Learning FastApi with Docker

Agenda

1) About this Course
2) Software Requirements
3) Why FastAPI & Docker

About this Course

What this course aims to achieve

What the course aims to achieve:

> **On completion of the workshop, participants should be able to (1) understand basic concepts of dockerization, (2) implement template example of FastApi and docker.**

This workshop focuses on DevOps (or Development operations). The main focus is on the introduction of moving from sandbox to production.

> *We did this not because it is easy, but because we thought it would be easy!*

This is a high level course and what the course does will NOT achieve is:

> **It will NOT turn individuals into DevOps experts. The aim is to provide participants with practical examples in order to recognize life implementations implications.**

- We wish to elevate people's knowledge and exposure to basic development operations principles to help guide them on their data journey.

Key outcomes

You should:

- Talk about docker as a management tool
- Understand API architecture

We also encourage the following behaviour throughout the course:

- Learn from each other and share knowledge in groups.
- Ask questions during the course - the instructor has a lot of knowledge that you should tap.

Session Breakdown: Setup

Monday 11th December (14:00 - 16:30):

- Course introduction.
- Install software for course.
- Linux basics review
- Learning about virtual environments
- Quarto and VSCode

Tuesday 12th December (10:00 - 12:30):

- Introduction to docker

Session Breakdown: API

Tuesday 12th December (14:00 - 16:30):

- Introduction to APIs (theory)

Wednesday 13th December (14:00 - 16:30):

- Dockerised API

Session Breakdown: API

Thursday 14th December (14:00 - 16:30):

- Dockerised API & dbutils

Friday 15th December (14:00 - 16:30):

- Dockerised API & dbutils

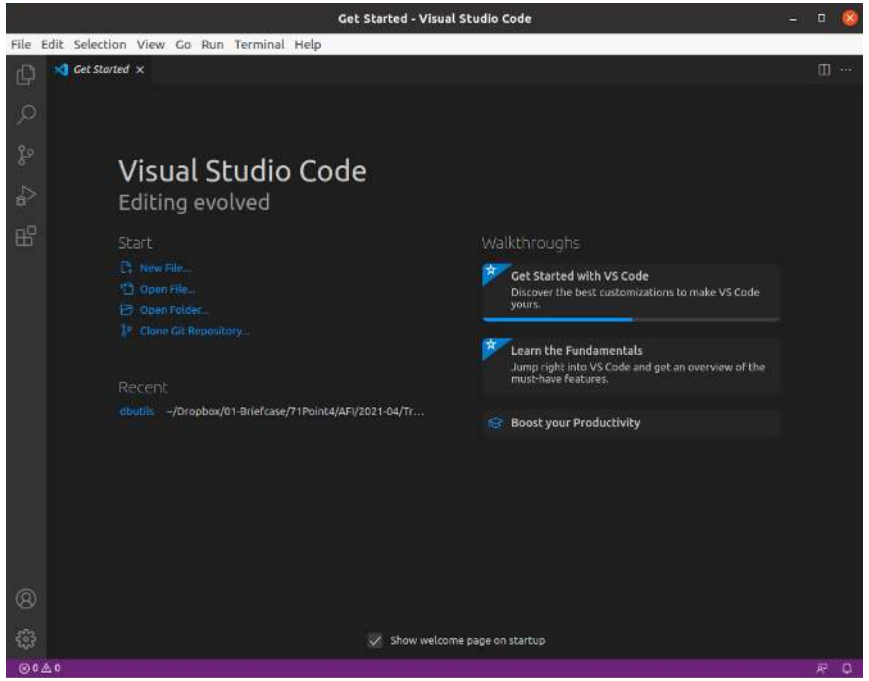
]

Learning to code in VSCode

> Why switch from RStudio to VSCode for SQL development?

The first few things we are gonna do in VSCode is:

- Interact with a remote server
- Connect to database on remote server
- Execute code and download results


Get Started - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Get Started
Visual Studio Code
Editing evolved
Start
New File...
Open File...
Open Folder...
Clone Git Repository...
Recent
dbutils ~/Dropbox/01-Briefcase/71Point4/AFI/2021-04/Tr...
Walkthroughs
Get Started with VS Code
Discover the best customizations to make VS Code yours.
Learn the Fundamentals
Jump right into VS Code and get an overview of the must-have features.
Boost your Productivity
Show welcome page on startup

Connecting to remote development environment

As in most instances, you will likely be developing code on a remote machine, but would like to use VSCode as your development environment.

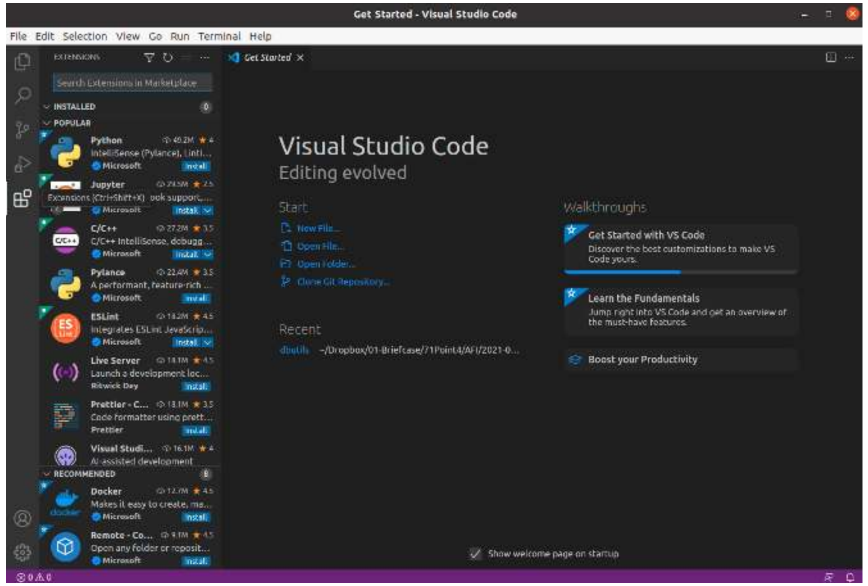
This can easily be achieved using the `Remote-SSH` feature in the IDE. This allows for:

- Develop on the same operating system you deploy to or use larger, faster, or more specialized hardware than your local machine.
- Quickly swap between different, remote development environments and safely make updates without worrying about impacting your local machine.
- Access an existing development environment from multiple machines or locations.
- Debug an application running somewhere else such as a customer site or in the cloud.

Connecting to remote development environment

> One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

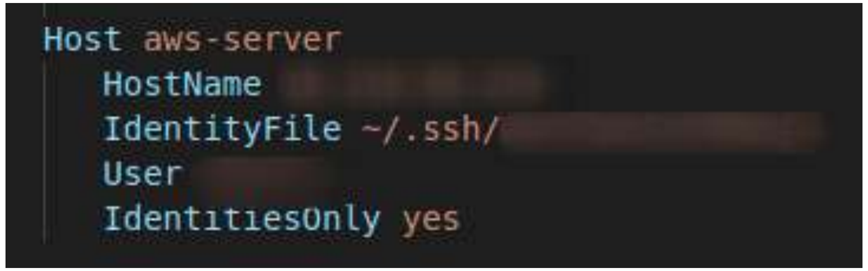
- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Connecting to remote development environment

> One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file

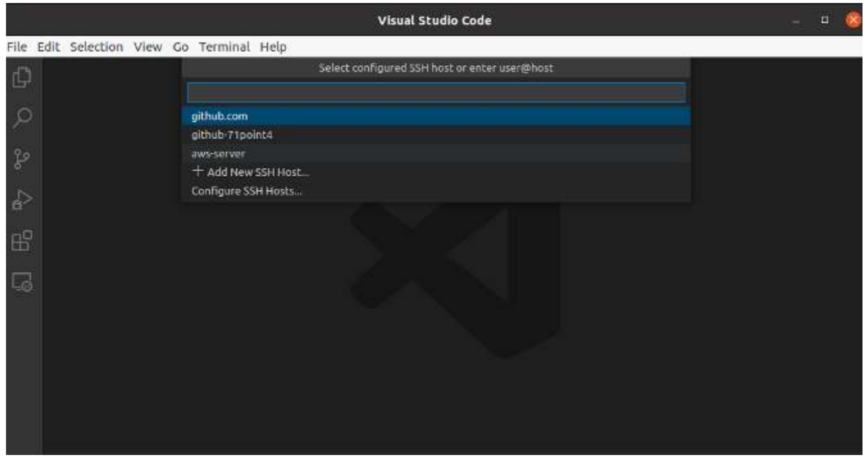


```
Host aws-server
    HostName [redacted]
    IdentityFile ~/.ssh/[redacted]
    User [redacted]
    IdentitiesOnly yes
```

Connecting to remote development environment

> One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

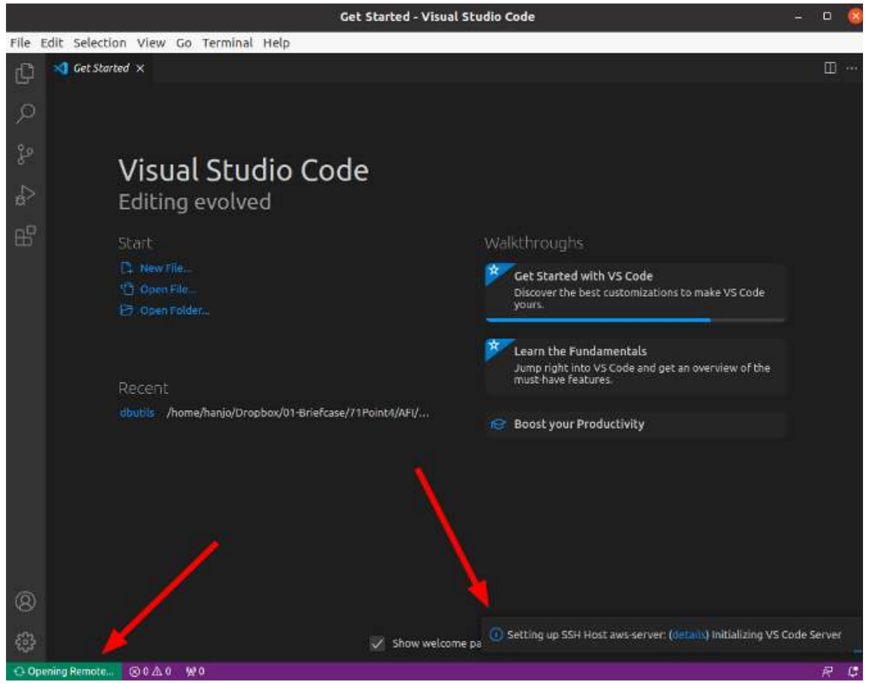
- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Connecting to remote development environment

> One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



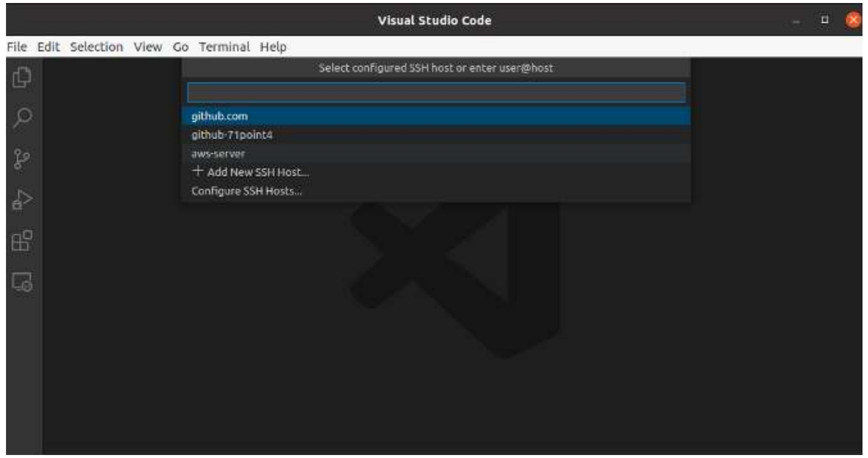
dbutils /home/hanjo/Dropbox/01-Briefcase/71Point4/AFI/...

Setting up SSH Host aws-server: (details) Initializing VS Code Server

Connecting to remote development environment

> One of the most used shortcuts in `VSCode` you will use is `Ctrl + Shift + P`. This takes you to the IDE's command console.

- Once in the command console, type `Remote SSH` and the search bar should come up with a couple of options.
- Select `Remote-SSH: Connect to Host`.
- In both Linux and Windows the easiest is to create a `.ssh/vscode-config` file



Install OpenSSH for Windows

- To install OpenSSH using PowerShell, run PowerShell as an `Administrator`. To make sure that `OpenSSH` is available, run the following `cmdlet`:

```
Get-WindowsCapability -Online | Where-Object Name -like 'OpenSSH*'
```

- Install the OpenSSH Client

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

- Test the service

```
ssh 183.204.102.12\ubunto@servername
```

Logging into Server

What is shell?

Whenever we talk about *black screen*, *command line* or *shell* we are essentially talking about the interface that takes input from the keyboard and sends it to the operating system (OS).

Almost all Linux distributions supply a shell program from the GNU Project called `bash` that looks like this:

```
hanjo@optimus:~$ penguin
```

This interface is called *shell prompt* and usually contains `username@machinename:directory`. If the last character of the prompt is a hash mark (`#`) rather than a dollar sign (`$`), the terminal session has superuser privileges (a little bit more on this later).

- Pressing the up ☝ arrow on your keyboard goes into your command history.
  - Be aware that history stores about 1,000 commands.

Different type of users

Superuser (root)

> With great power comes great responsibility!

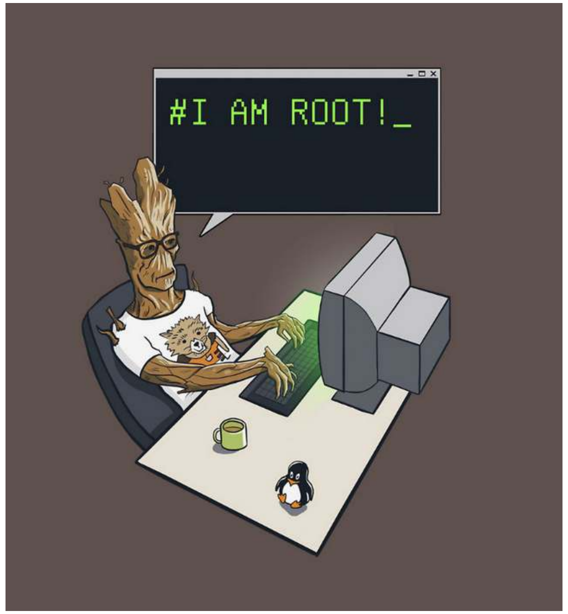
Different type of users

Superuser (root)

> With great power comes great responsibility!

On a Linux system Superuser refers to the root user, who has unlimited access to the file system with privileges to run all Linux commands.

- This responsibility is mostly given to experienced SysAdmins. The reason being there is no "take-backsies" in linux. Once a command has been executed under `sudo` (superuser do) , there is almost never a way to reverse the execution (ex. deleting a file).
- The Superuser/Root is also responsible for setting up security and thus, limiting the power to a single (or very few individuals is preferred).



Basic shell commands

Try these basic commands:

```
date
```

```
## Thu 30 May 2024 16:07:46 SAST
```

```
free -h
```

```
##                total        used        free      shared  buff/cache   available
## Mem:           62Gi       7.1Gi        41Gi       2.4Gi        13Gi        52Gi
## Swap:          19Gi          0B        19Gi
```

```
cal
```

```
##       May 2024
## Su Mo Tu We Th Fr Sa
##           1  2  3  4
##  5  6  7  8  9 10 11
## 12 13 14 15 16 17 18
## 19 20 21 22 23 24 25
## 26 27 28 29 30 31
##
```

Welcome to your new home

> Welcome to your new home, or `127.0.0.1` as I would like to call it.

```
hanjo@optimus:~$ ls -lart

## total 20
## drwxr-xr-x 6 root  root  4096 Dec 19 13:05 ..
## -rw-r--r-- 1 hanjo hanjo  807 Dec 19 13:05 .profile
## -rw-r--r-- 1 hanjo hanjo 3771 Dec 19 13:05 .bashrc
## -rw-r--r-- 1 hanjo hanjo  220 Dec 19 13:05 .bash_logout
## drwxr-xr-x 2 hanjo hanjo 4096 Dec 19 13:05 .
```

- Can anyone tell me what they think the `-rw-r--r--` stands for?

