

The background is a vibrant teal color with a complex geometric pattern of overlapping squares and cubes, creating a 3D effect. The shapes are rendered in various shades of teal, from light to dark, and some have white highlights on their top surfaces, giving them a three-dimensional appearance. The overall composition is abstract and modern.

GETTING STARTED WITH FASTAPI

ANDRÉS CRUZ YORIS
DEVELOPER

Getting started with FastApi

Here continue your roadmap in the development of web applications in Python
with FastApi

Andrés Cruz Yoris

This version was released: 2023-08-03

Tweet about the book!

Please help promote this book.

The suggested tweet for this book is:

I just bought the book "First steps with FastApi " from @LibreDesarrollo!

Get your copy at:

<https://www.desarrollolibre.net/libros/libro-primeros-pasos-con-fastapi>

About the Author

This book was prepared by Andrés Cruz Yoris, Bachelor of Computer Science, with more than 10 years of experience in the development of web applications in general; I work with PHP, Python and client side technologies like HTML, JavaScript, CSS, Vue among others and server side like Laravel, Flask , Django and CodeIgniter . I am also a developer in Android Studio, xCode and Flutter for the creation of native applications for Android and IOS.

I put at your disposal part of my learning, reflected in each of the words that make up this book, my twelfth book on software development but the second focused on Python, for the development of web applications with FastApi.

Copyright

No part of this book may be reproduced or transmitted in any way; that is, electronically or by photocopying without permission from the author.

Foreword

FastAPI is a great web framework for creating web APIs with Python; It offers us multiple features with which it is possible to create modular, well-structured, scalable APIs with many options such as validations, formats, typing, among others.

When you install FastAPI, two very important modules are installed:

- Pydantic that allows the creation of models for data validation.
- Starlette, which is a lightweight ASGI tooltip, used to create asynchronous (or synchronous) web services in Python.

With these packages, we have the basics to create APIs, but we can easily extend a FastAPI project with other modules to provide the application with more features, such as the database, template engines, among others.

FastAPI is a high-performance, easy-to-learn, start-up framework; It is ideal for creating all kinds of sites that not only consist of APIs, but we can install a template manager to return complete web pages.

This book is mostly practical, we will learn the basics of FastAPI, knowing its main features based on a small application that we will expand chapter after chapter.

Who is this book for

This book is aimed at anyone who wants to learn how to develop their first APIs in FastApi.

For those people who know how to program in Python or other web frameworks.

For those people who want to learn something new.

For people who want to improve a skill and who want to grow as a developer and who want to continue scaling their path in application development with Python.

Considerations

Remember that when you see the \$ symbol it is to indicate commands in the terminal; you don't have to write this symbol in your terminal, it is a convention that helps to know that you are executing a command.

At the end of each chapter, you have the link to the source code so you can compare it with your code.

The use of **bold** in paragraphs has two functions:

1. If we put a letter in **bold** it is to highlight some code such as names of variables, tables or similar.
2. Highlight important parts or ideas.

To display tips we use the following layout:

important tips

For the code snippets:

```
from fastapi import ***, Query
@app.get("/page")
def page(page: int = Query(1, ge=1, le=20), size: int = Query(5, ge=5, le=20)):
    return {"page": page, "size": size}
```

When using the ***

It means that we are indicating that there are fragments in the code that we presented previously.

As a recommendation, use Visual Studio Code as an editor, since it is an excellent editor, with many customization options, extensions, intuitive, lightweight, and that you can develop on a lot of platforms, technologies, frameworks, and programming languages; so all in all Visual Studio Code will be a great companion for you.

<https://code.visualstudio.com/>

This book has a practical focus, therefore, we will be presenting the main components of FastApi, ranging from a hello world to more complex structures and thus have a clear approach to this technology. Remember that you can consult from the book's index, the different components in case you want to review a particular topic at any time.

Errata and comments

If you have any questions, recommendations or found any errors, you can let them know at the following link:

<https://www.desarrollolibre.net/contacto/create>

By my email:

desarrollolibre.net@gmail.com

Or by Discord on the FastApi channel:

<https://discord.gg/sq85Zgwz96>

As a recommendation, before reporting a possible problem, check the version you are referring to and add it in your comment; it is found on the second page of the book.

Introduction

This guide is intended to get you started with FastAPI using Python; with this, we are going to raise two things:

1. It is not a book whose objective is to know FastApi 100%, or from zero to an expert, since it would be too big an objective for the scope of this guide, if not, to know what it offers us, to create the first methods that make up our API and how to customize them at the level of arguments, responses among others; in addition to learning how to integrate FastAPI with other third-party components.
2. It is assumed that the reader has at least basic knowledge of Python development.

This book has a practical approach, knowing the key aspects of technology and going into practice, gradually implementing small features of an application that has a real scope.

To follow this book you need to have a computer running Windows, Linux, or MacOS.

The book is currently in development.

Map

This book consists of 12 chapters, with which we will know in detail the most important and basic features of FastAPI:

Chapter 1: We present some essential commands to develop in FastApi , we will prepare the environment and we will give an introduction to the framework .

Chapter 2: One of the main factors in FastApi is the creation of resources for the API through functions, in this section we will deal with the basics of this, introducing routing between multiple files as well as the different options for the arguments and parameters of these routes.

Chapter 3: In this section, learn how to handle HTTP status codes from API methods and also handle errors/exceptions from API methods.

Chapter 4: In this section we will see how to create sample data to use from the automatic documentation that FastAPI offers for each of the API methods.

Chapter 5: In this chapter we will see how to implement the upload of files, knowing the different existing variants in FastAPI.

Chapter 6: In this chapter we will see how to connect a FastAPI application to a relational database such as MySQL.

Chapter 7: In this chapter we will see installing and using a template engine in Python, specifically Jinja, with which we can return responses in HTML format.

Chapter 8: In this chapter we will see installing and using a template engine in Python, specifically Jinja, with which we can return responses in HTML format.

Chapter 9: In this chapter we will learn how to use dependencies.

Chapter 10: In this chapter we will see how to use middleware to intercept requests to API methods and execute some procedure before the request or after generating the response.

Chapter 11: In this chapter we will see how to create a user module, to register users, login, generate access tokens and logout.

Chapter 12: In this chapter we will see how to implement unit tests.

Table of Contents

Getting started with FastApi	1
Tweet about the book!	3
About the Author	4
Copyright	5
Chapter 1: Required software and tools installation	1
Required Software	1
Python	1
Visual Studio Code	2
Extension	2
Web browser	2
Chapter 2: Introduction to FastApi	3
Verify command access to Python 3 and Pip	3
Basic commands	4
Prepare the environment	4
Virtual environment	5
Install dependencies and prepare the project	5
FastApi	6
Uvicorn	6
Hello World	6
Parameters in routes (path)	8
Types of routes	9
FastApi core libraries	9
Chapter 3: Routing in FastApi	11
Routing between multiple files	11
Limit allowed values (predefined data)	17
Parameters with Query()	18
Path with Path()	19
Body	20
Validation of request bodies using Pydantic models	23
Validations	24
Nest classes	26
Inheritance	27
Validation by fields	28
Optional values and default values	29
Field types	32
Use a list as an argument	33
FastAPI automatic documentation	34
Swagger	34
ReDoc	34
Conclusions	35
Capítulo 4: Errors and status codes	36

HTTP status codes	36
Customize response status code in methods	36
Error handling	38
What is an HTTP exception?	38
Evaluate that the index is not out of range	39
Evaluate that the task has not been registered	39
Conclusion	40
Chapter 5: Declare sample data	41
Sample data in models	41
Nested relations	42
Option list	44
Sample data from argument	45
Sample data from argument	45
Simple example	45
Option list	46
Chapter 6: File Upload	47
File	47
UploadFile	48
Save file	49
Handle multiple files	50
Chapter 7: Database	51
Install dependencies	51
Configure connection	51
Changes in models	52
Create models (tables) of the database	52
Use the connection and create the tables in the database	53
Inject database session dependency in routes	55
What is the Depends class used for?	57
Demo: yield vs return	57
CRUD methods for tasks, database	58
Pagination	59
Relations	61
One to Many relationship	61
Changes in the CRUD of tasks (database)	62
Get the relationship from the main entity	63
Inverted relations	64
Many to Many relationship	65
Operations in a many-to-many relationship	67
Task CRUD changes (API methods)	68
Methods to add/remove tags to a task	68
Move sample data to separate files	69
API methods	72
Check if id exists in getByld()	74
Integration of the SQLAlchemy and Pydantic classes in the response of the API methods	74
Reading and writing Pydantic models	75
Remove columns from queries in SQLAlchemy	77

Dates	78
Chapter 8: Template in FastAPI	79
About Jinja	79
First steps with Jinja	79
Control blocks	79
Conditionals	79
Bucle for	80
Filters	80
Filter default	80
Filter escape	81
Filter conversion	81
Filter max	81
Filter min	82
Filter round	82
Filter replace	82
Filter join	83
Filter lower	83
Filter upper	83
Filtro reverse	83
Filter length	84
Filter sort	84
Filter slice	85
Set variables	85
Blocks	85
Block raw	86
Block macro	86
Template inheritance	86
Master template	86
Include View Fragments	89
Using Jinja in FastAPI	89
Install Jinja	89
HTML task list	91
Requests in JavaScript	92
Create a task	92
Update a task	95
Delete a task	97
HTML forms	97
Chapter 9: Dependencies	102
Path dependencies (API method decorator)	103
Define dependencies as variables	103
Chapter 10: Introduction to Middleware	105
Create a middleware	105
Chapter 11: Authentication	107
Basic authentication	107
Database authentication	109
Methods for the API	112

Protect routes with authenticated user	114
Logout	115
Protecting API methods using the OAuth2PasswordBearer dependency	116
Verify access token at token creation time	117
Capítulo 12: Annotations, Ellipsis (...) notation, and return types	119
Ellipse notation	119
Annotated	120
Examples using Annotated	122
Return types	123
Chapter 13: Testing in FastAPI applications	124
pytest, for unit tests	124
Creating the first unit tests	124
Delete repetition with pytest fixtures	126
Extra: Loop event	128
Base configurations	132
Testing the user module	134
Create a user	134
Login	136
Logout	136
Testing the task module	137
Create a task (Body Json)	137
Update a task	138
Get all tasks	139
Get the detail of a task	139

Chapter 1: Required software and tools installation

Python is an excellent language to start programming, it is simple, modular, and due to its very structure, when using indentation to delimit the structure, the code is correctly indented; with basic knowledge of Python, we can use it to master a large part of the framework as we are going to learn in this book.

Required Software

As fundamental software, we need Python, this is the only program that you have to install on your computer, since it is the programming language in which FastAPI is programmed, and the one that we have to use to create our applications with this framework; the python web has been this:

<https://www.python.org/>

Python

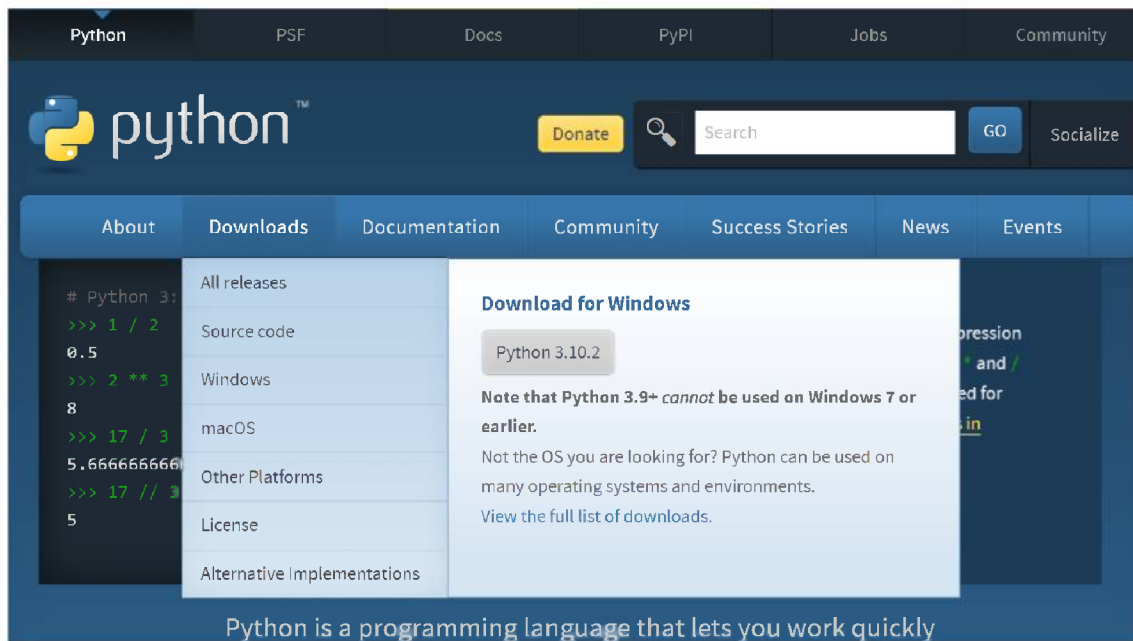
Before we go any further, let's talk a bit more about Python; Python is a high-level and interpreted programming language, object-oriented and with a very clean, friendly, easy-to-learn syntax and great readability with a large number of packages, modules, libraries and frameworks at our disposal make it very attractive for rapid application development.

Programmers often fall in love with Python because of the increased productivity it provides. Since there is no build step, the edit, test, and debug cycle is incredibly fast and easy.

There is no need for you to install a new version of Python since you can work with the one that already comes with your system.

But, in case you need to install Python, it's extremely easy; we go to the official website, in the download section:

<https://www.python.org/>



We download it, execute it and it is the typical “next next next” to install it; very important that you add Python to the Windows PATH:

Add Python X to PATH

Visual Studio Code

As a code editor, we are going to use Visual Studio Code since it is an excellent editor, with many customization options, extensions, intuitive, light and that you can develop on a lot of platforms, technologies, frameworks and programming languages; so overall Visual Studio Code will be a great companion for you; but, if you prefer other editors like Sublime Text, or similar, you can use it without any problem.

<https://code.visualstudio.com/>

Extension

If you use VSC, you have to install the python extension:

<https://marketplace.visualstudio.com/items?itemName=ms-python.python>

Web browser

As a web browser, I recommend Google Chrome; although it is true that when developing in web technologies, it is advisable to use more than one browser; by developing specifically for the server side and not focusing on developing on the client side; It doesn't make much sense for this book to use multiple browsers; that being said, you can use any other browser in case Google Chrome is not to your liking:

<https://www.google.com/intl/es/chrome/>

Chapter 2: Introduction to FastApi

In this first chapter we are going to learn about some essential commands to be able to work on a project with FastApi, as well as preparing the development environment through virtual environments and creating the first example with FastApi, that is, "Hello World".

Verify command access to Python 3 and Pip

Before you start, it's important to check which version of Python you're using; in some operating systems such as Linux or MacOS an old version of Python may be installed, often version 2 of Python with which you cannot use FastApi; to be able to create our applications with FastApi it is necessary that you have version 3.

Once Python 3 is installed on your computer, check what is the way to access it; in your terminal, if you run the following command:

```
$ python -V
```

And you see something like this:

```
command not found: python
```

It means that Python hasn't been installed, or at least it's not the way to access Python 3; if you see output like this:

```
Python 3.X
```

It means that to access the Python interpreter you have to simply type "python" in your terminal.

In case the previous command did not work, you can try running the command of:

```
$ python3 -V
```

And you should see an output like:

```
Python 3.X
```

Where "X" is a variant of Python 3, such as **3.10.2** which means this is the way to access the Python interpreter on your PC.

You must do the same procedure with the package installer for Python, the so-called pip:

```
$ pip3 -V
```

Or

```
$ pip -V
```

In at least one case you should see something like:

```
| pip 22.X from /Library/Frameworks/ Python.framework /Versions/***/pip (python 3.X)
```

Basic commands

In order to create our applications in Python in an organized way, you should at least know the following commands, which are very useful at the time of development; let's see what they are.

To install dependencies in Python, which can be from a plugin to perform the social login or a framework like Django, Flask or in this case FastApi, you must use the option "pip install" followed by the package or packages we want to install; for example:

```
| $ pip install fastapi
```

To uninstall a package, we have the command "pip uninstall":

```
| $ pip uninstall fastapi
```

Another very useful command is the one that allows us to know which packages we have installed in a project; as it happens with other environments like Node or PHP with Composer, in Python many times when the installed packages contain dependencies; for example, in the case of Django, when you install Django, the following dependencies are installed:

```
| $ pip install django
```

```
asgiref ==3.5.0  
django==4.0.3  
sqlparse ==0.4.2  
tzdata ==2021.5
```

To know the dependencies, we have the following command:

```
| $ pip freeze
```

Whose output we can save in a file, which by convention is called as **requirements.txt**:

```
| $ pip freeze > requirements.txt
```

We can now use this file to install the dependencies of a project with the exact versions; very useful when you are developing a python project with multiple people or when you go to production or simply when you need to run the project on multiple PCs:

```
| $ pip install -r requirements.txt
```

Prepare the environment

Having clarified some basic commands that we must know both to execute project files and to maintain the project and its dependencies, we are going to create the virtual environment to create the first application in FastApi.

Virtual environment

In Python, virtual environments are development environments isolated from the operating system, therefore, the packages that we install will not be installed in the operating system but in the virtual environment; we can create as many virtual environments as we want, usually we have one virtual environment per project and with this, all the packages and dependencies necessary for a Python project in general are managed independently.

In Python, virtual environments can be created and managed with the **venv** tool, which is included in the Python standard library, although it is also possible to install it using:

```
$ pip install virtualenv
```

This is the only package that needs to be installed at the global level of the operating system.

To create the virtual environment, we have the command:

```
$ python3 -m venv venv
```

We are going to execute the following command:

```
$ python3 -m venv vtasks
```

Which will generate the virtual environment, which is nothing more than a folder with several files and folders; to activate it, from the virtual environment folder:

```
$ cd vtasks
```

We run on MacOS or Linux:

```
$ source vtasks/bin/activate
```

On Windows the command would be like:

```
$ vtasks\Scripts\activate
```

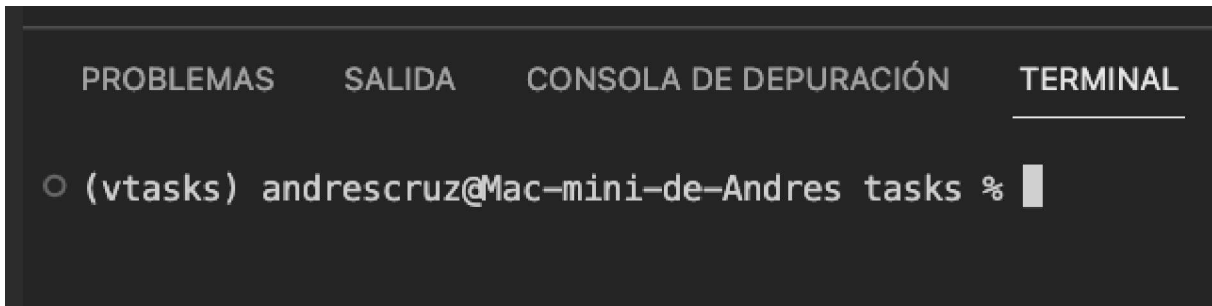
And with this, we already have everything ready to indicate our project; remember that to execute any Python or pip command you must always have the virtual environment active.

Install dependencies and prepare the project

For our first project, we are going to create a task app; to do this, inside the virtual environment folder, we will create the project folder that we will simply call tasks; leaving the project as follows:

- vtasks
 - include
 - lib
 - Scripts -- On Windows
 - bin -- On MacOS or Linux
 - pyvenv.cfg
 - **tasks**

This folder called "tasks " (not the vtasks folder, which corresponds to the virtual environment) is the one that I recommend you open in Visual Studio Code (or the editor of your choice); from the VSC terminal remember to activate the virtual environment:



In order to start working with FastApi, we need to install two packages that we will see next.

FastApi

FastAPI is a modern web framework used for creating APIs (application programming interfaces) in Python; to install the package we have:

```
$ pip install fastapi
```

Uvicorn

Uvicorn is an ASGI (Asynchronous Server Gateway Interface) HTTP web server that we can use in multiple asynchronous Python web frameworks such as Django, Flask and of course FastApi and to install the server we can do it as if it were a package more for Python with:

```
$ pip install uvicorn
```

Or we can install both with a single command:

```
$ pip install fastapi uvicorn
```

Remember to run the above command with the virtual environment active.

Hello World

Finally, it's time to create the "Hello World" with FastApi; that is, to implement the minimum necessary to be able to see something on the screen; so, we create a file inside the project (the tasks folder):

api.py

With the following code:

api.py

```
from fastapi import FastAPI
```

```
app = FastAPI()
@app.get("/")
def hello_world():
    return {"hello": "world"}
```

In this first example, the first thing we do is load a class that provides access to the FastAPI framework:

```
from fastapi import FastAPI
```

With this class, we create an instance of FastApi:

```
app = FastAPI ()
```

Which gives us access to multiple features of the framework, such as creating the application routes.

We define a GET request for the root and this is done through a decorator like the one we can see below:

```
@app.get("/")
```

Of course, we can access other types of requests such as POST, PUT, PATCH or DELETE indicating the corresponding method which has its direct equivalent with the name of the request to use; that is, to send a GET request, we use the function **get()**, to send a POST request, we use the **post()** function.

As with other web frameworks, each request is processed by a function, in the previous example, the GET request for the root is processed by a function called **hello_world()** which all it does is return a dictionary indicating the "hello world" message:

```
@app.get("/")
def hello_world():
    return {"hello": "world"}
```

With this, we have our first hello world example in FastApi; but, in order to see this message on the screen, specifically any program that allows processing HTTP requests such as a browser, we have to open the server associated with the previous application and this is where we use the previously installed uvicorn server; to do this, from the terminal and root of the project, we use the following command:

```
$ uvicorn api: app --reload
```

With the previous command, we indicate the name of the file, which in this case is called api.py:

```
api:app
```

And that the server stays tuned for changes; that is, with the option of:

```
--reload
```

The server will be reloaded every time changes are made to the application.

