python

LEARN HOW TO CRAFT PRODUCTION CODE

Learn everything expected from a modern day Python programmer

Learn Python With Rune

Python Developer

This **book** covers everything you need to know to get started with Python as a professional developer.

It will teach you all the things you need to craft maintainable Python code, which is easy to monitor and keep healthy a modern programming ecosystem.

What will this book teach you?

This **book** will cover the following topics.

- PyCharm – the most popular IDE for Python
- Debugging in PyCharm
- Why Logs and best practices?
- GitHub basics
- Why Docker and how to use it?
- Microservices and RestAPI
- Grafana monitoring and what monitoring can help you with
- Prometheus and how it helps you?
- MySql use and setup

What to expect from this book?

This **book** will not teach you how to program in Python. It will keep things as simple as possible and focus on teaching you the frameworks, tools, and how Python production code is monitored, maintained, and kept healthy for business use.

There might be a few things that surprises you.

- Production code is often quite simple.
- Finding bugs in a distributed system of microservices takes practice.
- Programming is only one aspect of being a professional developer.

The good news is.

- You don’t need to be expert to any of these tools and frameworks that are used.

You only need to understand why and to use them.

This will all be covered in this **book**.

What is the price of all the tools and services used in this book?

I have good news to you – everything we use is for free. There will be no extra costs.

What this book is not

The line between DevOps/SRE and a developer is not clear cut. The purpose of this book is not to setup a production CI/CD framework or make infrastructure to run production code.

If you don't know what DevOps/SRE or CI/CD frameworks are? Don't worry, we will cover that later.

When you develop code, you need to make a lot of things locally. You need to run Docker containers locally, you need to run monitoring locally, all in order to craft maintainable code.

In old times – the responsibility of running and keeping production code healthy was done by an operations team. These days, this is a joint effort. As a developer, you need to understand and craft the code that is easy to maintain and generate metrics to monitor the health of the ecosystem.

Today the line is blurry.

But developers do not setup infrastructure to run the production code – they might deploy and use it.

That said, this book will not.

- Teach you Kubernetes setup.
- Jenkins or similar tools.

This book will only focus on what you need to know and feel comfortable about as a developer.

Prerequisite for this book

This book assumes you are familiar with Python programming on a basic level. The purpose is not to learn Python and we will not use difficult Python programming.

We will use Python frameworks like **FastAPI** (to create REST APIs), but they will be introduced, like any other used framework in this book.

All you need to know – is basic Python programming.

Who is this book for?

This book is a great companion for a Python developer, which want to land her first job.

Many of these aspects are not taught at universities – and most learn the at their first job. But reality is, any employer would prefer to hire someone who knows all this. This will shorten the time for a junior developer to add significant value.

Sitting in your first job interview and being comfortable to talk about how to use logging, make metrics, and how you monitoring your code will set you apart from others in the industry.

Having this knowledge and understanding will separate you from being a junior developer and make you comparable to senior developers.

Are you ready to start?

Sit down at your computer – be connected to the internet.

This book will be a game changer for you. You will get deep insights into the world of Python programmers – what it takes to become one of the most highly paid talents in the programming industry today.

Are you ready to take your skills to the next level and get a high pay check?

Table of Content

Preface

Did you know Python started in the late 1980s as a hoppy project by Guido van Rossum at Centrum Wiskunde & Informatica (CWI)?

And did you know that I actually worked at CWI as part of my PhD journey? Yes, I lived in a tall pink building in Amsterdam. To get to CWI I used one of those awesome Dutch bikes riding along the roads like a king.

As a king?

My first ride would change the way I bike forever. I came to a cross section and I stopped to wait for a car to cross – as we usually do in Denmark to avoid getting hit by it. But something happened that I did not expect.

The car waited.

I got unsure of what to do – should I cross the road at the risk the car was just waiting to hit the gas pedal to end my life?

Eventually, after long waiting and great patience from the driver, I crossed the street, and to my surprise, I survived. The driver only cared about my safety. This is how cars look out for people biking in the city in Amsterdam.

How is this related to Python? Except CWI?

It does not – but the culture of caring and giving back to others is deeply rooted in the culture.

With this book I want to give you a great experience, an understanding of why things are done in certain ways, and how to apply them in modern day ecosystems when developing Python code.

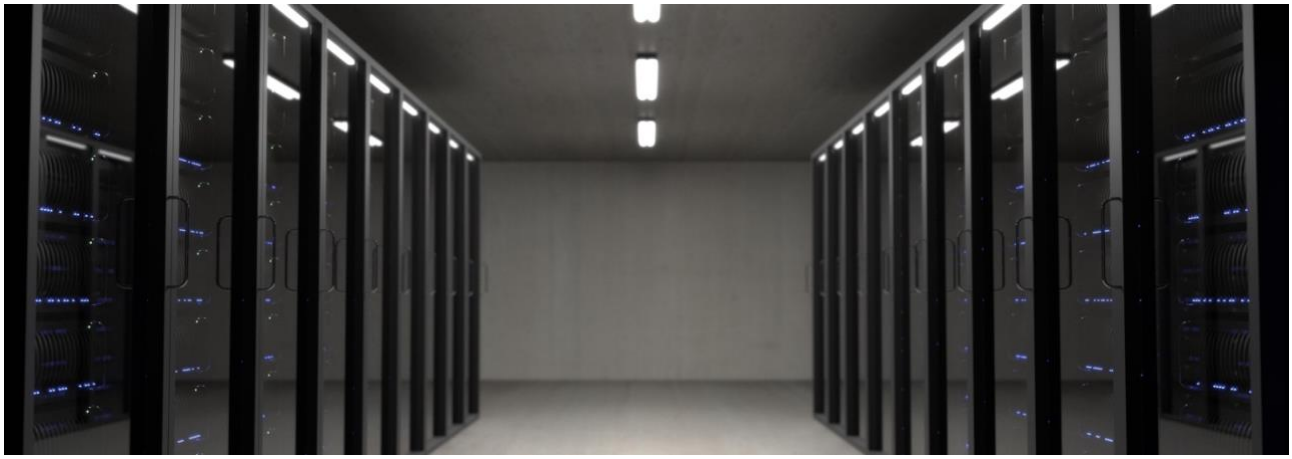
I wish nothing more than getting you safe from a junior developer to become senior developer.

Why Python?

Python is by far one of the most popular programming languages out there for many reasons. I was skeptical at first – like many are out there.

Let me tell a bit about my first encounter with Python.

I started as a C programmer working with optimizing and parallelizing code to run on HPC's – that is High Performance Computers – which is just a fancy word for a big cluster with a lot of CPU's. You probably seen pictures like this.



Yes – I programmed computers like that – and C does a great job at that.

While C is a small and extremely powerful programming language, it also has a price. Some things are just not easy to do.

One day I needed to make some analysis of a bunch of text files – and I started to write some C code to do that. Then a self-taught programmer introduced me to how Python 2.7 could do what I was doing in a few lines.

I was speechless.

He could actually do what had spent more than a day doing in less than 30 minutes. Giving the exact same result.

I was like most when they see Python.

It is dynamically-typed, it is interpreted and will always be slow, it can't be used for production code.

Let's just stop there and fast forward with a time machine to today – either you can use a McLaren from Back to the Future or just check that your clock is showing the current time.

Python has showed to the right programming language for many fields, including, making microservices, web apps, machine learning, financial analysis, automating stuff, and the list could go on.

The point is, there is no doubt it is the right tool for many things.

But what about dynamically-typed – why do many care so much about it?

Python is naïve, it will just assume that you know how to use the correct types, and it will run with that assumption. If you did not – the program will fail at run-time.

Hence, when you run production code, this is a big problem. You will first catch errors like wrongly typed variables at run-time. In a static typed language, the compiler will catch that at compile time. This is a big problem, as you do not want to deploy code to production that will fail because of these kinds of mistakes.

The good news is – it is not a problem. Python has a static typed checker – and the best part is: You will learn how to do that in this book.

What about speed?

Python is an interpreted language – so it will always be slow.

Actually, that is wrong too.

I worked with optimizing code – remember, the HPC's and the difficult C code?

Yes, and the first thing you learn when you optimize code to minimize clock cycles is, that you need to know what code to optimize.

In general, the saying is, that 80% of the time of a program is only 20% of the code. That is, that only 20% of the code is accountable for the run-time.

Another thing is – when you optimize code, it becomes difficult to understand. Therefore, you only want to optimize code which is needed to run fast. As you also will learn in this book – readability is more important than ugly fast code. Only try to optimize code if needed.

How does this help Python?

Well, Python comes with a lot of libraries and can make even backend data processing fast now. This is the amazing power of Python.

The libs can actually do the hard work for you – and you only need to make beautiful Python code to structure it all.

Isn't this exiting?

As a Python developer, you do all the nice and pretty stuff and let someone else to all the nasty ugly stuff.

I just love Python like many others do.

Why listen to me?

I always had low self-esteem. It has always taken me time to feel comfortable in new environments.

My first job was no different.

I was also afraid that they would think I was not smart enough as I came directly from university with my PhD degree.

This was a horrible combination.

When you start your first job there are many new things to learn. Most educational system do only focus on teaching how to program, solve programming challenges, theory, data structures, algorithms, and the list could go on – but there will be something missing.

How to run code in production in a way that is easy to maintain and monitor.

Simple concepts as logging was not taught in my long stay at the university.

Why do we use logging? How do you make good logging? How does it help you find bugs in running production code?

I didn't know any of that – and I didn't dare to ask.

Since then I have spent 15 years programming and helped many junior developers get started with it. The story is always the same – they need to learn a lot of new tools and concepts to make production code. Nothing is taught about it at the big educational places.

I have realized, I was not alone feeling like I did when I started. But more importantly, I love helping others and help them get this understanding and skills covered in this book.

For you, I want to do the same.

You might be a junior developer starting your first job. Or you want to prepare yourself for your first job interview – as they will ask how comfortable you are about this.

There might be many reasons for you to want this. Maybe, you just love to learn new things.

Well, this is a great book and it has been a lot of fun crating it and I am sure you will learn something from it.

"You don't have to be great to start, but you need to start to become great." – Zig Ziglar

Acknowledgements

If you want – your name can be here.

Thank you for taking the time to make this book better with me.

00 – How to Get Started

In this chapter we will get acquainted with the basics for a professional Python developer.

- Python versions and why we care – a small story from real life.
- Install Python.
- What is Git and how to install it.
- Why use PyCharm and how to install it.
- What are Virtual Environments?

If you have installed Python 3.10+, Git, PyCharm, and feel comfortable about virtual environments in Python – then feel free to skip to next chapter – unless you want insider stories?

Python 2.7 and Python 3 – Why don't they just rewrite it?

A lot of Python production code was written many years ago – back when Python 2.7 was the primary release.

This code still runs in production and it works fine.

You might wonder, why don't they re-write it in Python 3?

Well, there are a few factors that influences this.

Can you guess what they are?

Take a moment and think about, why would they keep old Python 2.7 code running in their production ecosystem?

Take 2 minutes to think about it and write down your top 3 guesses why big tech companies keep old Python 2.7 code in their production setup?

When you have done that – continue.

Here are the top 3 reasons.

- The cost of re-writing it in Python 3 can be large – modules are living in complex ecosystems, which can make it difficult to ensure you did it correctly.
- If you re-write something that works "good enough", then there is a risk you break it.
- With modern containerized environment, it is not a problem to have modules using old tech stacks. We will learn more about this in the book.

You see, what keeps old code running is:

- Price to change from Python 2.7 to 3 is high.
- The risk that something doesn't work correctly when migrating from Python 2.7 to 3 is too high.
- And finally, it is not a problem to keep it Python 2.7 running in production.

Why do I tell you this?

Well, let's explore another problem.

A new release of Python 3 – What now?

Python updates are not always good thing.

You are probably used to when a new update comes you want to use it immediately. All the new fancy features, nice improvements, optimized workflows, and the list could go on.

Yes, you wake up in the morning and see the new release notes. You just started your first job as a junior Python developer and you are really excited to be the first one to tell the breaking news at work.

Getting there all sweaty you are the first one.

You jump to your station and upgrade to the new version of Python and you hit deployment flow.

Ups!

No, here **Ups!** is not an abbreviation for something cool.

It's just a simple **Ups**!

The kind where you want to hide and never show up again. That kind of **Ups**!

Everything broke. Nothing is working!

Why did that happen?

Well, you have no idea.

Later your programming mentor comes not looking happy.

Luckily, he grabs a cup of coffee and in a few seconds, he seems calm and happy again. He got an alert on his phone telling him the entire service broke and was not available for customers.

Actually, it is a good thing, because he was telling management that we needed a better release pipeline ensuring not to release code without validation. Now he thinks he has a strong case to convince them.

But what happened?

He explains to you.

"The code we run depends on many libraries – these libraries are made for Python 3.9 and have still not been updated for newer versions. They simply do not work for newer versions of Python."

It's kind of funny – most libraries take time to be updated to newer versions. Some take really long, even though they are made by big corporations (no names here, but, yes, the really big ones).

What is the point?

I have worked in many groups delivering many products.

What you realize over time is, there is nothing perfect out there. Everyone does it a bit different. There is never time to do it right.

Let's emphasize that.

When you are promised to make Proof-of-Concept (PoC), Proof-of-Work (PoW), or whatever they call it. You think that you will get time to implement it correctly afterwards when you have the experience.

But now – this PoC / PoW will become production code.

It works! Why bother using your expensive time to make it as it is supposed to be.

This holds for many things – infrastructure, deployment pipelines, DevOps stuff, SRE things, you name it.

This holds for small startups, for billion dollars SaaS companies, everywhere you look.

Things are not perfect – they are just good enough.

Installing Python

Are you scared now?

What version of Python to install for our project here?

Or… *"Oh, no, I use a different version of Python, what to do?"*

Don't worry – we will cover all that.

The Python version we will use is Python 3.10 – if you feel like using another version, no, problem. But we will use Docker images with Python 3.10 later, hence, you can either change them or it will be good to have Python 3.10.

If you didn't understand what Docker images are – don't worry – we will cover that later.

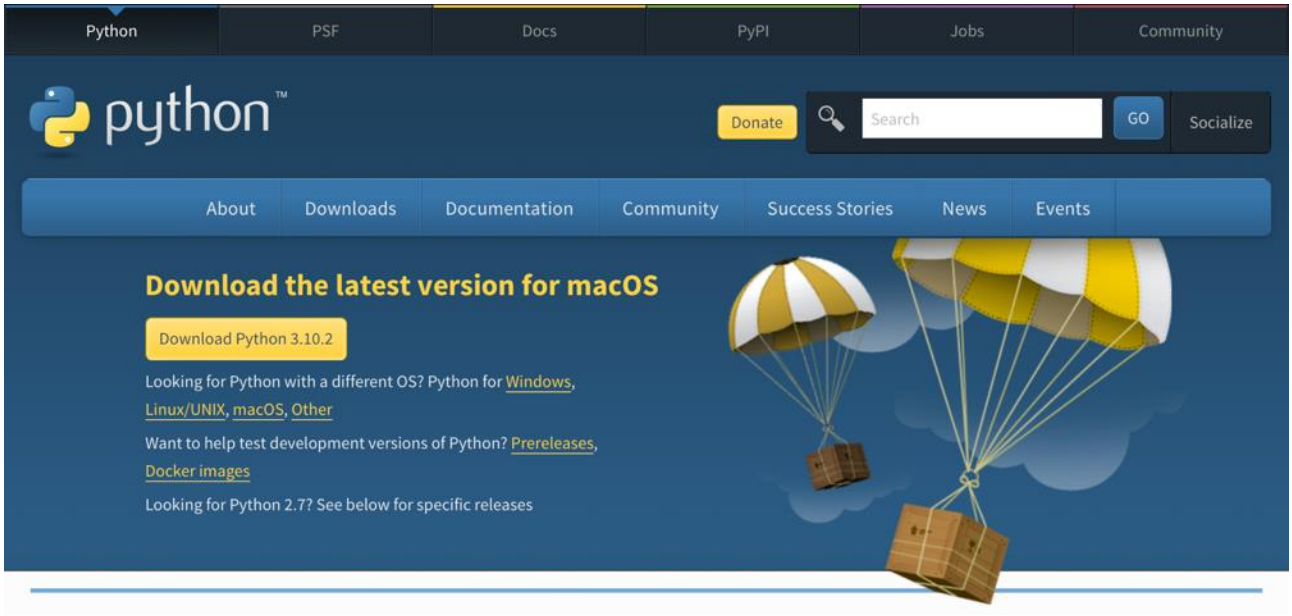
What do you do if:

- You have installed Python version 3.8 (or any other version)?
- Or you have version 3.10.2?

Let's consider if you have an older version (or newer) than 3.10 first.

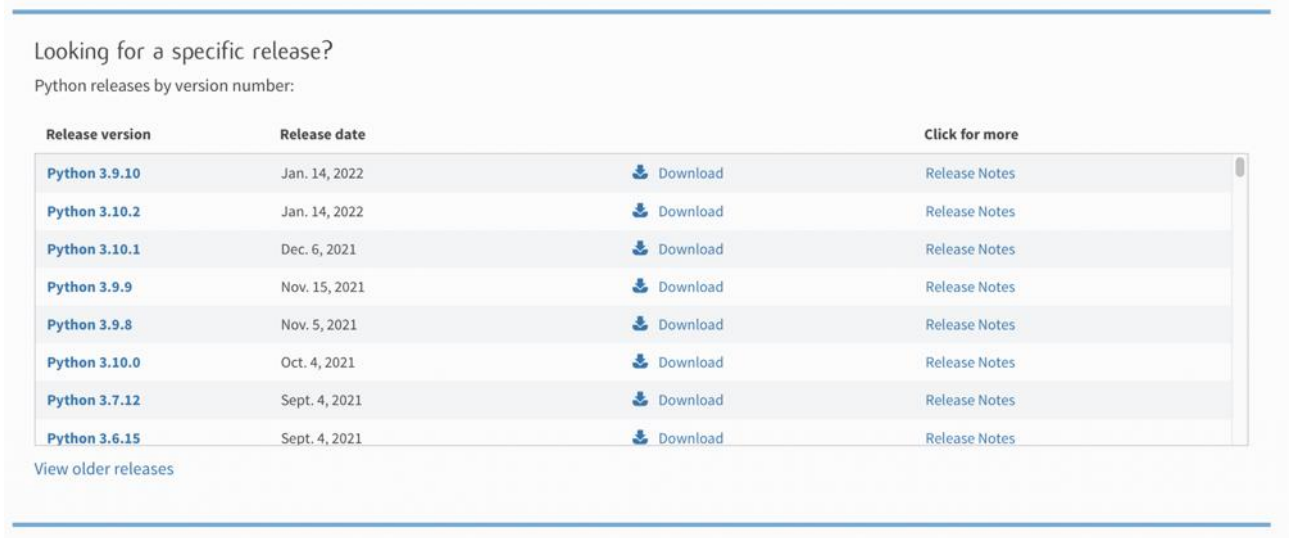
Nothing to worry about, **you have multiple versions of Python installed on your system simultaneously**.

All you need to do is to go to **python.org**.



If latest version is 3.10.* (change * with any number) – then use than one and follow installation instructions. Then you are done.

Otherwise you can find a 3.10.* release further down on the page.



Above you can see an example, where I would choose Python 3.10.2.

Then you are good to go.

- Download it and install it.

What is Git and GitHub and how to install Git

When a full team works on a code project (like **Python** project), you need a way to all work on it simultaneously and not being blocked by others.

How did **Word** solve this problem (Yes, you the tool where you can write text in, Microsoft Word)? Well, they let all edit the same document simultaneously. You can see how others edit the same document, what they do, what the delete, what they type, and so forth.

Can't we just do the same for our project code?

Well, it will make things complicated.

If you are about to fix a bug, another one is releasing a new feature, a third one is adding unit testing. And now you want to run the code to see if you fixed the bug.

What happens with all the other changes from your team?

Well, we hope for the best.

You see, this is not easy to manage.

This is why you will work on your own copy (or branch as it is called). You make your changes, commit them, then make a pull request (PR, as it is called), someone will review if your changes can be merged into the main branch.

This is where Git and GitHub comes into the picture. Git and GitHub, helps you with all that.

Don't worry about the details right now.

What you need to understand is.

- When more people work on the same piece of code – you cannot do it on the same copy.
- When you need to make some changes, you get a copy of the code that only you work on – your branch.
- Then when you are done – you make a request to get it merged into the main code. This is done by a pull request (PR).

Companies and organizations have different ways of the specifics. Don't worry too much about that – as it will be explained to you when you start.

This introduces us to Git and GitHub, but what is that?

What is Git and GitHub?

Git is a version control system that enables more to work on the same project with each their copy, enabling developers to create good habits to review code while merging it together.

While it sounds simple, it can be complex process when many works on the same code. There are many ways to use Git, but for the most part you just need to understand some simple things about it. The rest is project specific and there might be many different approaches.

GitHub is a service that stores the code. It is what is called a repository hosting service. Basically, that is where you all have the main code. This includes branches, the history of the code, you could say, the source of truth of the code.

Git is the tool you use to manage it all. GitHub is where you store the things you share in the project team.

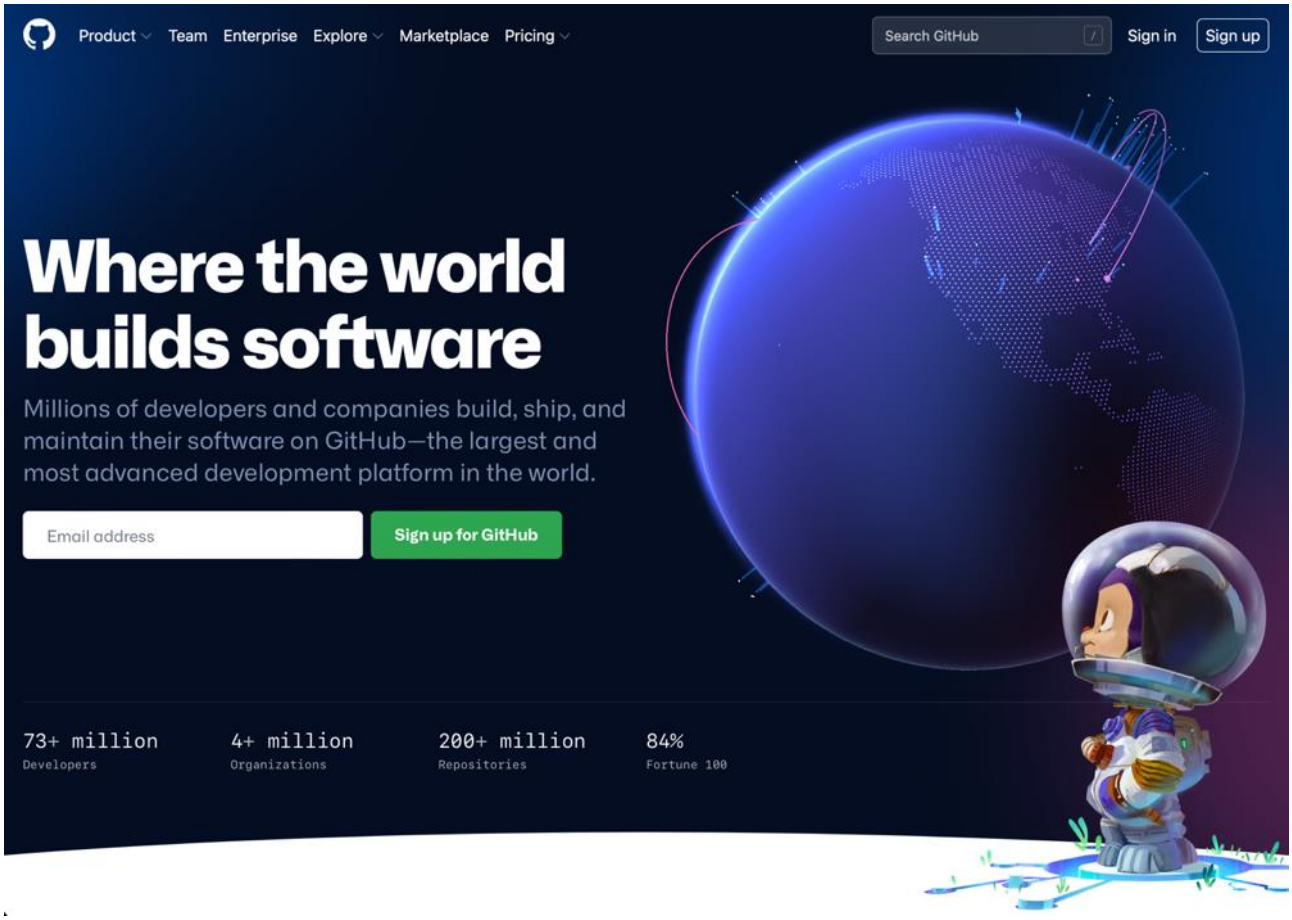
Installation of Git and get login to GitHub

To be a developer you need Git and be registered on GitHub.

If you are on Mac or Linux it is already installed. **You can see this guide to ensure it is there**.

On Windows you can install it from **here (also check this guide)**.

To get started with GitHub go **here and sign up**.



Why PyCharm and how to install PyCharm

"I love VS code – and I will not switch to PyCharm!"

Well, good for you.

I will not force you to use PyCharm and if you program in other languages, it might be beneficial to use VS code.

Most Python developers prefer PyCharm because it is easy to use and it is optimized for Python and all common use cases.

But wait a moment!

What is PyCharm?

"PyCharm is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools for Python developers, tightly integrated to create a convenient environment for productive Python […] development." (**jetbrains**)

Wow! What does that mean?

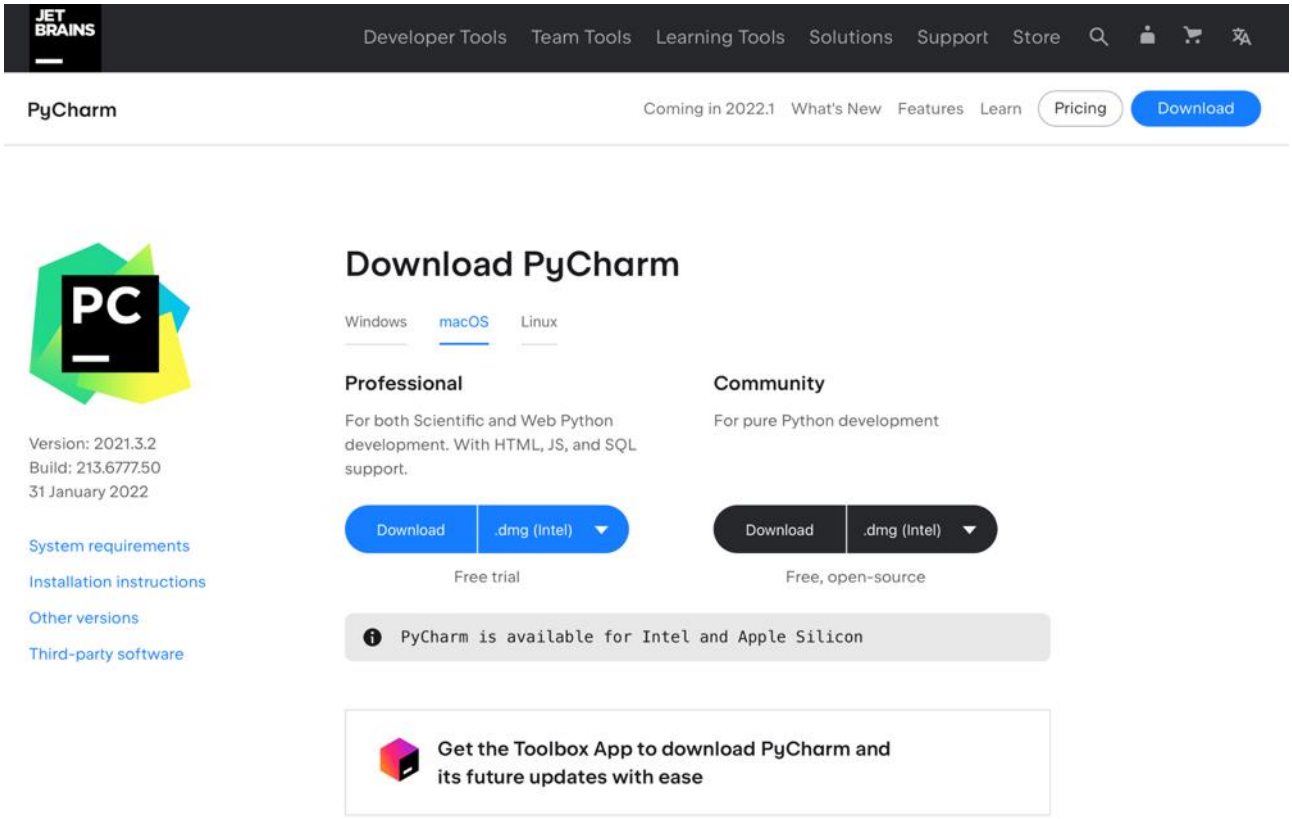
When you develop code (not just Python code), you need to manage your projects, debug code, navigate in your code base, setting up environment, commit code, and so forth.

If you don't know what it all means, don't worry, we will get back to that.

PyCharm makes all those "project" things easy for you to do.

How to install PyCharm

Get the free community version **here**.



Virtual Environments?

Remember in the beginning of this chapter? How we realized that each module might use a different version of Python? Also, how libraries might be dependent on specific versions?

How do you deal with that when you develop on your computer?

You see – one moment you work on a project using Python 3.10 with some new libraries. Then there is a bug and you need to make a hotfix on some old module using Python 3.8 and a different set of libraries.

This is where virtual environments come into the picture and saves you.

"A virtual environment is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments" (**python docs**)

How does PyCharm help with that?

Well, for each project you have, you will have a virtual environment with a specific Python version, and its own libraries and scripts.

This means, when you open the project, where you need to make a hotfix, then PyCharm ensures you use this environment with Python 3.8 and the libraries.

Then you can make the fix and deploy it in a Docker container (we will learn how to do that) with the correct Python version, libraries and scripts.

After that, you switch back to your main project and PyCharm ensures you use the correct environment with Python 3.10.

That said, PyCharm takes care of all the virtual environment management for you.

You need to understand what it is, but not how to handle it – PyCharm does that for you.

…are you ready to start? I am, so let's proceed.

Summary

In this chapter we learned the following.

- Why there are many different versions of Python running.
  - It is easy to maintain different versions running in each module or service.
  - It is expensive to update and some dependencies might not support it yet.
  - There is a risk to make new versions.
- The world of coding is not perfect.
  - Companies often put Proof-of-Concepts to production.
  - You are expensive and needed to do something more important – like developing more things – not rewriting something which works.
- How to install Python
  - You can have multiple versions of Python installed in the same system.
  - This is often needed, as modules and services use different versions.
- What is Git and GitHub
  - Git enables more developers to work on the same projects.
  - GitHub is where you save things you share in the project – including the history.
  - You need to have Git installed on your system.
  - Also, you need a login to GitHub.
- What PyCharm is and helps you with
  - PyCharm is an IDE – basically, it helps you navigate and maintain projects.
  - It makes developing, debugging, git cloning, environment, etc. easy for you.
- What is a Virtual Environment?
  - It makes it possible to have different Python versions, libraries and scripts for each project.
  - PyCharm takes care of all that for you.

01 – Starting our First REST API

In this chapter you will run your first API and make requests.

- What is a REST API
- Clone the project we use from GitHub
- Understand the structures
- What is requirements.txt?
- Use pip install
- Start the API and see it
- Call the API from web interface (Swagger UI)
- Use a Python script to all the API

What is a REST API?

You have probably heard about **REST API**'s (**RE**presentational **S**tate **T**ransfer).

If you google it, you will probably hear about some principles it needs to fulfill. And yes, there is the formal definition, but for me it seems difficult to translate the formal definition into something you understand what a **REST API** is.

Let's think about it differently.

First of all – what is an **API** (Application Programming Interface)?

- It enables software or modules in software to talk to each other.
- It doesn't have to be the same programming language.

Actually, API's are an awesome invention. They can also be thought of a contract between software modules. Let's say module A and B communicate (or talk) with each other through a specified API. Then you can change a module, say module B, if it still follows the API. This makes the software easier to maintain.

A **REST API** has some additional restrictions.

When most talk about **REST API**'s they mean a web **API** where they can send **HTTP** verb and a URL which describes the location of the resource.

That means a few things.

- A **REST API** is a client-server architecture. Like a browser (client) and webserver.
- A **REST API** has a URI and it is like a webserver with different pages or resources.

Funny note: Most juniors developers think they need to understand all these concepts (like REST API) in detail. In reality, most use these concepts in a vague manner and most seniors would not be able to tell all the design principles behind them.

What is the most important part about a REST API?

- **Stateless**. That means the server does not know what you just have done – you need to kind of explain everything in every call you do. This makes the server logic easy – it does

not need to check any history or state from the caller (or client), it knows exactly what to do from the path and parameters.

- **HTTP verbs**. A REST API uses HTTP request methods. Most common are.
  - **GET**. Retrieves resources.
  - **POST**. Submits new data to the server.
  - **PUT**. Updates existing data.
  - **DELETE**. Removes data.

Some common practices in REST API's are.

- **JSON**. Most REST API use **json** to transfer the request and answers.
- **Paths names are nouns**. It is common that the paths defining the endpoints are nouns.

On this journey we are on, we will create simple resources and expose them as REST API's.

We will only have endpoints (paths) that we use – this ensures we have a simple interface and focus on what matters to learn what we intend.

Let's get started with our first simple **REST API**.

Launch PyCharm and clone project

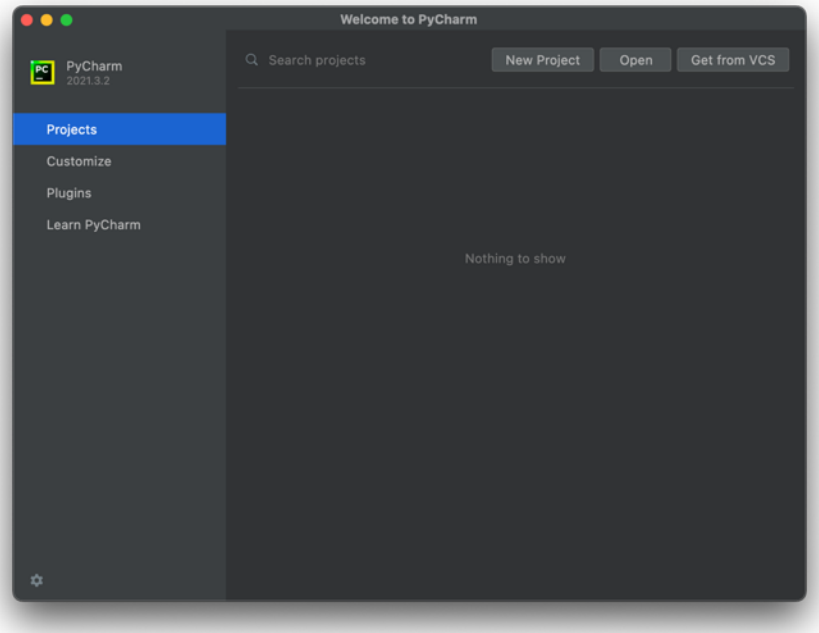
You have installed Python 3.10, Git, and PyCharm. Now you are ready to get started on this journey, where we will learn a lot of things – but for now, it will be simple and we will build upon it.

Here we will run an API from your developing machine. This is how you would develop it and possibly debug it the API from your machine.

Later we will learn how to have it running in a Docker container, how to add metrics with Prometheus, using Grafana for monitoring, and much more.

It all starts by launching PyCharm.

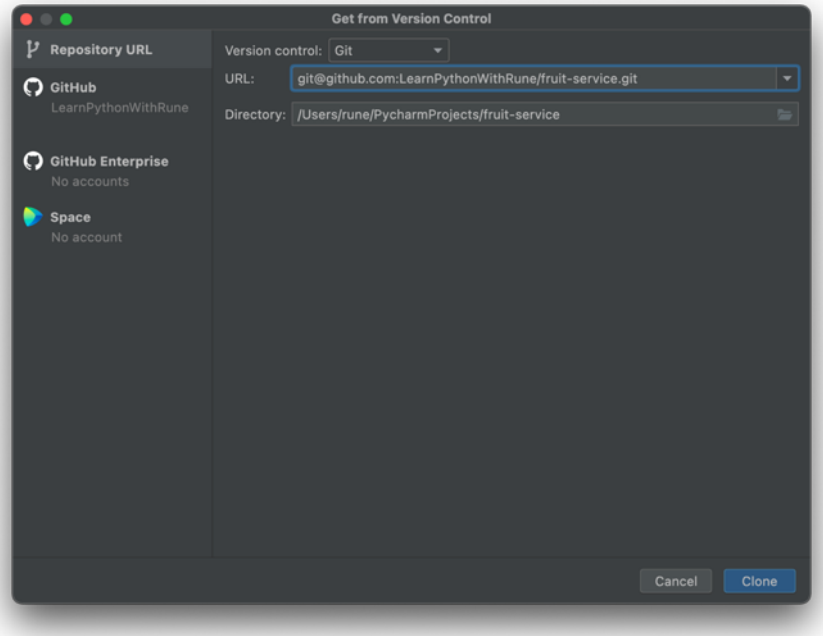
Do that and get to this view.



Click “Get from VCS”.

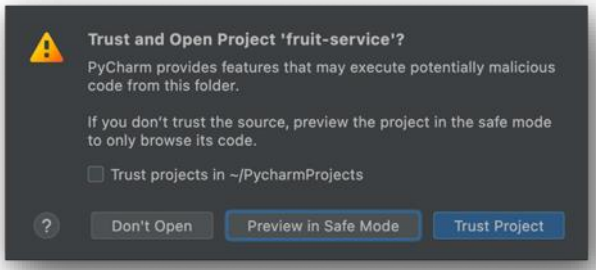
And past in the following:

git@github.com:LearnPythonWithRune/fruit-service.git



Then press Clone (lower right corner).

You might be prompted to trust the project.

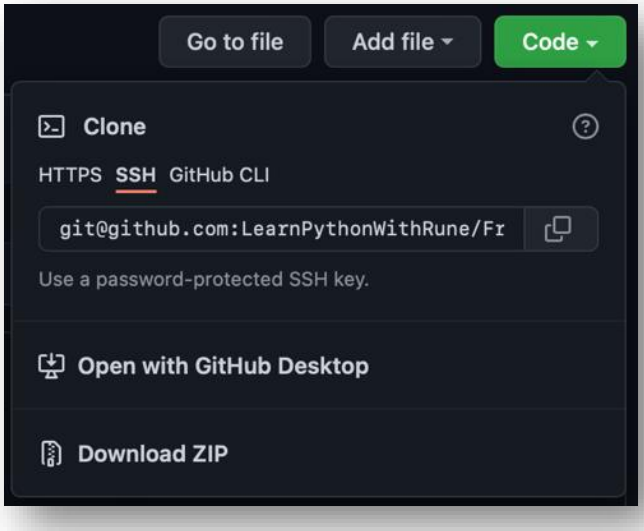


If you want to follow along in the project you need to press Trust Project.

If you feel uncomfortable about to click Trust Project you can find the code in the repository here at my GitHub: **https://github.com/LearnPythonWithRune/fruit-service**

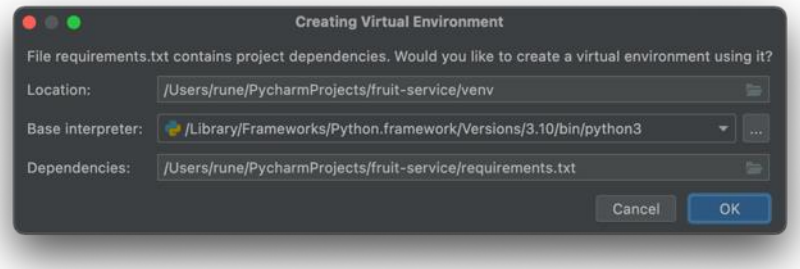
Then you can browse around and finally feel comfortable about the code before you trust it.

The string we parsed in PyCharm to clone the project also comes from GitHub. You find it from the green button Code. Push the dropdown and copy it from there.

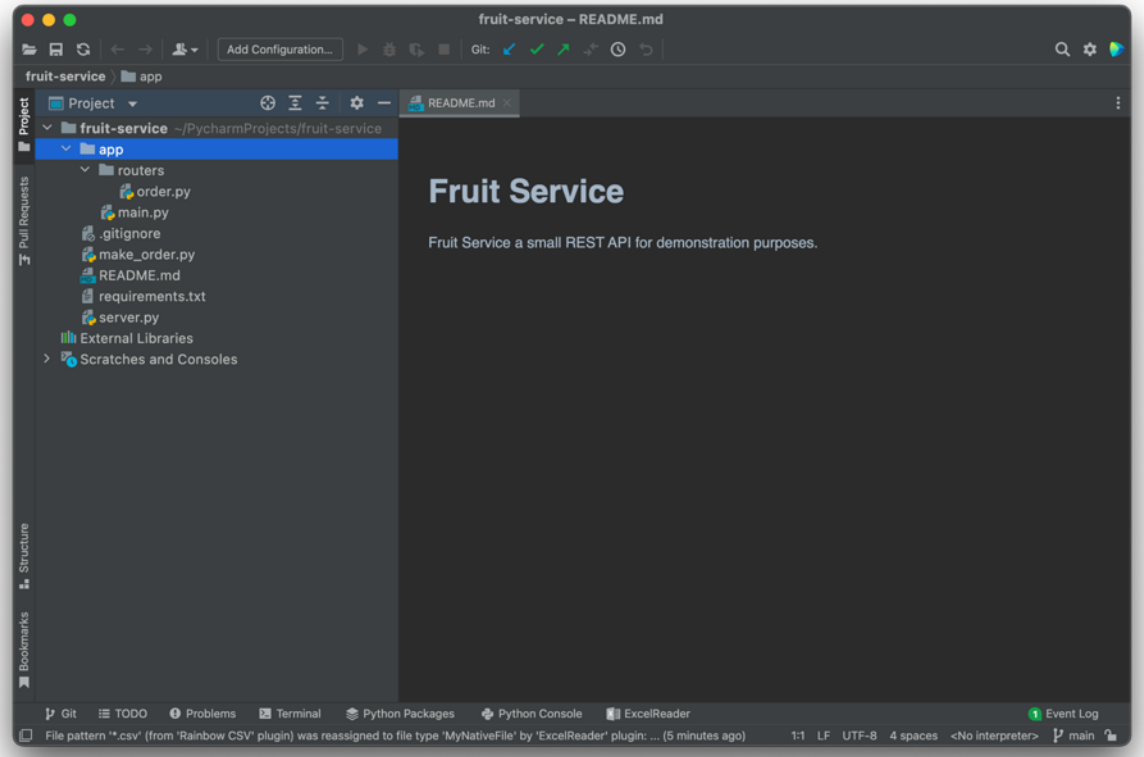


After cloning the project, you should be represented with something similar to this.

You might be asked about creating virtual environment. Click cancel for now.



Then you end up here.



Before we explore the files and the structure of the project, we will setup the virtual environment.

Setup virtual environment

I know I told that PyCharm would take care of all the virtual environment things, but there is one thing you need to tell it – what Python version to use.

In the last part we cloned the project – now we need to select the Python version we want to use. You might have multiple versions of Python and it can be a bit difficult to know which one to use.

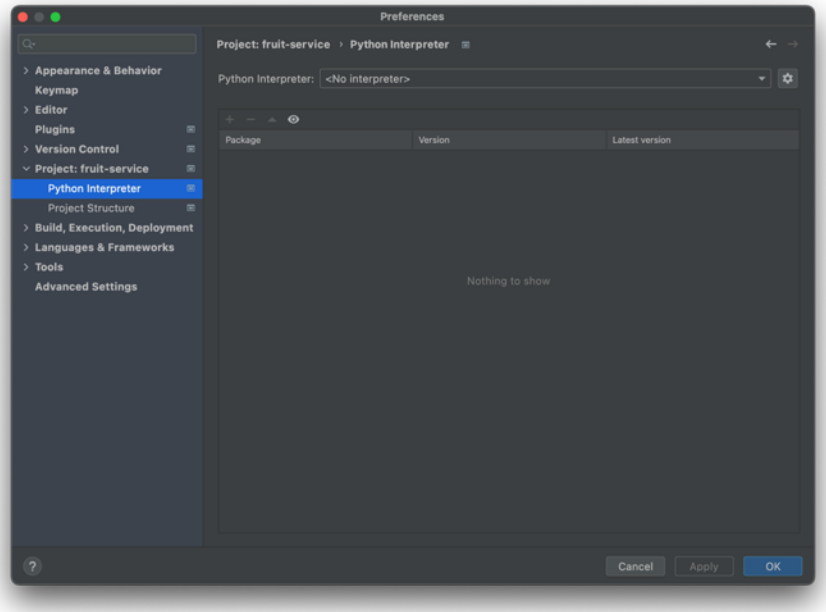
But let's try it together.

There is a difference between Windows and Linux and macOS (**ref**).

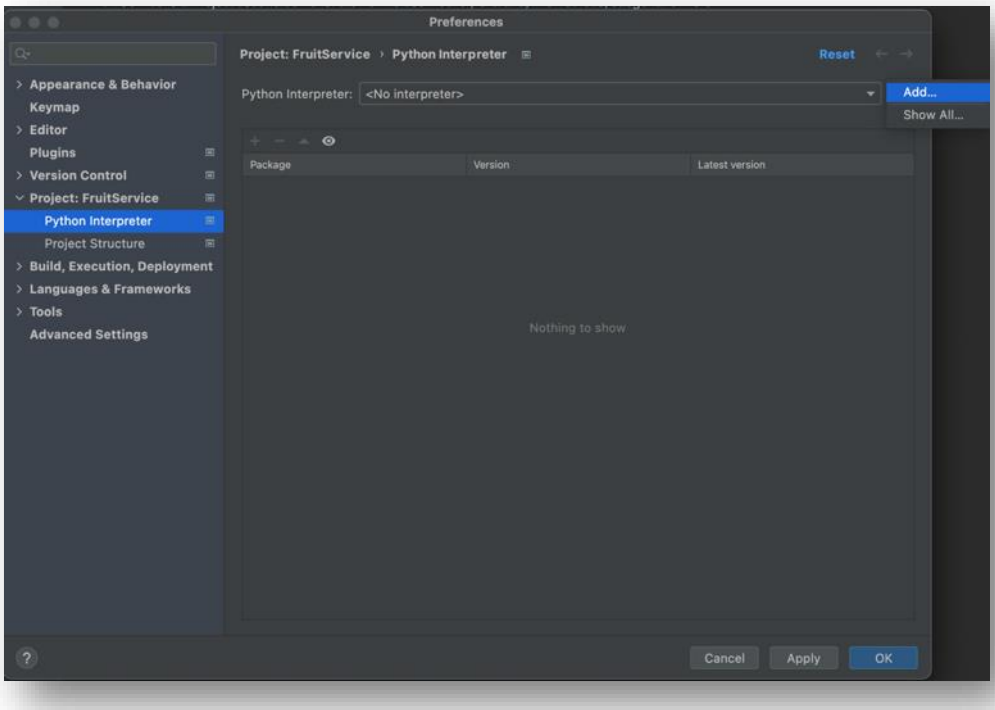
On Windows and Linux use the menu: File | Settings | Project | Python Interpreter

While on macOS use: PyCharm | Preferences | Project | Python Interpreter

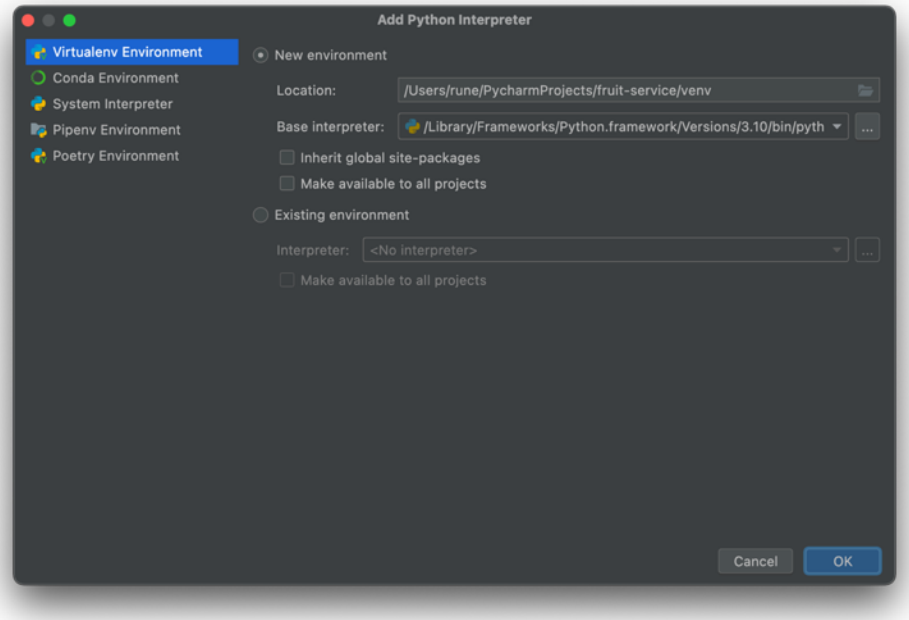
Then you should have the following window.



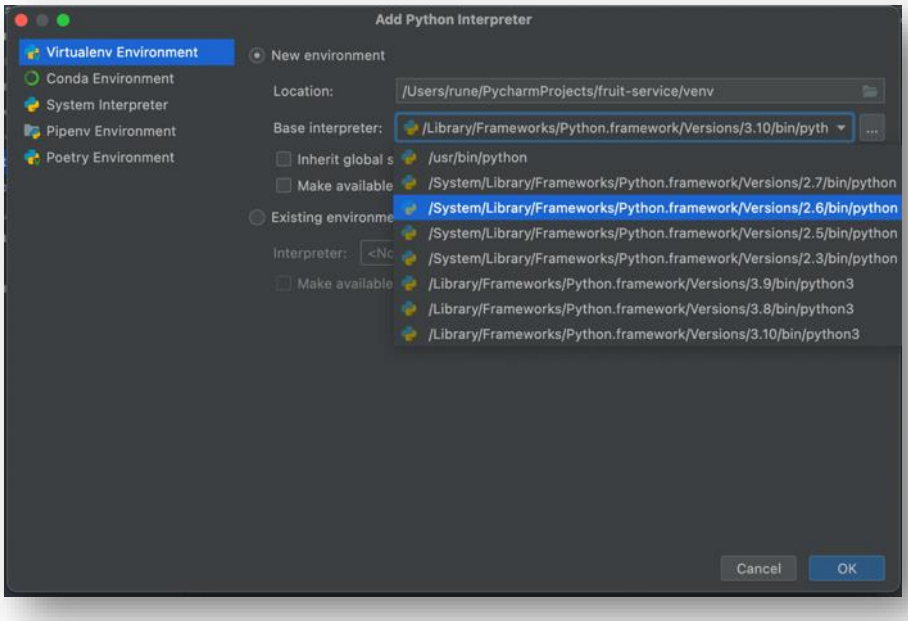
To add a Python interpreter, you need to click the small settings icon and press add.



And you should end up with something similar to this.

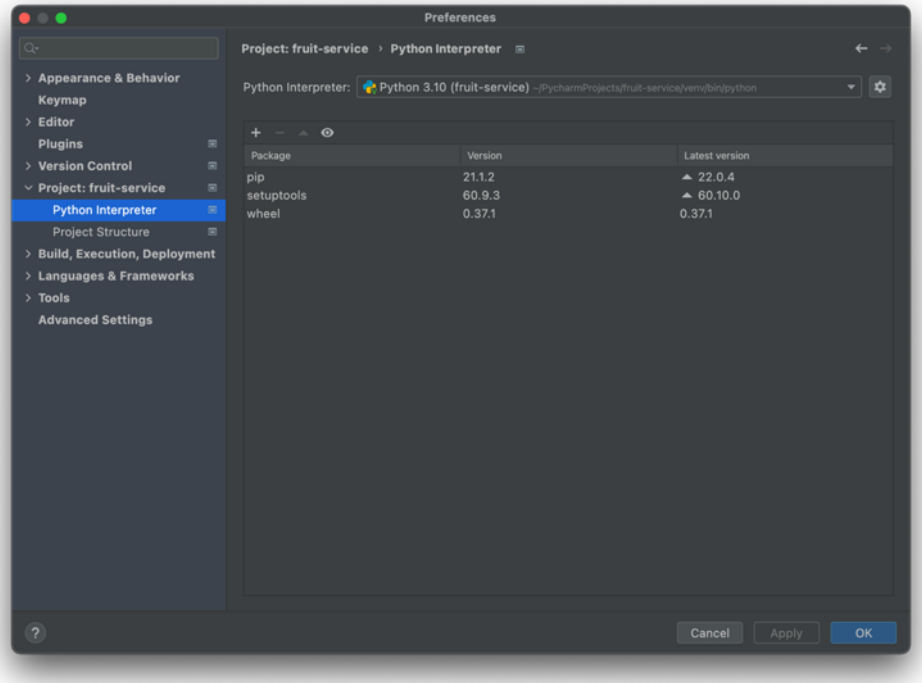


There you see the Base interpreter dropdown. Push it.

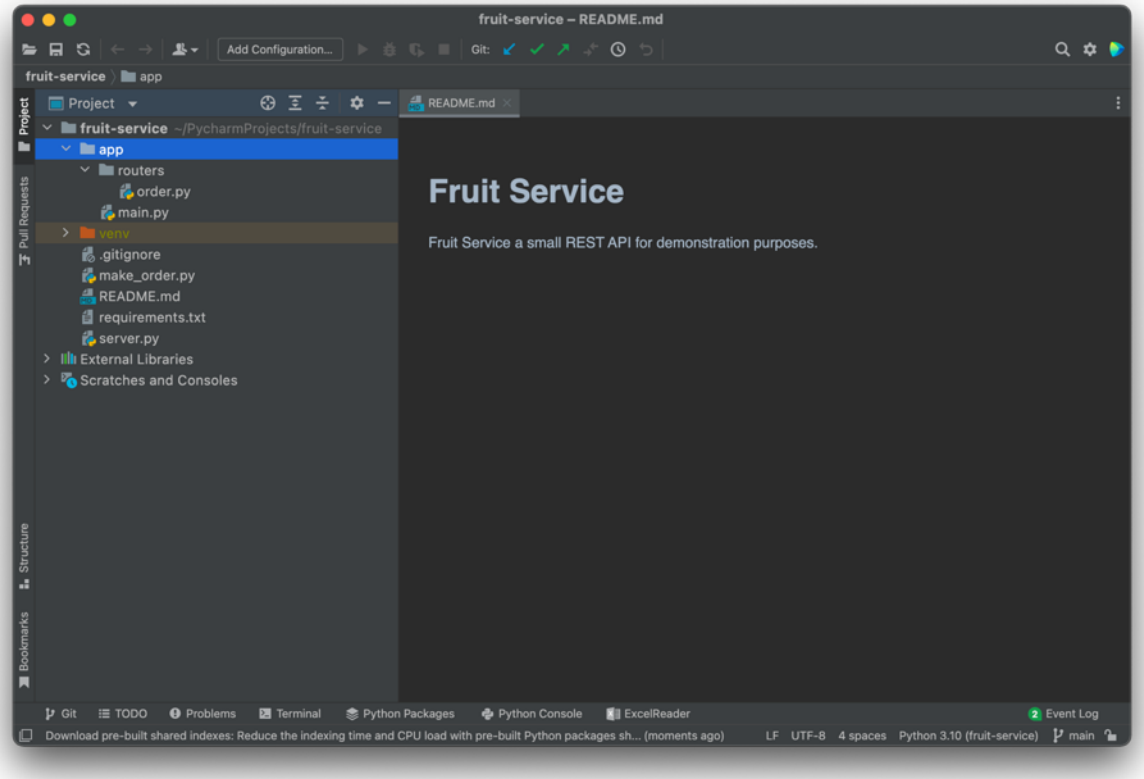


At my view you find Python 3.10 at the bottom.

Choose it and click OK. It will install the environment and you are ready.



Notice, that there will be added a **venv** folder in your project. That contains all the virtual environment files for your project.



What to do if you don't see the version when installing environment?

Well, you need to try to find the correct one – you installed it in Chapter 00.

On a macOS you can start a terminal and try to run a specific Python and check the version as follows.

```
(base) Rune@Runes-MacBook-Pro ~ % /usr/bin/python --version
Python 2.7.18
(base) Rune@Runes-MacBook-Pro ~ % /usr/local/bin/python3 --version
Python 3.10.2
(base) Rune@Runes-MacBook-Pro ~ %
```

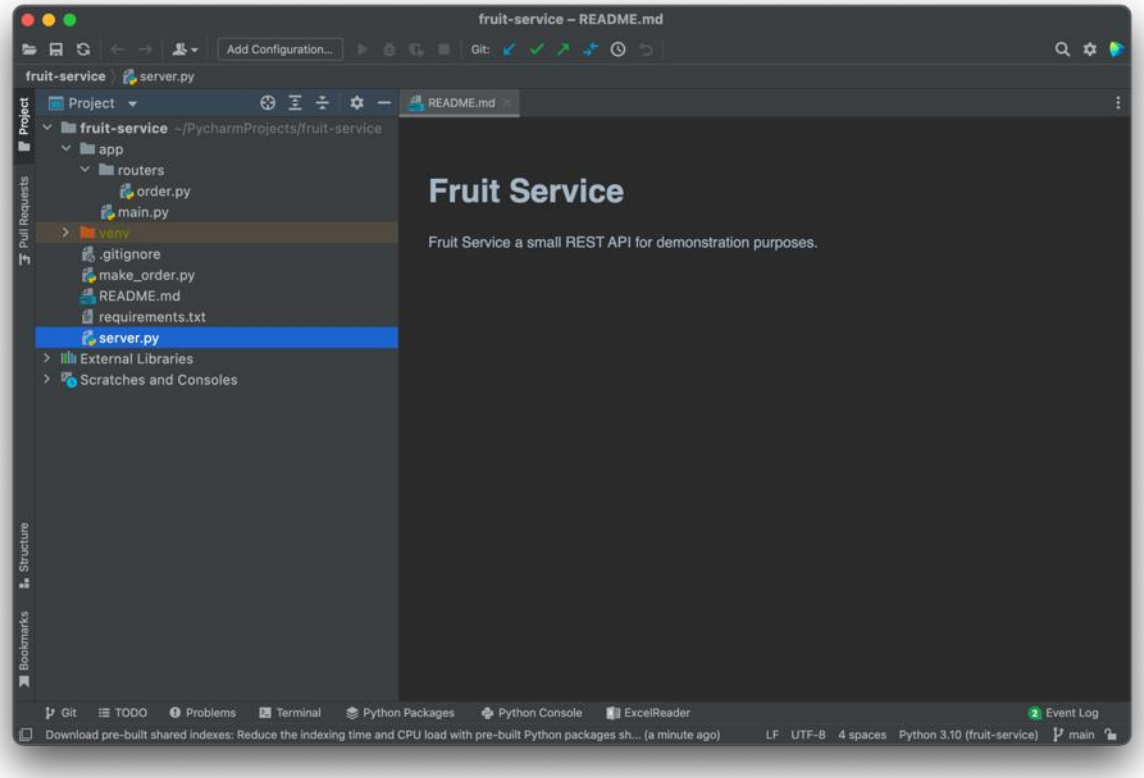
It will output the version.

On Windows you will do it similar – just in a powershell or cmd.

Go to the paths and run the specific python program with "--version" argument, then find the one with version 3.10.* (where * is any number).

Explore the project

You should be presented with something like the following.



We see the project contains some files.

- **README.md**
- **requirements.txt**
- **.gitignore**

- **server.py**
- **make_order.py**
- **app/main.py**
- **app/routers/order.py**

We will shortly explore them.

Just to note, the **venv** folder contains the virtual environment and you should ignore the content in it.

README.md

The **README.md** is an essential guide that gives other developers a description of your GitHub project. It is written in Markdown, which is a lightweight markup language for creating formatted text using a plain-text editor.

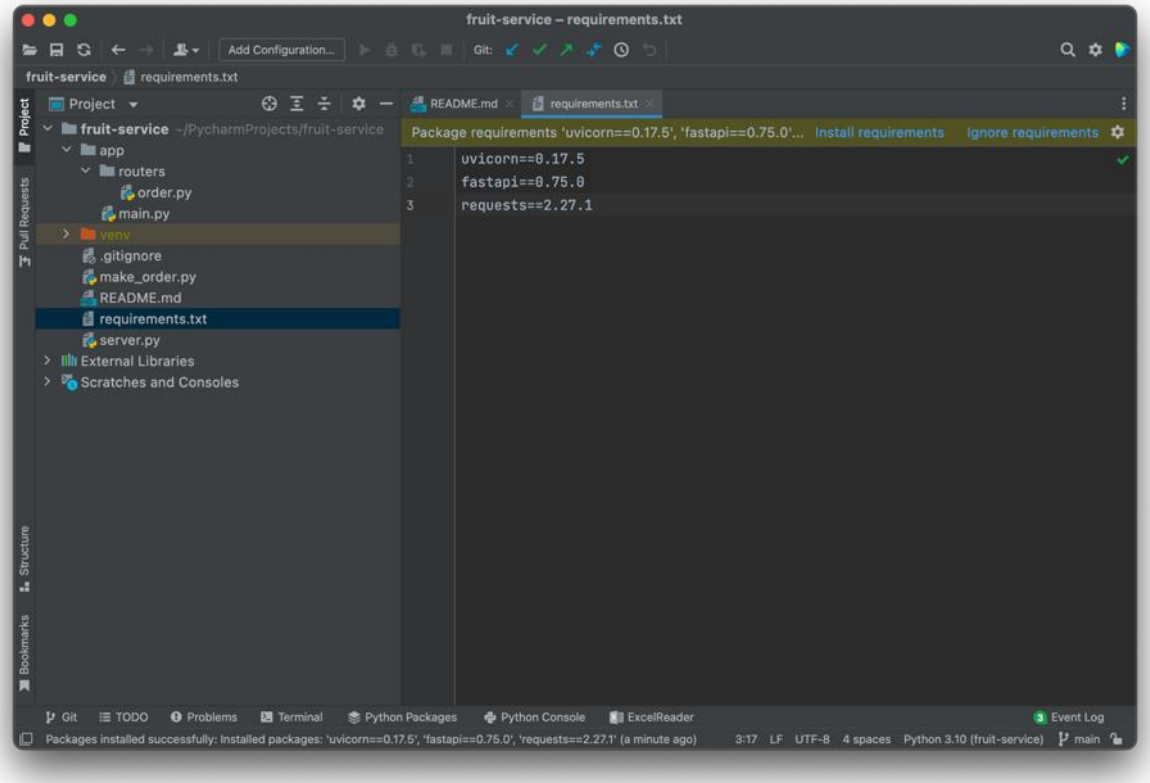
We will not explore this file further. Simply, think of it as a description for others to understand the project.

The detail level can vary a lot, as you see here.

requirements.txt

This file is essential – it contains a list of the libraries that are needed by the project.

Let's start by opening it inside the PyCharm. You do that by double clicking on it.



You see 3 lines in it with the libraries needed in the project.

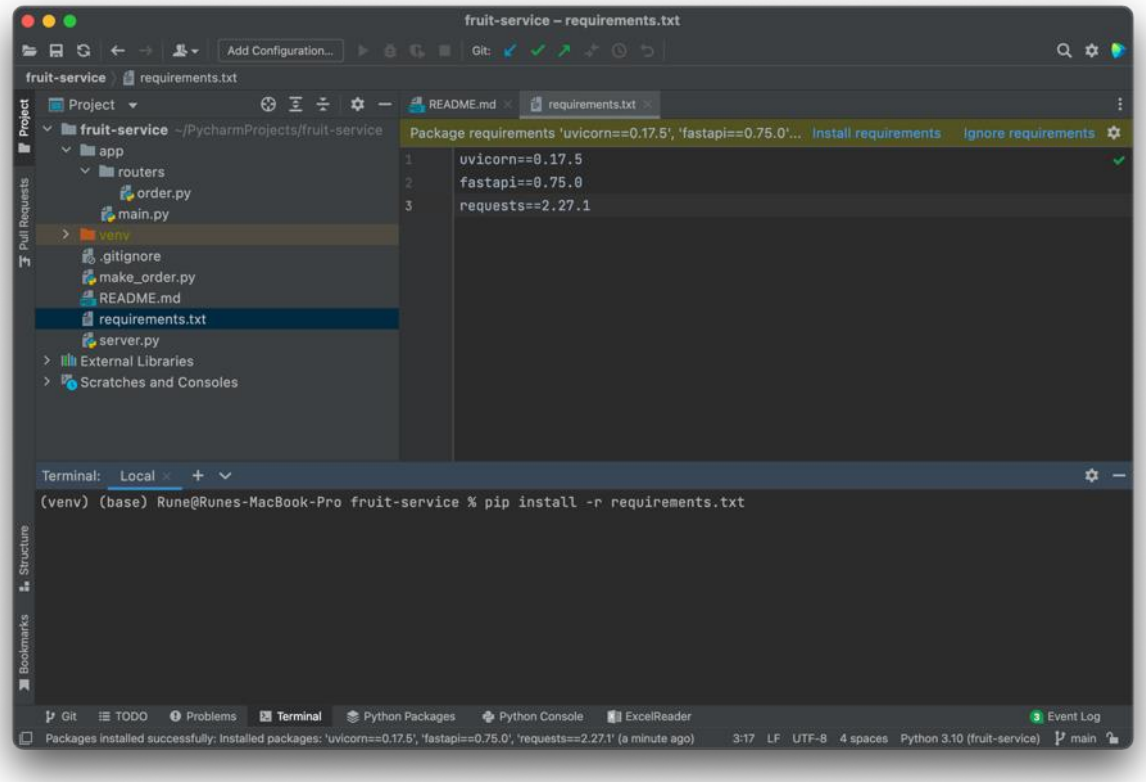
```
uvicorn==0.17.5
fastapi==0.75.0
requests==2.27.1
```

They are added with versions, which is not always the case. But what it tells us, is, that we need to install uvicorn version 0.17.5, fastapi version 0.75.0 and request version 2.27.1.