Although we will not go deep into security in this course, it is good to understand some basics.

Permissions

Owners, Group Members, and Everybody Else

One of the fundamentals that were built into Linux systems from the start is the concept of it being a *multiuser* system. This means that multiple users can log into the system at the same time without interfering (mostly) with each others processes and files.

In the Linux security model, a user may *own* files and directories.

- When a user owns a file or directory, the user has control over its access.
- Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners.
- An owner may also grant some set of access rights to everybody, which in Linux terms is referred to as the world.

Permissions

> Owners, Group Members, and Everybody Else

How does this look for the user I just created?

And for Superuser `ubuntu`?

```
ubuntu@optimus:~$ id ubuntu
## uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(au
```

Basic Commands 🤓

Listing directories

To find out where in the `tree` you are, we can use a simple command called: `pwd`

```
hanjo@optimus:~$ pwd
## /home/hanjo
```

Upon logging into a system, the terminal will always set your working directory to `home` also known as `~`.

- If you log in as a regular user, your `home` directory is the only place where you will be able to write and create files.

So, now that we are in the system, what directories are in my `home` folder [*]?

```
hanjo@optimus:~$ ls
## Data   Desktop   Documents   Pictures
```

To list the files and directories in the current working directory, we use the `ls` command. This command is very versatile as you will see in a minute.

[*] *Yours might look a bit different depending whether you are running Linux on a server or a desktop.*

Changing the current working directory

Obviously looking at files in your `home` directory doesn't take you very far. We need to be able to navigate the file system in a quick and efficient manner.

The `cd` command in Linux is a powerful way to navigate the tree folder structure that is the file system.

```
hanjo@optimus:~$ cd Data
hanjo@optimus:~/Data$
```

The two main methods for traversing the tree is: (1) Absolute Paths and (2) Relative Paths:

- **Absolute Paths** begins with the root redirectory `/` and expands to the folder you are interested in: `/home/hanjo/Data`
- **Relative Paths** starts at the working directory and starts navigation from there. These paths have a special notation, a single dot (`.`) and a dot dot (`..`). The `.` notation refers to the working directory, and the `..` notation refers to the working directory’s parent directory.

Changing the current working directory

Lets see an example of the **absolute** and **relative** path in action. Start by navigating the `/usr/bin` directory and listing all the files.

```
hanjo@optimus:~$ cd /usr/bin
hanjo@optimus:/usr/bin$ pwd
#/usr/bin
hanjo@optimus:/usr/bin$ ls
## 2to3-2.7 funzip mpiCC splitfon ...
```

Now lets move to the `/usr` directory from our working directory `/usr/bin`. There are two ways to do this, either **absolute** (`cd /usr`) or **relative**. Let us practice using the **relative** method.

```
hanjo@optimus:~$ cd /usr/bin
hanjo@optimus:/usr/bin$ cd ..
hanjo@optimus:/usr$ pwd
# /usr
hanjo@optimus:/usr$ ls
# bin/  games/  include/  lib/  lib32/  local/  sbin/  share/  src/
```

Changing the current working directory

There are also some nice shortcuts to be aware of:

- Change the working directory to your home directory: `cd ~`
- Change the working directory to the previous working dir: `cd -`
- Change the working directory to a specific user: `cd ~ubuntu`

Notes about filenames in Linux

Filenames in Linux are quite special and if you have worked closely with someone who works in Linux, you would have noticed some things. First and foremost:

- NEVER use a space in filenames use an underscore (`_`) instead - thank me later ;-)
    - ex. `this file Name SUCKS 1/30/23.txt` where `this_is_much_better.txt`
- Filenames that start with a `.` are hidden files. The `ls` command will not list these unless you use a *parameter* `ls -a`. These files usually relate to configuration settings.
    - ex. `.bashrc`.
- CASE MATTERS, so dont ever use Capitals for folders or filenames - it gets confusing.
    - ex. `This/path/IS/different/`. from `/this/path/is/different/`
- Linux does not have any concept of "file extensions". So remember to name your files in an appropriate manner if you would like them to be readable by the correct application.
    - ex. `mypdffile` and `mypdffile.pdf` is the same

See this presentation by Dr. Anna Krystalli for further tips on file naming.

Creating files and folders

Apart from knowing how to navigate folders, we must also know how to create files and folders.

The basic commands for this is:

- Create folder

```
mkdir scripts
mkdir scripts data analysis
```

- Create file

```
touch analysis.R
```

90% of the time you will be using the basic versions of these commands. But they can also do some pretty interesting things.

Tricks and Tips for mkdir

- Create folders within folders that do not already exist (recursively create).

```
mkdir project/analysis/scripts
# mkdir: cannot create directory ‘project/analysis/scripts’: No such file or directory

# Correct usage
mkdir -p project/analysis/scripts
```

- What if I wanted to create a `data`, `scripts` and `output` folder in a single line?

  ```
  # Note, there is NO spaces in the array
  mkdir -p project/analysis/scripts/{data,scripts,output}
  cd project/analysis/scripts/ && ll
  ```

- Current date in directory name

```
mkdir `date '+%Y%m%d'`
```

Viewing contents of files

To view the contents of a file, we use a program called `less`.

> The `less` program was designed as an improved replacement of an earlier Unix program called `more`. The name `less` is a play on the phrase "*less is more*" — a motto of modernist architects and designers.

`more` (developed in 1978) was replaced by `less` in 1983, first and foremost because `more` could only scroll forwards through a text file. `less` was written by Mark Nudelman and is currently being maintained by him to this day!

- Backwards movement
- Searching and highlighting
- Multiple files
  - Less allows you to switch between any number of different files, remembering your position in each file. You can also do a single search which spans all the files you are working with.
- Advanced features
  - You can change key bindings, set different tab stops, set up filters to view compressed data or other file types, customize the prompt, display line numbers, use "tag" files, and more.

http://www.greenwoodsoftware.com/less/faq.html#mail

Redirection in Linux 🤖

Redirection & Piping

This is maybe one of the coolest features of command line that you will learn: *Redirecting* or *piping* your results into another command. The *Input/Output* allows us to chain together commands and build pipelines of instructions.

- I/O redirection (`>`) allows us to change where output goes and where input comes from.
    - A good example of this would be the `ls` command we learned earlier.

```
hanjo@optimus0:~$ ls -l
hanjo@optimus0:~$ ls -l > all_files.txt
hanjo@optimus0:~$ less all_files.txt
```

We can also append a file using (`>>`):

```
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ ls -l >> all_files.txt
hanjo@optimus0:~$ less all_files.txt
```

Redirection & Piping

In the previous examples we redirected only the `stdout` of the command. But, we sometimes also want to redirect the errors or Standard Error (`stderr`).

To do this we add an additional command to the end of the line (`2>&1`):

```
hanjo@optimus0:~$ ls -l > all_files.txt 2>&1
hanjo@optimus0:~$ less all_files.txt
```

We redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation `2>&1`.

Once we know the concept of standard output and input, we can start stringing commands together. These are called *pipelines* and it looks this, `command1` *pipes into* `command2`:

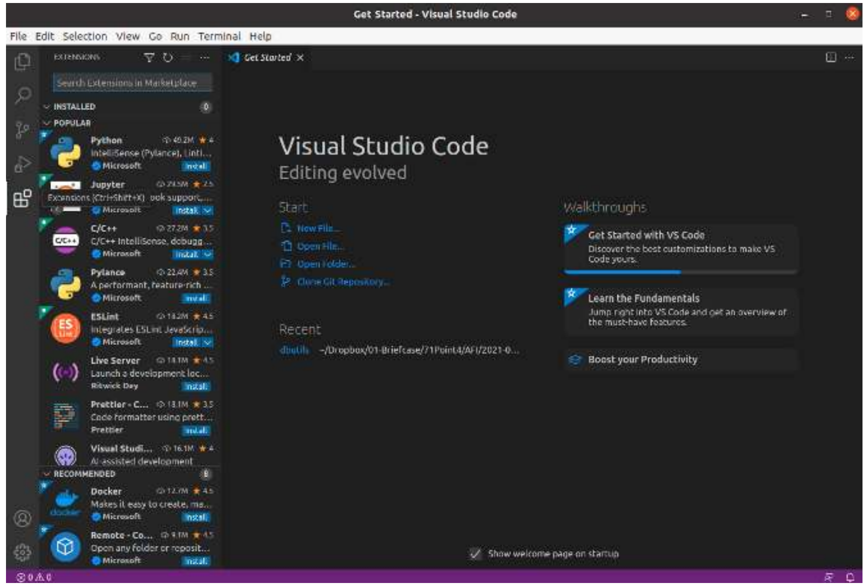
```
command1 | command2
```

Here we can see that `command2` takes `command1`'s output as its input. As you get more comfortable with the terminal, these become core concepts you will use every day.

Installing the recommended Extension

Installing *Extensions* in VSCode is pretty straight forward. Just navigation to the search tab using GUI. Then search and install the following:

- Remote -SSH
- Rainbow CSV
- autopep8
- R Extension for Visual Studio Code
- Spelling Checker for Visual Studio Code
- SQLTools



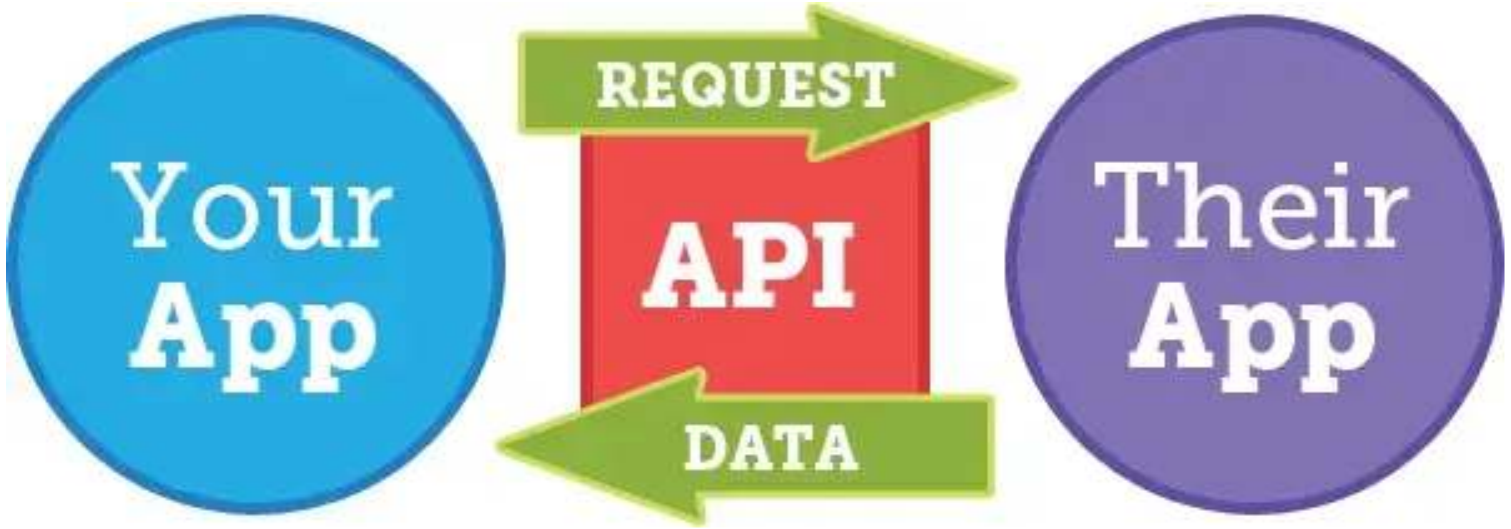
- Linux shortcut

```
wget -O extentions.sh https://bit.ly/3GrF5kn
bash extentions.sh
```

APIs

Definition

API is the acronym for *Application Programming Interface*. An API is made up of a set of defined rules that enables the communication between the different applications

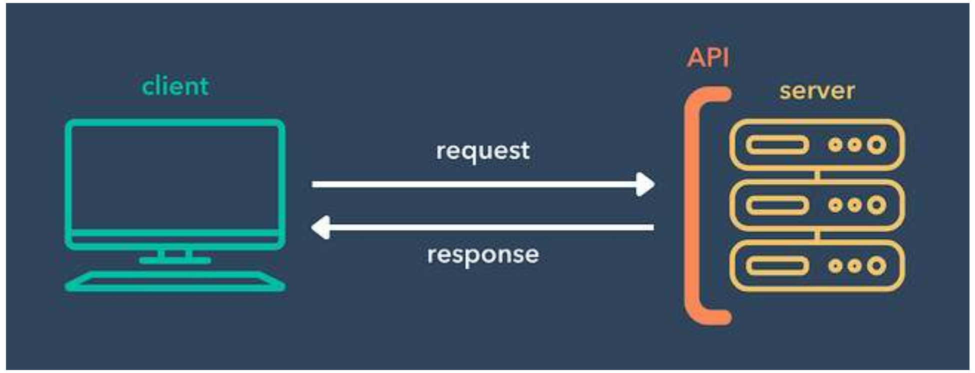


How does an API work?

For two software applications to integrate over the internet, one application — called the **client** — sends a **request** to the other application's **API**. Upon receiving and **validating** the client's request, the API performs the requested action, then sends a **response** back to the client. This response includes the status of the request (e.g., completed or denied) and any resources requested.

Steps:

1. Request
2. Validate
3. Respond



Making a simple API call

1. Find the URI of the external server or program

To make an API call, the first thing you need to know is the *Uniform Resource Identifier* (URI) of the server or external program from which you want the data. This is basically the digital equivalent of a home address.

The Cat Facts API URI, for example is https://catfact.ninja

2. Add an HTTP verb

Once you have the URI, then you need to know how to formulate the request.

The first thing you need to include is a request verb. The four most basic request verbs are:

- GET: To retrieve a resource
- POST: To create a new resource
- PUT: To edit or update an existing resource
- DELETE: To delete a resource

Making a simple API call (continue)

Example (Phyton)

Calling an API to get a random fun cat fact.