When executing the previous command, we will see in the terminal:

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [15820] using StatReload
INFO:      Started server process [4024]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      127.0.0.1:58209 - "GET / HTTP/1.1" 200 OK
INFO:      127.0.0.1:58209 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO:      127.0.0.1:58209 - "GET / HTTP/1.1" 200 OK
```

It tells us which is the route in which the application has been raised:

```
http://127.0.0.1:8000
```

And from this route, which we can execute in the browser, we will see the message of:

```
http://127.0.0.1:8000
```

```
{
  "hello": "world"
}
```

If necessary, you can also customize the port using the **port option** and indicate the port you want to use, in this example, port 8001:

```
$ uvicorn api:app --port 8001
```

With the **reload** option:

```
$ uvicorn api:app --port 8001 --reload
```

And in the output you will now see that the server is loading the application on port 8001

```
Uvicorn running on http://127.0.0.1:8001 (Press CTRL+C to quit)
```

Finally, the previous function can also indicate the return data type:

```
@app.get("/")
def hello_world() -> dict:
    return {"hello": "world"}
```

It is important to note that a GET type request is used, which is the typical type of request used to query data; and the only one that we can use directly from the browser without having to use a form.

Parameters in routes (path)

At the route level, you can also define parameters with which the end user interacts and they look like this:

```
@app.get("/posts/{id}")
```

As you can see, this route scheme allows the use of dynamic parameters and with this, being able to supply data to each of the API resources that we are building; in the previous example, it is indicated that for the route a parameter called `id` must be supplied, which is then supplied as an argument to the decorated function; that is to say:

```
@app.get("/posts/{id}")
def add(id: int):
    #TODO
    return {"POST ID": id}
```

In the previous example, a parameter called `"id"` must be indicated, which must be an integer type which can be used (for example) to search for the identifier of a post that you want to retrieve:

```
/post/123
```

Types of routes

There are different HTTP methods that we can use to make requests to the server; these methods are nothing more than a set of verbs that are used to perform different types of actions; the most common methods are GET, POST, PUT, PATCH, and DELETE:

- GET: used to get information from a web server. The information is sent in the request URL.
- POST: used to send information to a web server. The information is sent in the body of the request.
- PUT: used to update information on a web server. The information is sent in the body of the request.
- PATCH: used to partially update information on a web server. The information is sent in the body of the request.
- DELETE: used to delete information from a web server. The information is sent in the request URL.

All of these methods have their equivalent in FastApi; at the moment we have used GET type methods to obtain data, but it is also possible to use others:

```
@app.get(<URI>)
@app.post(<URI>)
@app.put(<URI>)
@app.patch(<URI>)
@app.delete(<URI>)
```

As you can see, we have a decorator function for each type of method and it has a direct relationship with the HTTP request type; that is, we have a decorator method `put()` to send requests of type PUT, and so on for the rest.

FastApi core libraries

To end this introduction with FastAPI, it is important to note that FastAPI is based on two main Python libraries: Starlette, a low-level ASGI web framework (<https://www.starlette.io/>), and Pydantic, a validation library. (<https://pydantic-docs.helpmanual.io/>), of which this last library, we will use heavily in the following sections.

Both libraries can be seen if you do a:

```
$ pip freeze
```

About your project in FastApi.

Chapter 3: Routing in FastApi

In the previous chapter, we saw how to create a "hello world" in FastApi by defining a route of type GET that returns the JSON response {"hello": "world"}. To do this, we first instantiate a **FastAPI** object that we imported before; this instance is saved in a variable called **app** which is the main object of the application that will connect all the API routes; With this object, it is possible to use a decorator to define the routes. This is not the best way for us to work with routes, since if we have additional files in which different routes are defined for the application using these decorators, we would have to pass the app object to all of these **files**; in FastApi we can also create an instance of the **APIRouter** class which can be used in the same way to decorate the functions (known as operation functions) of the route as we did with the FastApi instance.

So, with the above, we can create the "hello world" in FastApi as follows:

api.py

```
from fastapi import APIRouter
router = APIRouter()
@router.get("/")
def hello_world():
    return { "hello": "world" }

app.include_router(router)
```

The additional change we have to make is to add the routes defined before, for this, we use the **include_router()** function to include the instance used to define the routes:

```
app.include_router(router)
```

Remember to remove or comment the previous function to avoid conflicts between two equal routes; for example ("/").

Routing between multiple files

Router API class we saw earlier, we can create truly modular applications, separating functions that manage similar resources into files; In other words, put in the same group the functions that have a common behavior, such as task management, users, etc.

Thinking about the above, we are going to create a new file in which we will place the CRUD of the tasks:

task.py

```
from fastapi import APIRouter

task_router = APIRouter()
task_list= []

@task_router.get("/")
def get():
```

```

    return { "tasks": task_list }

@task_router.post("/{task}")
def add(task: str):
    task_list.append(task)
    return { "tasks": task_list }

@task_router.put("/")
def update(index: int, task: str):
    task_list[index] = task
    return { "tasks": task_list }

@task_router.delete("/")
def delete(index: int):
    del task_list[index]
    return { "tasks": task_list }

```

As you can see, the previous code consists of 4 functions, to create, update, delete and get a list respectively; in all of these examples, a list is used to simulate task management using the aforementioned methods.

To load the routes of this new file, we must do it from the main file, which is the **api.py** file; so:

api.py

```

from fastapi import FastAPI
from fastapi import APIRouter

from task import task_router

app = FastAPI()
router = APIRouter()

@router.get('/hello')
def hello_world():
    return { "hello": "world" }

app.include_router(router)
app.include_router(task_router)

```

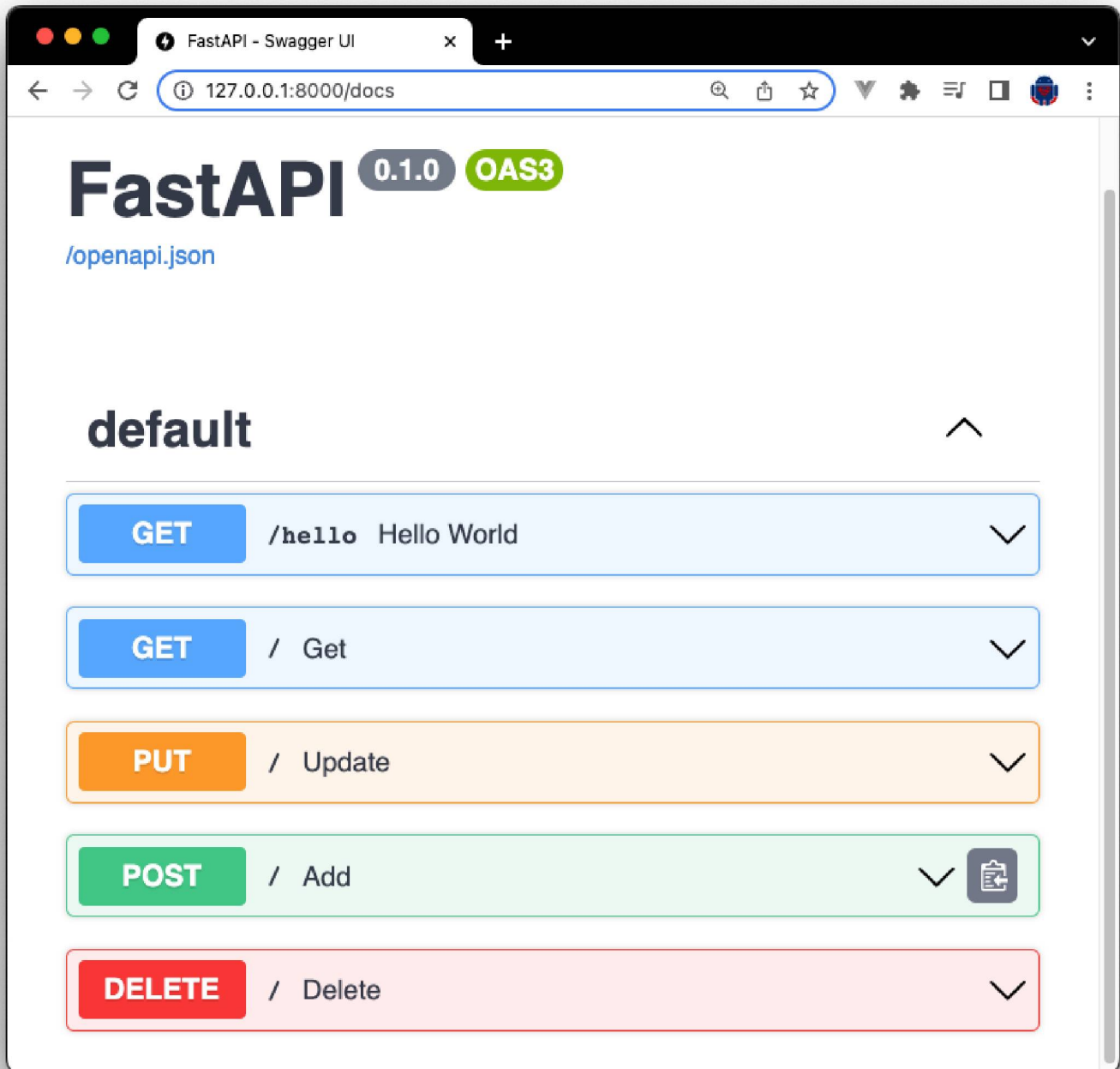
We can also add a prefix for this set of routes:

```
app.include_router(todo_router, prefix="/tasks")
```

From the following URL:

<http://127.0.0.1:8000/docs>

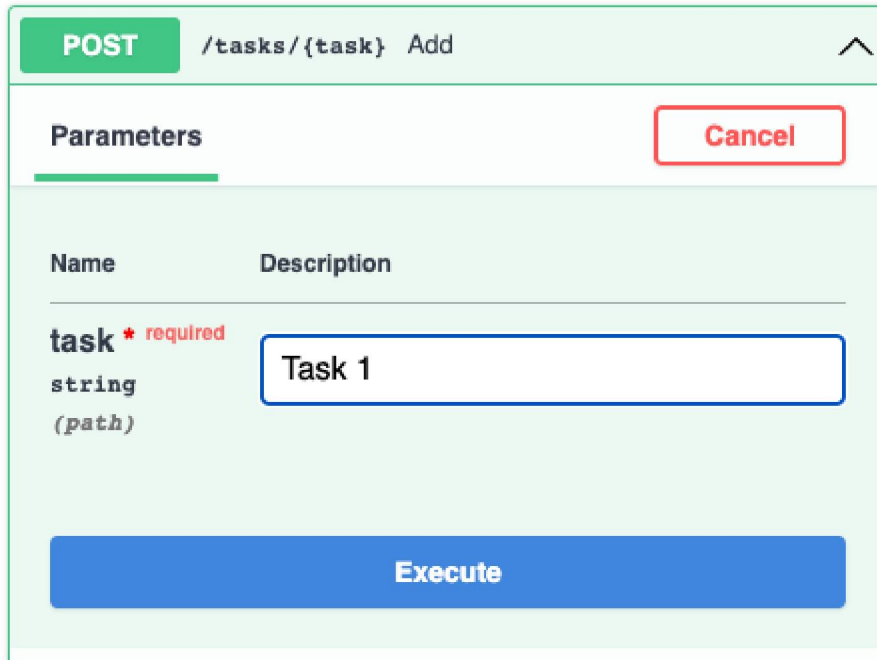
You can see all the rest methods or resources that you have created and even test the methods from that link, therefore, it is not necessary to use solutions like Postman since everything is integrated into the FastApi framework; currently, you should see the following:



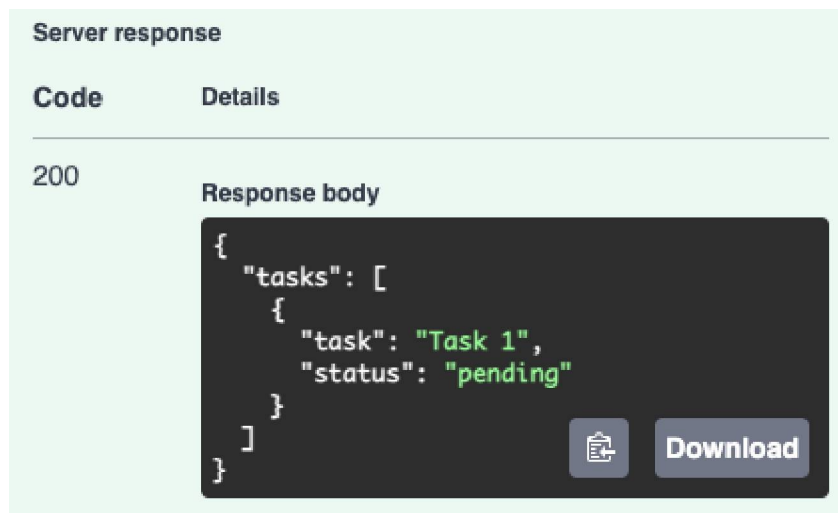
From this section, you can try the above resources:



Click on Try it out, and put the requested data (if requested):



And you send the request; you will see the answer:



It is also important to note that when defining the type, there are validations applied by default; in the previous case, we set that all the arguments must be of the integer type for the index and of the string type for the task; so, if for example a wrong data type is supplied; for example, a text instead of an integer, we will see an error like the following:

PUT
/tasks/ Update
^

Cancel

Name	Description
index * required integer <i>(query)</i>	<input style="width: 100%; border: 1px solid #f00;" type="text" value="abs"/>
task * required string <i>(query)</i>	<input style="width: 100%; border: 1px solid #ccc;" type="text" value="task"/>
status * required string <i>(query)</i>	<div style="border: 1px solid #ccc; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> done ▼ </div>

Please correct the following validation errors and try again.

- Value must be an integer

Execute

As homework, do several tests of the previous methods and evaluate the response, that is, the request and the created curl.

With this, you will have an exact reference of how to build your methods so that they have the behavior that you expect them to have.

Create a task (POST):

```
curl -X 'POST' \
  'http://127.0.0.1:8000/New%20Task' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://127.0.0.1:8000/New%20Task
```

Get all tasks:

```
curl -X 'GET' \  
  'http://127.0.0.1:8000/' \  
  -H 'accept: application/json'  
Request URL  
http://127.0.0.1:8000/
```

To update a task:

```
curl -X 'PUT' \  
  'http://127.0.0.1:8000/?task=Change%20Name%20Task&status=ready' \  
  -H 'accept: application/json'  
Request URL  
http://127.0.0.1:8000/?task=Change%20Name%20Task&status=ready
```

To delete a task:

```
curl -X 'DELETE' \  
  'http://127.0.0.1:8000/?index=0' \  
  -H 'accept: application/json'  
Request URL  
http://127.0.0.1:8000/?index=0
```

All the methods defined above use parameters via GET:

```
http://127.0.0.1:8000/?index=0
```

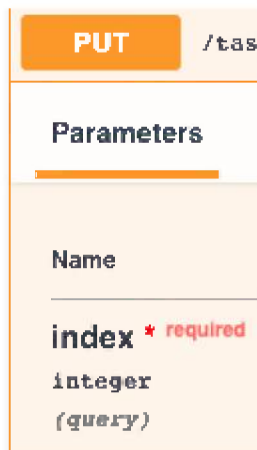
Except for the method to create a task; in which, the data (task to create) is defined directly in the URI:

```
http://127.0.0.1:8000/New%20Task
```

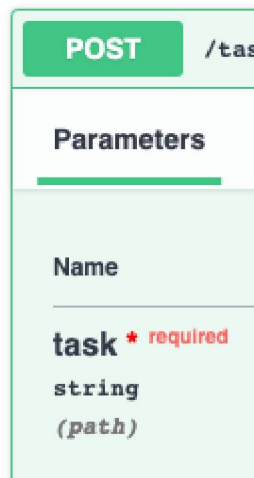
And this is because, in the case of creating the task, the argument is specified at the path level:

```
@task_router.post("/{task}")
```

Also, you can see specified in the parameter, the behavior of the data requested from the user, if it is through the query string:



Or be part of the path in the URL as in the case of the POST method:



And it is precisely because of the way in which we define the path for each method.

Limit allowed values (predefined data)

Many times we have some fields with fixed values, such as indicating a type of user, the status of a publication, or in this case, the status of the task; that can be, ready or pending; in Python, we have enumerated types that we can use for that purpose; we are going to create a new file in which we will place the status for the tasks:

models.py

```
from enum import Enum

class StatusType(str, Enum):
    DONE = "done"
    PENDING = "pending"
```

Then, to use it from the task CRUD, all we have to do is convert each position in the list to an enumerated type to receive both the task and the status; for example:

```
task_list.append({
```



```
'task':task,
'status': StatusType.PENDING
})
```

Having this clear, the next thing we do is adapt the **add()** and **update()** functions:

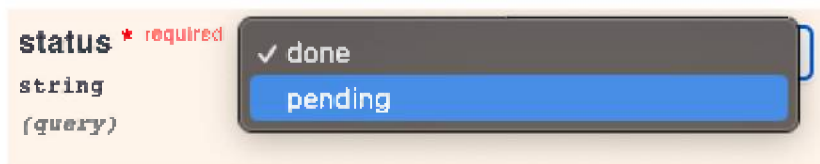
```
@task_router.post("/{task}")
def add(task: str):
    task_list.append({
        "task" : task,
        "status" : StatusType.PENDING,
    })
    return { "tasks": task_list }

@task_router.put("/")
def update(index: int, task: str, status: StatusType):
    task_list[index] = {
        "task" : task,
        "status" : status,
    }
    return { "tasks": task_list }
```

Note that in the function of **update()**, the enumerated type is received as an argument:

```
update(index: int, task: str, status: StatusType)
```

From the documentation section, we will see that when we are going to modify a task, we have a selection field with the allowed values:



Parameters with Query()

We also have access to more advanced validations through the **Query()** class in which we can indicate from minimum and maximum values, regular expressions and much more; for example, in the following function we can add validations in case of range limits:

```
from fastapi import ***, Query
@app.get("/page")
def page(page: int = Query(1, ge=1, le=20), size: int = Query(5, ge=5, le=20)):
    return {"page": page, "size": size}
```

With the **Query()** class we can indicate different arguments to perform numerical validations; in the example above, we use the options of:

- **gt** : Means greater than (Greater than).

- ge: It means greater than or equal to (Greater than or equal to).
- lt : Means less than (Less than).
- le: It means less than or equal to (Less than or equal to).

The first parameter corresponds to the default value that it will have; therefore, the parameters would be optional.

In the example above, de indicates that:

- **page** argument must be greater than or equal to 1 and less than or equal to 20.
- **size** argument must be greater than or equal to 5 and less than or equal to 20.

We can place other types of options as regular expressions:

```
@app.get("/phone/{phone}") # +34 111 12-34-56
def phone(phone: str =
Query(pattern=r"^(?+\[\d]{1,3}\)?\s?([\d]{1,5})\s?([\d][\s\.-]?){6,7}$" )):
    return {"phone": phone}
```

Including metadata:

```
@app.get("/page")
def page(page: int = Query(1, gt=0, title='Pagina a visualizar'), size: int = Query(10,
le=100)):
    return {"page": page, "size": size}
```

Remember that to use the **Query()** class the parameters do not have to be defined in the path since, to use the **Query()** class it means that the arguments are going to be passed through the query string, so if for the previous example, you were to place the parameter by the path:

```
@app.get("/phone/ {phone} ") # +34 111 12-34-56
def phone(phone: str = Query(pattern=r"^(?+\[\d]{1,3}\)?\s?([\d]{1,5})\s?
([\d][\s\.-]?){6,7}$" )):
    return {" phone ": phone }
```

When saving you would see an error like the following:

```
Cannot use `Query` for path param 'phone'
```

Path with Path()

As with the **Query()** class, we can define the same validation options, to indicate minimum or maximum values as well as regular expressions, with the difference that the attribute must be specified as part of the path and the **Path()** class receives no default values:

```
from fastapi import ***, Path
***
@app.get("/page/{page}")
def page(page: int = Path(gt=0), size: int = Query(10, le=100)):
    return {"page": page, "size": size}
```

It is important to define the attribute as part of the path since, if you do not place it:

```
from fastapi import ***, Path
***
@app.get("/page/")
def page(page: int = Path(gt=0), size: int = Query(10, le=100)):
    return {"page": page, "size": size}
```

When sending the request to the previous method, it will indicate an exception that indicates the Path type argument (**page** in this example) has not been found:

```
Code Details
422
Error: Unprocessable Entity

Response body
Download
{
  "detail": [
    {
      "loc": [
        "path",
        "page"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

Body

All the previous examples we have sent the data to the API through the URL, either through the path or the query string, if you have worked on other server-side web technologies, you should know that this is not the best way we have to pass data to the server when we want this data to be recorded persistently in the database or similar, the data that travels in the URL, are generally accompanied by a GET type request, when we want to persist this data or in general, change the data model (understand creating, updating or deleting something) POST, PUT, PATCH type requests are sent or DELETE and the data is encapsulated in the body of the request, that is, in the body.

The body is the part of the HTTP request that contains data to be processed by the server in a REST API, usually encoded in JSON and used to create or edit objects in a database; although, we are not going to persist this data yet, it is about time that we are going to correctly structure the data in the API methods according to the purpose of the API.

So, in a nutshell, to be able to send the data in the API via the body and not the URL, we can use the **Body()** class on the argument(s):

```

from fastapi import *** Body

@task_router.post("/{task} ")
def add(task: str = Body() ):
    task_list.append ({
' task':task ,
'status': StatusType.PENDING
})
print( task_list )
return task_list

```

In the terminal we will see an error like the following:

```

AssertionError: Cannot use `Body` for path param 'task'

```

Since, now the argument named task is set in the body and not as a parameter of the URL; so, it should be removed from the path:

```

@task_router.post("/")
def add(task: str = Body()):
    task_list.append({
        'task':task,
        'status': StatusType.PENDING
    })
    print(task_list)
    return task_list

```

For the update process, we make the corresponding changes, in the fields that we want the data to go in the request body, we use the **Body()** class:

```

@task_router.put("/")
def update(index: int, task: str = Body(), status: StatusType = Body()):
    task_list[index] = {
        "task" : task,
        "status" : status,

```

Now, if we go to the documentation, we will see that we have a field of type TEXTAREA to place the content:

The screenshot shows a REST client interface for a POST request to `/tasks/`. The interface includes a `POST` button, a `Cancel` button, and a `Request body` section with a `required` label and a dropdown menu set to `application/json`. The request body is a JSON object: `{ "id": 0, "name": "string", "description": "string", "status": "done" }`. An `Execute` button is at the bottom.

It is important that when placing the text you do not remove the quotes, since in this way it would not be treated as a text and you would see an error like the following:

```
curl -X 'POST' \  
  'http://localhost:8000/' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d 'Task Content'
```

Request URL

`http://localhost:8000/`

422

Error: Unprocessable Entity

Response body

Download

```
{
  "detail": [
    {
      "loc": [
        "body",
        0
      ],
      "msg": "Expecting value: line 1 column 1 (char 0)",
      "type": "value_error.jsondecode",
      "ctx": {
        "msg": "Expecting value",
        "doc": "Task Content",
        "pos": 0,
        "lineno": 1,
        "colno": 1
      }
    }
  ]
}
```

If you supply the argument correctly (with the quotes), you'll see a response like:

```
curl -X 'POST' \
  'http://localhost:8000/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '"Task Content"'
Request URL
http://localhost:8000/
```

Validation of request bodies using Pydantic models

Pydantic is a Python library for data validation; they are basically a series of classes where validations are defined and applied to the data that will be received in the user's request, therefore, the presentation of this data can be separated, as well as its structure from the methods that handle the request.

As mentioned earlier in the book, Pydantic is used by default as one of the core FastAPI libraries, so there is no need to install any additional packages as it is already built-in when you install FastAPI.