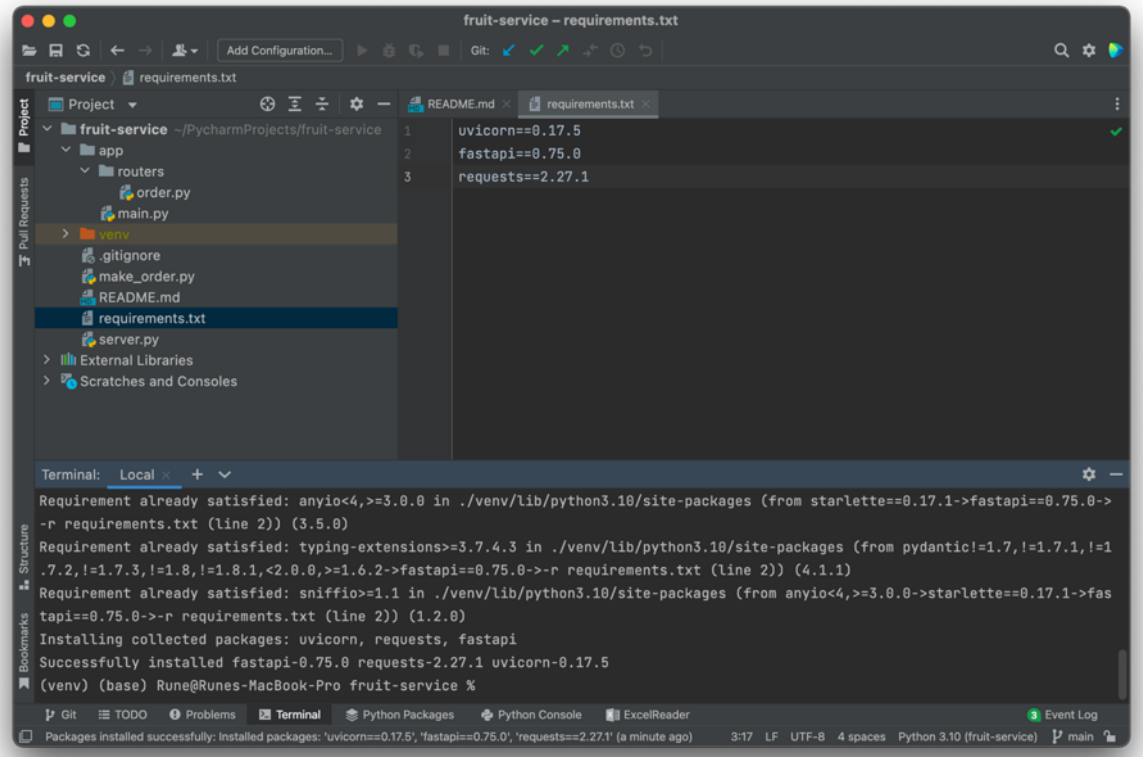
How do we do that?

Well, there are many ways and I will show you two ways to do.

- **Option 1**
  - Click on “Install requirements” above the lines in the view. This will do it automatically for you.
- **Option 2**
  - Use pip in the terminal.
  - Open the terminal in the bottom of PyCharm
  - Write: pip install -r requirements.txt



  - Hit enter



- Now it has installed all three libraries needed for the project.

Why bother about Option 2?

Well, it teaches you how it works when we will build Docker images later. We will use that command.

.gitignore

It tells Git which files to ignore when committing your project. For the most part, you can ignore the file as well.

The files: server.py, app/main.py, and app/routers/order.py

These files are the basis for the REST API we will run in a moment. They are all connected together and use the **FastAPI Python framework**.

While the API is very simple, and could be implement using a single file, I wanted you to show you how an API could be structured in a bigger project.

Why FastAPI?

We could use other frameworks, but FastAPI is a simple to learn and understand.

In a moment we will explore it further.

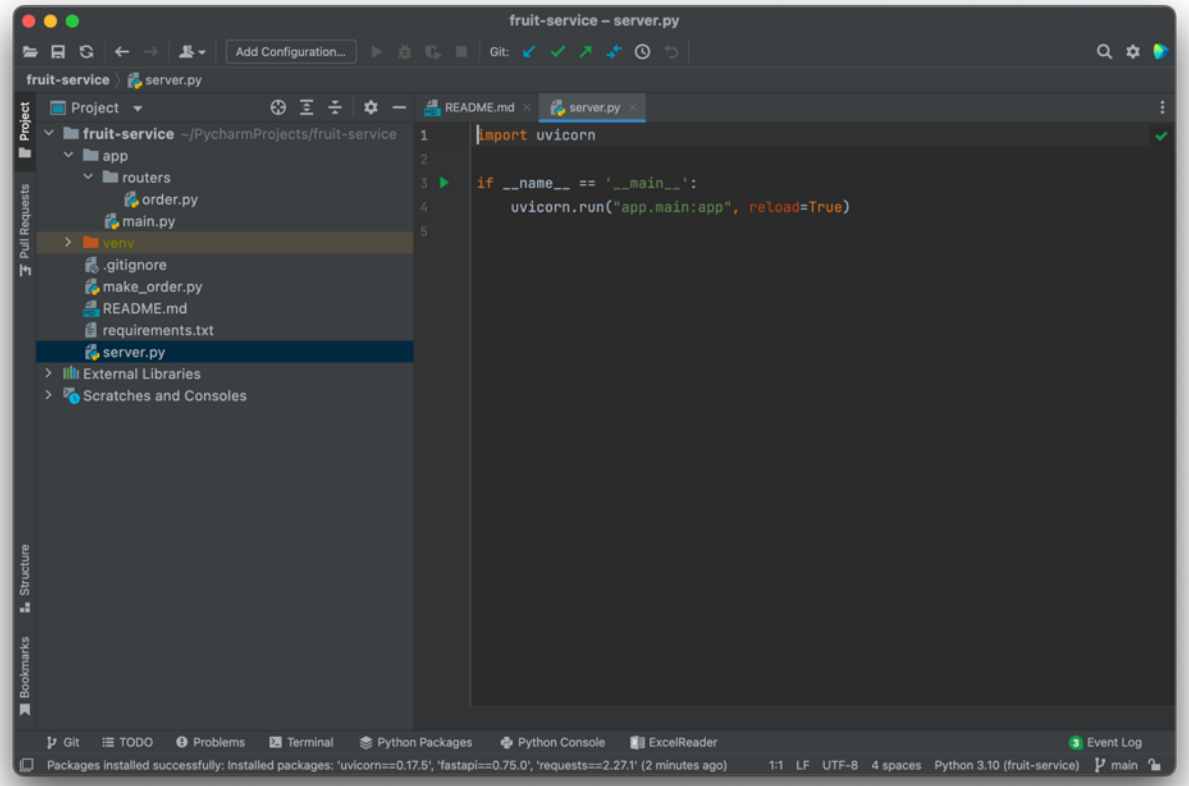
make_order.py

This script calls the API.

That is, when the API is running, you can use this script to call the API.

Let's explore the API code

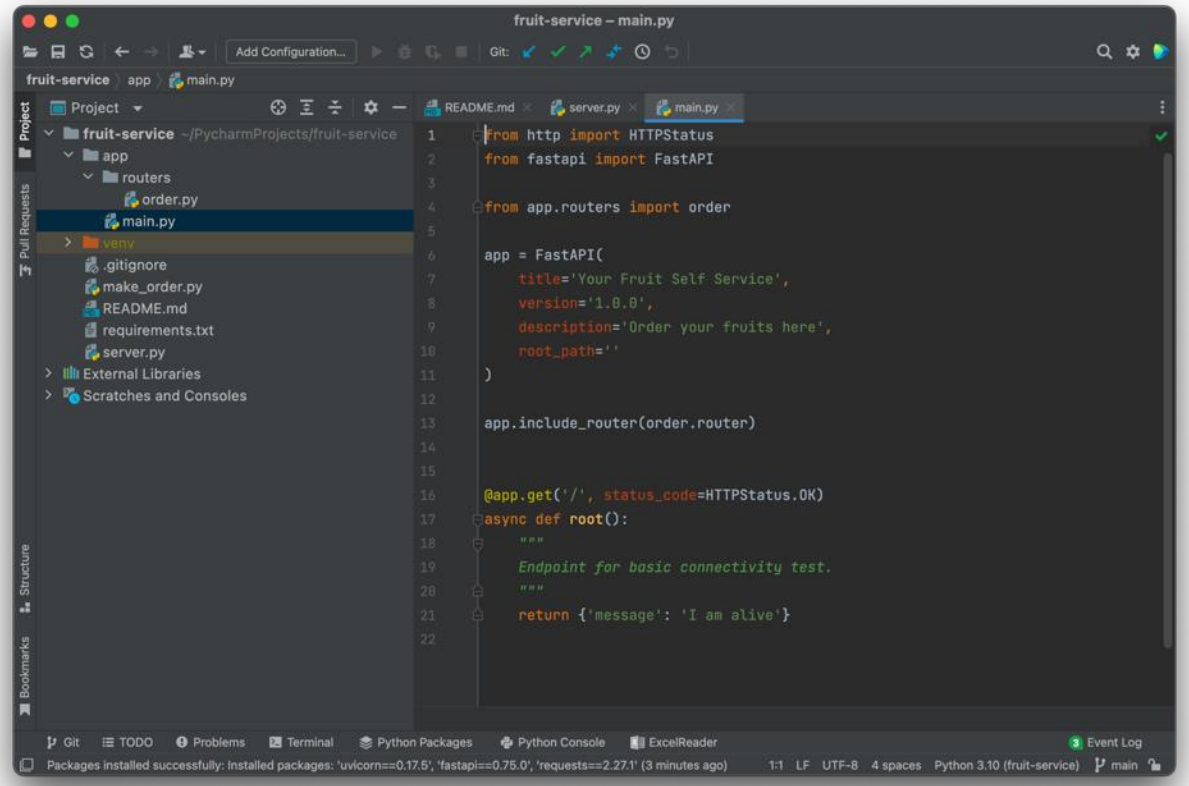
Open the server.py file.



This will start the server and make the API available.

What does the code mean?

Well, it uses the uvicorn to start app.main. Let's explore **app/main.py**



This is not a FastAPI tutorial, but let's get an idea of how it works.

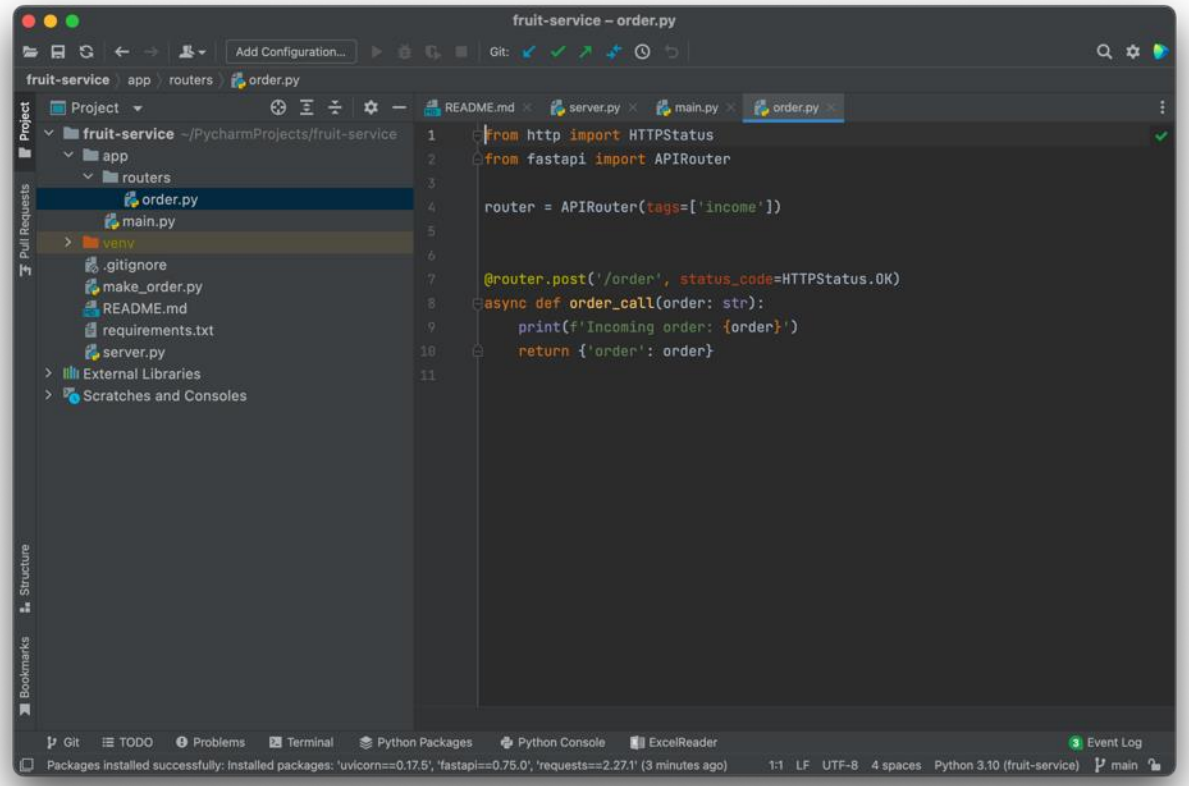
The file **server.py** sets up the **app** in **app/main.py**. As you see on line 6 to 11 it is initialized with some parameters.

On line 13 we include **router** from **order**, and **order** is imported on line 4, which is the file **app/routers/order.py**.

Finally, on line 16 to 21 it adds an API endpoint. This will be the default endpoint, which often will be used to test connectivity. It is a GET (line 16) and will return status OK (line 16) and the content on line 21.

In a moment it will be clearer, when we run it and try it.

Let's open **app/routers/order.py** and investigate it.



This is actually the endpoint we want to expose. It is added on lines 7 to 10.

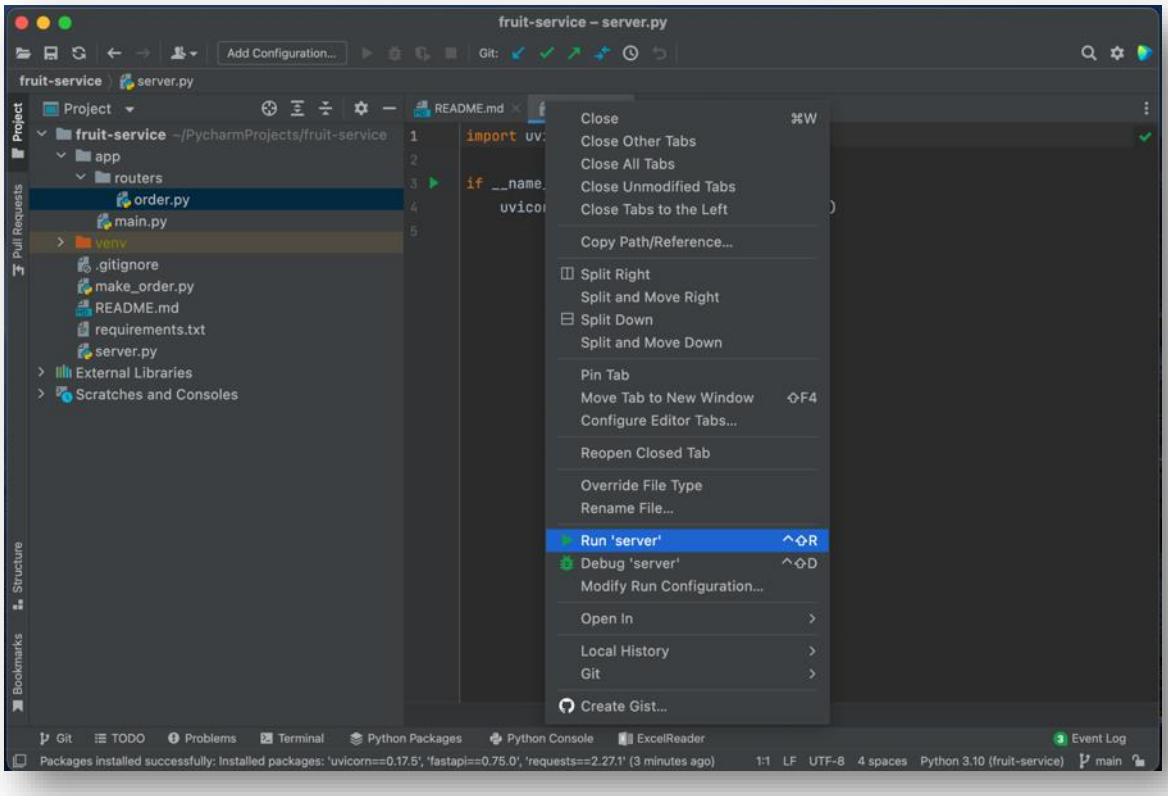
It is a POST endpoint called /order. It takes an argument order. It will print the order and return it.

Of course, this is not the functionality we are aiming for. This is only to demonstrate it.

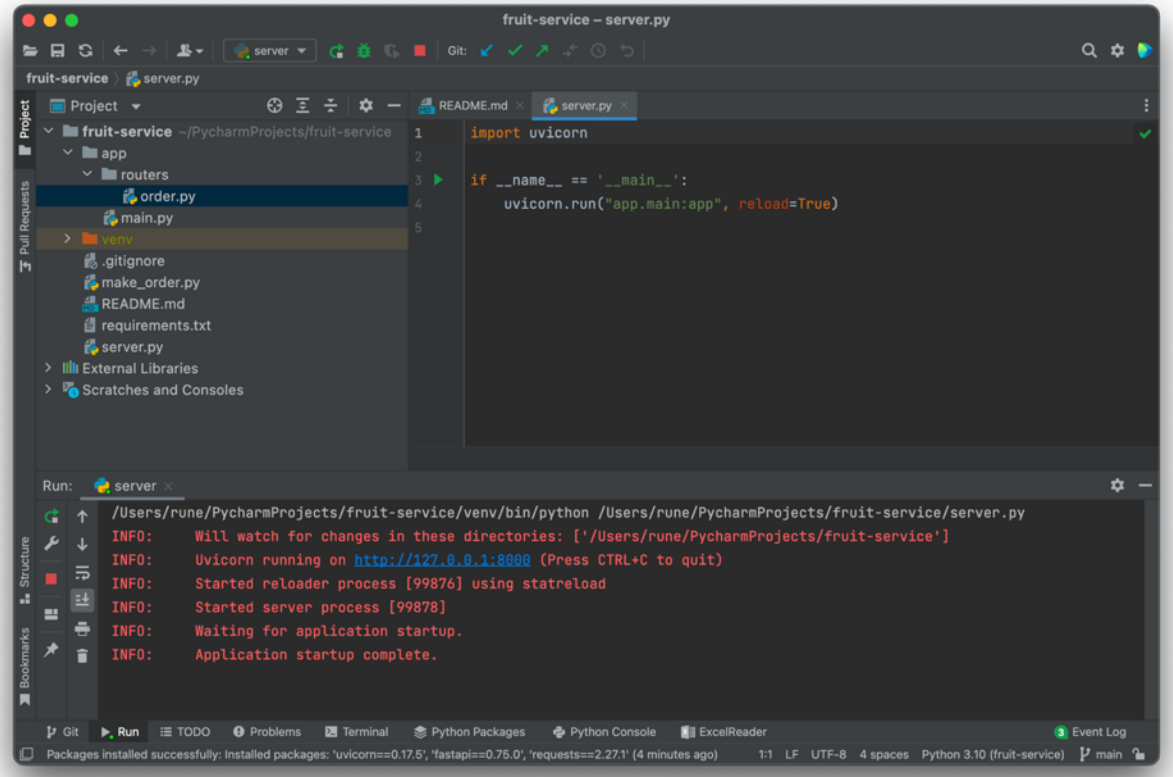
Are you ready for that?

Starting the API in PyCharm

One way to start the server is to right click on the server.py tab and choose run.



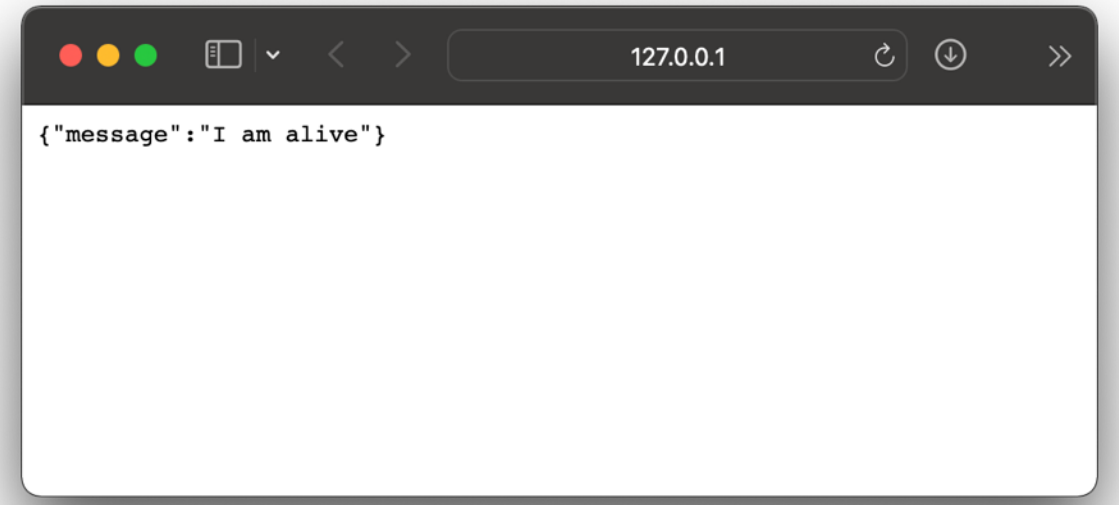
This should result in.



Now we are running the API. It is running on your localhost on port 8000.

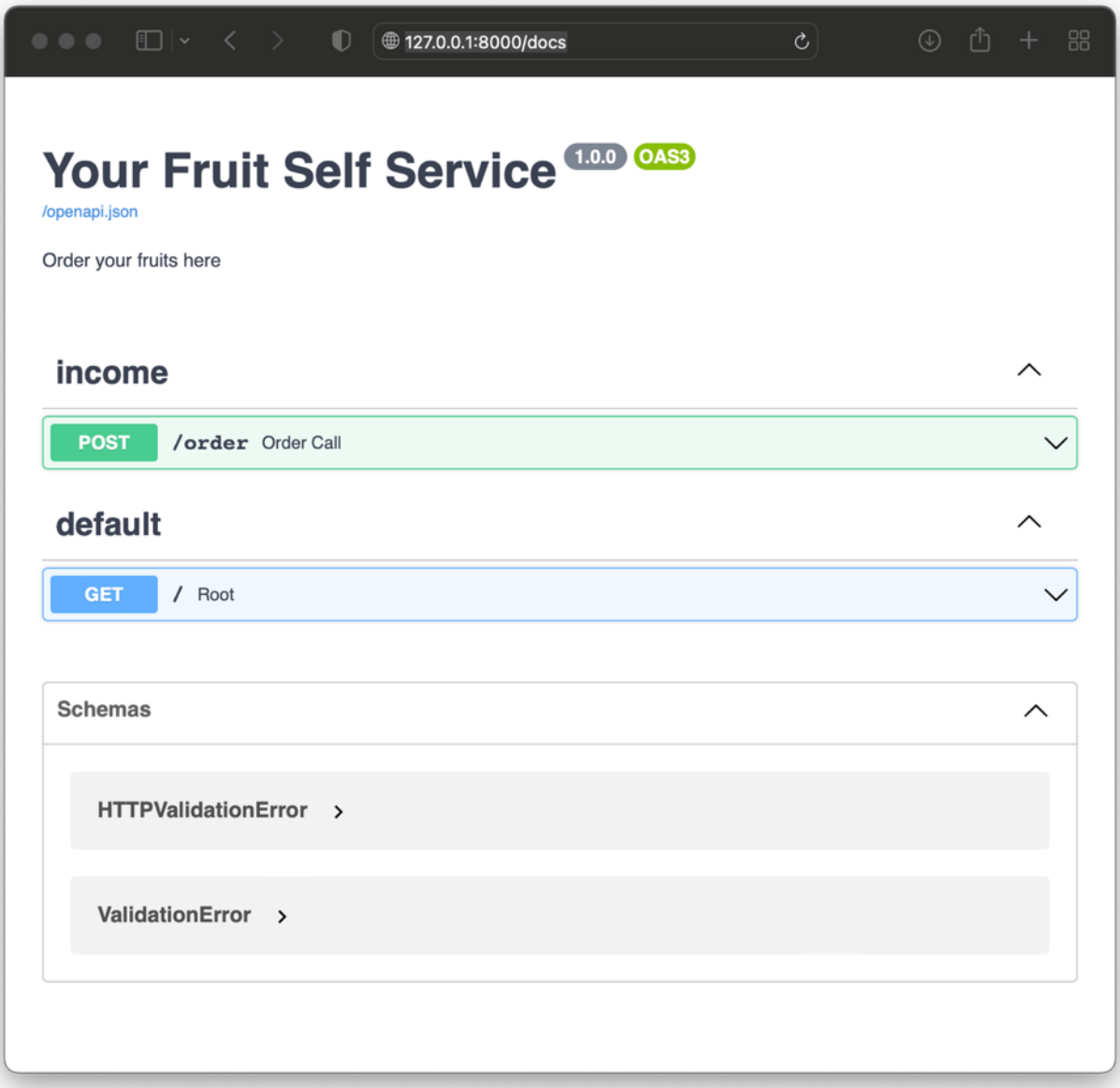
You can see that by the **http://127.0.0.1:8000**

And if you press the **http://127.0.0.1:8000** it will open in your browser.



How to call the API?

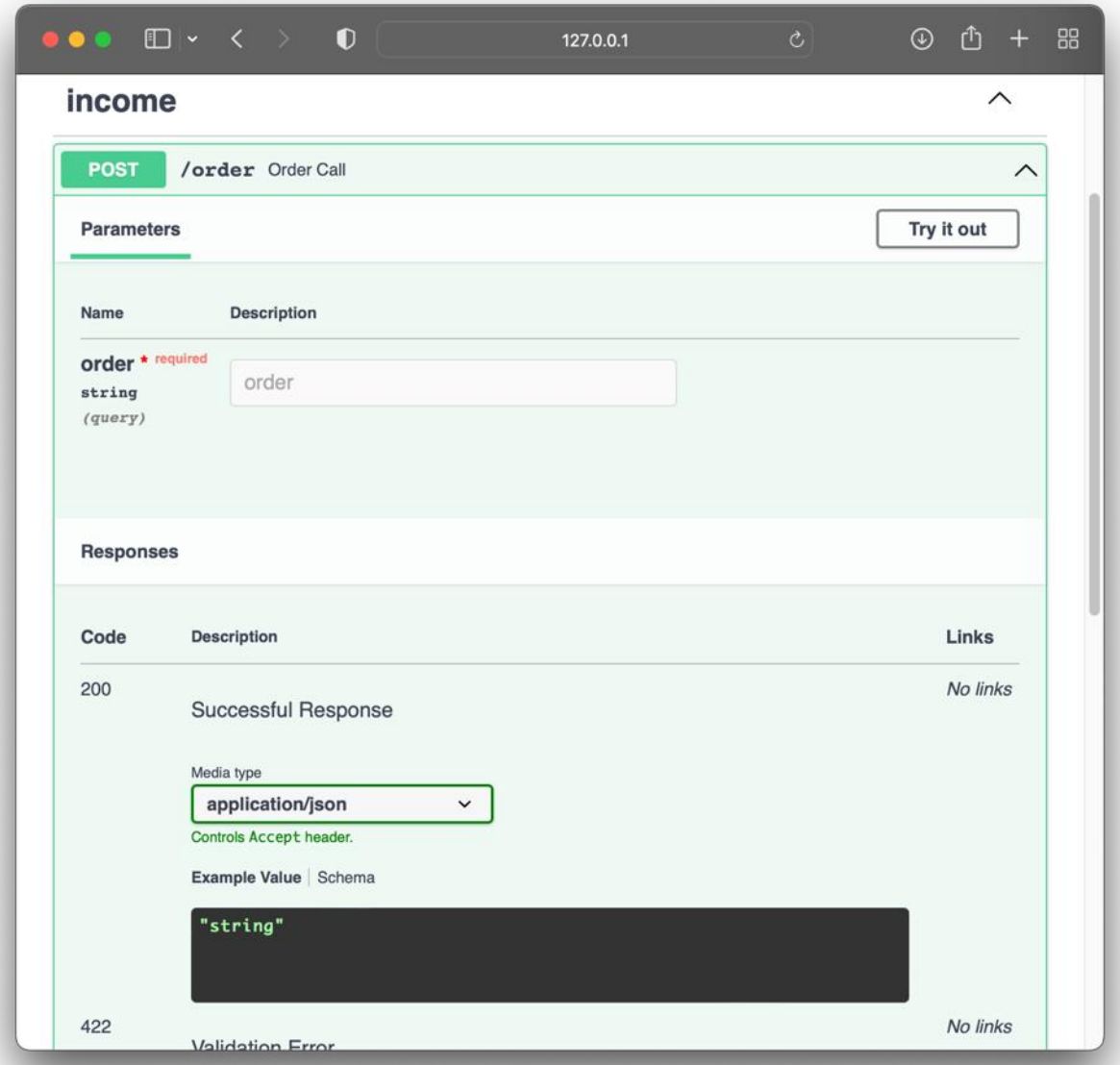
You can use the Swagger docs here: **http://127.0.0.1:8000/docs**



This shows the two endpoints we have exposed. The GET / root endpoint and the POST /order endpoint.

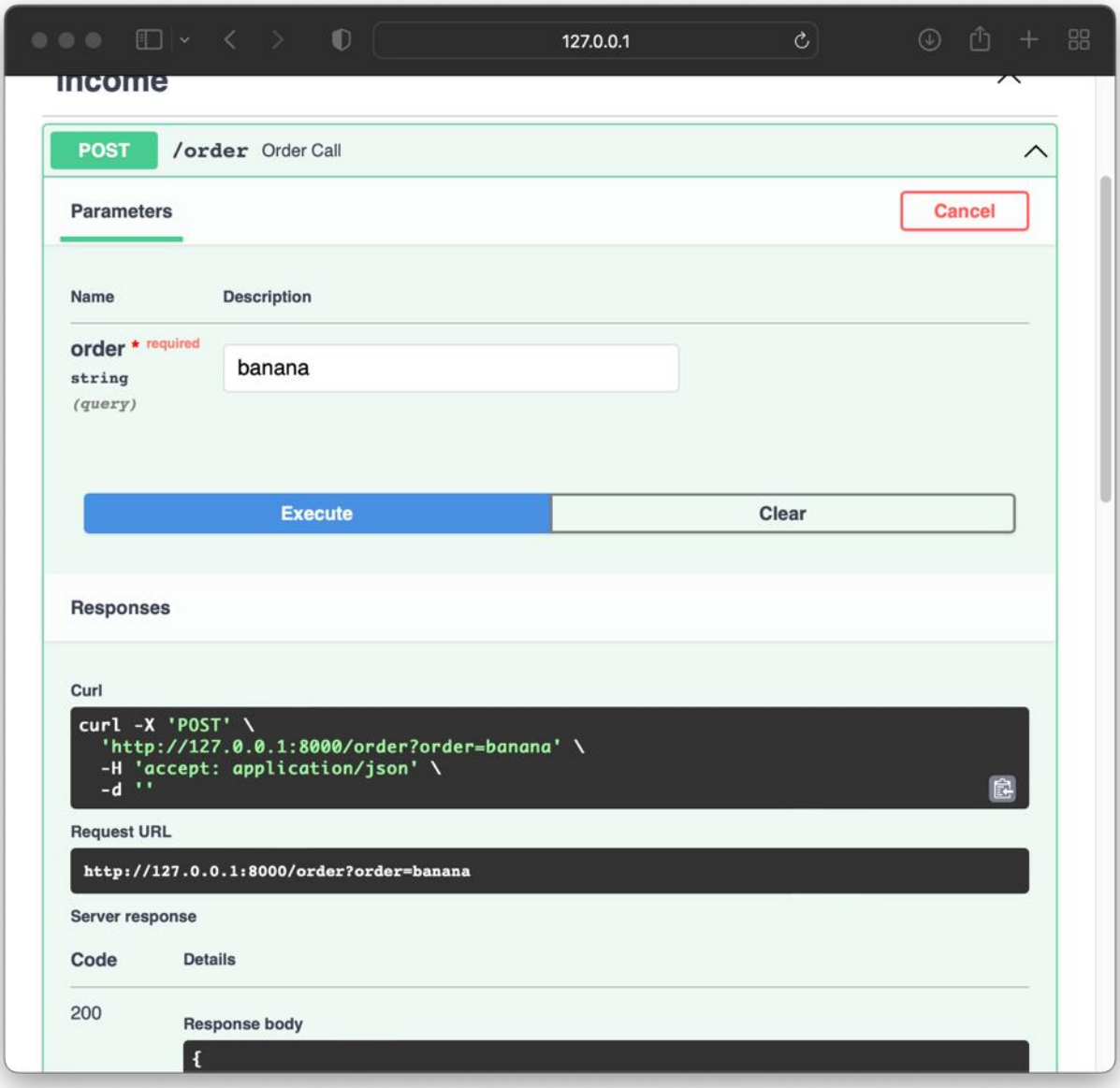
And the good part is, that you can actually call them from the **Swagger UI**.

Try to expand the POST /order.



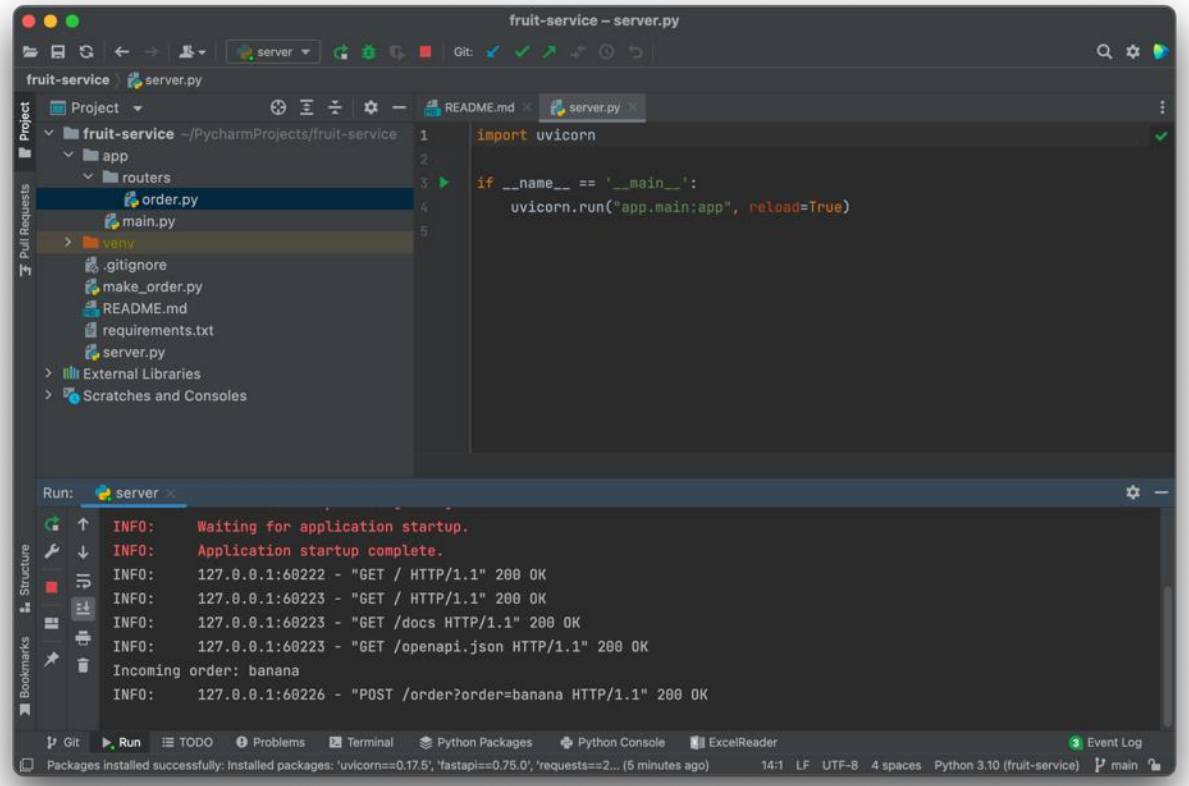
Then press the **Try it out** (top right corner).

Type in order field and press **Execute**.



What happened?

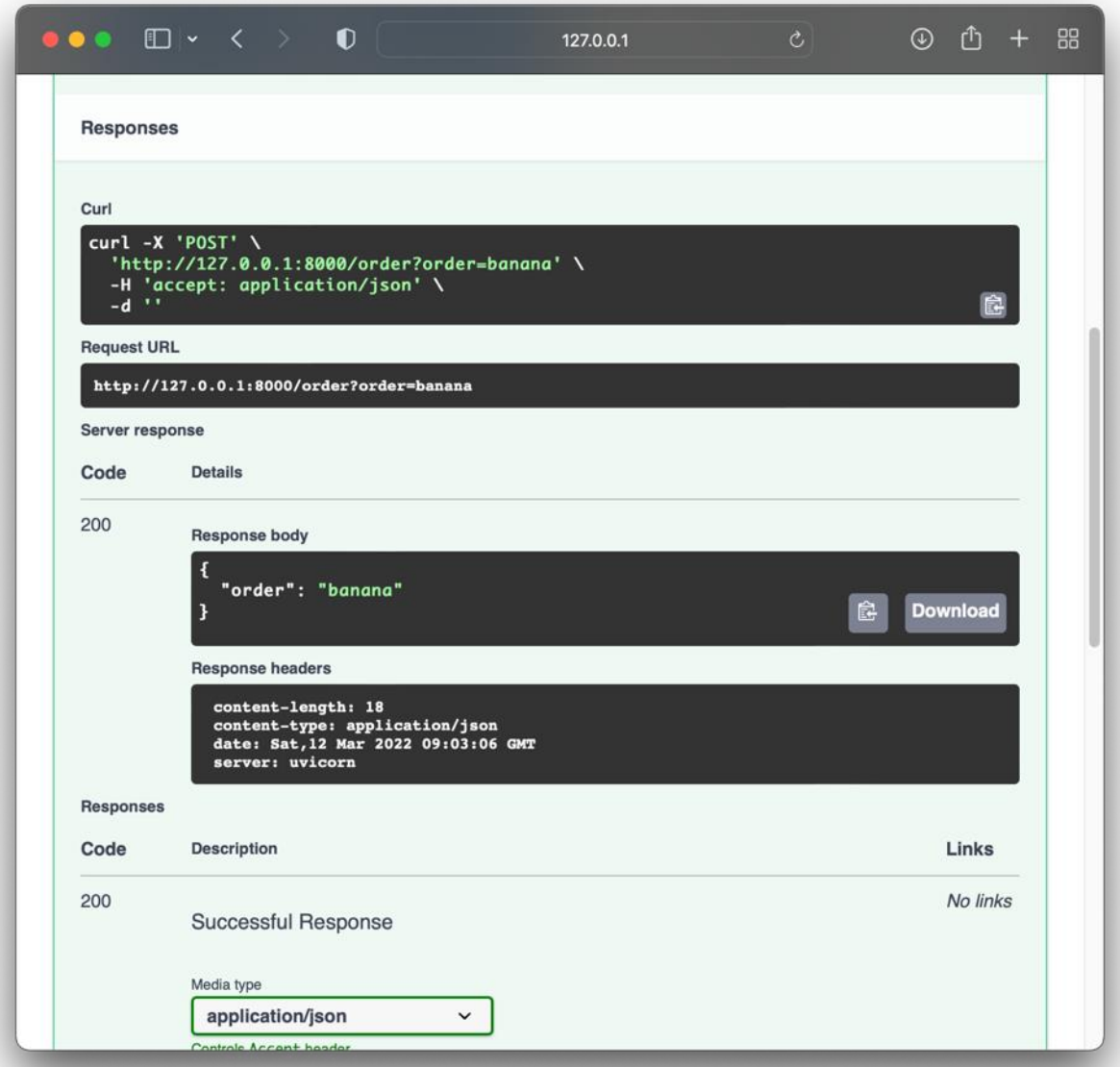
If you check back in PyCharm you might notice something.



It says, **Incoming order: banana**.

Wow.

And if you inspect the Swagger UI you will notice the response.



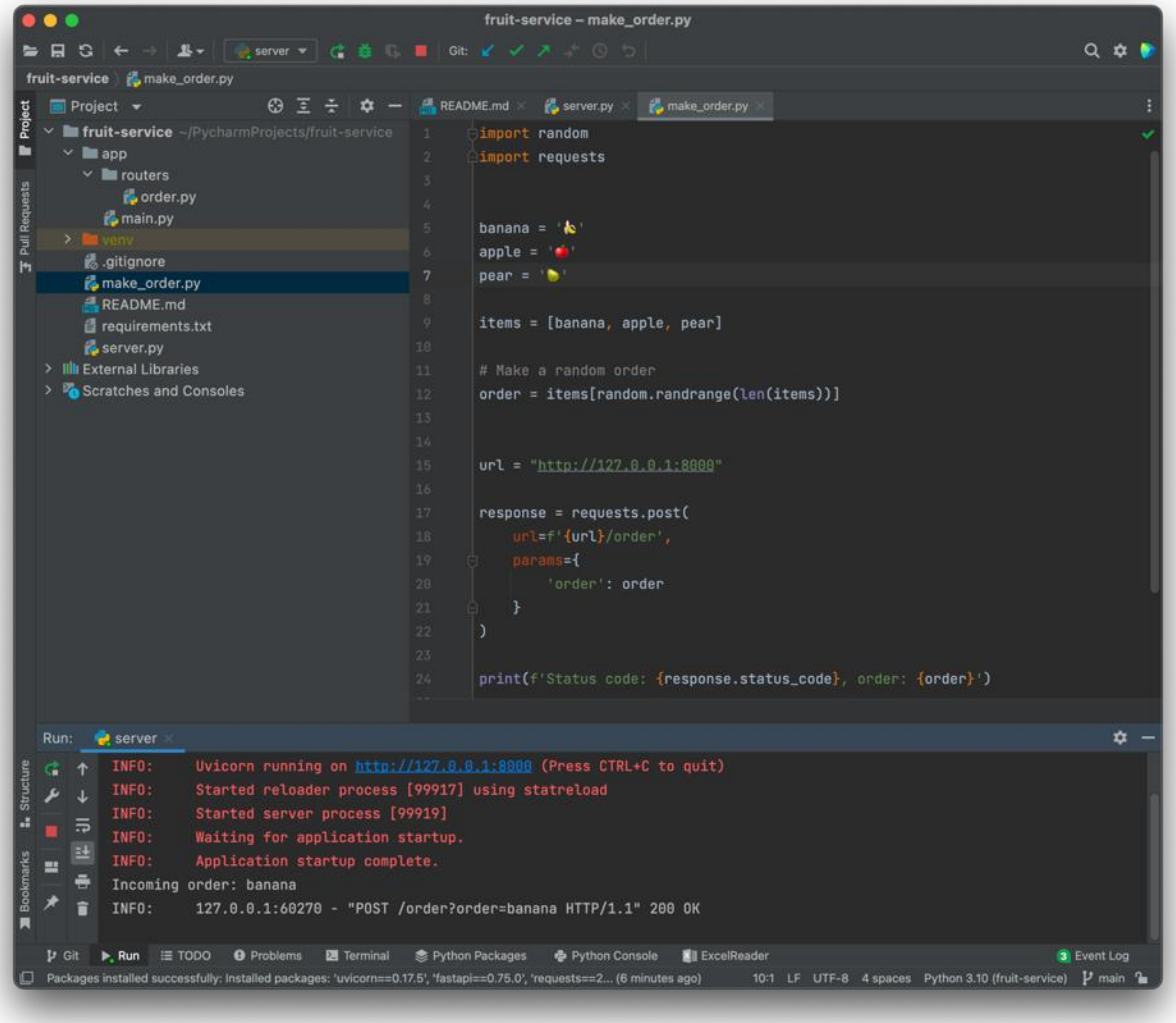
It says, **{"order": "banana"}**.

If you remember the code, it actually returns that.

It works!

What about make_order.py?

Good question. Let's open it.



If you should make a wild guess, what does it do?

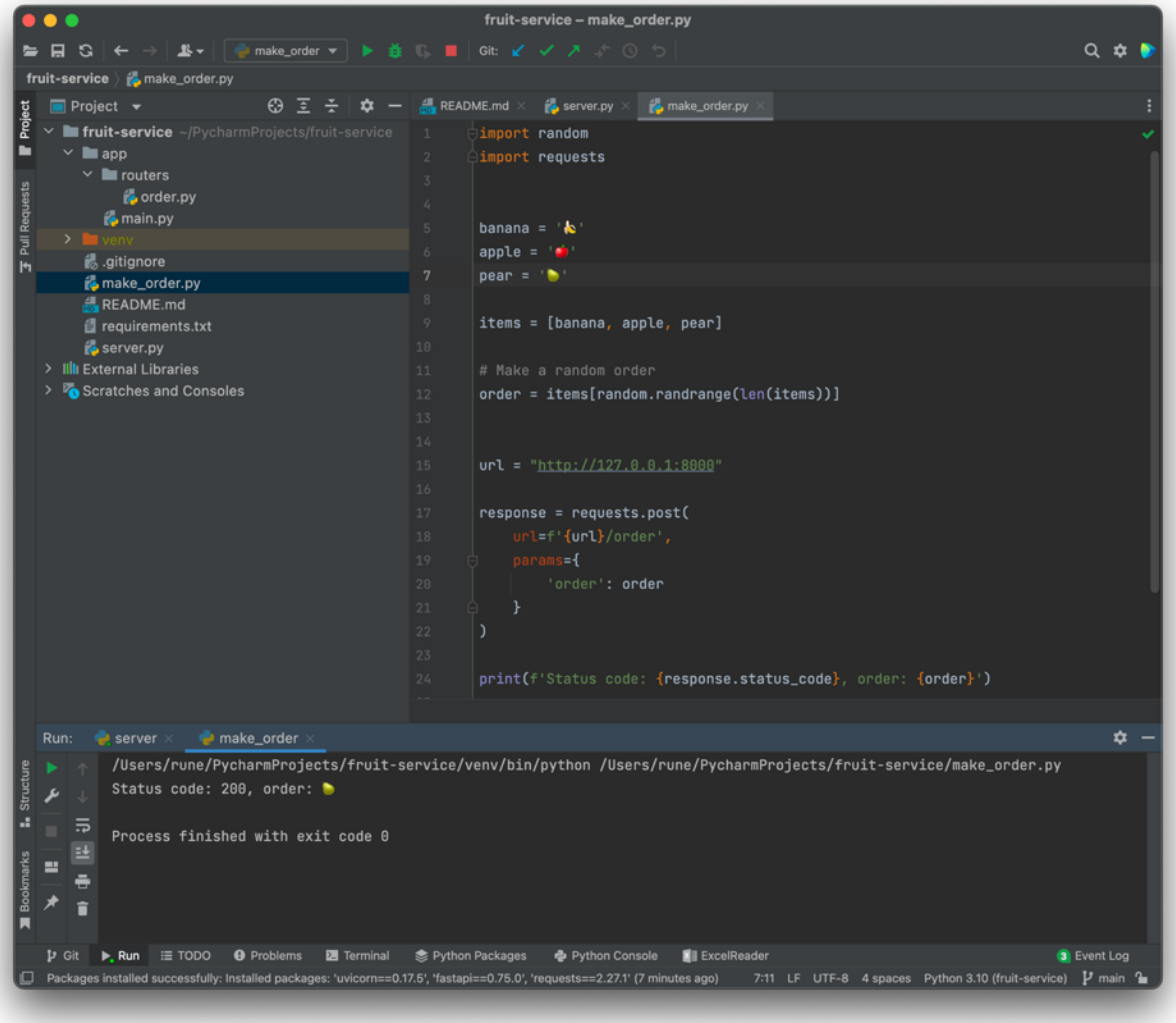
On line 12, it picks a random item. An item can be either a banana, apple or pear.

Then on lines 17 to 22 it makes a POST request to /order adding the parameter order (the random item).

Finally, on line 24 it prints the status code and order.

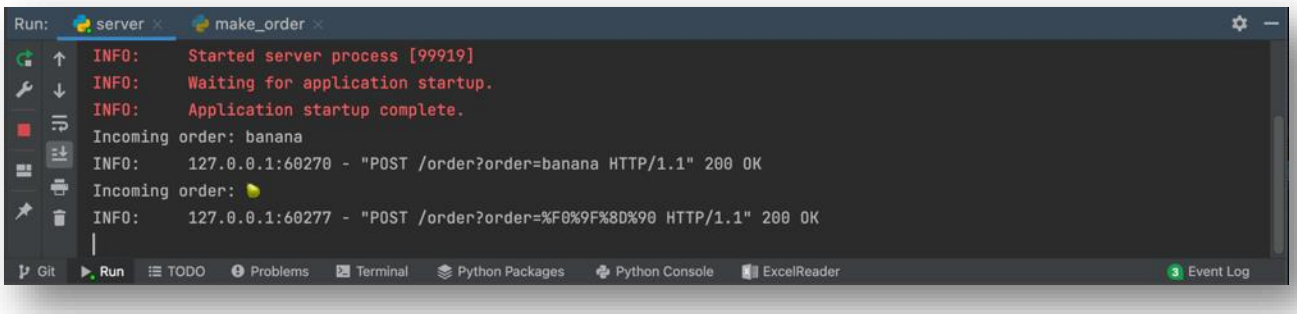
Can we just run it while the server is running?

Yes, actually it will only work if the **server.py** is running. To do that right click on **make_order.py** and choose **run** (just like we did with **server.py**).



In my case the status code was 200 (HTTP OK) and the order was a banana (you might get another order).

If you click on the **server** tab next to the **make_order** – then you will see the following.



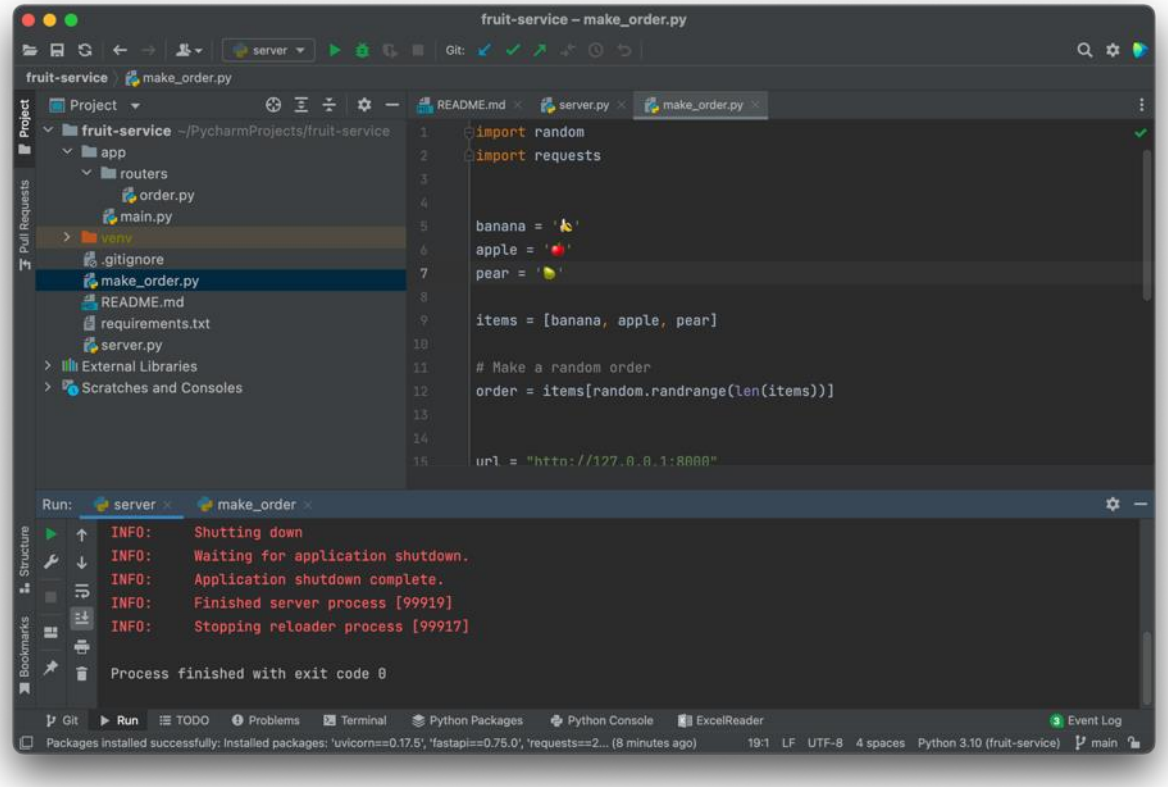
Incoming order: pear.

How fun is that?

Now we have learned how to run an API and make calls to it using either the Swagger UI or a Python script.

After you are done playing with it – you should shut down the server. You can do that by pressing the red stop.

This should result in the server is stopping in the run view.



Now that was a lot of things to learn.

Exercise

FastAPI is a great framework to create REST API's. A great idea is to work a bit with it. The documentation has examples to get you started.

Make your own repository with a Hello World API.

- Make a new project in PyCharm.
- Setup the Python environment.
- Install fastapi and uvicorn.
    - Execute the following commands in the PyCharm terminal.

    - pip install fastapi
    - pip install uvicorn[standard]
- Implement a root endpoint which returns “Hello World”.
    - HINT: Look at the code in main.py

Solution of the code can be found in the main docs: **https://fastapi.tiangolo.com/tutorial/first-steps/**

Summary

In this chapter we learned the following.

- How to clone a project from GitHub in PyCharm.
- Setup a virtual environment in PyCharm with the correct Python interpreter.
- Understanding the files in the project.
- How **requirement.txt** is used to describe the libraries in the project and how to install them.
- The a typical FastAPI is structured in multiple files.
- How to run the API in PyCharm.
- To make calls to the API using the Swagger UI.
- Also, how to make calls to the API using a Python script.

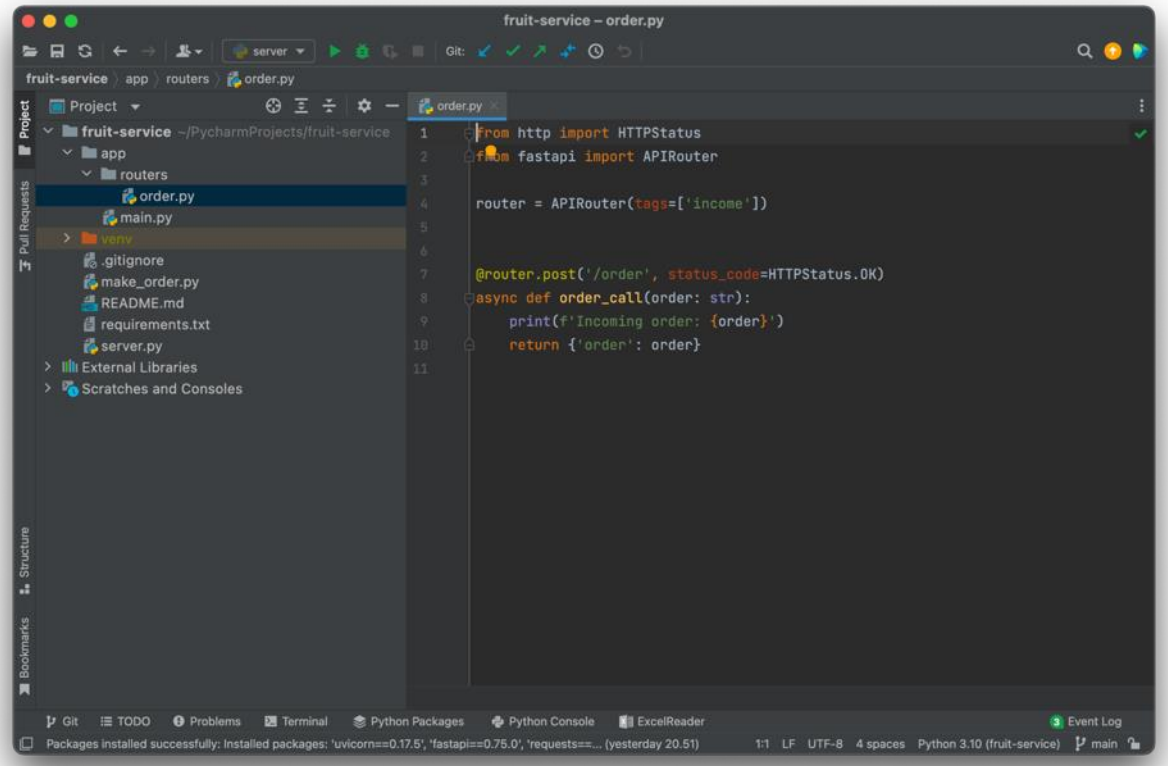
02 – Logging

In this chapter we will explore logging.

- Why do we need logging?
- Why not just use print statements?
- How logging works
- Adding logging to our project

First step – how will we proceed?

We will work with the same repository from last chapter.



We will add some logging to it and explain from that how it works.

Why do we need logging?

As a starting developer you focus – and I did too – mostly on getting the program to get the job done and less on how it was being done.

Little thought was given on design and best practices.

Over time this change, and you might wonder why?

Debugging. Extending the code. Adding modules to the ecosystem.

You might only relate to the first one, debugging.

That is, your program is not doing as intended, and it might be difficult to find the bug. Yes, you add print statements all over the code to figure out how it works. If you are more advanced, you might be using the PyCharm debugger to find it.

Bottom line is, it is difficult.

When your module become part of a bigger ecosystem, the non-intended behavior might be difficult to figure out – and you might not know where the error is.

Then adding print statements to all the modules in the ecosystem is not desirable.

While logging will not solve the problems, it will be a good tool to do that.

But logging is used for more than debugging.

- **Issue diagnosis**. Your service crashes when some user does something. Well, the user might not really remember what was causing it. Then good logging can help you figuring out how to replicate the scenario in your development environment.
- **Analytics**. Logs can give you information about load on your services, when and what modules are being used the most. This can help you to improve the experience for users.

Why not just use print statements?

We already discovered when you have bugs to catch, you will often insert print statements to see what happens. These print statements need to be removed afterwards.

That might be a lot of work. Especially, if the bug you are hunting might be part of several modules.

Also, we learned that logging is also used for issue diagnosis and analytics.

Yes, you might build your own way of making issue diagnosis and analytics, and it might work. But if you use standard modules for logging, it will integrate easy with other systems.

Don't build your own – if there is a good standard way to do it.

This holds for logging. As we will see later – we will make an easy integration of all our logs into Grafana.

When we learn a bit more about logs, you will also realize, that logs have different levels. One level is for debugging – the lowest level – where you get a lot of information to help you find the bug. This is equivalent to adding all the print statements – and when you are done – it will automatically remove them again. All done by adjusting the debug level.

How logging works

Logging comes in different levels.

- **DEBUG**. Used to diagnose problems.

- **INFO**. Confirmation on things are working as expected.
- **WARNING**. Something unexpected happened or indicating some problem in the near future (but software is still working as expected).
- **ERROR**. The software has not been able to perform some function as expected.
- **CRITICAL**. A serious error. Program might not be able to continue running.

See the **official docs here**.

A simple example of how logging works is given here.

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

This might be strange. But the default logging level is **WARNING,** which means that all logging from **WARNING** and above (**ERROR** and **CRITICAL**) will be output.

On the other hand, if logging level is set to **DEBUG** – then all logging messages will be output.

This can be achieved as follows as well as writing the log to a file.

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8',
level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

This will output all the logging messages to the file **example.log**.

Some good practices with logging

The best advice with logging is, do not add too many logs. At first you might want to have logs all over.

Use them as intended from the schema.

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.

Level	When it's used
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

The second thing to consider is, what to log?

- **When**. Always log the time – when you look at different logs form different services communicating with each other – the time stamp will help you. Also, just to identify when something happened, if it correlates with an incident.
- **Where**. What application or what file is the log coming from. This is also crucial when you have logs from many modules.
- **Level**. It is a great idea to have the level, it makes it easy to identify **WARNINGS** or similar.
- **What**. Then the actual message – what happened.

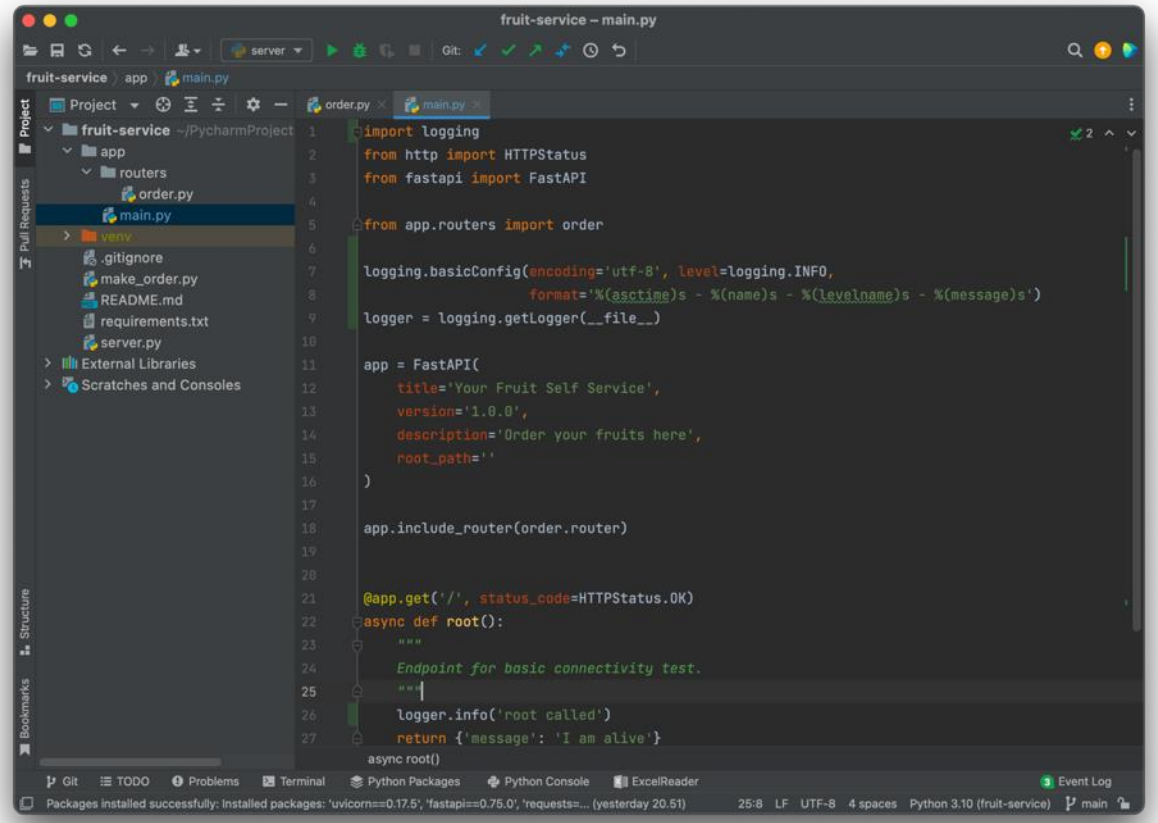
There are different ways to configure the logging.

Common ways include a log configuration file or directly in the code.

Here we will keep it simple and do it directly in the code.

Adding some logs to our API

Here we have added some logging to keep track on what is being called. So far, not much can go wrong. We just want to know it goes as expected.