Request

```python
import requests

api_url = 'https://catfact.ninja/fact'
response = requests.get(api_url)

print(response.text)
```

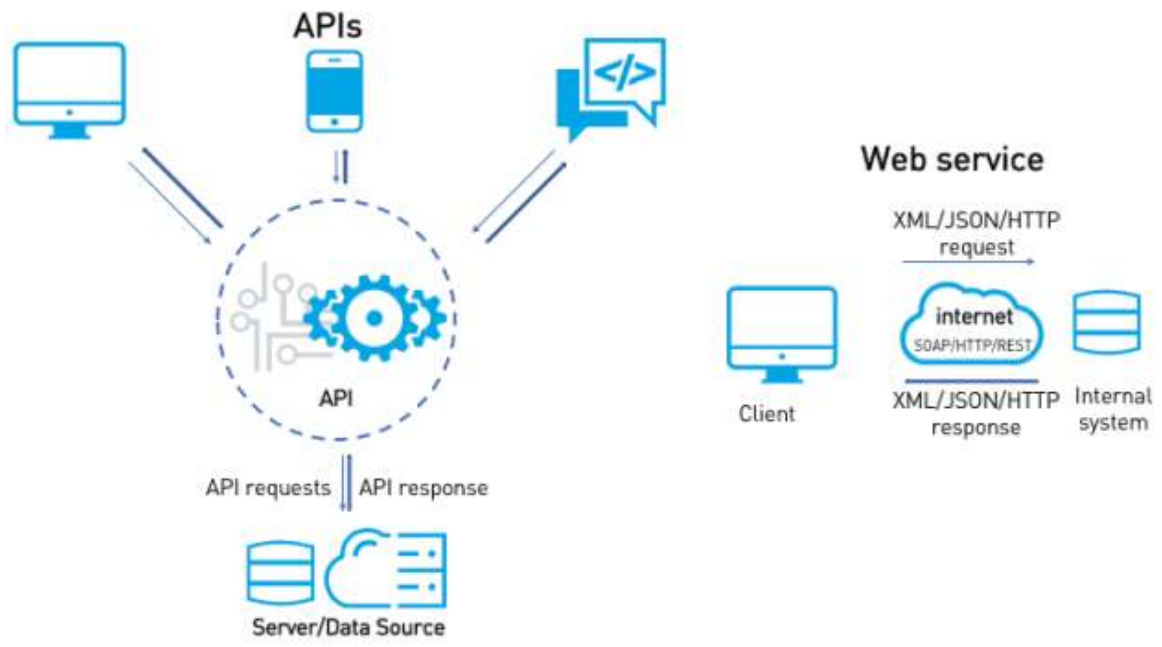
Response

```
{"fact":"Cats often overract to unexpected stimuli because of their extremely sensitive nervous system.","lengt
```

APIs vs. Web Services

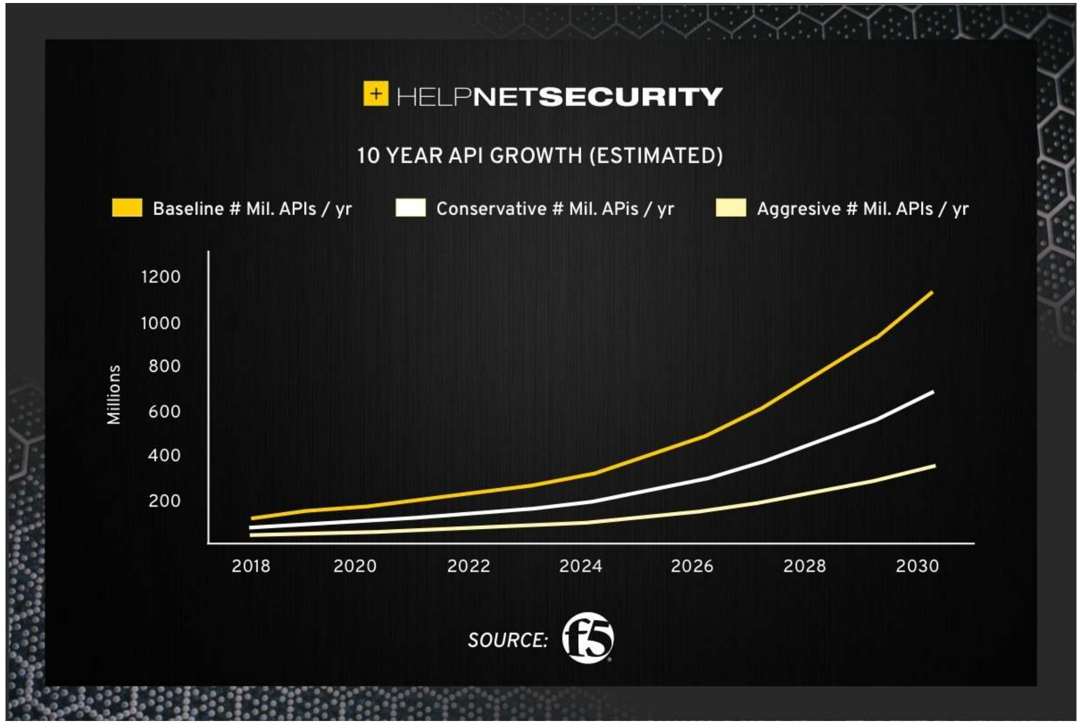
A **web service** is a software component that can be accessed and used to facilitate data transfers **via a web address**. Because a web service exposes an application's data and functionality to other applications, in effect, every web service is an API.

However, not every API is a web service. APIs are any software component that serves as an intermediary between two disconnected applications and does therefore not necessarily rely on using a web address.



Sprawl of APIs

APIs are increasingly recognised as a major driver of innovation, value creation, and revenue. From digital marketplaces and entertainment apps to the *Internet of Things* (IoT) and *Information Technology* (IT) microservices, APIs are at the heart of how the world conducts business. It is estimated by f5 -- leading technology company -- that the number of public and private APIs in 2021 was in the range of 200 million, and by 2031 that number could be in the billions.

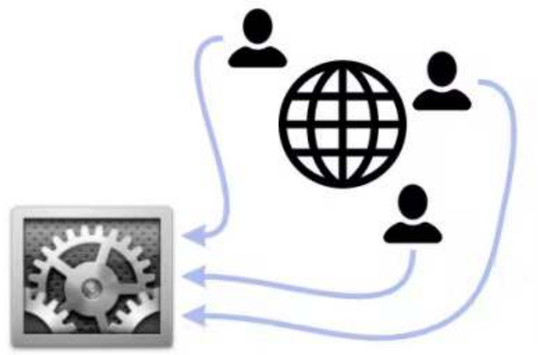


Different types of APIs

There are **four different types of APIs** commonly used: public, partner, private and composite. The API "type" indicates the intended scope of use.

1. Public/open/external

A public/open/external API is open and available for use by any outside developer or business. An enterprise that cultivates a business strategy that involves sharing its applications and data with other businesses will develop and offer a public API. These public APIs typically involve moderate authentication and authorisation. Enterprises might also seek to monetise their APIs by imposing a per-call cost to utilise the public API.

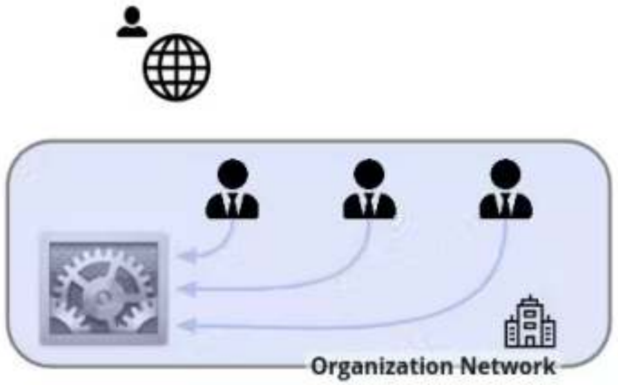


Public APIs

Different types of APIs (continue)

2. Internal/private

An internal/private API is intended only for use within the enterprise to connect systems/applications and data within the business. For example, an internal API might connect an organisation's payroll and *Human resources* (HR) systems. Internal APIs traditionally present weak security and authentication -- or none at all -- because the APIs are intended for internal use, and such security levels are assumed to be in place through other policies. This however is changing, as greater threat awareness and regulatory compliance demands increasingly influence an organisation's API strategy.

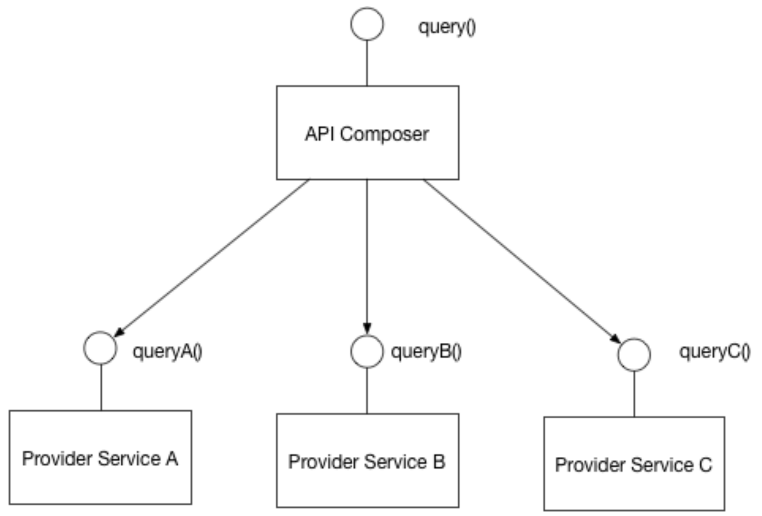


Private APIs

Different types of APIs (continue)

3. Composite

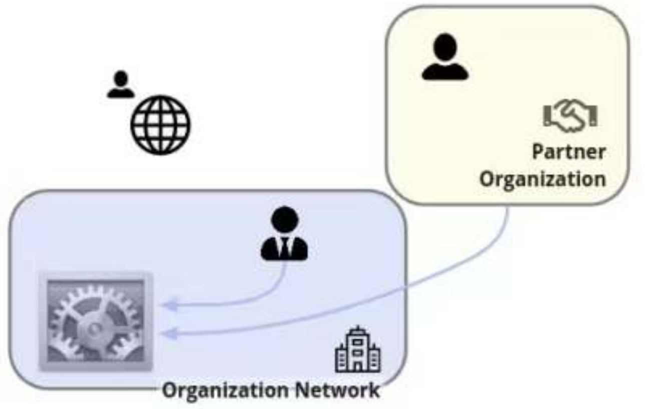
Composite APIs generally combine two or more APIs to craft a sequence of related or interdependent operations. Composite APIs can be beneficial to address complex or tightly related API behaviours and can sometimes improve speed and performance over individual APIs.



Different types of APIs (continue)

4. Partners

A partner API, only available to specifically selected and authorised outside developers or API consumers, is a means to facilitate business-to-business activities. For example, if a business wants to selectively share its customer data with outside *Customer Relationship Management* (CRM) firms, a partner API can connect the internal customer data system with those external parties -- no other API use is permitted. Partners have clear rights and licences to access such APIs. For this reason, partner APIs generally incorporate stronger authentication, authorisation and security mechanisms. Enterprises also typically do not monetise such APIs directly; partners are paid for their services rather than through API use.



Partner APIs

Why are APIs used?

Benefits of using APIs

There are several benefits to using APIs when setting up couplings between systems. **Five main benefits** of APIs are:

1. Improved collaboration

The average enterprise uses several different applications, many of which are disconnected. APIs enable integration so that these platforms and apps can seamlessly communicate with one another. Through this integration, companies can automate workflows and improve workplace collaboration and restrict the formation of information silos that compromise productivity and performance.

2. Accelerated innovation

APIs offer flexibility, allowing companies to make connections with new business partners, offer new services to their existing market, and, ultimately, access new markets that can generate massive returns and drive digital transformation.

Benefits of using APIs (continue)

3. Data monetisation

Many companies choose to offer APIs for free, at least initially, so that they can build an audience of developers around their brand and forge relationships with potential business partners. If the API grants access to valuable digital assets, the business can monetise it by selling access. This monetisation of APIs is known as the API economy.

4. System security

APIs separate the requesting application from the infrastructure of the responding service, and offer layers of security between the two as they communicate.

5. End-user security and privacy

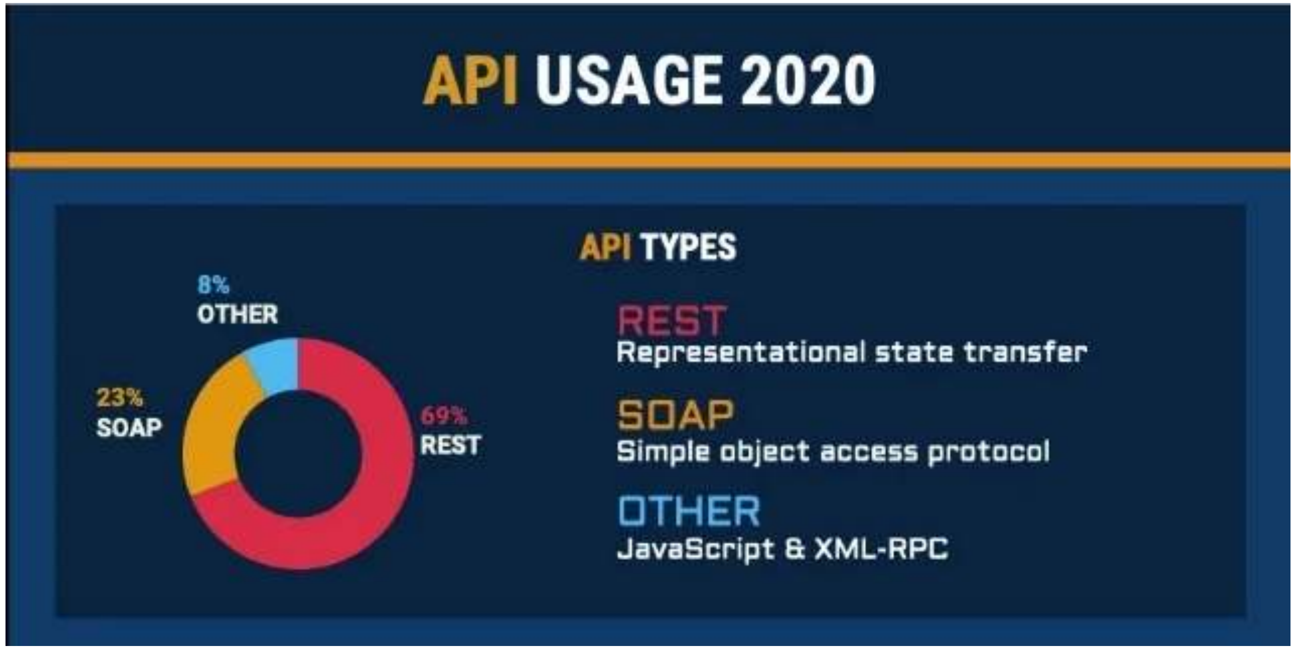
APIs also provide another layer of protection for personal users. When a website requests a user's location, which is provided via a location API, the user can then decide whether to allow or deny this request. Many web browsers and mobile operating systems have permission structures built-in when APIs request access to applications and their data. When the app must access files through an API, the operating system will use permissions for that access.

Protocols/Architectures

What are API protocols/architectures?

APIs exchange commands and data, and this requires clear protocols or architectures -- the rules, structures and constraints that govern an API's operation.

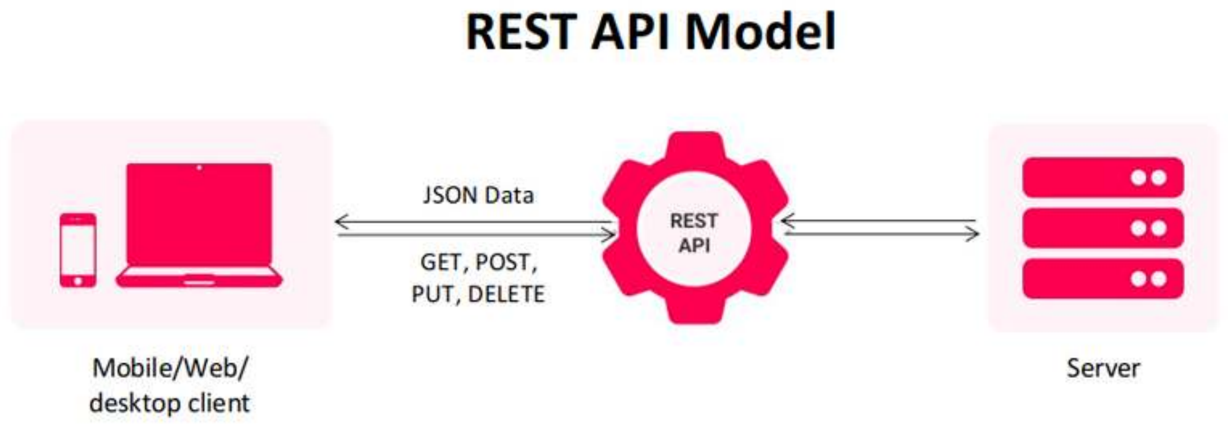
Today, there are four main categories of API protocols or architectures: REST, RPC, SOAP and GraphQL (there are a few other, less common protocols as well). Each of these architectures has unique characteristics and trade offs and is employed for different purposes.