An example model is the following:

models.py

```
from pydantic import BaseModel
from enum import Enum

class StatusType(str, Enum):
    READY = "ready"
    PENDING = "pending"
```

```
class Task(BaseModel):
    id: int
    name: str
    description: str
    status: StatusType
```

It is a typical class in Python that inherits from **BaseModel** from the Pydantic library; the previous change must be done in the **models.py** file; now, we can simplify the API methods as follows:

task.py

```
@task_router.post("/")
def add(task: Task):
    # task_list.append({
    #     "task" : task.name,
    #     "status" : task.status,
    # })
    task_list.append(task)
    return { "tasks": task_list }

@task_router.put("/")
def update(index: int, task: Task):
    task_list[index] = {
        "task" : task.name,
        "status" : task.status,
    }
    return { "tasks": task_list }
```

As you can see, there is no need to receive multiple arguments anymore, just one of them, an instance of the model class defined before; It is important to note that this instance is supplied directly in the body, therefore it should not be defined in the path as follows:

```
@task_router.post("/{task}")
def add(task: Task):
    ***
```

Or you will see an error like the following:

Path params must be of one of the supported types

Validations

In the classes of the models it is possible to apply validations; for this, the validation decorator is used (indicating the field to be validated) on a function that receives the model and the value; from this function you can define different rules and return an exception in case the value does not satisfy the rules; let's see an example:

models.py

```

from pydantic import BaseModel, ValidationError, field_validator

class Task(BaseModel):
    id: int
    name: str
    description: str
    status: StatusType

    @field_validator('id')
    def greater_than_zero(cls, v):
        if v <= 0 :
            raise ValueError('must be greater than zero')
        return v

```

In the previous example, a validation rule is applied to the `id` field, which consists in the fact that the value must be greater than zero; the value returned to the function corresponds to the value that the API methods will receive; usually we will want to return the value received by the user as shown in the code above.

From the documentation, if we enter a value less than or equal to zero, we will see an error like the following:

```

422
Error: Unprocessable Entity

Response body
Download
{
  "detail": [
    {
      "loc": [
        "body",
        "id"
      ],
      "msg": "must be greater than zero",
      "type": "value_error"
    },
    {
      "loc": [
        "body",
        "name"
      ],
      "msg": "must be alphanumeric",
      "type": "assertion_error"
    }
  ]
}

```

That is, the exception that we defined above; you can create more methods on the same field:


```

class Task(BaseModel):
    """
    @field_validator('id')
    def id_greater_than_zero(cls, v):
        if v <=0 :
            raise ValueError('must be greater than zero')
        return v

    @field_validator('id')
    def id_less_than_a_thousand(cls, v):
        if v >1000 :
            raise ValueError('must be less less than a thousand')
        return v

```

Or other fields, for example, the name:

```

class Task(BaseModel):
    """
    @field_validator('name')
    def id_name_alphanumeric(cls, v):
        assert v.replace(" ", "").isalnum(), 'must be alphanumeric'
        return v

```

In this validation, it is indicated that the name can only have alphanumeric characters and blank spaces; use the **replace()** function to remove whitespace and then use the **isalnum()** function which returns **true** if the string to be evaluated is composed of alphanumeric characters.

You can get more information at:

<https://docs.pydantic.dev/latest/usage/validators/>

Nest classes

We can easily nest classes to represent more complete objects, for example, we can indicate that each task has an associated category as follows:

```

class Category(BaseModel):
    id: int
    name: str

class Task(BaseModel):
    """
    category: Category

```

Or for example, a user who owns the task and the category:

```

class User(BaseModel):
    id: int

```

```

    name: str
    surname: str
    email: str

class Category(BaseModel):
    id: int
    name: str

class Task(BaseModel):
    """
    user: User
    category: Category

```

When creating or updating a task, we will see that the relationships are now part of the request:

```

{
  "id": 0,
  "name": "string",
  "description": "string",
  "status": "done",
  "category": {
    "id": 0,
    "name": "string"
  },
  "user": {
    "id": 0,
    "name": "string",
    "surname": "string",
    "email": "string"
  }
}

```

Inheritance

To keep these validation classes more modular, we can use basic primitives like inheritance; currently, we have 3 entities, tasks, users and categories, each one can handle an ID with the validations we defined before; thinking about this, we will create a base class like the following:

models.py

```

"""
class MyBaseModel(BaseModel):
    id: int
    @field_validator('id')
    def greater_than_zero(cls, v):
        if v <= 0 :
            raise ValueError('must be greater than zero')
        return v

```

```

@field_validator('id')
def less_than_a_thousand(cls, v):
    if v >1000 :
        raise ValueError('must be less less than a thousand')
    return v

```

And now, we inherit from this new class for each of the entities defined:

models.py

```

***
class User(MyBaseModel):
    id: int
    name: str
    surname: str
    email: str

class Category(MyBaseModel):
    id: int
    name: str

    # validaciones de nombre e email TODO

class Task(MyBaseModel):
    name: str
    description: str
    status: StatusType
    user: User
    category: Category

    @field_validator('name')
    def name_alphanumeric(cls, v):
        assert v.replace(" ", "").isalnum(), 'must be alphanumeric'
        return v

```

Validation by fields

As we saw with the use of the **Query()** or **Path()** classes, we can apply the same validations from the model classes; for example:

models.py

```

from pydantic import Field

class StatusType(str, Enum):
    READY = "ready"
    PENDING = "pending"

```

```

class MyBaseModel(BaseModel):
    id: int = Field(ge=1, le=100)
    ***

class User(MyBaseModel):
    name: str = Field(min_length=3)
    ***

```

These values are purely for tests and have no real use for the application we are building, therefore, you can do without these validations or customize them to your liking.

Optional values and default values

Another important factor is that, many times we have parameters that are optional, for example, for the task application, the description can be optional; so, thinking about that, we have:

models.py

```

from typing import Optional
***
class Task(MyBaseModel):
    ***
    description: Optional[str]

```

Or if you have validations:

```

from typing import Optional
***
class Task(MyBaseModel):
    ***
    description: Optional[str] = Field("No description", min_length=3)

```

Or if you want to set a default value when it is not specified at the time of making the request:

```

from typing import Optional
***
class Task(MyBaseModel):
    ***
    description: Optional[str] = Field("No description", min_length=3)

```

If you make a request without sending the description, you will see that it takes the default value:

```

curl -X 'POST' \
  'http://localhost:8000/tasks/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{

```

```

    "id": 100,
    "name": "string",
    "status": "done",
    "category": {
      "id": 100,
      "name": "string"
    },
    "user": {
      "id": 100,
      "name": "string",
      "surname": "string",
      "email": "user@example.com",
      "website": "string"
    }
  }'
{
  "tasks": [
    {
      "id": 100,
      "name": "string",
      "description": "No description",
      "status": "done",
      "category": {
        "id": 100,
        "name": "string"
      },
      "user": {
        "id": 100,
        "name": "string",
        "surname": "string",
        "email": "user@example.com",
        "website": "string"
      }
    }
  ]
}

```

Otherwise, if you put a valid description, you will see the set value:

```

curl -X 'POST' \
  'http://localhost:8000/tasks/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 100,
    "name": "string",
    "description": "string",

```

```

***
}'

200
Response body
Download
{
  "tasks": [
    {
      "id": 100,
      "name": "string",
      "description": "string",
      ***
    }
  ]
}

```

If you put an invalid description, in this example, because of the length, you will see the exception:

```

curl -X 'POST' \
  'http://localhost:8000/tasks/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 100,
  "name": "string",
  "description": "bu",
  ***
}'

422
Error: Unprocessable Entity

Response body
Download
{
  "detail": [
    {
      "loc": [
        "body",
        "description"
      ],
      "msg": "ensure this value has at least 5 characters",
      "type": "value_error.any_str.min_length",
      "ctx": {
        "limit_value": 5
      }
    }
  ]
}

```

```
}  
}'
```

You can also specify the default value, which is null:

```
from typing import Optional  
***  
class Task(MyBaseModel):  
    ***  
    description: Optional[str] = Field(None, min_length=3)
```

Field types

Many times we have to place validations for specific formats such as IPs, JSON or email; for these cases, there are already classes that we can apply directly to the properties of the class:

models.py

```
from pydantic import EmailStr, HttpUrl  
***  
class User(MyBaseModel):  
    ***  
    email: EmailStr  
    website: HttpUrl
```

If we see an error like the following:

```
email-validator is not installed, run `pip install pydantic[email]
```

You must install the additional package:

```
$ pip install pydantic[email]
```

If it gives you an exception:

```
no matches found: pydantic[email]
```

Try putting some quotes:

```
$ pip install 'pydantic[email]'
```

Now with this, we have access to additional validations and in case of not passing the validations, we will have an error like the following:

```
{  
  "detail": [  
    {  
      ***
```

```

{
  "loc": [
    "body",
    "user",
    "email"
  ],
  "msg": "value is not a valid email address",
  "type": "value_error.email"
},
{
  "loc": [
    "body",
    "user",
    "website"
  ],
  "msg": "invalid or missing URL scheme",
  "type": "value_error.url.scheme"
},
}

```

You have the complete list of validations that you can use in:

<https://docs.pydantic.dev/latest/usage/types/#pydantic-types>

Use a list as an argument

Another nice feature is being able to set a list of values or objects as part of the model class; for it:

models.py

```

from typing import List
***
class Task(MyBaseModel):
    name: str
    description: Optional[str] = Field("No description", min_length=3)
    status: StatusType
    user: User
    category: Category
    tags: List[str] = []

```

With this, we can pass a list of values:

```

{
  ***
  "tags": ["tag 1", "tag 2"]
}

```

If you want the values to be unique and not repeatable, you can use a `set()`:


```
class Task(MyBaseModel):
    name: str
    description: Optional[str] = Field("No description", min_length=3)
    status: StatusType
    user: User
    category: Category
    # tags: List[str] = []
    tags: set[str] = set()
```

If we pass the following data:

```
{
    ***
    "tags": ["tag 1", "tag 2", "tag 3", "tag 1"]
}
```

We will get the following list with unique values if the list was defined with the function **set()**:

```
***
"tags": [ "tag 1", "tag 3", "tag 2" ]
```

If the list was defined by the **List** class:

```
***
"tags": ["tag 1", "tag 2", "tag 3", "tag 1"]
```

FastAPI automatic documentation

As we have seen in the previous examples, FastAPI generates the documentation automatically for the routes that we have defined in the application. From these routes, we can make requests, indicating parameters to the routes, body and much more; but, currently there are two types of documentation that we can use:

Swagger

Here's the documentation we've used so far by:

<http://127.0.0.1:8000/docs>

And it provides an interactive environment to test our API, it does not matter what type of request has been defined that can be tested from here, so we have both the definition of the different routes of the application and also the option of being able to test the routes.

ReDoc

Like the previous one, this version offers an environment to be able to see the definition of the different routes as well as test the API; although, it could be said that it offers a more detailed and easily accessible interface for each of the API methods; you can try this version of the documentation from:

<http://127.0.0.1:8000/redoc>

The main difference between the two documentations is that in the redoc one we don't have the "Try it out" button to test the methods.

Conclusions

Each one of the sections that we saw before represent the different ways that exist in FastApi to do both the routing and pass the data to the Api; you can mix these solutions and adapt the one that best suits the method or methods of your API; we have seen different schemes to define the arguments, either through the path, the query string, the body, validations, default values, optionals and many other schemes.

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.1>

Capítulo 4: Errors and status codes

In this section, you will learn how to handle the HTTP status codes of the API methods and also how to handle errors/exceptions from the methods.

HTTP status codes

Status codes are codes issued by a server in response to a client request; there are many types of codes, but we can classify them into five categories, each of which indicates a different response:

- 1XX: The request has been received
- 2XX: The request was successful
- 3XX: Redirected request
- 4XX: There is a client error
- 5XX: There is a server error

The most used are the type 200, 400 and 500; for type 200 codes, among the main ones we have:

- 200 Ok
- 201 Created
- 202 Accepted
- 204 No content

For the 400:

- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed

Customize response status code in methods

We can use these codes to customize the code in the methods response; for this we use the **status_code** option on the API routes:

```
@task_router.post("/", status_code=status.HTTP_201_CREATED)
```

In FastAPI we have access to the **status** object with which we can obtain the references to the codes; among the main ones we have:

```
from fastapi import status

status.HTTP_200_OK
status.HTTP_201_CREATED
status.HTTP_204_NO_CONTENT
status.HTTP_403_FORBIDDEN
status.HTTP_404_NOT_FOUND
```

These constants stored in the status object are ultimately fixed values; for example, if you do a screen print of:

```
status.HTTP_200_OK
```

You will see:

```
200
```

Therefore you can either use the constants or directly the integer value. With this cleared up, let's see how we can use these status codes in the responses of the tasks' CRUD methods; to return a result, we can return a 200, which the code used by default:

task.py

```
from fastapi import ***, status
***
@task_router.get("/", status_code=status.HTTP_200_OK)
def get():
    return { "tasks": task_list }
```

You can also directly return the status code:

task.py

```
@task_router.get("/", status_code=200)
def get():
    return { "tasks": task_list }
```

To create, we already have a status code that we could use; which would be 201; so:

task.py

```
@task_router.post("/", status_code=status.HTTP_201_CREATED) #status_code=201
status.HTTP_200_OK
def add(task: Task):
    task_list.append(task)
    return { "tasks": status.HTTP_201_CREATED}
```

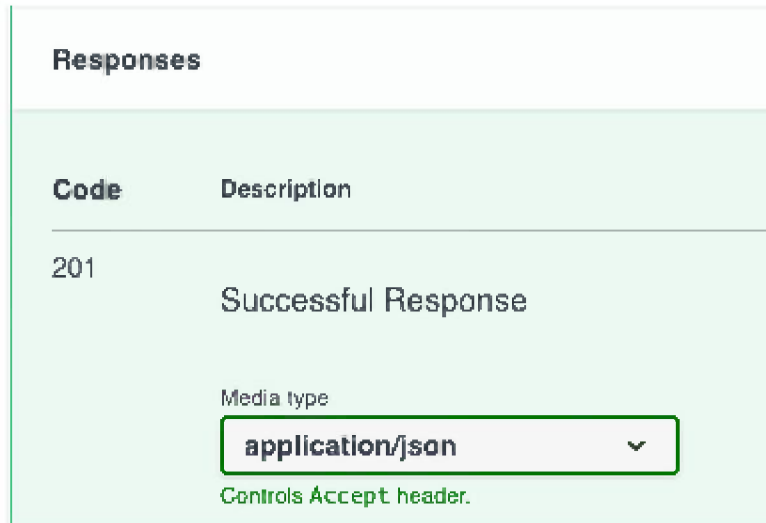
To delete or update, since they are operations that usually perform the action and do not return any content (although in these methods for our API we are returning values), we can use code 200 or 204:

task.py

```
@task_router.put("/", status_code=status.HTTP_200_OK) #status_code=200
def update(index: int, task: Task):
    task_list[index] = {
        "task" : task.name,
        "status" : task.status,
    }
    return { "tasks": task_list }
```

```
@task_router.delete("/", status_code=status.HTTP_200_OK) #status_code=200
def delete(index: int):
    del task_list[index]
    return { "tasks": task_list }
```

When making the requests successfully, you will see the established status code; for example:



Code	Description
201	Successful Response

Media type

application/json ▼

Controls Accept header.

Error handling

In the previous section, we saw how to customize the status codes for each of the methods that make up the application, we learned what status codes are and how they are useful to inform the client about the status of the request, but let's go to the next step; many times it is necessary to do extra validations on the data received or from the methods to start another process and all this at some point can fail and this is where we place the conditionals or similar to control a possible error and avoid a fatal exception in the response; some examples of errors are:

1. Connect to an external resource and it does not respond or responds with an unexpected response.
2. Access non-existent pages or resources.
3. Pages protected by authentication or similar.
4. Problems when processing user data.

This is to give an example, but there are many other schemes in which problems can occur.

What is an HTTP exception?

An HTTP exception is an event that is used to indicate a failure or a problem when processing the request, it can be the lack of data, its format or similar; in the case of our application, it may be due to passing an incorrect index when updating a task.

Errors in FastAPI are handled by raising an exception using the **HTTPException** class provided by the framework itself; let's see some examples.

Evaluate that the index is not out of range

We can create a condition to test if the supplied index is out of range and return an exception if the supplied index is out of range:

task.py

```
from fastapi import ** HTTPException
**
@task_router.put("/", status_code=status.HTTP_200_OK) #status_code=200
def update(index: int, task: Task):

    #verificamos que no este fuera de rango el indice
    if len(task_list) <= index:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Task ID doesn't exist",
        )

    task_list[index] = {
        "task": task.name,
        "status": task.status,
    }
    return { "tasks": task_list }
```

And even, we can create a new query method for the index:

task.py

```
@task_router.get("/{index}", status_code=status.HTTP_200_OK) #status_code=200
def get(index: int):

    #verificamos que no este fuera de rango el indice
    if len(task_list) <= index:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Task ID doesn't exist",
        )

    return { "tasks": task_list[index] }
```

Evaluate that the task has not been registered

As another possible validation, we are going to prevent the user from adding duplicate tasks (with the same name):

```
@task_router.post("/", status_code=status.HTTP_201_CREATED) #status_code=201
status.HTTP_200_OK
def add(task: Task):
```

```
#verificamos que la tarea no haya sido asignada
if task in task_list:
    raise HTTPException(
        status_code=status.HTTP_400_BAD_REQUEST,
        detail="Task " + task.name + " already exist",
    )
```

Conclusion

Changing the response status codes may seem unnecessary, but it is a key factor that is taken advantage of when consuming the response, since it will be much easier for the application that is going to consume our API to evaluate the code of status and from the response, that instead of evaluating the response directly, indicating the correct status code for each of the API methods, are good practices that we must follow.

Ultimately, status codes are useful to inform the client about the status of the response.

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.2>

Chapter 5: Declare sample data

*** This chapter is under development, currently it shows how to use the sample data for Pydantic V1, the Pydantic 2 version brings changes in the structure of the sample data.

Being able to configure some test data is ideal when our entities are growing in size; this saves us a few seconds each time we test the API with the automatic documentation.

Sample data in models

To create test data using a model we have the following structure:

models.py

```
class Task(MyBaseModel):
    """
    model_config = {
        "json_schema_extra": {
            "examples": [
                {
                    "id" : 123,
                    "name": "Salvar al mundo",
                    "description": "Hola Mundo Desc",
                    "status": StatusType.PENDING,
                    "tag":["tag 1", "tag 2"],
                }
            ]
        }
    }
```

With this, we will see from the documentation when creating (or similar processes) the previously defined format:

POST
/tasks/ Add

Parameters

No parameters

Request body required

Example Value
Schema

```

{
  "id": 123,
  "name": "Salvar al mundo",
  "description": "Hola Mundo Desc",
  "status": "pending",
  "tag": [
    "tag 1",
    "tag 2"
  ]
}

```

Nested relations

If there are relationships in the entity, as in our case with the task relationship, just specify them as you can see in the following example:

models.py

```

class Task(MyBaseModel):
    """
    model_config = {
        "json_schema_extra": {
            "examples": [
                {
                    "id" : 123,
                    "name": "Salvar al mundo",
                    "description": "Hola Mundo Desc",
                    "status": StatusType.PENDING,
                    "tag":["tag 1", "tag 2"],
                    "category": {
                        "id":1234,
                        "name":"Cate 1"
                    }
                }
            ]
        }
    }

```

```

    },
    "user": {
      "id":12,
      "name":"Andres",
      "email":"admin@admin.com",
      "surname":"Cruz",
      "website":"http://desarrollolibre.net",
    }
  ]
}

```

And we will have:

POST /tasks/ Add

Parameters

No parameters

Request body required

Example Value | Schema

```

{
  "id": 123,
  "name": "Salvar al mundo",
  "description": "Hola Mundo Desc",
  "status": "pending",
  "tag": [
    "tag 1",
    "tag 2"
  ],
  "category": {
    "id": 1234,
    "name": "Cate 1"
  },
  "user": {
    "id": 12,
    "name": "Andres",
    "email": "admin@admin.com",
    "surname": "Cruz",
    "website": "http://desarrollolibre.net"
  }
}

```

Option list

We can also declare a list of values to select; in this list, you can place various states, both valid and invalid states; for listings, the definition of the sample data is done directly in the argument of the API methods; for example:

```

@task_router.put("/product2/")
async def update_product(
    item: Task=

```

```

Body(
  examples={
    "normal": {
      "summary": "A normal example",
      "description": "A normal item works correctly.",
      "value": {
        "name": "Foo",
        "description": "A very nice Item",
        "price": 35.4,
        "tax": 3.2,
      },
    },
    "converted": {
      "summary": "An example with converted data",
      "description": "FastAPI can convert price `strings` to actual `numbers` automatically",
      "value": {
        "name": "Bar",
        "price": "35.4",
      },
    },
    "invalid": {
      "summary": "Invalid data is rejected with an error",
      "value": {
        "name": "Baz",
        "price": "thirty five point four",
      },
    },
  },
),
):
results = { "item": item}
return results

```

The first option corresponds to a descriptive name that you want to place; the **summary** options is the text that appears in the SELECT and the **description** is a descriptive text; where you have to place the values is in the **value** option; finally, we will have:

Examples:

- ✓ A normal example 1
- A normal example 2
- A invalid example 1

Sample data from argument

We can also declare the sample data from the API methods:

```
@task_router.put("/", status_code=status.HTTP_200_OK)
def update(index: int, task: Task = Body(
    example= {
        "id" : 123,
        "name": "Salvar al mundo 2",
        "description": "Hola Mundo Desc",
        "status": StatusType.PENDING,
        "tag":["tag 1", "tag 2"],
        "category": {
            "id":1234,
            "name":"Cate 1"
        },
        "user": {
            "id":12,
            "name":"Andres",
            "email":"admin@admin.com",
            "surname":"Cruz",
            "website":"http://desarrollolibre.net",
        }
    }
)):
    ***
```

Sample data from argument

In this section, we'll see how to add the sample data directly from the API function arguments.

Simple example

We can also declare test data for **Body()**, **Path()** and **Query()**; for example:

```
@app.get("/ep_phone/{phone}") # +34 111 12-34-56
def phone(phone: str =
    Path(pattern=r"^(?!\+[\d]{1,3}\?)\s?([\d]{1,5})\s?([\d][\s\.-]?){6,7}$", example="+34 111
12-34-56")):
    return {"phone": phone}
```

Or

```
@app.get("/e_phone/") # +34 111 12-34-56
def phone(phone: str =
    Query(pattern=r"^(?!\+[\d]{1,3}\?)\s?([\d]{1,5})\s?([\d][\s\.-]?){6,7}$", example="+34
111 12-34-56")):
    return {"phone": phone}
```

Option list

We can also declare a list of examples (important to note that the values option has no {}):

```
@app.get("/ep_phone/{phone}") # +34 111 12-34-56
def phone(phone: str =
Path(pattern=r"^(?!\+[\d]{1,3}\?)\s?([\d]{1,5})\s?([\d][\s\.-]?)\{6,7\}$" , examples={
    "normal":{
        "summary":"A normal example 1",
        "description":"A normal example",
        "value":
            "+34 111 12-34-56"

    },
    "normal 2":{
        "summary":"A normal example 2",
        "description":"A normal example",
        "value":
            "+34 111 12-34-59"
    }
}))):
    return {"phone": phone}
```

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.3>

Chapter 6: File Upload

File upload is a highly required functionality in many applications to upload all kinds of documents; this is a function that we can easily implement, but in order to use file upload, we must install the following package in our project:

```
$ pip install python-multipart
```

We create a new file to handle the different routes that we will see in this chapter:

myupload.py

```
from fastapi import APIRouter, File
upload_router = APIRouter()
```

```
@upload_router.post("/file")
def upload_file(file: bytes = File()):
    return { "file_size" : len(file) }
```

And we import from the main:

api.py

```
from myupload import upload_router
***
app.include_router(upload_router, prefix='/upload')
```

There are two classes to upload files in FastApi that we can use, the simplest solution using the **File** class and the most complete solution using the **UploadFile** class; let's look at both implementations and the features of each.