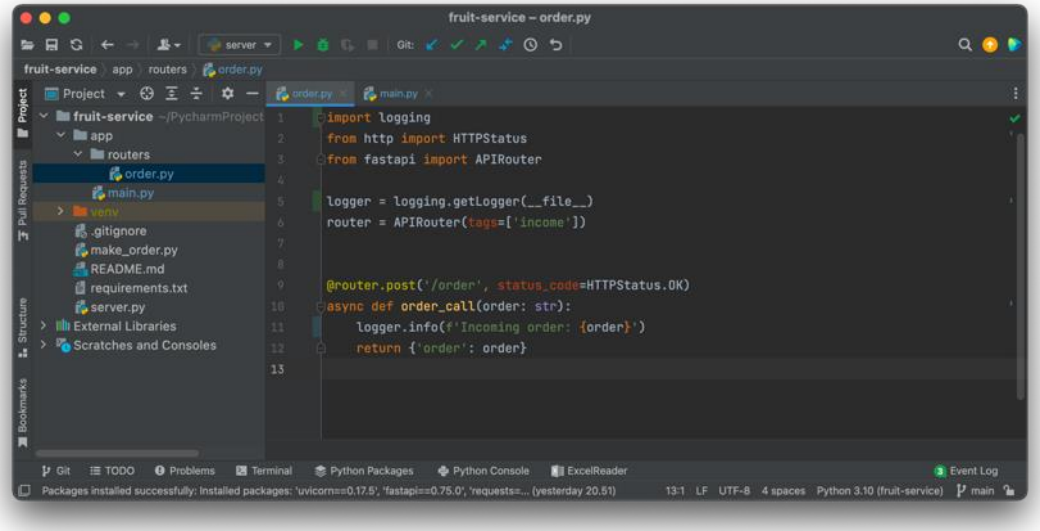
The main lines:

```
logging.basicConfig(encoding='utf-8', level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__file__)
```

You can see the new code indicated with the green indicator at the line numbers.

- **Line 1**. On line one we import the **logging** library.
- **Line 7-8**. We make a basic configuration – using encoding utf-8, logging level INFO and an output format with the *time*, *name*, *level*, and *message*.
- **Line 9**. Get a **logger** with the file name (you will see how it helps in a moment).
- **Line 26**. Creates an **INFO** log message.

Let's explore **order.py** as well.



Notice that we do not need to configure the logging module again. It is done once in **main.py**.

Here we only get the **logger** with the file name.

- **Line 1**. Import the logger
- **Line 5**. Sets the logger with the file name.
- **Line 11**. Creates an **INFO** log message (the print statement before).

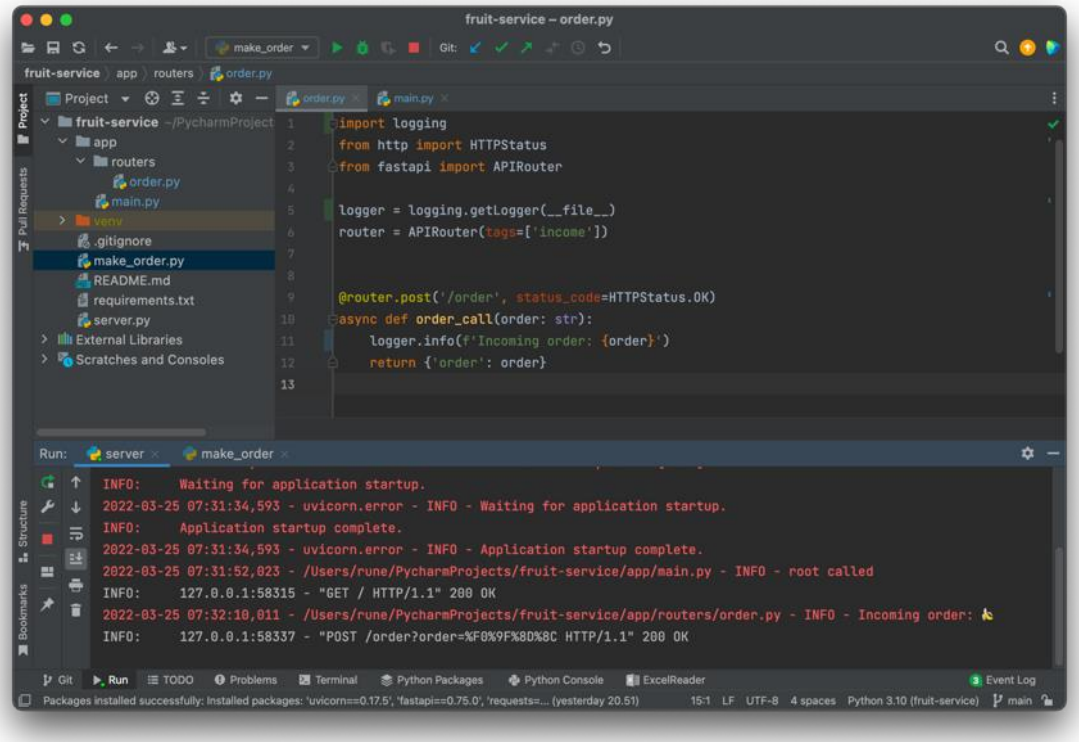
Run the server and call it

Now start the server again.

Right click on the **server.py** tab and click **run**.

Then run **make_order.py** (right click **make_order.py** click **run**).

Possibly use Swagger UI to call the root endpoint.



As you see in the output you have log entries.

> 2022-03-25 07:31:52,023 - /Users/rune/PycharmProjects/fruit-service/app/main.py - INFO - root called
>
> 2022-03-25 07:32:10,011 - /Users/rune/PycharmProjects/fruit-service/app/routers/order.py - INFO - Incoming order: 🍌

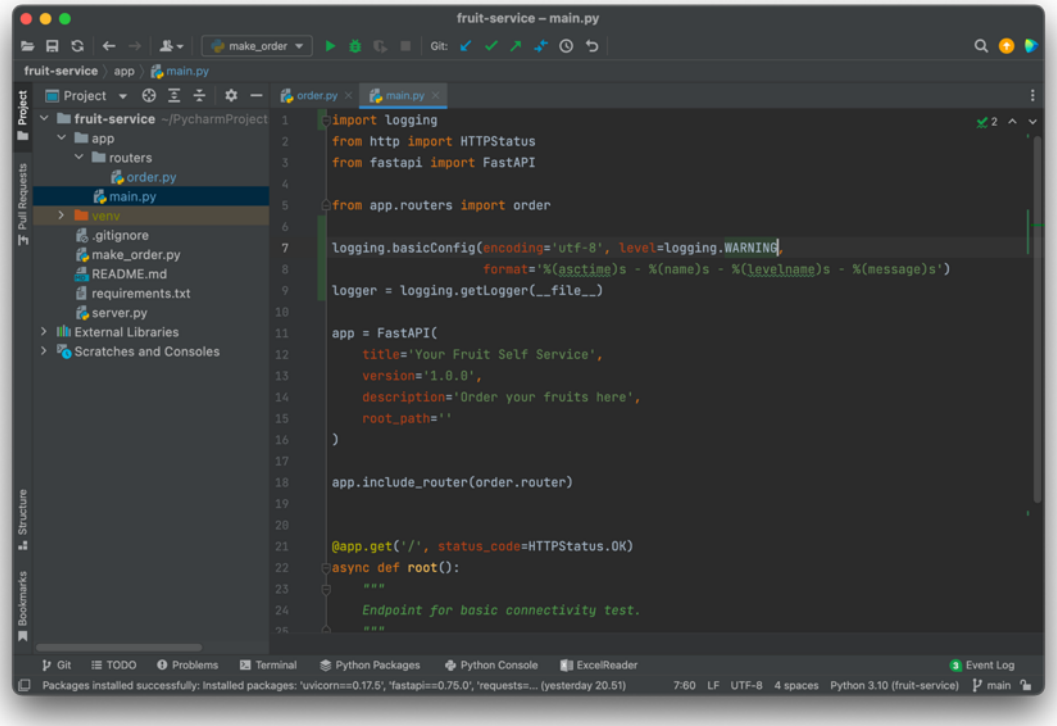
Of course, it is good practice not to use symbols like the banana here.

But notice you get the file of where the log message comes from. This is nice, when you are inspecting the logs.

You might notice, that the Uvicorn server also adds a lot of logs – with a different format.

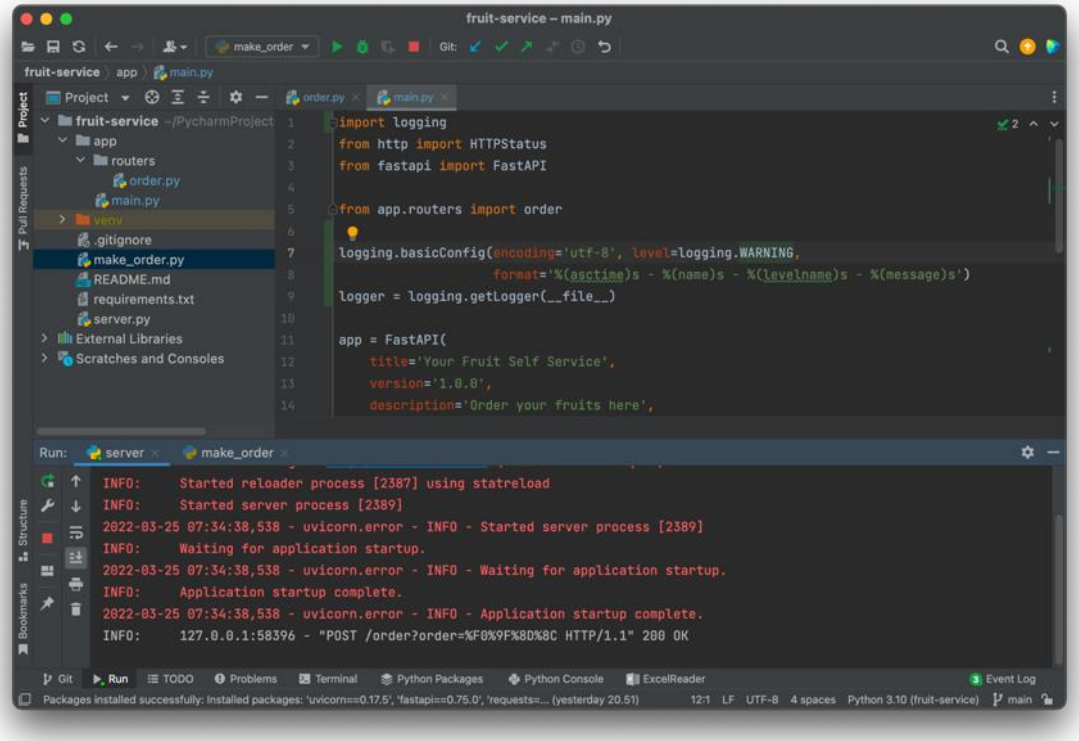
Changing the log level

Let's change the log level in the code to **WARNING** in **main.py.**



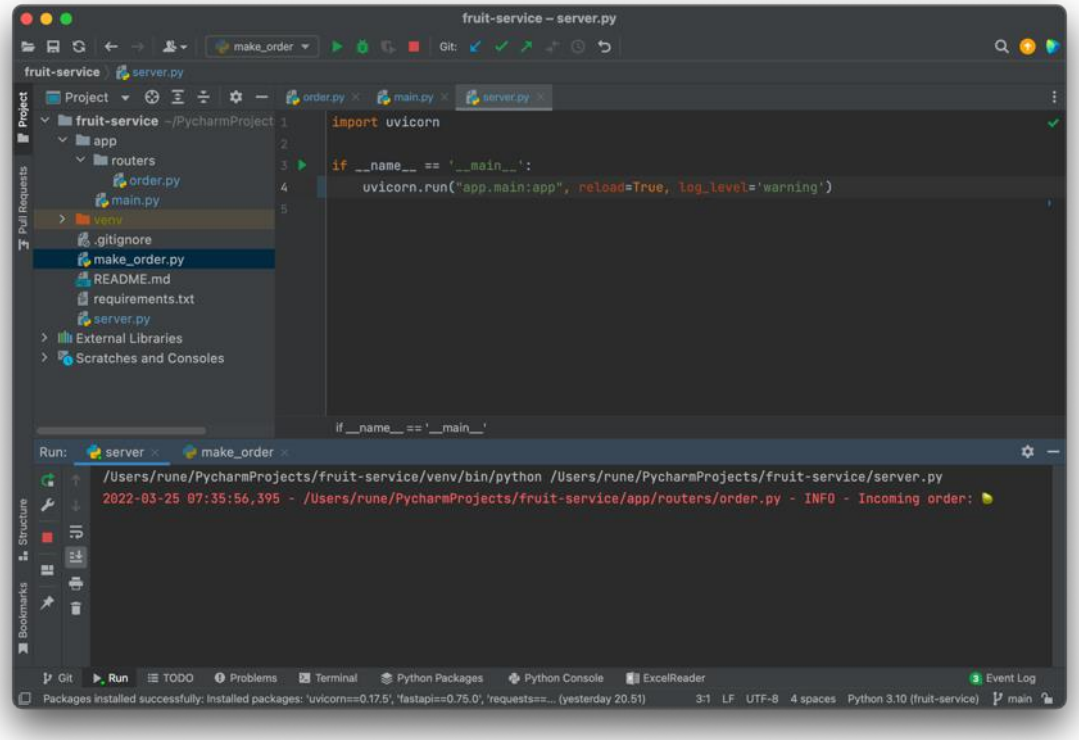
Now run it all again.

Notice, if you didn't shut down the server, it will restart itself when you make changes.



This shows that the log is gone but the Uvicorn **INFO** log is still there.

You can change the log level of Uvicorn in **server.py**.



The available levels are **critical**, **error**, **warning**, **info**, **debug**, **trace**, where the default is **info**.

This means you can set the log level of Uvicorn independently of our API.

Notice, that in the above screen I set the log level in **app/main.py** to **INFO** again.

More on Logging and best practices

In this project we use to name the logger after the file (__file__) – this makes it easy to start with and is simple to exactly locate the file where the log comes from.

It is common to use __name__ in when building modules, as it keeps a qualified name of the module. It has the advantage over __file__, that the output is shorter and as descriptive, also, it can have logging to inherit defined logging on a lower level.

It is beyond the scope to use logging on different levels and in this book, we keep it simple using the filename.

Exercise

It is always great to use and apply what you just have learned.

Continue with your code you made in the exercise in Chapter 01 and add appropriate logging to it.

- Configure you logging to print the standard output: time, where, level, message (see the format in this book).
- Create an info-log message for each request to your API.

Make sure it works as intended by changing log-levels and see the difference.

- Start by configuring it for INFO level, then try with WARNING and see if it has the desired effect.

Summary

In this lesson we have learned about logging and some best practices.

- What are the use cases of logging.
    - **Debugging**. Finding that nasty bug that is bugging you.
    - **Issue diagnosis.** When you get an issue and need to replicate it – then logs can help you figure out what happened.
    - **Analysis.** You want to know how much modules are used and by whom – then logs can be a great way.
- As a beginner it can be tempting to use print statements – do not fall trap for that urge.
- What are the different log-levels and how to use them.
    - Debug, info, warning, error, critical.
- How to make simple configuration of the logger and set log-levels.
    - A simple logging will have **time**, **filename**, **level**, and **message**.
- Then we added some info logs to our API.
- Finally, we set different log levels for Uvicorn and our API.
- The official Python logging guide is quite good (**docs**).
- Here we have covered the basics you need to understand. Specific setup very from project to project and organizations.

03 – tox + pytest + mypy

Oh my, what is all that? Well, it will help your code to become better and make some simple test.

- What is **tox** and why we use it?
- Adding unit tests with **pytest**.
- What is **mypy** and how to use it?

What is tox?

"It works on my machine!"

You just wrote this awesome program and a friend is trying it on her machine. Unfortunately, it doesn't work.

This is a well-known pain point, and you have no idea why it is the case and you end up saying, *it works on my machine!*

As you already know, there might be many reasons – different Python versions, different library versions, can play a factor.

Our goal is to deploy our code to Docker containers, but before that, we need to learn some good tools to help you get there easy and not have unexpected pain points when you deploy it.

This is where **tox** can help you.

> *"**tox** aims to automate and standardize testing in Python. It is part of a larger vision of easing the packaging, testing and release process of Python software."* (**tox.wiki**)

Maybe you understand that – I don't.

But what it does simply explained – it creates new virtual environments and tests the code.

Why do we need to do that?

- Say, you write the code and adds the **requirements.txt** file in the **GitHub** repository.
- Later someone else clones the code and installs the **requirements.txt** but it does not work.

There can be many reasons for that.

- There might be missing some environment variables.
- A setup script might be needed.
- Libraries missing in requirements.txt.
- Different Python versions can also cause the issue.
- You might be using different OS.

Well, **tox** can help you with all of that and more. Because **tox** makes it easy to:

- Test multiple Python versions.
- Test different dependency versions.
- Run setup commands.
- Isolate environment variables – as **tox** does not pass environment variable to the testing.

- Test against Windows, macOS, and Linux.

You see – this can highly improve the chances that your program works in different setups and help you understand what it takes to run it.

How does tox work?

I think the best way to think of **tox** is as follows.

- It will generate a series of virtual environments.
- Install the dependencies for each environment (defined in a config).
- Run setup commands and commands.
- Return the results from each run.

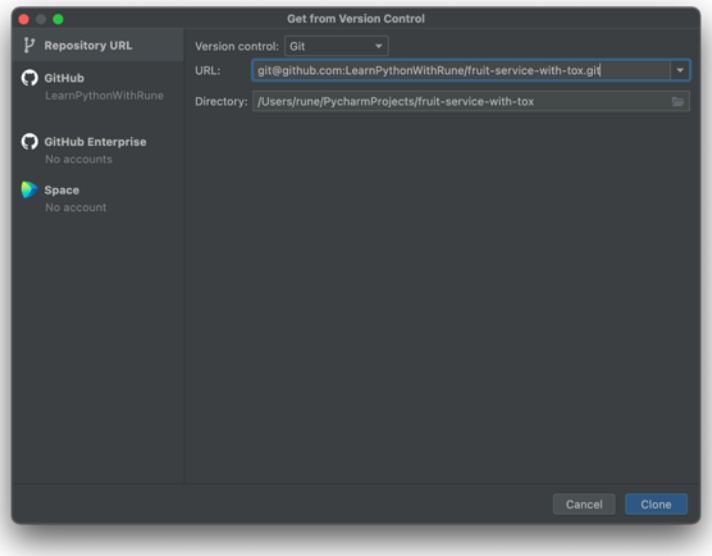
How to use tox – let's try it?

Good question. You need to install it and have a configuration file.

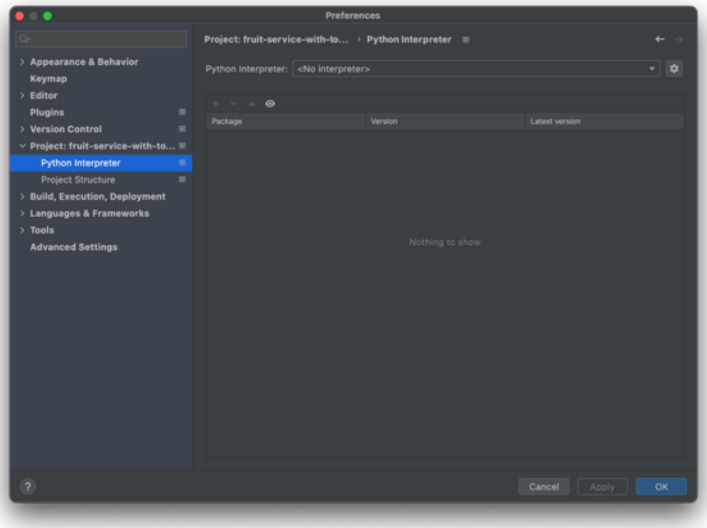
But let's clone a project with all you need.

Remember how to do that?

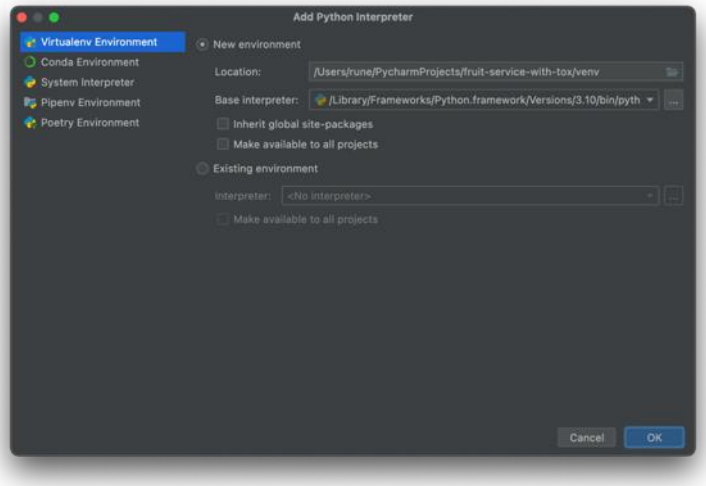
- **Clone**. You can use the Git -> Clone menu and insert **git@github.com:LearnPythonWithRune/fruit-service-with-tox.git**



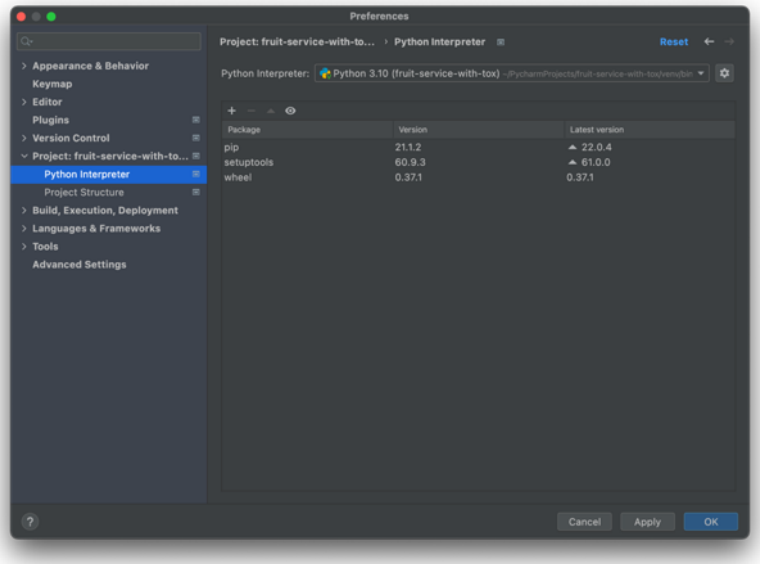
- Python interpreter. Preferences -> Project -> Python Interpreter. Use the tool to add one.



- Add the right Python interpreter (version 3.10)

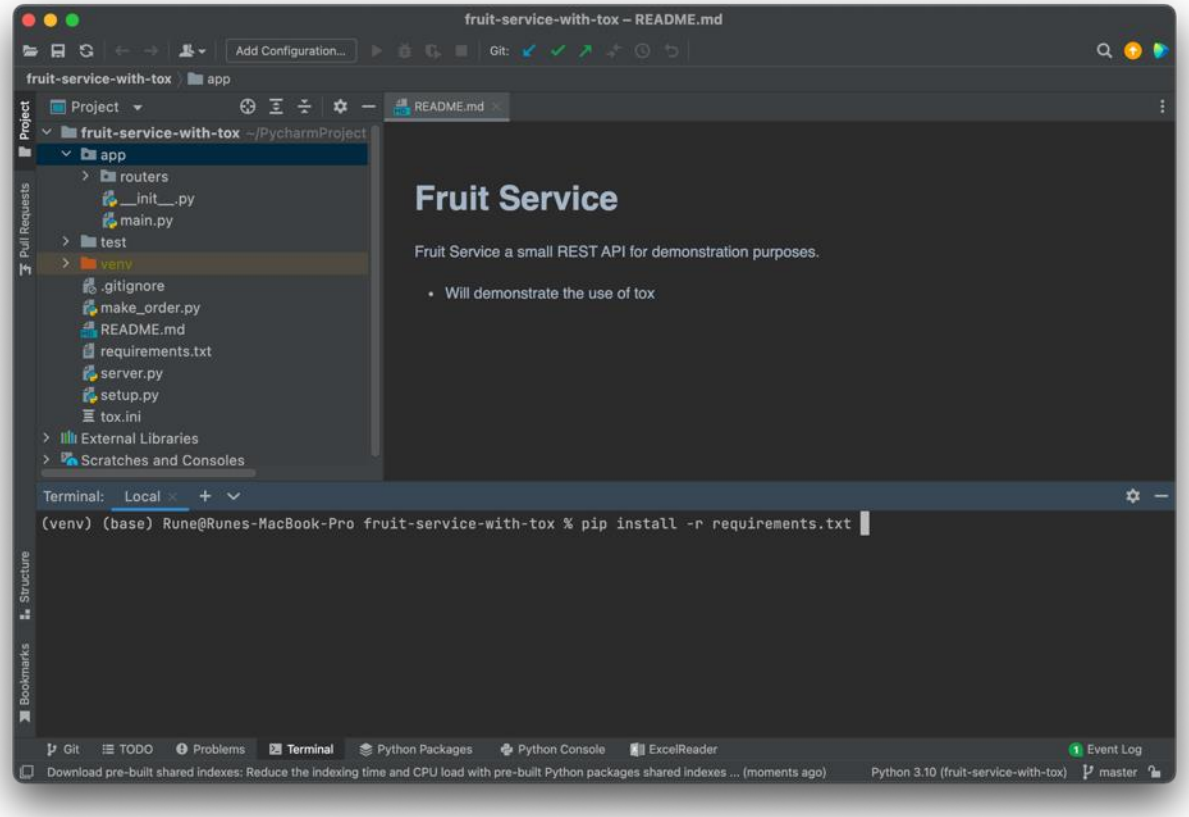


  - You might have different Python location – but it should show 3.10 after OK as seen below.



- Then press OK to setup environment

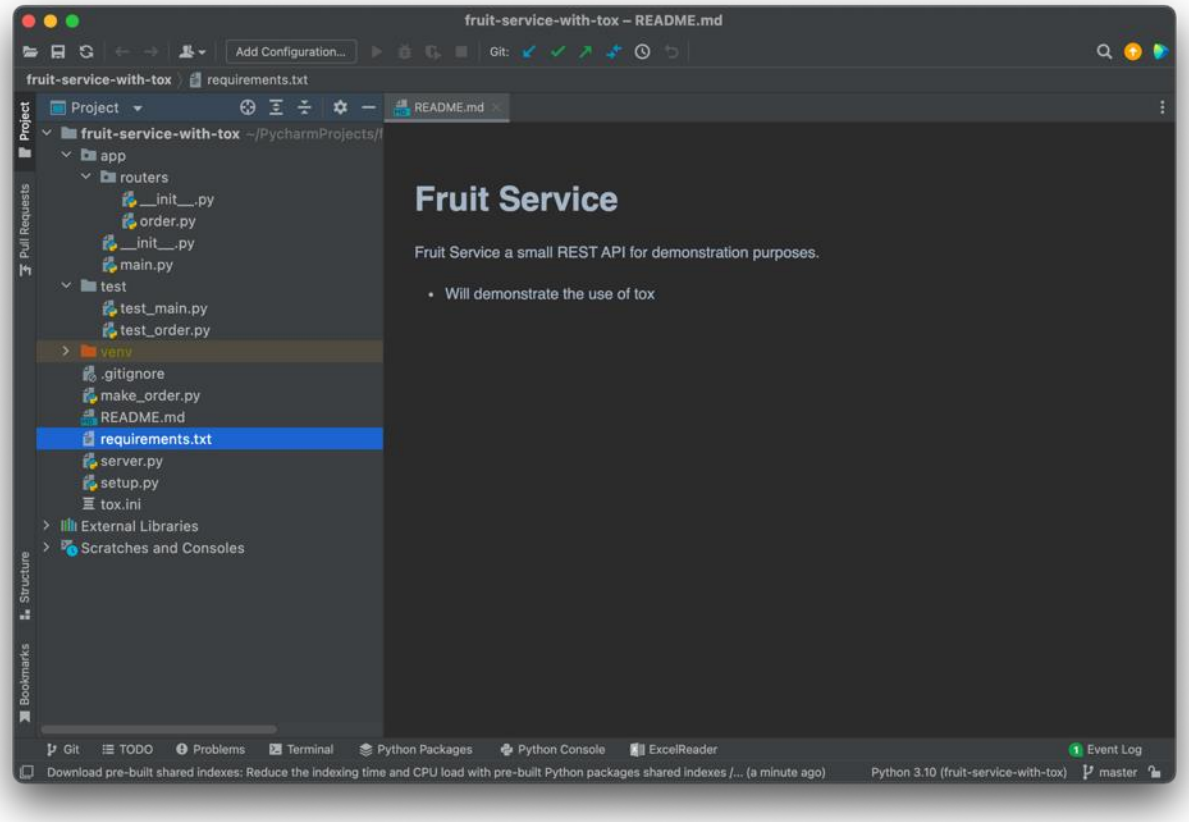
- Install **requirements.txt**



Now you have cloned the project and setup the environment.

Let's explore the difference of the landscape

Look at the files.



We see the following additions.

- **app/routers/__init__.py**
- **app/ __init__.py**
- **test/test_main.py**
- **test/test_order.py**
- **setup.py**
- **tox.ini**

__init__.py

The **__init__.py** files makes the folders to packages. That is, we can use them correctly as packages in our imports. This makes things easier for us, as we can treat our project as a module we can import.

They are empty and do not contain any functionality.

Read more about them in **Python docs**.

test/test_main.py and test/test_order.py

These files are testing files and are part of the tests we will make.

Notice, this is not a book on testing, which requires a full book by itself. But we have them here for demonstration purposes.

setup.py

This makes it a package you can install.

Pip ("pip install -e .") will use setup.py to install this module.

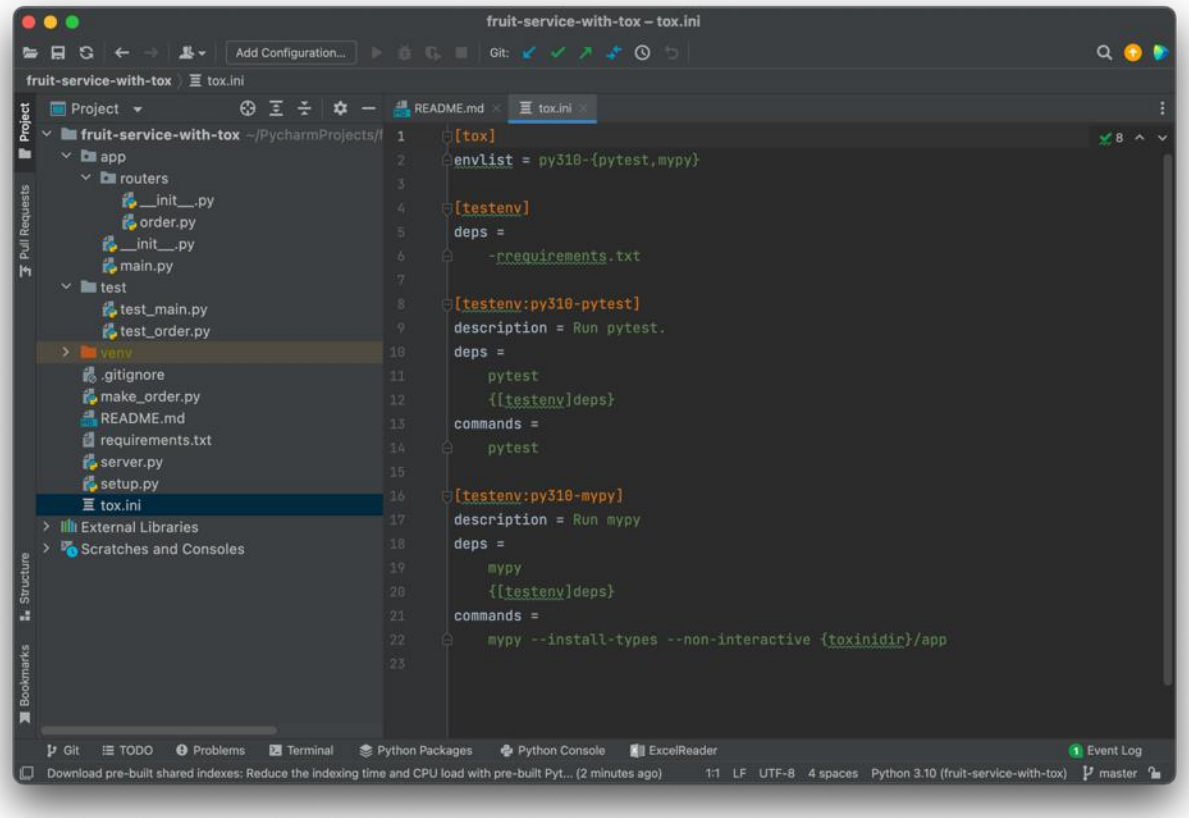
See **Python docs** for more details.

tox.ini

This is what we are looking for.

Let's explore the tox.ini file

When we run **tox** (which we will), it will use the **tox.ini** file to figure out what to do.



This is a simple **tox.ini** file.

On a high level it creates a test environment (as we talked about) and run tests.

The **tox.ini** is made quite simple, but still a bit more complex than most examples with only one environment part. This file has 4 sections.

- **[tox]** With a list of environments. Here we use the syntax **py310-{pytest,mypy}**, which is short for **py310-pytest, py310-mypy**. This tells tox to run tests in these two environments. The **py310** part is saying it should be Python 3.10.
- **[testenv]** This part has some general setup for the environments to be created. Here we have just some dependencies (**deps**), which will be installed with **pip** (**pip install -rrequirements.txt**).
- **[testenv:py310-pytest]** This is the first test virtual environment. It has dependencies **pytest** and the ones defined in **testenv** (**{[testenv]deps}**). It will run the command **pytest** in this virtual environment.
- **[testenv:py310-mypy]** This is the second virtual environment and is quite similar. It installs **mypy** and runs a **mypy** command.

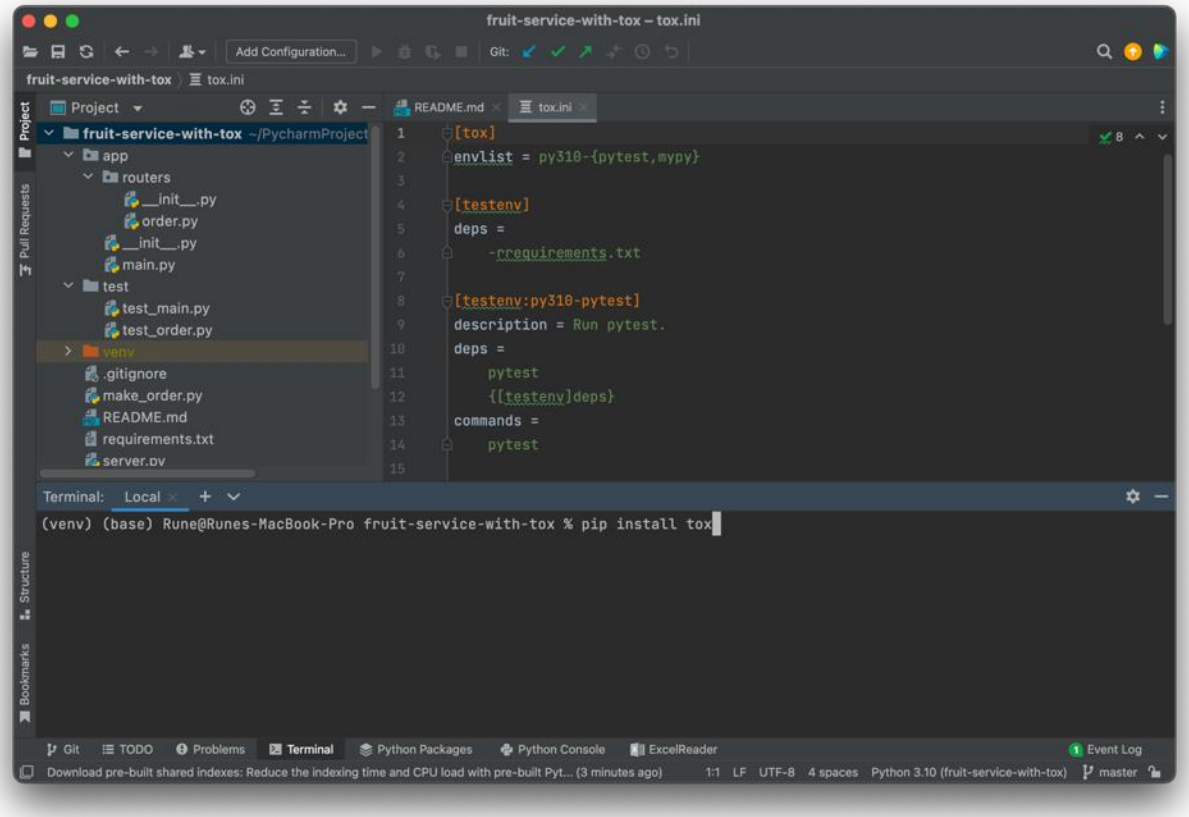
I think **tox** can seem a bit more complex than it actually is. In this chapter we will first learn how to use it and how to make some modifications and adding more test cases.

We will explore both and also which other tests could be done.

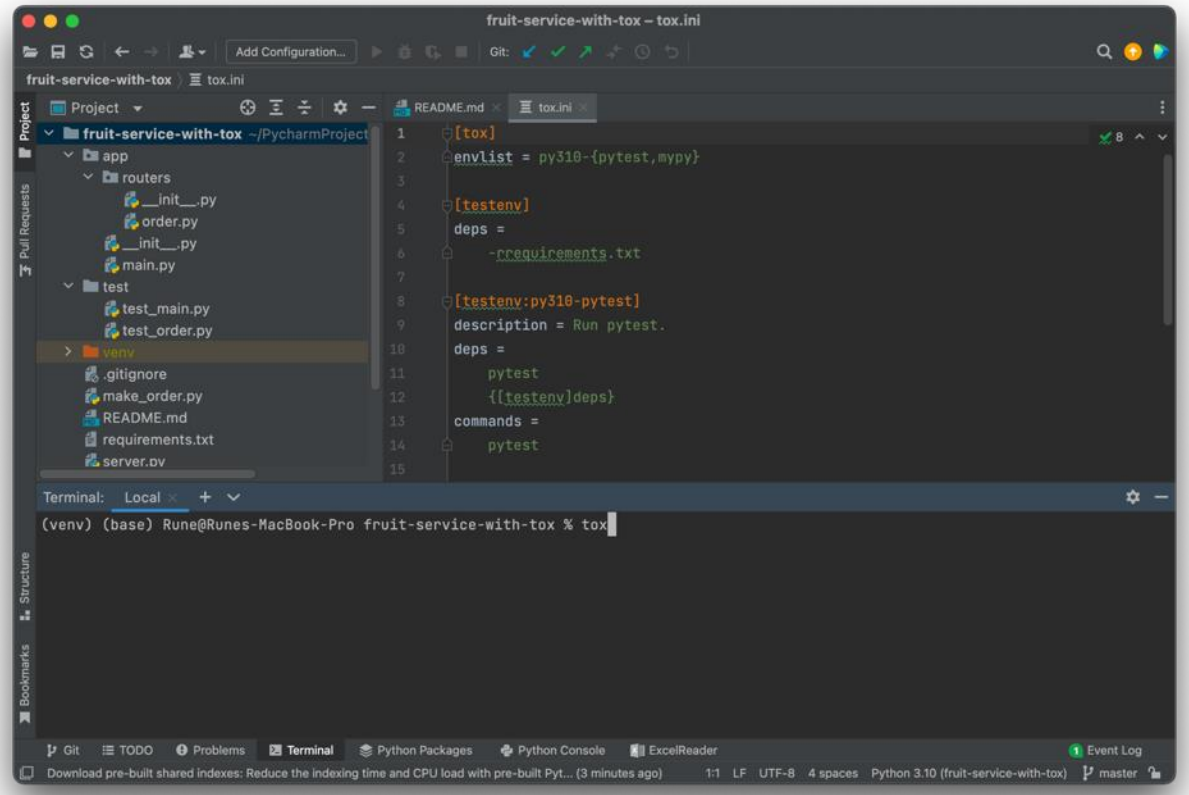
Let's run tox and see what happens?

You might need to install tox.

This is done by pip install tox.



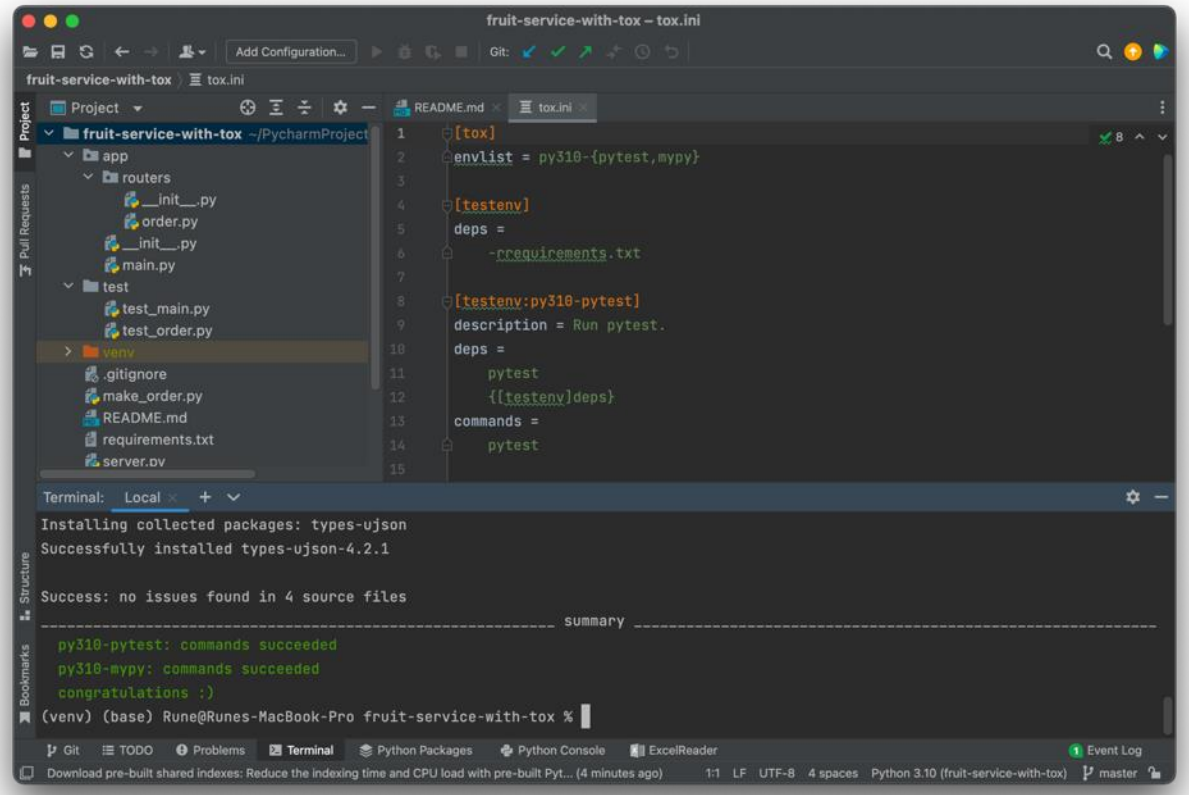
Then you can run **tox** by typing **tox**.



Then it will run a bunch of things and it can take some seconds to finish.

It will create a new wrapper virtual environment, install the requirements using the correct Python version. Then run the tests from **pytest** and **mypy**.

It should eventually end with something similar to this.



It should succeed.

Congratulations.

Thanks.

But let's try to break stuff and see what happens to learn how this works.

Wait a minute – did you notice?

This run created the following folders.

- **.mypy_cache** A folder created by **mypy**
- **.tox** A folder created by **tox**, containing the virtual environments.
- **fruit_service.egg-info** Which is the package (module) of our Fruit Service.

You do not need to worry about the content of these folders.

What does pytest do?

First of all, we will not become test masters and there are many other test frameworks. They all work in a similar manner with some differences (of course). **pytest** is one very commonly used, so knowing the basics will get you a long way.

Do you need to install pytest?

That is actually what **tox** does in the environment where it tests **pytest**.

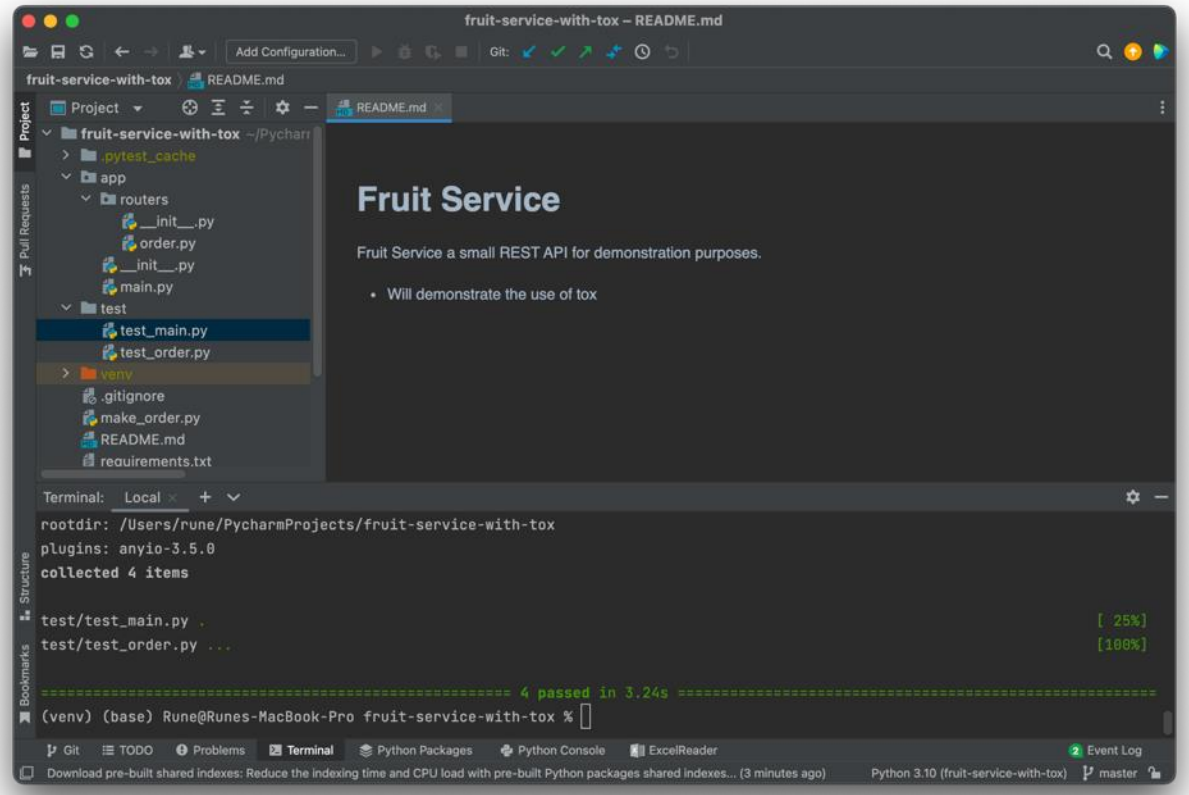
```
[testenv:py310-pytest]
description = Run pytest.
deps =
    pytest
    {[testenv]deps}
commands = pytest
```

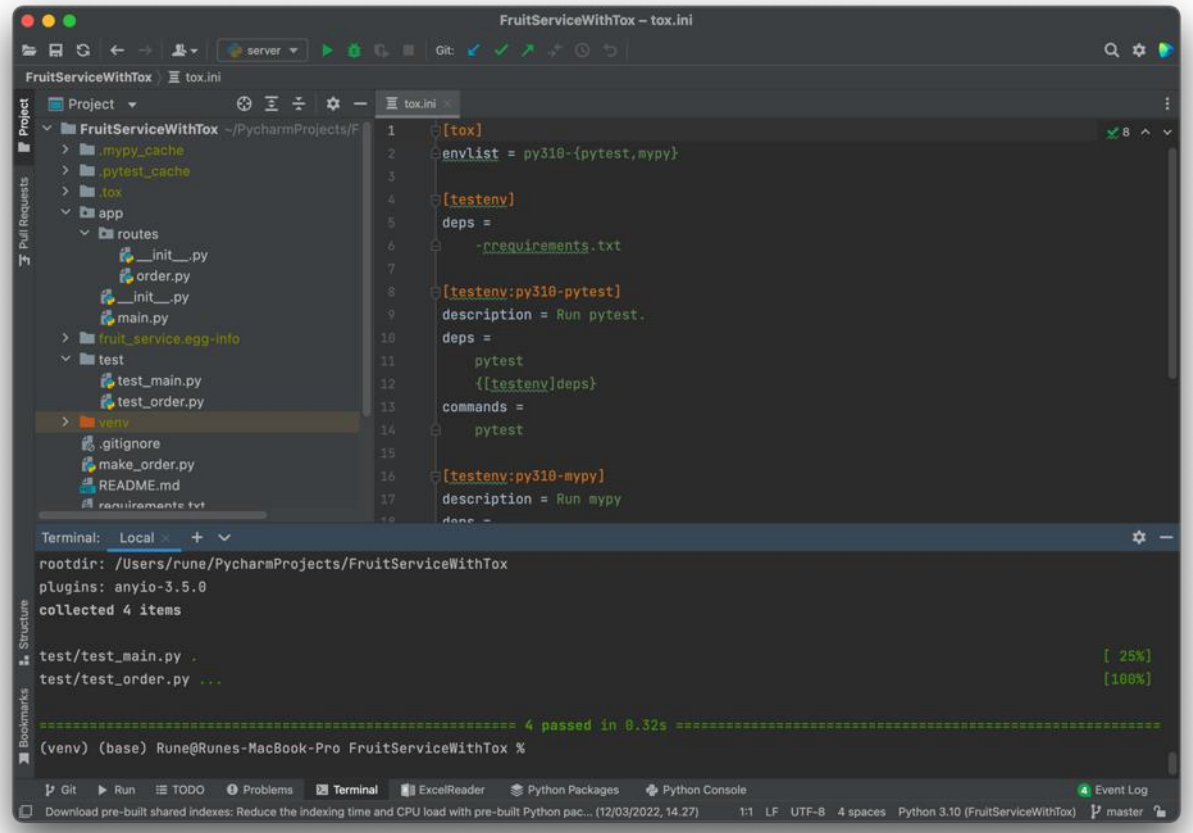
You see, it has a dependency on **pytest**.

If you want to you can install it in your environment as follows (but this is not needed):

```
pip install pytest
```

To run **pytest**, it simply writes executes **python -m pytest** in the environment (commands).





To summarize.

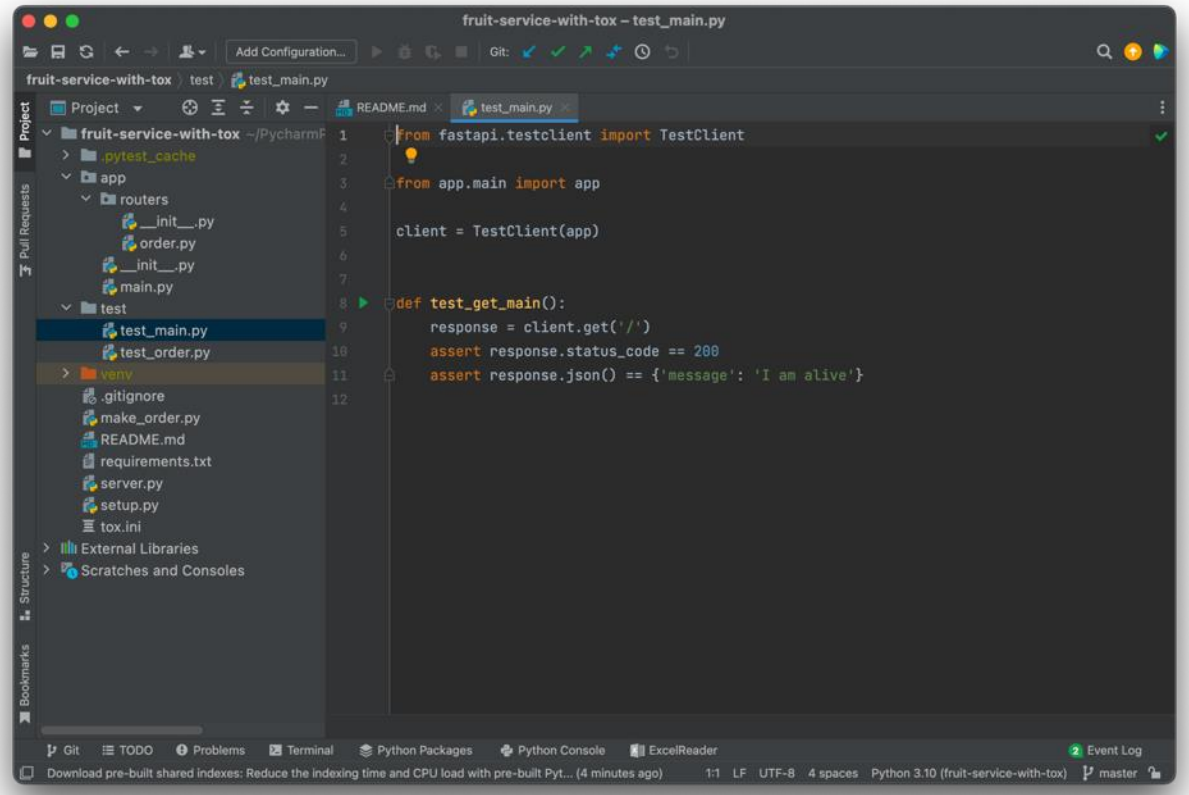
- **python -m pytest** runs the test files in folder **test**.

Exploring the test files

As already mentioned – we will only learn the world of unit testing as part of the setup. We will not dive into making great tests.

The scope is not to master testing (or unit testing), it is a big subject. The purpose is to learn all the frameworks you need to understand as a Python developer.

Now let's explore the first test file **test_main.py.**



You might already have guessed that we have structed the tests as follows.

- **test_main.py** for testing the app/main.py file.
- **test_order.py** for testing the app/routers/order.py

The **pytest** framework will run the file and call all functions called **test_something()**, where **something** can be anything, as you see.

Before it initializes a **TestClient(app).** This is specific for **FastAPI** testing, which you can see in their official test **guidelines**.

```
client = TestClient(app)
```

Inside the first test (and only test function) **test_get_main()** it calls the default availability endpoint.

```
def test_get_main():
    response = client.get('/')
    assert response.status_code == 200
    assert response.json() == {'message': 'I am alive'}
```

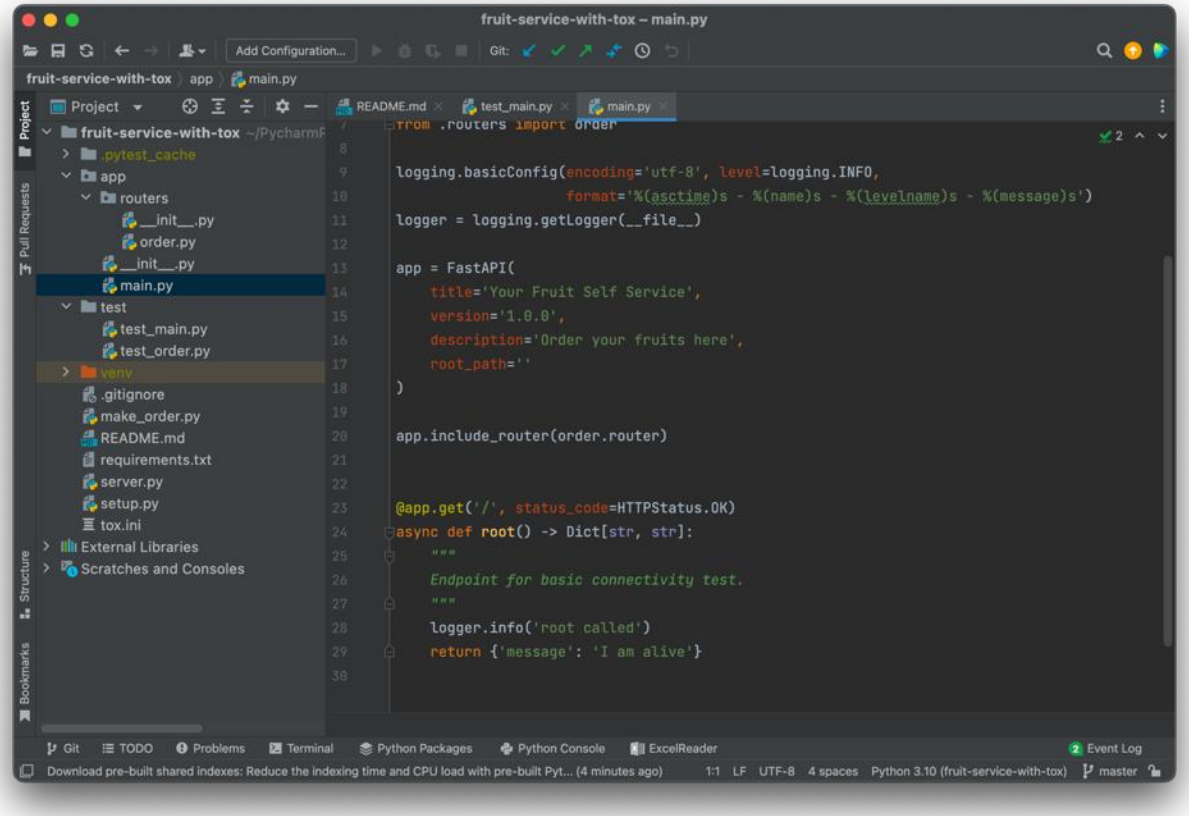
This is stored in the response.

Then there are two **assert** statements. These are the actual tests.

The expression after the **assert** is a Boolean expression. You should design your tests to evaluate to True, if things happen as expected and False if not.

Hence, if all **assert**s are evaluated to **True**, then the test passes. If one or more evaluate to **False**, then the tests fail.

Now we need to refresh our memory of what happens in the default endpoint in **main.py**.



We see (lines 23-29) that it will return **HTTPStatus.OK** (which is code 200) and the **json** data **{'message': 'I am alive'}**.

This is exactly what we assert for.

Now let's try to change something and see what happens.

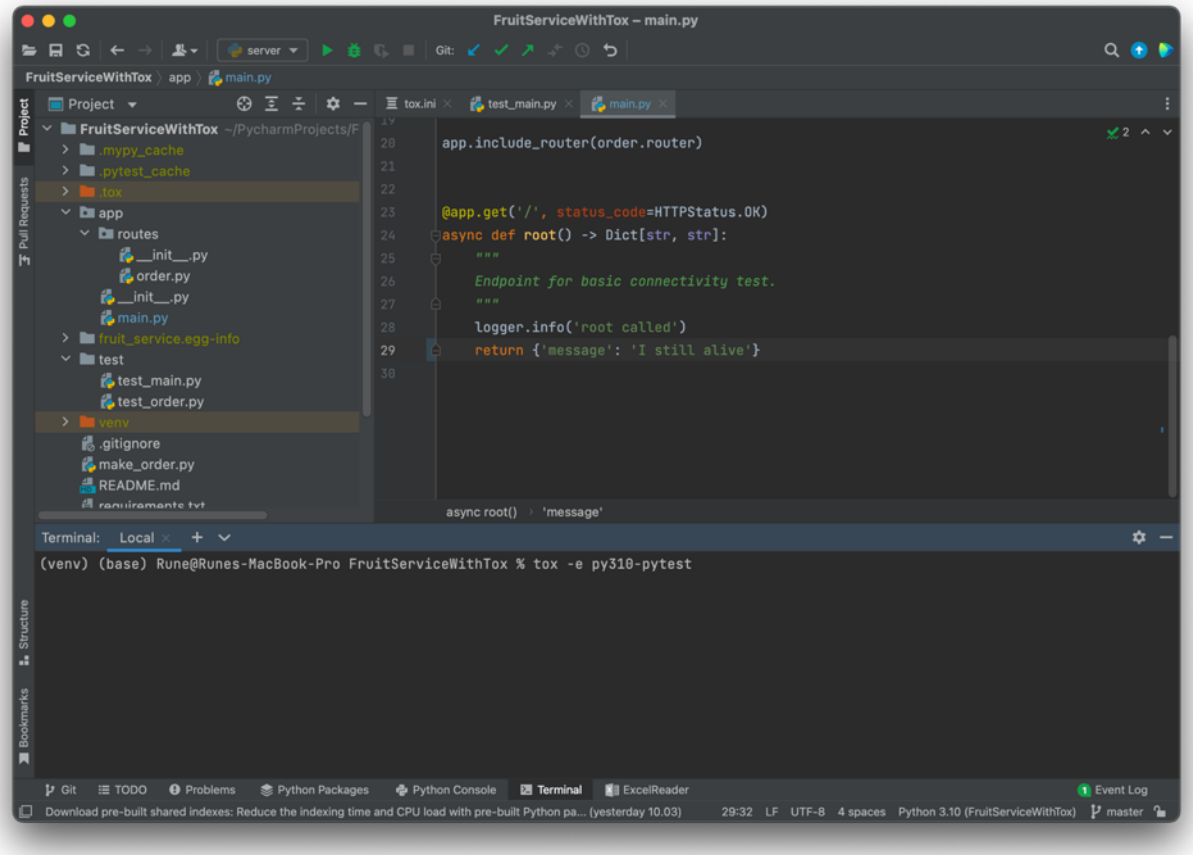
What happens if tests fail

Let's change the message to be '**I am still alive**' and re-run the test.

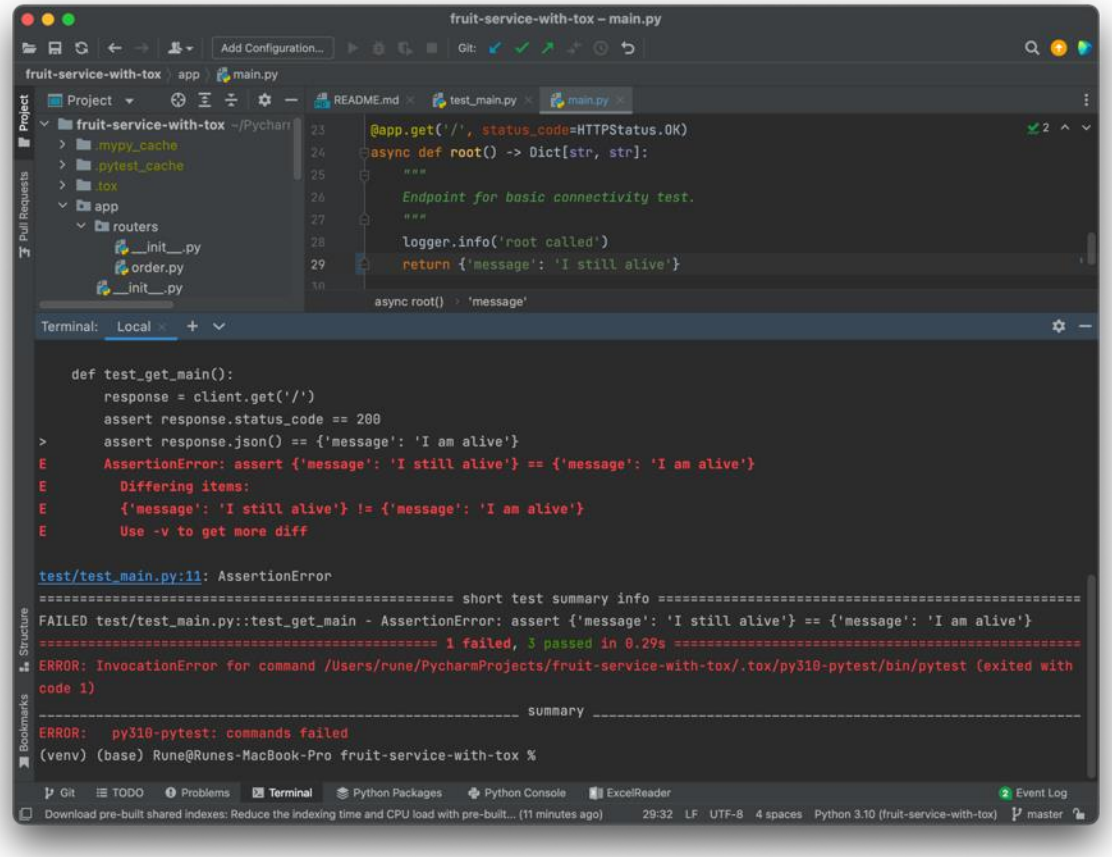
To only run the **pytest** part of **tox**, then you can type

```
tox -e py310-pytest
```

Remember that the **pytest** is called **py310-pytest** (we will learn more about that soon).



Hit enter and see what happens



The status is:

```
1 failed, 3 passed in 0.40s
```

We broke one test.

Actually, if we look a bit more on the output we see what happened.