Common protocols/architectures

1. REST

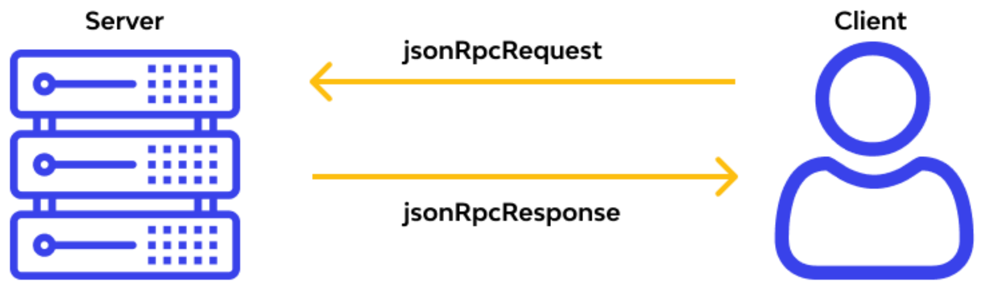
The *Representational State Transfer* (REST) architecture is perhaps the most popular approach to building APIs. REST relies on a client/server approach that separates front and back ends of the API and provides considerable flexibility in development and implementation. REST is stateless, which means the API stores no data or status between requests. REST supports caching, which stores responses for slow or non-time-sensitive APIs. REST APIs, usually termed RESTful APIs, can also communicate directly or operate through intermediate systems such as API gateways and load balancers.



Common protocols/architectures (continue)

2. RPC

The *Remote Procedural Call* (RPC) protocol is a simple means to send multiple parameters and receive results. RPC APIs invoke executable actions or processes, while REST APIs mainly exchange data or resources such as documents. RPC can employ two different languages, *JavaScript Object Notation* (JSON) and *eXtensible Markup Language* (XML), for coding; these APIs are dubbed JSON-RPC and XML-RPC, respectively.



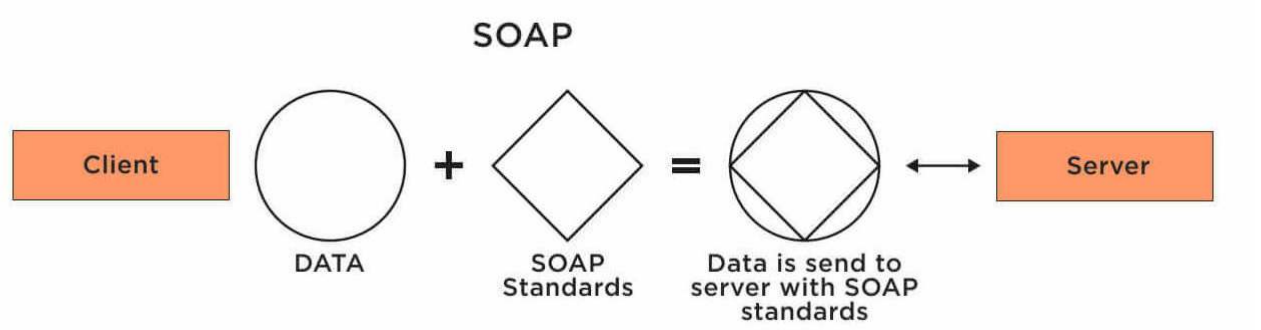
Example:

request:
{"method":"my_method","params":[1,2,3],"id":"my_id"}
response:
{"result":"my_result","error":null,"id":"my_id"}

Common protocols/architectures (continue)

3. SOAP

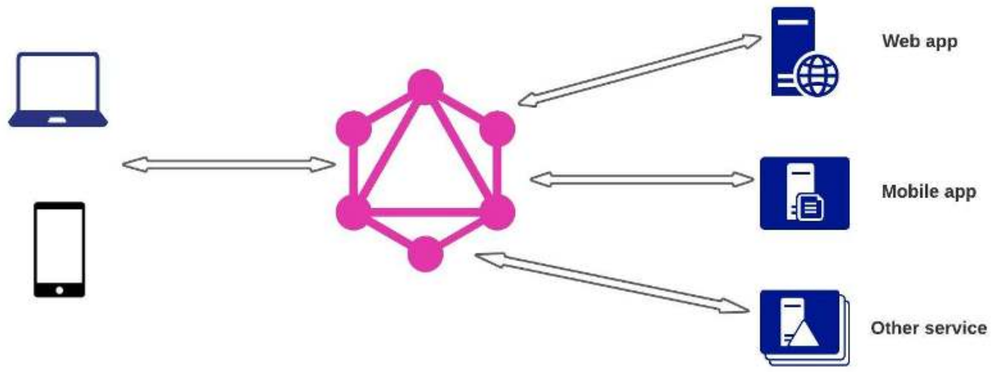
The *Simple Object Access Protocol* (SOAP) is a messaging standard defined by the *World Wide Web* (WWW) Consortium and broadly used to create web APIs, usually with XML. SOAP supports a wide range of communication protocols found across the internet, such as *Hypertext Transfer Protocol* (HTTP), *Simple Mail Transfer Protocol* (SMTP) and *Transmission Control Protocol/Internet Protocol* (TCP/IP). SOAP is also extensible and style-independent, which enables developers to write SOAP APIs in varied ways and easily add features and functionality. The SOAP approach defines how the SOAP message is processed, the features and modules included, the communication protocol(s) supported and the construction of SOAP messages.



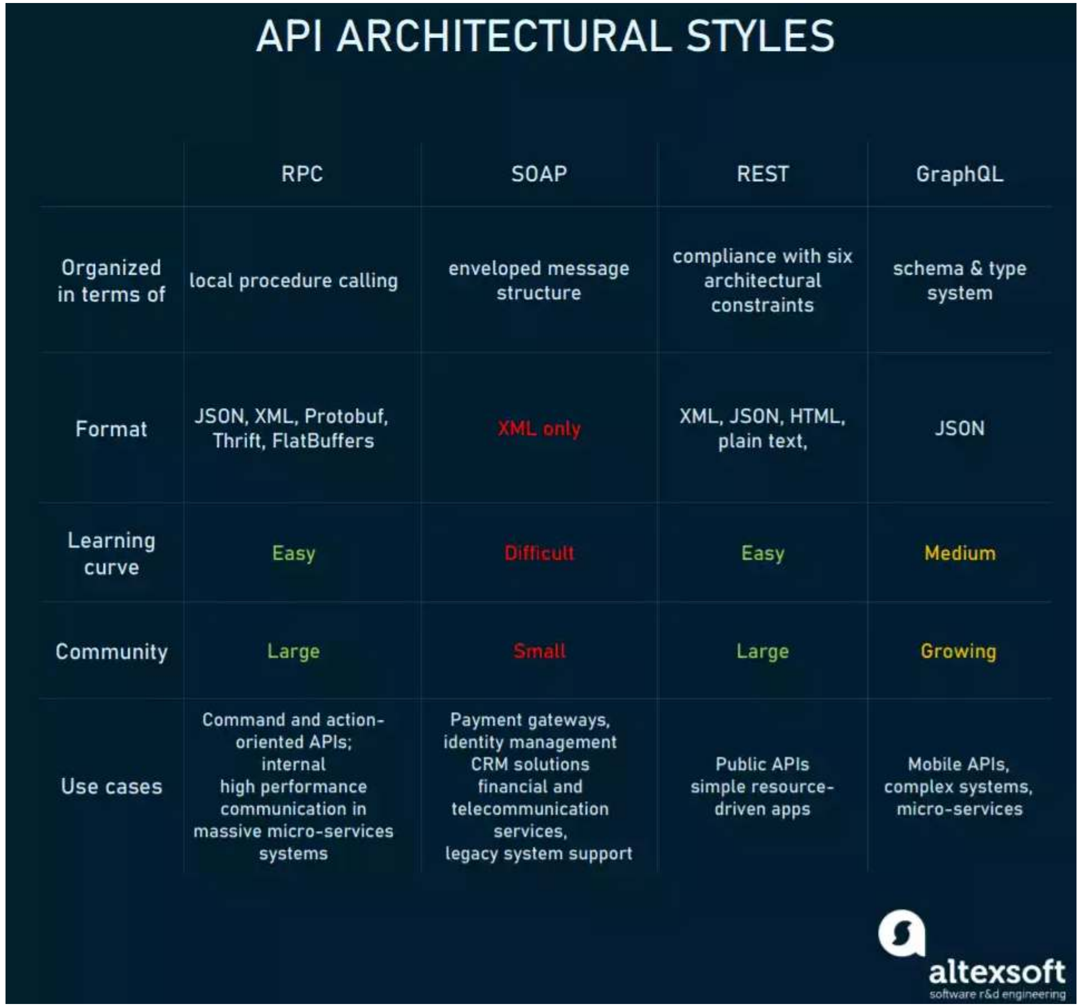
Common protocols/architectures (continue)

4. GraphQL

GraphQL is a query language and server-side runtime for APIs that prioritises giving clients exactly the data they request and no more. GraphQL is designed to make APIs fast, flexible, and developer-friendly. It can even be deployed within an *Integrated Development Environment* (IDE) known as GraphiQL. As an alternative to REST, GraphQL lets developers construct requests that pull data from multiple data sources in a single API call. Additionally, GraphQL gives API maintainers the flexibility to add or deprecate fields without impacting existing queries. Developers can build APIs with whatever methods they prefer, and the GraphQL specification will ensure they function in predictable ways to clients.



Diffrence between protocols/architectures



API ARCHITECTURAL STYLES

	RPC	SOAP	REST	GraphQL
Organized in terms of	local procedure calling	enveloped message structure	compliance with six architectural constraints	schema & type system
Format	JSON, XML, Protobuf, Thrift, FlatBuffers	XML only	XML, JSON, HTML, plain text,	JSON
Learning curve	Easy	Difficult	Easy	Medium
Community	Large	Small	Large	Growing
Use cases	Command and action-oriented APIs; internal high performance communication in massive micro-services systems	Payment gateways, identity management CRM solutions financial and telecommunication services, legacy system support	Public APIs simple resource-driven apps	Mobile APIs, complex systems, micro-services

altexsoft
software r&d engineering

Choosing the correct API

Whatever API protocol/architecture you choose to use there are several important factors that developers should consider when choosing an API.

1. Clear and complete documentation

APIs are software, and like any software, they require comprehensive documentation that provide developers with how-to guidance, reference usage and example use cases designed to help developers apply the API quickly and successfully.

2. Easy adoption

Choose a simple API, establish an easy method of acquiring the API and ensure solid and knowledgeable API support that can address any developer questions.

Choosing the correct API (continue)

3. Ease of use

A good API is simply easy to use with sensible and intuitive call structures. Simplicity, consistency, clarity and backward compatibility -- involving clear deprecation -- are hallmarks of a good API.

4. Stability and reliability

Good APIs are developed just like any other software, this includes comprehensive testing for bugs and clear metrics for scalability and performance.

5. Security

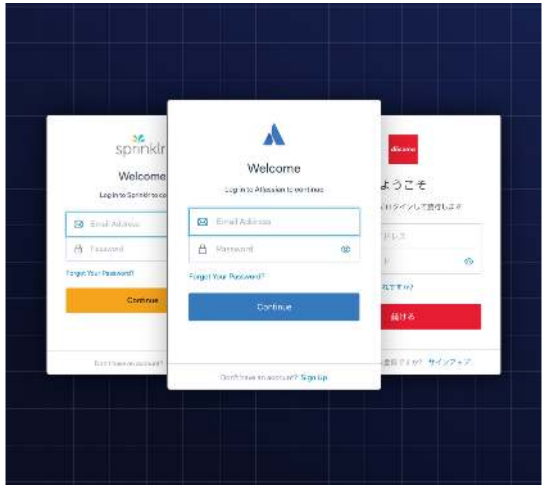
APIs must support security through clear authentication -- where only authorised users can use the API. In addition, any data exchanged across the API should be encrypted or otherwise shielded from snooping and theft.

APIs in action

APIs have become a valuable aspect of modern business that we as users interact with every day. Here are some popular examples of API uses that users encounter almost every day.

1. Universal logins

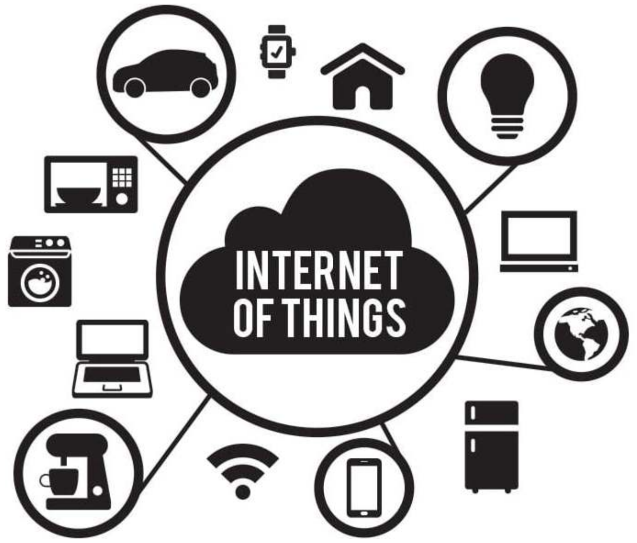
A popular API example is the function that enables people to log in to websites by using their Facebook, Twitter, or Google profile login details. This convenient feature allows any website to leverage an API from one of the more popular services for quick authentication, saving them the time and hassle of setting up a new profile for every web application or new membership.



APIs in action (continue)

2. Internet of Things (IoT)

These “smart devices” offer added functionality, such as internet-enabled touchscreens and data collection, through APIs. For example, a smart fridge can connect to recipe applications or take and send notes to mobile phones via text message. Internal cameras connect to various applications so that users can see the contents of the refrigerator from anywhere.



APIs in action (continue)

3. Travel booking comparisons

Travel booking sites aggregate thousands of flights, showcasing the cheapest options for every date and destination. This service is made possible through APIs that provide application users with access to the latest information about availability from hotels and airlines, either via a web browser or the travel booking company's own application. With an autonomous exchange of data and requests, APIs drastically reduce the time and effort involved in checking for available flights or accommodation.

Cheapflights

APIs in action (continue)

4. Mapping apps

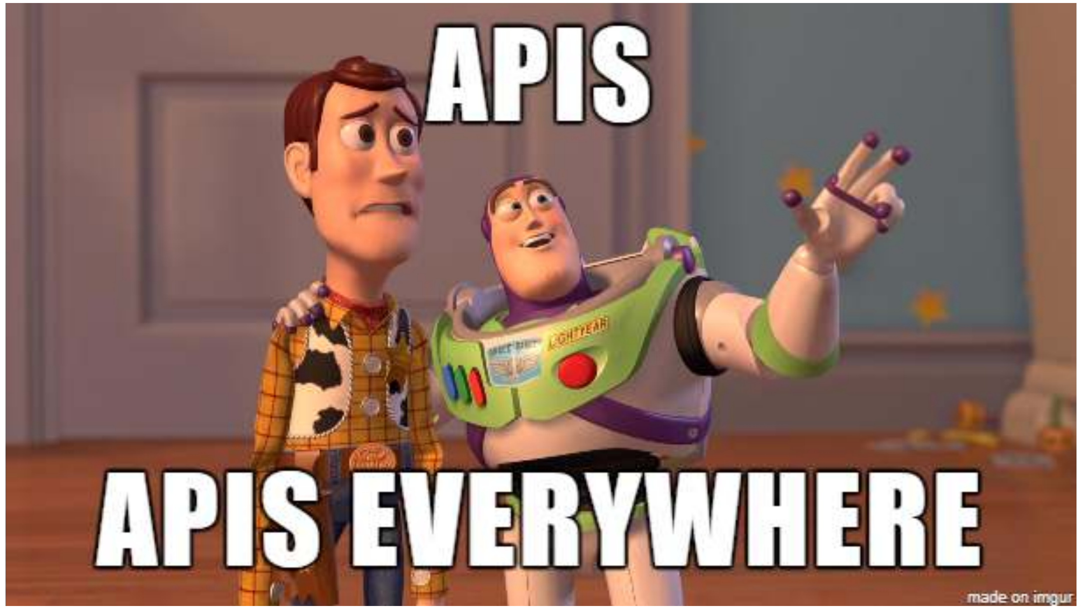
In addition to the core APIs that display static or interactive maps, these apps use other APIs and features to provide users with directions, speed limits, points of interest, traffic warnings and more. Users communicate with an API when plotting travel routes or tracking items on the move, such as a delivery vehicle.