File

The minimum code to upload an image or a file is as follows:

myupload.py

```
from fastapi import File
***
@upload_router.post("/file")
def upload_file(file: bytes = File()):
    return { "file_size" : len(file) }
```

With the above code, a file is loaded as a **byte** array; therefore, we cannot do additional operations such as verify file data such as name, extension, among others; in the example above, the size of the file is simply returned using the **len()** function.

You will see in the documentation:

Request body **required**

file * **required**

string(\$binary)

Seleccionar archivo

Captura de pantall...-03-22 062639.png

And you will have as output, the size in bytes of the file:

200

```
{
  "file_size": 161206
}
```

UploadFile

One drawback of the above approach, which the **File** class uses, is that the uploaded file is stored entirely in memory. So while it will work with small files, you will likely have problems with larger files. Also, manipulating a **bytes** object is not always convenient or easy for file handling.

To get around this problem, FastAPI provides a class called **UploadFile**. This class will store the data in memory up to a certain size limit, but if the file is very large, it is automatically stored on disk in a temporary location (Python's **SpooledTemporaryFile** object); with this, we can safely and efficiently handle large files without worrying about out-of-memory exceptions occurring in the application. Also, with the instance of the file type object, we can use the file's metadata such as its name and extension, in addition to having a file instance ready to save to a folder or any other location; this object has the following attributes:

- **filename**: Name of the uploaded file.
- **content_type**: Type of content of the file. For example, an image file image/jpeg.
- **file**: A Python file object.

As a minimal example we have:

myupload.py

```
from fastapi import UploadFile

import shutil

@upload_router.post("/uploadfile1")
def upload_uploadfile1(file: UploadFile):
    return {
        "filename" : file.filename,
        "content_type" : file.content_type,
    }
```

You will see in the documentation:

```
{
  "filename": "12_overflow_seleccionada.png",
  "content_type": "image/png"
}
```

Save file

As we introduced at the beginning, the object of type **UploadFile** has an attribute of type **file** that refers to a Python file; so, if we want to save the file on disk (in an application folder), we can do something like the following:

myupload.py

```
@upload_router.post("/uploadfile2")
def upload_uploadfile2(file: UploadFile):

    with open("image.png","wb") as buffer:
        shutil.copyfileobj(file.file, buffer)

    return {
        "filename" : file.filename
    }
```

In Python, the **with** statement is used in exception handling to make code cleaner and much more readable. It simplifies the management of common resources such as the file stream in this case, in the previous example to create (or open if it exists) a file in write mode in binary format.

If you want to register the image (or file) in a folder (which has to exist):

myupload.py

```
@upload_router.post("/uploadfile2")
def upload_uploadfile2(file: UploadFile):

    with open("img/image.png","wb") as buffer:
        shutil.copyfileobj(file.file, buffer)

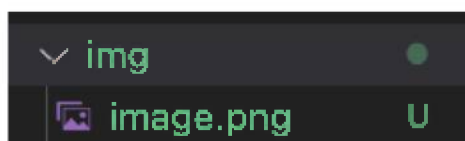
    return {
        "filename" : file.filename
    }
```

If the folder does not exist, you will see an error like the following:

```
FileNotFoundError: [Errno 2] No such file or directory: 'img/image.png'
```

The **shutil.copyfileobj()** method in Python is used to copy the contents of an object to a file.

Finally, when we load the image we will have:



Handle multiple files

To handle multiple files, we must define the parameter in the function of type **List[UploadFile]**; with this, using a loop we can process each file independently:

```
from typing import List
```



```
@upload_router.post("/uploadfile3")
def upload_uploadfile3(images: List[UploadFile] = File()):

    for image in images:
        with open("img/"+image.filename,"wb") as buffer:
            shutil.copyfileobj(image.file, buffer)
```

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.4>

Chapter 7: Database

Usually when this type of application is created in which data is to be managed (in this case, tasks) a database is used to save all this data persistently; at the moment, you will see that when the browser is reloaded or the server is stopped, all the data (tasks) are lost, since persistent storage is not being used and this is where the databases come in as a very common solution to solve these kinds of problems. With FastAPI, we can use both relational databases, such as MySQL or PostgreSQL, or NoSQL databases, such as MongoDB. Because the most common schema is used, we are going to learn how to use a database, for example in MySQL with FastAPI.

SQLAlchemy is an Object Relational Mapping (ORM) library in Python that is used to interact with relational databases in an easy way; using this library, we can make most of the queries to the database, since it allows us to work with objects in Python instead of direct SQL queries, from the selection, insertions, updates and deletions of records; in addition to being able to create the tables in the database using Python classes and being an ORM, it automatically maps the records from the queries to objects, or list of objects in case the query returns more than one result.

SQLAlchemy also supports the use of the most widely used database engines, such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, SQLite, among others, but for this you must install the corresponding connector. This is the library that we are going to use together with FastAPI to store the data persistently.

Install dependencies

We are going to install the necessary dependencies to carry out this chapter, which would be SQLAlchemy ORM (Object Relational Mapper) and the connector for MySQL:

```
$ pip install sqlalchemy mysql-connector-python
```

We will create a MySQL database either through a manager or through SQL:

```
$ mysql -u root -p
>> CREATE DATABASE tasks;
```

With this, we complete the basic configuration in the project.

Configure connection

We will create a new file to carry out the basic configurations to be able to use the database created in the previous section:

database/database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "mysql+mysqlconnector://root:root@localhost:3306/tasks"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Let's explain the above code:

- To connect to the database using the **create_engine()** function.
- Using the **sessionmaker()** function, the session is created.
 - The **autoflush** and **autocommit** of the database are disabled and through the **bind**, we bind the database engine to the session.

The **declarative_base()** function allows you to create a class instance that we will use to create the models and with this, the tables of the application's database, although we will deal with creating these models in the next section.

Remember to place the credentials corresponding to your database following the following scheme:

```
DATABASE_URL =  
"mysql+mysqlconnector://<USUARIO>:<CONTRASENA>@<HOST>:<PUERTO>/<BASEDEDATOS>"
```

Although it may seem like a somewhat abstract and gimmicky setup, this is a typical setup for connecting to a database using SQLAlchemy; with this configuration, you can connect to other engines such as PostgreSQL, SQLite, among others; although, several changes to the application still need to be made.

Changes in models

Let's rename the file to:

models.py

To

schemas.py

This is not something you must do, but it is a common structure that you will find in other projects that use FastAPI together with SQLAlchemy.

The next change to make is to add the Pydantic **from_attributes** property which will tell the Pydantic model to read the data as a dictionary and as an attribute:

schemas.py

```
class Task(MyBaseModel):  
    class Config:  
        from_attributes = True
```

Create models (tables) of the database

The disadvantage that we have when using SQLAlchemy with FastAPI is that we must have a duality between the models (which are going to be used to convert into tables in the database) and the model classes defined in previous chapters to define each one of the arguments requested in the request, as well as their validations among other characteristics; these classes still need to be used in the application to fulfill the same purpose they have been serving.

Now that the application is connected to the MySQL database, we are going to create the tables, or rather, the structure of the tables, for this, models are used, specifically Python classes that inherit from the **Base** class:

```
Base = declarative_base()
```

Defined above; each class is equivalent to a table and each property of the class is represented by a Column in the database.

database/models.py

```
from sqlalchemy.schema import Column
```

```

from sqlalchemy.types import String, Integer, Text, Enum

from database.database import Base
from schemes import StatusType

class Task(Base):
    __tablename__ = "tasks"
    id=Column(Integer, primary_key=True, index=True)
    name = Column(String(20))
    description = Column(Text())
    status = Column(Enum(StatusType))

```

The above code is very easy to understand, due to its organization it is self-explanatory; to create the SQLAlchemy models, the **Base** class is imported from **database.py**. Then, the **__tablename__** attribute is added to indicate the name of the table in the database, telling SQLAlchemy what name to use in the database for the model.

We import the **Column()** function which receives the type; for example, **Integer**, **String**, **Text**, or **Boolean**.

To receive unique data, we add the unique parameter, to indicate that it is a primary key, the **primary_key** argument is used.

You can learn more about SQLAlchemy at:

<https://docs.sqlalchemy.org/en/20/orm/quickstart.html>

Use the connection and create the tables in the database

With the above code, nothing has been created in the database so far, it's time to make the first connection and test all the changes made previously; to do this, we are going to go to the startup file and indicate the creation of the tables:

api.py

```

from database.database import Base, engine
from database.models import Task

app = FastAPI()
router = APIRouter()

Base.metadata.create_all(bind=engine)

```

And the connection:

database/database.py

```

***
Base = declarative_base()

```

```
def get_database_session():
    try:
        db = SessionLocal()
        yield db
    finally:
        db.close()
```

This line of code is essential since it allows you to create all the tables in the database:

```
Base.metadata.create_all(bind=engine)
```

Once the above line is defined, you must stop the server (Control/Command + C in the terminal) and start the server again, as soon as you start the server, the tables should be created.

You can use other schemas like [Alembic](#) to create your tables in the database, but to avoid adding additional complexity, we use a simpler and more direct but less scalable approach; for real projects, you should consider using a toolkit like the one above.

It is necessary to import the **Task** model, before executing the **create_all(bind=engine)** and the **tasks** table can be created in the database as we showed in the previous implementation:

```
from database.models import Task
```

Finally, the **get_database_session()** function will handle the connection to the database; from this function will create and close the session in all routes of the application; you could define this function in the **app.py**, which is the main file of the application, but, since we want to use the database connection in multiple files (paths defined in multiple files), we will leave this function in the **database.py**.

Continuing with the definition of the **get_database_session()** function, it may seem somewhat confusing to use **yield** instead of a **return**; although **yields** have many implementations in Python and a functioning that can be considered similar to that of **return**, the fundamental difference is that the **yield** will only execute the code until the **yield** is used; in the case of the above code, it would be just up to the assignment:

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Here you can see some differences with respect to **return**; if you put a **return** in its place, the session would be closed before doing the query; later, after we finish configuring the application, we will see an example of this.

An important factor is that the value returned in the **yield (db)** is injected into all the routes as we will show a little later:

```
def get_db():
    db = SessionLocal()
```

```
try:
    yield db
finally:
    db.close()
```

The important thing to note here is that the code that follows the statement (where the **yield** is) is executed after the response has been delivered:

```
async def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

And then a new session will be created for the database for the next request.

You can get more information at:

<https://fastapi.tiangolo.com/tutorial/dependencies/dependencies-with-yield/>

Inject database session dependency in routes

Now is the time to be able to use the connection to the database previously made from the API, that is, from each of the API methods that make up the application; to do this, we must inject a dependency in each of these methods:

```
from fastapi import *** Depends
from sqlalchemy.orm import Session
***
@app.<TYPE>("<URI>")
def page(db: Session = Depends(get_database_session)):
    ***
```

The operation is as follows, we need to have a separate database session/connection (**SessionLocal**) per request, the same session is used on all client requests and then closed after the request is finished (when sending response to client).

This dependency injection must be done in each API method that we want to use the connection to the database, leaving as:

api.py

```
from fastapi import *** Depends
from sqlalchemy.orm import Session

from database.database import get_database_session
***
```

```
@app.get("/page/")
def page(db: Session = Depends(get_database_session)):
    get_task(db,1)
    #create_user(db)
    return {"page": 1}
```

And for the crud of the tasks:

database/crud.py

```
from fastapi import *** Depends
from sqlalchemy.orm import Session

from database.database import get_database_session
***

@task_router.get("/", status_code=status.HTTP_200_OK)
def get(db: Session = Depends(get_database_session)):
    ***

@task_router.post("/", status_code=status.HTTP_201_CREATED)
def add(task: Task = Body(
    examples={***}), db: Session = Depends(get_database_session)):
    ***

@task_router.put("/", status_code=status.HTTP_200_OK)
def update(index: int, task: Task = Body(
    example= {***}
), db: Session = Depends(get_database_session)):
    ***

@task_router.delete("/", status_code=status.HTTP_200_OK)
def delete(index: int, db: Session = Depends(get_database_session)):
    ***
```

What is the Depends class used for?

The **Depends** class is used to inject dependency into FastAPI applications. The **Depends** class takes a source as an argument, just like a function is, and is passed as an argument in API methods, specifically in the route.

Demo: yield vs return

As we commented in the previous section, we have yet to make an example of the use of the **yield** with respect to the **return** and with this, understand its inclusion in the project in a more practical way; for this, we start from the following code:

database/database.py

```
def get_database_session():
    print("*****Init DB")
    try:
        db = SessionLocal()
        yield db
        # return db
    finally:
        print("*****End app and DB")
        db.close()
```

You must comment/uncomment the **yield/return** for each test.

With **yield**:

```
*****Init DB
test
INFO:      127.0.0.1:57852 - "GET /page2/ HTTP/1.1" 200 OK
*****End request and DB
```

With **return**:

```
*****Init DB
*****End request and DB
test
INFO:      127.0.0.1:57852 - "GET /page2/ HTTP/1.1" 200 OK
```

As you can see, when using the **yield**, the connection is closed when the function ends and therefore, when returning the response, in the case of using **return**, it is closed as soon as the connection instance (**db**) is returned.

The important thing to note is that when using the **yield**, the session to the database is created when the request is received; that session to the database is used to resolve the request (for example, to create a task in the database) and then the session to the database is closed after the request completes (when the response is sent to the client).

CRUD methods for tasks, database

We are going to show the implementation of the CRUD using SQLAlchemy, the operations are self-explanatory, in the same way after this code snippet, a summary of the operations performed is shown:

database/task/crud.py

```
from sqlalchemy.orm import Session

from schemes import Task
from database import models

def getById(db: Session, id: int):
```



```

# task = db.query(models.Task).filter(models.Task.id == id).first()
task = db.query(models.Task).get(id)
return task

def getAll(db: Session):
    tasks = db.query(models.Task).all()
    return tasks

def create(task: Task, db: Session):
    taskdb = models.Task(name=task.name, description= task.description, status=task.status)
    db.add(taskdb)
    db.commit()
    db.refresh(taskdb)
    return taskdb

def update(id: int, task: Task, db: Session):

    taskdb = getById(id, db)

    taskdb.name = task.name
    taskdb.description = task.description
    taskdb.status = task.status

    db.add(taskdb)
    db.commit()
    db.refresh(taskdb)
    return taskdb

def delete(task: Task,db: Session):
    db.delete(task)
    db.commit()

```

Summary of previous operations

- With **db.query(task)** we indicate which model we want to connect to; It has been the equivalent of SELECT.
- With **filter()** we indicate the filters to search for matches by ID, NAME or similar, it has been the equivalent of WHERE.
 - You can also use: **db.query(task).get(id)**.
- With the **all()** function, a set of records is fetched, therefore, a list of objects is obtained.
- With **first()** we obtain the first record, which is the equivalent of FIRST.
- With the **add()** function, it is possible to create a record (if the supplied object is not passed the PK -it does not have the id set-) or update (if the object has the id set).
- The **delete()** function allows you to delete a record.
- With the **commit()** function, apply the changes to the database.
- Using the **refresh()** function allows you to update the instance with the changes.

Here we can see the advantages of using an ORM with SQLAlchemy since it allows you to use a set of functions to create the connection to the database instead of using SQL.

As an important point, the 'task' argument corresponds to the instance of the Pydantic class, and the one used internally in the functions (**create()**, **update()**, **delete()** among others) corresponds to an instance of the class SQLAlchemy model.

You can try the above methods from the API methods, at the moment, they are not being linked to the task received through the request; this adaptation will be made later in the chapter; so, to test them, you can supply fixed values like the following:

```
crud.getById(db=db,id=1).name
crud.getAll(db=db)[1].name

crud.create(Task(name='Test',description='Descr', status= StatusType.DONE),db=db)

crud.update(index,Task(name='New Task',description='Descr', status= StatusType.DONE),db=db)
```

Pagination

In order to use pagination in FastAPI we do not have functions that do it automatically, as it happens with other frameworks like Flask or Django, therefore, we can do it manually, or use an existing solution like the following:

<https://github.com/jayhawk24/pagination-fastapi/blob/main/pagination.py>

What we will copy in:

database/pagination.py

Staying like:

database/pagination.py

```
from typing import Generic, List, TypeVar

from pydantic import BaseModel, conint
# from pydantic.generics import GenericModel

class PageParams(BaseModel):
    page: conint(ge=1) = 1
    size: conint(ge=1, le=100) = 10

T = TypeVar("T")

class PagedResponseSchema(BaseModel, Generic[T]):
    """Response schema for any paged API."""

    total: int
```

```

page: int
size: int
results: List[T]

def paginate(page_params: PageParams, query, ResponseSchema: BaseModel) ->
PagedResponseSchema[T]:
    """Paginate the query."""

    paginated_query = query.offset((page_params.page - 1) *
page_params.size).limit(page_params.size).all()

    return PagedResponseSchema(
        total=query.count(),
        page=page_params.page,
        size=page_params.size,
        results=[ResponseSchema.from_orm(item) for item in paginated_query],
    )

```

From this file, we will use a function called **paginate()** that takes the following arguments:

- Parameters to indicate the number of pages and the number of records per page; for this, the **PageParams** class is used, which is already implemented in the previous solution and has a couple of arguments to specify the page and page size.
- The query.
- The Pydantic class.

We create the CRUD function that uses the previous pagination function:

database/task/crud.py

```

from database.pagination import PageParams, paginate
***
def pagination(page: int, size:int, db: Session):
    pageParams = PageParams()
    pageParams.page = page
    pageParams.size = size
    return paginate(pageParams, db.query(models.Task).filter(models.Task.id > 2), Task)

```

You can customize the **query** to your liking to indicate a filter, for example:

```
db.query(Task).filter(Task.id > 1)
```

To test the above function, you can do something like the following:

```
pagination(db,1,2)
```

Relations

As we mentioned at the beginning, with SQLAlchemy we can interact with relational databases, that is, we can define the relationships in the database using classes that are later translated into tables as we did before. Let's look at the most common types of relationships and how to apply them in SQLAlchemy.

One to Many relationship

How we initially configured, the tasks are established to a category and a user; to define these relationships from the **Task** model for the database we have:

database/models.py

```
from sqlalchemy.schema import Column, ForeignKey
from sqlalchemy.types import String, Integer, Text, Enum
from sqlalchemy.orm import relationship
***
class Task(Base):
    ***
    category_id = Column(Integer, ForeignKey('categories.id'),
        nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'),
        nullable=False)

class Category(Base):
    __tablename__ = 'categories'
    id = Column(Integer, primary_key=True)
    name = Column(String(20))

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(20))
    surname = Column(String(20))
    email = Column(String(50))
    website = Column(String(50))
```

As you can see, we created two extra model classes, one for the user and another for the category and its foreign relationship with the tasks through the **ForeignKey** class that receives as a parameter, the column to relate to, which in this example would be the id column of the categories and users tables.

Remember that once you define the previous models, delete all the tables from your project database, take the server down and bring it up again so that the previous tables are created.

Changes in the CRUD of tasks (database)

Now, we have new columns for the task model and associated table, specifically, the category and users columns, first of all, create an example category and user directly in the database, i.e. in their respective tables:

```
INSERT INTO `categories` (`name`) VALUES ('Cate 1');
```

```
INSERT INTO `users` (`name`, `surname`, `email`, `website`) VALUES ('andres', 'cruz',  
'admin@admin.com', 'http://desarrollolibre.net');
```

Once this is done, we can include these changes at the level of the CRUD operations:

database/task/crud.py

```
def create(task: Task, db: Session):  
    taskdb = models.Task(name=task.name, description=task.description, status=task.status,  
category_id = task.category_id, user_id = task.user_id)  
    db.add(taskdb)  
    db.commit()  
    db.refresh(taskdb)  
    return taskdb  
  
def update(id: int, task: Task, db: Session):  
  
    taskdb = getById(id, db)  
  
    taskdb.name = task.name  
    taskdb.description = task.description  
    taskdb.status = task.status  
    taskdb.category_id = task.category_id  
  
    db.add(taskdb)  
    db.commit()  
    db.refresh(taskdb)  
    return taskdb
```

As you can see, the changes made consist of defining the user identifier and category fields when creating and editing a task; to use the above changes, we do:

```
crud.create(Task(name='Test',description='Descr', status= StatusType.DONE, category_id=1,  
user_id=1),db=db)  
crud.update(1,Task(name='HOLA MUndo 2',description='Descr', status= StatusType.DONE,  
category_id=2, user_id=1),db=db)
```

Get the relationship from the main entity

If we consult the detail of a task by any of the new fields:

```
db.query(Task).filter(Task.id == task_id).first().category_id  
>> 1
```

We will see that we only have the identifier, that is, the PK of the relationships, whether it is the user or the category, but we do not have the entity; that is to say, if given the detail of a task, we want to obtain the name of

the associated category, currently we cannot do it, we would have to make another query to the database, looking for the detail of the category given the PK:

```
db.query(Category).filter(Category.id == category_id).first().name
>> Cate 1
```

This process is optimal since in many circumstances we will want to access the foreign relationship (category/user) from the main relationship (tasks).

To do this, we can create a property in the special class that allows us to obtain the foreign relation:

```
from sqlalchemy.orm import relationship

class Task(Base):
    """
    category = relationship('Category', lazy="joined")
```

With this, it is now possible to fully access the foreign relation from the main relation; for example:

```
db.query(Task).filter(Task.id == task_id).first().category
>> <database.models.Category object at 0x107b05711>
```

Resulting in the related entity and we can access any property of it, such as the id, name, etc; for example:

```
db.query(Task).filter(Task.id == task_id).first().category.name
```

With the lazy option, the type of load is indicated when get the relation:

```
relationship(<MODEL>, lazy="joined")
```

You can indicate several values, but among the main ones we have:

lazy loading or deferred loading, this option is available through **lazy='select'**, in this way of loading a SELECT statement is issued at the time of access to the attribute to lazy load a related reference in a single object to the time; in our example, this means that when obtaining the category from the detail of a task:

```
db.query(Task).filter(Task.id == task_id).first().category
```

An additional query is made to the database to load the category using a SELECT.

joined loading: This option is available by **lazy='joined'**, this form of loading applies a JOIN to the given SELECT statement so that related rows are loaded in the same result set, only one connection is made to the database and not two as in the previous case; following our example, when obtaining a task:

```
db.query(Task).filter(Task.id == task_id).first()
```

The relationships are also loaded, how would the categories and users in a JOIN.

Clarified the operation, we apply the same logic that applies to users:

database/models.py

```
class Task(Base):
    """
    user = relationship('User', lazy="joined")
```

Inverted relations

Also, it is possible to get the relationship reversed in a one-to-many relationship; for the moment, with the relationship created in the task model, we have the direct relationship, where a task has a single category established, but it is also possible to obtain the inverted relationship where, given a category, we can obtain all the tasks to which it is assigned; to do this, we must make the following changes:

database/models.py

```
class Task(Base):
    """
    category = relationship('Category', lazy="joined", back_populates='tasks')

class Category(Base):
    """
    tasks = relationship('Task', back_populates='category', lazy="joined")
```

Being a one-to-many relationship, we create a new tasks relationship for the categories, but since the relationship between the tasks and categories is recorded in the tasks entity, it would not be possible to create a relationship for the relationship inverted, that is, given the category, get the tasks; however, in SQLAlchemy we can use the **back_populates** argument which allows the automatic generation of a new relation that will be automatically added to the ORM mapping. Its use is very simple, you must place in the current relation (the one of tasks) the name of the relation that you are going to place in the inverse relation (the one of categories) and vice versa; in the end, both relations are referring to each other.

With this, we can obtain the direct relationship:

```
db.query(Task).filter(Task.id == task_id).first().category
>> <database.models.Category object at 0x107b06710>
```

And the hint (which will return a list of tasks, the tasks assigned to a category):

```
db.query(Category).filter(Category.id == 1).first().tasks
>> [<database.models.Task object at 0x107b05710>]
```

The same logic applies for users:

database/models.py

```
class Task(Base):
    """
    user = relationship('User', lazy='joined')
```

```
class User(Base):
    """
    tasks = relationship('Task', back_populates='user', lazy='joined')
```

Many to Many relationship

To handle many to many relationships, a pivot table must be used to manage the relationships; for this example, we'll create a tag model:

database\models.py

```
class Tag(Base):
    __tablename__ = 'tags'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

In which a tag can be assigned to many tasks and a task can have many tags assigned to it; from the task relationship, we create the tag relationship:

database\models.py

```
class Task(Base):
    """
    tags = relationship('Tag', secondary=task_tag, back_populates='tasks')
```

On this occasion, the **secondary** option is defined to indicate in the name of the pivot table that it is the one that will save the PKs of each one of these entities, that is, it will save the PK of the tasks and labels:

database\models.py

```
from sqlalchemy import Table
"""
task_tag = Table('task_tag',
                 Base.metadata,
                 Column('task_id', Integer, ForeignKey('tasks.id'), primary_key=True),
                 Column('tag_id', Integer, ForeignKey('tags.id'), primary_key=True))
```

Since we're not interested in instantiating **task_tag** directly, there's no need to have a model for this pivot table; for this we use the **Table** class that allows us to create an association table between two classes (the **Task** and **Tag** model); in addition to supplying the foreign keys of both tables to relate the **tasks** and **tags** and establishing them as primary keys to avoid repeated records, the **Table** class receives the metadata (**Base.metadata**) as a parameter so that the foreign keys can be located with the tables to link.