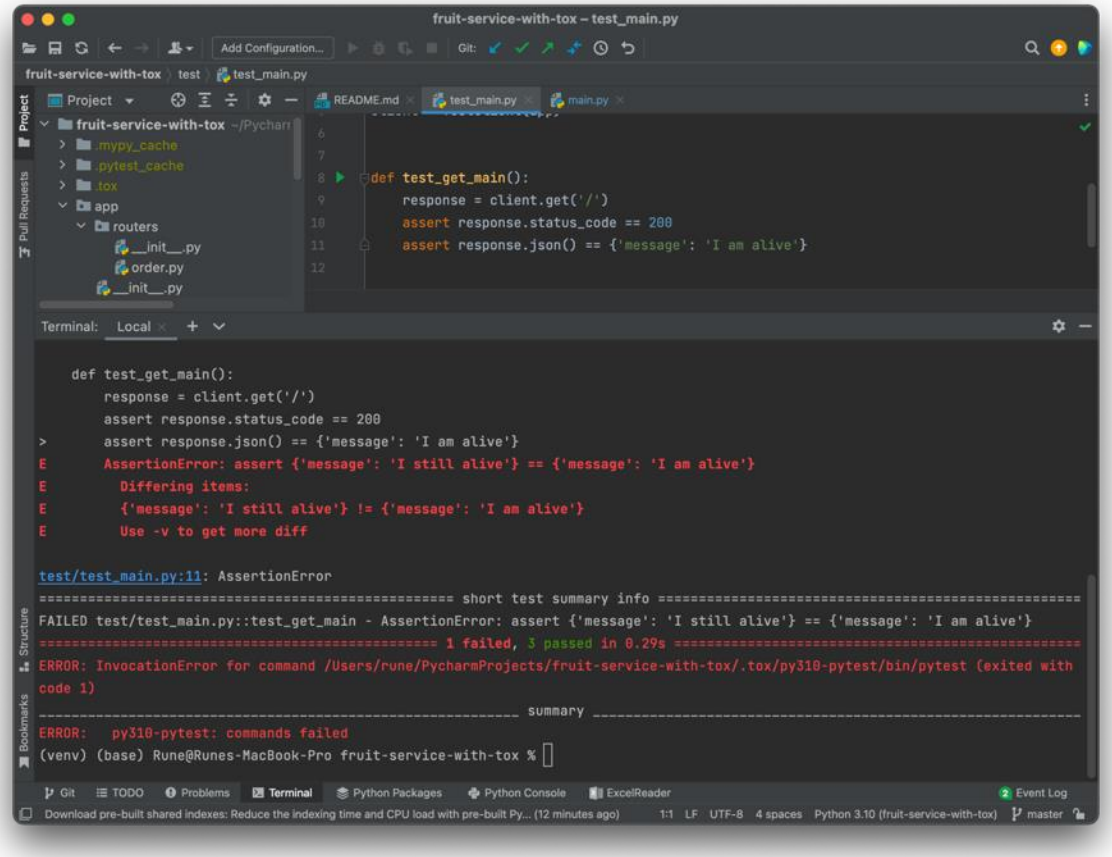
```
>       assert response.json() == {'message': 'I am alive'}
E       AssertionError: assert {'message': 'I still alive'} == {'message': 'I am
alive'}
E         Differing items:
E         {'message': 'I still alive'} != {'message': 'I am alive'}
E         Use -v to get more diff
```

This is amazing. It says which **assert** fails.

And where the assert is.

```
test/test_main.py:11: AssertionError
```

Clicking on that (in your PyCharm) will put your cursor on the failed assert.



After pressing it will take your courser to the failing assert.

The purpose of this test is to ensure it returns the correct code (HTTP status code OK: 200) and the expected message – which is formatted in **json**.

You can see a list of HTTP status codes on **Wikipedia**.

As already mentioned, the purpose of this endpoint is to check if service is running. The actual message is not important and could be different.

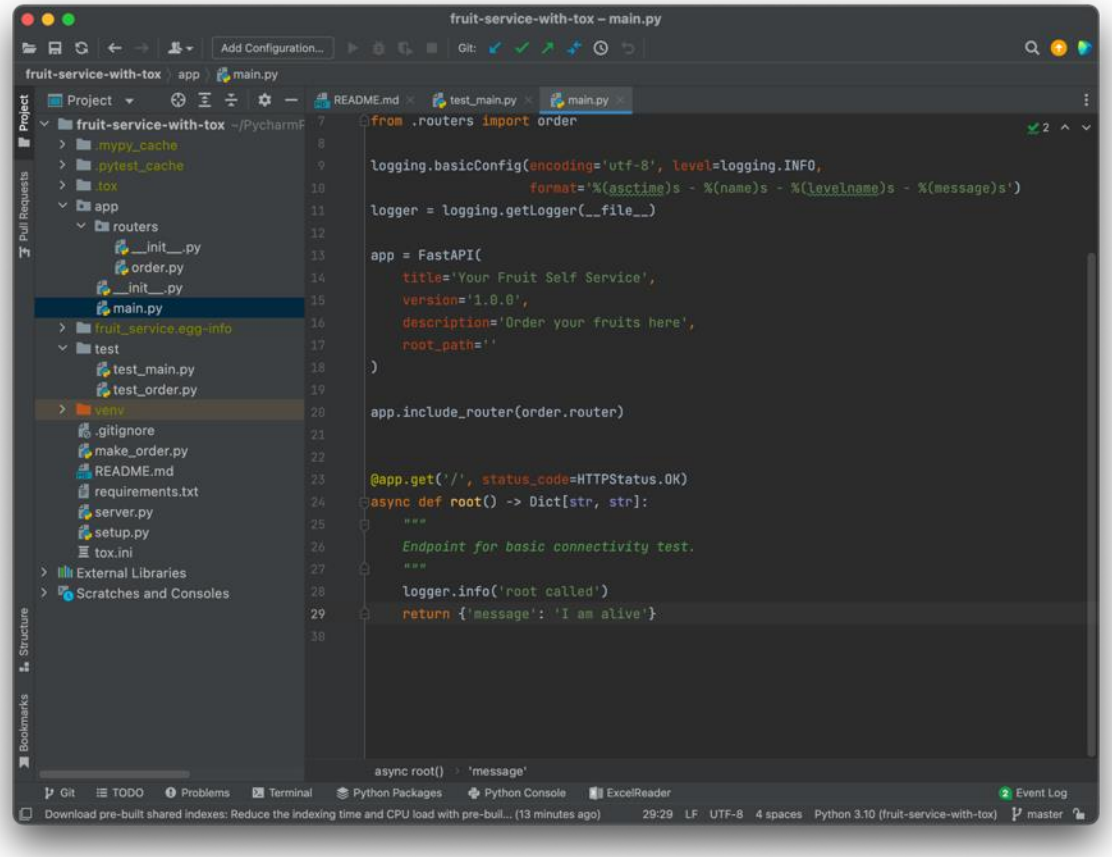
Why test the response message?

Often you will have a monitoring system to check if all the services are running. This service can use the message to check the message. Hence, if someone changed the message this would make the services seem unavailable. Therefore, a simple test like this makes sense to have.

Said differently, the test ensures we do not publish breaking changes to the ecosystem our service lives in.

Let's change it back.

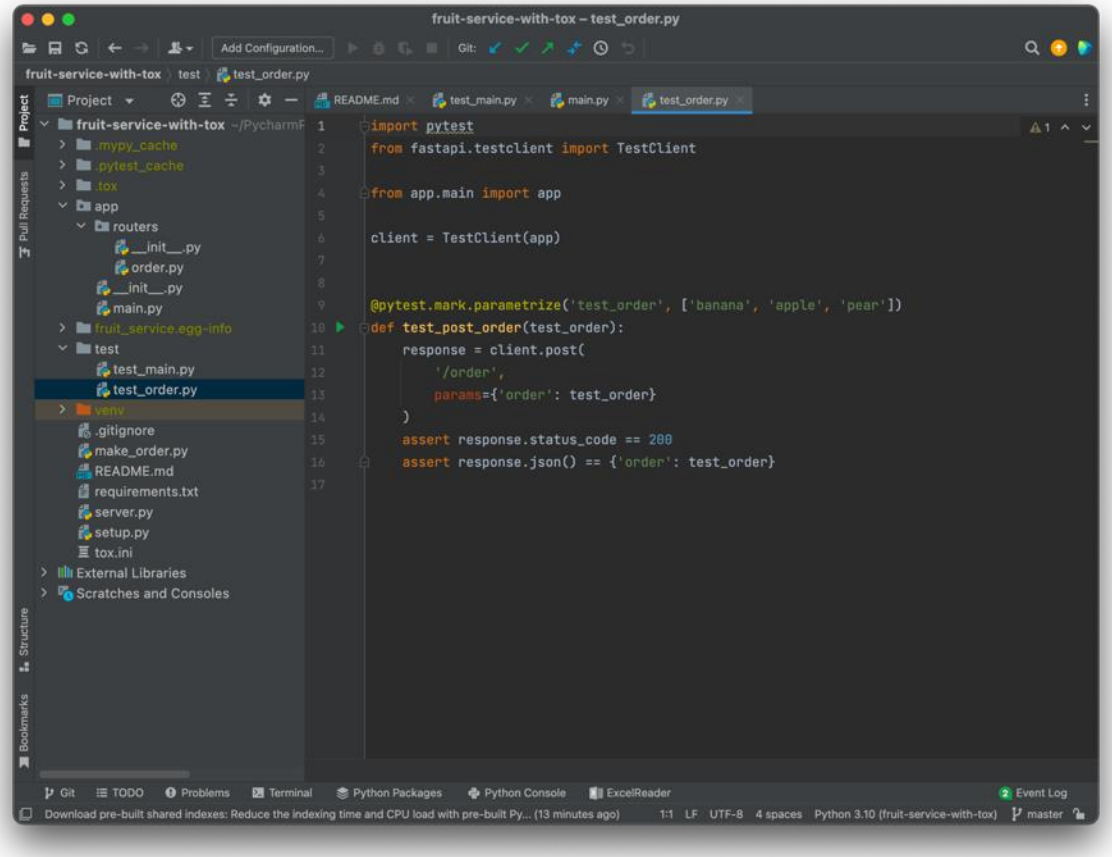
To do that we need to change the **app/main.py** file to be as follows.



Then run the test again (tox -e py310-pytest) and it should pass.

Explore test_order.py file

The test file **test/test_order.py** is a bit more involved.



We see that the test function (**test_post_order(test_order)**) takes an argument. The arguments are given by the decorator on line 9:

```
@pytest.mark.parametrize('test_order', ['banana', 'apple', 'pear'])
```

What is a decorator?

A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

Here this decorator makes calls **test_post_order** with argument **test_order** 3 times:

- **test_order_order(test_order='banan')**
- **test_order_order(test_order='apple')**
- **test_order_order(test_order='pear')**

This reduces the code you need to write. Without this decorator you would need to make three test functions, one for each order (as you could not take the argument).

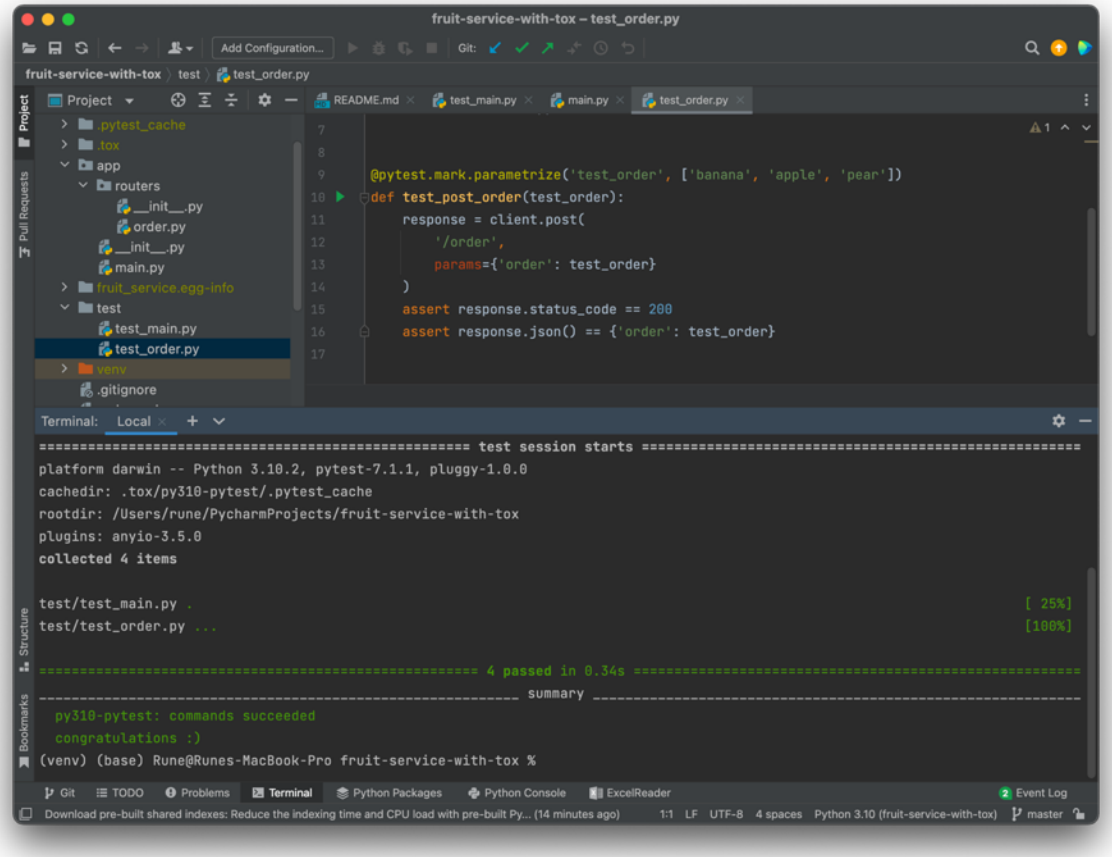
If it is confusing, just think that it calls your test function 3 times.

Could you make more calls?

Yes, just extend the list.

Remember in the output of tox pytest?

It writes 4 tests passed.



And if we look closely.

```
test/test_main.py .                                                      [ 25%]
test/test_order.py ...                                                   [100%]
```

This is one dot (.) after the **test_main.py** – which only does one test.

There are three dots (.) after **test_order.py** – which has the 3 tests.

Also, it says 25% (1 out of 4 tests is 25%) after **test_main.py** and 100% (4 out of 4 tests is 100%) after **test_order.py**.

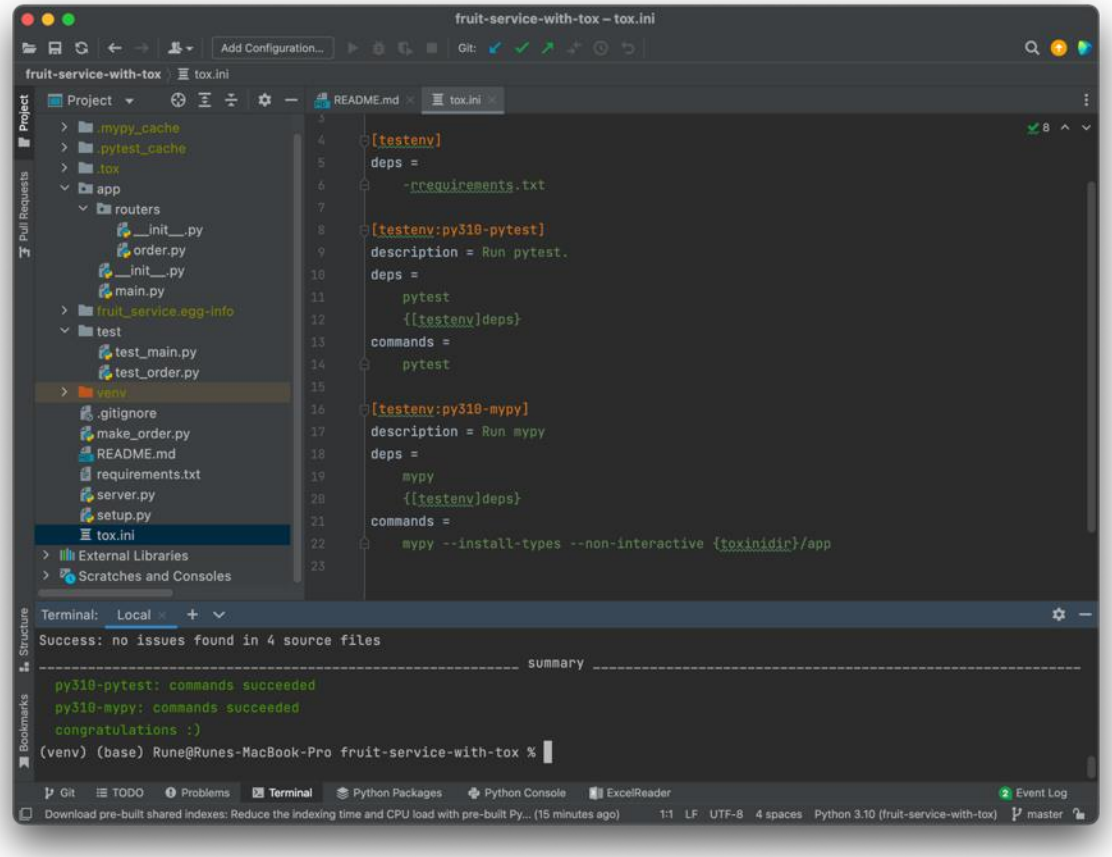
If you want to learn more about unit testing with **pytest** there is a 400+ page pdf documentation guide on their official page: **pytest Documentation**.

On their official page they have quick guides and how-to guides: **See here**.

For testing **FastAPI** see their official testing guide: **FastAPI testing**.

What does mypy do?

What did this mypy do?



Well, it succeeded.

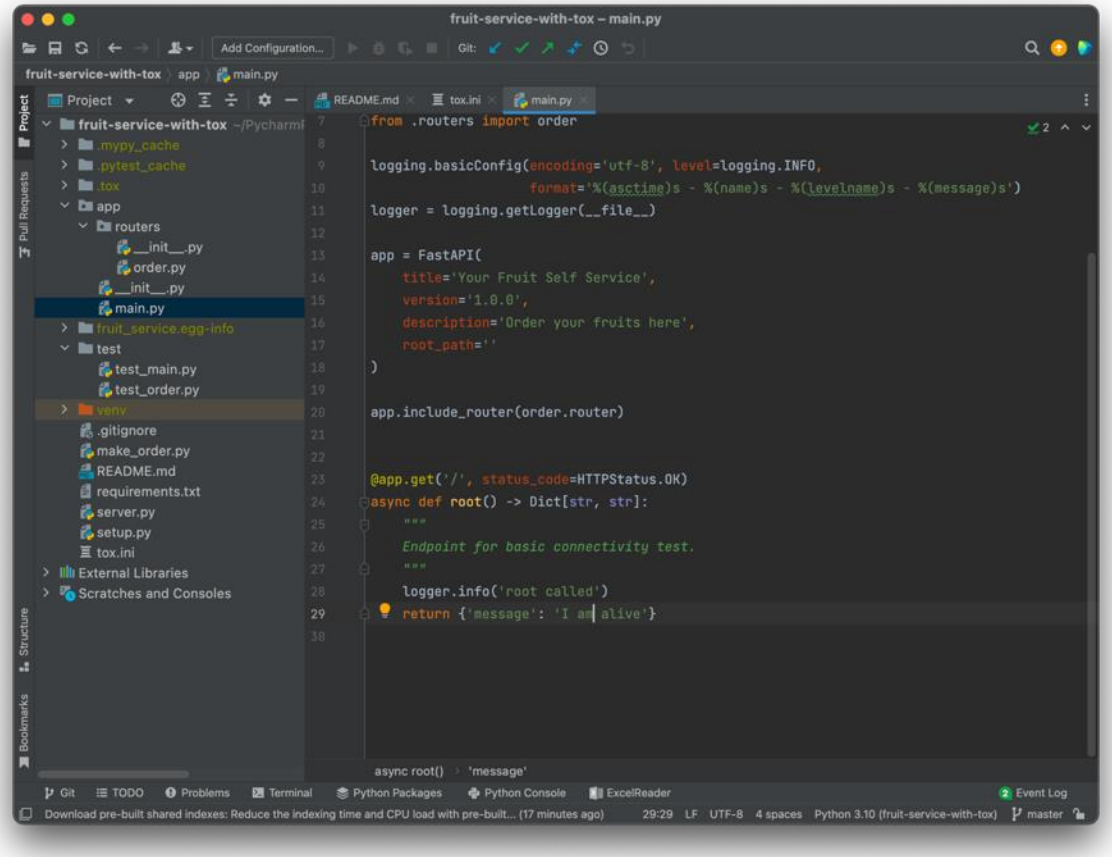
And on the official documentation it states (**reference**).

"Mypy is a static type checker for Python 3 and Python 2.7"

Why would you use it (**source**)?

- Static typing can make programs easier to understand and maintain. Type declarations can serve as machine-checked documentation. This is important as code is typically read much more often than modified, and this is especially important for large and complex programs.
- Static typing can help you find bugs earlier and with less testing and debugging. Especially in large and complex projects this can be a major time-saver.
- Static typing can help you find difficult-to-find bugs before your code goes into production. This can improve reliability and reduce the number of security issues.
- Static typing makes it practical to build very useful development tools that can improve programming productivity or software quality, including IDEs with precise and reliable code completion, static analysis tools, etc.
- You can get the benefits of both dynamic and static typing in a single language. Dynamic typing can be perfect for a small project or for writing the UI of your program, for example. As your program grows, you can adapt tricky application logic to static typing to help maintenance.

Enough talking – how does it look like.



As you see, we changed a few things in the root function.

```
async def root() -> Dict[str, str]:
    """
    Endpoint for basic connectivity test.
    """
    logger.info('root called')
    return {'message': 'I am alive'}
```

It is the **-> Dict[str, str]** part.

What is tells the type checker is that the function **root()** should return a dictionary with string to string key-value pairs.

Why is that important?

Because now you can make static type checks.

Let's make a simple example.

Assume someone is calling this function from somewhere else. This programmer knows that the function returns a dictionary with key-values of type string-string.

Therefore, he feels safe to assume that.

Now you are told to make some changes in the function and you end up with the following.

```
async def root() -> Dict[str, str]:
    """
```

```
    Endpoint for basic connectivity test.
    """
    if random.uniform(0, 1) < 0.05:
        return None
    logger.info('root called')
    return {'message': 'I am alive'}
```

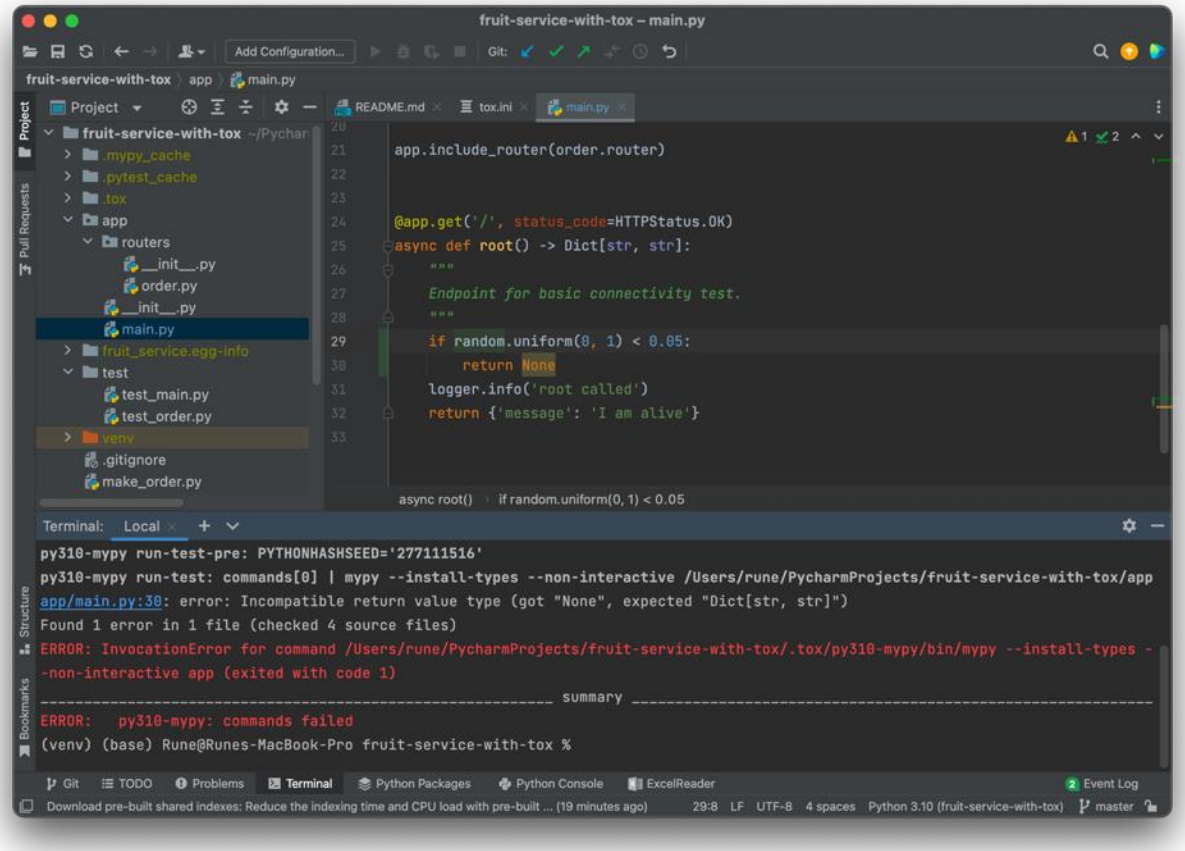
(notice you need to **import random** in the top).

Now the function might return None, which is not the type expected.

This will have consequences of the code of your fellow programmer. His code will fail whenever your function returns **None**.

That is one of the pain points with dynamic typing.

Luckily mypy will catch that (run **tox -e py310-mypy** in terminal).

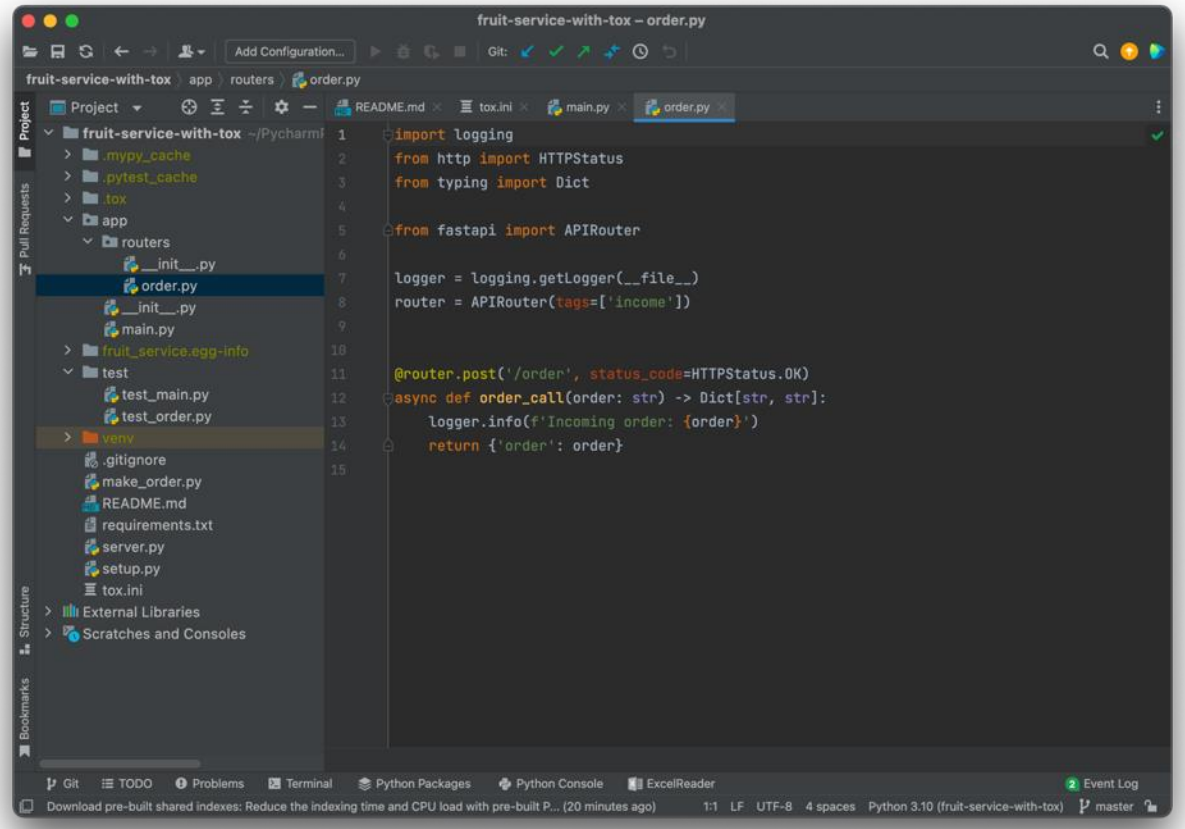


It says.

```
app/main.py:30: error: Incompatible return value type (got "None", expected
"Dict[str, str]")
```

It got “None” and expected “Dict[str, str]”.

In order.py we have and extra check.




```
async def order_call(order: str) -> Dict[str, str]:
    logger.info(f'Incoming order: {order}')
    return {'order': order}
```

The argument has type **str** (**order: str**). This ensures that the caller needs to provide the argument **order** of type **str**.

It takes a bit practice to understand it fully and for some types you need import them, like the Dict.

```
from typing import Dict
```

This was also done in **main.py**.

There is 250+ pages documentation of mypy: **mypy docs**

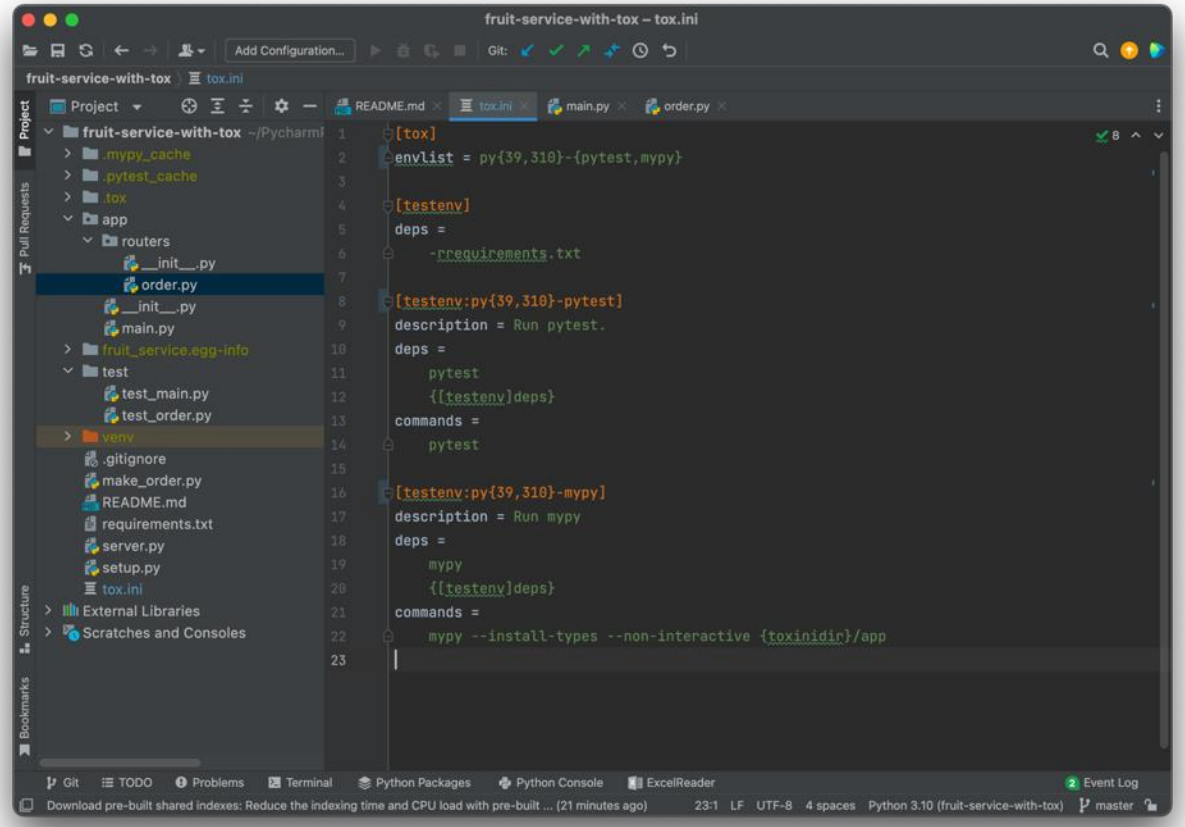
My advice is.

- Keep it simple – you will learn along the way. Start with what you understand.
- Define types at variable declarations: **a: int = 20**
- Define types of arguments to functions (see example above).
- Define types of functions (see example above).

Testing against more Python versions

Right now, our **tox** is setup to test against only one Python version, 3.10.

If you want, you could try against many different versions. Let's first try to add one more version.



As you see, we use this notation.

```
envlist = py{39,310}-{pytest,mypy}
```

This will create a list.

py39-pytest, py39-mypy, py310-pytest, py310-mypy

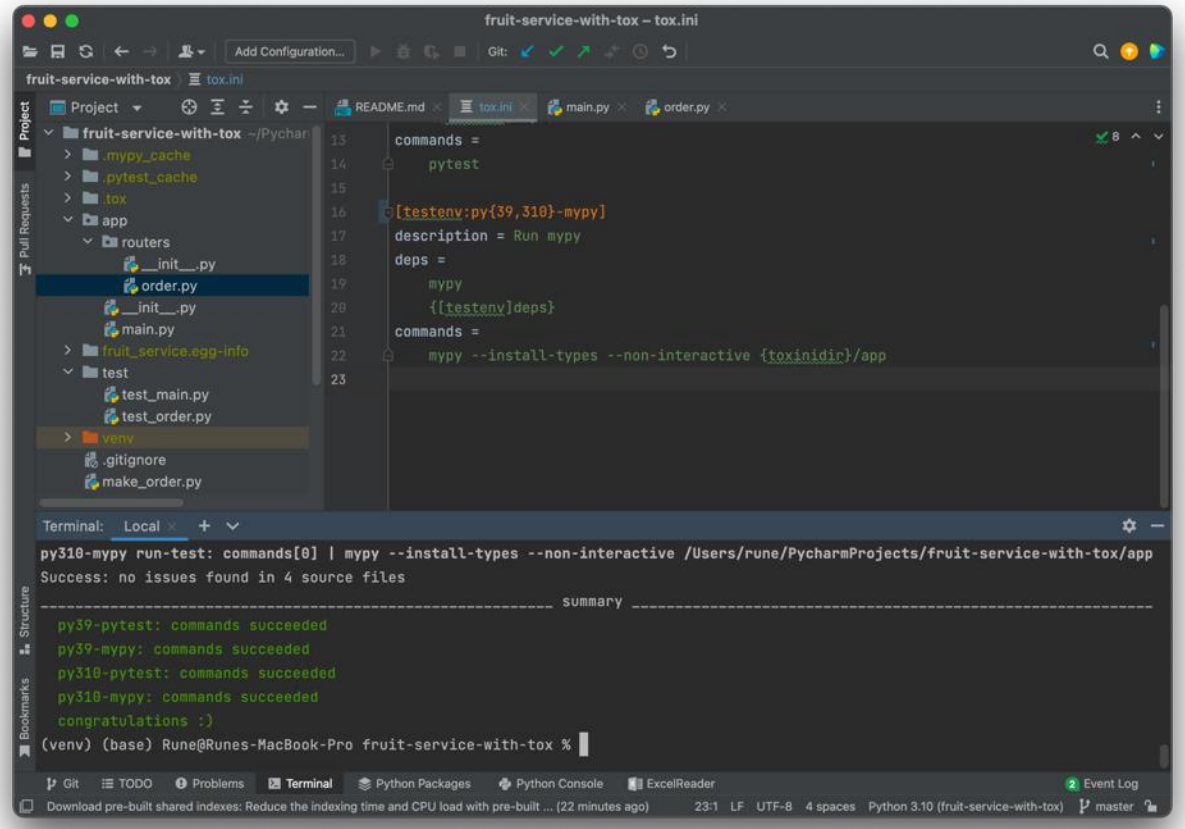
Also, they need a test. We do not have to setup different things for the 2 Python versions we test. Therefore, they can be done for one rule each.

```
[testenv:py{39,310}-pytest]
```

And as follows.

```
[testenv:py{39,310}-mypy]
```

When you run **tox** in the terminal you will see it takes longer time and you end up with 4 tests.



Notice, that you need to have installed Python 3.9 for this to succeed.

Let's try something crazy.

Test it against Python 3.8 as well.

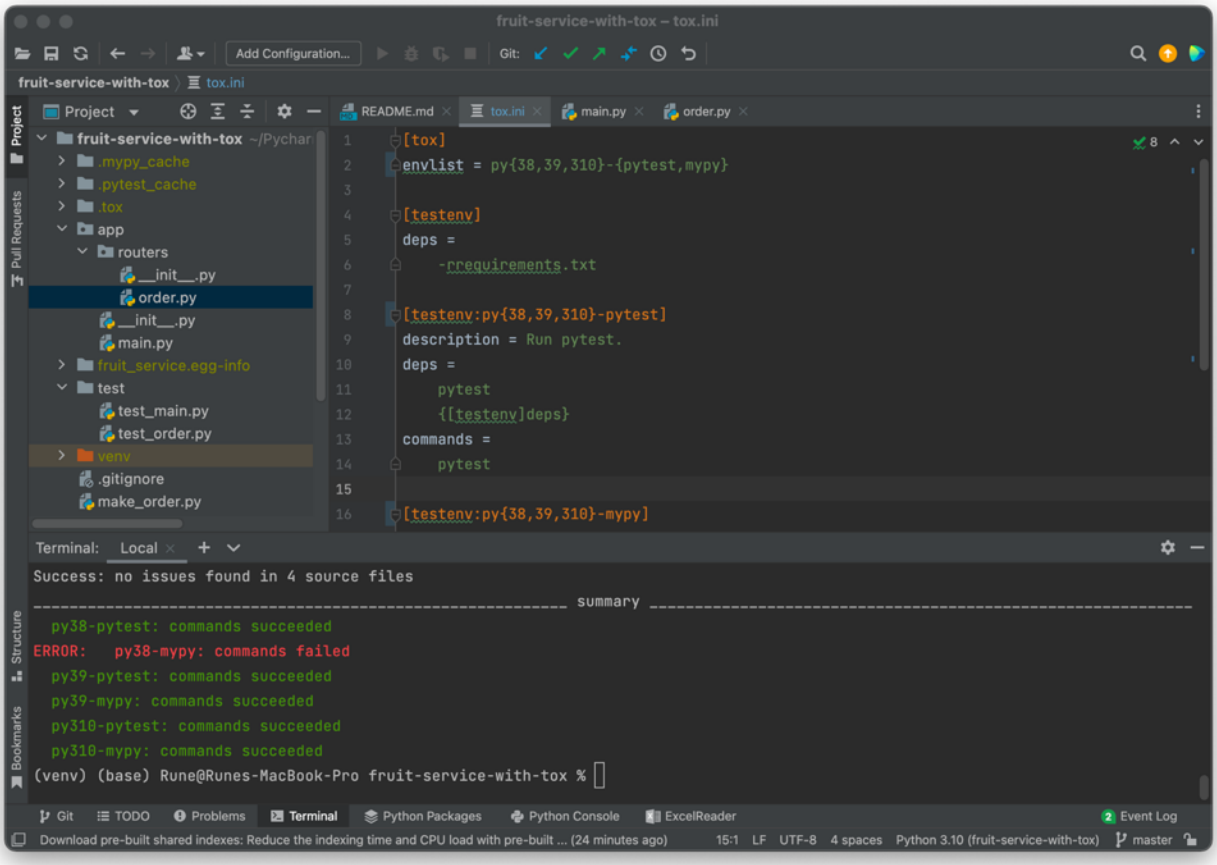
```
[tox]
envlist = py{38,39,310}-{pytest,mypy}

[testenv]
deps =
    -rrequirements.txt

[testenv:py{38,39,310}-pytest]
description = Run pytest.
deps =
    pytest
    {[testenv]deps}
commands =
    pytest

[testenv:py{38,39,310}-mypy]
description = Run mypy
deps =
    mypy
    {[testenv]deps}
commands =
    mypy --install-types --non-interactive {toxinidir}/app
```

Then run **tox**.

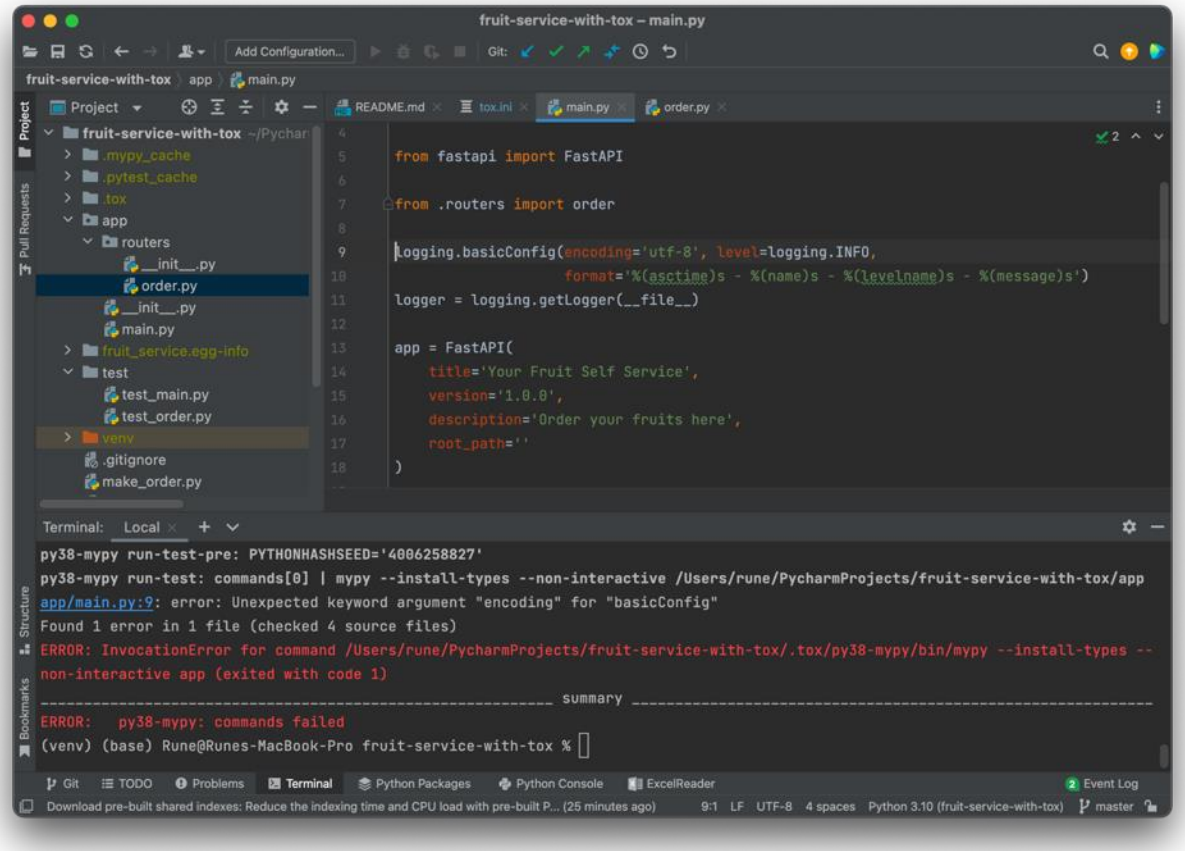


No! Mypy failed for Python version 3.8.

Why?

Let's try to figure it out.

To run that test you type: **tox -e py38-mypy**



Well, there we have it.

```
app/main.py:9: error: Unexpected keyword argument "encoding" for "basicConfig"
```

This might confuse you right?

Why is this not a problem for Python 3.9 or 3.10.

Let's check the docs (**see here**) with shortened view here:

`logging.`**`basicConfig`**`(`*`**kwargs`*`)`

> Does basic configuration for the logging system by creating a `StreamHandler` with a default `Formatter` and adding it to the root logger. The functions `debug()`, `info()`, `warning()`, `error()` and `critical()` will call `basicConfig()` automatically if no handlers are defined for the root logger.
>
> [...]
>
> The following keyword arguments are supported.

Format	Description
filename	Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.
Filemode	If *filename* is specified, open the file in this mode. Defaults to `'a'`.
format	Use the specified format string for the handler. Defaults to attributes `levelname`, `name` and `message` separated by colons.
Datefmt	Use the specified date/time format, as accepted by `time.strftime()`.
Style	If *format* is specified, use this style for the format string. One of `'%'`,`'{'` or `'$'` for printf-style, `str.format()` or `string.Template` respectively. Defaults to `'%'`.
Level	Set the root logger level to the specified level.
Stream	Use the specified stream to initialize the `StreamHandler`. Note that this argument is incompatible with *filename* – if both are present, a `ValueError` is raised.
Handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* – if both are present, a `ValueError` is raised.
Force	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.
Encoding	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file.
Errors	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value 'backslashreplace' is used. Note that if `None` is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing 'errors'.

Changed in version 3.2: The *style* argument was added.

Changed in version 3.3: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

Changed in version 3.8: The *force* argument was added.

Changed in version 3.9: **The *encoding* and *errors* arguments were added.**

The last line says that in version 3.9 the encoding argument was added.

That explains it. In Python version 3.8 there was no argument encoding.

Suddenly you realize this kind of testing is crucial. Because if you ship this code to run in an environment with Python earlier than 3.9, it will not work.

Luckily, we will also learn how to create Docker containers with a specific Python version. Hence, we can remove all these other Python version tests for our purpose here.

What else could we test with tox?

There are other things that can be common to test for with **tox**.

Here we have a list of common frameworks. We will not go through them, but they are provided with links to the official documentation pages. Most have a decent get started guide.

You should be able to add similar sections to your **tox.ini** file to create these tests if you like.

Here are some of the most common one. Remember, it might not be necessary to add them all. I have introduced you to the two most important ones in most use cases.

- **bandit**. Checks for common security issues
- **pylint**. Checks for errors, enforces coding standards, looks for code smells.
- **Flake8** Analyze and detect some errors.
- **pycodestyle**. Checks against some of the style conventions in PEP 8.
- **pydocstyle**. Checks compliance with Python docstring conventions.

Just to mention a few common ones.

Exercise

A great thing to try is to extend your project to use tox. This will require a few things.

- You need to have a requirement.txt file.
- Add a tox.ini file (use the same given in this chapter).
- Setup test folder in your project.
- Add test file to call your end-point (see example given in this chapter).
- Add typing to your functions.
- Install tox: pip install tox
- Run tox and fix what is not working.

Summary

In this chapter we covered essential practices to ensure your code runs as expected.

- How **tox** can help you test that your code will work in a new clean environment.

- Make simple tests of your code using **pytest** – this ensures you do not make breaking changes.
- How a testing framework is testing using **assert**.
- Finding out why a test fails – reading the output and fixing it.
- Using **mypy** for static typing.
- Why static typing makes your code more robust – as you make type declarations and avoid wrong usage and common mistakes.
- How to test multiple versions of Python running your code.
- Showing that there can be changes which makes your code fail on different versions of Python.

04 – Docker

In this chapter we will install Docker, create Docker containers, and deploy them locally.

- What is Docker?
- How to install Docker
- Create Dockerfile and deploy them

What is Docker?

Docker enables developers to easy deploy their applications in containers to run in a specific environment decided by you.

That might be a bit difficult to understand at first.

Take our Python API – we already know, that it does not run with Python 3.8. But we want to be able to deploy it different places without the trouble of having correct Python, libraries, environment variables etc. This is what Docker can do for us.

You can package Python 3.10 (in our case) with the correct libraries in a Linux OS.

What about this Linux, you might wonder. I don't run Linux.

Neither do I – but I can run it in my Docker Daemon.

This is the beauty of it, you can run it on any machine or server with a Docker Daemon.

This actually solves a big problem. Imagine you also had another API running, but it only worked with Python 3.8. Your new does not work with 3.8.

But each of these API's are running in each their package with each their environment.

Therefore, you can have it running isolated in their own containers (as they are called) – also, they can call each other endpoints (API's) and communicate with each other.

You suddenly understand the power of Docker.

But isn't Docker DevOps or SRE?

Yes and no.

The boundary between **DevOps** (or **SRE**) is blurry and often dependent on company culture.

What do **DevOps** do, you might wonder?

A **DevOps** goal is to ensure a rapid release of stable and secure software.

Often that is creating infrastructure for developers to deploy software with testing pipelines. But you see, it is the developer's responsibility to deploy it.

In the past you would have operational (or **Ops**) staff, which were responsible for stable software. The developers would built the software and give it to **Ops** – then **Ops** had responsibility of it. This

often made a clash, and developers were more focused on creating more functionality, rather than stable functionality. This created a siloed culture.

Therefore, a movement of giving the developers the responsibility of stable software was created with the **DevOps** movement, where they work closer together. The focus of **DevOps** is more about making it easy to make deployments, and developers got responsible for keeping it running in the infrastructure built by **DevOps**.

Building a **CI/CD** pipeline is what **DevOps** provide and rarely something you as a develop need to worry about.

What **CI/CD** gives you, is makes it possible to deploy your work to production, this includes testing of your work (like we did with **tox**), but also the monitoring aspect.

As a developer we don't build that piece, but we need an understanding of each element. We will build **Docker images**, run **Docker containers** on our own **Docker Daemon**.

Developers are responsible for having a stable running software, while **DevOps** are building the infrastructure to realize that.

The good news is – you do not need to be an expert with **Docker** or any of the other tools in the **CI/DC** pipeline.

I am by far any expert in that – I constantly ask **DevOps** questions how to do specifics. But I have and need a basic understanding of Docker – this is what we will learn here.

I also mentioned **SRE**.

There is an overlap of **DevOps** and **SRE** and it is also blurry. Some say, it is two approaches to delivering a product release cycle in collaboration with developers, using automation and monitoring to achieve it.

SRE can be more focused on safety, health, uptime, and helping to spot unforeseen problems arising.

I have worked with people identifying themselves as **DevOps**, others that emphasized they were not **DevOps** but **SRE**, and some that didn't care. In all truth, it has been difficult to exactly understand the difference.

I know there are a lot of articles on the internet pointing out the differences – but in reality, I am not the right one to explain it. It seems to cover the similar aspects seen from a developer.

Dockerfile, Docker image, Docker container?

There can seem to be a lot terminology with Docker.

The good news is, you only need to understand the high-level concepts of how it works – the precise concepts can ease the communication, but most people understand you if you call a Dockerfile the Docker image.

The terminology used is as follows.

- **Dockerfile**. The commands to create a Docker image.
- **Docker images**. The blueprints of the containers.

- **Docker container**. Created from Docker images and run the actual application – this is the running code.
- **Docker Daemon**. The background service running on the host that manages building, running and distributing Docker container.
- **Docker Client (CLI)**. The command line tool that allows users to interact with the daemon.
- **Docker Hub**. A registry of Docker images.
- **Docker Desktop**. An easy to install UI including Docker Daemon and Docker client.

I think we will learn by doing, but in short you need the following.

- Install Docker Desktop.
- Create Dockerfile.
- Compose a Docker image.
- Deploy the image to run as a container in Docker Daemon.
- Then know how to shut them down and restart.

Don't worry too much about understanding all this now. You will learn it in this chapter.

Installing Docker Desktop

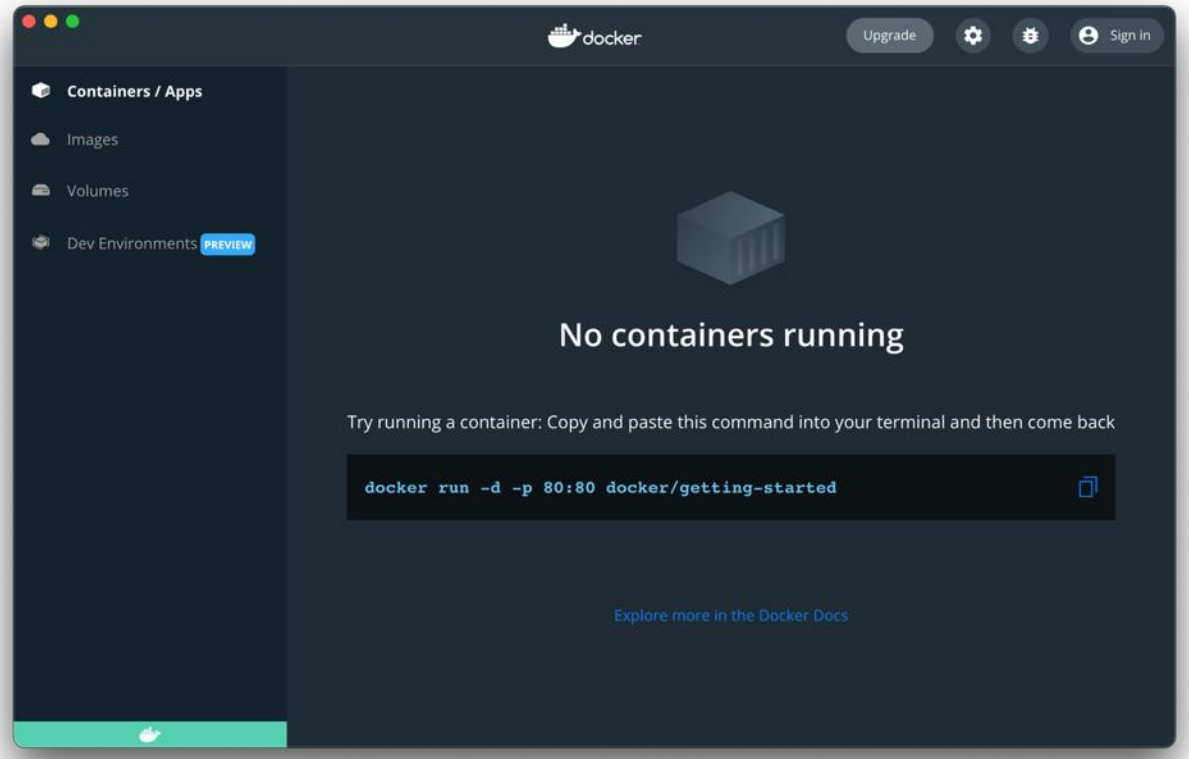
You can install Docker Desktop and it will automatically install all you need to manage all Docker tools need.

Go to Docker official download and get Docker Desktop for your OS (**here**).

Follow the installation instructions and you are ready to go.

- **Linux Notice**. Please notice that if you use Linux there is no Docker Desktop you will need to use command lines when we use the Desktop. For the most part we only use to see the containers and see what is running. This can be done from a command line as well.

You can launch Docker Desktop, which will start the Docker Daemon in the background.



Clone Project FruitServiceWithDocker in PyCharm

The goal of this section is to run our Fruit Service in Docker. That is, we will run it in a Docker container in our new installed Docker Daemon (installed with Docker Desktop).

Are you ready for that?

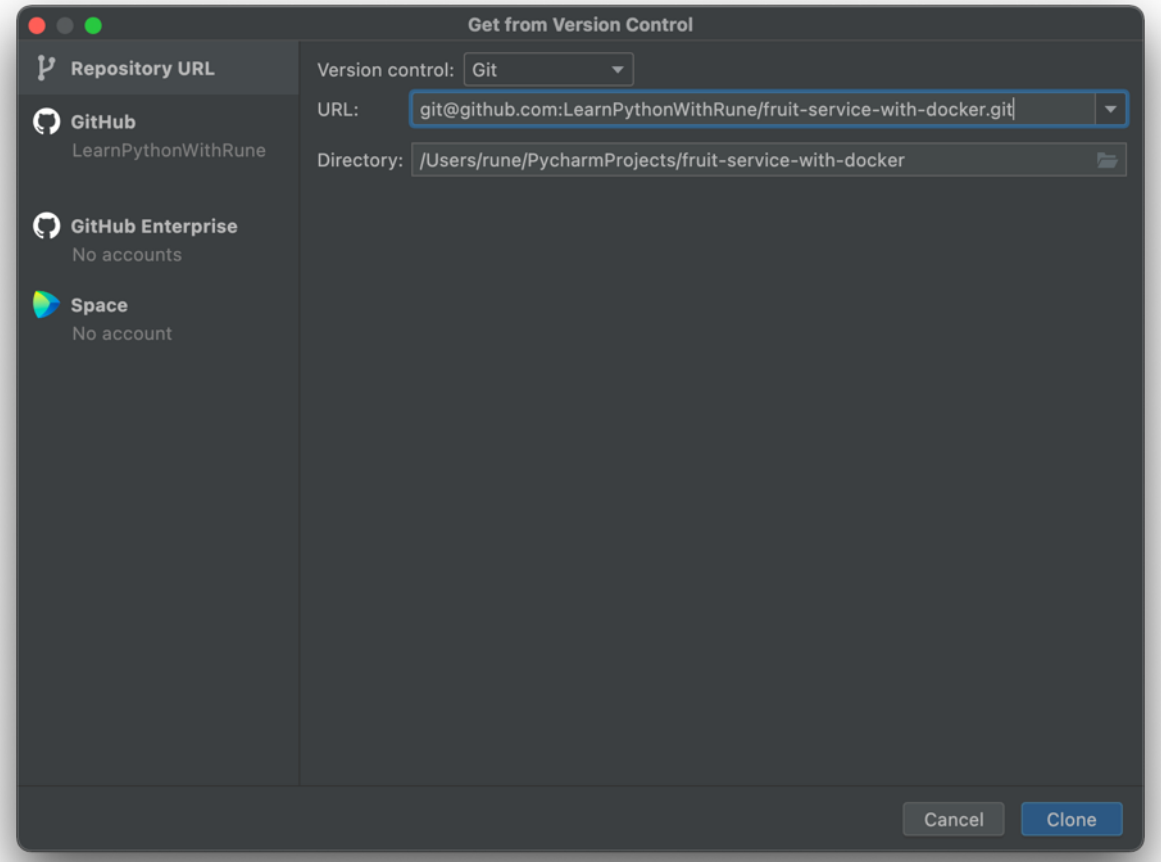
First, we need to clone the project in PyCharm.

You might wonder why we need to clone all these projects?

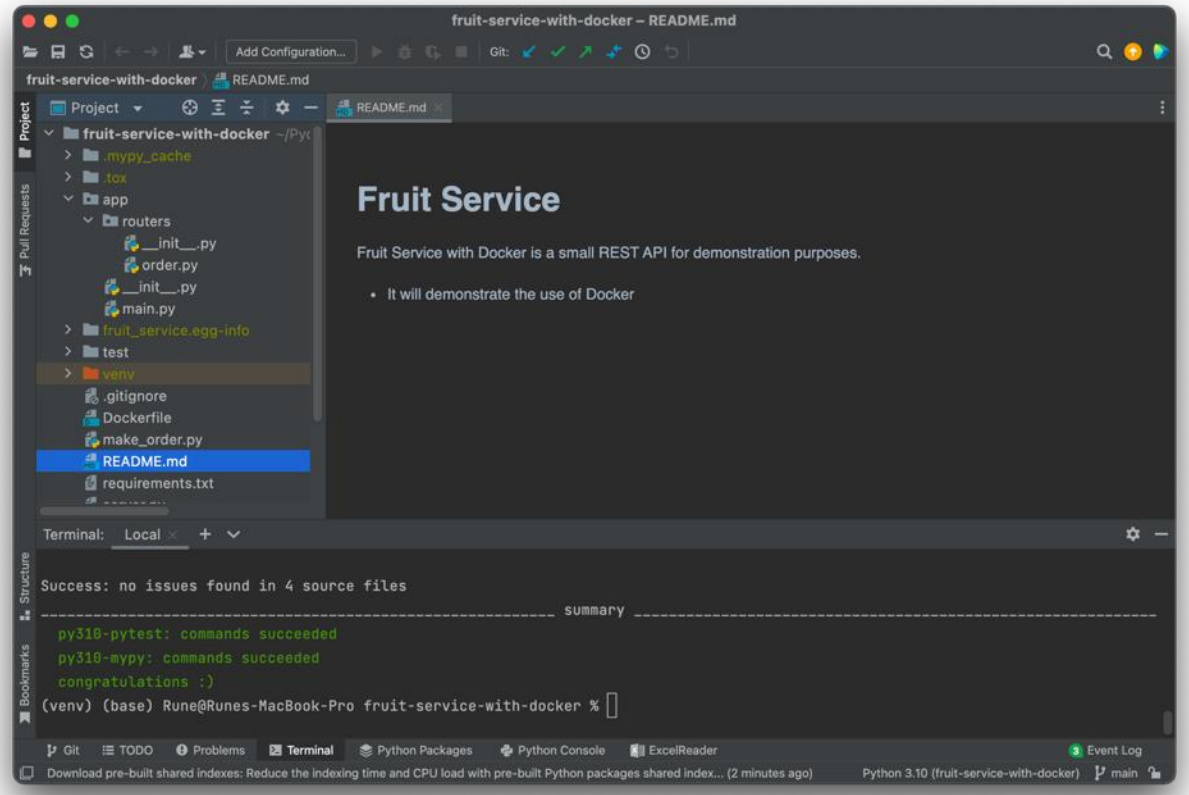
This is part of being a developer to clone all kinds of services – and you want to learn to do this seamlessly and ensure you do it correct.

Do you remember how to do it?

- Choose clone project in PyCharm and insert:
  git@github.com:LearnPythonWithRune/fruit-service-with-docker.git
  - If PyCharm is open in a project already, you can do that from menu **Git -> Clone…**
  - If you are not in a project you can clone a project using **Get from VCS** and paste it into. Make sure it has Git as



- Install Python interpreter (remember to use Python 3.10).
  - If you don't remember how to do it, then look in Chapter 1.
  - You can do it **Preferences -> Project -> Python Interpreter** and click **add** (hidden under the tool icon). Find the correct Python installation. Click OK and OK.
- Check correct Python version is installed.
  - Open terminal and run **python –version**
  - It should output **3.10**.
- Install **requirements**.
  - Open terminal and run **pip install -r requirements.txt**
- Then run **tox** (this is good practice to ensure the project works).

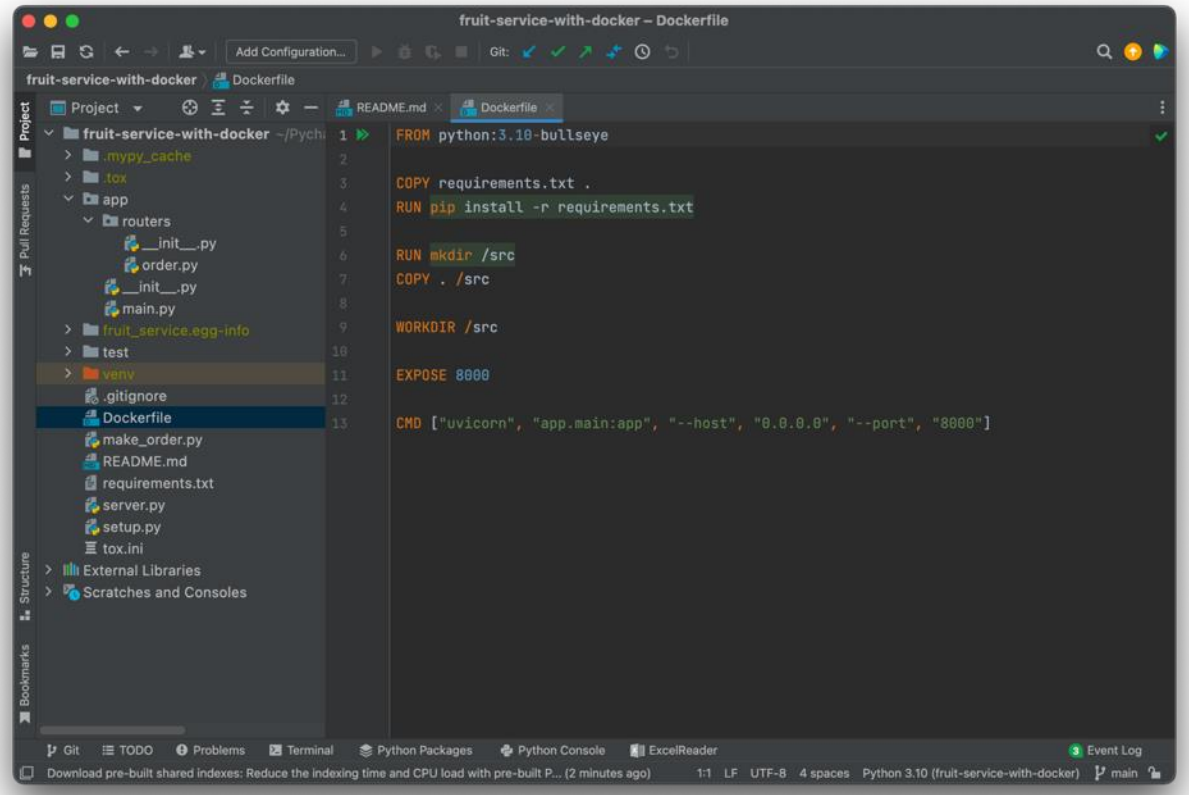


Now you are ready to build your first Docker image.