Google Maps

APIs in action (continue)

5. SaaS applications

APIs are an integral part of the growth in *Software-as-a-Service* (SaaS) products. Platforms like *Customer Relationship Management Tools* (CRM) often include a number of built-in APIs that let companies integrate with applications they already use, such as messaging, social media, and email apps. This drastically reduces time spent switching between applications for sales and marketing tasks. It also helps reduce or prevent data silos that may exist between departments using different applications.

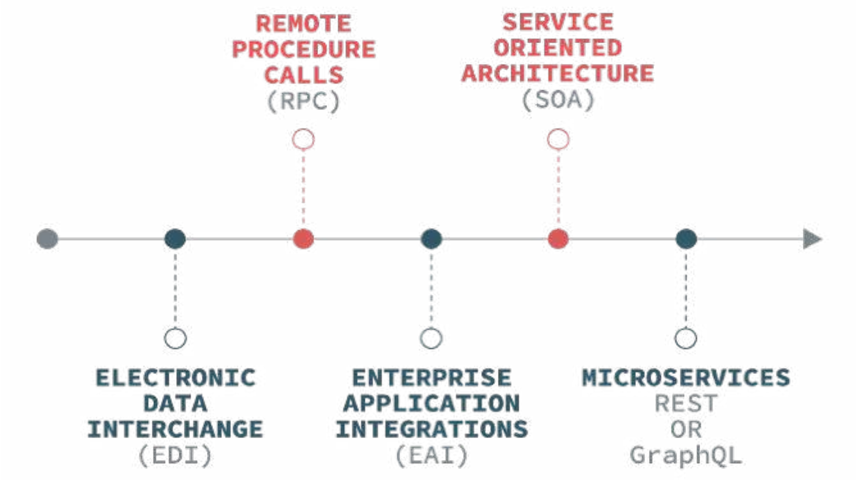


REST API

History

REST was developed towards the end of the **1990s** and fundamentally changed the API landscape. The first companies to use a REST API were **eBay** and Amazon. Only a selection of partners got access to eBay's well documented and user-friendly REST API. As a result, eBay's marketplace was not only accessible through direct visits but through any website that accessed the eBay API.

Flickr also launched a REST API in **2004**, just in time for the rise of social networking and blogs on the web. This paved the way for social sharing, which Facebook and Twitter later joined.


REMOTE PROCEDURE CALLS (RPC)
SERVICE ORIENTED ARCHITECTURE (SOA)
ELECTRONIC DATA INTERCHANGE (EDI)
ENTERPRISE APPLICATION INTEGRATIONS (EAI)
MICROSERVICES REST OR GraphQL

The six principles of REST

The REST API is founded on **6 principles**.

1. Client-server architecture

The principle behind the client-server architecture is the separation of problems. Dividing the user interface from data storage improves the portability of that interface across multiple platforms. It also has the advantage that different components can be developed independently from each other.

2. Statelessness

Statelessness means that the communication between client and server always contains all the information needed to execute the request. There is no session state on the server, it is kept entirely on the client. If access to a resource requires authentication, the client must authenticate itself on each request.

The six principles of REST (continue)

3. Caching

The client, server, and any intermediate components can cache all resources to improve performance. The information can be classified as cacheable or non-cacheable.

4. Uniform interface

All components of a RESTful API have to follow the same rules to communicate with each other. This also makes it easier to understand interactions between the various components of a system.

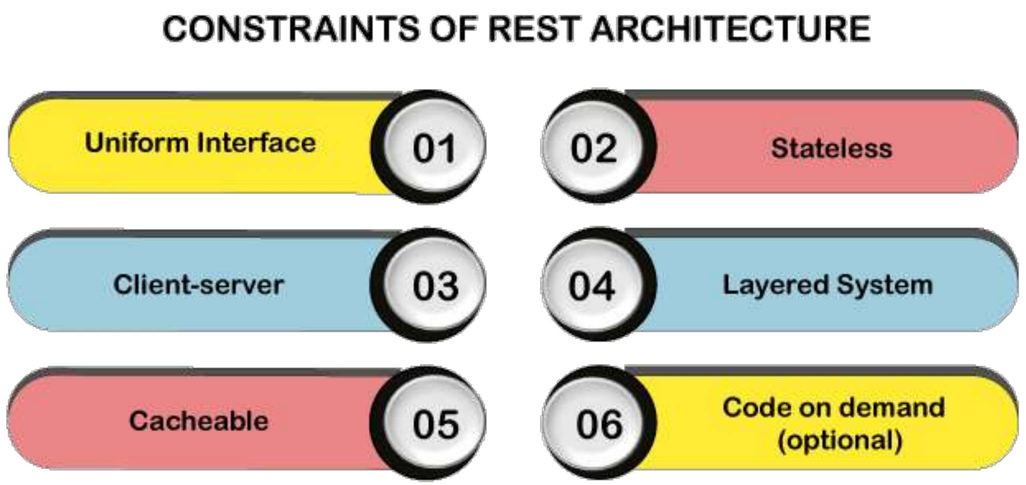
5. Layered system

Individual components cannot see beyond the immediate level they interact with. This means that a client that connects to an intermediate component such as a proxy does not know what is behind it. Therefore, components can be easily exchanged or expanded independently of each other.

The six principles of REST (continue)

6. Code-on-demand

Additional code can be downloaded to extend client functionality. However, this is optional because the client may not be able to download or execute this code.



The advantages of REST

The complete separation of the user interface from server and data storage offers some advantages for the development of an API.

Example:

- Improves the portability of the interface
- Increases project scalability
- Indipendent development
- Increases overall flexibility

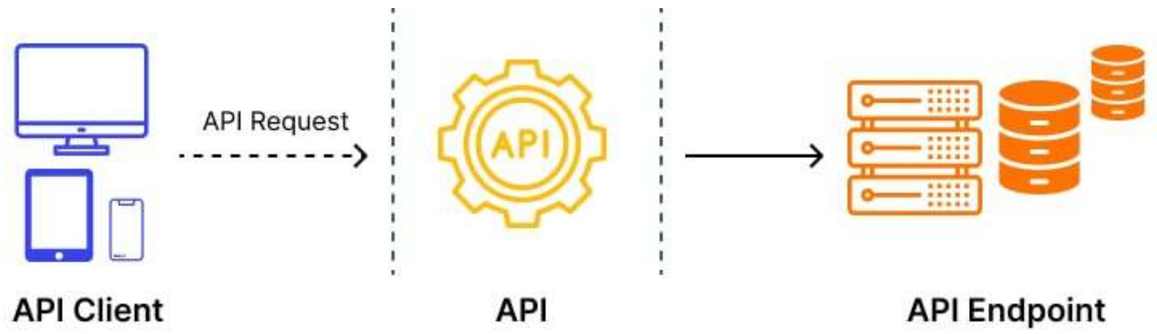
A REST API is always independent of the type of platform or languages used, it adapts to the type of syntax or platform used. This provides great freedom when changing or testing new environments within a development. You can use **PHP**, **Java**, **Python** or **Node.js** servers with a REST API. Only responses to requests must be in the language used for information exchange, usually XML or JSON.

What is an API Endpoint?

Definition

An endpoint is one end of a communication channel. When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or service. Each endpoint is the location from which APIs can access the resources they need to carry out their function.

APIs typically allow access to many different resources on a server. These different resources can be specified by using the correct endpoint. In their requests, clients specify an endpoint as a URL. This URL tells the server, “The resource I want is at this location.” The process is similar to how you access web pages in a browser. Web browsers load web pages by sending a URL to a web server, and the server responds with the requested page. Similarly, the client needs the right endpoint URL to request a particular resource from an API.



Endpoint vs API

It’s important to note that endpoints and APIs are different. An endpoint is a component of an API, while an API is a set of rules that allow two applications to share resources. Endpoints are the locations of the resources, and the API uses endpoint URLs to retrieve the requested resources.

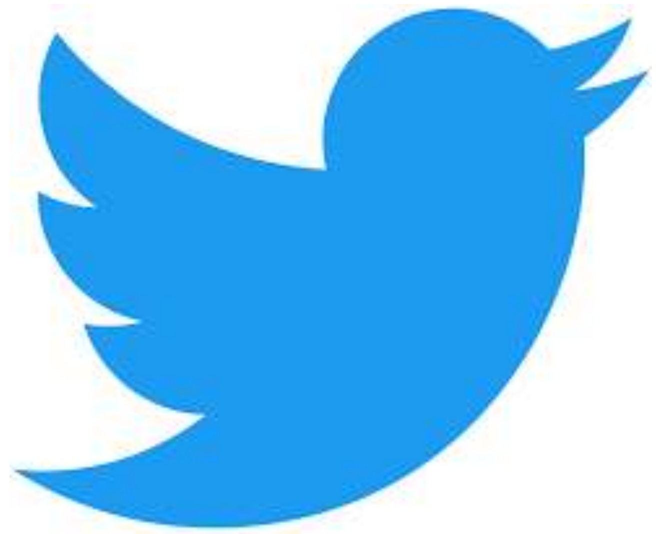
Why are API endpoints important?

Without properly structured and functioning endpoints, an API will be confusing at best and broken at worst. As you make more data available through your API, it’s vital to ensure that each endpoint provides valuable resources for clients.

Examples of API endpoints

Twitter

The Twitter API exposes data about tweets, direct messages, users, and more. Let's say you want to retrieve the content of a specific tweet. To do this, you can use the tweet lookup endpoint, which has the URL https://api.twitter.com/2/tweets/{id} (where {id} is the unique identifier of the tweet). Now, say you want your website to stream public tweets in real-time so your visitors stay informed on a specific topic. You can use Twitter's filtered stream endpoint, whose URL is https://api.twitter.com/2/tweets/search/stream.



Examples of API endpoints (continue)

Spotify

Spotify's API gives developers access to song, artist, playlist, and user data. For example, if you want to get a specific album, you can access any album in the Spotify catalog with the endpoint https://api.spotify.com/v1/albums/{id} (where {id} is the album's unique identifier). Or, say you want to send a request that makes a user follow a playlist. In this case, send a PUT request with the endpoint https://api.spotify.com/v1/playlists/{playlist_id}/followers (where {playlist_id} is the unique identifier of the playlist).

API Standards

REST API design best practices

Use JSON as the format for sending and receiving data

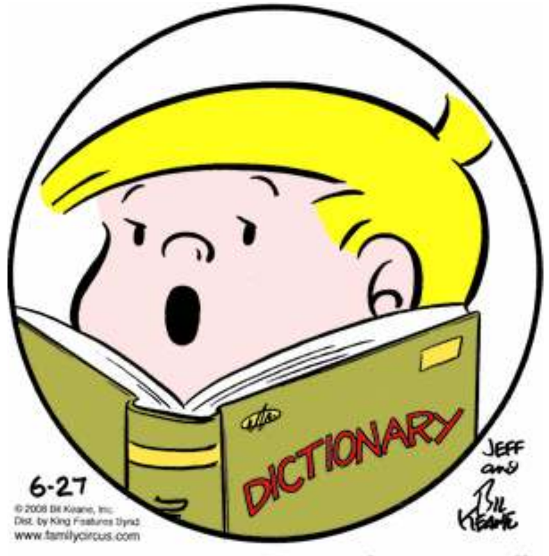
In the past, accepting and responding to API requests were done mostly in XML and even *HyperText Markup Language* (HTML). But these days, JSON has largely become the de-facto format for sending and receiving API data. This is because, with XML, for example, it's often a bit of a hassle to decode and encode data -– so XML isn’t widely supported by frameworks anymore.

REST API design best practices (continue)

Use nouns instead of verbs in endpoints

When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does.

This is because HTTP methods such as GET, POST, PUT, PATCH, and DELETE are already in verb form for performing basic Create, Read, Update, Delete (CRUD) operations. The HTTP



“That’s weird. ‘VERB’ is a noun.”

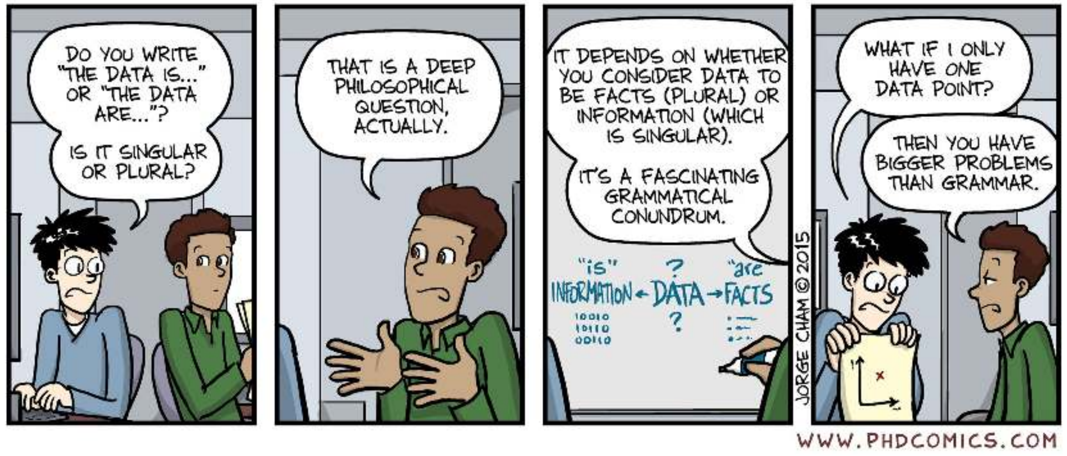
REST API design best practices (continue)

Name collections with plural nouns

You can think of the data of your API as a collection of different resources from your consumers.

If you have an endpoint like https://mysite.com/post/123, it might be okay for deleting a post with a DELETE request or updating a post with PUT or PATCH request, but it doesn't tell the user that there could be some other posts in the collection. This is why your collections should use plural nouns.

So, instead of https://mysite.com/post/123, it should be https://mysite.com/posts/123.


DO YOU WRITE "THE DATA IS..." OR "THE DATA ARE..."?
IS IT SINGULAR OR PLURAL?
THAT IS A DEEP PHILOSOPHICAL QUESTION, ACTUALLY.
IT DEPENDS ON WHETHER YOU CONSIDER DATA TO BE FACTS (PLURAL) OR INFORMATION (WHICH IS SINGULAR).
IT'S A FASCINATING GRAMMATICAL CONUNDRUM.
"is" INFORMATION ← DATA → FACTS "are"
WHAT IF I ONLY HAVE ONE DATA POINT?
THEN YOU HAVE BIGGER PROBLEMS THAN GRAMMAR.
JORGE CHAM © 2015
WWW.PHDCOMICS.COM

REST API design best practices (continue)

Use status codes in error handling

You should always use regular HTTP status codes in responses to requests made to your API. This will help your users to know what is going on – whether the request is successful, if it fails, or something else.