From the tags table optionally, we can also define the relationship with the tasks:

```
class Tag(Base):
    """
    tasks = relationship('Task', secondary=task_tag)
```


The problem with the above code is that SQLAlchemy will give a warning in the terminal like the following:

```
SAWarning: relationship 'Tag.tasks' will copy column tags.id to column task_tag.tag_id, which conflicts with relationship(s): 'Task.tags' (copies tags.id to task_tag.tag_id). If this is not the intention, consider if these relationships should be linked with back_populates, or if viewonly=True should be applied to one or more if they are read-only. For the less common case that foreign key constraints are partially overlapping, the orm.foreign() annotation can be used to isolate the columns that should be written towards. To silence this warning, add the parameter 'overlaps="tags"' to the 'Tag.tasks' relationship. (Background on this warning at: https://sqlalche.me/e/20/qzyx) (This warning originated from the `configure_mappers()` process, which was invoked automatically in response to a user-initiated operation.)
```

In the previous error it is indicated that there should be a main relationship; so, we can leave the relations as follows, indicating the option of **back_populates** from the main relation (the one of tasks):

```
class Task(Base):
    """
    tags = relationship('Tag', secondary=task_tag, back_populates='tasks')

class Tag(Base):
    """
    tasks = relationship('Task', secondary=task_tag)
```

Another way to solve the previous problem would be to use the legacy option of **backref**; with the **backref** option, the task relation for the labels is automatically created; works in a similar way to **back_populates** although the latter does not create the relationship automatically; It is important to mention that currently the **backref** option is going to be replaced by the **back_populates** option.

```
class Tag(Base):
    """
    #tasks = relationship('Task', secondary=task_tag)

class Task(Base):
    """
    tags = relationship('Tag', secondary=task_tag, backref='tasks')
```

Operations in a many-to-many relationship

In this section, we will see how to perform the operations of, obtain the detail of the tasks/tags and how to insert and delete records in the many-to-many relationship.

To add a tag from a task:

```
task = db.query(models.Task).get(2)
tag1 = db.query(models.Tag).get(1)
task.tags.append(tag1)
```

```
db.add(task)
db.commit()
```

To remove a tag from a task:

```
task = db.query(models.Task).get(2)
tag1 = db.query(models.Tag).get(1)
task.tags.remove(tag1)

db.add(task)
db.commit()
```

To add a task from a tag:

```
task = db.query(models.Task).get(2)
tag1 = db.query(models.Tag).get(1)
tag1.tasks.append(task)

db.add(tag1)
db.commit()
```

To remove a task from a tag:

```
task = db.query(models.Task).get(2)
tag1 = db.query(models.Tag).get(1)
tag1.tasks.remove(task)

db.add(tag1)
db.commit()
```

Finally, to get the list of tags given the tasks:

```
db.query(models.Task).get(2).tags
```

Or to get the list of tasks given the tag:

```
db.query(models.Tag).get(1).tasks
```

You can get more information about relationships at:

https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html

Task CRUD changes (API methods)

In this section, we will make several changes and tests to the API methods so that it can be successfully integrated into the database using the CRUD created above.

Methods to add/remove tags to a task

Once the new relationship has been created, we will create in the CRUD the operations for assigning and removing the tags of a task, together with a helper method to obtain the detail of a tag:

database/task/crud.py

```
***
#*****TAGS

def getTagById(id:int, db: Session):
    tag = db.query(models.Tag).get(id)

    if tag is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)
    return tag

def tagAdd(id:int, idTag:int, db: Session):
    task = getById(id,db)
    tag = getTagById(idTag,db)

    tag.tasks.append(task)
    db.add(tag)
    db.commit()

    return task

def tagRemove(id:int, idTag:int, db: Session):
    task = getById(id,db)
    tag = getTagById(idTag,db)

    task.tags.remove(tag)
    db.add(task)
    db.commit()

    return task
```

From the API methods, we create the methods for the management:

task.py

```
***
#***** tag
@task_router.put("/tag/add/{id}", status_code=status.HTTP_200_OK)
def tagAdd(id: int = Path(ge=1), idTag:int = Body(ge=1), db: Session =
Depends(get_database_session)):
    return crud.tagAdd(id,idTag,db)

@task_router.delete("/tag/remove/{id}", status_code=status.HTTP_200_OK)
```

```
def tagRemove(id: int = Path(ge=1), idTag:int = Body(ge=1), db: Session = Depends(get_database_session)):
    return crud.tagRemove(id, idTag, db)
```

Move sample data to separate files

One of the elements that we must update is about the sample data used to create the tasks; let's start by moving the data, the dictionary for selecting various examples from the signature of the **create()** function, to a separate file:

dataexample.py

```
from schemes import StatusType

taskWithOutORM = {
    "normal1": {
        "summary": "A normal example 1",
        "description": "A normal example",
        "value": {
            "id" : 123,
            "name": "Salvar al mundo",
            "description": "Hola Mundo Desc",
            "status": StatusType.PENDING,
            "tag":["tag 1", "tag 2"],
            "category": {
                "id":1234,
                "name":"Cate 1"
            },
            "user": {
                "id":12,
                "name":"Andres",
                "email":"admin@admin.com",
                "surname":"Cruz",
                "website":"http://desarrollolibre.net",
            }
        }
    },
    "normal2":{
        "summary": "A normal example 2",
        "description": "A normal example",
        "value": {
            "id" : 12,
            "name": "Sacar la basura",
            "description": "Hola Mundo Desc",
            "status": StatusType.PENDING,
            "tag":["tag 1"],
            "category_id": {
                "id":1,
                "name":"Cate 1"
            }
        }
    }
}
```



```

        "id" : 12,
        "name": "Sacar la basura",
        "description": "Hola Mundo Desc",
        "status": StatusType.PENDING,
        "tag":["tag 1"],
        "category":2,
        "user_id": 1
    }
},
"invalid":{
    "summary":"A invalid example 1",
    "description":"A invalid example",
    "value":{
        "id" : 12,
        "name": "Sacar la basura",
        "description": "Hola Mundo Desc",
        "status": StatusType.PENDING,
        "tag":["tag 1"],
        "user_id": {
            "id":12,
            "name":"Andres",
            "email":"admin@admin.com",
            "surname":"Cruz",
            "website":"http://desarrollolibre.net",
        }
    }
}
}
}
}

```

With this, we gain readability and a more modular scheme for the application; this way we can easily extend and change the examples and use them from the API:

task.py

```

from dataexample import taskWithoutORM
***
@task_router.post("/", status_code=status.HTTP_201_CREATED)
def add(task: Task = Body(examples=taskWithoutORM), db: Session =
Depends(get_database_session)):
    ***

@task_router.put("/", status_code=status.HTTP_200_OK)
def update(id: int, task: Task = Body(examples=taskWithoutORM), db: Session =
Depends(get_database_session)):
    ***

```

In the file created above, we have two variables, one that holds the sample data for Pydantic:

```
taskWithoutORM
```

And another used for SQLAlchemy:

```
taskWithORM
```

From now on, we'll leave the sample data set for the Rest API:

task.py

```
from dataexample import taskWithORM
***
@task_router.post("/", status_code=status.HTTP_201_CREATED)
def add(task: Task = Body(examples=taskWithORM), db: Session =
Depends(get_database_session)):
    ***

@task_router.put("/", status_code=status.HTTP_200_OK)
def update(id: int, task: Task = Body(examples=taskWithORM), db: Session =
Depends(get_database_session)):
    ***
```

API methods

The next change to make is to update each of the API methods; for the listing one, we will now use the **getAll()** method to get all the tasks:

```
@task_router.get("/", status_code=status.HTTP_200_OK)
def get(db: Session = Depends(get_database_session)):
    return { "tasks": crud.getAll(db) }
```

By now using the SQLAlchemy classes instead of the Pydantic ones, we will see that we now have access to the entire entity, including its relationships, therefore we have output like the following:

```
{
  "tasks": [
    {
      "status": "done",
      "category_id": 2,
      "id": 1,
      "name": "Test Name",
      "description": "Data Description",
      "user_id": 1,
      "user": {
        "surname": "cruz",
        "name": "andres",
        "website": "http://desarrollolibre.net",
        "id": 1,
        "email": "admin@admin.com"
      }
    }
  ]
}
```

```

    },
    "category": {
        "name": "Cate 2",
        "id": 2
    }
},
***
]
}

```

For the create method, the task is supplied at creation time and the created task is returned:

task.py

```

@task_router.post("/", status_code=status.HTTP_201_CREATED)
def add(task: Task = Body(
    examples=taskWithORM
), db: Session = Depends(get_database_session)):

    return { "tasks": crud.create(task,db=db) }

```

The same as in update; apart from sending the task, the id of the task is supplied through the URL and the updated task is returned:

task.py

```

@task_router.put("/{id}", status_code=status.HTTP_200_OK)
def update(id: int = Path(ge=1), task: Task = Body(
    examples=taskWithORM), db: Session = Depends(get_database_session)):

    return { "task": crud.update(id, task, db) }

```

To delete, using the ID supplied by the URL, we delete the task:

task.py

```

@task_router.delete("/{id}", status_code=status.HTTP_200_OK)
def delete(id: int = Path(ge=1), db: Session = Depends(get_database_session)):
    crud.delete(id,db)
    return { "tasks": crud.getAll(db) }

```

We implement a task detail method for the API:

task.py

```

@task_router.get("/{id}", status_code=status.HTTP_200_OK)
def get(id: int = Path(ge=1), db: Session = Depends(get_database_session)):
    return crud.getById(id, db)

```


Check if id exists in getById()

In the detail method of the task for the crud of the database, we check if the supplied id exists, to do this, if the id of a task that does not exist, the task is null; in that case, an exception is returned:

crud.py

```
from fastapi import HTTPException, status
***
def getById(id: int, db: Session):
    # task = db.query(models.Task).filter(models.Task.id == id).first()
    task = db.query(models.Task).get(id)

    if task is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)
    return task
```

The 404 exception implemented above will occur in any method that uses the **getById()** function, such as update, detail, and delete.

Integration of the SQLAlchemy and Pydantic classes in the response of the API methods

Currently, we are using SQLAlchemy's **Task** class as input to generate the responses and we can clearly see this in the methods to get the detail and/or list for the tasks:

```
from fastapi import HTTPException, status
***
def getById(id: int, db: Session):
    task = db.query(models.Task).get(id)
    ***
    return task
```

Depending on the application you want to build, you may want to customize this output, for example, use Pydantic's classes to generate these responses instead of SQLAlchemy's; for this, we have access to certain methods to convert from one class to another; that is, from the SQLAlchemy class (which is the one used by default when generating the responses by the API provided by the different methods implemented to communicate with the database) to a Pydantic class.

Recall that at the beginning of this chapter, we specified the ORM option on Pydantic models:

```
class Config:
    from_attributes = True
```

With it, we have access to methods like the following:

```
<ClassPydantic>.from_orm
```

With which we can automatically convert an instance of SQLAlchemy to Pydantic; for example, in the listing, if we wanted to convert to a Pydantic class, we have:

```
@task_router.get("/", status_code=status.HTTP_200_OK)
def get(db: Session = Depends(get_database_session)):
    # return { "tasks": crud.getAll(db) }
    return { "tasks": [Task.from_orm(task) for task in crud.getAll(db) ]}
```

Or in detail:

```
@task_router.get("/{id}", status_code=status.HTTP_200_OK)
def get(id: int = Path(gt=1), db: Session = Depends(get_database_session)):
    # return crud.getById(id, db)
    return Task.from_orm(crud.getById(id, db))
```

The advantage of this scheme is that it is possible to customize the output by changing the Pydantic model, which, as we will see in the next section, it is very easy to customize the Pydantic model for each API method.

It is important to mention that this conversion consumes additional resources (to do the conversion) and you should do it when necessary, as a recommendation, use the class provided by the database to generate the response whenever possible.

Reading and writing Pydantic models

When working with Pydantic models you may want to vary the structure when we are going to create or update; in the application that we are building, it would not make sense to pass the user when the task is being updated, since the user corresponds to the owner of the task and this owner is the creator of the task, maybe also when you want to obtain the detail or list of tasks, also customize the output; for example, not indicating the relationship of the user.

To handle these situations, we can have a model for each situation, that is, one of the read and another to create and/or update:

schemes.py

```
class TaskBase(BaseModel):
    name: str
    description: Optional[str] = Field("No description",min_length=5)
    status: StatusType

    category_id: int = Field(gt=0)
    class Config:
        from_attributes = True

class TaskRead(TaskBase):
    id:int

class TaskWrite(TaskBase):
    id: Optional[int] = Field(default=None)
    user_id: Optional[int] = Field()
```

In the code presented above, we can see that we have a base model for the tasks and from this, new classes are generated by adding or customizing their properties; in the case of reading (detail and get all the tasks), it would not be necessary to supply the user since the user would be the same, the authenticated user, therefore, it would be an additional load that consumes unnecessary resources, but, to create a task, if it would be necessary to send the user, something similar happens when it is going to update, it would not be necessary to send the user; therefore the user is defined as optional for the write model. Regarding the ID, it is usually required to show the detail/list of the task but to create a task it is not necessary (it is generated when inserting into the database), when updating it is necessary to indicate the ID, although, from the API is supplied by en path of the route.

Defining the ID at the class level, for the management that we currently have, would not be necessary, since, in the editing method, it is received by the URL path; as we discussed earlier, this implementation is for demo only.

As a demonstration, from the API, we can define it as follows:

task.py

```
***
from schemes import Task, TaskRead, TaskWrite

***

@task_router.get("/", status_code=status.HTTP_200_OK)
def get(db: Session = Depends(get_database_session)):

    # return { "tasks": crud.getAll(db) }
    return { "tasks": [ TaskRead.from_orm(task) for task in crud.getAll(db) ] }

@task_router.get("/{id}", status_code=status.HTTP_200_OK)
def get(id: int = Path(ge=1), db: Session = Depends(get_database_session)):
    # return crud.getById(id, db)
    return Task.from_orm(crud.getById(id, db))

@task_router.post("/", status_code=status.HTTP_201_CREATED)
def add(task: TaskWrite = Body(
    examples=taskWithORM
), db: Session = Depends(get_database_session)):

    # return { "tasks": TaskWrite.from_orm(crud.create(task,db=db)) }
    return { "tasks": crud.create(task,db=db) }

@task_router.put("/{id}", status_code=status.HTTP_200_OK)
def update(id: int = Path(ge=1), task: TaskWrite = Body(
    examples=taskWithORM), db: Session = Depends(get_database_session)):

    return { "task": crud.update(id, task, db) }

***
```

It is important to mention that the code presented in this section is demo and you can customize it according to the needs of the application you are developing, but, with the scheme presented above, you will see that in the reading methods, the ID is always present in the output, for the writing methods in the input, the user must not be supplied in a mandatory way.

Remove columns from queries in SQLAlchemy

From SQLAlchemy, it is also possible to remove columns from the query by using the `load_only()` function together with the `options()` function and specifying the columns; for example:

```
from sqlalchemy.orm import load_only
***
def getById(id: int, db: Session):
    task = db.query(models.Task).options(load_only(models.Task.name,
models.Task.status)).get(id)
    ***
```

And you get as output:

```
{
  "id": 2,
  "name": "Salvar al mundo",
  "status": "pending",
  "user": {
    "email": "admin@gmail.com",
    "id": 1,
    "name": "andres",
    "surname": "cruz",
    "website": "http://desarrollolibre.net/"
  },
  "category": {
    "id": 1,
    "name": "Cate 1"
  }
}
```

Dates

It is also possible to define dates to the entities, for example, for when it is created or updated, define the date of the operation; for it:

```
from sqlalchemy.types import *** DateTime
from sqlalchemy.sql import func

class Task(Base):
    ***
    created = Column(DateTime(timezone=True), server_default=func.now())
```

```
updated = Column(DateTime(timezone=True), onupdate=func.now())
```

Now when a task is created (by the database or API), the current date will be recorded in the **created** column; when the task is updated (using the API), the update date will be recorded in the **updated** column.

Key points

- Using the **server_default** option we can set the date at the time of creation, the current date is used **func.now()**, this will activate the **CURRENT_TIMESTAMP** option for the column in the database.
- Using the **onupdate** option, a callback is executed in SQLAlchemy so that when the record is updated (**onupdate**) the current date is established **func.now()**.

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.5>

Chapter 8: Template in FastAPI

A template engine is used to separate presentation logic from data. Through impressions, filters, conditionals, bucles or imports, it allows you to generate the HTML code necessary to represent a web page. The template engine are widely used in server-side programming, since they allow the presentation of data without having to modify the source code.

In short, with a templating engine it is possible to present the data handled by the server, specifically FastApi, and generate the HTML in the process; for example, if we have a list of users, we can print it in a table or similar in HTML using certain functionalities (directives).

About Jinja

Jinja is a template engine written in Python designed to be simple and expressive; Jinja is the templating engine used in other Python web frameworks, such as Flask, it is also very similar to the one used by Django.

With Jinja as a template engine we can easily render API responses.

Jinja's templating engine uses square brackets { } to distinguish its expressions and syntax from HTML:

1. The {{ }} syntax is called a variable block and we can use it to print in the template.
2. The {% %} syntax is used to control structures such as if/else, loops, and macros.
3. The {# #} syntax is for comments.

First steps with Jinja

In this section we are going to learn about the main features of Jinja, which range from the use of control blocks to the use of filters and the template in general.

Control blocks

One of the crucial elements in template engines are control blocks; we can perform from conditionals to for loops that work in the same way as in Python; with this it is possible to create conditional logic to render HTML blocks if a condition is met, as well as iterate complete data collections, for example a list of tasks in a table or the like.

Conditionals

We can make use of conditionals to evaluate true and false conditions:

```
<div>
  {% if True %}
    Is TRUE
  {% endif %}
</div>
```

Conditionals in Jinja are exactly the same as in Python, we can use else, or, and, etc:

```
<div>
  {% if True and False %}
    Is True
  {% else %}
    Not is True
  {% endif %}
</div>
```

Bucle for

Just like the for loop in Python, we can use it in Jinja:

```
<ul>
  {% for item in items %}
    <li>{{ item }}</li>
```

```
{% endfor %}  
</ul>
```

In a for loop block, you can access some special variables, the main ones being:

- `loop.index`: Gets the current index of the iteration starting from one.
- `loop.index0`: Gets the current index of the iteration starting from zero.
- `loop.first`: True if this is the first iteration.
- `loop.last`: True if it is the last iteration.
- `loop.length`: Returns the length of the list.

For example, to use the variable **last**:

```
<ul>  
  {% for e in [1,2,3,4,5,6,10] %}  
  <li>{{ e }} {{ loop.last }} </li>  
  {% endfor %}  
</ul>
```

All variables must be used in the for loop.

Filters

Filters in Jinja are one of the most important and versatile features that exist in this template engine; allow you to manipulate the content displayed in a template. There are several predefined filters in Jinja although we can extend the existing ones with custom filters; the filters are applied to the variables that are used, for example, to count the length of a text string, join texts into one, separate by a separator, format dates, numbers and a long etc; in short, they allow for custom manipulations of the content.

To use a filter, use the pipe operator (`|`) after specifying the argument or arguments, and after the pipe operator, the filter you want to use:

```
<DATA> | <FILTER>
```

Let's look at some filters in Jinja.

Filter default

The default filter is used to replace the output if the variable has a null value (not defined):

```
{{ task | default('This is a default Task') }}
```

For example, if `task` is null, we will have an output like:

```
This is a default Task
```

If `task` were defined or we evaluated, for example, a number.

```
{{ 5 | default('This is a default Task') }}
```

We would see the number on the screen:

```
| 5
```

Filter escape

Replace the &, <, >, ', and " characters in the supplied string with safe characters to display text that contains the HTML characters:

```
| {{ "<title>Todo Application</title>" | escape }}
```

We will have as output:

```
| &lt;title&gt;Todo Application&lt;/title&gt;
```

Filter conversion

These filters include int and float filters used to convert from one data type to another:

```
| {{ 3.142 | int }}
```

We will have as output:

```
| 3
```

Or for example:

```
| {{ 31 | float }}
```

We will have as output:

```
| 31.0
```

Filter max

Returns the largest element in the list:

```
| {{ [1, 2, 3, 4, 5] | max }}
```

And we will have:

```
| 5
```

Filter min

Returns the smallest element in the list:

```
| {{ [1, 2, 3, 4, 5] | min }}
```

And we will have:


```
1
```

Filter round

Allows you to round a floating number:

```
{{ 3.141|round }}  
{{ 3.5|round }}
```

And we will have:

```
3.0  
4.0
```

We can also indicate the precision:

```
{{ 42.55|round(1) }}
```

And we will have:

```
42.5
```

Or

```
{{ 3.595|round(2) }}  
{{ 3.595|round(2, 'floor') }}
```

And we will have:

```
3.59  
3.6
```

Filter replace

This filter allows you to replace one piece of text with another and receives two arguments:

1. The first argument is the text string to be replaced.
2. The second is the replacement text string.

```
{{ "Hello FastAPI"|replace("Hello", "Hi") }}
```

And we will have:

```
Hi FastAPI
```

Filter join

This filter is used to combine text strings into one:

```
{{ ['Hello', 'world', 'in', 'FastAPI!'] | join(' ') }}
```

And we will have:

```
Hello world in FastAPI!
```

Filter lower

Convert a text to lowercase:

```
{{ "Hello FastAPI"|lower }}
```

And we will have:

```
hello fastapi
```

You can also implement the filters as follows:

```
{% filter lower %}
    Hello FastAPI
{% endfilter %}
```

Filter upper

Convert a text to uppercase:

```
{{ "Hello FastAPI"|upper }}
```

Or

```
{% filter upper %}
    Hello FastAPI
{% endfilter %}
```

And we will have:

```
HELLO FASTAPI
```

Filtro reverse

Inverts a text or list:

```
{{ "Hello FastAPI"|reverse }}
```

And we will have:

```
IPAtsaF olleH
```

For a list:

```
{% for e in [1,2,3]|reverse %}
    {{ e }}
```

```
{% endfor %}
```

Filter length

This filter is used to get the length of the supplied object or array, it works the same way as the `len()` function in Python:

```
Task count: {{ tasks | length }}
```

And we will have:

```
Task count: 5
```

Or a list:

```
{{ [1,2,3,4,5]|length }}
```

And we will have:

```
5
```

Filter sort

Allows you to sort a list:

```
{% for n in [10,2,43,-5,0]|sort %}  
  {{ n }}  
{% endfor %}
```

And we will have:

```
-5 0 2 10 43
```

It is also possible to indicate some parameters:

- **reverse**: Sort descending instead of ascending.
- **case_sensitive**: When sorting strings, sort case separately.
- **attribute**: When sorting objects or dictionaries, you specify an attribute or key to sort on. You can use dot notation like "address.city" or a list of attributes like "age, name".