Explore and understand the Dockerfile

Do build a Docker image you need a **Dockerfile**.

In the project you will find the file named Dockerfile. Open it and inspect it.



A Docker image is build in layers. Let's try to explore this one.

```
FROM python:3.10-bullseye

COPY requirements.txt .
RUN pip install -r requirements.txt

RUN mkdir /src
COPY . /src

WORKDIR /src

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

On the first line we take the base image, which is the most recent Python 3.10 version, called **python:3.10-bullseye**.

- If you at a later stage want to user other Python versions in a Docker image, you can find them here: **https://hub.docker.com/_/python**
- Notice, that some are early alpha releases and it is not recommended to use them unless you know what you are doing.

The base image is like the foundation of what you are running it on. This is a prebuild image and contains all the standard libraries for the Python 3.10 release.

- It should be clear that whatever you will run from a container created by this image, cannot access anything except what is provided by this image and what we also add to it.

This makes the next line understandable.

```
COPY requirements.txt .
```

This will copy the requirements.txt file to the Docker image.

Notice that we need the requirement.txt to be in same folder as the **Dockerfile** where we will build the image.

```
RUN pip install -r requirements.txt
```

The next line will run a command: pip install -r requirements.txt.

Remember what that did?

Yes, you are right, it installs all the requirements.

Notice, that the RUN will execute this command inside the Docker image, and hence, the installation is inside the Docker image.

```
RUN mkdir /src
```

This will create a folder **/src** inside the Docker image.

```
COPY . /src
```

This will copy all the files to the Docker image.

Now you might wonder about one thing?

- Why first copy requirements.txt and install it and then copy everything thing afterwards?

This has something to do with how Docker images are created. And, yes, you could do it, but I will tell why the approach in the **Dockerfile** might be good.

Docker images are built in layers. And when you re-build an image, it will only build from the layer there has been changes.

Each line in the Dockerfile represents a layer.

Now consider this.

- Imagine you already have a built image of the Dockerfiler
- Then you make a change to app/main.py file.
- What happens?

This change does not affect the requirements.txt file. Hence the installation of all the required libraries will not be executed again. It will start from the **COPY . /src** command.

You see, it will save time the next time you build your image.

```
WORKDIR /src
```

This will set the working folder (or directory). This means, when you execute a command, it will run from this folder.

```
EXPOSE 8000
```

This will inform Docker that the container will listen to that specified network port.

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Finally, the **CMD** is the command.

This can seem a bit confusing. But it takes the list of arguments and runs the command **uvicorn** with all the arguments.

uvicorn app.main:app –host 0.0.0.0 –port 8000

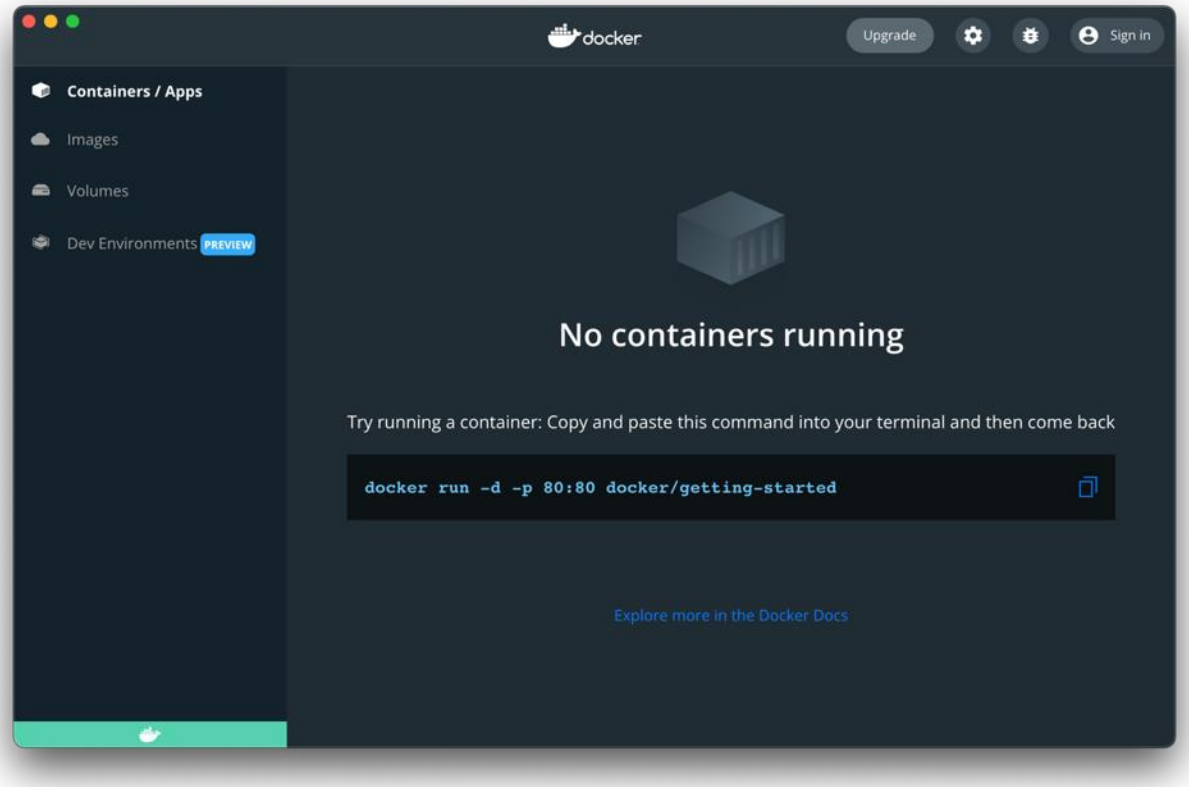
You might wonder.

- Why don't we run **python server.py**?
    - We could, but we will use the **uvicorn** instead here.

Now we have some basic understanding of the **Dockerfile**, we are ready to build the Docker image.

Build your first Docker image

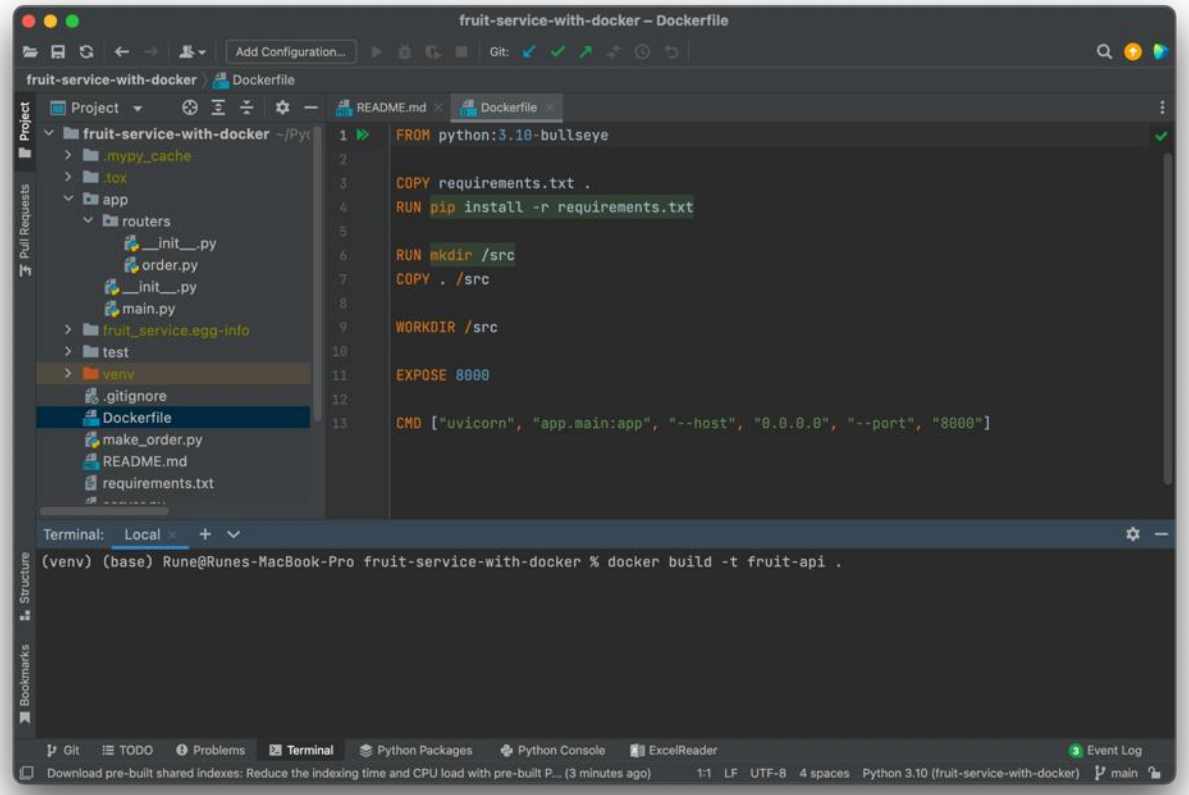
Before you build your first Docker image, you need to ensure that the Docker Desktop is running.



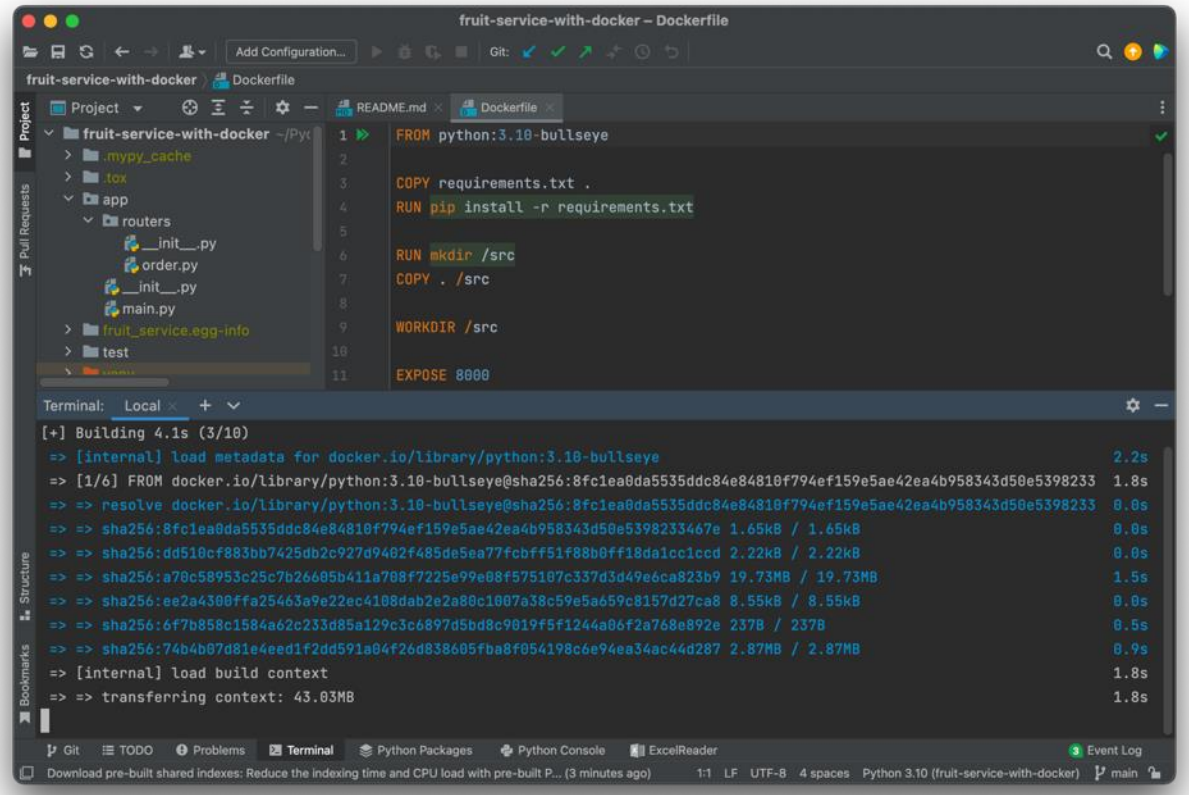
This is needed to have the Docker daemon running.

Now run the command: **docker build -t fruit api .**

Notice the space and dot at the end of the command.

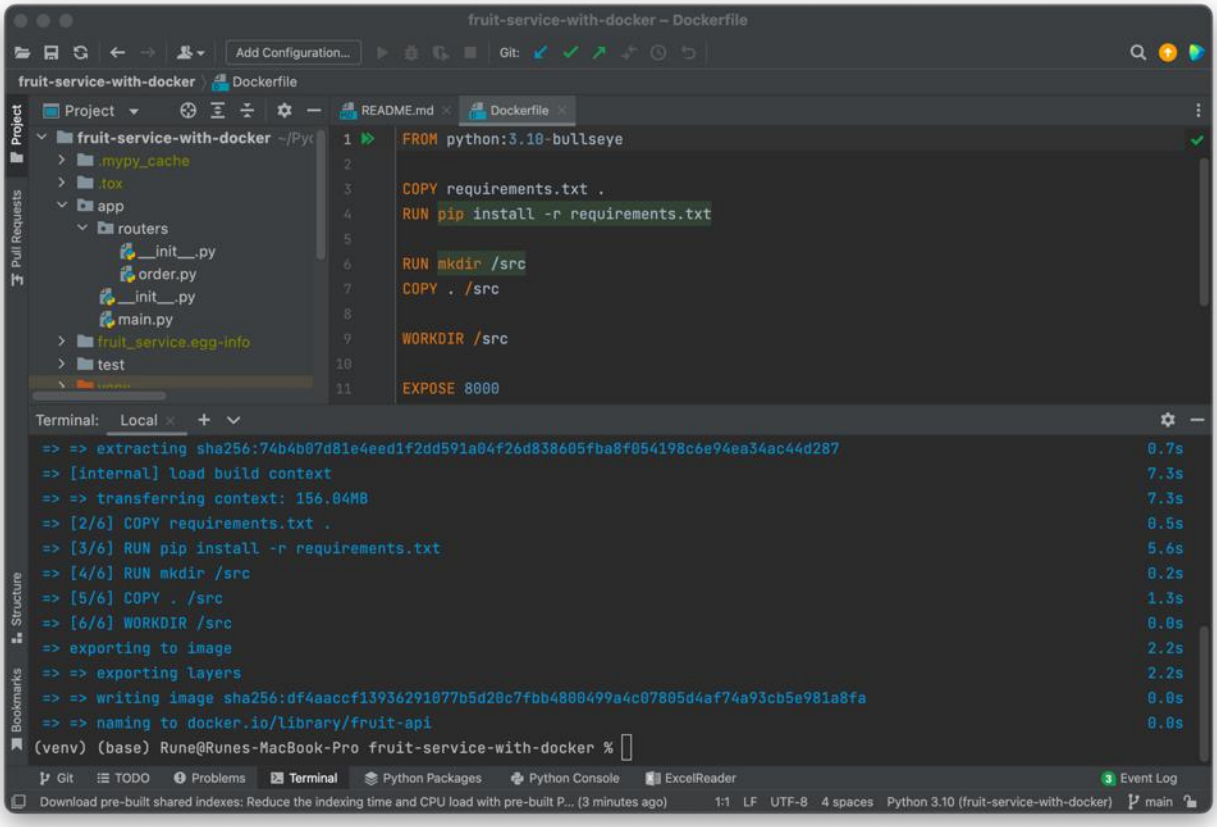


Then it will start making a lot of stuff.



You need to wait until it finishes.

If you look at the output (see below), then you will notice how it builds layer by layer.

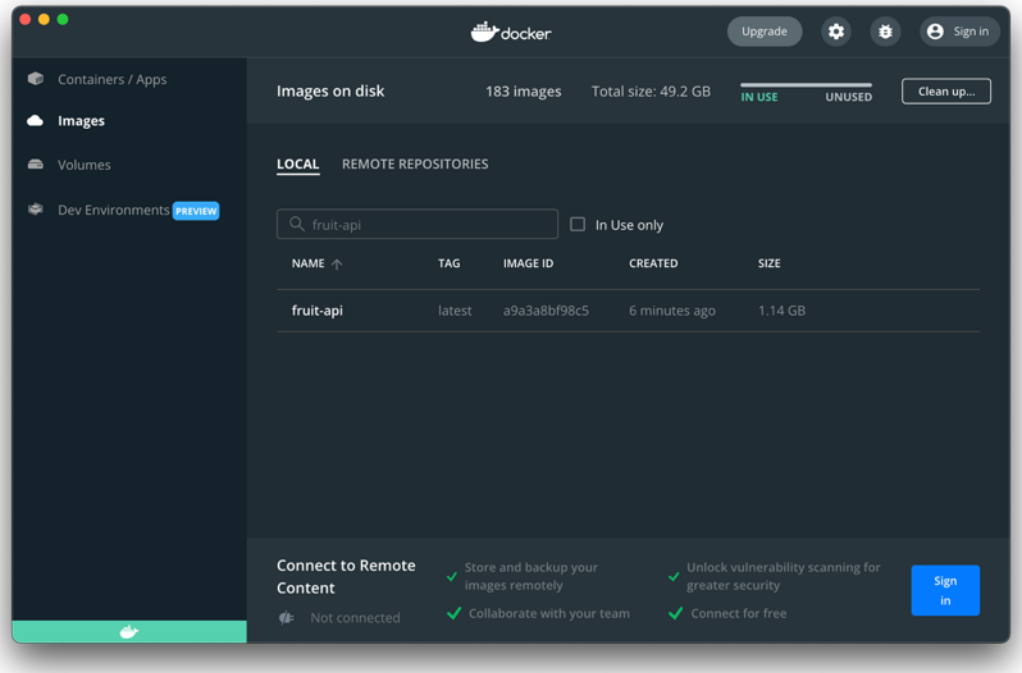


It tells you to use docker scan. This is beyond the scope of what we will cover and requires you create a login.

The image is created from the official Python image and the files in your project.

Now where is the image?

Look in Docker Dashboard (Docker Desktop) and click images and search for the fruit-api.



Now there is your image.

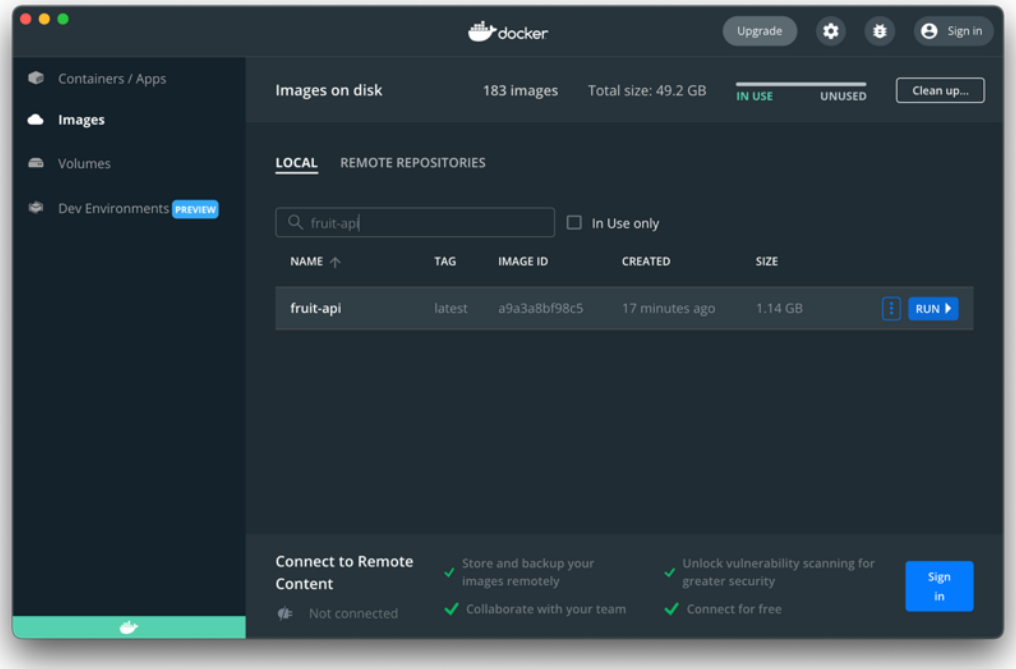
Can you access it?

It is not running. Remember, you need to launch it in a Docker container.

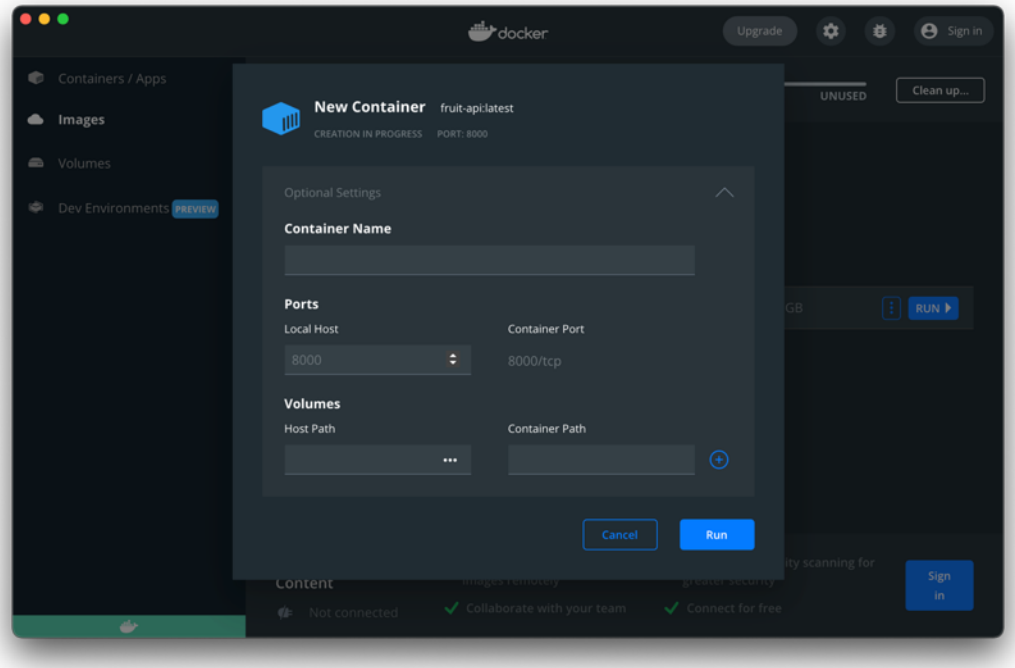
Run your first Docker container

There are two ways to run the Docker container.

1. You can do it from Docker Dashboard (Docker Desktop) by hitting the blue RUN when your mouse is hovering over the image.



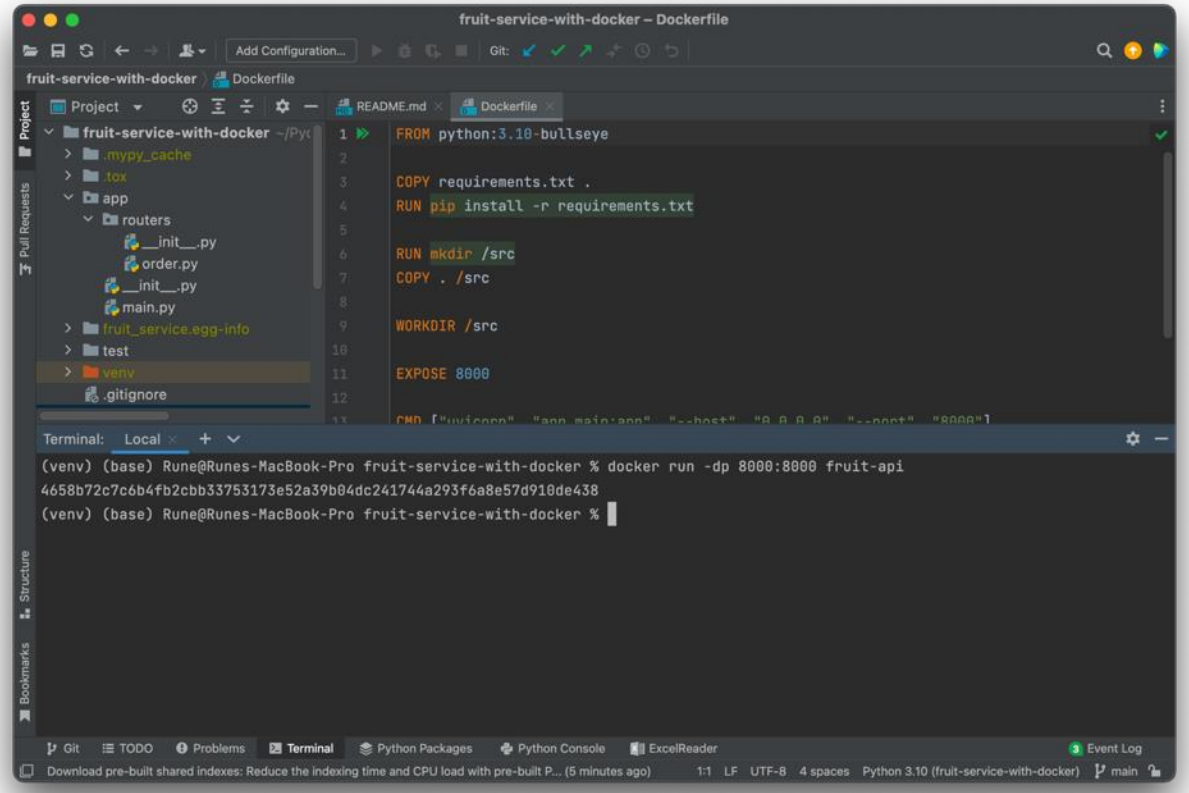
a. You need to choose optional settings and add local host port 8000.



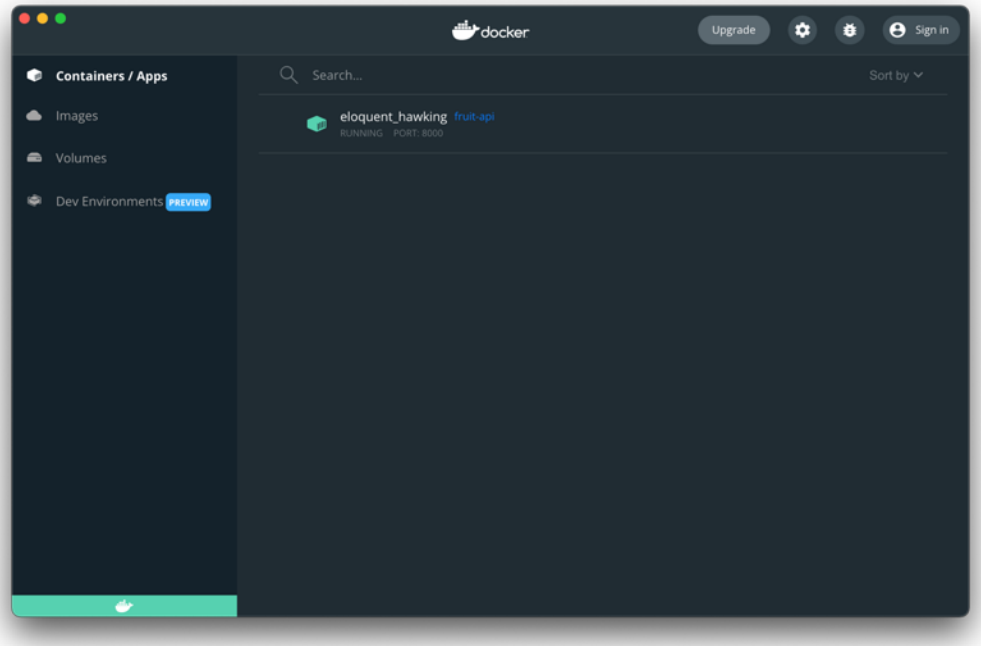
b. This is needed for Docker to know where to publish the port on local host. You could choose another port, it does not have to be port 8000, which is exposed inside the Docker container.

2. Run the command: **docker run -dp 8000:8000 fruit-api**

a. The **-d** will detach it from the command line, this is convenient as it will occupy the terminal until you finish the container.
b. The **p** part (of **-dp**) will publish the port 8000:8000 on host from container.

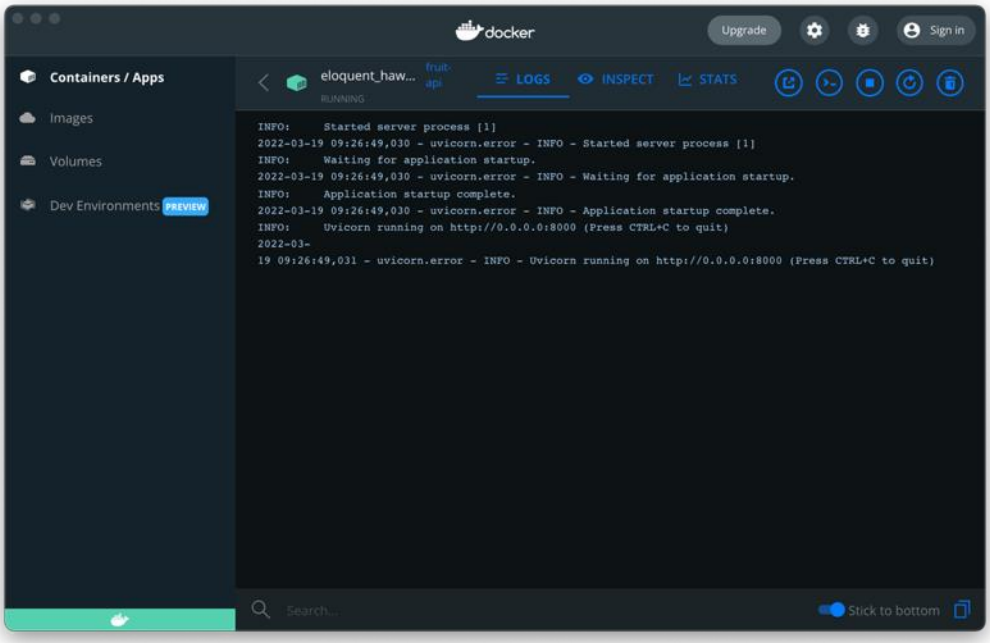


Whichever way you choose, then it should have a Docker container running.

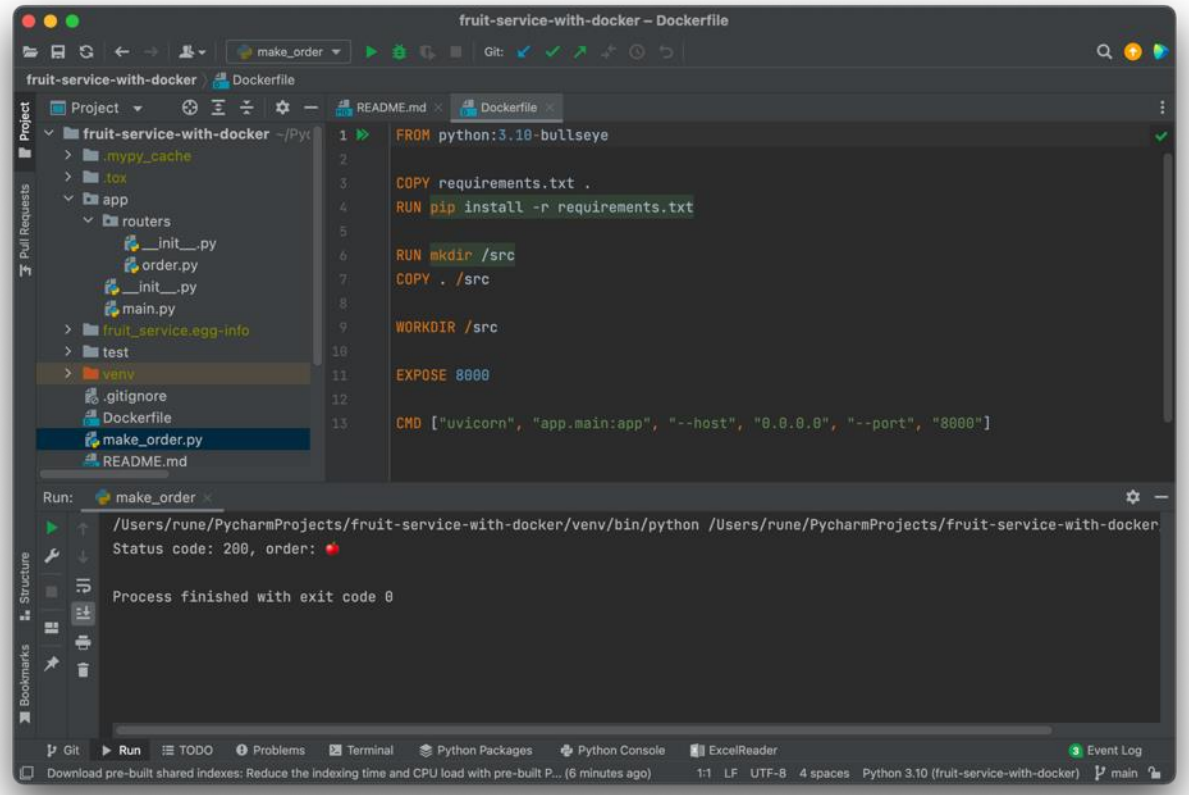


The name (**eloquent_hawking**) will most likely be different in your case. This is simply a random name.

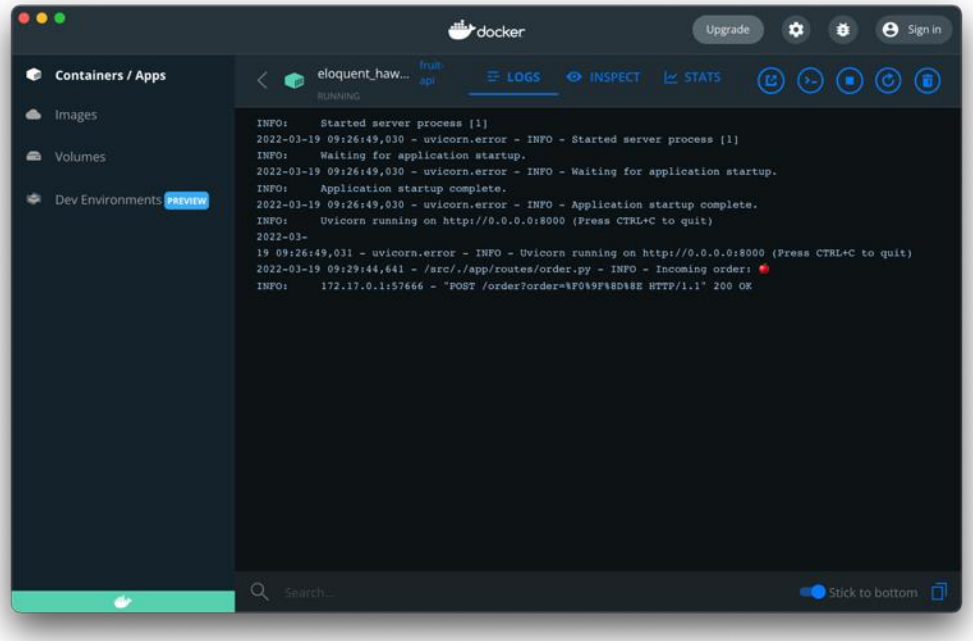
If you click on it you will see the output.



Now in our PyCharm project run **make_order.py**



This notice this in our Docker Dashboard.

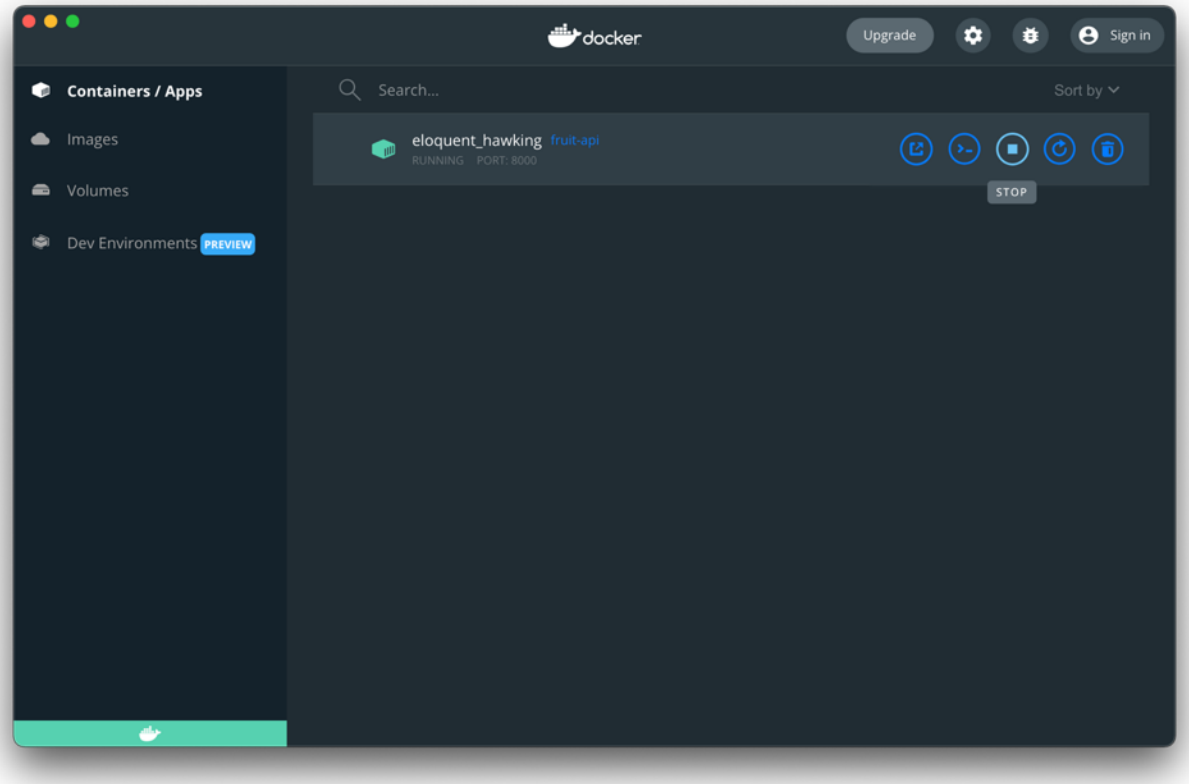


You see the order.

How to turn off your Docker container?

When you want to shut down your Docker container – that is, you don't want to have the service running on port 8000, what do you do?

If you hover the mouse over you will see the following options.



It has a stop button (and some more).

If you click the stop button, it will stop.

- Don't believe me? Try to stop it and run **make_order.py** and it will fail.

Then you can start it again with the RUN button (the STOP will change to a RUN button).

- Try it and run **make_order.py** and see it works again.

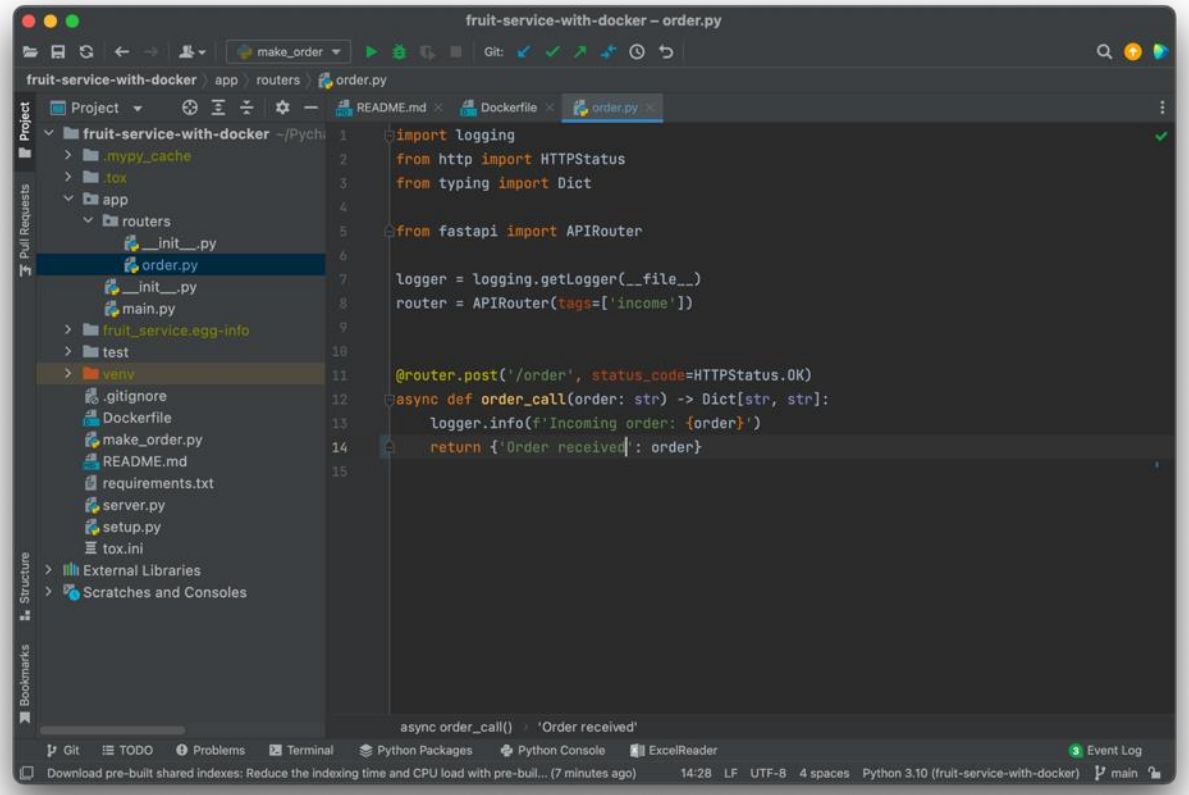
What if I make changes to my API?

Right now, you have built your Docker image and you can run it in a Docker container.

If you make changes to the code in PyCharm, those changes will not be reflected if you run the container we have available.

To make changes you need to build a new image and run that image as a container.

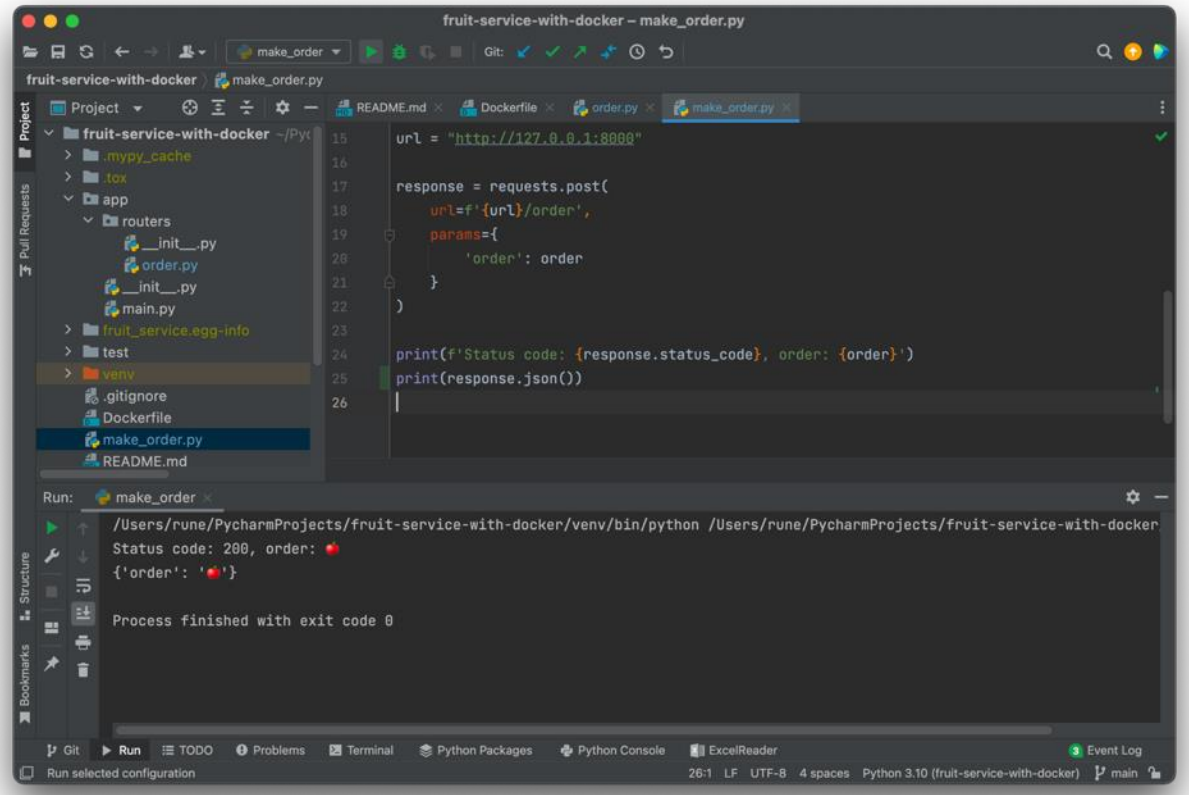
Let's try that.



Let’s make this simple change to line 14.

If you start the Docker container, you will see it is the old version running.

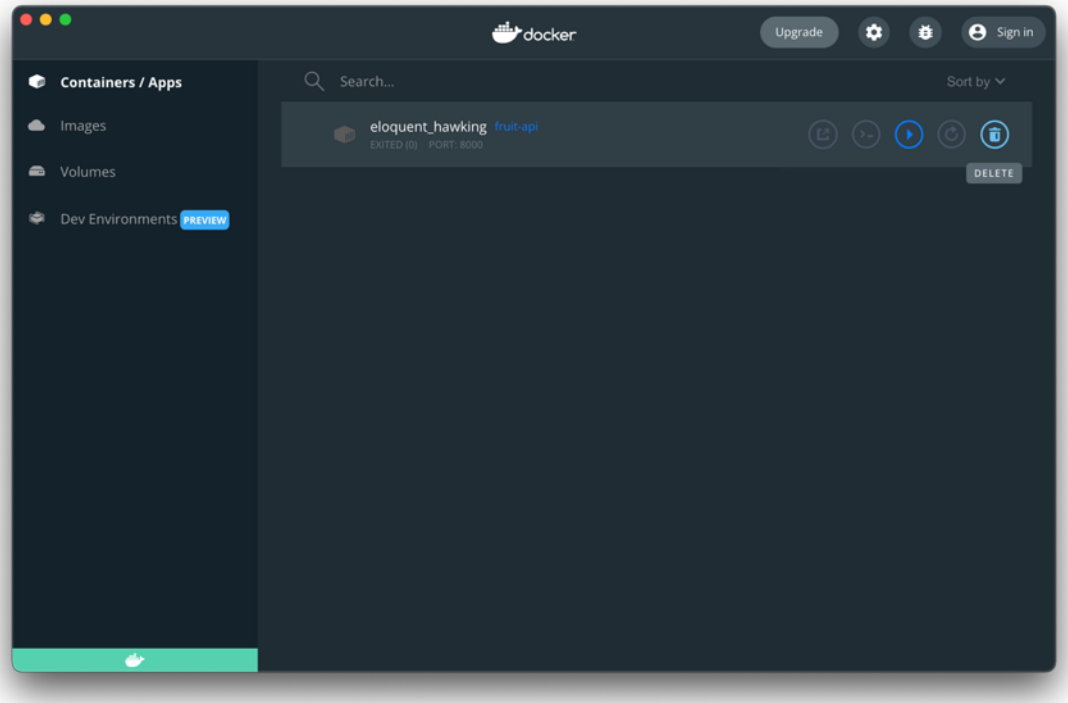
To be clear, you can modify make order a bit to print the response (see modification on line 25 below).



As you see – it returns the old response we had.

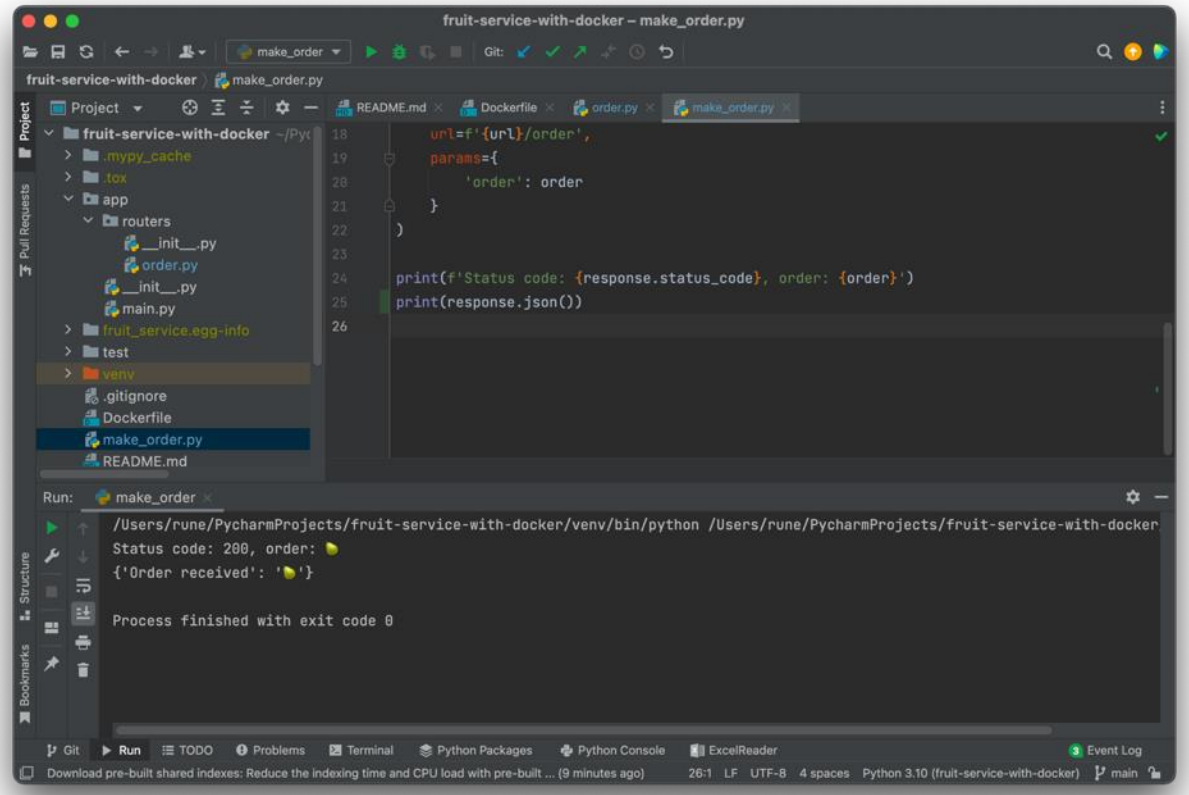
What to do?

1. Stop the container.
    a. You can do that as we saw with the STOP button in Docker Dashboard.
    b. Or run the command: docker **stop eloquent_hawking**
        i. Of course, you need to change the name after stop to fit with the name it gave your instance of the container.
2. Delete the container.
    a. This can also be done from Docker Dashboard (use the DELETE button – hiding under the symbol).



   b. Or run the command: **docker container rm eloquent_hawking**
      i. Again, you need to change it to your containers name.
3. Build the image again.
   a. Run **docker build -t fruit-api .**
   b. This is done in the terminal.
   c. Notice, that it will be faster, as it does not need to build all layers.
4. Run the image
   a. Run the command: **docker run -dp 8000:8000 fruit-api**
   b. This could also be done from Docker Dashboard.
      i. Go to images
      ii. Find the image
      iii. Run it and remember to publish the port.

Now make an order (run **make_order.py**)



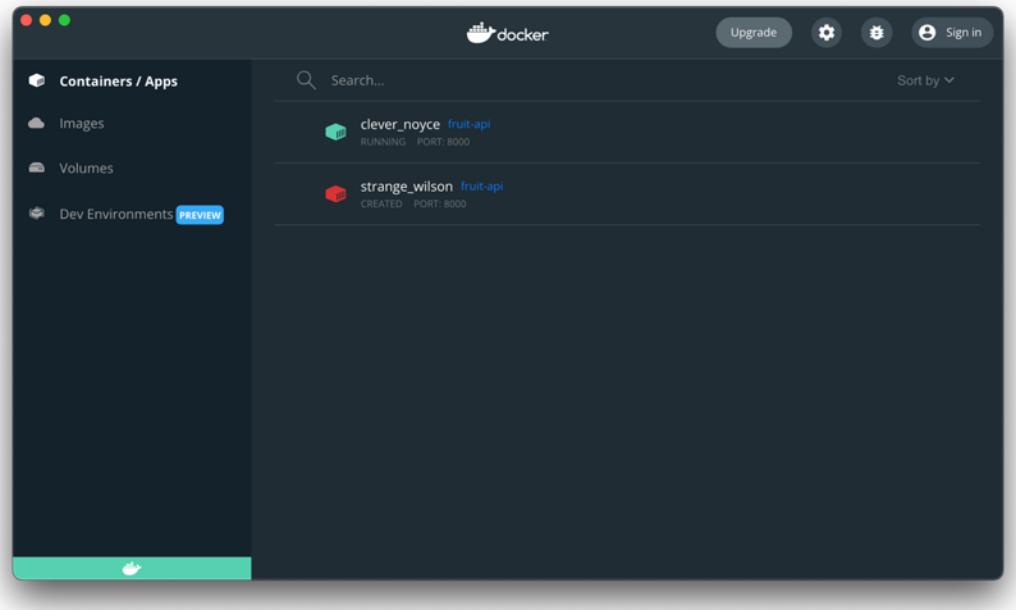
And it prints our changes.

A few common problems from Docker

You start your docker image (**docker run -dp 8000:8000 fruit-api**) and the response is similar to this.

```
docker: Error response from daemon: driver failed programming external
connectivity on endpoint strange_wilson(…) : Bind for 0.0.0.0:8000 failed: port
is already allocated.
```

It says, something is running on port 8000 already. This is most likely because it is already running.



In this case DELETE them both.

If for some reason, it is not the case, you can publish the API on a different port by running: **docker run -dp 8007:8000 fruit-api**

Now it will be published on port 8007.

Exercise

Let's try to change the Dockerfile and see what happens.

- Remove a line from requirements.txt (delete the line with fastapi)
- Then type: **docker build -t fruit-api-exercise .**
- From which layer does it build?
- Are the following layers also build again?
- Try to run the new image: **docker run -dp 8000:8000 fruit-api-exercise**
- Does it work? Why or why not?
- Does this destroy the image we build in the chapter (fruit-api)?

Summary

In this chapter we learned the following.

- That Docker can help us run our API in an isolated environment with own version of Python and libraries.
- Learned about Docker terminology: Dockerfile, Docker image, Docker container, Docker Desktop, Docker daemon to mention the most important ones.
- How the Dockerfile defines the layers in the Docker image.
- Creating a Docker image.

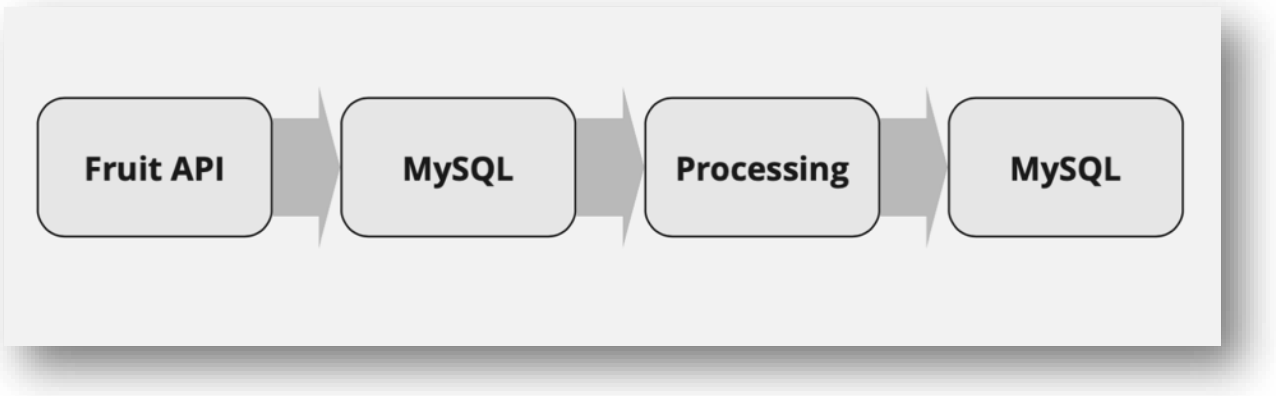
- Running a Docker image and publishing an exposed port on your machine, such that you can call the API from your PyCharm environment.
- How to stop and delete a Docker container.
- Also, how to make changes and rebuild the image and run it again.
- Finally, a few common troubles you might encounter.

05 – Adding Another Service

In this chapter we will add another service to our ecosystem.

- First let's explore what we want to create at the end of this book.
- You will learn how to have multiple services running in Docker.
- How Docker containers can reach host ports, where other containers have published ports.

Overview of the Ecosystem



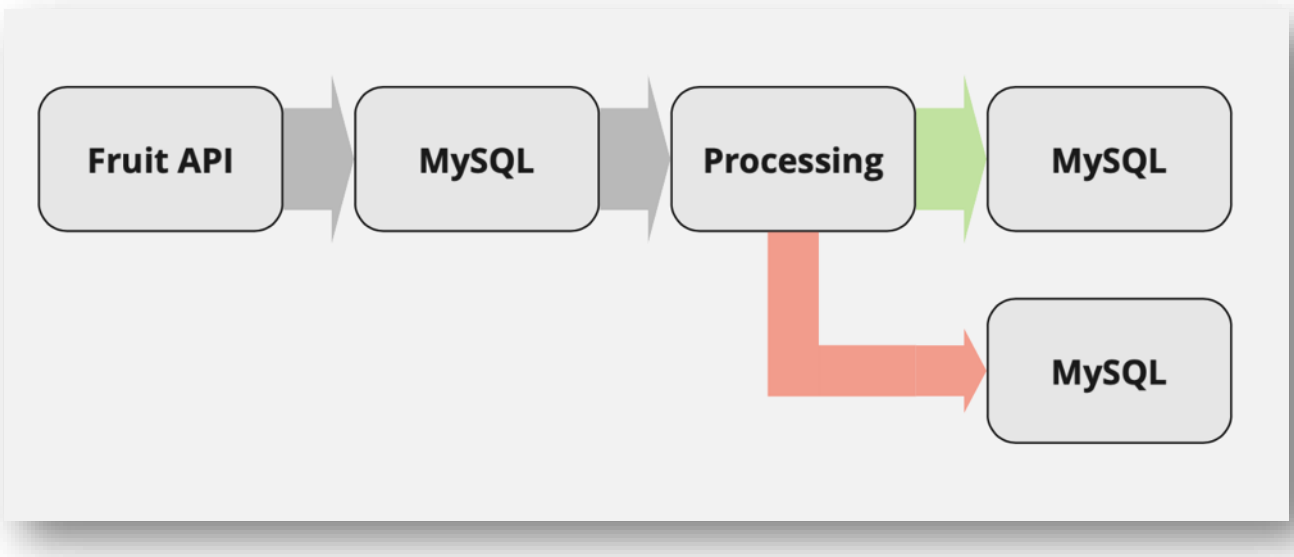
For now, we have created the Fruit API service, but it does not really have any functionality. You can make an order and nothing happens.

First step would be to store the raw order somewhere as it is. After it has been stored we would like to process it, to ensure it fulfill all requirements.

Why do we store the raw request (or order)?

- When working with data, especially data you cannot get again, then you would want to have a raw copy of how it was when it first entered in our system.
- Imagine you are crawling webpages – they change all the time – hence, if we crawl it and process it and it fails, we might not be able to get the data again.
- That is why we would like to keep the original data.

Actually, a normal setup of processing raw data is as follows.



We will get the raw data and store it, then we will process it and store the data that succeeds in one place and data the does not succeed in another place.

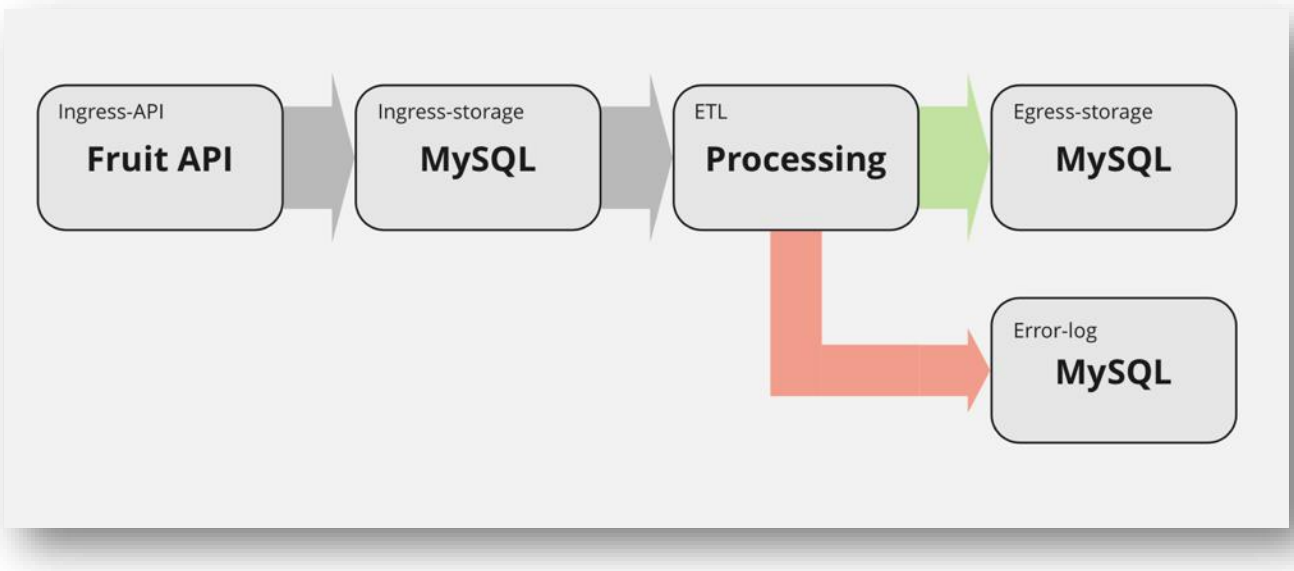
This gives us the following.

- We have the raw data the we can go back to figure out what failed (if it did).
- Also, with the raw data we can process it again, if we need to (this often happens as things fail or we make improvements).
- We have all the correctly processed data available in a MySQL database.
- A clear view of what data failed.

Do we need 3 MySQL databases?

- No – you can have it all in same database in different tables.
- But there might be reasons to have separate databases, like ensuring performance and have critical infrastructure separated.

What are all these components called?



First all, there are a lot of names for each module and we will keep it simple here. Hence, if you see another naming, then it is quite normal.

- The **Ingress-API** is the entry point to our ecosystem.
- **Ingress-storage** is where we store the raw data – other very common names are **stage** and **landing**.
- **ETL** is standing for Extract-Transform-Load. It reads some data from a source (here the **Ingress-storage**), then it makes some transformation (extracting and converting values) and loads it into the storage (here the **Egress-storage**).
- **Egress-storage** is where we have data prepared for further usage – other common names are **uniform** or **ODS** (Operational Data Store).
- **Error-log** is our storage to keep records of what failed – this is a crucial element and will be valuable when you need to figure out why something is not working.

You should see the value of this design as it gives you a great way to build a robust system that takes input data, makes a transformation or validation before the data is ready.

This can be used in many settings, as we will see later, this is how you can have multiple web scraper work and make them easy to monitor and bug fix, and be able to run data again, if something failed.

Fruit API and MySQL

In this chapter we will work with Fruit API and MySQL for the ingress storage.

- We will build Docker images.
- Will run the them in Docker.
- Try to call them and understand the world of your two services.
- Run them in PyCharm and see how that works.
- How to debug such services, while they are running in PyCharm.

To get there – we first need to clone and prepare the two repositories.

And yes, we have a new version of Fruit API.

Why is that needed?

Well, the new version needs to communicate with Storage API.

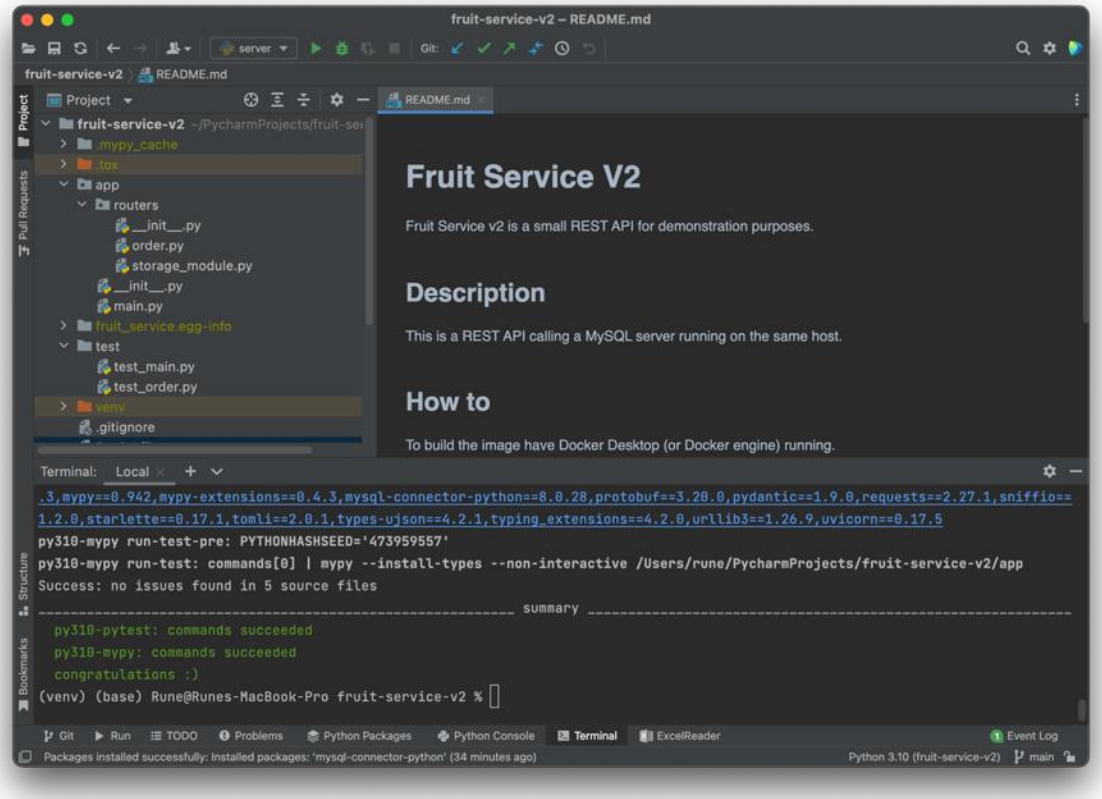
Get ready for this chapter

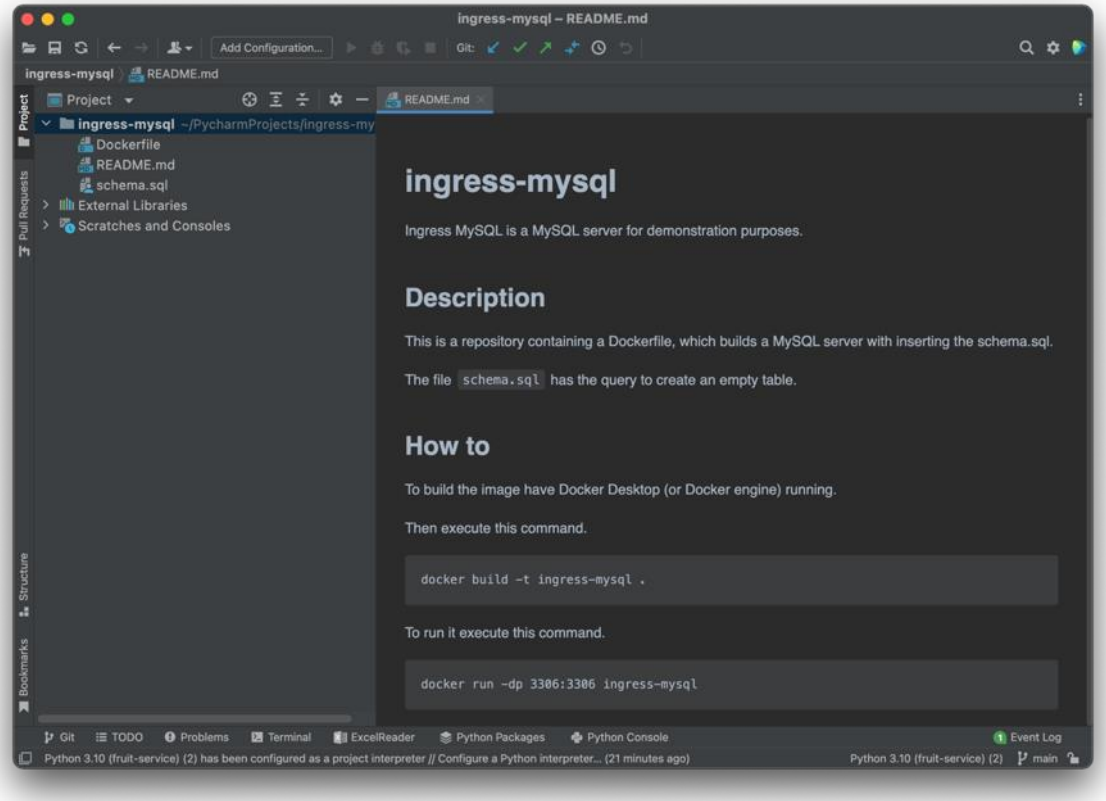
You will need to do the following to get started with this chapter.

- Clone the following two repositories.
  - **git@github.com:LearnPythonWithRune/fruit-service-v2.git**
  - **git@github.com:LearnPythonWithRune/ingress-mysql.git**
- This includes the following for Fruit Service V2
  - Cloning them
  - Install python in a virtual environment
  - Install all requirements (**pip install -r requirements.txt**)
  - Check if all okay (run **tox**)

This procedure will become second nature to you.

At the end you should have two PyCharm windows looking similar to these.





Explore ingress-mysql

Let's start with the Dockerfile.