Common error HTTP status codes include:

- 400 Bad Request – This means that client-side input fails validation
- 401 Unauthorised – This means the user isn't not authorised to access a resource. It usually returns when the user isn't authenticated
- 403 Forbidden – This means the user is authenticated, but it's not allowed to access a resource
- 404 Not Found – This indicates that a resource is not found
- 500 Internal server error – This is a generic server error
- 502 Bad Gateway – This indicates an invalid response from an upstream server
- 503 Service Unavailable – This indicates that something unexpected happened on the server side (It can be anything like server overload, some parts of the system failed, etc.)

REST API design best practices (continue)

Use nesting on endpoints to show relationships

Oftentimes, different endpoints can be interlinked, therefore you should nest them so that it is easier to understand them.

For example, in the case of a multi-user blogging platform, different posts could be written by different authors, so an endpoint such as https://mysite.com/posts/author would make a valid nesting in this case.

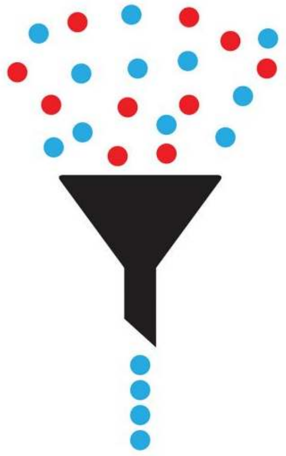
In the same vein, the posts might have their individual comments, so to retrieve the comments, an endpoint like https://mysite.com/posts/postId/comments would make sense. You should avoid nesting that is more than 3 levels deep as this can make the API less elegant and readable.

REST API design best practices (continue)

Use filtering, sorting, and pagination to retrieve the data requested

Due to the size of databases these days retrieving data can be very slow. Filtering, sorting, and pagination are all actions that can be performed on the collection of a REST API that can speed up requests. This lets it only retrieve, sort, and arrange the necessary data into pages so the server doesn't get too occupied with requests.

An example of a filtered endpoint is the one below: https://mysite.com/posts?tags=javascript This endpoint will fetch any post that has a tag of JavaScript.



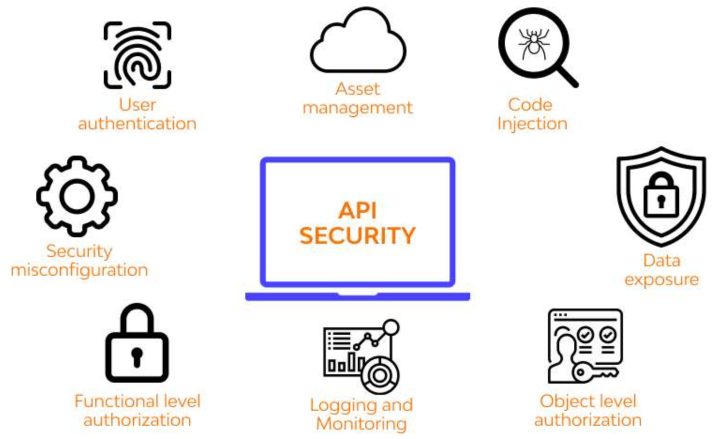
REST API design best practices (continue)

Use SSL for security

Secure Sockets Layer (SSL) is crucial for security in REST API design. This security layer will make your API less vulnerable to malicious attacks. The clear difference between the URL of a REST API that runs over SSL and the one which does not is the "s" in HTTP: https://mysite.com/posts runs on SSL. http://mysite.com/posts does not run on SSL.

Other security measures:

- Making the communication between server and client private
- Consumers only get what they request



REST API design best practices (continue)

Clear Versioning

REST APIs should have different versions, so you don’t force clients (users) to migrate to new versions. This might even break the application if you're not careful.

Cache data

We can add caching to return data from the local memory cache instead of querying the database to get the data every time we want to retrieve some data that users request. The good thing about caching is that users can get data faster. However, the data that users get may be outdated. This may also lead to issues when debugging in production environments when something goes wrong as we keep seeing old data.

If you are using caching, you should also include Cache-Control information in your headers. This will help users effectively use your caching system..

REST API design best practices (continue)

Accurate API documentation

When you make a REST API, you need to help clients (consumers) learn and figure out how to use it correctly. The best way to do this is by providing good documentation for the API.

The documentation should contain:

- Relevant endpoints of the API
- Example requests of the endpoints
- Implementation in several programming languages
- Messages listed for different errors with their status codes

One of the most common tools you can use for API documentation is Swagger. You can also use Postman, one of the most common API testing tools in software development, to document your APIs.

What is OpenAPI?

OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs.

An OpenAPI file includes:

- Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
- Operation parameters input and output for each operation
- Authentication methods
- Contact information, license, terms of use and other information

API specifications can be written in *YAML Ain't Markup Language* (YAML) or JSON, the format is easy to learn and readable to both humans and machines.

Open API
Specification

What is Swagger?

Swagger is a set of open-source tools built around the OpenAPI Specification that can help you design, build, document and consume REST APIs.

Swagger tools include:

- Swagger Editor – browser-based editor where you can write OpenAPI definitions
- Swagger UI – renders OpenAPI definitions as interactive documentation
- Swagger Codegen – generates server stubs and client libraries from an OpenAPI definition
- Swagger Editor Next (beta) – browser-based editor where you can write and review OpenAPI and AsyncAPI
- Swagger Core – Java-related libraries for creating, consuming, and working with OpenAPI definitions
- Swagger Parser – standalone library for parsing OpenAPI definitions
- Swagger APIDom – provides a single, unifying structure for describing APIs across various description languages and serialization formats

Interacting with APIs

Definition

One can interact with APIs through several different avenues, with these interactions ranging from just using existing APIs to building and testing your own. These avenues include using packages or libraries in your favourite programming languages or purpose built software programs from third parties that are also known as API management tools. The choice between using a free package/library or custom is not always straightforward.

Using free packages/libraries will have a low bar of entry with regards to money needed for startup but a higher bar of entry with regards to the skills required to use it. API management tools will normally have an easy to understand UI and good long term software maintenance but in most cases will ask some sort of fee to use.

Examples of API management tools

Mulesoft

Mulesoft's Anypoint Platform can be used to quickly design, test, and publish API products. Manage APIs, monitor and analyse usage, control access, and protect sensitive data with security policies. While Anypoint API Community Manager, provides self-service API documentation, forums, support, and personalised resources developers need to be successful.

Examples of API management tools (continue)

IBM API Connect

IBM API Connect lets you expertly secure and manage your entire API ecosystem across multiple clouds —- including boosting socialisation and monetisation efforts. IBM API Connect is a complete, intuitive and scalable API platform that lets you create, expose, manage and monetise APIs across clouds.

Examples of API management tools (continue)

Axway

The Axway API Management Platform provides a comprehensive platform for managing, delivering, and securing APIs. It provides integration, acceleration, governance, and security for Web API and SOA-based systems. Axway offers the full lifecycle API management for the next generation, as well as automating the discovery, reuse, and governance of all your APIs across multiple gateways, environments, and vendor solutions.

Examples of API management tools (continue)

Postman

Postman is a collaboration platform for API development. Postman's features simplify each step of building an API and streamline collaboration so you can create better APIs. Quickly and easily send REST, SOAP, and GraphQL requests directly within Postman. Automate manual tests and integrate them into your CI/CD pipeline to ensure that any code changes won't break the API in production.

Python

What is Python?

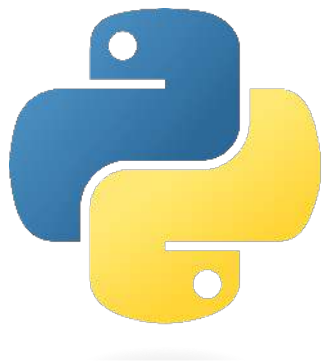
Python is a popular open source general-purpose programming language that can be used for a wide variety of applications.

History

Python was created by Guido van Rossum, and first released on February 20, 1991. Python is maintained by the Python Software Foundation, a non-profit membership organization and a community devoted to developing, improving, expanding, and popularizing the Python language and its environment.

Why use Python?

- Ease of Comprehension
- It is easy to obtain, install and deploy
- Used in many industries
- Flexibility



Consuming and building APIs using Python

Consuming

To write code that interacts with REST APIs, most **Python** developers turn to module called **requests** to send HTTP requests. This library abstracts away the complexities of making HTTP requests.

Building

You can create APIs using different frameworks -- Python frameworks automate the implementation of several tasks and give developers a structure for application development.

Popular API frameworks

- Flask Restful
- Eve
- Django REST
- Falcon
- FastAPI

FastAPI

More about FastAPI

FastAPI is one of the most efficient and high-performance Python API frameworks. It has a compact coding structure that claims to allow code to be developed 200% to 300% faster than with other API development frameworks.

Uses of FastAPI

The tool is primarily used to build asynchronous web applications, as it is founded on Asynchronous JavaScript and XML. It also features a Swagger user interface to call and test APIs from a browser.

Why use FastAPI

There are many reasons to use the FastAPI framework for API development but some of the mains ones are listed below:

- Supports an intuitive editor and VSCode/PyCharm
- Integrated security and authentication and dependency injection system
- Fully compatible with Starlette and Pydantic

APIs using VSCode

What is VSCode?

Visual Studio Code is a free, lightweight but powerful source code editor that runs on your desktop and on the web and is available for Windows, macOS, Linux, and Raspberry Pi OS. It comes with built-in support for JavaScript, TypeScript, and Node.js and has a rich ecosystem of extensions for other programming languages (such as C++, C#, Java, Python, PHP, and Go), runtimes (such as .NET and Unity), environments (such as Docker and Kubernetes), and clouds (such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform).

VSCode to interact with APIs

VSCode has several different extensions that can be installed that make it easy for developers test APIs. The two most popular extensions are:

REST Client

REST Client allows you to send HTTP request and view the response in Visual Studio Code directly.

What can it do?

- Send HTTP Requests directly from VS Code
- GraphQL support
- cURL commands support
- Multiple requests per file
- Different authentication protocols support
- Environments and variables support
- Generate code snippets for requests
- Cookies memory

VSCode to interact with APIs (continue)

Thunder Client

Thunder Client is a lightweight Rest API Client Extension for Visual Studio Code, and-crafted by Ranga Vadhineni with simple and clean design

Main features

- UI resembling Postman
- Collections and Environment variables
- GraphQL support
- Scriptless API Testing
- Git sync
- Import from Postman, Insomnia, etc.

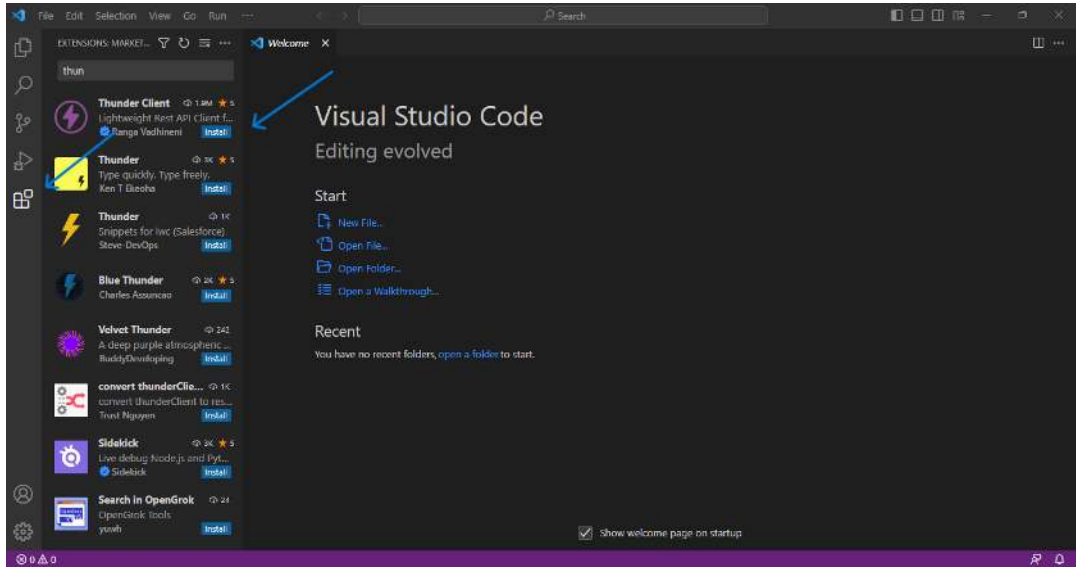
Example: Thunder Client API

Step 1

Download VSCode: https://code.visualstudio.com/

Step 2

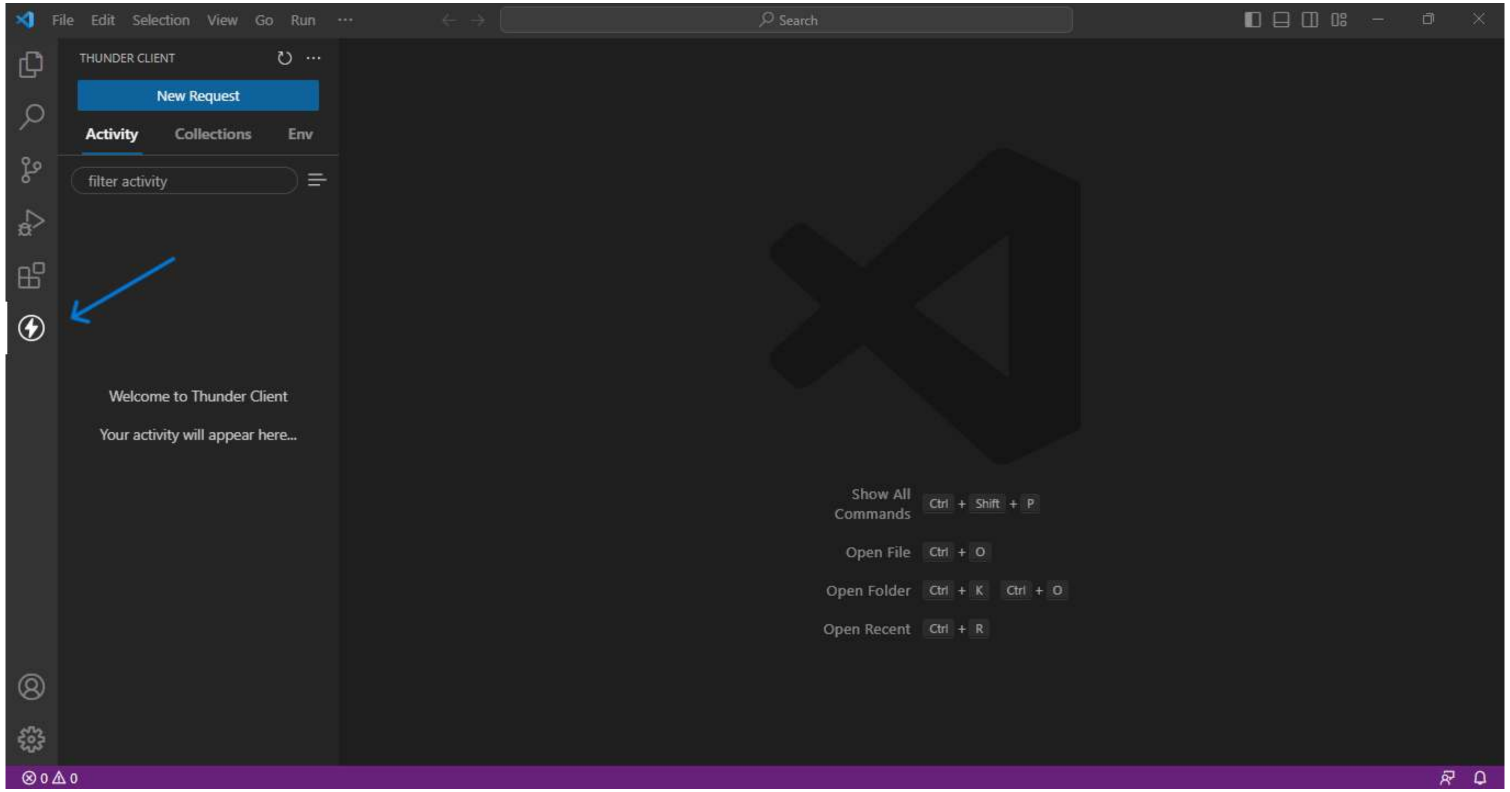
Download Thunder Client: To download Thunder Client, you can find it on VS Code marketplace. Just search for "Thunder Client" when you're prompted and then install it.