For example, to sort backwards:

```
{% for n in [10,2,43,-5,0]|sort(reverse=true) %}  
  {{ n }}  
{% endfor %}
```

And we will have:

```
43 10 2 0 -5
```

Filter slice

Cut a list into chunks and return a list of list:

```
{% for n in [10,2,43,-5,0]|slice(3) %}  
  {{ n }}  
{% endfor %}
```

And we will have three lists contained in one list:

```
[10, 2] [43, -5] [0]
```

We will have as output:

```
<title>FastAPI</title>  
var1
```

Set variables

If we wanted to define variables in a template:

```
{% set var1 = 'FastAPI' %}  
{% set var2 = var1 %}  
  
{{ var1 }}
```

Blocks

Many of the features that we can use in Jinja reside in blocks of code. These look like:

```
{% <BLOCKTYPE> %}  
<BODY>  
{% <ENDBLOCKTYPE> %}
```

Code blocks allow both a better ordering for the code and to divide the block into sections using different types of blocks that allow a certain function to be performed.

We have already seen code blocks, such as the if and for control blocks, but there are others as we will see below.

Block raw

With the **raw** block you can escape variables, prints or blocks; this way the content enclosed by the **raw** block would not be processed by Jinja:

```
{% set var1 = '<title>FastAPI</title>' %}  
{% set var2 = var1 %}  
  
{{ var1 }}  
{% raw %}
```

```
var1
{{ var1 }}
<title>FastAPI</title>
{% endraw %}
```

We will have as output:

```
var1
{{ var1 }}
<title>FastAPI</title>
```

Block macro

Macros are the Jinja equivalent of functions; that is, it is a callable block to which arguments can be passed.

```
{% macro suma (v1, v2) %}
    <p>{{ valor1 }} + {{ valor2 }} = <strong> {{ valor1 + valor2 }} </strong>
{% endmacro %}
```

We call the previous macro with:

```
{{ suma(5, 4) }}
```

We will have as output:

```
5+4=9
```

Template inheritance

In this section, we will see how to use templates in Jinja; specifically how to use master templates to be able to reuse a base skeleton in multiple views and how to include view fragments (other templates) in templates.

In this section we will use the terms view, template and template in an equivalent way.

Master template

Many times it is necessary to have a master view where certain key content is updated; for example, if we have a management module, the common elements between the screen could be a header to define the links, as well as a logo, the footer and a container, the dynamic elements were presented in the middle of the screen.

In Jinja we can use master templates to implement this type of logic; in the end, a master view is nothing more than a template, where we have blocks or special sections (**block**) to indicate where the content that is going to vary goes and this master template is included in the final templates and through the specified blocks in the master template, they are referenced in the final templates with the final content; this Jinja functionality is known as template inheritance.

Let's create a base template or skeleton that is going to be used to be inherited from other templates:

master.html

```

<!DOCTYPE html>
<html lang="es">
<head>
  {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - DesarrolloLibre</title>
  {% endblock %}
</head>
<body>
  <div id="content">{% block content %}{% endblock %}</div>
  <div id="footer">
    {% block footer %}
      DL by <a href="#">Contact</a>.
    {% endblock %}
  </div>
</body>
</html>

```

In the above code, 3 blocks are defined:

1. One for the title in the header.
2. One for the title in the content.
3. Another for the content, which would be the place where we place the list, form among others.

These blocks or sections can also have default content; for example:

```

<head>
  {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
  {% endblock %}
</head>

```

Notice that in the case of the block named **head**, it contains a **link** and title tag.

You can have as many blocks as you need in any layout of the HTML page; you can also create as many master templates as you need and reuse in the app; these templates are really useful when we want to import common CSS, JavaScript or HTML for example and can be easily reusable, therefore, when changing the content of the template, it is automatically updated in the templates that inherit from this master template, the calls child templates.

Another important point is the names of the blocks:

```

<div id="content">{% block content %}{% endblock %}</div>

```

In the previous case, the name would be content, since, with these blocks, they are the ones that are used from the child views (those that inherit the HTML, CSS and JavaScript defined in the master template) and it is specified where it is going to place the content and in which block.

At the moment, there is a template with some blocks, this template is nothing more than an HTML page with some markup; the real utility of this we can see when we use the template in other HTML pages; for example:

index.html

```
{% extends "master.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome to my awesome homepage.
    </p>
{% endblock %}
```

As you can see, this child page specifies the master template to use:

```
{% extends "master.html" %}
```

And then each of the master template blocks are implemented:

```
{% block title %}Index{% endblock %}
{% block  %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome to my awesome homepage.
    </p>
{% endblock %}
```

It is important that you use the same names of the blocks so that you place the content where it really should be.

The **super()** function is used to override the code set in the master template.

Include View Fragments

Another feature that we have in template engines is being able to include view fragments within other views; for this, the include directive is used:

```
{% include 'header.html' %}
```

In the example above, we have a view at the root called **header.html** that we can include in other views.

You can get the full documentation on this template engine at:

<https://jinja.palletsprojects.com/>

Using Jinja in FastAPI

In this section, we will see how to install Jinja 2 in a FastAPI project and different examples of how we can use both technologies together to create a web application that will not only process requests on the server and return a response in JSON format, but also also process requests to return HTML pages as responses with which we can manage data.

Install Jinja

To install the template engine known as Jinja, we execute in the project:

```
$ pip install Jinja2
```

To use Jinja, we'll create a folder to store the templates:

```
templates
```

And a section for homework:

```
templates/task
```

And an HTML file:

```
templates/task/index.html
```

With the following content:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>FastAPI</title>
</head>

<body>
  <h1>Hello world</h1>
</body>
</html>
```


In the `api.py` file, we'll create a `Jinja2Templates` instance object that takes the root folder of the templates as a parameter:

`api.py`

```
from fastapi import Request
from fastapi.templating import Jinja2Templates

templates = Jinja2Templates(directory="templates/")
```

And we create a function that receives as a parameter the location of the template to be rendered and the user's request:

```
@app.get('/page')
def index(request: Request):
    return templates.TemplateResponse('task/index.html', {"request": request})
```

If we access from the browser, we will see:

```
Hello world
```

To make the application more reusable, we are going to create a master template:

`templates/task/master.html`

```
<!DOCTYPE html>
<html lang="en">

<head>
    {% block head %}
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>{% block title %} {% endblock %} - FAST API</title>
    {% endblock %}
</head>
<body>
    <div class="container">
        {% block content %}

        {% endblock %}
    </div>

    {% block footer %}
        <p>DL by <a href="#">Contact</a></p>
    {% endblock %}

</body>
</html>
```

And we reuse it in the previously created view:

templates/task/index.html

```
{% extends "task/master.html" %}

{% block head %}
    {{ super() }}
    <meta name="description" content="List of task">
{% endblock %}

{% block title %} List {% endblock %}

{% block content %}
<ul>
    <li>Task Example 1</li>
</ul>
{% endblock %}
```

HTML task list

In this section, we are going to adapt the index page to display a list of tasks in a table; to do this, we return the list of all tasks:

```
@app.get("/")
async def index(request: Request, db: Session = Depends(get_database_session)):
    return templates.TemplateResponse("task/index.html", { "request": request, 'tasks':
getAll(db) })
```

And from the template we iterate the list, we iterate the list:

```
{% block content %}
<table>
    <tr>
        <td>Id</td>
        <td>Name</td>
        <td>Category</td>
        <td>Status</td>
        <td>Options</td>
    </tr>
    {% for t in tasks %}
    <tr>
        <td>
            {{ t.id }}
        </td>
        <td>
            {{ t.name }}
        </td>
    </tr>
    {% endfor %}
</table>
{% endblock %}
```

```

        </td>
        <td>
            {{ t.category.name }}
        </td>
        <td>
            {{ t.status }}
        </td>
        <td>
            --
        </td>
    </tr>
    {% endfor %}
</table>

{% endblock %}

```

Requests in JavaScript

In this section, we will see how to make requests through JavaScript to create, update and delete tasks; the advantage of this approach is that it can be used in real applications built by ourselves, whether these applications are created in Vue, React, Angular, mobile, etc and using the equivalent.

Create a task

The next interaction that we are going to carry out is to create tasks using the template to implement the logic necessary to create the task, that is, without using the documentation and with our own implementation; for this, we will use a **fetch()** request in JavaScript with which we can connect to the HTTP channel through JavaScript like the following:

templates/task/_create.html

```

<button id="createTask">Create task</button>

<script>
    document.getElementById("createTask").addEventListener('click', function () {
        let datos = {
            'name': 'Task 1',
            'category_id': 1,
            'description': 'Description',
            'status': 'done',
        }

        fetch('tasks/', {
            method: 'POST',
            body: JSON.stringify(datos),
            headers: { "Content-type": "application/json; }
        }).then(response => response.json())
            .then(data => console.log(data));
    })

```

```
</script>
```

Currently, the API method for creating tasks receives the data in the request **body** in JSON format (which must be specified in the **Content-type** of the request; thinking about that, we create the above code.

The previous view fragment is included in the **index.html** view in order to use the previous code:

```
templates/task/index.html
```

```
***
{% include 'task/_create.html' %}
{% endblock %}
```

According to the current implementation, to create a task, we must supply the data in the request **body**, which means that we cannot do it using an HTML form (for this we would have to make changes to the API methods, but this is we'll see later); to send the data by body using the fetch, we use the **body** option and pass the data in json format, for this the **JSON.stringify()** function is used.

Now, we are going to adapt the previous code so that the user can place the data of the task to be created; to do this, we will use form fields that are then referenced by **querySelector()** or equivalent using JavaScript:

```
templates/task/_create.html
```

```
<div id="createForm">
  <label for="">Name</label>
  <input type="text" name="name">

  <label for="">Description</label>
  <textarea name="description" id="" cols="30" rows="10"></textarea>

  <label for="">Status</label>

  <select name="status">
    <option value="pending">Pending</option>
    <option value="done">Done</option>
  </select>

  <label for="">Categories</label>

  <select name="category_id">
    <option value=""></option>

    {% for c in categories %}
    <option value="{{ c.id }}">{{ c.name }}</option>
    {% endfor %}
  </select>

  <input type="hidden" name="user_id" value="1">
```

```

    <button id="createTask">Create task</button>
</div>

<script>
    document.getElementById('createTask').addEventListener('click', function () {

        let data = {
            "name": document.querySelector('#createForm input[name="name"]').value,
            "description": document.querySelector('#createForm
textarea[name="description"]').value,
            "status": document.querySelector('#createForm select[name="status"]').value,
            "category_id": document.querySelector('#createForm
select[name="category_id"]').value,
            "user_id": document.querySelector('#createForm input[name="user_id"]').value
        }

        fetch('/tasks/', {
            method: 'POST',
            body: JSON.stringify(data),
            headers: { "Content-type": "application/json" }
        })
            .then(res => res.json())
            .then(data => console.log(data))
        })
</script>

```

From the API method, we supply the necessary categories to build the list:

api.py

```

#templaples
@app.get('/page')
def index(request: Request, db: Session = Depends(get_database_session)):
    categories = db.query(Category).all()
    return templates.TemplateResponse('task/index.html', {"request": request, 'tasks':
crud.getAll(db), 'categories': categories})

```

Important to note that the categories are used in the child view of **_create.html** and not in the one of **index.html**; data is inherited to child views.

Update a task

From the listing table, we are going to place a button to edit a task in the table:

templates/task/index.html

</table>

```
{% include 'task/_update.html' %}
{% endblock %}
```

In addition to the button, custom attributes (those beginning with **data-**) are created so that they can later be referenced by JavaScript and preload the data. With this button, we implement a click event that, when pressed, loads the task information; for the simplicity of the exercise, we will create a new template:

templates/task/_update.html

```
<div id="editForm">
  <label for="">Name</label>
  <input type="text" name="name">

  <label for="">Description</label>
  <textarea name="description" id="" cols="30" rows="10"></textarea>

  <label for="">Status</label>

  <select name="status">
    <option value="PENDING">Pending</option>
    <option value="DONE">Done</option>
  </select>

  <label for="">Categories</label>

  <select name="category_id">
    <option value=""></option>

    {% for c in categories %}
    <option value="{{ c.id }}">{{ c.name }}</option>
    {% endfor %}
  </select>

  <input type="hidden" name="user_id" value="1">
  <input type="hidden" name="id" value="">

  <button id="editTask">Edit task</button>
</div>

<script>

  // listen click button list

  document.querySelectorAll('.edit').forEach((b) => {
    b.addEventListener('click', function () {

      document.querySelector('#editForm input[name="name"]').value =
b.getAttribute('data-name')
```

```

        document.querySelector('#editForm input[name="id"]').value =
b.getAttribute('data-id')
        document.querySelector('#editForm textarea[name="description"]').value =
b.getAttribute('data-description')
        document.querySelector('#editForm select[name="status"]').value =
b.getAttribute('data-status')
        document.querySelector('#editForm select[name="category_id"]').value =
b.getAttribute('data-category')
    })
})

// send data
document.getElementById('editTask').addEventListener('click', function () {

    let data = {
        "name": document.querySelector('#editForm input[name="name"]').value,
        "description": document.querySelector('#editForm
textarea[name="description"]').value,
        "status": document.querySelector('#editForm
select[name="status"]').value.toLowerCase(),
        "category_id": document.querySelector('#editForm
select[name="category_id"]').value,
        "user_id": document.querySelector('#editForm input[name="user_id"]').value
    }

    let id =document.querySelector('#editForm input[name="id"]').value

    fetch('/tasks/'+id, {
        method: 'PUT',
        body: JSON.stringify(data),
        headers: { "Content-type": "application/json" }
    })
        .then(res => res.json())
        .then(data => console.log(data))
    })
</script>

```

As other important changes compared to create, the **toLowerCase()** method is used to convert the status text to lower case and the route and method type of the **fetch()** function are updated to be able to update the tasks.

Delete a task

We place a button to delete the task and indicate the id as a parameter, the id of the task to be deleted:

templates/task/index.html

```

<td>
    <button class="edit" data-id="{{ t.id }}" data-name="{{ t.name }}" data-description="{{
t.description }}"

```

```
        data-category="{{ t.category.id }}" data-status="{{ t.status.name }}">Edit</button>
        <button class="delete" data-id="{{ t.id }}">Delete</button>
    </td>
```

And the listener event that is responsible for carrying out the fetch request with the DELETE type request indicating the id of the task to be deleted:

templates/task/index.html

```
<script>
    document.querySelectorAll('.delete').forEach(b => {
        b.addEventListener('click', function () {
            let id = b.getAttribute('data-id')
            fetch('/tasks/' + id, {
                'method': 'DELETE',
            })
                .then(res => res.json())
                .then(data => console.log(data))
        })
    })
</script>

{% endblock %}
```

HTML forms

As we discussed in the previous section, the reason why we do not use an HTML form to test the API methods; is that they are not compatible; now we're going to make some changes to make the API methods compatible.

schemas.py

```
class TaskBase(BaseModel):
    name: str
    description: Optional[str] = Field("No description",min_length=5)
    status: StatusType

    category_id: int = Field(gt=0)
    class Config:
        from_attributes = True

    @classmethod
    def as_form(
        cls,
        name: str = Form(),
        description: str = Form(),
        status: str = Form(),
        category_id: str = Form(),
    ):

```



```

        return cls(name=name, description=description,
status=status,category_id=category_id)

class TaskRead(TaskBase):
    id:int

class TaskWrite(TaskBase):
    id: Optional[int] = Field(default=None)
    user_id: Optional[int] = Field()
    @classmethod
    def as_form(
        cls,
        name: str = Form(),
        description: str = Form(),
        status: str = Form(),
        user_id: str = Form(),
        category_id: str = Form(),
    ):
        return cls(name=name, description=description,
status=status,category_id=category_id, user_id=user_id)

```

Key points

The **Form** class is used when you need to receive form fields instead of JSON.

This time the Python decorator called **classmethod** is used; whatever method this decorator has, in this example **as_form()** are bound to a class instead of an object; this class method will receive the class as its first argument, in case it needs to be used for anything and is usually called **cls** by convention; class methods can be called in both; by class:

```
TaskWrite.as_form
```

As for objects:

```
task = TaskWrite
task.as_form()
```

Or

```
TaskWrite().as_form()
```

Class methods receive the class as their first argument instead of the instance of that class (as we normally do with methods); that is, you can use the class and its properties inside that method without having to instantiate the class:

```
class MyClass
    @classmethod
    def classmethodcustom(cls, text):
```

```
pass
```

```
MyClass.classmethodcustom("FastAPI")
```

Finally, the class method can return an object of the class.

A class method is a method that is bound to a class instead of its object. It does not require the creation of a class instance.

From the API method, we inject the new function as a dependency:

```
@task_router.post("/", status_code=status.HTTP_201_CREATED) #status_code=201
status.HTTP_200_OK
def add(request: Request, task:TaskWrite = Depends(TaskWrite.as_form), db: Session =
Depends(get_database_session)):
    create(task, db)
```

With this, if we go to the documentation, we will see that now it requests the data based on form fields and not a JSON:

POST /tasks/form-create Addform

Parameters

No parameters

Request body * required

name <small>* required</small> string	<input type="text" value="name"/>
description <small>* required</small> string	<input type="text" value="description"/>
status <small>* required</small> string	<input type="text" value="status"/>
category_id <small>* required</small> string	<input type="text" value="category_id"/>
user_id <small>* required</small> string	<input type="text" value="user_id"/>

And therefore, we can configure an HTML form to create tags:

templates/task/_create.html

```
<!-- form html -->
<form action="/tasks/form-create" method="POST">
  <label for="">Name</label>
  <input type="text" name="name">

  <label for="">Description</label>
  <textarea name="description" id="" cols="30" rows="10"></textarea>

  <label for="">Status</label>

  <select name="status">
    <option value="pending">Pending</option>
    <option value="done">Done</option>
```

```
</select>

<label for="">Categories</label>

<select name="category_id">
  <option value=""></option>

  {% for c in categories %}
  <option value="{{ c.id }}">{{ c.name }}</option>
  {% endfor %}
</select>

<input type="hidden" name="user_id" value="1">

<button type="submit">Create task by Form</button>
</form>
```

You can do the same process to update a task or similar methods you need to implement.

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.6>

Chapter 9: Dependencies

Dependencies are nothing more than a set of attributes or functions passed through function parameters that are necessary for the API methods to do their job correctly; we previously used the dependencies to inject the connection to the database:

```
from fastapi import *** Depends
from sqlalchemy.orm import Session

from database.database import get_database_session
***
@app.get("/page/")
def page(db: Session = Depends(get_database_session)):
    get_task(db,1)
```

But the dependencies can be used in many other scenarios; as you can see, to do the dependency injections, the class called **Depends** is used; its operation is very simple, when the **Depends** class is placed, FastAPI knows that it has to resolve the function it receives as a parameter (in the previous example, the function called **get_database_session**) each time an API method is called that maintains this dependency; in this implementation, the dependency returns a value (database connection) together with the user's request and the rest of the parameters to resolve the request.

In this way we can easily inject parameters to reuse codes, as it would be in the previous example, the connection to the database; unlike other parameters that are defined in API methods:

```
@app.post("/{task}")
def add(task: str):
    task_list.append(task)
    return { "tasks": task_list }
```

These parameters can be resolved by FastAPI; however, you can also use parameters per query (**Query** class) and inject by dependencies; for example:

```
def pagination(page:Optional[int] = 1, limit:Optional[int] = 10):
    return{'page':page-1,'limit':limit}

@app.get("/{task}")
def index(pag: dict = Depends(pagination)):
    return pag
***
@app.get("/{user}")
def index(pag: dict = Depends(pagination)):
    return pag
```

In this way, you can create functions with a specific behavior and that receive specific parameters and be able to easily reuse them by inheriting the behavior in other functions by injecting them through a dependency.

Path dependencies (API method decorator)

Another possible use of dependencies is to inject directly into the API method's decorator; this is useful when the function associated with the dependency is not required to return a value, if not, some verification; for example, verify a token:

```
from fastapi import ***, HTTPException, status, Header

def validate_token(token: str = Header()):
    if token != "TOKEN":
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,)

@app.get("/get-task", dependencies=[Depends(validate_token)])
def protected_route(index: int):
    return {"hello": "FASTAPI"}
```

In these cases, the dependency does not have to return a value; This type of validation is useful to execute a process before the API function, and if there is a problem (in the previous example that the token is invalid) an exception is executed and with this, avoid executing the body of the API function.

In the previous example, we will see that the token set in the dependency and the rest of the additional parameters are requested, such as the index argument.

Define dependencies as variables

Another advantage that we have when working with dependencies is the fact that we can improve code reuse and with this, their readability; suppose we have a function with many arguments:

```
@app.get("/items/")
def read_items(user: User = Depends(get_current_user)):
    ***
```

And that they are used in various functions:

```
@app.get("/items/")
def read_items(user: User = Depends(get_current_user)):
    ***

@app.post("/items/")
def create_item(*, user: User = Depends(get_current_user), item: Item):
    ***

@app.get("/items/{item_id}")
def read_item(*, user: User = Depends(get_current_user), item_id: int):
    ***

@app.delete("/items/{item_id}")
def delete_item(*, user: User = Depends(get_current_user), item_id: int):
```

```
***
```

We can easily create an annotated dependency and reference this variable in functions instead:

```
from typing_extensions import Annotated
***
CurrentUser = Annotated[User, Depends(get_current_user)]

@app.get("/items/")
def read_items(user: CurrentUser):
    ***

@app.post("/items/")
def create_item(user: CurrentUser, item: Item):
    ***

@app.get("/items/{item_id}")
def read_item(user: CurrentUser, item_id: int):
    ***