```
FROM mysql/mysql-server

ENV MYSQL_DATABASE=DB \
    MYSQL_ROOT_PASSWORD=password \
    MYSQL_ROOT_HOST=%

ADD schema.sql /docker-entrypoint-initdb.d

EXPOSE 3306
```

It takes the official MySQL image.

It sets a few environment variables, adds a **schema.sql**, and exposes port 3306.

Starting with the last part.

A default MySQL server runs on port 3306. This enables you to connect to the MySQL server and run queries on that port. We will learn more about that when exploring Fruit Service V2.

The schema.sql file contains what SQL query is executed when the image is created.

Here we see the content.

```
DROP DATABASE IF EXISTS `DB`;
CREATE DATABASE `DB`;
```

```
USE `DB`;

DROP TABLE IF EXISTS `ingress`;
CREATE TABLE `ingress` (
  `ingest_time` datetime,
  `ingest_value` varchar(100) NOT NULL
);
```

If the Database **DB** exists, then delete it.

Then Create the **DB** database and use it.

After that, we will delete the **ingress** table if it exists and create it afterwards.

If you build the image, it will create an empty MySQL database called **DB** with an empty table ingress with two columns: **ingest_time** and **ingest_value**.

I just got an idea.

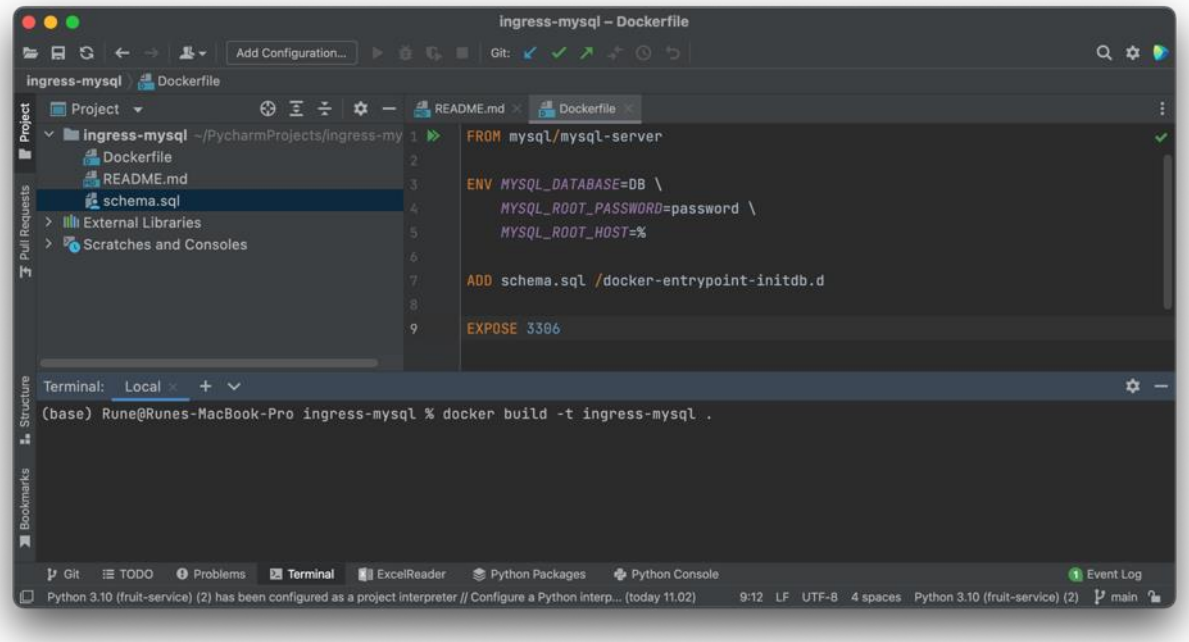
Let's try to create one.

Running ingress-mysql

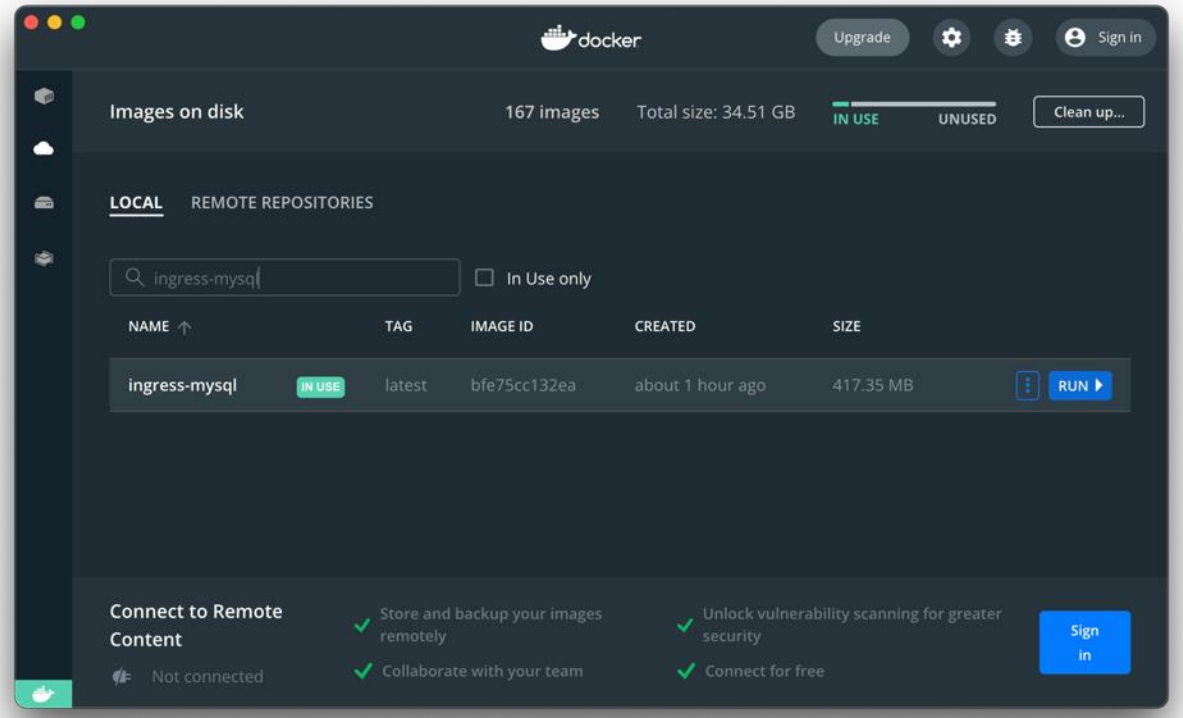
We first need to build an image of it.

This is done by running the following command in the terminal.

docker build -t ingress-mysql .



This should create an image in your Docker Desktop. You can see it by searching your images.



Then you can run it by pressing the blue run button (you need to hover your mouse over the line to see it) or by running the following command from the terminal.

docker run -dp 3306:3306 ingress-mysql

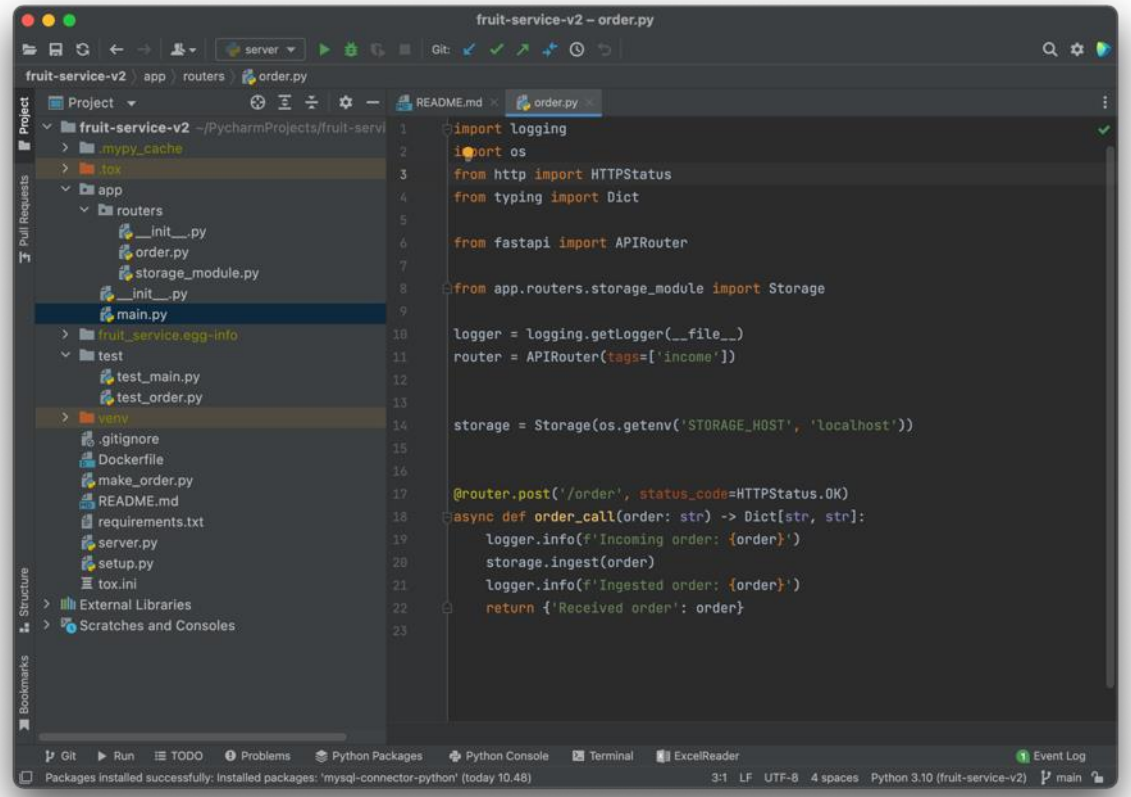
This start it and expose port 3306 on your localhost.

Now it is ready to be used by the Fruit Service V2 API.

Explore Fruit Service V2

Let's look at some of the changes in Fruit Service V2 to get familiar with it.

We start by inspecting the **app/routers/order.py**



As you will notice, a storage module is added here on line 8.

```
from app.routers.storage_module import Storage
```

Now something a bit trickier happens on line 14.

```
storage = Storage(os.getenv('STORAGE_HOST', 'localhost'))
```

This requires some explanation.

- We want our API to run in a Docker container but also in our PyCharm environment.

When we run ingress-mysql in Docker and Fruit API in PyCharm, then they will use port 3306 and port 8000, respectively.

This all works fine.

The problem comes when Fruit API is also running in a Docker container. Then localhost will refer to inside the Docker container, where Storage API is not running.

If we inspect the Dockerfile we will see the following code.

```
FROM python:3.10-bullseye

COPY requirements.txt .
RUN pip install -r requirements.txt

RUN mkdir /src
COPY . /src
```

```
WORKDIR /src

EXPOSE 8000
ENV STORAGE_HOST=host.docker.internal

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Where we see the **ENV STORAGE_HOST** is set to **host.docker.internal**

This is exactly what this is looking for.

```
storage = Storage(os.getenv('STORAGE_HOST', 'localhost'))
```

If the environment (**ENV**) variable **STORAGE_HOST** is set, use the value it is set to, otherwise use the default value **localhost**.

Inside the Docker container it is set to use **host.docker.internal**, but when we run in PyCharm it will use: **localhost.**

The **host.docker.internal** will refer to the localhost Docker is running on and will access the port there.

The **/order** endpoint is now calling the **Storage** with **create_file**.

Let's inspect the **app/routers/storage_module.py**

```
from datetime import datetime

import mysql.connector  # type: ignore


class Storage:
    def __init__(self, host: str):
        self.host = host

    def _setup_db(self):
        self.mydb = mysql.connector.connect(
            host=self.host,
            user="root",
            password="password"
        )

        self.cur = self.mydb.cursor()
        self.cur.execute("USE DB")

    def _query_db(self, sql_stmt: str):
        self._setup_db()
        self.cur.execute(sql_stmt)
        self.mydb.commit()
        self._close()

    def _close(self):
        self.cur.close()
        self.mydb.close()

    def ingest(self, ingest_value: str):
        sql_stmt = f"INSERT INTO ingress(ingest_time, ingest_value)
VALUES('{datetime.utcnow()}', '{ingest_value}')"
        self._query_db(sql_stmt)
```

This is a simple module which can connect to a MySQL database. It exposes one call to ingest a value.

The ingest method adds a timestamp with **datetime.utcnow().** This can help to keep track of when data is ingested. It is good practice and can help you figure out what happened at what time, what new data is there since last check, and if you need to get data from a specific ingest time and forward.

The Storage module opens and closes the connection for each call. This might not be necessary to do, and you might want to keep it open depending on the volume of queries and time between them.

The Storage module is a great way to decouple the code.

- This makes it easy to change the type of storage we use.
- If we later want to change to a Postgres (PostgreSQL) data base, a simple file storage, a big data storage, or whatever you need – then you only need to change the code in this class.
- Also, if you want to keep the connection open, then you also just need to change the code here.

Why do we use **MySQL**?

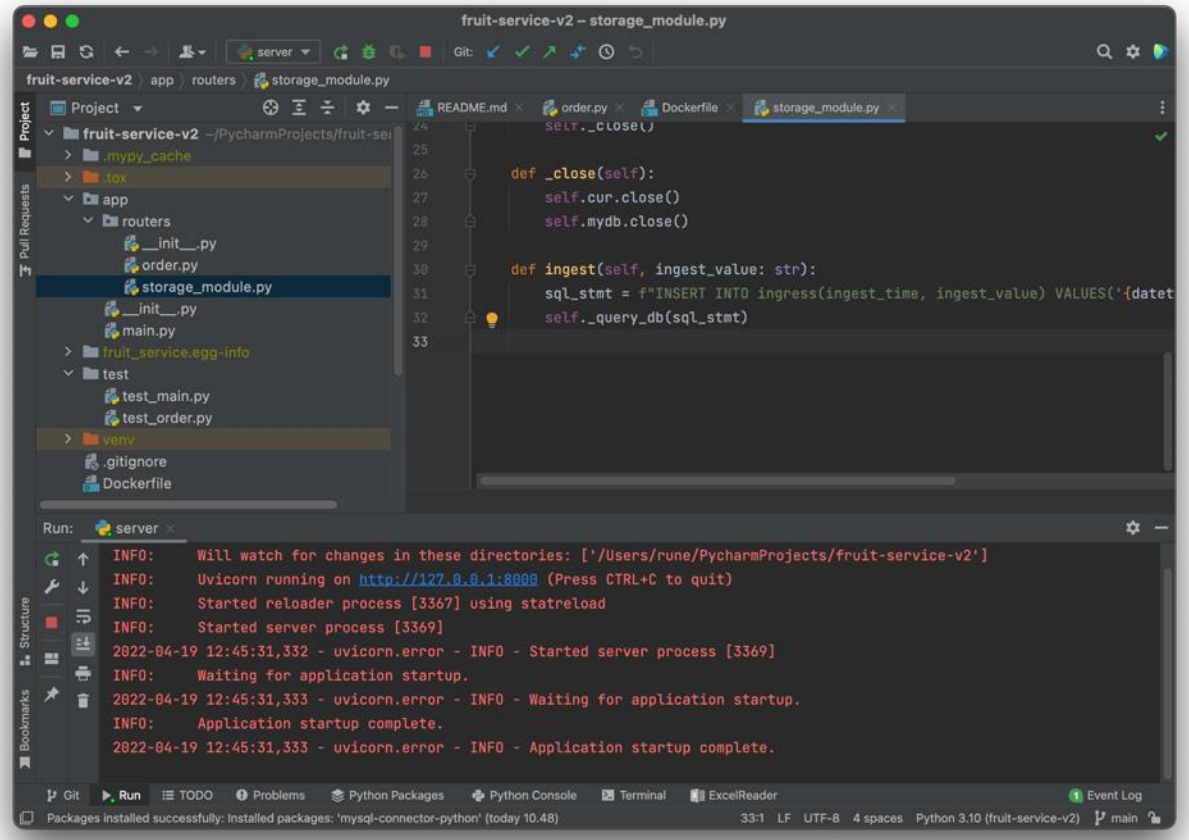
- **Postgres** (PostgreSQL) is another great free alternative. It requires more resources to run, but is more efficient in a bigger production setup.
- **MySQL** is more lightweight and sufficient for our purpose.
- **Postgres** more suited for bigger systems and can model more than just relational databases.
- But you can use whatever you want – but for this project and many others, **MySQL** is fine.

Now we are ready to run it.

Run Fruit Service V2 in PyCharm

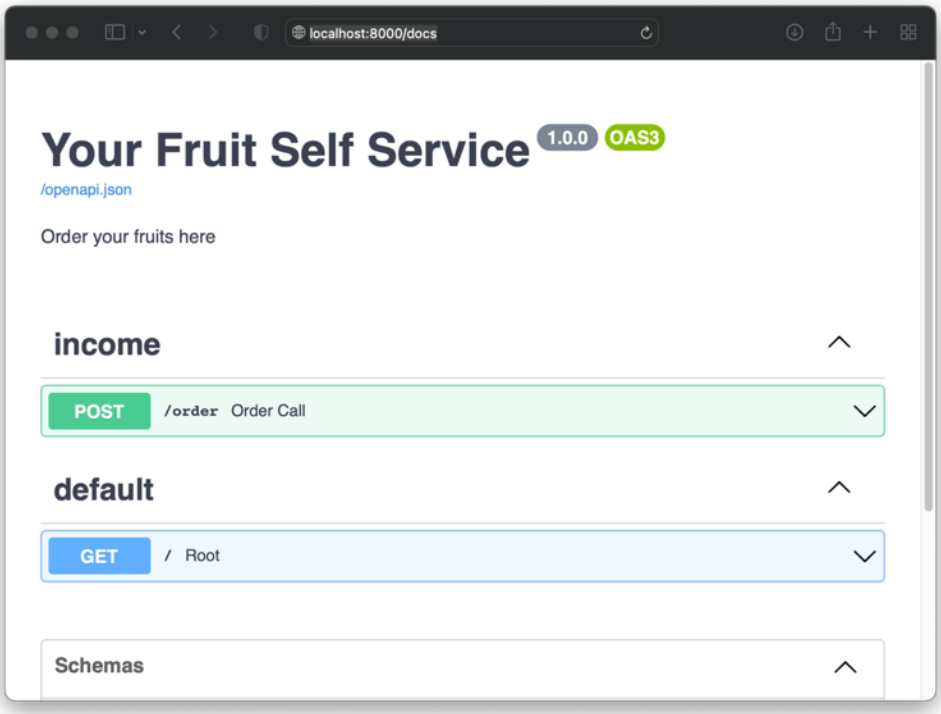
Let's start by running the Fruit Service in PyCharm.

Do that by running the **server.py** file like we did previously. This will expose the API as we know it.



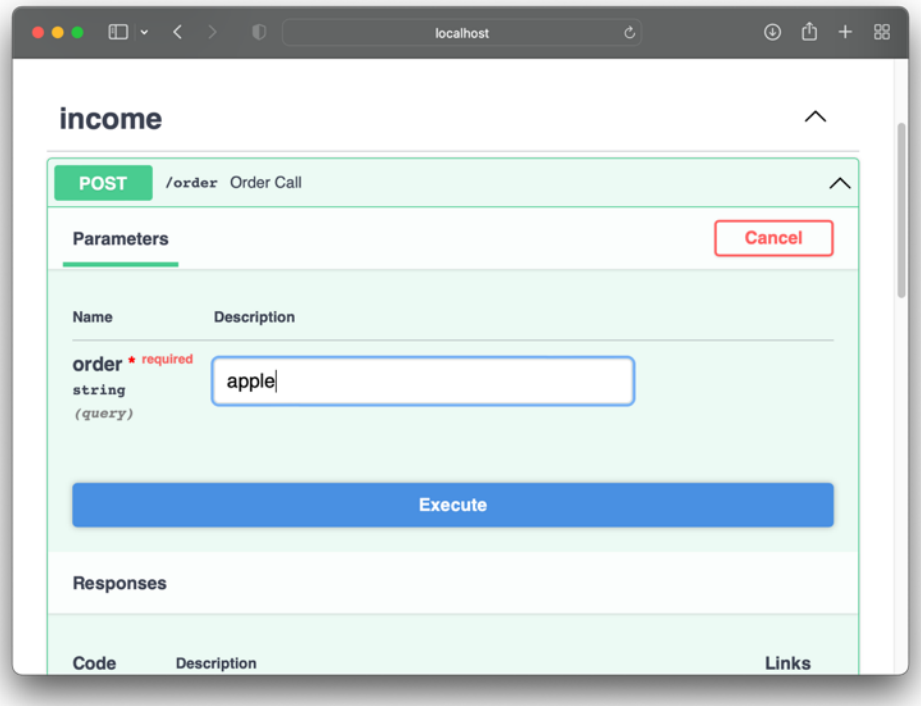
And in the browser you can access the API here.

http://localhost:8000/docs

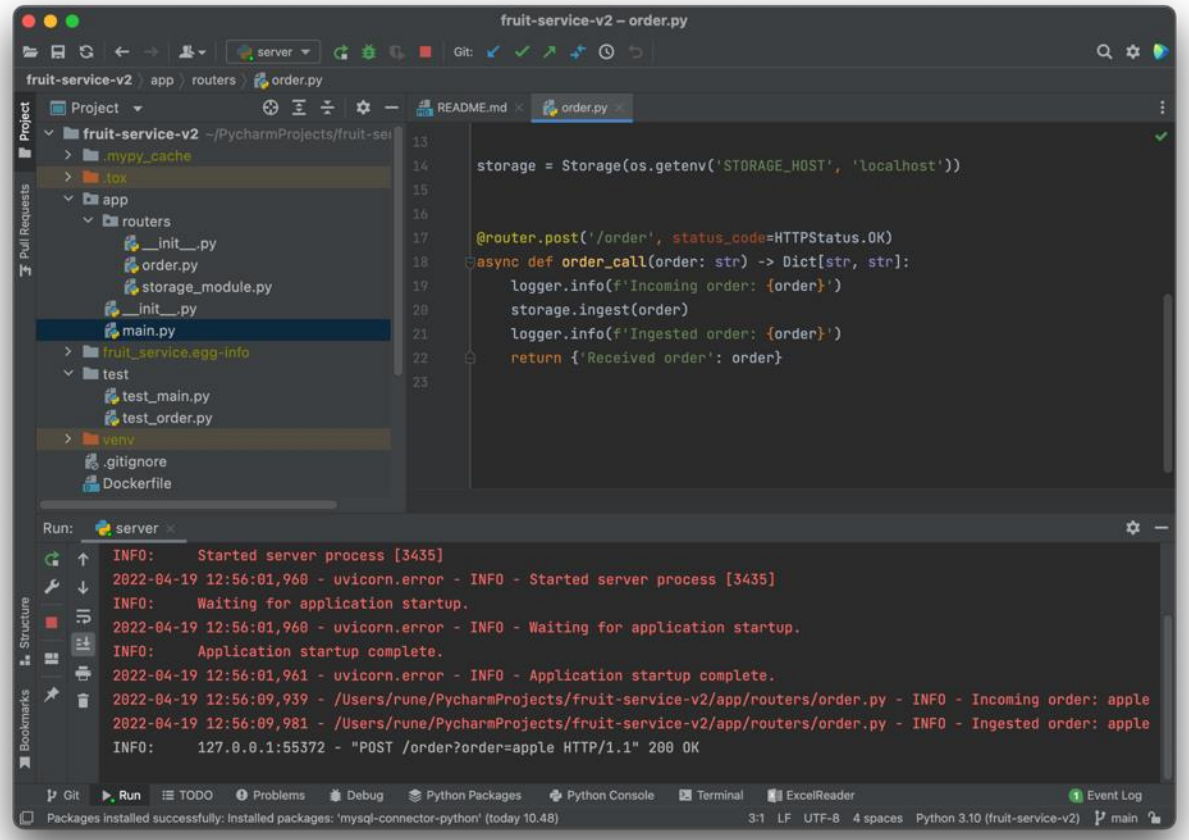


Let's make an order:

-



Resulting in an reaction in PyCharm.



Now we should be able to see it in the Database.

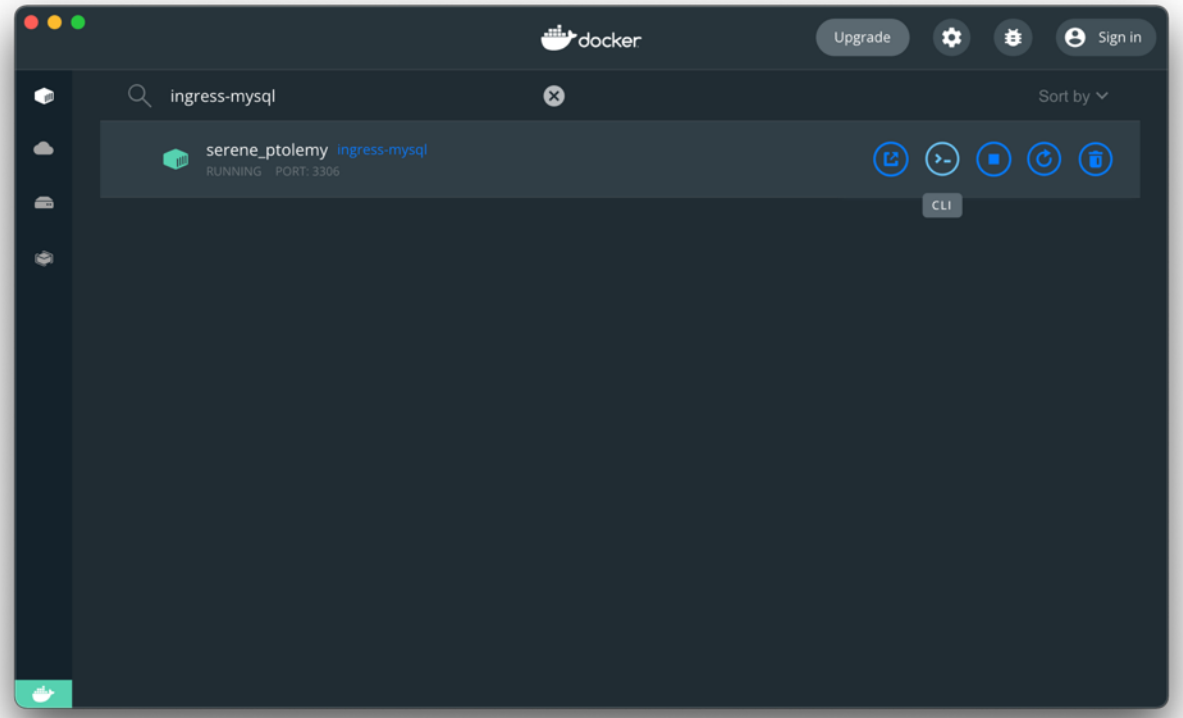
Wait a minute, how do we do that?

Well there are many options, including.

- Using MySQL Workbench (free tool to download).
- Connecting to a CLI (bash terminal) to Docker container.
- Writing a Python script to query the database.

And many other options. But we will connect CLI to the docker container. This will make you comfortable to work with Docker containers and figure out how they work.

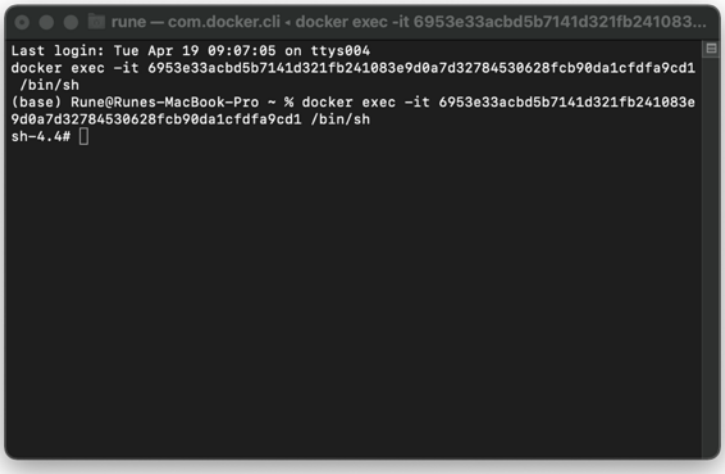
Go to Docker Desktop and hover your mouse over the right running container (you might only have the one running).



You will notice that there is a CLI button.

Click it and see what happens.

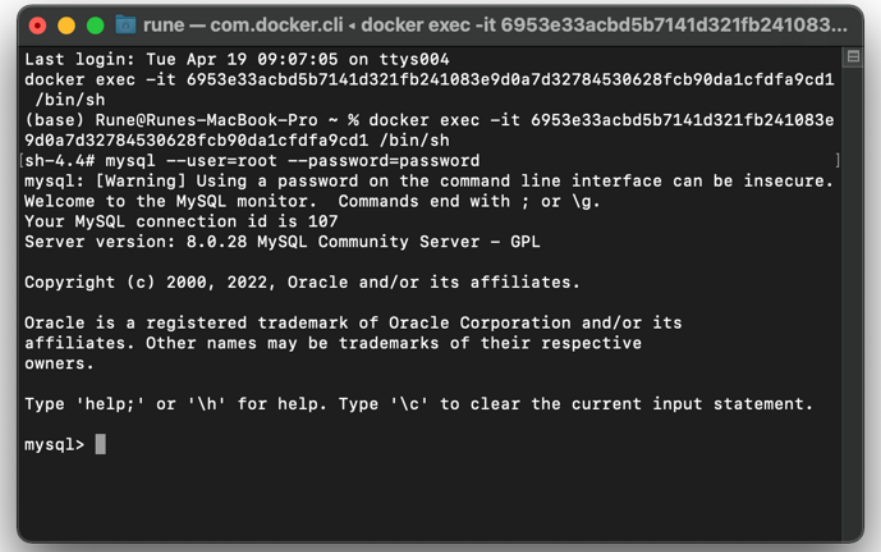
You get a command prompt connected to container.



From that you can connect to the mysql database by writing the following.

mysql --user=root --password=password

You should get something similar to this.



If you write the following and execute it.

show databases;

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| DB                 |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

Then we can see our DB database.

To use it, write.

use DB;

```
mysql> use DB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
```

Database changed

This is quite fun.

Now let's see if we have anything in our database.

Write the following command.

show tables;

```
mysql> show tables;
+--------------+
| Tables_in_DB |
+--------------+
| ingress      |
+--------------+
1 row in set (0.01 sec)
```

This shows our one table.

Let's see what's in it.

Write.

select * from ingress;

```
mysql> select * from ingress;
+---------------------+--------------+
| ingest_time         | ingest_value |
+---------------------+--------------+
| 2022-04-19 11:09:56 | apple        |
+---------------------+--------------+
1 row in set (0.00 sec)
```

Now we know it works.

Now let's shut down our server running in PyCharm and create an Docker image and run it.

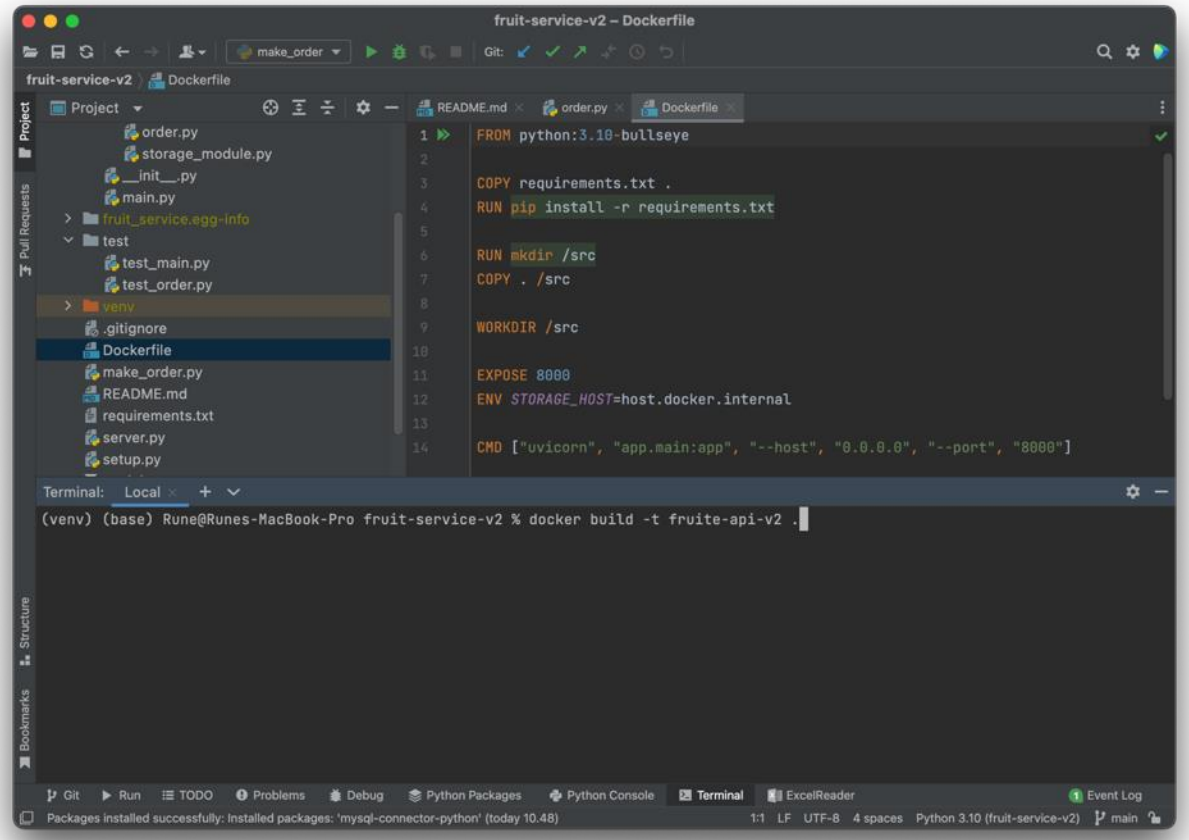
Build the Docker image for Fruit Service V2

Inside PyCharm in our Fruit Service V2 – let's create a Docker image.

We will call the image fruit-api-v2.

This is done with the following command, which you should run from the terminal.

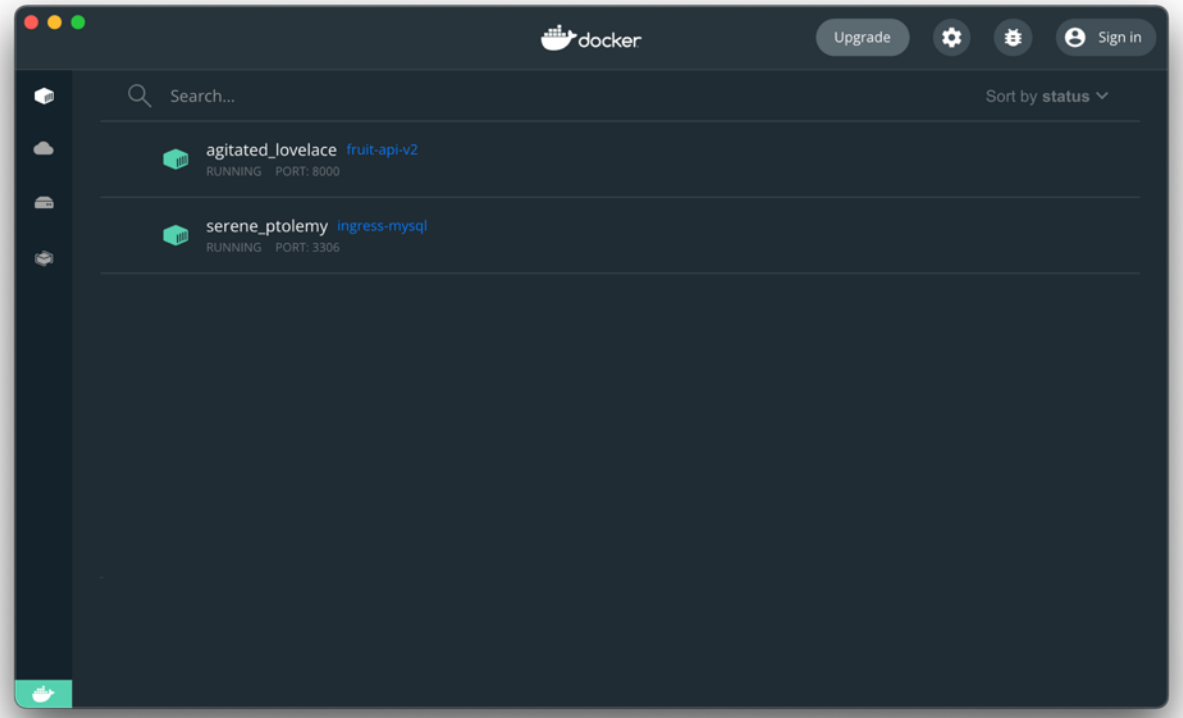
Run **docker build -t fruit-api-v2 .**



And then you can run it with the following command.

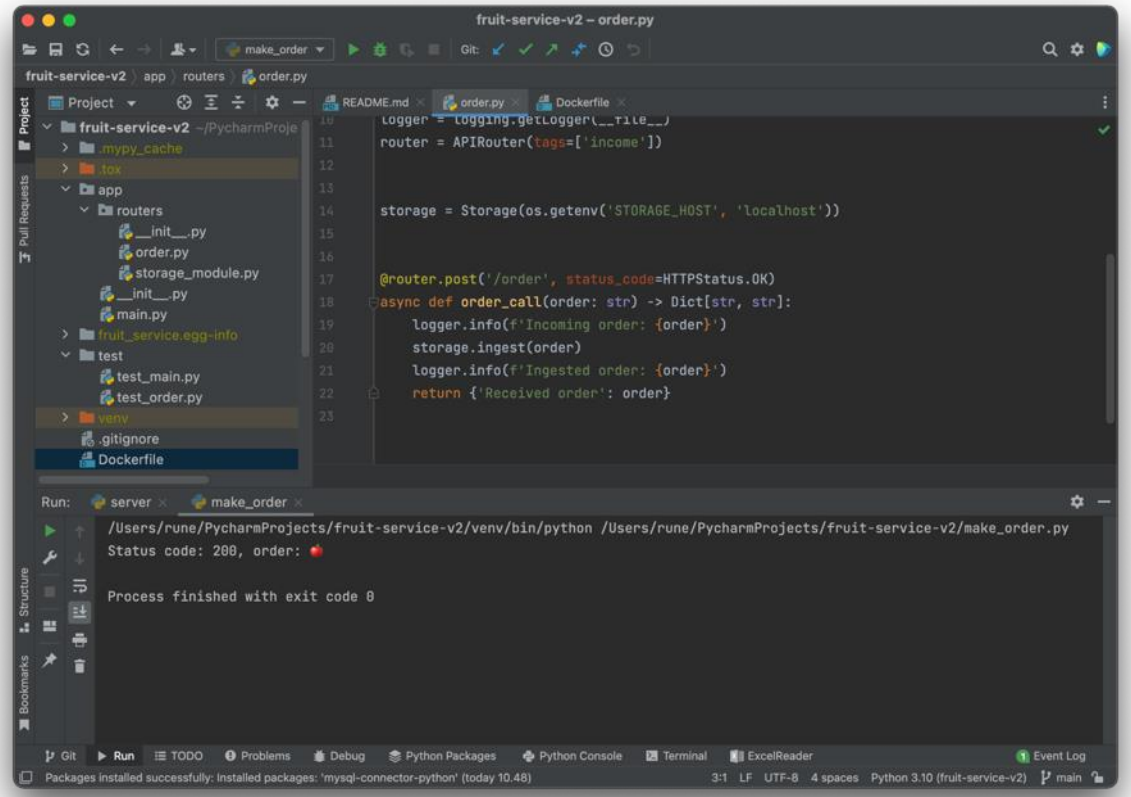
Run **docker run -dp 8000:80000 fruit-api-v2**

This should result in two containers running in Docker desktop.

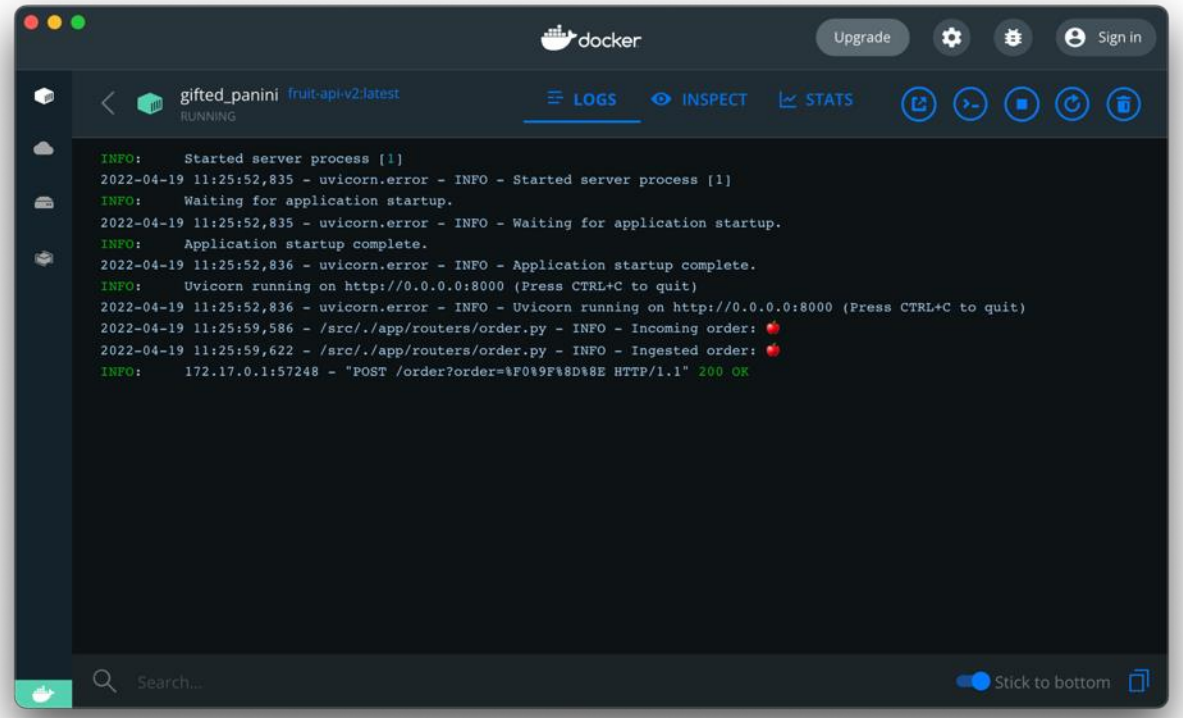


Your container names will most likely be different.

Try to run **make_order.py** from PyCharm.



If you then in Docker Desktop inspect the containers (by clicking on the container names).



You see everything works as charm. Also, you could check the Database in the terminal we used before. Notice that you will most likely not get the output of the emoji but rather a question mark.

Here is an example of how it could look like.

```
mysql> select * from ingress;
+---------------------+--------------+
| ingest_time         | ingest_value |
+---------------------+--------------+
| 2022-04-19 11:09:56 | apple        |
| 2022-04-19 11:26:00 | ?            |
+---------------------+--------------+
2 rows in set (0.00 sec)
```

Exercises

Exercise 05-00

- What happens if you stop and start the MySQL container (ingress-mysql)?
- Is the data lost?
- Is that expected?

Exercise 05-01

- How would you have two MySQL containers running simultaneously?
- HINT: We exposed our REST API on port 8000 and 8007 (end of Chapter 04).
- Try to do it.

Exercise 05-02

- How would the code need to be changed if the MySQL server was using port 3307
- HINT: This is not an easy question – but try to investigate it and think of it as good practice.

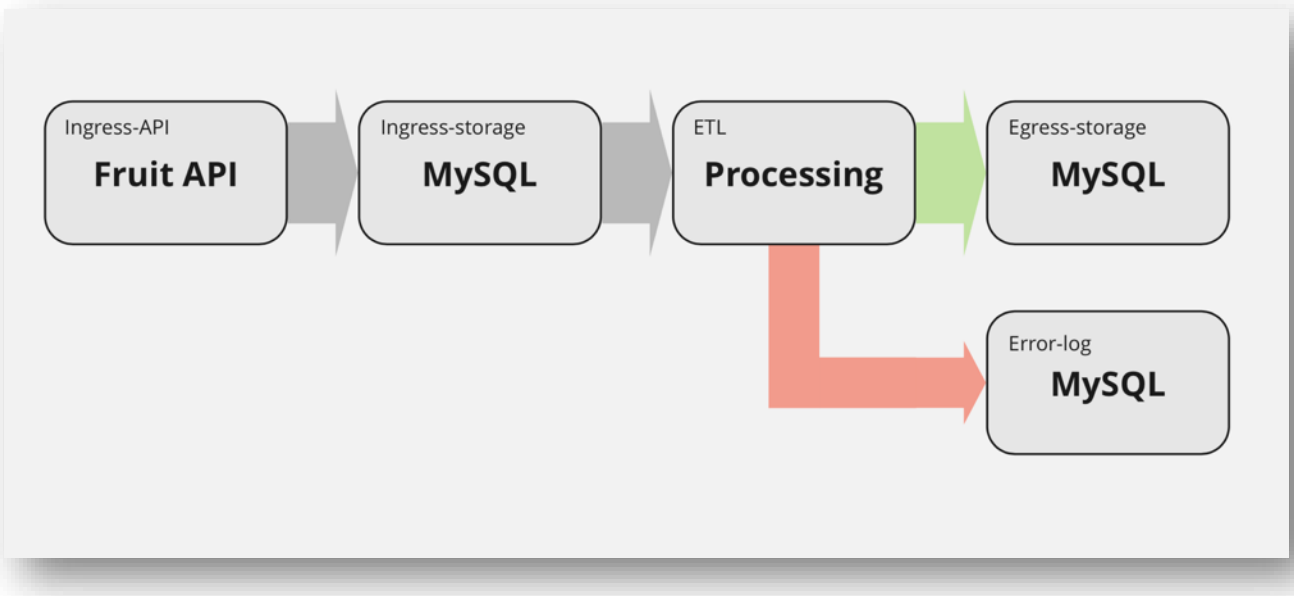
Summary

In this chapter we learned the following.

- How to create a Docker container with a MySQL server running.
- This includes setting it up with a clean database and table.
- How to have our Fruit Service API use the database.
- That there is a difference when running our API from PyCharm or from a Docker container, as it needs access to the host ports.
- You access the host system from a Docker container by using **host.docker.internal**
- How to get a CLI (terminal) connected a Docker container.

06 – Extract-Transform-Load

In this chapter we will add the Extract-Transform-Load (ETL) to the ecosystem.



The MySQL database will in our case be the same – you will not have 3 instances running.

- The Fruit-API is the same from last chapter
- The MySQL database is a new one with 3 tables, one for each entity in the above diagram: **Ingress-storage**, **Egress-storage**, and **Error-log**.
- The ETL is the main new component, which will be a batch process running as a cron-job.

Get ready for this chapter

You will need to do the following to get started with this chapter.

- Clone the following three repositories.
  - **git@github.com:LearnPythonWithRune/fruit-service-v2.git**
  - **git@github.com:LearnPythonWithRune/ingress-egress-mysql.git**
  - **git@github.com:LearnPythonWithRune/extract-transform-load.git**
- The first one you should already have from last chapter – the other two you need to clone.

Explore the MySQL Docker image

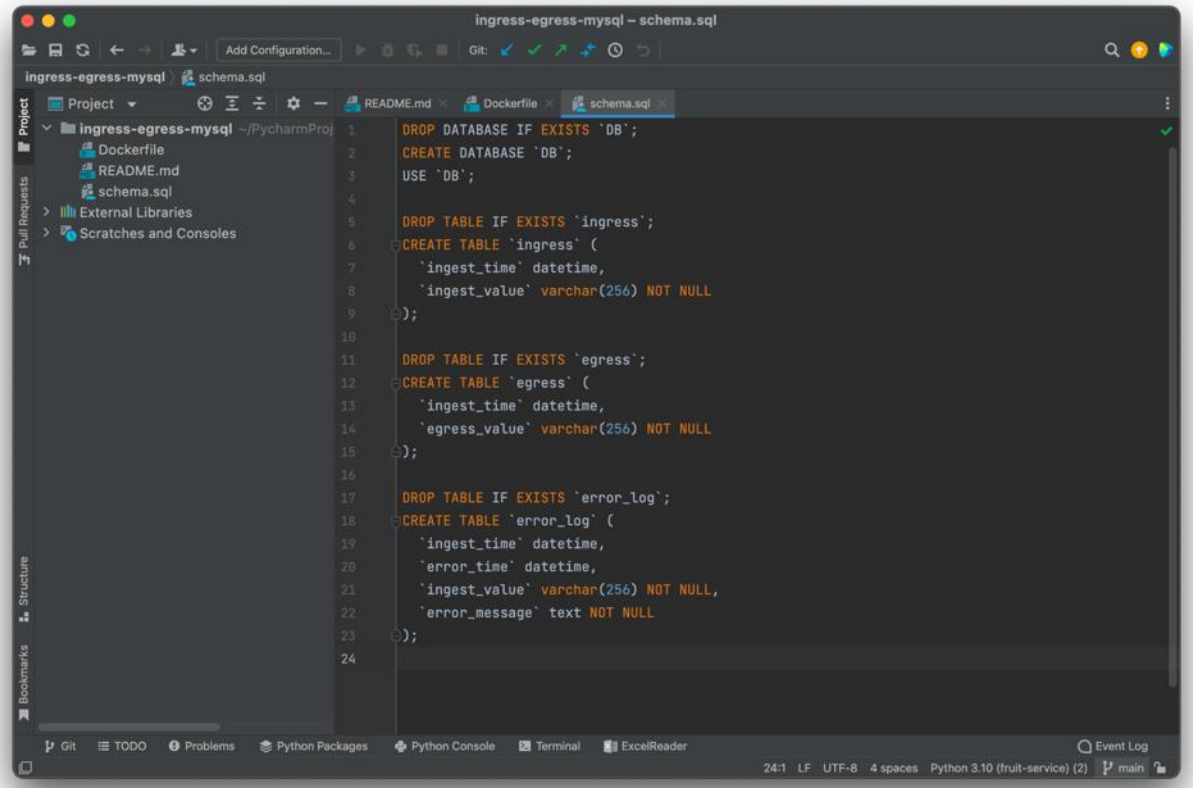
First let's explore the MySQL repository.

It consists of 3 files.

- **Dockerfile** The Docker file to build the image.
- **README.md** The read me file with description.

- **schema.sql** The database schema containing the SQL statements to create the database and tables.

The only new thing in this repository is the **schema.sql** file.



It creates the database DB and three tables **ingress**, **egress**, and **error_log**, representing the three entities in the diagram in the start of this chapter.

Build and run the MySQL service

What do we need to remember before we build a docker image?

- To have Docker Desktop running.

You are doing good my friend.

That is right, if the Docker Desktop is not running, then you cannot build an image.

To build the image you need to be in the terminal in PyCharm, this will have in the correct folder where the Dockerfile is.

Then run the command.

```
docker build -t ingress-egress-mysql .
```

This will build the image layer by layer.

When it is done you can run it in Docker.

```
docker run -dp 3306:3306 ingress-egress-mysql
```

This will start it detached and exposing the port 3306 on your local host.

If you didn't stop the **ingress-mysql** server from last chapter, then you will get into troubles.

You know why?

- The **ingress-mysql** Docker container exposes the MySQL server on the same port 3306.

You are absolutely correct and starting to get good at this.

Docker is exposing port 3306, hence, if you run both services, they both want to expose their services on the same port. This makes a conflict, as you cannot have two services exposed on the same port.

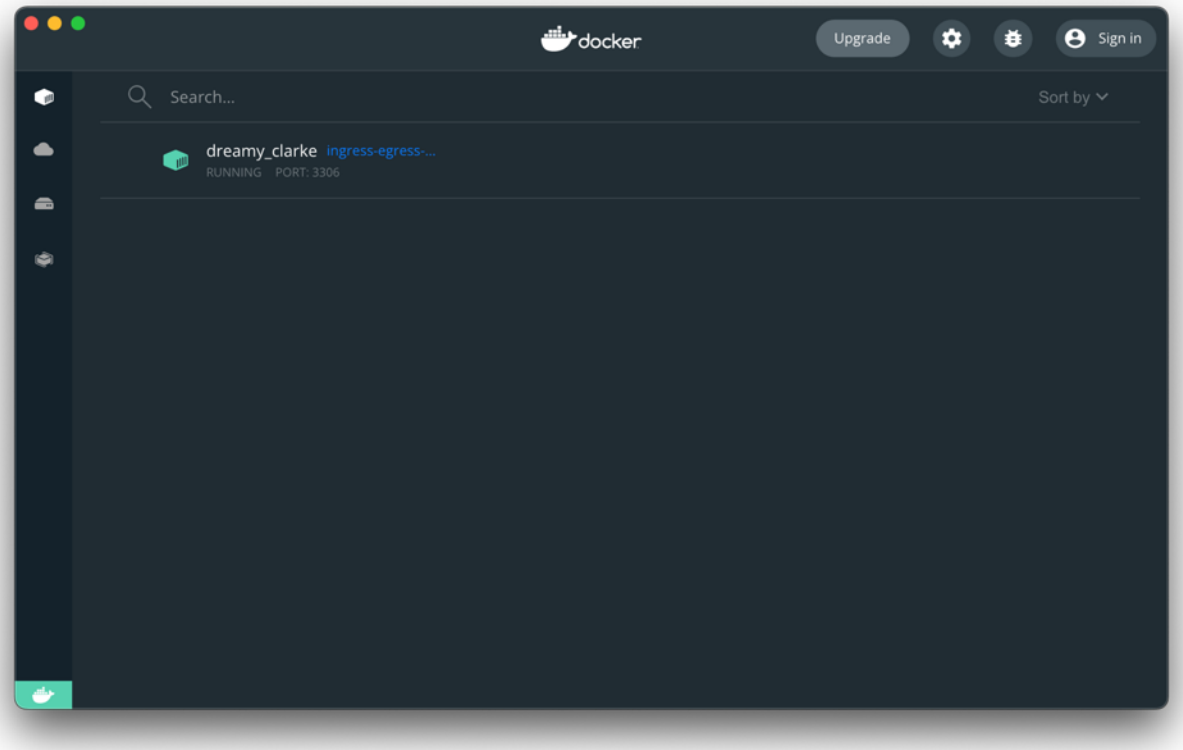
Can you work around this?

- Yes, you can expose one of them on another port, but let's leave that out of the scope now and only keep one service running at the time.

What does all this mean?

- If you had ingress-mysql running, stop it or delete it in Docker Desktop first.
- Then run the command again: **docker run -dp 3306:3306 ingress-egress-mysql**

Now you should have the container **ingress-egress-mysql** running.



That's it.

Now you are ready to start the fruit-service-v2.

Start your Fruit Service V2

You have done this before.

Also, did you shut the service down from last chapter? If not, it might still be running.

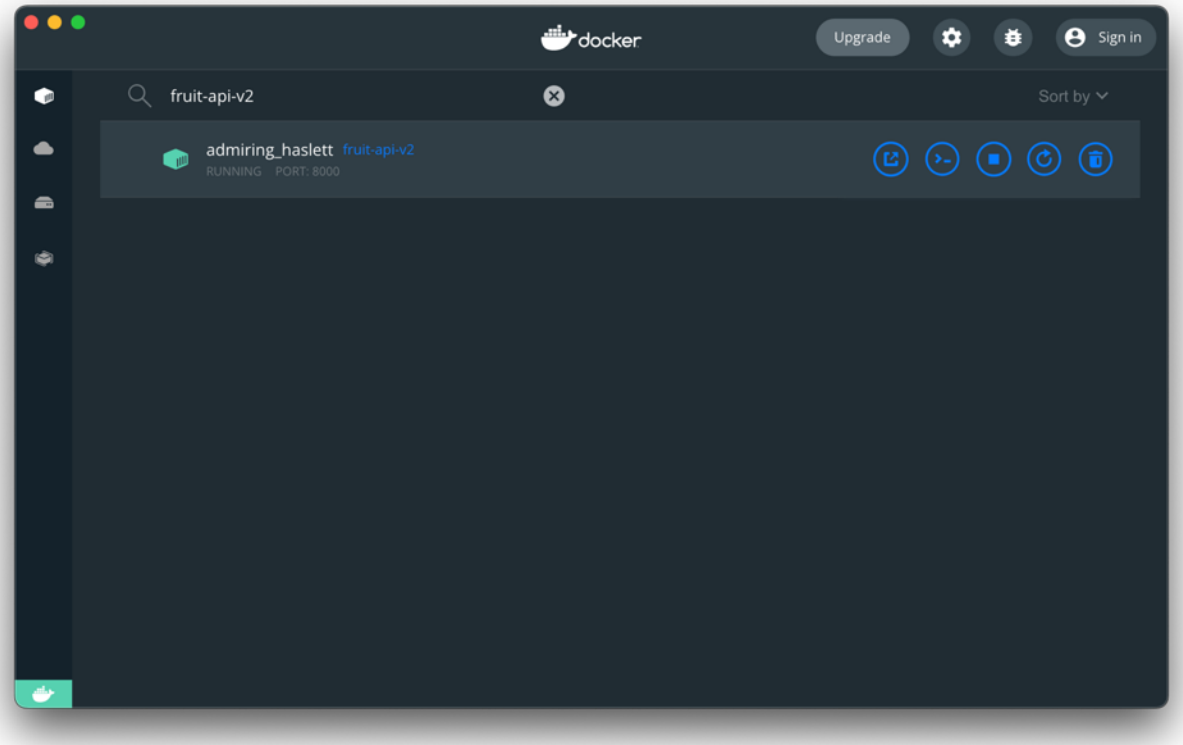
You can see the in Docker Desktop.

Also, the image should be built.

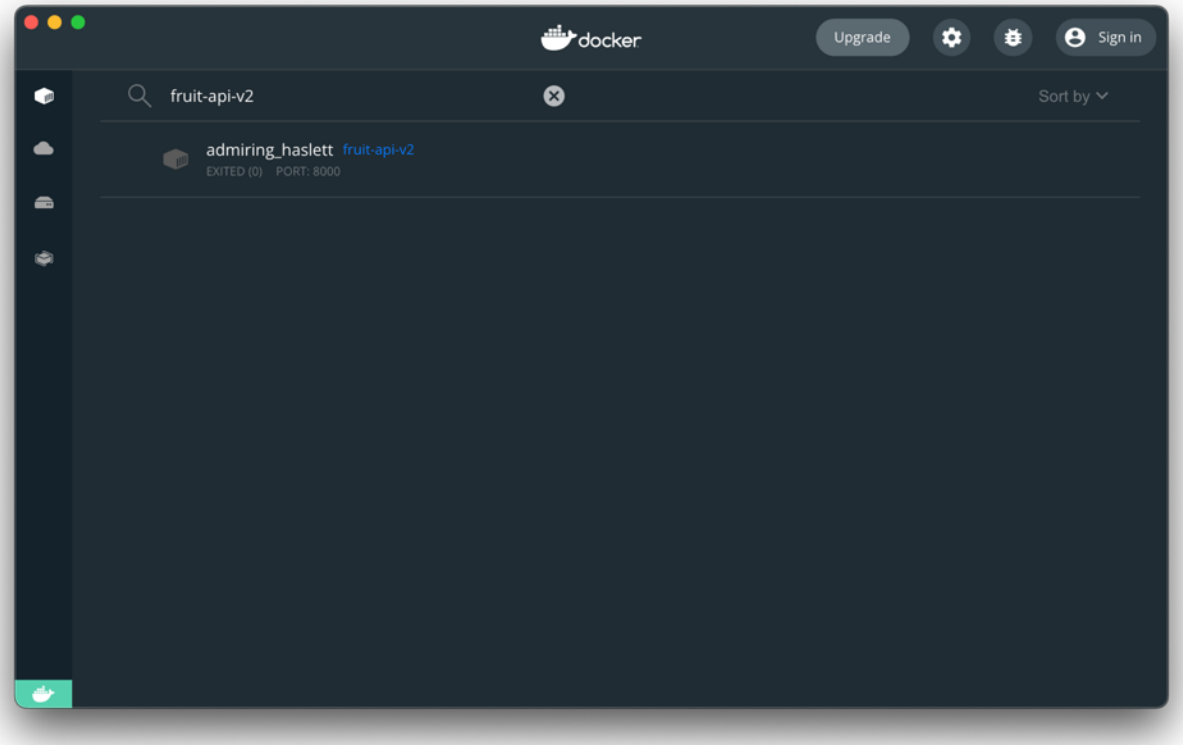
How to check all that?