Example: Thunder Client API (continue)

Step 3

Launch Thunder Client: Click on the new icon that's been added in VS Code to launch Thunder Client.



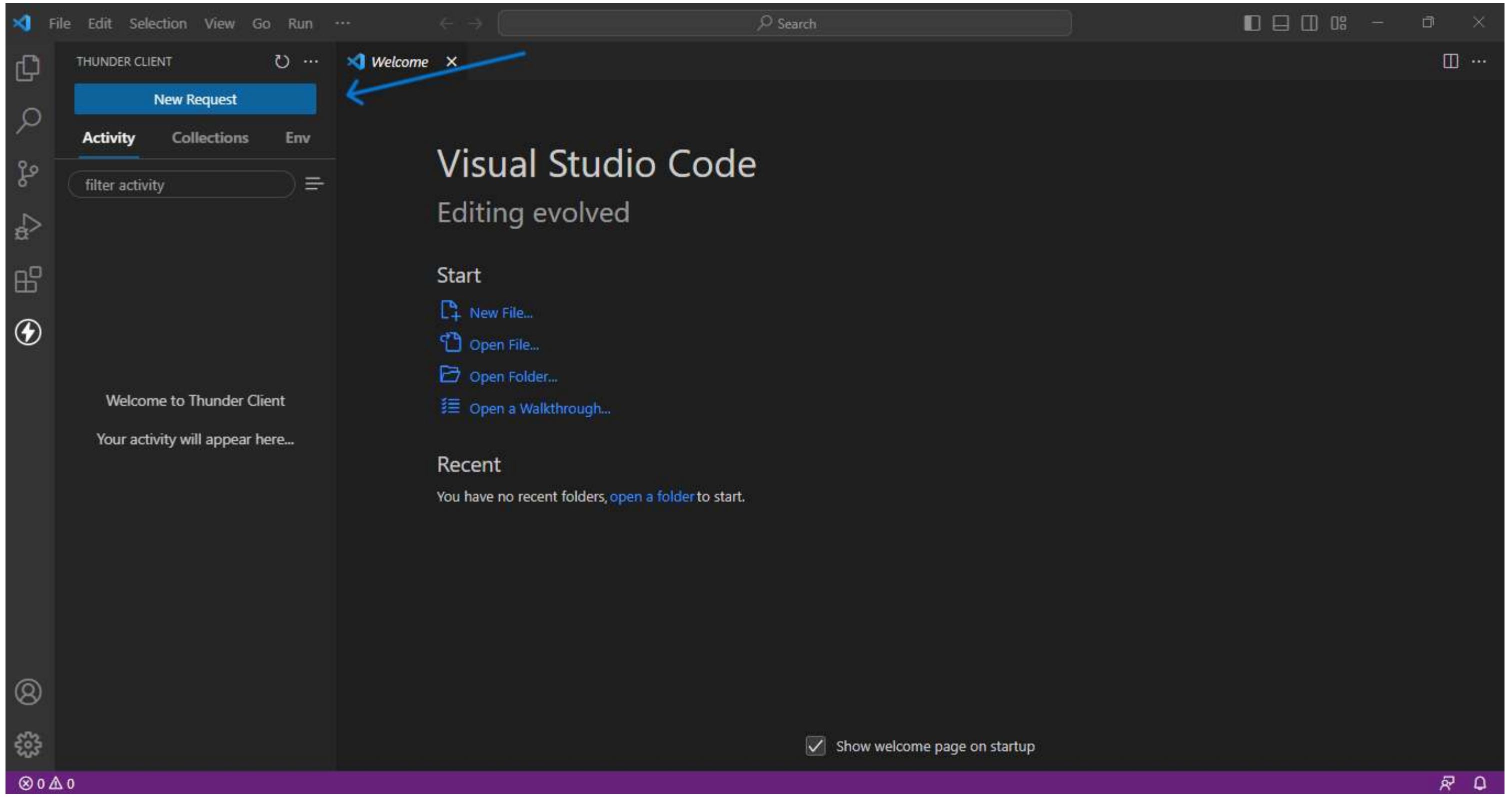
Welcome to Thunder Client

Your activity will appear here...

Example: Thunder Client API (continue)

Step 4

Make a new request: Click on the new request button.



Welcome to Thunder Client

Your activity will appear here...

Visual Studio Code

Editing evolved

Start

New File...

Open File...

Open Folder...

Open a Walkthrough...

Recent

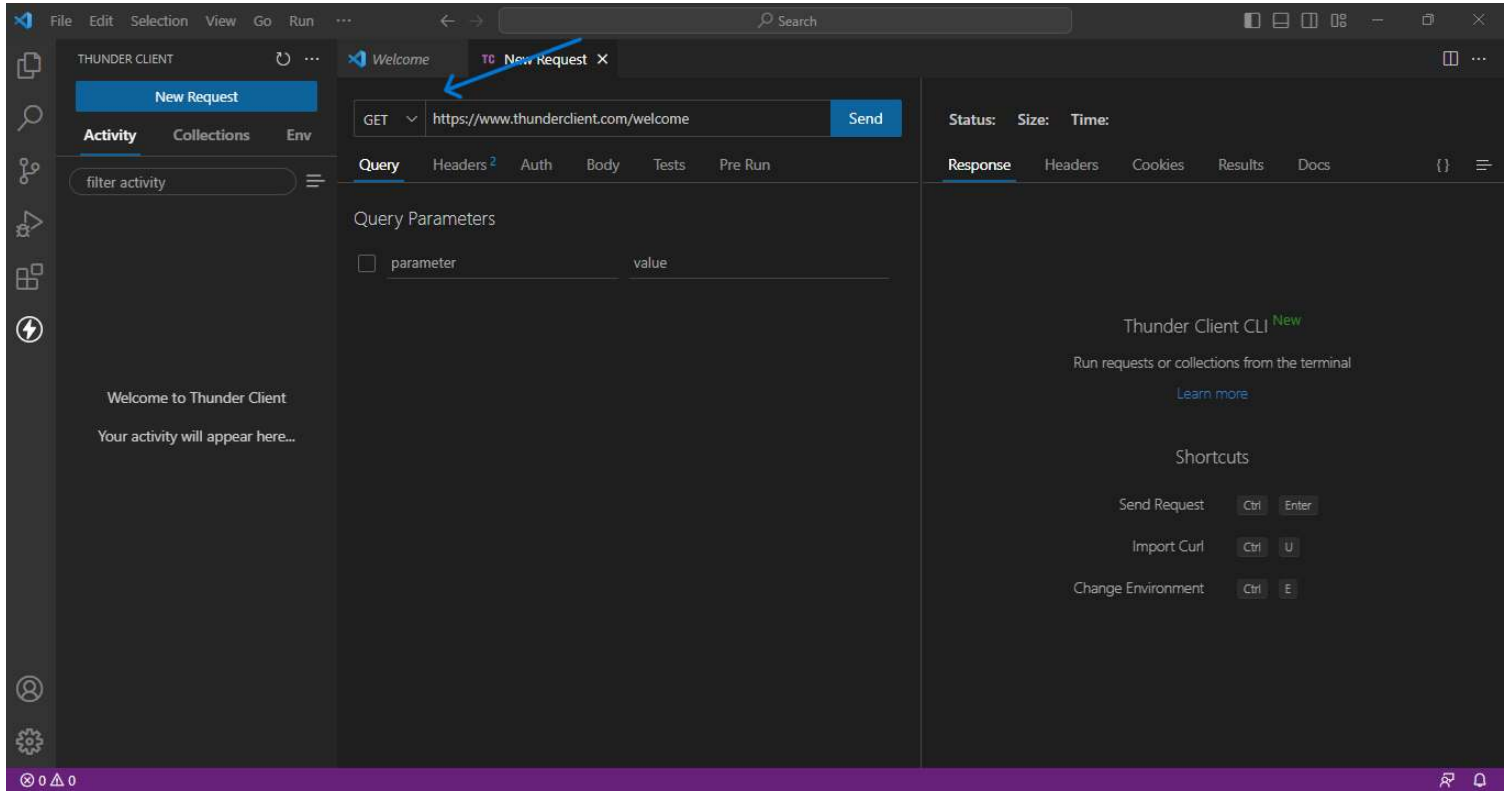
You have no recent folders, open a folder to start.

Show welcome page on startup

Example: Thunder Client API (continue)

Step 5

Select the correct verb: Depending on the type of Request, Thunder Client offers a list of HTTP VERBS for requests such as GET, POST, PUT, DELETE, and PATCH.



Welcome to Thunder Client

Your activity will appear here...

Thunder Client CLI New

Run requests or collections from the terminal

Learn more

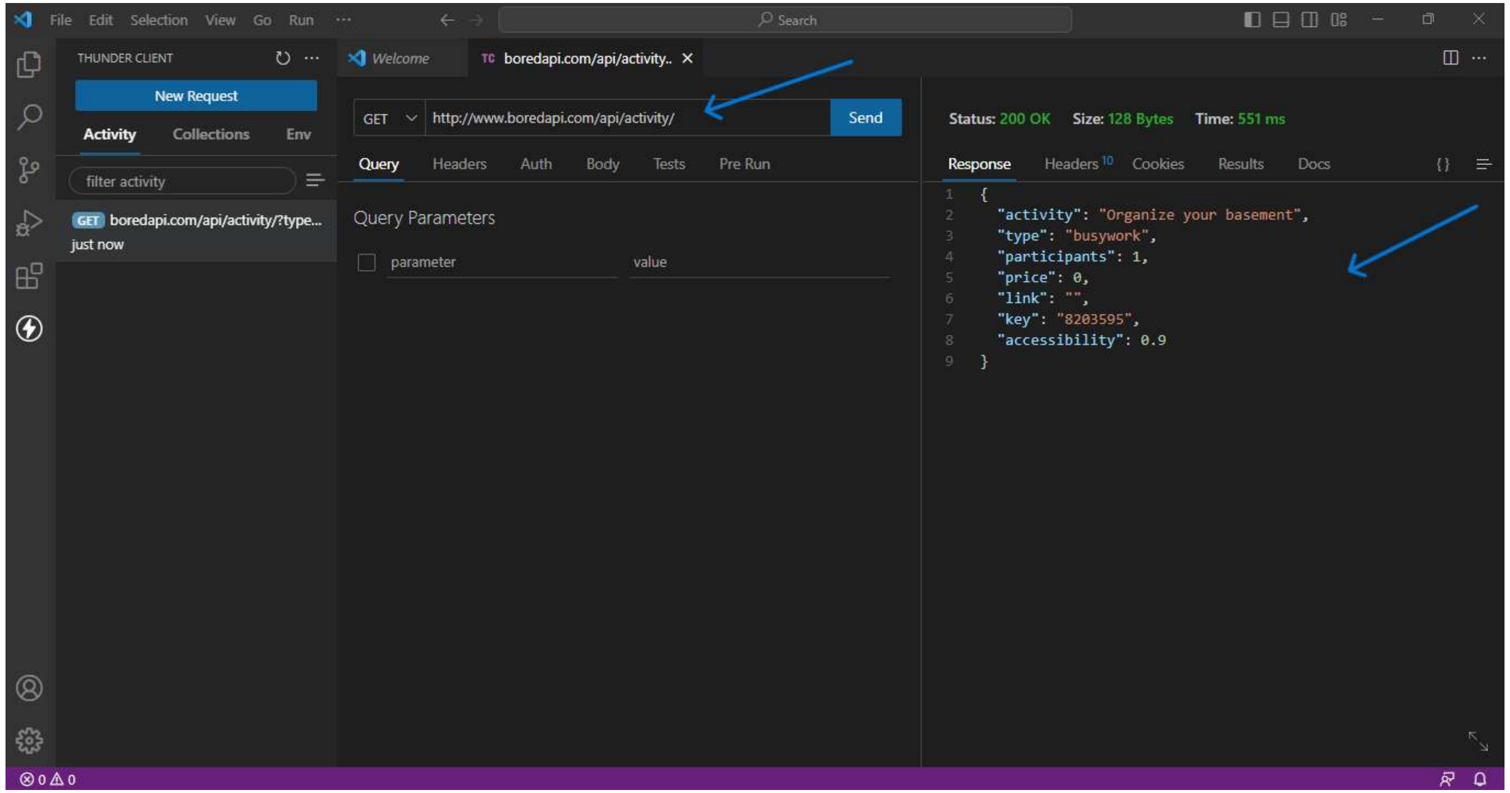
Shortcuts

Send Request	Ctrl Enter
Import Curl	Ctrl U
Change Environment	Ctrl E

Example: Thunder Client API (continue)

Step 6

Conduct a request and get a response: Using the *The Bored API* find something to do using the https://www.boredapi.com/api/activity/ call.


File Edit Selection View Go Run
THUNDER CLIENT
New Request
Activity Collections Env
filter activity
GET boredapi.com/api/activity/?type...
just now
Welcome
boredapi.com/api/activity..
GET
Send
Query Headers Auth Body Tests Pre Run
Query Parameters
parameter
value
Response Headers 10 Cookies Results Docs

```
http://www.boredapi.com/api/activity/
```

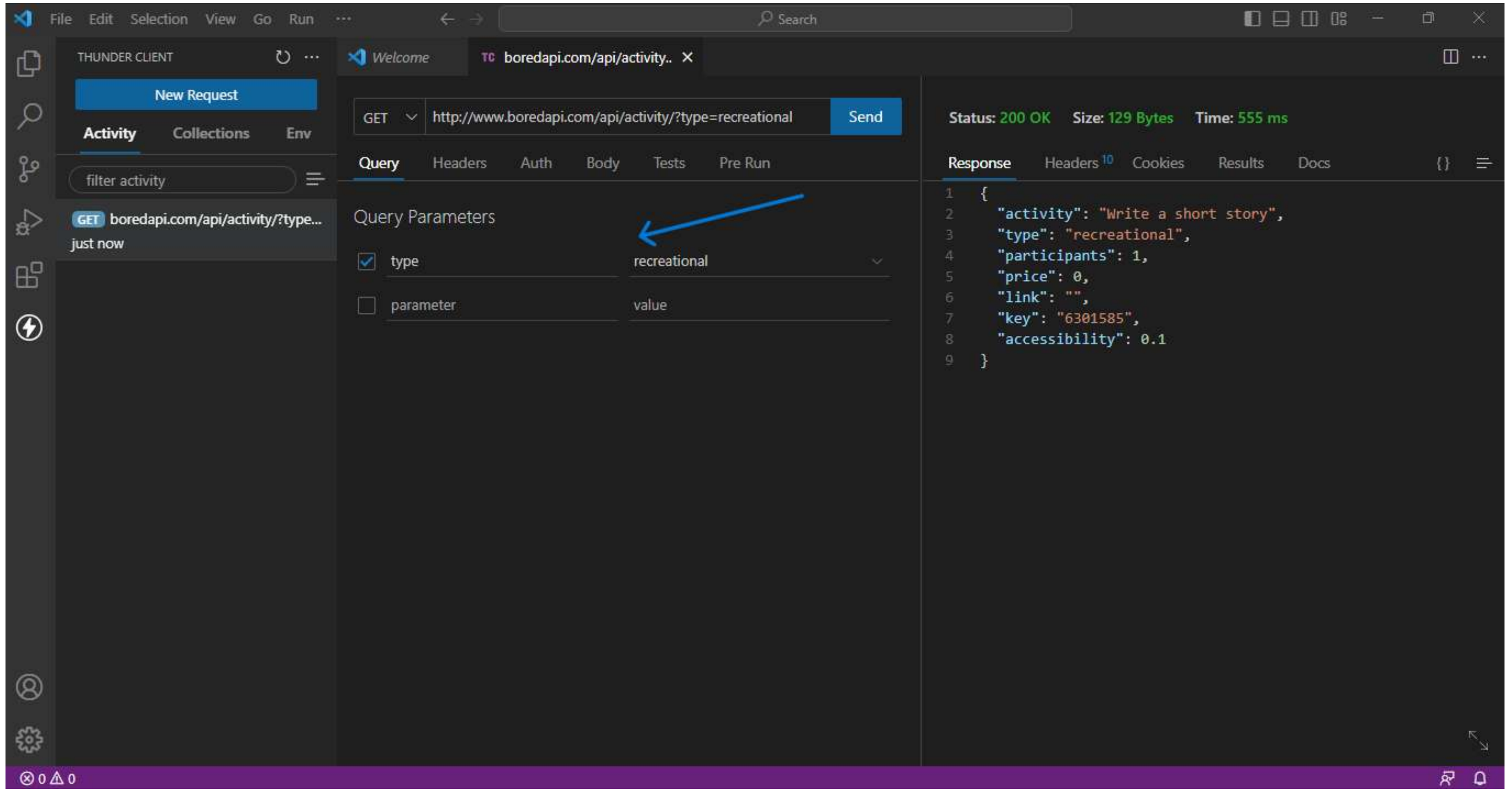
Status: 200 OK Size: 128 Bytes Time: 551 ms

```
{
  "activity": "Organize your basement",
  "type": "busywork",
  "participants": 1,
  "price": 0,
  "link": "",
  "key": "8203595",
  "accessibility": 0.9
}
```

Example: Thunder Client API (continue)

Step 7

Add parameters: Add a parameter to only show activity *type* = *recreational* in the response.


