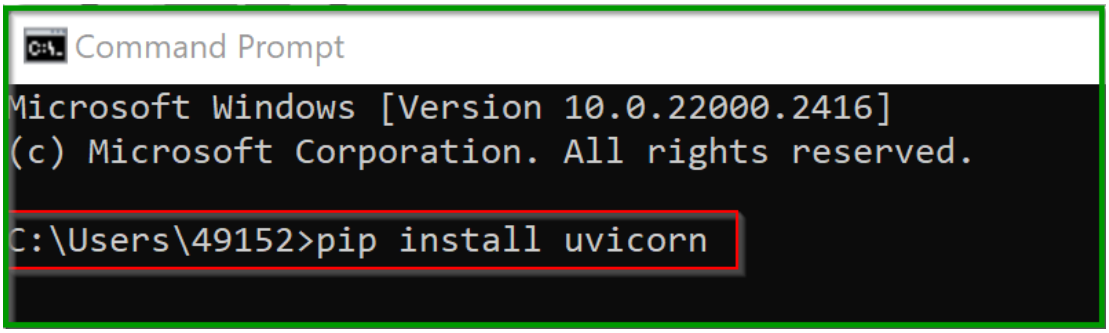
Test your API with Postman





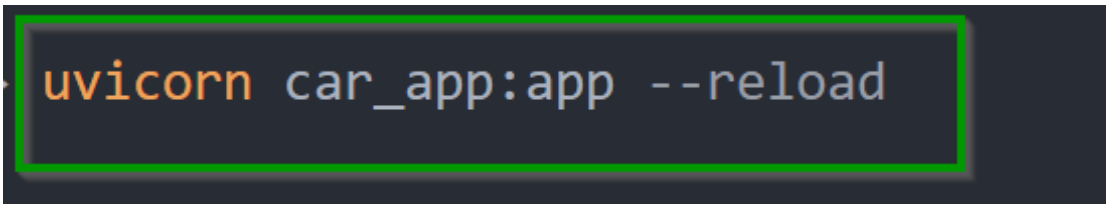
Run your script

Install uvicorn



Run your application

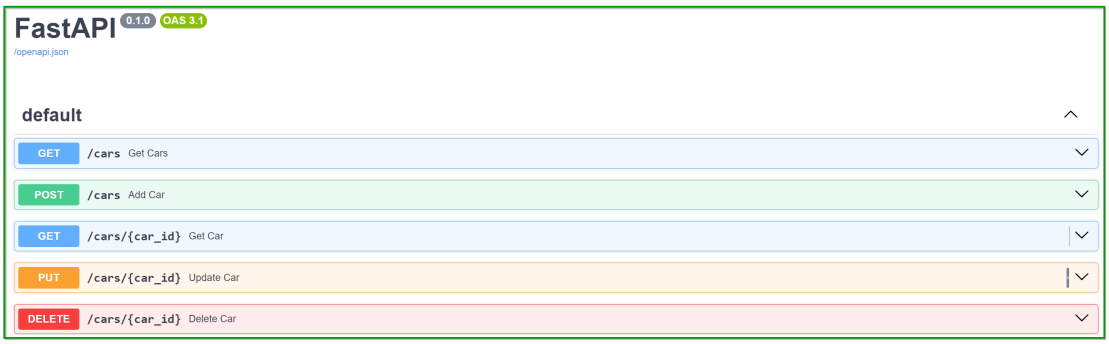
- In this example, the name of your script is "car_app.py", "app" is the instance of the FastAPI class (see above)
- In the context of running a FastAPI application using uvicorn, the "--reload" option is used to enable automatic reloading of the server when code changes are detected.



Part 4 - API Documentation

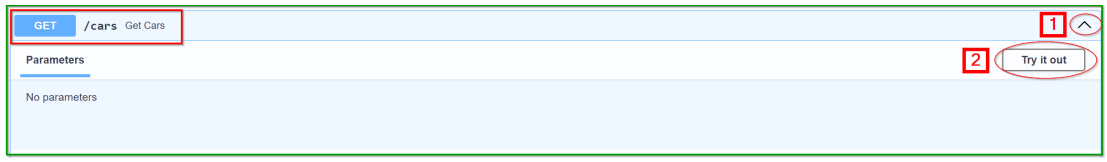
Swagger / Open API Overview

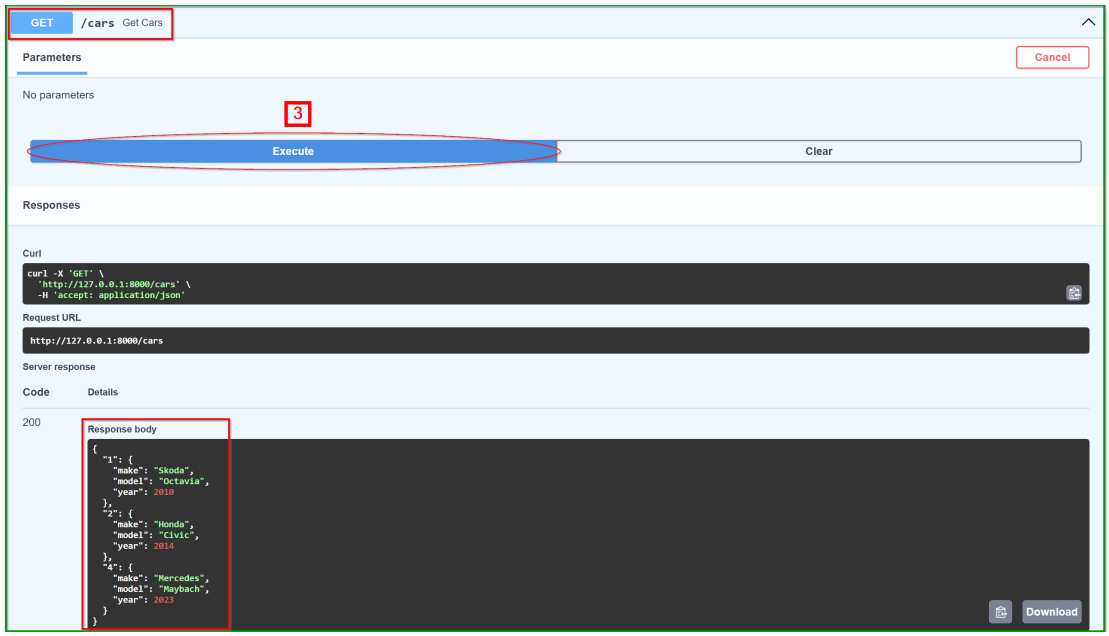
- Once your server is running, navigate to the following URL: http://localhost:8000/docs
- Replace "localhost:8000" with the appropriate host and port if you specified different values when running your application.



Swagger/Open API - Interact with the APIs

- 1. Expand the desired API
- 2. Click "Try it out"
- 3. Provide a Request Body if needed (required for POST and PUT)
- 4. Click on "Execute"


GET
/cars Get Cars
1
Parameters
2
Try it out
No parameters


GET
/cars Get Cars
Parameters
Cancel
No parameters
3
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://127.0.0.1:8000/cars' \
-H 'accept: application/json'
Request URL
http://127.0.0.1:8000/cars
Server response
Code
Details
200
Response body
{
"1": {
"make": "Skoda",
"model": "Octavia",
"year": 2010
},
"2": {
"make": "Honda",
"model": "Civic",
"year": 2014
},
"4": {
"make": "Mercedes",
"model": "Maybach",
"year": 2023
}
}
Download