Well, you can do that in Docker Desktop – let's first check if it running.

In the search you can write fruit-api-v2 (if needed, it will only filter down the view of containers in Docker).

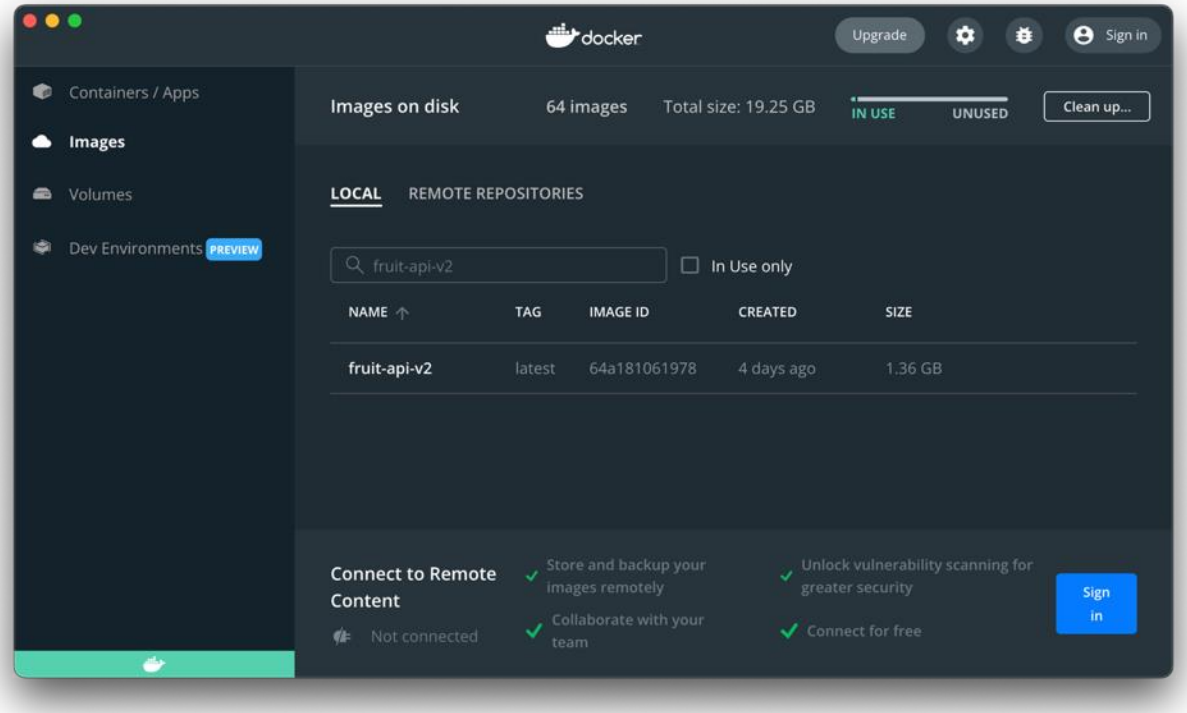


If nothing shows up or if no green (RUNNING) process is showing up, then it is not running. In the above image it is running.



Here is an example of it not running (EXITED).

To check if your Docker contains the image, you can search it in images.



You choose the Images on the left side and then search to see if it there (it should be from last chapter).

What to do if it is not there?

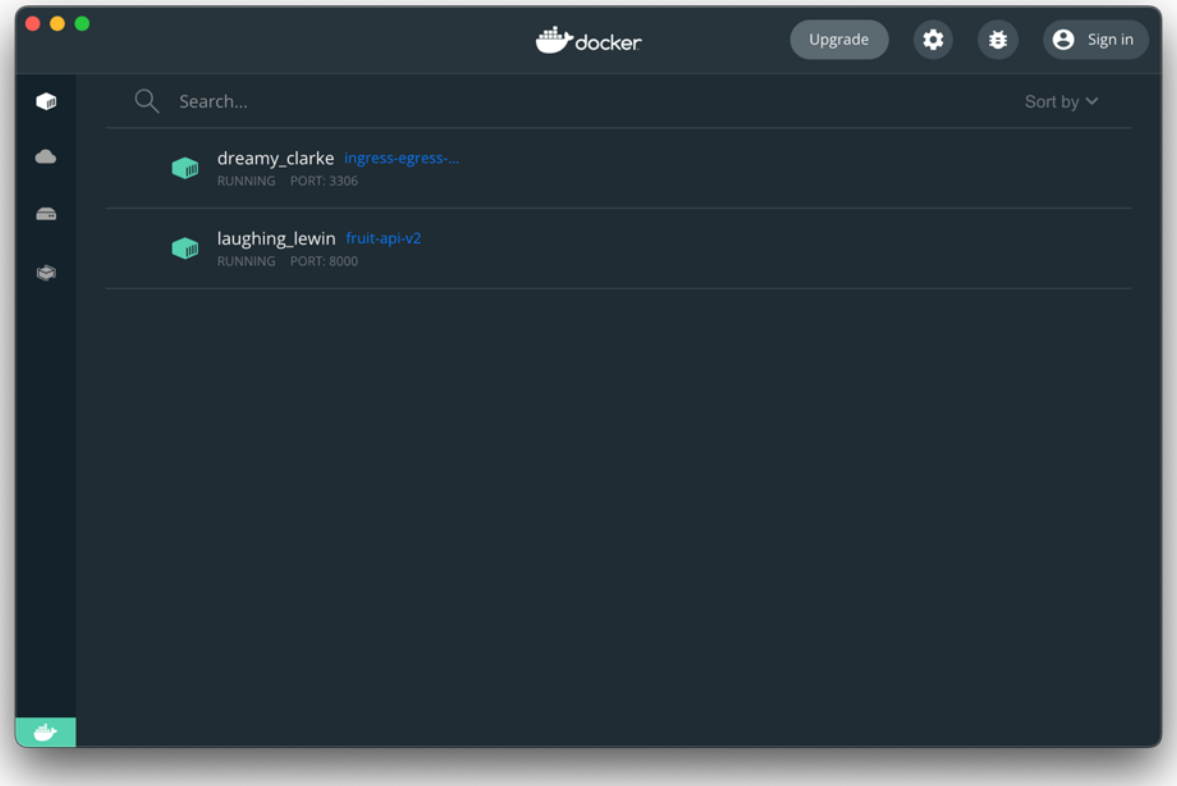
- You simply build it again with the command: **docker build -t fruit-api-v2 .**

Now you are ready to start **fruit-api-v2**.

To start it, you can do that from the terminal in PyCharm.

```
docker run -dp 8000:8000 fruit-api-v2
```

Now you should have both **ingress-egress-mysql** and **fruit-api-v2** running.



The names of your containers are most likely different.

You might be wondering?

- Does **fruit-api-v2** just work with **ingress-egress-mysql** without any changes?

Great question.

Remember, **fruit-api-v2** is using a MySQL database on port 3306 with a table called **ingress**. And this is exactly what **ingress-egress-mysql** has and some more – the some more is not making any troubles for **fruit-api-v2**.

Now we are ready to explore the **ETL**.

The Extract-Transform-Load (ETL) module

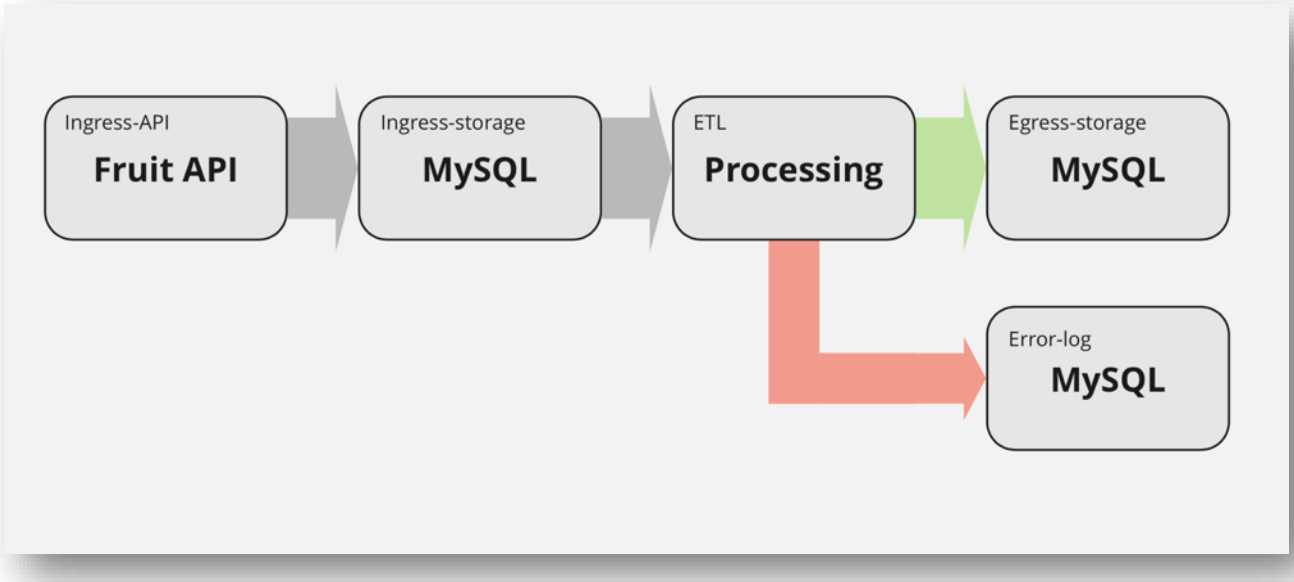
The repository consists of the following files.

- **crontab** This is setting up the cron jobs in the container (more about that in this chapter).
- **Dockerfile** The file building the Docker image.
- **etl.py** The ETL process.
- **list_database_contetn.py** Just to easily list the content of the database.
- **README.md** The read me file.
- **requirements.txt** The python libraries to be installed.
- **state.json** A configuration file used to keep track of how much is processed.

We will explore most of the files in this chapter – but let's start with **etl.py**, as this is the actual job we want to run.

What is it **etl.py** does?

- It checks if new data is available in the **ingress** table in the MySQL database.
- If so, it will **transform** it from the raw input format to a uniform format.
- Then it will load it into the **egress** table in the MySQL database.
- If any error occurs, like malformatted **ingress** data, it will load it into the **error_log** table in the MySQL database.



Why is this a good design?

- First of all, we want to keep raw data from the **ingress-api** (**fruit-api-v2**) to be able to re-process it with the **ETL**. If you need to get all data processed again, you can do that. This might be due to an error or similar. This ensure you always have the full dataset.
- This is good practice and decouples ingest and transformation of data, and makes it easier to find and locate error. Juniors would often make one monolith (one program) doing it all. This makes it difficult to locate where the error is.

- This also makes it easy to scale up and down on each module in a production system.

At first look at the etl.py file it can seem a bit difficult to figure out what it does.

But let's try to break it down.

We start at the bottom of the file.

```
def main():
    arg_parser = __init_argparse()
    args, _ = arg_parser.parse_known_args()
    host = args.host
    state_file = args.state_file
    logger.info(f'Config ({host=}, {state_file=})')

    process(host, state_file)


if __name__ == '__main__':
    main()
```

The **if__name__ == '__main__'** – part is to make sure that it will only run the **main()** function if this is the main file execute and it is not imported.

This is good practice to keep, which enables you to import this file without any code being executed.

All the if-statement does is calling main().

In main, we see it sets up a **__init_argparse()** and parses know arguments, gets the **host** and **state_file**, logs it and call **process** with **host** and **state_file**.

The only magic is **__init_argparse().**

```
def __init_argparse() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser()

    parser.add_argument('--host',
                        help='The MySQL host',
                        default='localhost')

    parser.add_argument('--state_file',
                        help='Json state file',
                        default='state.json')

    return parser
```

This is enables you to set command line arguments to the program, which we will need to set the host and specify where the state file is located.

Why is that needed?

- If we run the ETL program in PyCharm it will need the MySQL host to be **localhost**, while if it is running inside a Docker container, it should be **host.docker.internal**.
- Cronjobs are a bit funny, they do not run in the same environment and some path location, hence, we cannot directly use the same approach with setting an environment variable as we did with **fruit-api-v2** (this line in the Dockerfile: **ENV STORAGE_HOST=host.docker.internal**).

- This also requires us to specify the exact path to the state.json file, as the cronjob will not be running from our workdir.

Are there other ways to deal with this?

- Yes, but that is beyond the scope of this book as there are so many possibilities.
- Just as you know it could have been done in another way.

How to use it the command line arguments?

- They return default values, which are fitted for running **etl.py** in PyCharm.
- When we run it as a cronjob in a Docker container later, we will need to set command line arguments.
- Example: **python etl.py –host host.docker.internal –state_file /src/state.json**
- Actually, when you run it as cronjob you need to specify full paths for both Python and the the Python script.
- Example: **/usr/local/bin/python /src/etl.py –host host.docker.internal –state_file /src/state.json**
- But we will get back to that later.

Now **main()** calls **process(host, state_file)**.

```
def process(mysql_host: str, state_filename: str):
    with open(state_filename) as f:
        content = f.read()
        state = json.loads(content)

    connector = MySqlConnector(mysql_host)

    last_ingest = datetime.strptime(state['last_ingest'], '%Y-%m-%dT%H:%M:%S')
    logger.info(f'processing {last_ingest=}')

    # Extract
    rows = extract(connector, last_ingest)

    # Transform
    rows, error_rows = transform(rows)

    # Load
    error_rows, last_ingest = load(connector, rows, error_rows, last_ingest)
    load_error_log(connector, error_rows)

    # After a successful ETL we store the new state
    with open(state_filename, 'w') as f:
        f.write(
            json.dumps(
                {
                    'last_ingest': last_ingest.strftime('%Y-%m-%dT%H:%M:%S')
                },
                indent=2
            )
        )

    connector.close()
    logger.info(f'processed {len(rows)} rows {last_ingest=}')
```

This is the high-level ETL processing.

- It reads the **state_filename** and loads it as json data, extracts the a datetime stamp **last_ingest**.
- Connects to the database on **mysql_host** (**connector**).
- Then it **extracts** new data and gets new rows of data.
- It will **transform** the data and get correctly transformed **rows** and rows with error (**error_rows**).
- Then it **loads** the data **rows**, and ads possibly new errors to the log (**error_rows**) and get the last successful ingested data (**last_ingest**).
- Also, it loads the error logs (**error_rows**) to the database.
- Then it writes a new **state_filename** with the last ingested timestamp.
- It closes the connection to the database.
- Finally, it logs the processing of number of **rows** and **last_ingest** time.

This gives you a high-level understanding of how it works.

Here we will only explore what the transform function does, then you can explore and understand the rest of the code.

```
def transform(rows_to_process: List) -> Tuple[List, List]:
    transformation_map = {
        '🍏': 'apple',
        '🍐': 'pear',
        '🍌': 'banana'
    }

    transformed_rows = []
    error_log = []
    for ingest_time, ingest_value in rows_to_process:
        try:
            # Here we should do our transformation
            egress_value = transformation_map[ingest_value.strip()]

            transformed_rows.append((ingest_time, egress_value))
        except Exception as e:
            error_msg = f'{type(e).__name__}: {e}'.replace('\'', '"')
            error_log.append((ingest_time, datetime.utcnow(), ingest_value,
error_msg))
    return transformed_rows, error_log
```

This is the core of what we want to do in the ETL – the transform part.

It takes a list of **rows_to_process** and returns the transformed list (**transformed_rows**) and an **error_log**.

What we want is to map **emojis** to text.

To achieve that we have a **transformation_map**, a dictionary with **key-value** pairs of **emojis-text**. This makes the transformation easy.

It does that by iterating over the **rows_to_process** of the **ingest_time** and **ingest_value** and tries to use the map to get the **egress_value**.

This is appended to the **transformed_rows**.

If an exception occurs, we add that with the exception text and error time to the **error_log**.

Are you ready to try this?

Good. Let's first try it in PyCharm.

- Ensure that you have **ingress-egress-mysql** and **fruit-api-v2** running in Docker.

Then run the **etl.py** file in PyCharm.

```
2022-04-23 09:17:43,456 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - Config (host='localhost', state_file='state.json')
2022-04-23 09:17:43,526 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - processing last_ingest=datetime.datetime(2022, 4, 19, 14,
18, 1)
2022-04-23 09:17:43,538 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - processed 0 rows last_ingest=datetime.datetime(2022, 4, 19,
14, 18, 1)
```

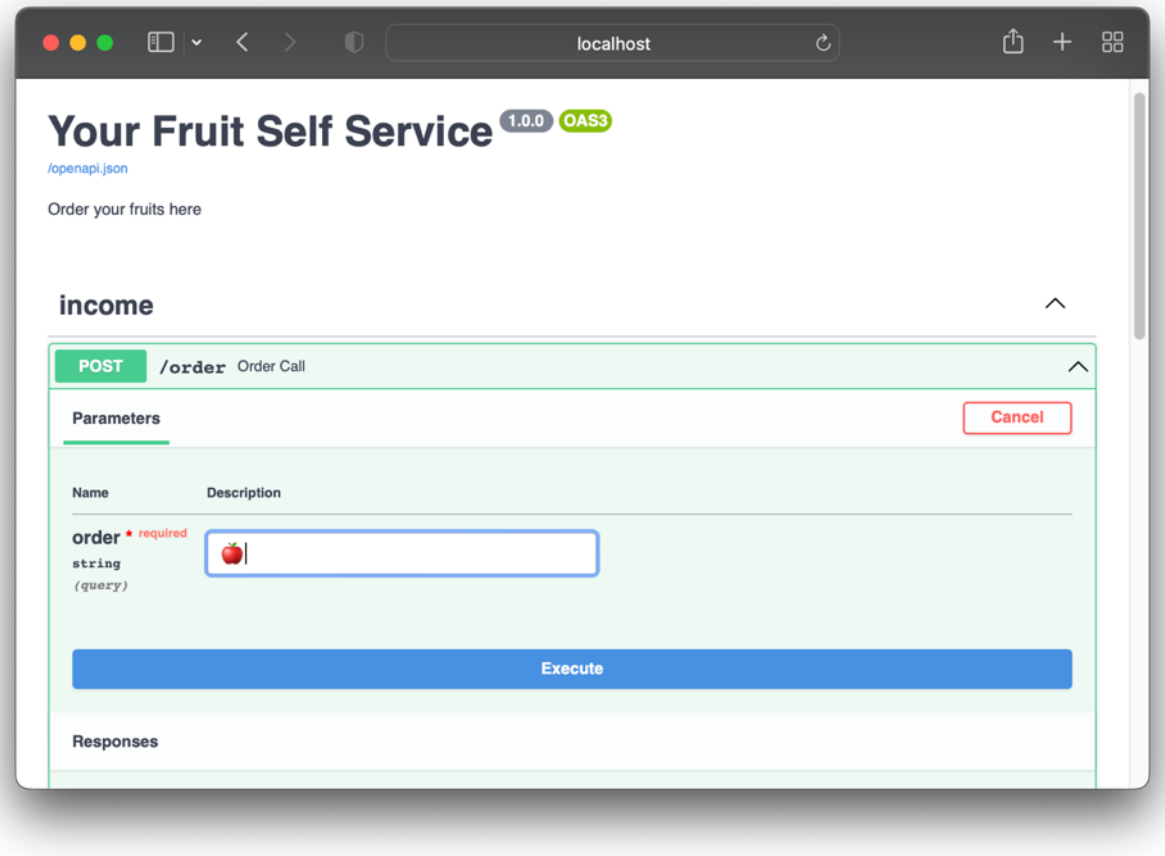
Most likely, you will get something similar.

No processed rows (processed 0 rows).

No worries – let's ingest something into the **fruit-api-v2**.

Go to **http://localhost:8000/docs#/income/order_call_order_post**

Click Try it out and insert order: 🍎



Click execute.

Then run **etl.py** in PyCharm again.

```
2022-04-23 09:23:11,425 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - Config (host='localhost', state_file='state.json')
2022-04-23 09:23:11,496 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - processing last_ingest=datetime.datetime(2022, 4, 19, 14,
18, 1)
2022-04-23 09:23:11,507 - /Users/rune/PycharmProjects/extract-transform-
load/etl.py - INFO - processed 1 rows last_ingest=datetime.datetime(2022, 4, 23,
7, 22, 46)
```

It processed one row.

Try the same with order: apple

What happens?

- Yes, it processed 0 rows.
- Why? Because it is not an emoji and is not transformed. It is added to the **error_log**.

You can check it in the MySQL Docker container like we did in last chapter or you can run the scripts added to list it.

Here we just run **list_database_content.py** in PyCharm to get the table content.

```
ingress:
  (datetime.datetime(2022, 4, 23, 7, 22, 46), '🍏')
  (datetime.datetime(2022, 4, 23, 7, 25, 48), 'apple')
egress:
  (datetime.datetime(2022, 4, 23, 7, 22, 46), 'apple')
error_log:
  (datetime.datetime(2022, 4, 23, 7, 25, 48), datetime.datetime(2022, 4, 23, 7,
25, 53), 'apple', 'KeyError: "apple"')
```

This shows the content of the tables.

- **ingress**: The two ingested values: '🍎' and 'apple'
- **egress**: The successfully transformed values (the emoji to 'apple')
- **error_log**: The 'apple' that could not be transformed as it was not an emoji.

Now we are ready to learn about cronjobs.

What is a cronjob?

First of all, what is a cronjob?

- A cronjob is a time-based job scheduler in Unix or Unix-like computer operating systems. You can use cron to schedule jobs, i.e. to execute commands or shell scripts at specified times, dates, or intervals.

This is what we want in our system.

We want to schedule our ETL to run in specific intervals.

But wait a minute – you might think! It says, Unix or Unix-like computer.

No worries, we will do this in a Docker container with Linux, which is a Unix-like computer (or container thanks to Docker).

This means, you do not need to install anything to get it running, it will happen inside a Docker container, which we will build.

This way we can have a Docker container running, which will run the cronjob on an interval.

How to setup our cronjob in Docker

Before we take a look at the Dockerfile I want to emphasize one thing.

In a production environment you would have Kubernetes run the cronjob and not run a cronjob inside a Docker container. We only do it here to keep things simple and fun and keep to the tools a developer would use. And no, a developer would not use Kubernetes on her local machine.

That said, you could set the cronjob to run from your computer – but it is system specific. This is the power of Docker, where you can setup the cronjob inside it.

I just want to be sure you understand, running a cronjob in a Docker container is not how you would do it in a production setup.

Perfect – enough talking.

Let's take a look at the Dockerfile.

```
FROM python:3.10-bullseye

RUN apt-get update && apt-get -y install cron
COPY requirements.txt .
RUN pip install -r requirements.txt

RUN mkdir /src

COPY crontab /etc/cron.d/crontab
RUN chmod 0644 /etc/cron.d/crontab
RUN /usr/bin/crontab /etc/cron.d/crontab
COPY . /src

WORKDIR /src

CMD ["cron", "-f"]
```

It does the following.

- Takes the python:3-10 image.
- Installs cron to enable to run cronjobs.
- Then copy and installs **requirements.txt**.
- Makes the **/src** directory.
- Copy a file **crontab** to host, changes permissions and runs it.
- Then it copies files to **/src**.
- Finally, it starts **cron** with flag **-f**.

Said differently, it creates a Docker image with python and running the **cronjob** defined in the file **crontab**.

Let's take a look at **crontab** file.

```
# run python script every 5 minutes
*/5 * * * * /usr/local/bin/python /src/etl.py --host host.docker.internal --
state_file /src/state.json > /proc/1/fd/1 2>/proc/1/fd/2
```

It will execute the command **/usr/local/bin/python /src/etl.py --host host.docker.internal --state_file /src/state.json** every 5 minutes.

Let's break it down.

- The first line with **#** is just a comment.
- ***/5 * * * ***: is the interval schedule. Here we run it every 5 minutes, which most likely is a bit overkill (too often) for the purpose, but this is not to sit and wait forever to have it run. Hence, if you made a production system, you might want to run it every hour or similar. To understand how to make a different schedule, you can use pages like this one to figure out the format: **https://crontab.guru**
- After that we have the command line.
    - The first part is **/usr/local/bin/python /src/etl.py --host host.docker.internal --state_file /src/state.json –** where we add the full path of everything, as it is run as cronjob and is not aware of location of python, the python script, the state-file, etc.
    - Then this part: **> /proc/1/fd/1 2>/proc/1/fd/2** is to forward **stdout** and **stderr** of the program to the **stdout** and **stderr** the container can see – otherwise, we would not be able to see any output of the program.

Build the image run the following command in the PyCharm terminal.

```
docker build -t etl .
```

To run it execute the following command in the PyCharm terminal.

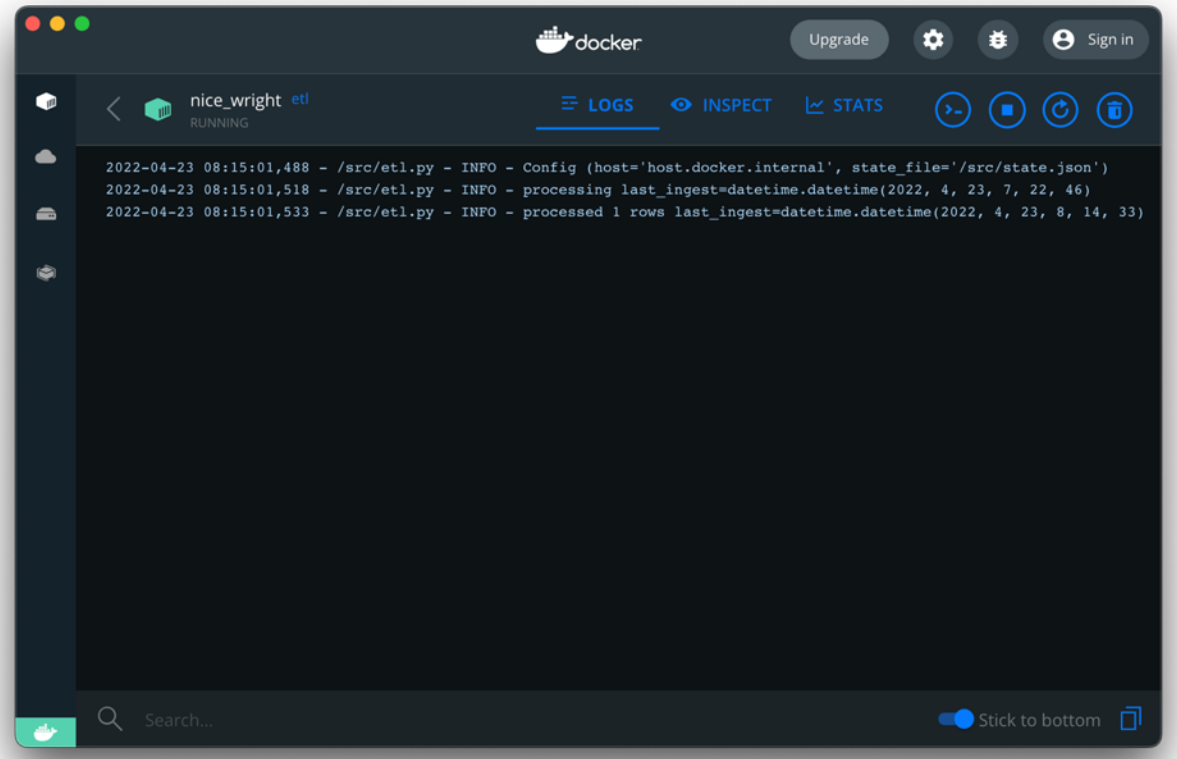
```
docker run -d etl
```

Then every 5 minutes it will run the **ETL** process.

You can see the log.

You can run the **make_order.py** script from **fruit-service-v2** or use the web interface to make a manual order.

Example of the log after processing 1 row.



You can also see the content of the databases by running **list_database_content.py** in **extract-transform-load** repository.

```
ingress:
  (datetime.datetime(2022, 4, 23, 7, 22, 46), '')
  (datetime.datetime(2022, 4, 23, 7, 25, 48), 'apple')
  (datetime.datetime(2022, 4, 23, 8, 14, 33), '')
egress:
  (datetime.datetime(2022, 4, 23, 7, 22, 46), 'apple')
  (datetime.datetime(2022, 4, 23, 8, 14, 33), 'banana')
error_log:
  (datetime.datetime(2022, 4, 23, 7, 25, 48), datetime.datetime(2022, 4, 23, 7,
25, 53), 'apple', 'KeyError: "apple"')
```

Your view might look different depending on whether you ordered a banana or not.

Possible challenge with this setup and alternatives

For the most part this setup will be good for small setups.

There are a few challenges that you might encounter.

- If you run the cronjob often, like every minute. Then it might not finish before the next one starts. In this case you might have two cronjobs running, which will mess up state file **state.json** and process the same data multiple times.

- This can be avoided by not schedule it too often, or you can use **flock** in your command line in crontab file. It works by creating a lockfile and only execute if it is released. This ensures that you only run one instance of a cronjob and not more simultaneously.

Possible alternatives could be.

- Using a message queue like Kafka. This basically has a queue waiting for messages with payload, every time a new message comes, it will trigger some other processes to run. This could be the ETL process we created. This is a bit overkill, for this setup, as we don't expect to process a lot of messages.
- You could setup Kubernetes (K8S), which can make cronjobs ensuring no concurrent jobs running. This is actually a great idea to do – this will be ideal also if you only expect to run the job every hour. It will keep resources free and only run the container when it processes the ETL workflow. In the setup presented in this chapter, we have container running all the time and taking up resources from your host system. Why didn't we do it here? Well, Kubernetes is beyond the scope for most programmers and would be a DevOps taking care of setting up.

Hence, the way to deal with it here is not to have a too often schedule of the cronjob.

Exercises

Exercise 06-00

- Change the transformation to successfully transform ingested values: **apple**, **banana**, and **pear**.
    - HINT: Add them to the **transformation_map** dictionary in file **etl.py**.

Exercise 06-01

- Change the transformation to successfully transform input of multiple emojies.
    - Example: order: '🍌🍎🍐' should result in three orders in egress: 'banana', 'apple', and 'pear'.
    - HINT: You can iterate over the string **ingest_value** in **transform** function in **etl.py**.

Exercise 06-02

- Change the cron schedule to every hour.
    - HINT: Change it in the file **crontab** and build a new docker image, shut the old one down (if running), run the new docker image.
    - HINT: You can use link **https://crontab.guru/every-1-hour** to find the schedule format in **crontab** file.

Summary

In this chapter we learned the following.

- How to setup up a new MySQL database in Docker with additional tables.
- That as long as the MySQL database has the same database and table as fruit-api-v2 expects, then it will continue to work.

- How our design is easy to debug and can transform the data again, as we keep the ingested values in an ingress-storage.
- How the ETL process work.
- What a cronjob is.
- How to setup a cronjob on an interval schedule in a Docker container.
- Some considerations and alternatives to this setup.

07 – Grafana and how to monitor

Who cares about logs and monitoring?

Your system is running and has been running for 3 weeks now.

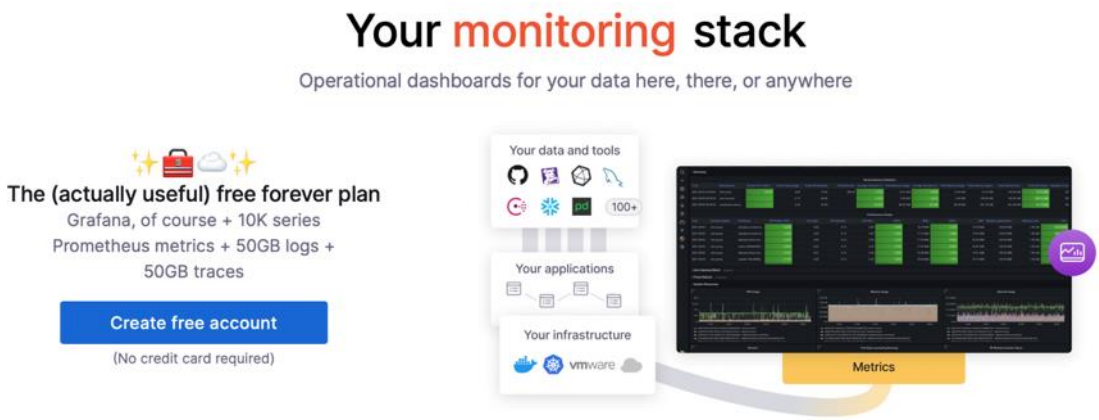
…or so you think.

- Is it healthy – meaning, does it do what you expect.
- Are all modules running as expected?
- How can you correct errors before your customers complain that there has been no new data for the 3 days?

This is where monitoring comes into the picture and Grafana is a great free tool to use for that.

In this chapter we will learn to setup some basic things and how to create data to Grafana.

What is Grafana and why do we care?



Well, this is not a commercial, but Grafana can help you with understanding how the health of your system is.

Actually, it can a lot more. We will keep our focus on the following.

- Learn about metrics and how to use them in Grafana. This includes how to generate metrics from our modules easy. This will be the subject of next chapter.
- How to use Grafana to see dashboards of data from your MySQL database. This helps you understand if everything is running and what is in your **error_log** and more.

It is often a DevOps task to setup great boards to monitor and make alerts based on the dashboards inside Grafana.

You can actually setup Grafana to send you messages to, e.g. Slack, so you can get notifications on your smartphone (if you install Slack) when there is an alert. Now that is smart.

Yes, that is right. You can setup alerts to warn you on your smartphone, when something is not as expected. This way, you don't have to check your ecosystem of running services (in Docker), if they are doing as expected.

How to start Grafana in Docker

You can start you own clean instance of Grafana, but we will use a Grafana instance in Docker with some connections and dashboards available.

You will find it in the repository: **git@github.com:LearnPythonWithRune/grafana-setup-example.git**

Clone it and then run the following command in the terminal in PyCharm.

```
docker run -d -p 3000:3000 -v ${PWD}/grafana-data:/var/lib/grafana
grafana/grafana
```

This should work on Linux, Mac, and Windows Powershell.

If you are using Windows command you should run:

```
docker run -d -p 3000:3000 -v %cd%/grafana-data:/var/lib/grafana grafana/grafana
```

It should result in Grafana running in a container and exposing the web UI on port 3000.

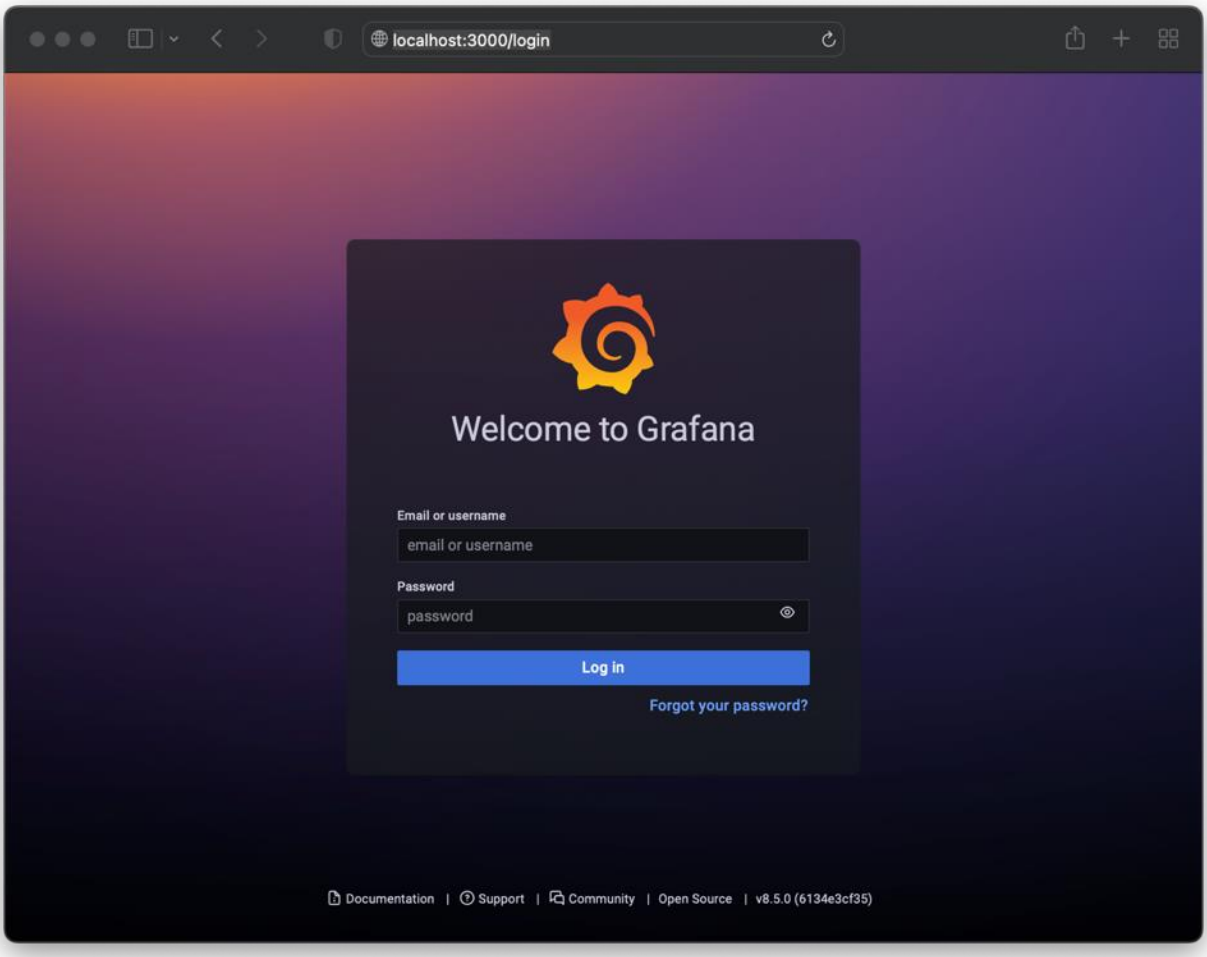
What the **-v** does in the command line, is to mount the volume to keep the configurations.

This Grafana configuration expects the MySQL server running from last chapter (**ingress-egress-mysql**). Make sure it runs in another Docker container.

Now login: **http://localhost:3000/login**

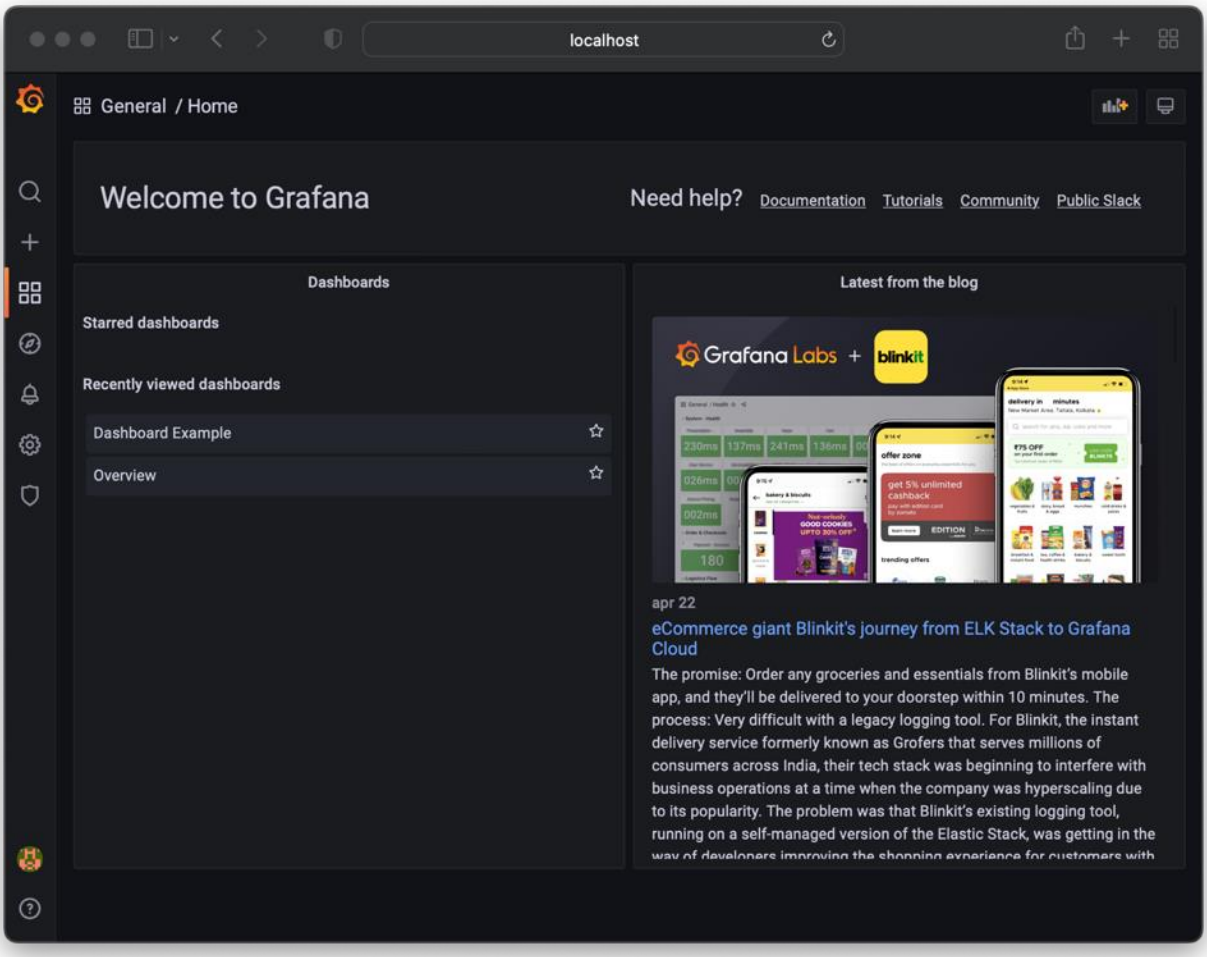
Use the email or username: **admin**

And password: **admin**

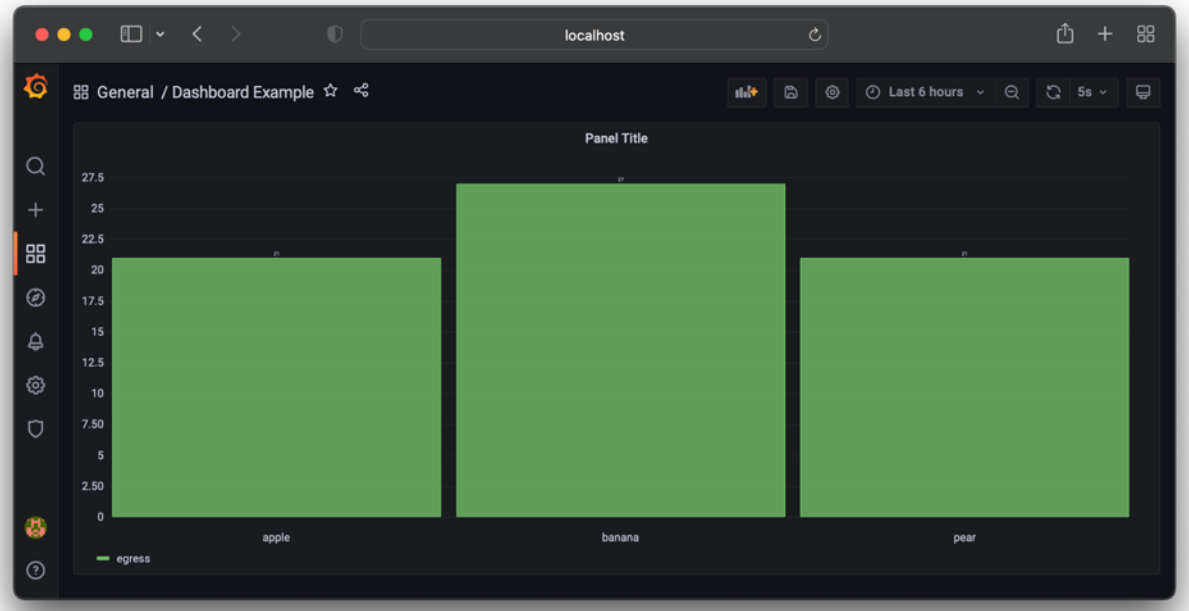


It will ask you to set a new password, but you may skip it by pressing **Skip** below **Submit**. If you choose to change the password, you should remember it yourself.

Then you should get to the this.



Let's try to look at the **Dashboard Example** by clicking it.

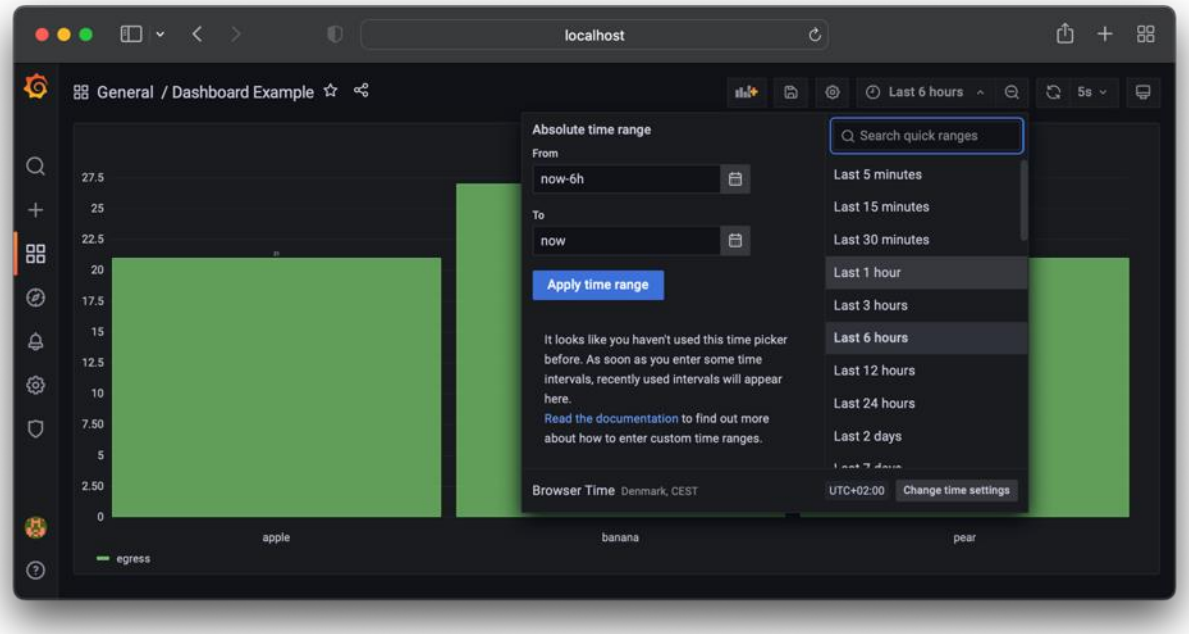


This shows how many orders have been done of each type, but with a twist.

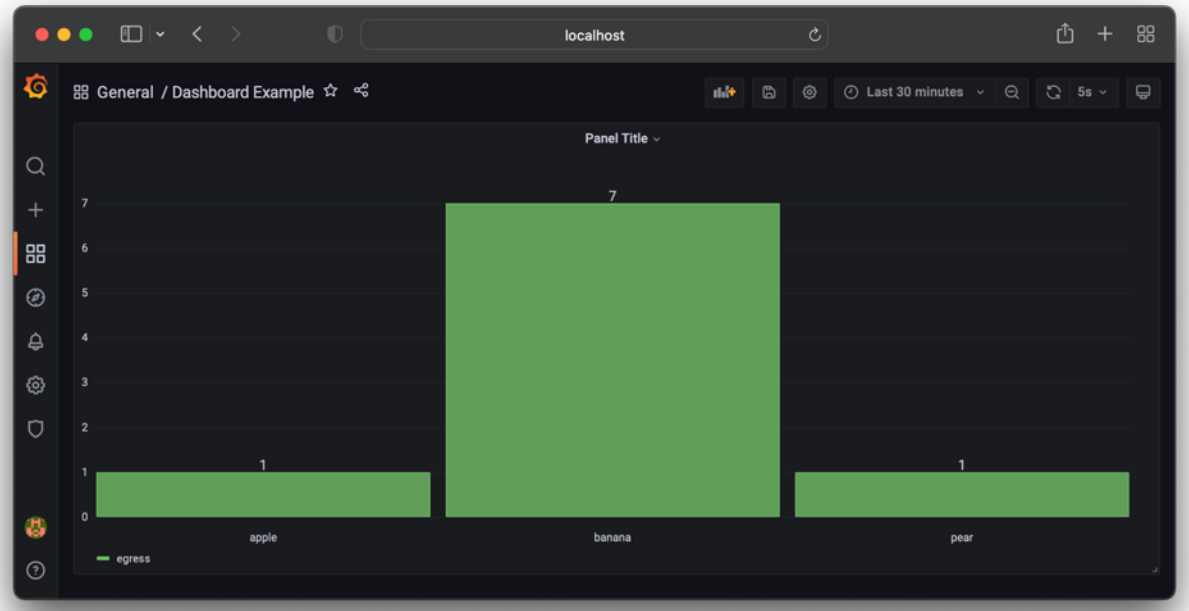
It shows how many orders have been done in the Last 6 hours and it will refresh the board every 5 seconds.

If you make more orders, they will appear once they have been processed and the board updates.

You can change the period by clicking the dropdown menu.



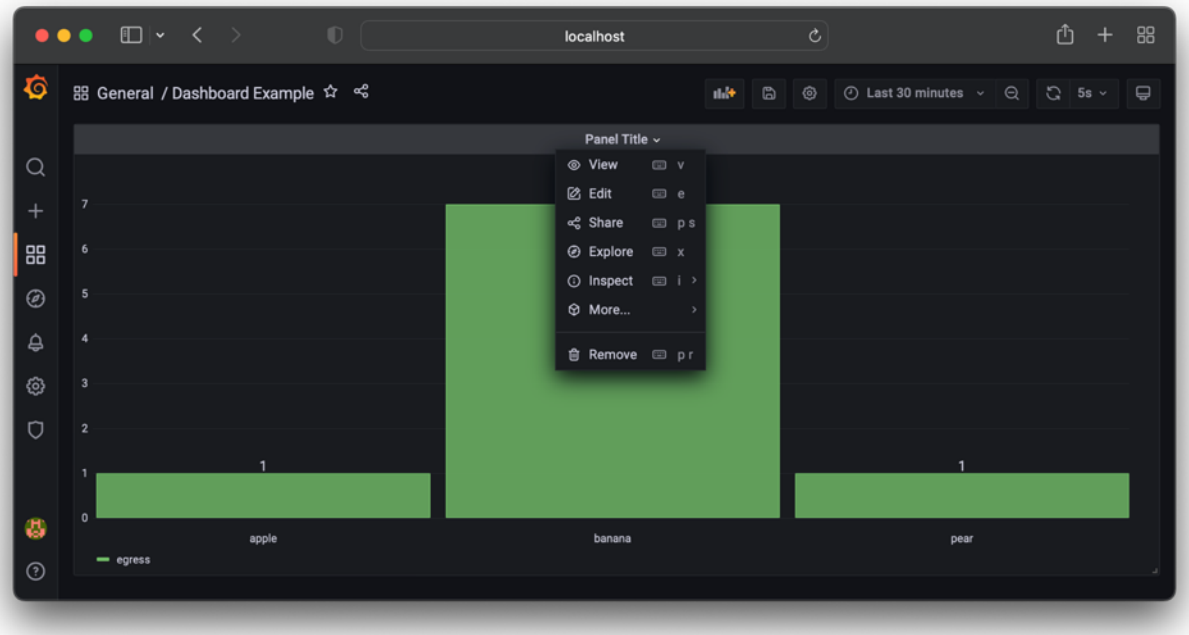
Here is an example of the last 30 minutes.



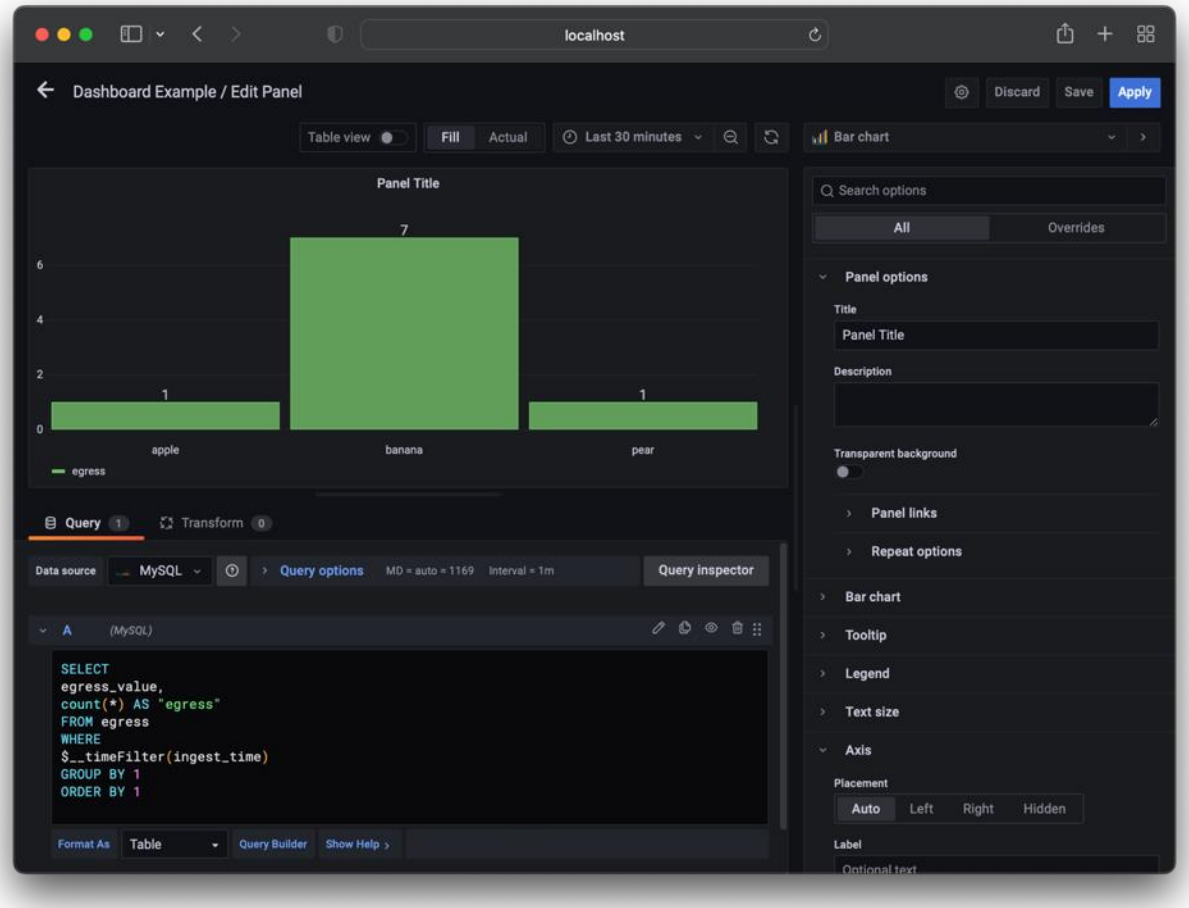
As you see, the orders in my system within the last 30 minutes has primarily been **bananas**.

I know, my customers have gone bananas.

You can see how it is generated by clicking the Panel Title dropdown and choose Edit.



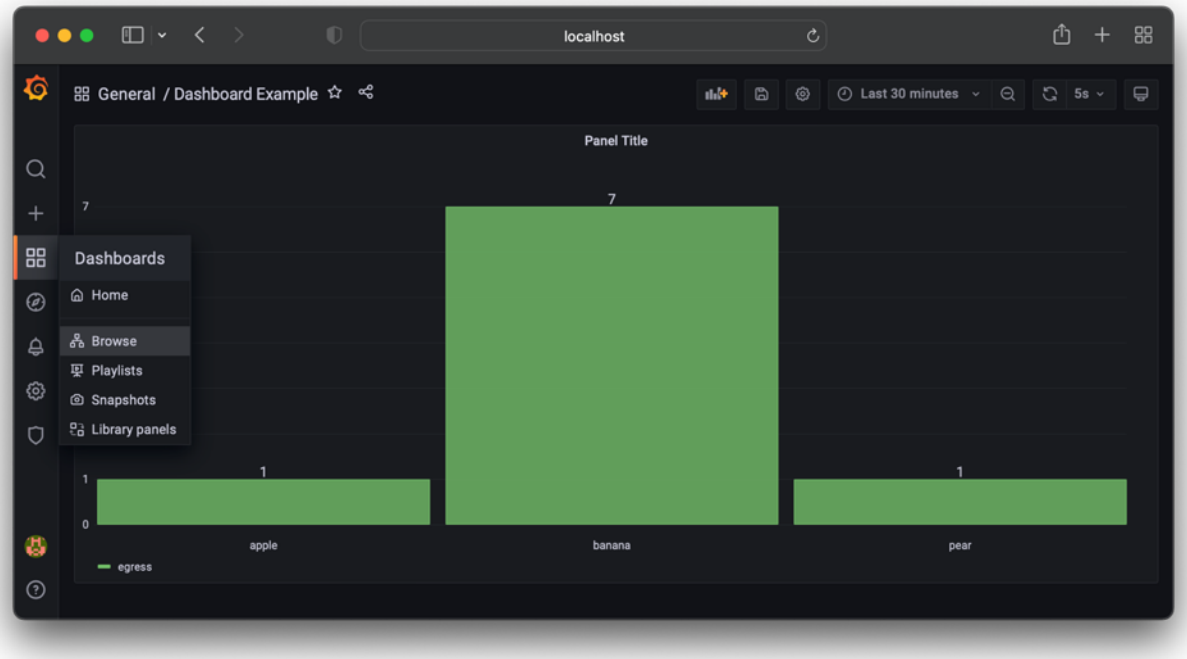
This will show something similar to this.



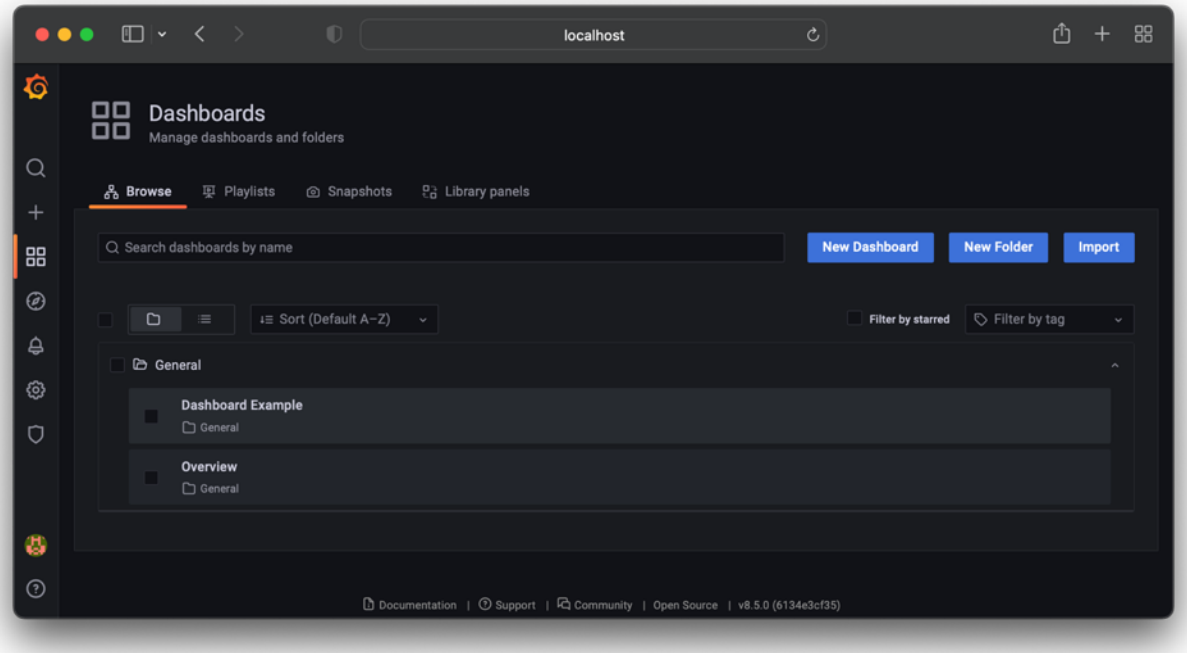
You can see the SQL syntax generating the above Bar chart.

Now let's take a look at another Dashboard.

Click Discard in the top right corner then from the menu in the left side find the board view.



Then you will get here.



Here choose **Overview** board.

This gives an overview of the tables in our MySQL database.

Again, some things to notice.

- It has a time view (here: Last 6 hours), which you can change.
- It updates automatically (here: Every 30 seconds), which you also can change.
- There are some underlying SQL queries that generate the 4 charts.
- First, is the **Ingested** chart shows how much data was ingested at a specific hour.
- The same in **Egress** chart contains a view of the transformed data. That is how much data was transformed in a specific hour.
- The **Error log** chart shows how many errors occurred at a specific time.
- Finally, the Data gives all the charts together to see how they differ.

As you probably see immediately, this gives great value of how the system is doing. This could in many cases be enough for a simple system like this.

When we make it a bit more advanced, it will be beneficial to have more insights into our ecosystem. Remember, this only tells you what happens in the database.

What else could be nice?

While this board in the database gives us great insights, it might not be enough.

What about the **fruit-api-v2**?

- What if that fails all the time?

Can you see that here?

- Not without any further knowledge.
- Maybe you know, that we get new orders every single minute, then you can see if data is not ingested.
- But if you don't know anything about how often the fruit-api-v2 is used, then it can be difficult.

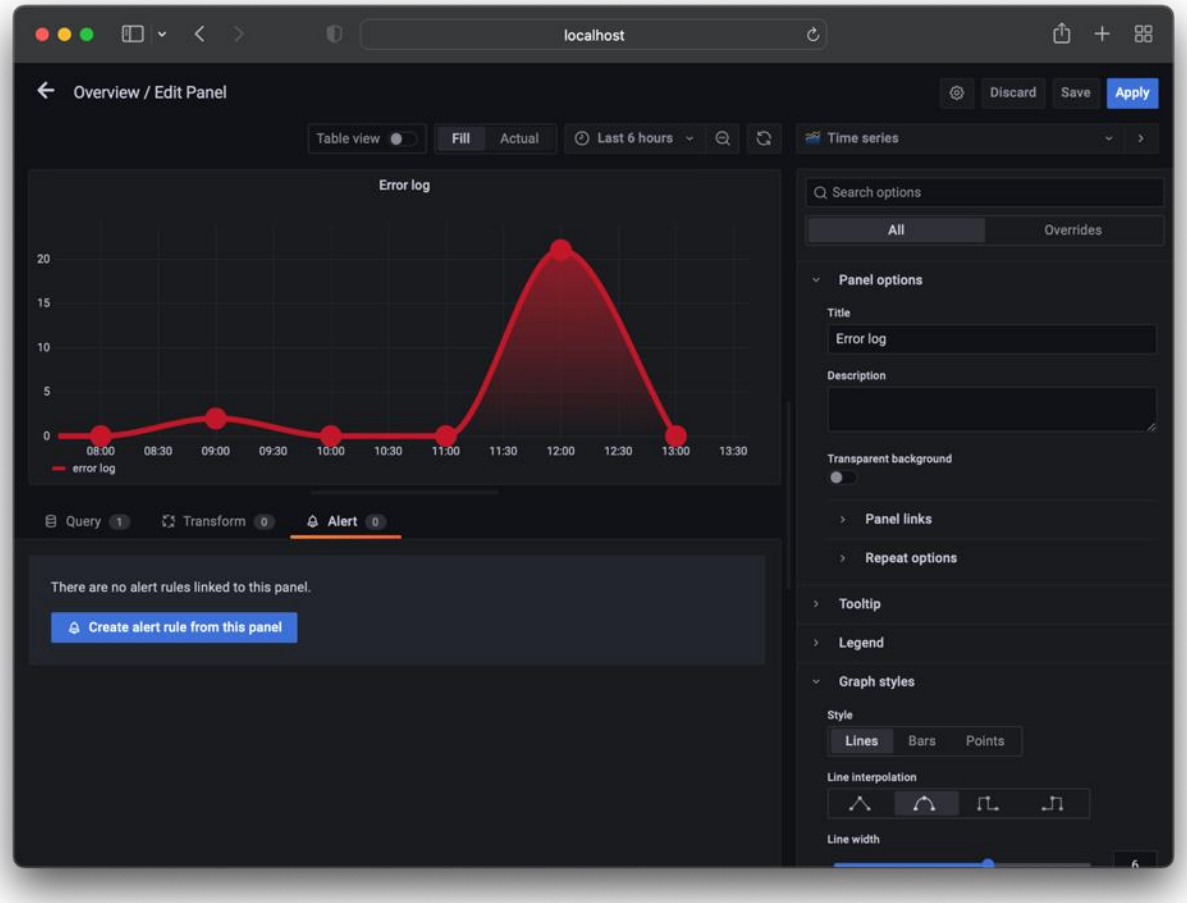
This is where metrics come into the picture.