@app.delete("/items/{item_id}")
def delete_item(user: CurrentUser, item_id: int):
    ***
```

This can be considered the most important advantage of using dependencies together with annotations, since one of the potential drawbacks of FastAPI is that the API methods tend to grow quite horizontally in terms of the definition of the arguments to them, but with this technique it is possible to shorten the size of the function definition of the API methods.

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.7>

Chapter 10: Introduction to Middleware

A "middleware" is an intermediary, it is nothing more than a function that we implement and it is executed every time a request is received; therefore, middlewares are functions that are established at the application level. In this chapter, we are going to see an introduction to middleware.

Middlewares can be executed in two ways:

1. Before calling the API function.
2. After the response has been generated, that is, before returning the response.

To show the operation of middleware in a more exemplary way, consider the following steps:

1. It takes every request that comes to the app.
2. Once inside the middleware, you can manipulate the request or run any additional code, such as a verification of the authenticated user.
3. As the last step, the middleware passes the request to be processed by the rest of the application; that is, it passes the request to an API method and the application follows its normal flow.

There is also another way:

1. Once the response has been generated by the API methods, it is possible to configure a middleware that handles the response or execute any additional code.
2. As the last step, the middleware returns the response.

Ultimately, middleware is useful when the user wants to execute some additional common code for the application before or after the user's request.

Create a middleware

To create a middleware in a project in FastAPI, we must use the `@app.middleware("http")` decorator in the function/middleware. The middleware can execute any type of code, but, in the end, it must execute a `call_next()` function that will receive the request as a parameter and allows it to continue with the normal flow of the application.

A middleware receives two parameters:

1. The Request.
2. And the next function to execute.

Finally, we implement a middleware like the following:

```
import time
***
@app.middleware("http")
async def add_process_time_to_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response
```


FastAPI, having support for ASGI-type asynchronous servers, can perfectly and efficiently process asynchronous-type functions like the previous one (the middleware); and asynchronous functions can be defined in any of the API methods; for example:

```
@app.get('/')
async def read_results():
    results = await some_library()
    return results
```

It is important to clarify that the asynchronous definition for API functions is aimed at using background operations or executing operations concurrently.

Asynchronous functions have not been used in this book at the API (endpoint) function level simply because they were not necessary and also to avoid generating additional logic and unnecessarily complicating the exercises.

Finally, in the middleware function called `add_process_time_header()` defined above, we query the current time:

```
start_time = time.time()
```

Then, we call the next function to be executed, that is, any of the API methods, which is an asynchronous request:

```
response = await call_next(request)
```

Then we consult the current time:

```
process_time = time.time() - start_time
```

If we subtract the first time (**start_time**) with the time taken after the execution of the next function to be executed, we have the time it took to resolve the API method, which is injected as a parameter in the HEADER:

```
response.headers["X-Process-Time"] = str(process_time)
```

This time will appear when making requests to any of the API methods. finally, the response must always be returned, that is, call the following function, which are the API methods:

```
return response
```

If you wanted to return an error based on a status code, you could use the class **JSONResponse**:

```
from fastapi.responses import JSONResponse
***
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    ***
    return JSONResponse(status_code=401, content='No Authenticate')
```

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.8>

Chapter 11: Authentication

In most scenarios we will want to at least protect parts of the API; whether you're using a templating engine like Jinja or not, you'll almost always need to protect your API methods; for this, the most basic way is to use an authentication system, either based on session or authentication tokens; with these mechanisms, the user is granted rights so that he can access the private methods of the API and we can know at all times (done this implementation) when the user manipulates the data and at what level.

In this section we will see some authentication methods that are directly integrated into the automatic documentation and other custom schemes that consist of a username and password, integrating with the database and the generation of authentication or access tokens.

Basic authentication

The most basic authentication scheme that can be considered is where the user's credentials are managed based on a token established in the request header, specifically in the header called **Authorization** to be later consulted and evaluated by the request application:

api.py

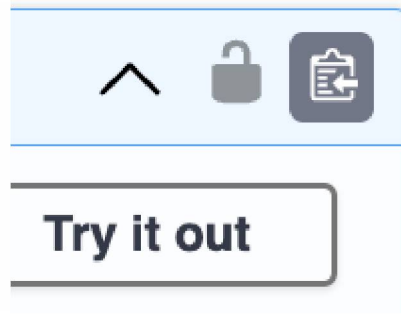
```
from fastapi.security import APIKeyHeader
from fastapi import Depends, FastAPI, HTTPException, status

API_KEY_TOKEN = "SECRET_PASSWORD"

api_key_header = APIKeyHeader(name="Token")
@app.get("/protected-route")
async def protected_route(token: str = Depends(api_key_header)):
    if token != API_KEY_TOKEN:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN)
    return {"hello": "FASTAPI"}
```

In the example above, a generic token is set using **API_KEY_TOKEN** and it is checked if the token supplied in the request header is equal to the token; the **APIKeyHeader** class (which returns a 403 error when the token is not supplied) is used to retrieve a header value (in the above example the variable called **token**). Finally, it is established as one more dependency of the request.

If we go to the documentation, in the method created previously, we will see a lock, which indicates that it is protected:

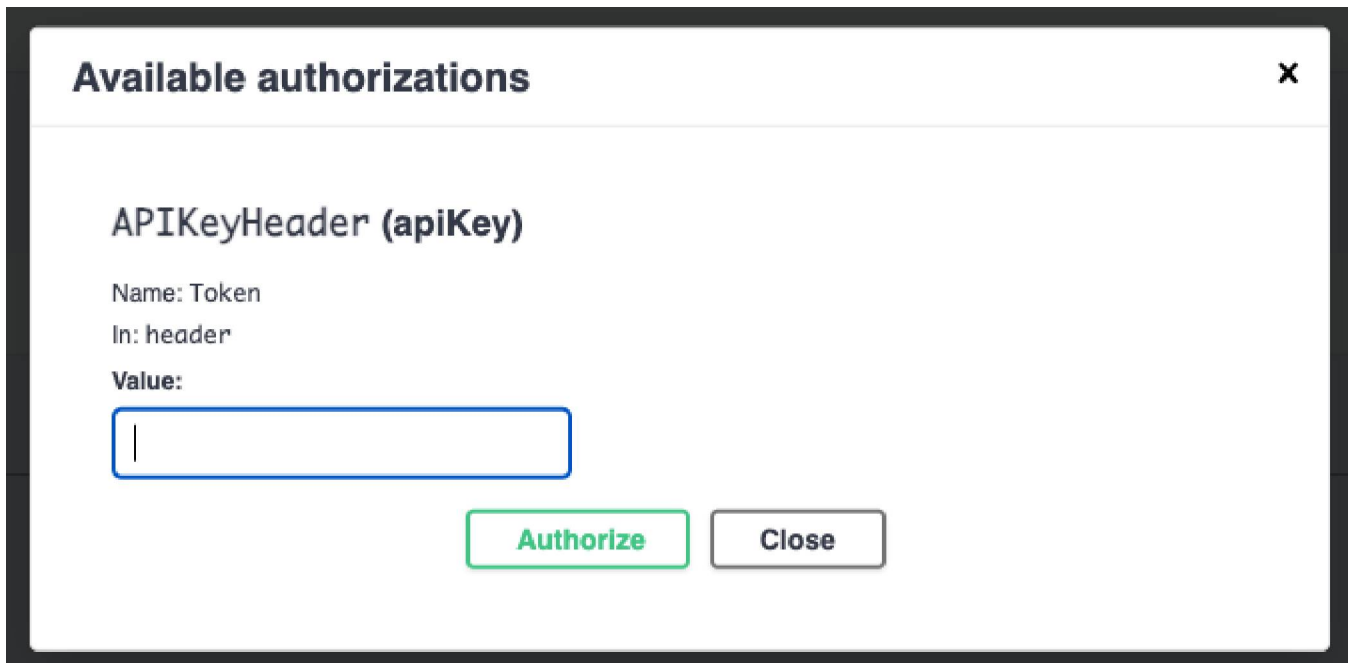


If we make the request without authenticating ourselves, we will see a 403 error (the one established above):

```
Code Details
403
Undocumented
Error: Forbidden

Response body
Download
{
  "detail": "Not authenticated"
}
```

If we click on the lock and set the token correctly (a valid token):



The request will be successfully processed with a status code of type 200.

The problem with this approach is that we can only protect one route; we can create a function that performs the above check:

```
from fastapi.security import APIKeyHeader
```

```
from fastapi import Depends, FastAPI, HTTPException, status
```

```
API_KEY_TOKEN = "SECRET_PASSWORD"
```

```
async def authenticate(token: str = Depends(APIKeyHeader(name="Token"))):  
    if token != API_KEY_TOKEN:  
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN)  
    return token
```

And it is injected into each of the API methods in which we require the user to be authenticated; for example:

```
@app.get("/page/")  
def page(db: Session = Depends(get_database_session), dependencies=Depends(authenticate)):  
    print(getAll(db))  
    #create_user(db)  
    print(dependencies)  
    return {"page": 1}
```

These examples offer a very basic authentication scheme where the received token is verified against a constant; authenticated users are not handled and there is no way to know which user is making the request.

Database authentication

The previous authentication can be used to define simple schemes, but, in most cases, it is necessary to handle more complete schemes, in which we will have multiple users with the already known username/password combination to be able to enter the application, for these cases, we can use the following scheme.

For this development, we are going to need to install a package to the project with which we can convert plain texts into hashes, specifically we will use it to generate a hash of the user's password:

```
$ pip install 'passlib[bcrypt]'
```

Next, we will modify the users model to specify the password column (in hash format) and create a relational entity to the user with which we will manage the access tokens:

database\models.py

```
class User(Base):  
    """  
    hashed_password = Column(String(255))  
  
class AccessToken(Base):  
    __tablename__ = 'access_tokens'  
    user_id = Column(Integer, ForeignKey('users.id'), primary_key=True)  
    access_token = Column(String(255))  
    expiration_date = Column(DateTime(timezone=True))  
    user = relationship('User', lazy="joined")
```

With each of the columns of the previous relation, we have:

- **user_id** allows you to handle the foreign relationship with users. It is important to note the primary key for this column, which means that there can only be one access token per user.
- **access_token** holds the access token of the authenticated user.
- **expiration_date** specifies the lifetime of the token.

Remember to delete all the tables in the database to create the new relationship and see the change reflected in the user model.

We create a schema of the new entity and modify the user entity to indicate the password field:

schemas.py

```
class User(BaseModel):
    name: str = Field(min_length=5)
    surname: str
    email: EmailStr
    website: str #HttpUrl
    class Config:
        from_attributes = True

class UserCreate(User):
    password: str

class UserDB(User):
    hashed_password: str

class AccessToken(BaseModel):
    user_id: int
    access_token: str
    expiration_date: datetime
    class Config:
        from_attributes = True
```

For the user, if a user is going to be created, then the password is in plain text, since it is introduced by the user (**UserCreate.password**), but, when it comes to the user obtained from the database, the password is in hash format (**UserDB.hashed_password**).

To manage the password, we will use a new file:

authentication\password.py

```
import secrets
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"])

def get_password_hash(password: str) -> str:
```

```

    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)

def generate_token() -> str:
    return secrets.token_urlsafe(32)

```

Explanation of the above functions

- With **schemes=['bcrypt']** the encryption algorithm to be used is specified.
- With the function **get_password_hash()** it allows to create a hash of the password in plain text; this function will be used later when registering a user.
- with the function **verify_password()** allows you to verify a password; the main characteristic of hashes is that they cannot be reverted to a plain text once converted, therefore, there is a function to compare if a hash corresponds to a plain text, in this case, the plain text corresponds to the password in plain text; this function will be used later to perform the login.
- Using the **generate_token()** function allows you to generate a token using the Python function **token_urlsafe()** which allows you to generate 32-byte safe tokens.

To carry out the user authentication process and generate the access token, we will use a new file:

authentication\authentication.py

```

from sqlalchemy.orm import Session
from datetime import datetime, timedelta

# from time
from database.models import User, AccessToken
from authentication.password import verify_password, generate_token

def authenticate(email: str, password: str, db: Session) -> User|None:
    user = db.query(User).filter(User.email == email).first()

    if user is None:
        return None

    if not verify_password(password, user.hashed_password):
        return None

    return user

def create_access_token(user:User, db: Session) -> AccessToken:
    tomorrow = datetime.now() + timedelta(days=1) # time.time + 60 * 60 * 24

    accessToken = AccessToken(user_id=user.id, expiration_date=tomorrow,
access_token=generate_token())

    db.add(accessToken)

```

```
db.commit()
db.refresh(accessToken)

return accessToken
```

Explanation of the above functions

- The **authentication()** function is not very complicated, it simply looks up the user by email and checks the plaintext password against the hash of the password stored in the database.
- The **create_access_token()** function allows you to generate an access token, for this, a record is inserted in the **access_tokens** table specifying the user id, the date (tomorrow to indicate the expiration date of the token) and generate the token for this, the **generate_token()** function created in the previous section is used.

Methods for the API

We will create a new file where we implement the options to register a user and create the user's token (login):

user.py

```
from fastapi import APIRouter, HTTPException, status, Depends
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session

from schemes import UserCreate
from authentication import password, authentication
from database import database, models

user_router = APIRouter()

@user_router.post('/token')
def create_token(form_data : OAuth2PasswordRequestForm =
Depends(OAuth2PasswordRequestForm), db: Session = Depends(database.get_database_session)):
    email = form_data.username
    password = form_data.password

    user = authentication.authenticate(email,password,db)

    if user is None:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)

    token = authentication.create_access_token(user, db)
    return {"access_token": token.access_token}

@user_router.post('/register', status_code=status.HTTP_201_CREATED)
def register(user: UserCreate, db:Session = Depends(database.get_database_session)): # ->
models.User

    user_exist = db.query(models.User).filter(models.User.email == user.email).first()
```

```

if user_exist:
    raise HTTPException(status_code=status.HTTP_409_CONFLICT,
                        detail="User email already exist")

hashed_password = password.get_password_hash(user.password)
print(hashed_password)

userdb = models.User(name=user.name,
                    email= user.email,
                    surname = user.surname,
                    website= user.website,
                    hashed_password=hashed_password)

db.add(userdb)
db.commit()
db.refresh(userdb)

return userdb

```

With the **OAuth2PasswordRequestForm** class, it is injected as a dependency of the function to generate the token and allows to declare in a form body with the following fields:

- Username.
- Password.
- An optional scope field.
- A key for the public and private client is also optional.

Of course, you could use the similar Pydantic classes or the **Body**, **Form**, or classes instead, something like:

```

create_token(email: str = Form(), password: str = Form())

```

Finally, the previous routes are included in the application:

api.py

```

from user import user_router
***
app.include_router(user_router)
***

```

And with this we will have:

Request body ^{required}

grant_type string pattern: password	<input type="text" value="grant_type"/>	<input checked="" type="checkbox"/>	Send empty value
username * required string	<input type="text" value="username"/>		
password * required string	<input type="text" value="password"/>		
scope string	<input type="text" value="scope"/>	<input checked="" type="checkbox"/>	Send empty value
client_id string	<input type="text" value="client_id"/>	<input checked="" type="checkbox"/>	Send empty value
client_secret string	<input type="text" value="client_secret"/>	<input checked="" type="checkbox"/>	Send empty value

Protect routes with authenticated user

The process for protecting routes per authenticated user is almost the same as what we created earlier for basic authentication; to do this, we create a function that, given the token, returns the authenticated user, a 401 error if the token does not exist:

api.py

```
def protected_route(db: Session = Depends(get_database_session), token: str = Depends(api_key_header)):
    user = db.query(User).join(AccessToken).filter(AccessToken.access_token == token).first()

    if user is None:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)
```

Which is then injected into the methods that require the user to be authenticated by a dependency:

api.py

```
@app.get("/get-task", dependencies=[Depends(protected_route)])
def protected_route(index: int):
    return {"hello": "FASTAPI"}
```

We can improve the previous verification implementation, creating a new function based on it, with which we will verify the authentication token supplied in the header, with the token, the existence of the token and the expiration date are verified:

authentication\authentication.py

```
from fastapi import Depends
from sqlalchemy.orm import Session
from database.database import get_database_session
from fastapi.security import APIKeyHeader

api_key_header = APIKeyHeader(name="Token")
def verify_access_token(token: str = Depends(api_key_header), db: Session =
Depends(get_database_session)) -> dict:
    access_token = db.query(AccessToken).join(User).filter(AccessToken.access_token ==
token).first()

    if access_token is None:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Token invalid"
        )

    if datetime.now() > access_token.expiration_date:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Token expired!"
        )
    return access_token.user
```

Which is then easily used anywhere in the application; for example:

api.py

```
@app.get("/get-task")
def protected_route(index: int, user: User = Depends(verify_access_token)):
    return {"hello": "FASTAPI"}
```

Logout

The next functionality to implement is to destroy the token; for it:

authentication\authentication.py

```

api_key_token = APIKeyHeader(name='Token')
def logout(token: str = Depends(api_key_token), db: Session =
Depends(get_database_session)):
    access_token = db.query(AccessToken).join(User).filter(AccessToken.access_token ==
token).first()

    if access_token is None:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST)

    db.delete(access_token)
    db.commit()

```

And we implement the API method that calls the above function:

user.py

```

@user_router.delete("/logout", status_code=status.HTTP_200_OK,
dependencies=[Depends(authentication.logout)])
def logout():
    return {'msj': 'ok'}

```

Protecting API methods using the OAuth2PasswordBearer dependency

In this section, we'll learn how to use the **OAuth2PasswordBearer** dependency to protect route methods; to do this, let's implement the dependency function that will be injected into the API method routes. This function will serve as the only source to retrieve a user from an active session using the token; we will apply the following changes:

authentication\authentication.py

```

from fastapi.security import ***, OAuth2PasswordBearer
***
auth_scheme = OAuth2PasswordBearer(tokenUrl="/token")
def verify_access_token(token: str = Depends(auth_scheme), db: Session =
Depends(get_database_session)) -> dict:
    access_token = db.query(AccessToken).join(User).filter(AccessToken.access_token ==
token).first()
    ***

```

The first thing to note is that we use FastAPI's **OAuth2PasswordBearer** dependency which behaves similarly to the **OAuth2PasswordRequestForm** we saw earlier; with this class, the endpoint for getting a token is specified using the **tokenUrl(/token)** argument; through the automatic documentation, you can call the authentication function that is responsible for generating the access token through the login that we saw previously:

Available authorizations x

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)

Token URL: /token
Flow: password

username:

password:

Client credentials location:

client_id:

client_secret:

Verify access token at token creation time

An important factor is to verify and return the access token; that is to say:

- If you try to login and there is already an access token created and valid (not expired), it is returned.
- If you try to log in and there is already an access token created but it has already expired, it is deleted and a new one is created.

We will add these new conditions in the authentication method:

database\authentication.py

```
def create_access_token(user: User, db: Session) -> AccessToken:

    access_token = db.query(AccessToken).filter(AccessToken.user_id == user.id).first()
    if access_token is not None:
```

```
    if datetime.now() > access_token.expiration_date:
        db.delete(access_token)
    else:
        return access_token
***
```

Chapter source code:

<https://github.com/libredesarrollo/curso-libro-fast-api-crud-task-1/releases/tag/v0.9>

Capítulo 12: Annotations, Ellipsis (...) notation, and return types

FastAPI, like all modern development technologies, evolves over time; for this reason, if you consult old documentation, you can see slight in the syntax of various components that have been added, removed or simply changed and this is what we want to cover in this chapter.

Ellipse notation

For some classes in FastAPI the three ellipsis (...) notation is used, this notation is known as Ellipsis to indicate as a default argument value; for example, to indicate that the argument is mandatory, which is the implementation that several FastAPI classes have by default, for example, that of:

```
Path(...)
```

So, if for example, we have the following implementation:

```
@app.get("/page/")
def page(page: int = Path(gt=0), size: int = Query(10, le=100)):
    return {"page": page, "size": size}
```

To use the Ellipsis notation together with our arguments, we have:

```
@app.get("/page/")
def page(page: int = Path(..., gt=0), size: int = Query(10, le=100)):
    return {"page": page, "size": size}
```

It is important to mention that this notation has multiple implementations in Python and can be used in various contexts, meaning a different operation than the one discussed above; for example, to access and split multidimensional arrays.

Until now, the use of this operator has not been presented, to avoid complicating the readability of the code and not hinder the implementation of the different functionalities of the framework that we present, but if you look for other materials on the internet, you will see that the use of Ellipsis often appears; this notation, in recent versions can be omitted; if we review the following release:

<https://github.com/tiangolo/fastapi/releases/tag/0.78.0>

Which corresponds to an old version of FastAPI, you will see that in the description, place the following:

```
Add support for omitting ... as default value when declaring required parameters with:
```

```
Path()
Query()
Header()
Cookie()
```

```
Body()  
Form()  
File()
```

Since version 0.78, the use of the Ellipsis operator can be omitted in the previously indicated classes; by default, when using each of these classes, the parameter is required, so it would not be necessary to use it.

(Ellipsis) was the way to declare a required parameter in FastAPI. However, since version 0.78.0, you can override the default value to do this.

Annotated

Annotations are a modern mechanism in Python that allows adding metadata to parameters in Python and that we can also use in FastAPI; with annotations, we have a more expressive and cleaner syntax when using parameters; It's important to mention that the use of annotations is optional and a relatively new change in FastAPI.

It is also important to mention that with the Annotated we can do more than a metadata annotation; you can even easily reuse the definition as we will see in the official example in FastAPI.

Annotated are a standard in Python.

In the exact words of the FastAPI team:

The version 0.95.0 adds support for dependencies and parameters using **Annotated** and recommends its usage.

This has several benefits, one of the main ones is that now the parameters of your functions with Annotated would not be affected at all.

If you call those functions in other places in your code, the actual default values will be kept, your editor will help you notice missing required arguments, Python will require you to pass required arguments at runtime, you will be able to use the same functions for different things and with different libraries (e.g. Typer will soon support Annotated too, then you could use the same function for an API and a CLI), etc.

Because Annotated is standard Python, you still get all the benefits from editors and tools, like autocompletion, inline errors, etc.

One of the biggest benefits is that now you can create Annotated dependencies that are then shared by multiple path operation functions, this will allow you to reduce a lot of code duplication in your codebase, while keeping all the support from editors and tools.

For example, you could have code like this:

```
def get_current_user(token: str):  
    # authenticate user  
    return User()
```

```

@app.get("/items/")
def read_items(user: User = Depends(get_current_user)):
    ...

@app.post("/items/")
def create_item(*, user: User = Depends(get_current_user), item: Item):
    ...

@app.get("/items/{item_id}")
def read_item(*, user: User = Depends(get_current_user), item_id: int):
    ...

@app.delete("/items/{item_id}")
def delete_item(*, user: User = Depends(get_current_user), item_id: int):
    ...

```

There's a bit of code duplication for the dependency:

```

user: User = Depends(get_current_user)
...the bigger the codebase, the more noticeable it is.

```

Now you can create an annotated dependency once, like this:

```

CurrentUser = Annotated[User, Depends(get_current_user)]

```

And then you can reuse this Annotated dependency:

```

CurrentUser = Annotated[User, Depends(get_current_user)]

```

```

@app.get("/items/")
def read_items(user: CurrentUser):
    ...

@app.post("/items/")
def create_item(user: CurrentUser, item: Item):
    ...

@app.get("/items/{item_id}")
def read_item(user: CurrentUser, item_id: int):
    ...