This will be the subject of the next chapter.

Alerts in Grafana

As you probably already thought about, it would be nice to get an Alert if the Error log is high, as this indicates something is wrong.

This can be done by adding an **Alert** to a chart. Choose the **Error log** chart and edit it (from **Overview** dashboard).

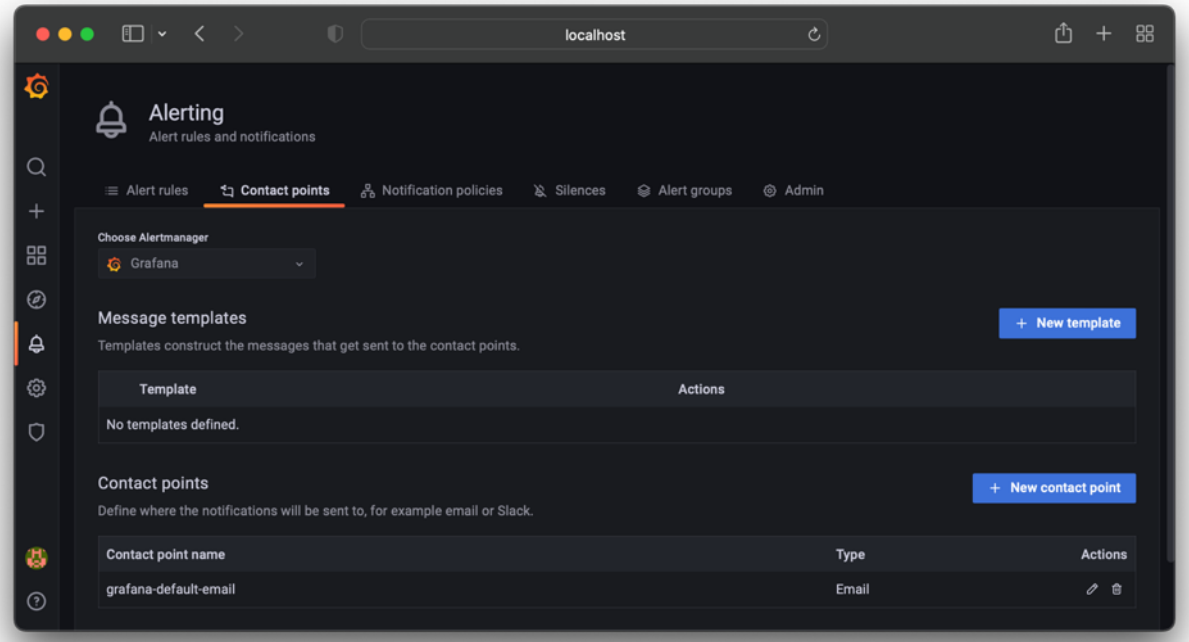


Then you can Create alert rule from this panel.

Follow the guide to setup the level of alert you want.

After this you need to setup how to be alerted.

- This can be by email.



But you can also be more advanced at setup with Slack.

- If you install Slack on your smartphone you can get instant alert on it.

This requires a bit more setup and installation but this guide can help you set it up.

https://medium.com/@_oleksii_/grafana-alerting-and-slack-notifications-3affe9d5f688

While Alerting is nice to have, it is not where I would have my focus. It is to learn the full benefit of Grafana, and use it with metrics which will open a new door of opportunities for you.

This is the subject of the next chapter.

Alerting and integrate the full logging from all Docker containers is a normal part of a production setup. But it is more the job of a DevOps, and you would normally not setup this locally when you develop code. Therefore, these aspects are beyond the scope of this book.

Exercises

Exercise 07-00

- Let your setup run for some time and make queries.
- See how the charts update.
- How does the update time and ETL time correspond to what you observe?

Exercise 07-01

- Change the colors in the General/Overview dashboard in Grafana.

- Make Ingested green and Egress yellow.

Exercise 07-02

- Adding a Dashboard can be the job of a developer depending on many factors.
- Add a new dashboard in Grafana.
- The Dashboard could show a time series over the different orders – assuming only banana, apple, and pear are being ordered.
- HINT: This requires an understanding of SQL – see the queries in the **Overview** board.

Summary

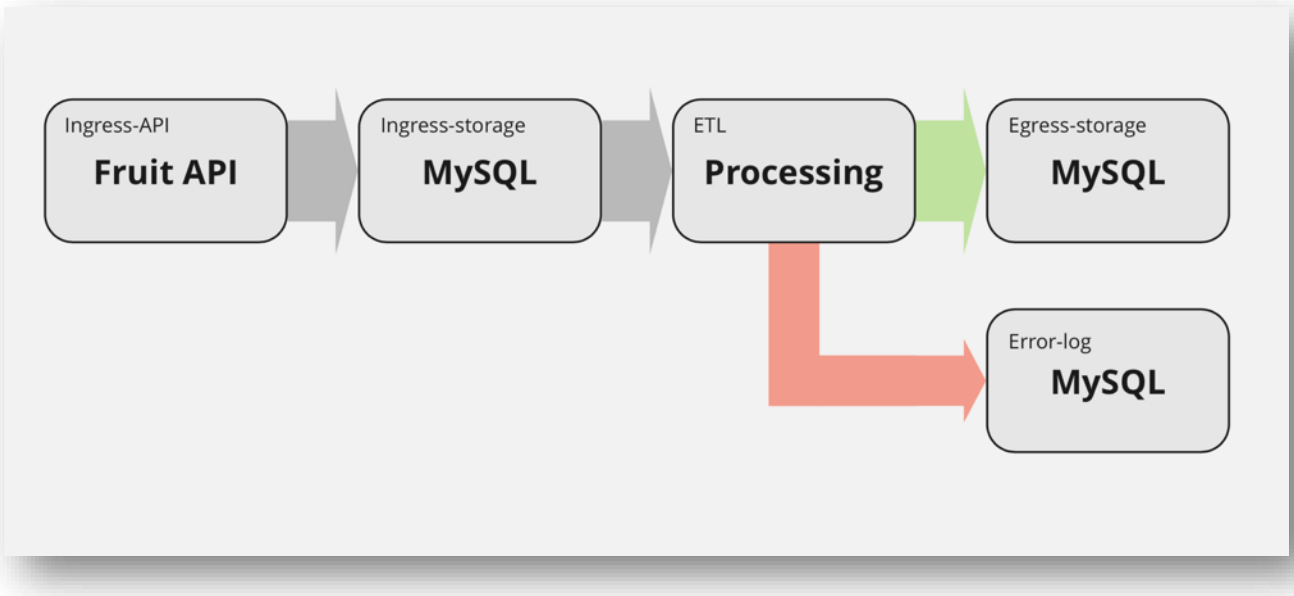
We have covered the following in this chapter.

- That Grafana can give you valuable insights into your ecosystem of services.
- How to setup a local instance of Grafana in Docker using local configuration.
- The idea of Dashboards in Grafana.
- Understanding how a Dashboard of our MySQL database gives insights into how our ecosystem is doing (is it healthy).
- What Alerts are and how to use them.
- That we need further insights to understand the full ecosystem as we have no insights into if our fruit-api-v2 is running correctly.

08 – Prometheus and Metrics

In this chapter you will learn about what Metrics are and how Prometheus can generate a lot of useful Metrics that we can monitor in Grafana.

Our Ecosystem, our insights, and what we need more



In last chapter we got the following insight into our ecosystem with Grafana.

- When data is processed in MySQL we see it.
- We can see error logs.
- Insight into all three MySQL tables.

But there is something we do not have insight into.

- What if our Ingress-API is failing?
- What if data is not put into our ingress-storage (MySQL)?

It can fail before the data is available in our setup in Grafana in last chapter.

Luckily, this can easily be achieved with simple (almost) out-of-the-box add-ons in our ecosystem.

In this chapter we will introduce the concept of metrics.

I think at first sight, metrics can be a bit confusing, and possibly difficult to see what value they give.

But let's dive into it and demystifying what metrics are and how they add value.

Demystifying Metrics

A metric is nothing but an observed value in our system.

Let's start with a simple example.

A metrics could be the number of calls to our API – a simple counter.

How can that be useful?

- If we call the endpoint and the counter doesn't increase – then we know something is not working as it should. It could be the counter, it could be the API service, or a combination. But we know something is wrong.

But let's translate the to an API service, which we have running. How can we use this counter metric?

Well, if the counter doesn't change the entire day, well, something might be wrong, right?

Actually, we don't know that from the counter itself.

But, say, if we usually get 500 requests per day and we had zero (according to the counter), then it is a good indicator something is not working as it should and we need to investigate it further.

This is a great lesson.

- Metrics in themselves, are not able to tell us the truth about if it is working or not (in general), but they are indicators.

So why bother?

- Because they are really good at indicating something is wrong, and if you know something about the system, you can setup alerts if a metric is outside the norm.

What does that mean?

- Take our counter, if the counter is at zero for 3 hours (as an example), it could fire an alert to you – that something might be wrong.

Yes, it could be a false positive (that nothing is wrong, it was just 3 hours without any call). You learn from that and adjust the alert accordingly, say, now it first fires an alert after 5 hours of no increase in our counter.

Types of Metrics

There are more types of metrics than only counters.

In general, I am not a fan of just listing a lot of stuff and assume the reader will remember it. But sometimes it makes sense to mention them. Don't get discouraged, if you don't remember them all afterwards. Many things first stick to your memory when you use them on a daily basis.

We will use Prometheus is an open-source systems monitoring and alerting toolkit – which can easily create useful metrics for us with Fast API (and other common frameworks).

Prometheus uses the following core metrics.

- **Counters** – as we discussed, they also emphasize it can only increase. That is, it counts only up.
- **Gauge** – is a measured value that can go up and down. It could be memory usage, temperature, or similar.
- **Histogram** – is samples of observed values. It is often used to represent a summary (or histogram) of the request duration.
- **Summary** – like histograms, but samples observations. Could be request duration with response sizes.

While this can seem like a mouthful, the real value and understanding comes by using them.

How to think of Metrics

Now we have some idea of what metrics are and a vague idea of how to use them in an alerting system to indicate something might be wrong.

Using metrics, you need to know what the normal picture looks like.

Then create alerts when the metrics are far enough from the norm.

What that "far enough" is, you need to figure out with experience and what the demands are. You don't want to be woken up at 3am because there have been no requests for 3 hours, if you normally only get requests in working hours.

Let's take two examples on how a metric can be used.

Counter

Let's say our Fruit API gets orders from local customers, and from 8am to 9pm it gets about 300-600 requests per hour.

In lowest hours it has been down on 20 requests, but never lower.

In this case an alert could go off if we have less than 10 requests in one hour between 8am and 9pm (to avoid alerts at night while you sleep and your sales are probably close to zero).

Depending on how critical this is, you could be more aggressive, and say, if there have been less than 5 requests within the last 15 minutes.

Again, this depends on the norm picture.

What it gives you, is that you know you have to investigate if something is wrong and you are losing orders from valuable customers.

Histogram

Let's say a normal request duration is 300 micro seconds.

Suddenly, your system request durations take up to 3-4 seconds. This can be an indication that some system is not doing that great.

There can be many reasons for that – it can be system resources are running low, too many requests, running out of disk, too little memory to handle all processing.

The bottom line is, that if the request duration increases consistently, you are most likely about to have a problem.

Hence, it is a good idea to have to have alerts when the request duration is above some level.

Caution on Alerts

Metrics can be addictive – just kidding – but once you understand them you suddenly can get a lot of ideas of metrics that could add value.

But it is not your goal to create as many metrics for any possible failure you can image.

Getting too many alerts (where most are not really valuable) will make you drown in them, and you stop to take them seriously.

On the other hand, if you have some really awkward alert you thought of, then it fires the first time 6 months later, and you actually forgot what value it was adding – then what is the value.

The point is – you need to balance the alerts.

Keep them simple and easy to understand.

Better a few simple ones you understand – then if you see the need for some later, then add them. Don't make up alerts you think can be useful.

Start simple.

Final Thoughts

A great way to think of alerts is in the following two concepts.

Leading and lagging indicators.

What is that.

- **Lagging indicator** – something is wrong now. It could be the counter that say is down on zero, it says, the system is not working as it should. This is critical, as it is already down, you are losing orders from hungry customers.
- **Leading indicators** – Something is about to go wrong. It could be the histogram that says request durations are too high. The system is still working, but it might be a question of time before it is down.

This is a really great way to think about metrics. Some give you insights into if the system is working now (lagging indicators), and some that the system might be on the edge to failure (leading indicators).

In the ideal world you just have the best leading indicators, and prevent all failures before they happen.

Well, I haven't seen it. You cannot prepare for all possible failures.

Therefore, you need both types of indicators in your monitoring system.

Adding Prometheus?

This is actually surprisingly easy to add some default Prometheus metrics to our Fruit API.

All you need to do is to clone the repository **git@github.com:LearnPythonWithRune/fruit-service-v3.git**

Then open the **app/main.py** file.

```
import logging
from http import HTTPStatus
from typing import Dict

from fastapi import FastAPI
from starlette_exporter import PrometheusMiddleware, handle_metrics

from .routers import order

logging.basicConfig(encoding='utf-8', level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__file__)

app = FastAPI(
    title='Your Fruit Self Service',
    version='1.0.0',
    description='Order your fruits here',
    root_path=''
)

app.add_middleware(
    PrometheusMiddleware,
    app_name=__name__,
    group_paths=True
)

app.add_route('/metrics', handle_metrics)
app.include_router(order.router)


@app.get('/', status_code=HTTPStatus.OK)
async def root() -> Dict[str, str]:
    """
    Endpoint for basic connectivity test.
    """
    logger.info('root called')
    return {'message': 'I am alive'}
```

What is new is the PrometheusMiddleware and handle_metrics in the import section.

Then we add_middleware in the code.

```
app.add_middleware(
    PrometheusMiddleware,
    app_name=__name__,
    group_paths=True
)
```

And adding a route with metrics.

```
app.add_route('/metrics', handle_metrics)
```

This is all we need in the code.

This will include a middleware to FastAPI, which will generate all the metrics. Also, it will expose the metrics on your web service /metrics.

That is, you can see the metrics generated once it is running on you host.

We will get back to that later.

To have Prometheus collecting the metrics, we will use a Docker image and a configuration file.

prometheus.yml (also new in the repository).

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    monitor: "app"

rule_files:

scrape_configs:
  - job_name: "prometheus"

    static_configs:
      - targets: ["host.docker.internal:8000"]
```

We will also get back to this file.

Starting our new setup

Adding Prometheus to our REST API is quite easy, as we just did.

To setup our ecosystem now is quite a lot of things.

- Ingress-Egress-MySQL
- Fruit API v3
- Extract-Transform-Load
- Prometheus
- Grafana-Setup-Example-v2

But in the next chapter we will learn how to set it up smarter. For now, we do it manually.

Let's start from an end.

- **Start Docker Desktop**
  - Otherwise the following commands will not work
  - If you have done the previous chapters, you should have most repositories and have built images for them in Docker – hence you only need to execute the Docker run part.
  - If it says **(new)** then you need to close, build and run.
- **ingress-egress-mysql**
  - Repository: **git@github.com:LearnPythonWithRune/ingress-egress-mysql.git**
  - Docker build: **docker build -t ingress-egress-mysql .**
  - Docker run: **docker run -dp 3306:3306 ingress-egress-mysql**
- **Fruit-service-v3 (new)**

    - Repository: **git@github.com:LearnPythonWithRune/fruit-service-v3.git**
    - Docker build: **docker build -t fruit-api-v3 .**
    - Docker run: **docker run -dp 8000:8000 fruit-api-v3**
- **extract-transform-load**
    - Repository: **git@github.com:LearnPythonWithRune/extract-transform-load.git**
    - Docker build: **docker build -t etl .**
    - Docker run: **docker run -d etl**
- **prometheus**
    - Repository: The configuration file is in the fruit-service-v3 repository. Hence, to run the docker run command, you need to be in that folder.
    - Docker run: **docker run -dp 9090:9090 -v ${PWD}/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus**
    - This will collect the latest image and setup config.
- **granfana- setup-example-v2 (new)**
    - Repository: **git@github.com:LearnPythonWithRune/grafana-setup-example-v2.git**
    - Docker run: **docker run -d -p 3000:3000 -v ${PWD}/grafana-data:/var/lib/grafana grafana/grafana**

Now that was a lot.

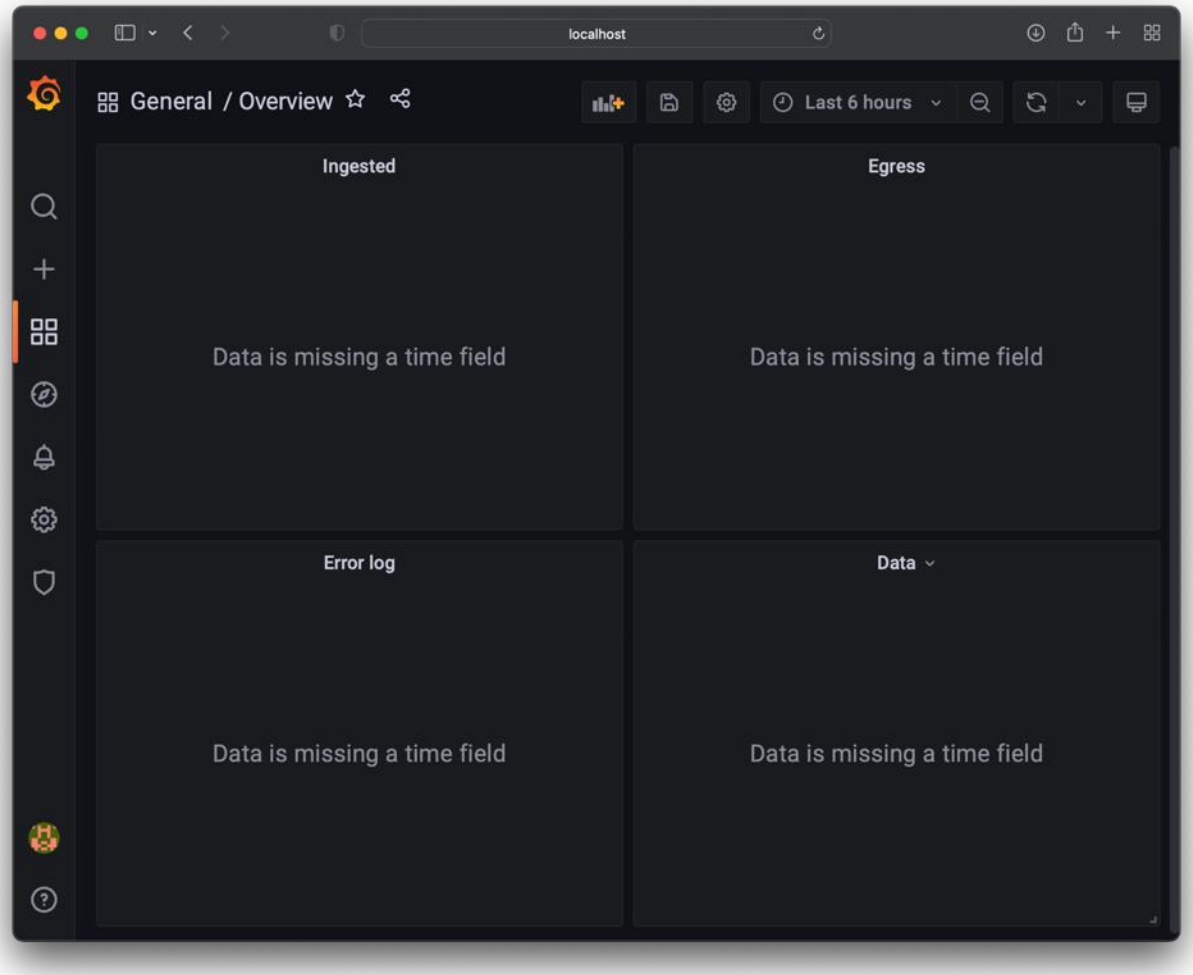
Let's look into Grafana.

Logging into Grafana

Go to **http://localhost:3000/**

If you check your simple Dashboards from last Chapter, there are still there.

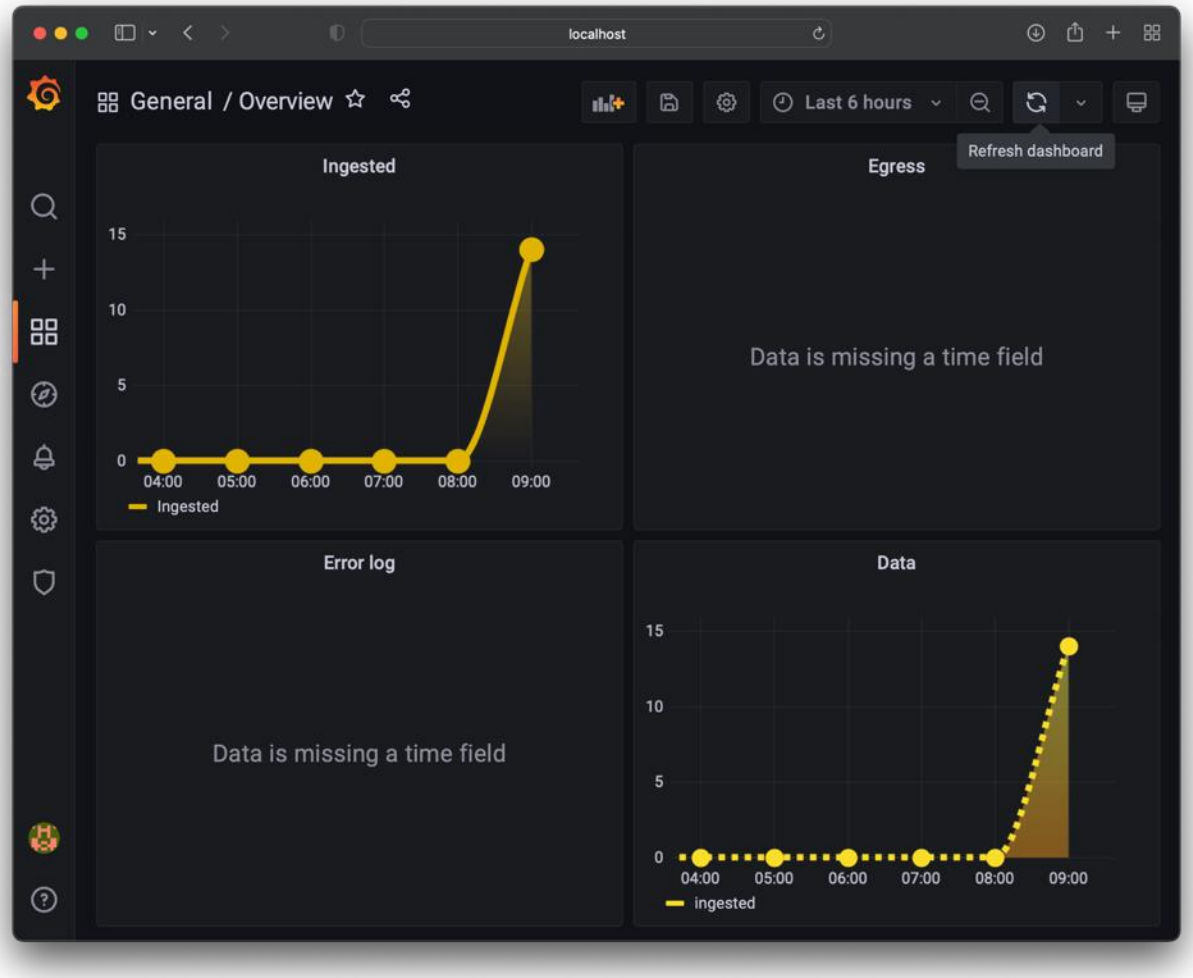
Most, likely they will be empty.



There is a good explanation for that. You didn't use your API yet.

You can do that by running **make_order.py** in fruit-service-v3 a few times.

Then you will get some data. First in Ingested and Data.

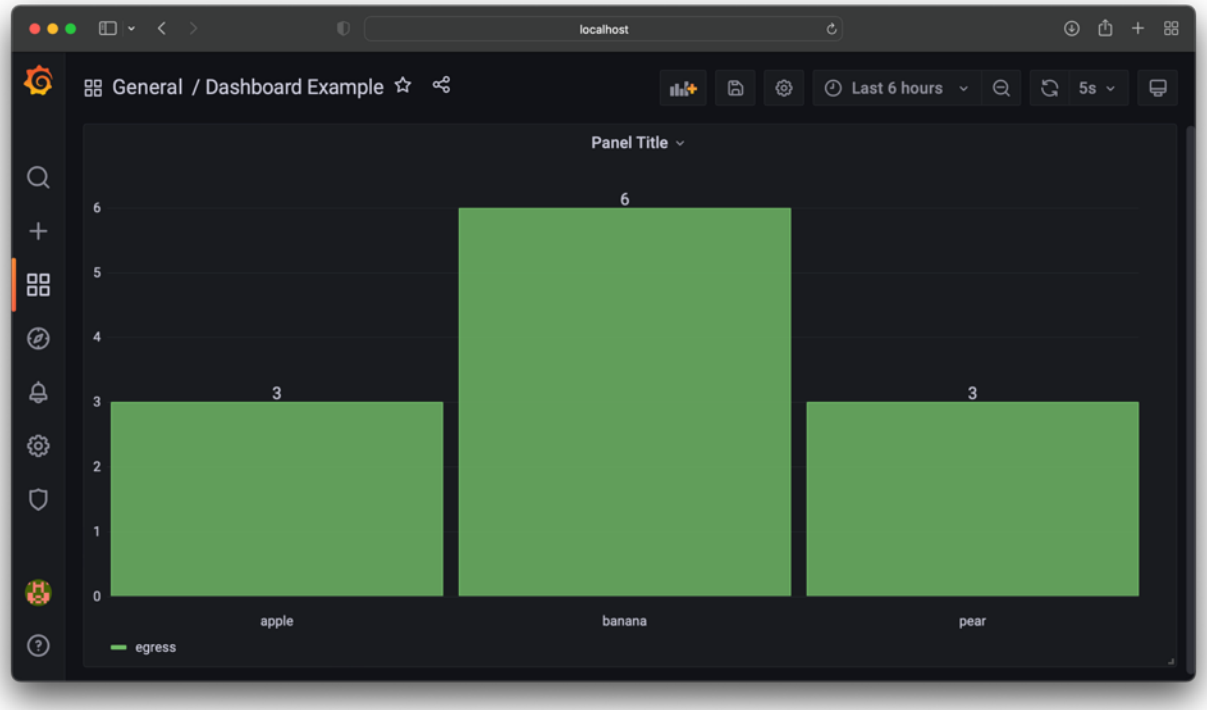


This is because the backend ETL process has not run yet. It only does it every 5 miniutes.

Oh no – error log.

Well, I did that on purpose. I made a few faulty orders, that would not process. We will not dive into that now – it was just for demonstration purposes.

Also we have the first border we made.



Let’s explore the new board.

The FastAPI Dashboard.

There are a lot of charts.

- **Total requests per minute** – How many requests are there per minute. It is currently around 60 POST calls and 4 GET calls per minute.
- **Errors per second** – There are none. It is important to notice, these are responses from the server which are not return code 200, which is a bit restrictive. But also, understand that this has nothing to do with wrongly processed items in our ETL. This is the REST API, is it responding 200 codes, or something else.
- **Requests per second** – A bit redundant metric.
- **Average response time** – How long does it take to process a request.
- **Request duration p50** – How long do the longest 50% of request take.
- **Requests under 100ms** – How many percentage requests are under 100ms. Maybe, you aim to keep all under 100ms, then this is a good metric to have.
- **Request duration p90** – How long are the slowest 10% requests.
- **Memory usage** – Great one to follow, to see if it keeps growing, or it gets close to system limit.
- **CPU usage** – Also a good metric to monitor, to see if it pushes the CPU to the limit.

Exercises

Exercise 08-00

- As we know you can setup alerts in the Dashboard.
- Setup an alert if you get more than 100 requests per minute.

Summary

This chapter has introduced you to the world of metrics.

- Metrics do not by themselves tell if a system is healthy.
- Metrics need context knowledge to set appropriately.
- Metrics are divided into leading and lagging indicators.
- Lagging indicators are alerts when the system is failing.
- Leading indicators are early warnings that the system might fail in the near future.
- Metrics are used to make you aware of problems, such that your customers or users have to contact you when the system is not working appropriately.
- Start with simple metrics and alerts.
- Add more when you understand the system and metrics better.

09 – Docker Compose

We are at a point where it is quite complex to start the entire ecosystem of services. Luckily, there is a solution for that: Docker Compose.

- In this chapter you will learn how to use Docker Compose to start a collection of Docker containers together.

What is Docker Compose

> "Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration." *(**docker.com**)*

Docker docs does a great job at explaining.

But what does it mean for us?

- Remember last chapter that we needed to start 5 docker containers?
- That was a pain if you need to debug and start them, shutting them down, starting them again.

This is what Docker Compose can do for us.

- With one command start all 5 containers.
- With one click, stop or delete all containers.

Let's try to learn how to do it.

Docker Compose file

We need to learn a bit of understanding yml-files (pronounced: Yammel files).

Let's clone a repository and get started.

- Repository: **git@github.com:LearnPythonWithRune/docker-compose-example.git**

It has the following file.

docker-compose.yml

```
services:
  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
  ingress-egress-mysql:
```

```
    image: ingress-egress-mysql
    ports:
      - "3306:3306"
    depends_on:
      - prometheus
  fruit-api-v3:
    image: fruit-api-v3
    ports:
      - "8000:8000"
    depends_on:
      - prometheus
      - ingress-egress-mysql
  etl:
    image: etl
    depends_on:
      - fruit-api-v3
  grafana:
    image: grafana/grafana
    depends_on:
      - prometheus
      - ingress-egress-mysql
    ports:
      - "3000:3000"
    volumes:
      - ./grafana-data:/var/lib/grafana
```

The structure is from indentation.

On the top-level we have the service.

The service consists of:

- promehtues
- ingress-egress-mysql
- fruit-api-v3
- etl
- grafana

These are the 5 containers we want running.

Each of the container has some or all of the following attributes.

- They all need to have an **image**. This image must either be in the local Docker registry or in a remote one it is connected to. In our case, we have the ones we built in previous chapters and some available ones (Prometheus and Grafana).
- Ports if we need to expose ports on localhost. This is needed, as we have learned, if we expose some service.
- Volumes is needed if we want specific configuration or persistence between runs.
- Depends on is to tell Docker Compose that this container should needs the following containers to be running before it is started.

This corresponds exactly to the docker run commands we have had.

What to understand before we run it

Docker Compose does not build the images, if they do not exist. We need to do that from the individual projects we have. If you did not delete any images in Docker Desktop and you have done everything in all chapters, they should be available.

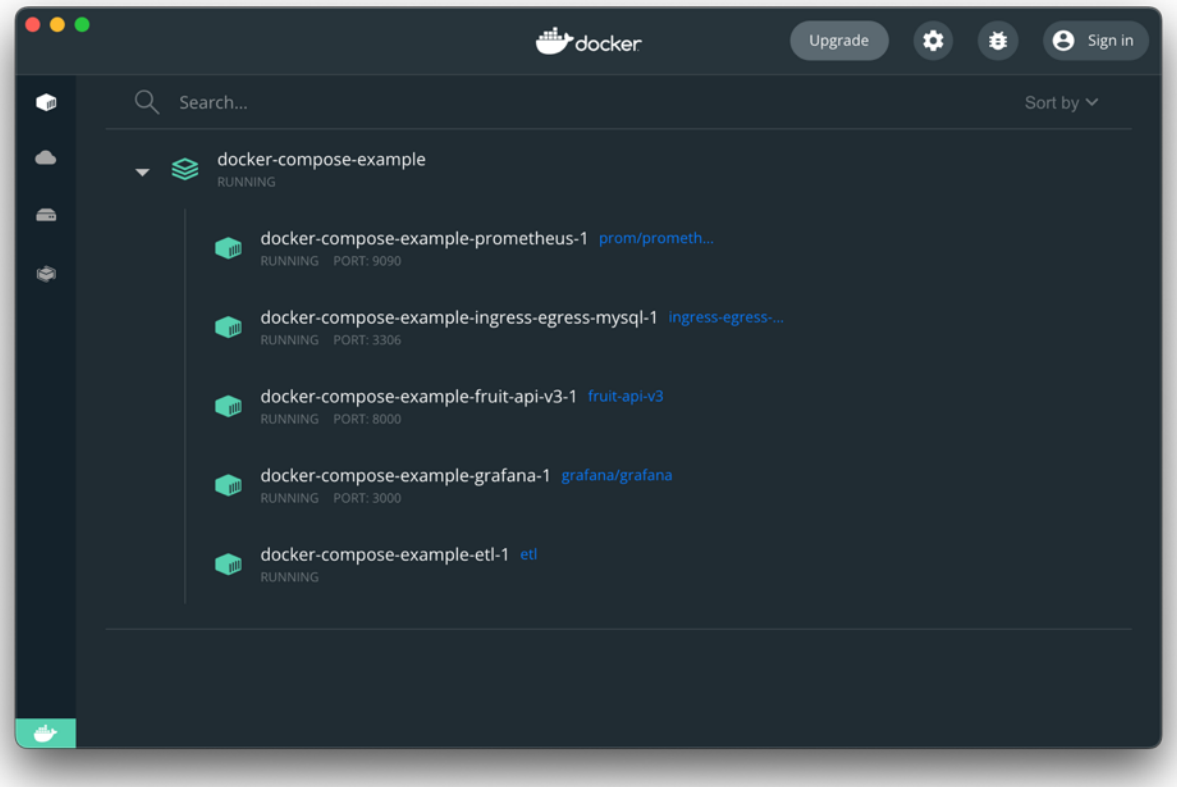
How to start it

Before you start it, you need to remember to shut down and possibly delete the containers you have running from previous chapters.

Then you can run docker with compose as follows from the terminal in PyCharm in the project we just cloned (docker-compose-example).

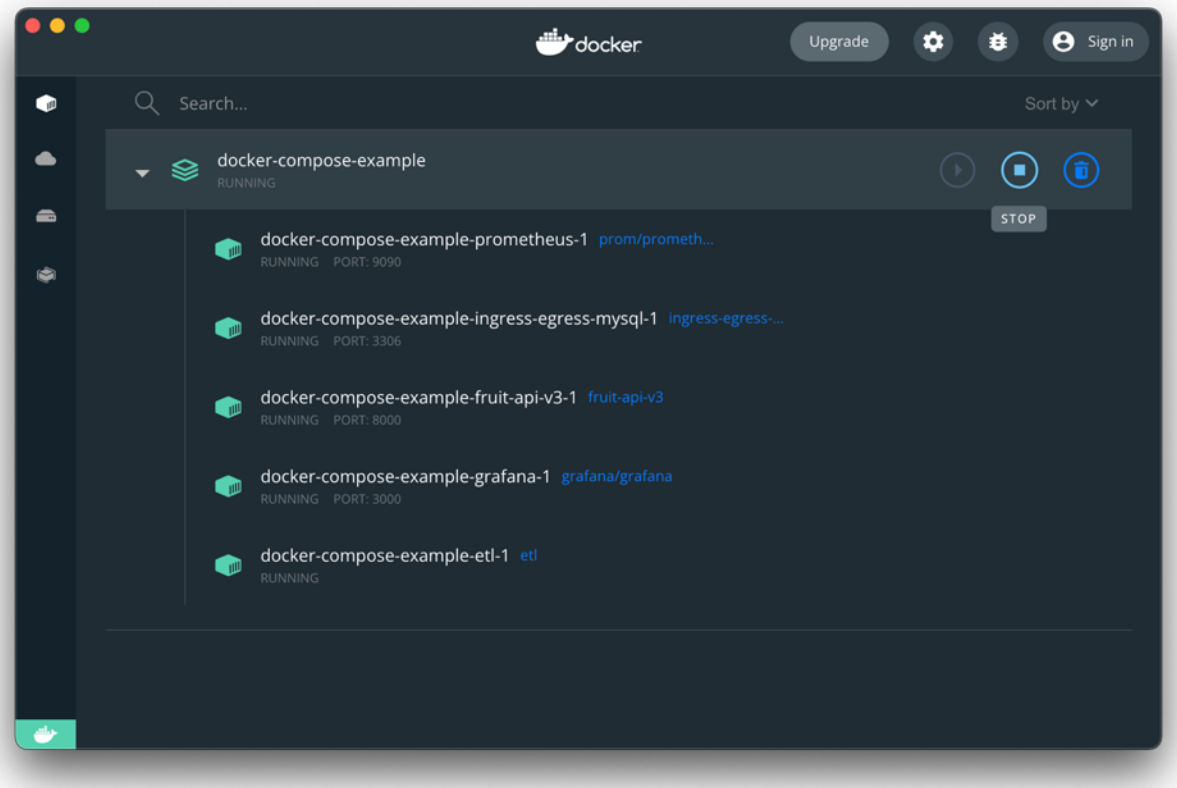
```
docker compose up
```

This will start all 5 containers in a bundle.



Now that was easy.

You can stop it or delete simply by pressing the button inside Docker Desktop, if you hover over the docker-compose-example



You might have notice that in the terminal it keeps it occupied.

This is because we started without the detach flag (-d).

Hence, if you start docker compose as follows.

```
docker compose up -d
```

Then it will detach and the terminal is freely available.

How to edit the docker-compose.yml file

A few things many find frustrating at first with yml-files, is, that it is very strict with whitespaces, tabs, empty lines with whitespace or similar.

That is, if you have an “empty” line with a single whitespace, it might not want to run.

Anyhow, just be cautions.

But the format is straight forward as mentioned already.

Summary

We learned that Docker Compose can help us use multiple containers in one bundle.

- **docker compose up** will start the composition of docker containers found in docker-compose.yml.

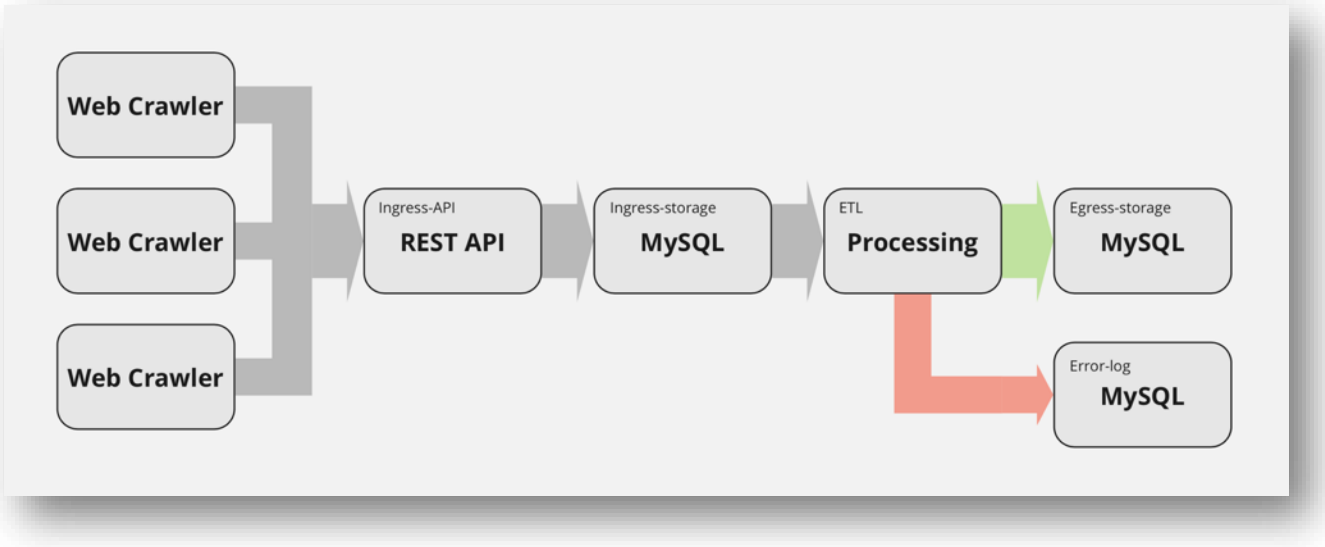
- Docker compose does not build the images. You need to do that from the projects, where the Dockerfiles are.

10 – Capstone Project

In this chapter we will use all our knowledge and setup a framework where we have multiple web-crawlers ingest data to an ingress REST API with data, an ETL background process will transform the ingested data (ingress data) to egress and expose it.

- We will see how simple it all is with all the modules with minor modification can accomplish that.
- At the end of the chapter you will know how to use it for multiple web-crawlers.
- We will discuss design of it and other alternatives.

Project Overview



If you first ignore the 3 Web-Crawlers, then we actually have the same design.

But let's just go through the modules here.

- **Web Crawler** – The idea is, that there can by many Web Crawlers (3 on the illustration), which each crawl one page on some kind of schedule. We will get back to how these crawlers can be made simple an easy to maintain.
- **REST API** – The crawlers ingest the "raw" data they extract from the pages. Often the raw data is just an integer represented as string ("23"), or an amount with a dollar sign ("$ 10").
- **Ingress-storage** – This is where the input data is store in raw format.
- **ETL** – This is a background process that will do the transformation, converting the string "23" to an integer 23, or the string "$ 10" to an integer 10.
- **Egress-stroage** – This is where we store the successfully transformed values.
- **Error-log** – This is where we store the unsuccessful transformations.

First two questions.

1. Why not let the Web Crawler write directly to the MySQL server ingress-storage?
2. Why not let the Web Crawler do the transformation?

The short answer to both is.

- **You could.**

But you might gain something with the design above.

- If you choose or need to use another storage than MySQL, say, you need to switch to PostgreSQL, because it performs better and you have too high load on your MySQL database. You have 50 Web Crawler running. Then you do not need to change anything in the Web Crawlers, you only need to change something in the REST API.
- Also, the monitoring we get in the REST API is easy to understand and monitor, independent of the storage below. We do not need to find some monitoring, which gives us the right insights with MySQL or whatever technology we use.
- Why wait with the transformation? Well, some transformations might be very simple. But in general, you want to keep your modules as simple as possible with only simple functionality in it. This makes them easy to monitor and understand what is failing. In the above design, you know whether it is the crawling or the transformation that fails from a Dashboard.
- Also, as discussed earlier, if you did the transformation wrong, you can re-do on all the data again, because you saved the raw format.

This might lead to more questions.

- **Why not keep the entire webpage every time?** Yes, you could. It might pay off. But sometimes you need advanced crawlers like Selenium, and it makes it more difficult to keep them and reprocess them. But sure, you might want to do that.

All I am saying, yes, for you it might be overkill to have all the running – in other cases it might not be enough.

Unfortunately, there is not one design, which is perfect for all settings. There are some principles, you might want to learn and understand why you make them.

What we want to achieve, is to make a setting where.

- It is easy to add more Web Crawlers.
- Make sure the data quality (the transformation) is good, and we know it before the customer complains.
- Monitor that all runs as expected – again, we want to know it before customer complains.

If this design is good for your use-case. Difficult to say.

But you will learn something about making easy simple design, which is easy to monitor and maintain.

Are you ready to get started?

What repositories will we use?

To setup our work we need the following 4 repositories.

- **capstone-ingress-api**
  - Contains the REST API to ingest data from the crawlers (or spiders)
  - Repo: **git@github.com:LearnPythonWithRune/capstone-ingress-api.git**
- **capstone-etl**
  - The ETL process
  - Repo: **git@github.com:LearnPythonWithRune/capstone-etl.git**
- **capstone-infrastructure**
  - Setup for MySQL database, Grafana, Prometheus, and docker compose
  - Repo: **git@github.com:LearnPythonWithRune/capstone-infrastructure.git**
- **capstone-spider**
  - The web crawlers to crawl the web
  - Repo: **git@github.com:LearnPythonWithRune/capstone-spider.git**

Let's have a small walkthrough of all the repositories.

Ingress API our entry point

This is where the Spiders (or web crawlers) will ingest data they have crawled. The data will in general be quite raw and we will let the transformation process in ETL make the transformation, to keep each module simple and easy to monitor.

The structure of the Ingress API is as we know it.

Some design principles we have we know already.

- No heavy functionality in the Ingest API. It only writes to the database.
- We have the Ingest API and do not let the crawlers talk directly to the databases to keep them decoupled from that. This way we can change technology and how we store the data, without changing anything in the crawlers. This can be important, if you later have 50+ different kind of crawlers.
- The Spider sends the full state – we add a ingest time, for later debugging.
- All times are stored in UTC.

A brief look at the app/routers/ingest.py file.

```
import logging
import os
from datetime import datetime
from http import HTTPStatus
from typing import Dict

from fastapi import APIRouter

from app.routers.mysql_module import Storage

logger = logging.getLogger(__file__)
router = APIRouter(tags=['income'])
```

```
storage = Storage(os.getenv('STORAGE_HOST', 'localhost'))


@router.post('/ingest', status_code=HTTPStatus.OK)
async def order_call(crawl_time: datetime, pipeline_name: str, ingest_key: str,
ingest_value: str) -> Dict[str, str]:
    logger.info(f'Incoming ingest: {crawl_time=}, {ingest_key=},
{ingest_value=}')

    ingest_time = datetime.utcnow()
    storage.ingest(crawl_time, ingest_time, pipeline_name, ingest_key,
ingest_value)

    logger.info(f'Stored ingest({crawl_time=}, {ingest_time}, {pipeline_name=},
{ingest_key=}, {ingest_value})')

    return {'ingest_key': ingest_key, 'ingest_value': ingest_value}
```

Everything is quite similar.

We have more arguments.

- **crawl_time**: The time inserted by the crawler (or Spider).
- **pipeline_name**: We want to have multiple pipelines to be able to ingest. It can be, as we will here, that multiple crawlers ingest to the same pipeline.
- **ingest_key**: To have key-value pairs of values. Might be necessary.
- **ingest_value**: To have key-value pairs of values. Might be necessary.

Remember to build an image of it to your Docker registry.

```
docker build -t capstone-ingress-api .
```

You should not run it – we will start it from our infrastructure as you will see later.

ETL module

This one is similar to the one we know, but with a more flexible design.

What we want to achieve with it.

- For each needed transformation, we want to build a class to represent it.
- Then when parsing data from a pipeline, it will have transformation configured.
- Hence, each pipeline has each their transformation.

A fast walkthrough the files.

- **crontab**: To schedule a pipeline run – nothing new in this file.
- **db_connector.py**: To connect to the database – nothing new in this file.
- **Dockerfile**: To build an image – nothing new in this file.
- **etl.py**: The processing class and subclasses. We will look at this file.
- **list_databse_content**: To see content in database – nothing new in this file.
- **main**.py: Setting up and running the pipelines from a state file. We will look at this file.
- **README.md**: The readme file.
- **requirements.txt**: What to install

- **state.json**: Used to store the state.

Let's start with the **etl.py** file.

```
import csv
import logging
from abc import abstractmethod, ABC
from datetime import datetime
from typing import List, Any, Tuple, Dict

from mysql.connector import ProgrammingError

from db_connector import MySqlConnector


logger = logging.getLogger(__file__)


class ETL(ABC):
    def __init__(self,
                 db_connect: MySqlConnector,
                 ingest_time: datetime,
                 pipeline_name: str):
        self.db_connect = db_connect
        self.ingest_time = ingest_time
        self.pipeline_name = pipeline_name

    def extract(self):
        sql_stmt = f"SELECT crawl_time, ingest_time, pipeline_name, ingest_key,
ingest_value " \
                   f"FROM ingress WHERE ingest_time>'{self.ingest_time}' AND
pipeline_name='{self.pipeline_name}'"
        new_rows = self.db_connect.fetchall_query(sql_stmt)
        return new_rows

    @abstractmethod
    def _transform(self,
                   crawl_time: datetime,
                   ingest_time: datetime,
                   pipeline_name: str,
                   ingest_key: str,
                   ingest_value: str):
        raise NotImplementedError

    def transform(self, rows_to_process: List) -> List:
        transformed_rows = []
        for crawl_time, ingest_time, pipeline_name, ingest_key, ingest_value in
rows_to_process:
            try:
                transformed_values = self._transform(crawl_time, ingest_time,
pipeline_name, ingest_key, ingest_value)
                transformed_rows.append((ingest_time,) + transformed_values)
            except Exception as e:
                error_msg = f'{type(e).__name__}: {e}'.replace('\'', '"')
                self.log_error(ingest_time, ingest_key, ingest_value, error_msg)
        return transformed_rows

    def _load_sql_statement(self, row_to_load: Tuple) -> str:
        raise NotImplementedError
```

```
    def load(self, rows_to_load: List):
        last_successful_ingest_time = self.ingest_time
        for ingest_time, *row_to_load in rows_to_load:

            sql_stmt = self._load_sql_statement(row_to_load)
            try:
                self.db_connect.commit_query(sql_stmt)
            except ProgrammingError as e:
                error_msg = f'{type(e).__name__}: {e}'.replace('\'', '"')
                self.log_error(ingest_time, 'sql_stmt', sql_stmt, error_msg)

            # We update the last_successful_ingest_time for both cases (whether
it worked or it was logged to error_log)
            if ingest_time > last_successful_ingest_time:
                last_successful_ingest_time = ingest_time

        return last_successful_ingest_time

    def log_error(self, ingest_time: datetime, ingest_key, ingest_value,
error_msg):
        sql_stmt = f"INSERT INTO " \
                   f"error_log(ingest_time, error_time, pipeline_name,
ingest_key, ingest_value, error_message) " \
                   f"VALUES('{ingest_time}', '{datetime.utcnow()}',
'{ingest_key}', '{ingest_value}', '{error_msg}')"
        self.db_connect.commit_query(sql_stmt)

    def process(self):
        logger.info(f'processing {self.pipeline_name=} {self.ingest_time=}')

        # Extract
        rows = self.extract()

        # Transform
        rows = self.transform(rows)

        # Load
        last_ingest = self.load(rows)

        logger.info(f'processed {len(rows)} rows {last_ingest=}')

        return last_ingest


class EtlWeather(ETL):
    def _transform(self,
                   crawl_time: datetime,
                   ingest_time: datetime,
                   pipeline_name: str,
                   ingest_key: str,
                   ingest_value: str):
        csv_reader = csv.reader([ingest_value], quotechar='"', delimiter=',',
quoting=csv.QUOTE_ALL)
        values = list(csv_reader)

        city, report_time_str, weather, temperature_str = values[0]

        city = city.split(',')[0]
```

```
        report_hour_minute = report_time_str.split(' ')[-1]
        report_hour = int(report_hour_minute.split('.')[0])
        report_minute = int(report_hour_minute.split('.')[-1])

        report_time = crawl_time.replace(hour=report_hour)
        temperature = int(temperature_str)

        return report_time, city, weather, temperature

    def _load_sql_statement(self, row_to_load: Tuple) -> str:
        report_time, city, weather, temperature = row_to_load

        sql_stmt = f"INSERT INTO egress(report_time, city, weather, temperature)
" \
                   f"VALUES('{report_time}', '{city}', '{weather}',
{temperature})"

        return sql_stmt
```

It contains two classes.

- **ETL(ABS)**
  - An abstract class that contains the machinery of the ETL pipeline run.
- **EtlWeather(ETL)**
  - An actual ETL pipeline with implementation of the abstract methods.

The ETL(ABS) class has the following methods.

- **__init__**
- **extract**
- **_transform**
- **transform**
- **_load_sql_statement**
- **load**
- **log_error**
- **process**

To understand it, you should start in the process() method, which will process a pipeline run.

```
def process(self):
    logger.info(f'processing {self.pipeline_name=} {self.ingest_time=}')

    # Extract
    rows = self.extract()

    # Transform
    rows = self.transform(rows)

    # Load
    last_ingest = self.load(rows)

    logger.info(f'processed {len(rows)} rows {last_ingest=}')

    return last_ingest
```

It makes the ETL process: Extract, Transform, and Load.

The extract only needs to know what the **pipeline_name** is and from what ingest time to extract data. It knows this from the class parameters.

```
def extract(self):
    sql_stmt = f"SELECT crawl_time, ingest_time, pipeline_name, ingest_key,
ingest_value " \
               f"FROM ingress WHERE ingest_time>'{self.ingest_time}' AND
pipeline_name='{self.pipeline_name}'"
    new_rows = self.db_connect.fetchall_query(sql_stmt)
    return new_rows
```

The transform will iterate though each line and call abstract method _transform for each row.

```
def transform(self, rows_to_process: List) -> List:
    transformed_rows = []
    for crawl_time, ingest_time, pipeline_name, ingest_key, ingest_value in
rows_to_process:
        try:
            transformed_values = self._transform(crawl_time, ingest_time,
pipeline_name, ingest_key, ingest_value)
            transformed_rows.append((ingest_time,) + transformed_values)
        except Exception as e:
            error_msg = f'{type(e).__name__}: {e}'.replace('\'', '"')
            self.log_error(ingest_time, ingest_key, ingest_value, error_msg)
    return transformed_rows
```

The load method will need to know how to store the values – hence, it uses the abstract method **_load_sql_statement**.

```
def load(self, rows_to_load: List):
    last_successful_ingest_time = self.ingest_time
    for ingest_time, *row_to_load in rows_to_load:

        sql_stmt = self._load_sql_statement(row_to_load)
        try:
            self.db_connect.commit_query(sql_stmt)
        except ProgrammingError as e:
            error_msg = f'{type(e).__name__}: {e}'.replace('\'', '"')
            self.log_error(ingest_time, 'sql_stmt', sql_stmt, error_msg)

        # We update the last_successful_ingest_time for both cases (whether it
worked or it was logged to error_log)
        if ingest_time > last_successful_ingest_time:
            last_successful_ingest_time = ingest_time

    return last_successful_ingest_time
```

To use this, you we need a class which implements the abstract methods.

This is what we have in **EtlWeather(ETL)**.

```
class EtlWeather(ETL):
    def _transform(self,
                   crawl_time: datetime,
                   ingest_time: datetime,
                   pipeline_name: str,
                   ingest_key: str,
                   ingest_value: str):
        csv_reader = csv.reader([ingest_value], quotechar='"', delimiter=',',
quoting=csv.QUOTE_ALL)
        values = list(csv_reader)
```

```
        city, report_time_str, weather, temperature_str = values[0]

        city = city.split(',')[0]
        report_hour_minute = report_time_str.split(' ')[-1]
        report_hour = int(report_hour_minute.split('.')[0])
        report_minute = int(report_hour_minute.split('.')[-1])

        report_time = crawl_time.replace(hour=report_hour)
        temperature = int(temperature_str)

        return report_time, city, weather, temperature

    def _load_sql_statement(self, row_to_load: Tuple) -> str:
        report_time, city, weather, temperature = row_to_load

        sql_stmt = f"INSERT INTO egress(report_time, city, weather, temperature)
" \
                   f"VALUES('{report_time}', '{city}', '{weather}',
{temperature})"

        return sql_stmt
```

This creates a working ETL pipeline.

It should be clear that.

- You can add any number of ETL subclasses (like EtlWeather) for each pipeline you need.

The main.py file is using the state file to run what is know.

```
import argparse
import json
import logging
from typing import List, Dict, Any
from datetime import datetime

from db_connector import MySqlConnector
from etl import EtlWeather

logging.basicConfig(encoding='utf-8', level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__file__)


def __init_argparse() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser()

    parser.add_argument('--host',
                        help='The MySQL host',
                        default='localhost')

    parser.add_argument('--state_file',
                        help='Json state file',
                        default='state.json')

    return parser
```

```
def get_state(state_filename: str):
    with open(state_filename) as f:
        content = f.read()
        state = json.loads(content)
    return state


def save_state(state_filename: str, state: List[Dict[str, Any]]):
    with open(state_filename, 'w') as f:
        f.write(json.dumps(state, indent=2))


def main():
    arg_parser = __init_argparse()
    args, _ = arg_parser.parse_known_args()
    host = args.host
    state_file = args.state_file
    logger.info(f'Config ({host=}, {state_file=})')

    state = get_state(state_file)

    connector = MySqlConnector(host)

    for pipeline in state:
        pipeline_name = pipeline['pipeline_name']
        last_ingest = datetime.strptime(pipeline['last_ingest'], '%Y-%m-
%dT%H:%M:%S')

        if pipeline_name == 'weather_spider':
            etl = EtlWeather(connector, last_ingest, pipeline_name)
        else:
            raise ValueError(f'Pipeline name {pipeline_name=} not supported')

        last_ingest = etl.process()

        pipeline['last_ingest'] = last_ingest.strftime('%Y-%m-%dT%H:%M:%S')

    save_state(state_file, state)


if __name__ == '__main__':
    main()
```

Here we have the call to EtlWeather and see how it works.

Finally, you need to build an image of it.

```
docker build -t capstone-etl .
```

Then you are ready for the infrastructure.

Infrastructure

This module is setting up your Prometheus, MySQL server, Ingress API, ETL process, and Grafana. It contains the following files.

- **docker-compose.yml**: This sets it all up. Similar to what we know.

- **dockerfile.mysql**: Building our MySQL database server image.
- **prometheus.yml**: Configuration for our Prometheus image.
- **schema.sql**: Used to setup our MySQL database.

You need to build the following Docker image.

```
docker build -t capstone-mysql -f Dockerfile.mysql .
```

Then if you have build the images **capstone-ingress-api** and **capstone-etl** (done above) then you are ready to start it all.

Ready?

Run this.

```
docker compose up
```

If something fails it is probably because of one of the following:

- Docker Desktop is not running.
- You lack a Docker image in your local registry.

The Spiders

The spider is a simple example of a web crawler, which will crawl the weather in cities around the world.

A few design pointers of it.

- It will only crawl one city – you should run it for each city you want. This keeps the code simple and easy to maintain. If something fails, then you can see if all fail or just for specific city. This gives you fast insights into what happens.
- Web pages change all the time; therefore, it is also good to have them all separated and configured for each query you have against a page, and not try to combine more in one. Again, this keeps it simple and easy to maintain.

The **spider.py** file has the following content.

```
import argparse
import logging
from datetime import datetime
from bs4 import BeautifulSoup
import requests


logging.basicConfig(encoding='utf-8', level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__file__)


def __init_argparse() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser()

    parser.add_argument('--host',
                        help='The Ingress REST API',
                        default='localhost')
```

```
    parser.add_argument('--city',
                        help='City to crawl weather',
                        default='Copenhagen')

    return parser


arg_parser = __init_argparse()
args, _ = arg_parser.parse_known_args()
HOST = args.host
CITY = args.city
logger.info(f"Configuration: {HOST=} {CITY=}")


# Google: 'what is my user agent' and paste into here
headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)'
                         ' AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.3
Safari/605.1.15'}

logger.info(f"User-agent' {headers['User-Agent']}")


def ingest(crawl_time: datetime, name: str, ingest_key: str, ingest_value: str):
    response = requests.post(
        url=f'http://{HOST}:8000/ingest',
        params={
            'crawl_time': crawl_time,
            'pipeline_name': name,
            'ingest_key': ingest_key,
            'ingest_value': ingest_value,
        }
    )
    return response.status_code


def weather_info(city):
    city = city.replace(" ", "+")
    res = requests.get(
        f'https://www.google.com/search?q={city}&hl=en',
        headers=headers)

    soup = BeautifulSoup(res.text, 'html.parser')

    # To find these - use Developer view and check Elements
    location = soup.select('#wob_loc')[0].getText().strip()
    time = soup.select('#wob_dts')[0].getText().strip()
    info = soup.select('#wob_dc')[0].getText().strip()
    weather = soup.select('#wob_tm')[0].getText().strip()

    crawl_time = datetime.utcnow()
    name = 'weather_spider'
    ingest_value = f'"{location}","{time}","{info}","{weather}"'

    response = ingest(crawl_time, name, 'weather-info', ingest_value)
    logger.info(f"Ingested {ingest_value=} @ {crawl_time} with response
{response=}")
```

```
if __name__ == '__main__':
    weather_info(f"{CITY} Weather")
```

As you see – you have the following options.

- You can set the **city** and **host**.
- The **host** is needed to run in a Docker container.
- The **city** is to crawl the weather for different cities.

You can build an image as follows.

```
docker build -t capstone-spider .
```

As default it will run for Copenhagen.

To run it for specific city you can run as follows.

```
docker run --rm capstone-spider python spider.py --host host.docker.internal --
city Stockholm
```

I use the –rm to remove the image after it has run.

We just want data to play around with. Hence you can setup a script to run it every minute.

See **script.sh** for inspiration.

This will ingest data every minute.

Then you can log into Grafana and see if it is working.

http://localhost:3000/

Considerations

A few things to consider.

- This setup is not fully production ready. It can be used as such, but the purpose was not to become DevOps. It was to teach you how to be a Python developer and not a DevOps.
- A better setup would use Kubernetes (K8s) to schedule cronjobs, like the crawlers (Spiders) and the ETL pipeline. This gives better flexibility.
- The design might not be the right one for all web scraping tasks. It is not always possible to divide the flow as easy as this project. The idea of keeping the history of scraped data might be good, but unfortunately not always possible due to the complexity of web scraping.

Possible options

- Some love Kafka queues to process data. This could be an option to use that. Here we have focused on keeping the tech stack and design simple.

Summary

In this chapter we have covered the following.

- How to make a setup running multiple Spiders that crawl the web.
- Using all the knowledge we have learned to make it easy.
- We have learned to use Docker, Dockerfiles, Docker Compose, Docker Desktop in our local setup.
- How we use containers to easily setup MySQL servers, Prometheus metrics, Grafana. This helps us understand what we need to use in the production setup.

What we are not.

- Not DevOps that can setup a great production setup.

What next?

Learning all this takes time and it can be a bit overwhelming at first.

You should not get frustrated if you do not remember it all now. This book is here to help you recall how and why we do all of it.

This book taught you what you need to know, not to be an expert in it all. Most senior developers are not good at this, but they know their way around.

Don't worry too much about not knowing the perfect answer in a job interview. Most google to recall how to do something. The key is, now you know it can be done and how it should be done.

I have participated in more than 100 job interviews as part of my career to hire new talent. Here are some differences between Junior and Senior developers I have noticed.

Juniors think they need to know the perfect answer to all.

Seniors know that you do not need to remember anything by heart – we have Stackoverflow to help us.

When given a programming problem to solve, Juniors often say nothing and dare not to say what is on their mind, as they are afraid it might be wrong.

Seniors just say what is on their mind and feel comfortable about being wrong at first.

As this book also has revealed to you – the world of programming is far from perfect. You are a developer and you will be one of the most expensive resources in your organization. They pay you to create code that is good enough – not perfect.

Perfect code costs too much – perfect takes experience with what you do, not only senior experience, but experience with the thing you build for the first time.

Try to make simple designs and decouple the code as much as possible.

Always think about how this code will be for someone else to debug or extend – this will keep your focus on writing good code.

Stay connected

Programmers love others with the same passion.

One funny thing – most think programmers are a bunch of introverts. Okay, we are. But something surprises them, when they get to know the programming community better.

Programmers might sit at a dinner party and say nothing. But when they are surrounded by other like-minded, they are interacting and talking like crazy.

They look just like any group of extroverts having fun.

Yes, we talk about other stuff – tech stuff, programming, weird jokes most don't get.

But that is some of the fun of being in this community.

I enjoy every minute with all of them and I have been part of the community all my life.

I love teaching and connect with you all.

If you want to be part of that, I would suggest you join our community on your favorite social media.

YouTube: **https://www.youtube.com/c/LearnPythonwithRune**

Facebook: **https://www.facebook.com/learnpythonwithrune**

Twitter: **https://twitter.com/PythonWithRune**

Tutorials and free courses: **https://www.learnpythonwithrune.org**

You can always read me on email: **rune@lpwr.org**

Thank you

You are awesome.

Thank you for taking the time to learn some great aspects of programming.

Remember, we all started like rookies making trivial mistakes. I have been there a million times, my low confidence helped me feel horrible about my lack of skills.

Today I know I am not the best, but it is okay. There will always be someone better at what you do. I have learned to ask for help when I see someone great at something, and it has helped me grow as a developer.

Remember the start of the book:

"You don't have to be great to start, but you need to start to become great." – Zig Ziglar

My way to keep motivated and learn is as follows.

- Stay positive and try to learn from others.
- Always think long-term in your learning. Nobody just mastered everything from day one.
- Have fun while doing it. Life is your biggest gift – make your time here awesome!

Rune.