@app.delete("/items/{item_id}")
def delete_item(user: CurrentUser, item_id: int):

```

You can get more information at:

<https://github.com/tiangolo/fastapi/releases/tag/0.95.0>

As important points with the annotations in Python we have:

- Python itself doesn't do anything with annotations. And for editors and other tools, the type is still the basic primitives like string, or integers.
- The important thing to remember is that the first type parameter you pass to the Annotated is the primitive or type in general. The rest is just metadata.

Examples using Annotated

Knowing the importance of annotations in FastAPI and in Python in general, and that it is a change that can be considered recent in FastAPI as well as being optional, we are going to see a series of examples where we will go from the classic definition of parameters to annotations; in this way, you can know in a practical way, the syntax of the annotations on the project that we have carried out up to this moment:

```
def page(page: Annotated[int, Query(ge=1, le=20, title='Esta es la pagina que quieres ver')] = 1, size: Annotated[int, Query(ge=5, le=20, title='Cuantos registros por pagina')] = 5 ):
# def page(page: int = Query(1, ge=1, le=20, title='Esta es la pagina que quieres ver'), size: int = Query(5, ge=5, le=20, title='Cuantos registros por pagina')):
    ***

def phone(phone: Annotated[str, Path(pattern=r"^(?!\+[\d]{1,3}\?)\s?([\d]{1,5})\s?([\d][\s\.-]?)\{6,7\}$" )]):
# def phone(phone: str = Path(pattern=r"^(?!\+[\d]{1,3}\?)\s?([\d]{1,5})\s?([\d][\s\.-]?)\{6,7\}$"):
    ***

def validate_token(token: Annotated[str , Header()]):
# def validate_token(token: str = Header()):
```

models.py

```
from fastapi import File

@task_router.post("/files/")
async def create_file(file: Annotated[bytes, File()]):
def create_file(file: bytes = File()):
    return {"file_size": len(file)}

from typing import List

async def image(images: List[UploadFile] = File(...)):
    for image in images:
        with open(image.filename, "wb") as buffer:
            shutil.copyfileobj(image.file, buffer)
```

For more examples and details, you can consult the official documentation where you will see that several of the different examples are presented with variants, such as using annotations or not.

Return types

Important to mention that you can put the return types in the functions:

```
def pagination(page:Optional[int] = 1, limit:Optional[int] = 10): -> dict
    return {'page':page-1, 'limit':limit}
```

If it does not return a value:

```
def validate_token(token: str = Header()) -> None:
    if token != "TOKEN":
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)
```

Another example that we can show is the CRUD of tasks:

```
@task_router.post("/form-create",status_code=status.HTTP_201_CREATED)
def addForm(task: TaskWrite = Depends(TaskWrite.as_form), db: Session =
Depends(get_database_session)) -> dict:
    return { "tasks": crud.create(task,db=db) }
```

And if we try to create a task, we will get an exception like the following:

```
Unable to serialize unknown type: <database.models.Task object at 0x103fd3250>
```

Since an instance of the SQLAlchemy class is not serializable, we'll need to return an instance of Pydantic instead; therefore:

```
@task_router.post("/form-create",status_code=status.HTTP_201_CREATED)
def addForm(task: TaskWrite = Depends(TaskWrite.as_form), db: Session =
Depends(get_database_session)) -> dict:
    return { "tasks": Task.from_orm(crud.create(task,db=db)) }
```

And we will be able to create the task perfectly.

Chapter 13: Testing in FastAPI applications

Testing is a crucial part of any application that we are going to create, regardless of the technology, it is always recommended to carry out automatic tests to test the system when new changes are implemented; in this way we save a lot of time since there is no need to carry out many of the tests manually, if not, by executing a simple command.

Unit tests consist of testing components individually; in the case of the application we have built, it would be each of the API methods, as well as any other dependencies on these methods; thus, when these automated tests are run, if the application passes all the tests, it means that no errors were found, but if it does not pass the tests, it means that changes need to be made at the application level or implemented unit tests.

pytest, for unit tests

To perform unit tests, we will use the pytest library, which is a library for Python unit tests (and not exclusive to FastAPI). Let's install pytest in our project by:

```
$ pip install pytest
```

Creating the first unit tests

As good practice, we'll create a folder called tests to perform unit tests, and a file to perform the first few tests whose names usually contain the test_ prefix:

```
tests/test_math_operations.py
```

In which, we create some functions with which they solve the mathematical operations:

```
tests/test_math_operations.py
```

```
def add(a: int , b: int) -> int:
    return a + b
def subtract(a: int, b: int) -> int:
    return b - a
def multiply(a: int, b: int) -> int:
    return a * b
def divide(a: int, b: int) -> int:
    return b / a
```

At the moment, they are not tests destined to the application, if not, basic mathematical operations, but, with them we will be able to know in a better way the operation of the unit tests; now, in the above file, we define the following functions, which correspond to unit tests:

```
tests/test_math_operations.py
```

```
***
# test
def test_add() -> None:
```

```
    assert add(1,2) == 3
def test_subtract() -> None:
    assert subtract(5,2) == 3
def test_multiply() -> None:
    assert multiply(10,10) == 100
def test_divide() -> None:
    assert divide(25,100) == 0
```

As you can see, these are just Python functions, but, using the **assert** keyword which is used to verify the outputs; specifically, we will be testing that the arithmetic operations are equal to the expected values, that is, if we add 1+1 for the addition operation, the expected result is 2.

Finally, to test the previous tests we execute:

```
$ pytest tests/test_math_operations.py
```

You can also just run:

```
$ pytest
```

To run all the unit tests, which in this case corresponds to only one file; in either case we will have an output like the following:

```
===== test session starts
=====
platform darwin -- Python 3.11.2, pytest-7.4.0, pluggy-1.2.0
rootdir: ***
plugins: anyio-3.7.1
collected 4 items

tests/test_math_operations.py ....
[100%]

===== 4 passed in 0.01s
=====
```

In which, as you can see, all the operations were executed satisfactorily, therefore, the application passed the tests; one factor that you must take into account is that, even if the application passes the tests, it does not guarantee that it does not have errors, simply that enough tests were not done or defined to test all possible scenarios; even so, it is a good meter to know the state of the application and to repair possible problems.

If, for example, we put an invalid output at the time of doing the unit tests:

```
def test_add() -> None:
    assert add(1, 2) == 4
```

We will see an output like the following:

```
===== FAILURES
=====
_____ test_add

def test_add() -> None:
>     assert add(1, 2) == 4
E     assert 3 == 4
E     + where 3 = add(1, 2)

tests/test_math_operations.py:17: AssertionError
===== short test summary
info =====
FAILED tests/test_math_operations.py::test_add - assert 3 == 4
===== 1 failed, 3 passed in
0.02s =====
```

In which it clearly indicates that an error occurred and that it corresponds to the addition operation:

```
FAILED tests/test_math_operations.py::test_add - assert 3 == 4
```

If, on the contrary, the error is at the application level, for example, in the addition function, instead of adding a+b, we add a+a:

```
def add(a: int , b: int) -> int:
    return a + a
```

We will have as output:

```
FAILED tests/test_math_operations.py::test_add - assert 2 == 4
```

But, if in the output function it was like the following:

```
def test_add() -> None:
    assert add(1, 1) == 2
```

We will see that the unit test is accepted, although we clearly have problems at the application level; therefore, it is not a sufficient condition that the application passes the tests to indicate that the application does not have problems, it is simply that not enough tests were created or the tests were not implemented correctly.

Delete repetition with pytest fixtures

Pytest **fixtures** are nothing more than reusable functions whose implementation returns the data needed to evaluate in test functions; they are useful for executing code that goes into the endpoints, for example by setting the construct to a parameter of a Pydantic class:

tests/test_fixture.py

```
import pytest
```

```
#from ..schemes import Category
from tasks.schemes import Category

@pytest.fixture
def category() -> Category:
    return Category(name="Book FastAPI")

def test_event_name(category: Category) -> None:
    assert category.name == "Book FastAPI"
```

Where "tasks" is the name of the project:

```
vtasks
  bin
  include
  lib
  tasks
```

To run the above test, we have:

```
$ pytest tests/test_fixture.py
```

And we will have an output like the following:

```
===== 1 passed in 0.33s
=====
(vtasks) andrescruz@Mac-mini-de-Andres tasks % pytest tests/test_fixture.py
===== test session starts
=====
platform darwin -- Python 3.11.2, pytest-7.4.0, pluggy-1.2.0
rootdir: /Users/andrescruz/Desktop/proyectos/fastapi/curso/vtasks/tasks
plugins: anyio-3.7.1
collected 1 item

tests/test_fixture.py F
[100%]
```

Try placing an invalid check like:

```
category.name == "Book FastAPI 2"
```

And evaluate the result.

If at the time of executing the unit test, we see an error like the following:

```
ModuleNotFoundError: No module named 'tasks'
```

or

```
ModuleNotFoundError: No module named 'schemes'
```

Convert the folders to Python modules by creating an empty file called `__init__.py`:

```
tasks
  __init__.py
  ***
  tests
    ***
    __init__.py
```

Extra: Loop event

The event loop is a fairly extensive topic in Python, but in essence, it allows us to execute asynchronous tasks of all kinds such as I/O operations, execute threads or, in our case of interest, execute network connections to test our API; for the unit tests that we are going to implement, it is necessary to create an event loop since `pytest-asyncio`, which is a package that we are going to install, executes each of the functions implemented to perform tests in an event loop through a function called `event_loop()`; this function will be executed by all the asynchronous tests automatically requested to be able to internally make the connection with the API methods; to get a basic understanding of the event loop in Python, let's run a few little experiments.

Let's start by installing two packages for our project:

```
$ pip install httpx pytest-asyncio
```

- The **pytest-asyncio** package is a plugin for **pytest** which allows you to run asynchronous code in unit tests by **pytest** using the **asyncio** library.
 - **asyncio** is a library for writing asynchronous code using the **async/await** syntax.
- **httpx** is an HTTP client for Python, that is, with this package, we can make requests in the HTTP channel.

Both packages are necessary to be able to connect to the methods created for our API (through the HTTP channel) and make the requests, which, being HTTP requests, are asynchronous.

As a first example to understand the event loop in Python, we have the process that initializes the loop and executes an asynchronous task (`taskAsync()`), which consists of printing a message on the console:

```
tests\demo\loop1_example.py
```

```
import asyncio

async def taskAsync(text = "Message"):
    print(text)

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
# loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(taskAsync())
```

```
except KeyboardInterrupt:
    pass
finally:
    loop.close()
```

Explanation of the previous code

- We create a new loop with `asyncio.new_event_loop()` and assign it as an entry point to execute the code asynchronously using `asyncio.set_event_loop(loop)`.
- `run_until_complete()`, runs the task until it finishes and will block the execution of the next code.
- `KeyboardInterrupt`, the exception is handled when the user uses the keyboard key combination of "CTRL + C" or "CTRL + Z" to stop the running process from the console/terminal.

If we execute:

```
$ python .\tests\demo\loop1_example.py
```

We will see that the "Message" text that corresponds to the task to be executed is printed by the console.

The important thing to note from the previous code is that we have different functions to be able to define what will be the scope to execute the code asynchronously; for the example above, it would be:

```
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
# loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(taskAsync())
except KeyboardInterrupt:
    pass
finally:
    loop.close()
```

In which, we first create the channel so that the operating system can create a thread to execute the task asynchronously:

```
loop = asyncio.new_event_loop()
# loop = asyncio.get_event_loop()
asyncio.set_event_loop(loop)
```

From this point, we can execute the asynchronous task:

```
loop.run_until_complete(taskAsync())
```

And like any process, it closes and the resources are released, and at this point, the tasks or asynchronous tasks have finished, therefore, it corresponds to the closing of the program:

```
loop.close()
```


It is a process similar to when we work with a file, that when opening the file for editing or reading, it must be closed and with this, the resources are released.

As a variant, we indicate in the loop that it runs forever, therefore, the process would never end and it would not be released:

tests\demo\loop1_example.py

```
import asyncio

async def taskAsync(text = "Message"):
    print(text)

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
# loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(taskAsync())
    loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    loop.close()
```

We will see that the message is displayed only once and the application does not finish executing, leaving the console with the active process/loop. This is a very interesting case, since we can clearly see the two elements in this program:

1. The task, which can be anything like a request using the HTTP channel, sending emails, managing a file, which in this example is to print a message on the screen using the **taskAsync()** function.
2. The loop, which is a thread at the operating system level, where the task is executed, but all the manipulation we do through the variable called loop, is what allows us to indicate how we are going to manage this thread, either to request it to the OS through **set_event_loop()**, executing the task and ending the process through **run_until_complete()** or finally closing the loop and with this, freeing or destroying the thread through the **close()** function.

In short, using the event loop in Python, we have access to define tasks as well as resource management through the OS thread.

Let's create another example in which we will place an infinite loop on the task and sleep the process for a second:

tests\demo\loop2_example.py

```
import asyncio

async def taskAsync(text = "Message"):
    while(True):
        print(text)
```

```

        await asyncio.sleep(1)

# loop = asyncio.new_event_loop()
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(taskAsync())
    loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    loop.close()

```

And we execute:

```
$ python .\tests\demo\loop2_example.py
```

We will see that the "Message" message is printed every second on the screen, therefore, the thread will always have a job to do, since the task, in this example if it is asynchronous, this task does not have to be infinite, as in the previous example, it can be any heavy task such as sending 1000 emails, generating PDFs. etc; taking advantage of this feature of being able to execute asynchronous code, we can define another asynchronous task:

```
tests\demo\loop2_example.py
```

```

import asyncio

async def taskAsync(text = "Message"):
    while(True):
        print(text)
        await asyncio.sleep(1)

# loop = asyncio.new_event_loop()
loop = asyncio.get_event_loop()
try:
    # loop.run_until_complete(taskAsync())
    asyncio.ensure_future(taskAsync())
    asyncio.ensure_future(taskAsync("Other Message"))
    loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    loop.close()

```

And we execute:

```
$ python .\tests\demo\loop2_example.py
```

And we will see that both processes print at the same time, in parallel, and this is because the application is running asynchronously.

The `ensure_future()` function allows you to create and run an asynchronous task and runs it in the background, without explicitly waiting for execution to finish.

With this example, we end the section where we introduce the Python loop event in an exemplary way.

Base configurations

With the event loop clarified, as you can suppose it will be necessary to create an event loop so that requests can be sent to the API created through FastAPI.

Let's start by changing the database to a sample one, which logically you have to create in your database together with the application tables:

database\database.py

```
DATABASE_URL = "mysql+mysqlconnector://root@localhost:3306/fastapi_task_testing"
```

The next step to perform is to create a configuration file called `pytest.ini` with the following code:

pytest.ini

```
[pytest]
asyncio_mode=auto
```

With the previous configuration, when executing the unit tests with `pytest`, all the tests are automatically executed in asynchronous mode, which, as we have commented, is necessary to execute HTTP requests to the API. This configuration file is processed directly by `pytest`, therefore it does not have to be imported anywhere in the application.

With the previous configuration file ready, we are going to need to create another file to be able to carry out the unit tests, this file is called `conftest.py` and it will be responsible for creating an instance of the application, which is required by the test files that are going to connect to our API:

tests\conftest.py

Let's start by defining the loop session fixture:

```
import asyncio

import httpx
import pytest
from api import app

@pytest.fixture(scope="session")
def event_loop():
    # loop = asyncio.get_running_loop()
    loop = asyncio.new_event_loop()
    yield loop
```

```
loop.close()
```

In essence, the previous function has a structure similar to the function to connect to the database that we implemented in previous chapters:

```
def get_database_session():
    try:
        db = SessionLocal()
        yield db
        #return db
    finally:
        db.close()
```

But now we use the event cycle (loop) to make the connections to the API; this is a function that we will not use directly, otherwise it is used by pytest when creating the unit tests as we will see later:

tests\conftest.py

```
import asyncio

import httpx
import pytest
from api import app

@pytest.fixture(scope="session")
def event_loop():
    # loop = asyncio.get_running_loop()
    loop = asyncio.new_event_loop()
    yield loop
    loop.close()
```

We are going to need to create another function with which we define the client fixture; this is nothing more than an instance of our application executed asynchronously by **httpx**:

tests\conftest.py

```
***
@pytest.fixture(scope="session")
async def default_client():
    async with httpx.AsyncClient(app=app, base_url="http://app") as client:
        yield client
```

This has been the equivalent of when we manually go to the browser and enter the application through:

```
@app.get('/page')
def index(request: Request, db: Session = Depends(get_database_session)):
    ***
```

As in the database, the **yields** are used to return the value (customer) and this is kept alive until the unit test finishes executing; It is exactly the same scenario when we injected the database as arguments of the API methods and that a connection was created for each user request and this request was closed automatically when sending the response to the client. Finally, this is another pytest control file that we didn't import from the project.

These are the steps that we must follow to configure a simple environment and carry out the tests that we will see in the following sections.

Testing the user module

In this section, we are going to carry out the tests of registering a user and making a user login; to do this, we will use the settings we made before.

Create a user

Let's start by defining the process for registering a user, which looks like:

tests/test_user.py

```
import httpx
import pytest

@pytest.mark.asyncio
async def test_sign_new_user(default_client: httpx.AsyncClient) -> None:
    payload = {
        "email": "admintest@admin.com",
        "name": "andres",
        "surname": "cruz",
        "website": "https://www.desarrollolibre.net/",
        "password": "12345",
    }

    headers = {
        "accept": "application/json",
        "Content-Type": "application/json"
    }

    response = await default_client.post("/register", json=payload, headers=headers)

    assert response.status_code == 201
    assert response.json() == {
        "message": "User created successfully"
    }
```

The code is quite easy to follow, all we do is create test data, for this we create a variable called `payload`, define the json type headers using a `headers` variable and finally a POST type request is made to the API using the client configured in **confest.py**; finally, the response is evaluated using the assertions that we introduced before.

From the function to register a user in our API; we are going to return a JSON with the message "User created successfully" when registering a user:

user.py

```
@user_router.post('/register', status_code=status.HTTP_201_CREATED)
def register(user: UserCreate, db:Session = Depends(database.get_database_session)): # ->
    models.User
    ***
    # return userdb
    return {
        "message": "User created succefully"
    }
```

We do this to carry out the test satisfactorily, since, from the test we do not have access to the entity of the user created with the ID and we cannot compare its value.

We execute:

```
$ pytest .\tests\test_user.py
```

And we should see a successful output in case the data of the user to be created is valid and it is not already registered in the database:

```
===== test session starts
=====
platform win32 -- Python 3.10.2, pytest-7.4.0, pluggy-1.2.0
rootdir: C:\Users\andres\OneDrive\Escritorio\proyectos\fastapi\test\vtasks\pro
configfile: pytest.ini
plugins: anyio-3.7.1, asyncio-0.21.1
asyncio: mode=auto
collected 1 item

tests\test_login.py .
[100%]

=====
```

And if you query the database, you should see the registered user.

If you get an error saying it can't find a module in api like:

```
ModuleNotFoundError: No module named 'authentication'
```

Try running pytest as a python module:

```
$ python -m pytest tests/test_user.py
```

Login

The code to create the auth token looks like:

tests/test_user.py

```
@pytest.mark.asyncio
async def test_create_token(default_client: httpx.AsyncClient) -> None:
    payload = {
        "username": "admintest@admin.com",
        "password": "12345",
    }

    headers = {
        "accept": "application/json",
        # "Content-Type": "application/json"
    }

    response = await default_client.post("/token", data=payload, headers=headers)

    assert response.status_code == 200
    assert "access_token" in response.json()
```

An important change is the start of the data option instead of the json option, since, for this method in the API, the data is obtained through a FormData and not through a JSON.

To test the above method, just run the pytest command again:

```
$ python -m pytest tests/test_user.py
```

And we would see an output like the following:

```
***
plugins: asyncio-0.21.1, anyio-3.7.1
asyncio: mode=Mode.AUTO
collected 2 items

tests/test_user.py ..
```

Logout

To close the session, a process similar to the previous examples, but with the following changes:

1. The token is set in the header.
2. The API method response is checked as a dictionary.
3. A delete request is used instead of the post request.

Finally, the complete code looks like:

tests/test_user.py

```

@pytest.mark.asyncio
async def test_logout(default_client: httpx.AsyncClient) -> None:
    headers = {
        'accept': 'application/json',
        'Token': 'WKXvHzLI0mQIpPwmuIvu_N9GY2Pb0f4qtQ09sf9Drrk',
        'Content-Type': 'application/json'
    }

    response = await default_client.delete('/logout', headers=headers)

    assert response.status_code == 200
    assert response.json()['msj'] == "ok"

```

And of course, try the previous method with the command used above.

Testing the task module

In this section, we are going to perform the unit tests for the previously implemented task CRUD.

Create a task (Body Json)

Let's start by defining the process for creating a task, which is similar to registering a user:

tests/test_task.py

```

import httpx
import pytest

@pytest.mark.asyncio
async def test_create_task(default_client: httpx.AsyncClient) -> None:
    payload = {
        'name': 'Tasks 1',
        'description': 'Description Task',
        'status': 'done',
        'user_id': '1',
        'category_id': '1',
    }

    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json'
    }

    response = await default_client.post('/tasks/', json=payload, headers=headers)

    assert response.status_code == 201
    assert response.json()['tasks']['name'] == payload['name']

```


For this test, it is assumed that the category and user with identifier of 1 exist and as a recommendation for the reader, place a category and/or user that does not exist in the database and evaluate the result.

The process to create a task using the formData looks:

tests/test_task.py

```
@pytest.mark.asyncio
async def test_create_task_form(default_client: httpx.AsyncClient) -> None:
    """
    headers = {
        'accept': 'application/json',
        # 'Content-Type': 'application/json'
    }

    response = await default_client.post('/tasks/form-create', data=payload,
headers=headers)
    """
```

Update a task

As important points in the unit test to update, we have the use of a PUT type request to a route that must be appended through the path, the ID of the task to update:

tests/test_task.py

```
@pytest.mark.asyncio
async def test_update_task(default_client: httpx.AsyncClient) -> None:
    payload = {
        'id': '1',
        'name': 'Tasks 2',
        'description': 'Description Task',
        'status': 'pending',
        'user_id': '1',
        'category_id': '1',
    }

    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json'
    }

    response = await default_client.put('/tasks/'+payload['id'], json=payload,
headers=headers)

    assert response.status_code == 200
    assert response.json()['task']['name'] == payload['name']
```

Get all tasks

For the process of getting all the tasks, it looks like:

tests/test_task.py

```
from sqlalchemy.orm import Session

from database.database import get_database_session
from database.task import crud

@pytest.mark.asyncio
async def test_all_tasks(default_client: httpx.AsyncClient, db: Session =
next(get_database_session())) -> None:

    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json'
    }

    tasks = crud.getAll(db)
    response = await default_client.get('/tasks/', headers=headers)

    assert response.status_code == 200
    assert len(tasks) == len(response.json()['tasks'])
```

As you can see in the previous code, since there is no data to supply, the `json` or `data` options are not used; for this test, it is necessary to have a connection to the database to be able to compare the expected results; not being able to inject the connection as a dependency and the `get_database_session()` function returning a generator, the `next()` function is used which returns the next element in an iterator.

Get the detail of a task

To obtain the detail of a task, it is a process similar to the previous one, make sure that the identifier that you put in the `id` variable exists in the database:

tests/test_task.py

```
@pytest.mark.asyncio
async def test_by_id_task(default_client: httpx.AsyncClient, db: Session =
next(get_database_session())) -> None:

    id = 1

    headers = {
        'accept': 'application/json',
        'Content-Type': 'application/json'
    }
```

```
task = crud.getById(id,db)
response = await default_client.get('/tasks/'+str(id), headers=headers)

assert response.status_code == 200
assert id == response.json()['id']
assert task.name == response.json()['name']
```