File Edit Selection View Go Run
THUNDER CLIENT
New Request
Activity Collections Env
filter activity
GET boredapi.com/api/activity/?type...
just now
Welcome
boredapi.com/api/activity..
Query Headers Auth Body Tests Pre Run
Query Parameters
parameter
value
Response Headers 10 Cookies Results Docs

```
GET http://www.boredapi.com/api/activity/?type=recreational  Send
```

✓	type	recreational

Status: 200 OK   Size: 129 Bytes   Time: 555 ms

```
{
  "activity": "Write a short story",
  "type": "recreational",
  "participants": 1,
  "price": 0,
  "link": "",
  "key": "6301585",
  "accessibility": 0.1
}
```

Make an API call youself

Question 1

Find something to do using the free *The Bored API* using *VSCode* and *Thunder Client*.

Question 2

Limit your call using parameters to only show results with *key = 1000000*.

HINTS:

Thunder Client: https://www.freecodecamp.org/news/thunder-client-for-vscode/

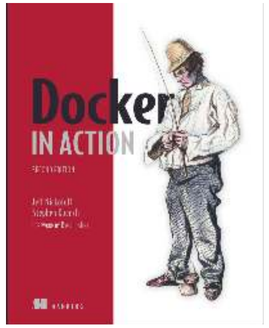
The Bored API: https://www.boredapi.com/

Introduction to Docker

What is Docker?

A great quote:

> A best practice is an optional investment in your product or system that should yield better outcomes in the future. Best practices enhance security, prevent conflicts, improve serviceability, or increase longevity. **Best practices often need advocates because justifying the immediate cost can be difficult.**



Again, in English: Docker allows you build an application **that will run anywhere** and **will not**

In a nutshell, Docker is a **container engine** that allows developers like yourselves to build production-grade applications in **isolated, stable and easily portable environments**.

interfere with any other program running on your machine (anywhere there is Docker, which is everywhere).

What is Docker?

What problems does Docker help solve?

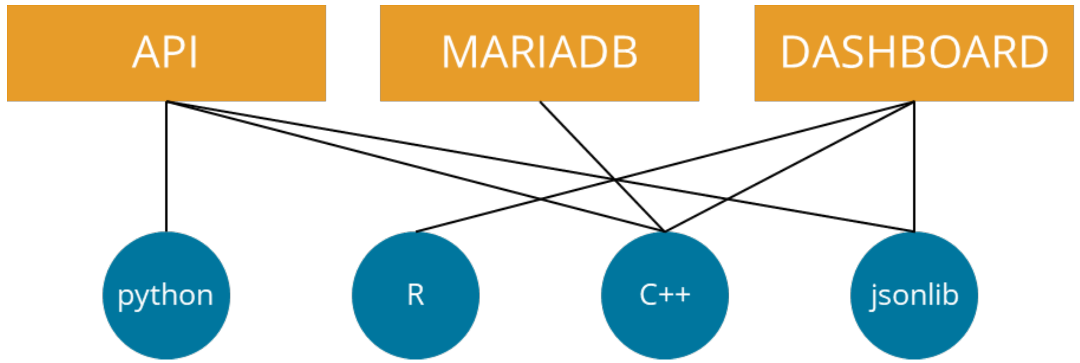
- Well, it works on my machine...
- The database has gone down. Did the server get rebooted recently?
- The API is not working. Did a new version of R get installed?

The key concept behind Docker is **containerisation**: a fancy term for being able to write code in an independent, isolated environment where you can have explicit control of the versions of packages and other underlying software supporting your project.

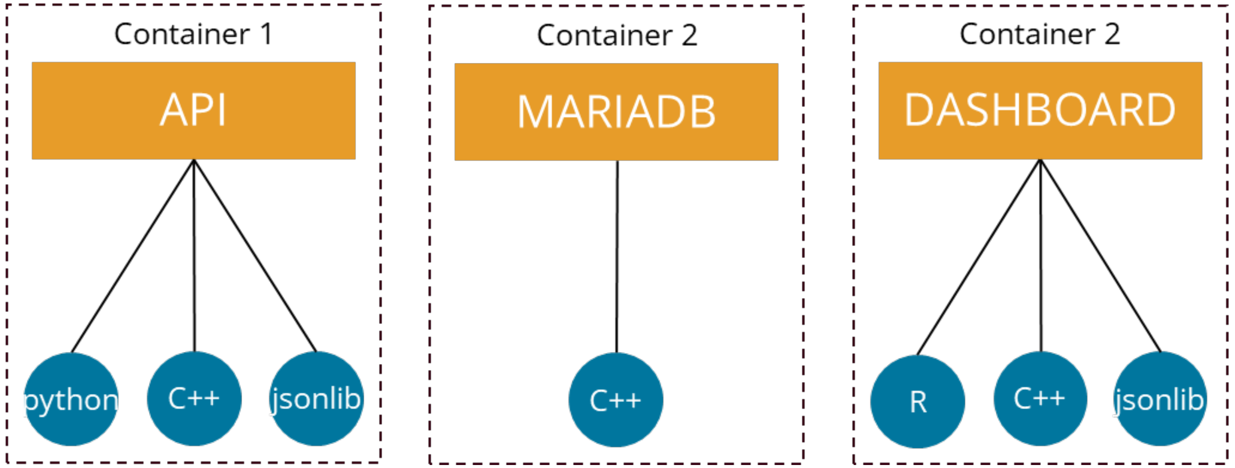
What this means in practise is that you can use different versions of python, R, java, GDAL, postman, etc depending on what your project requires. Not only can you **choose** from a variety of package versions, you can use them **without fear** that they're going to interfere with each other.

Why Docker?

This figure below illustrates the web of dependencies created by running multiple applications natively:



Docker cleans up this web by running each application inside a container:



What is Docker?

You have likely already come in to contact with containers in the form of java `jars`, python `.virtualenvs`, or `{renv}` in R. All of these solutions allow you to control your package versions to make your projects more sustainable and portable. However, in this day and age, you are likely to be using **many different languages** in your projects, some of which will depend on system packages.

Docker is generalised version of the **language-specific** tools mentioned above. What Docker aims to do is to create a container that goes around a whole operating system and everything installed on it. This means that you can specify **even the OS** that you want to use for a project, and then install the right dependencies on top of that.

Docker was inspired by the adoption of a standard shipping container by the shipping industry. Creating standard dimensions for carrying goods increased the efficiency of freight and the shipping industry took off. For more details see this keynote from the founder of Docker.

Why Docker?

Docker was originally intended as a tool to be able to easily transfer applications into other machines as the demand for their use grows. This is really in the realm of hardcore software engineering and DevOps. So you might be thinking: "Is this not overkill? How is this necessary for me as a data scientist?"

While the applications and dashboards you are building may never be required to serve as many users as say Twitter or Facebook, you will nonetheless be required to be writing **production code**. This means that whatever you build must have an emphasis on **stability** and **security**, so that your users can consistently access a high-quality service. This is how Docker can help you to do this:

- **Stability**: Running an app inside a docker container isolates that application from other activities or processes happening on your server. This mean that you can upgrade, downgrade, delete or whatever else your system packages **with no fear of affecting the running app** and be more confident that your application is offering uninterrupted service. It also has the added bonus of restarting automatically over system reboots, meaning you don't have to manually restart it.
- **Security**: Docker containers are isolated from the server they sit on. Containerising an app that is exposed to the public or even other computers in a network is a good way to insure that in an event of your app becoming compromised, the rest of your server is less at risk than if it was running *natively* (that is, directly on the server).

A quick note on Virtual Machines

Many of you will have heard of **Virtual Machines**, and you might be thinking that Docker is a Virtual Machine. Indeed, they can serve the same function as a Docker container. However there are some key differences that you should be aware of.

Technical difference - JARGON WARNING:

- **A Docker container** is a clever way of *isolating* operating system processes. Although many different containers could be running on a system, at the end all their processes are running on the same operating system and in the same kernel.
- **Virtual Machine** is just what it claims it is. It is a virtual simulation of a physical computer, which means it has its own specially partitioned and ringfenced system resources **in addition to** software.

The key difference is that **Docker containers don't use any hardware virtualization**. This makes them much more efficient that a Virtual Machine.

Installing Docker

The online Docker user manuals, called Docker Docs, is incredibly comprehensive repository for all things Docker related, and it should be your first point of reference when you run into trouble!

You can find a guide to installing Docker here. Follow this now!

Once you're done, check the results of `docker --version` to see if it works.

With confirmation that Docker is successfully installed, it's time to spin up a Docker for the first time.

```
docker run hello-world
```

What happens?

The Docker group

You should see that your `docker run hello-world` attempt will fail with this error message:

> `docker: permission denied while trying to connect to the Docker daemon socket`

While this is likely the first time you will have run into this error, it most certainly will not be the last!

Docker controls the ability to interact with images, containers and anything else Docker related via the `docker` group. If you aren't assigned to this group, whatever docker command you run you will be met with the same fate. Use `id` to check what groups have been assigned to you, and then use `sudo usermod -aG docker $USER`

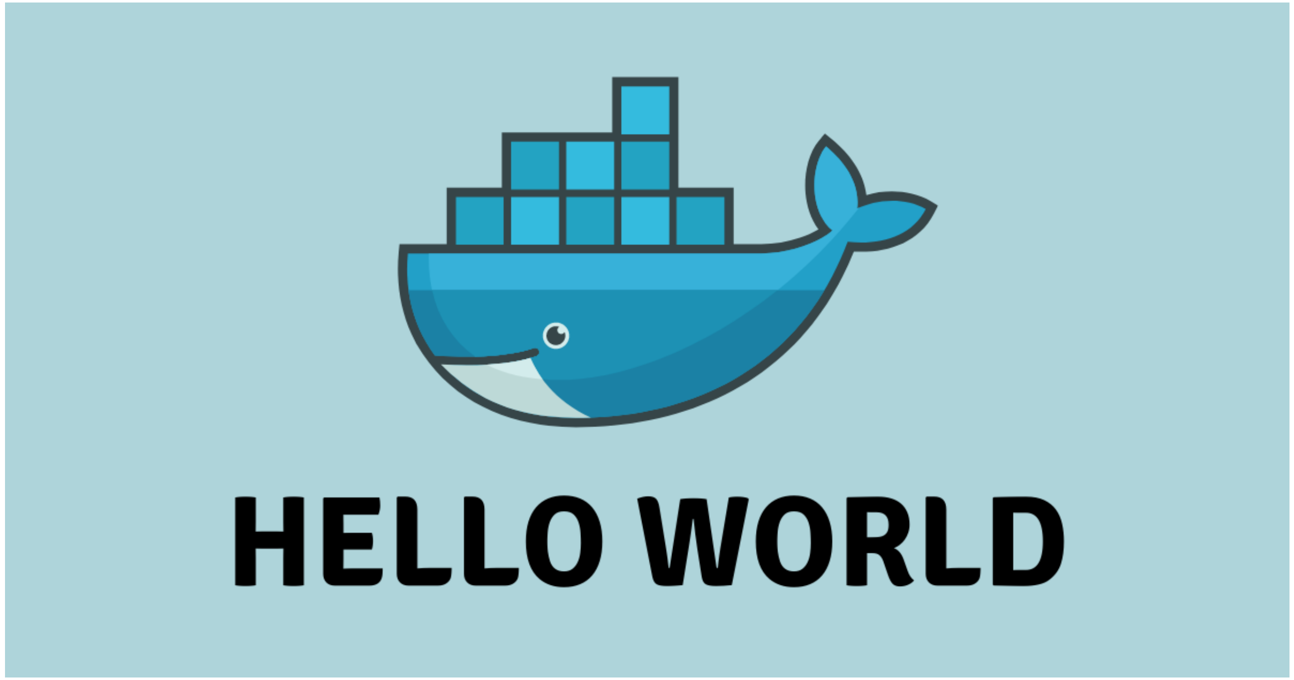
- `usermod` is the command to (surprisingly) **mod**ify a **user**'s characteristics
- `-aG` is the shorthand for **a**dd **G**roup
- `docker` is the name of the group
- `$USER` is the system variable that identifies the current user in the shell

Once you have done this, relog into your machine and check that everything has worked by repeating `id`. If you don't relog you won't see any changes, as your user information is only re-evaluated once a new shell session is started.

First Docker

Now that annoying admin is out of the way, we can actually get to running our first docker. What happens?

```
docker run hello-world
```



Understanding Docker

What has just happened here is not particularly clear, and raises more questions than answers. What makes this any different from a plain old `echo`? What is an image? What is Docker Hub? Let's try and clear this up by getting to grips with some concepts that are core to the way Docker works.

To recap: Docker allows you to create an isolated environment for an application (in this case, think API!). The easiest way to think about this is as a second computer that has completely different packages installed it, that can be upgraded or deleted entirely independently from your machine. But how is this environment defined? What packages are installed in it? What operating system is it running?

All of these questions are defined in a Docker **image**. You can think of this as a list of specifications that outline the **software setup** of a computer. Note that it does not ever mention anything about hardware allocation, since this is not a Virtual Machine!

Now then: a Docker **container** is the actual running isolated environment. Every Docker container is based on a Docker image. You can have multiple Docker containers running from the same image, but you cannot have one Docker container based on more than one image.

Fortunately, you don't have to build an image from scratch. Docker Hub is an online repository for pre-made images free for anyone to use!

Running our second Docker container

Let's try to put these concepts into practise. As referenced in the output of `hello-world`, we can run something a little more ambitious. How about an ubuntu container? The command above first downloads an ubuntu **image** from Docker Hub. You can find a description of the image here. This image is a minimal Ubuntu OS that is a great base for the start of any application. Once the image is succesfully downloaded, a Docker **container** is immediately spun up based on the Ubuntu image. Once the container is ready, Docker opens an interactive `bash` session for you(the `-it` option) within that container. You should notice that you are now on what looks like a different machine, logged in as the root user!

```
docker run -it ubuntu bash
```

Try to move around and run some basic bash commands. You should see that it functions in exactly the same way as the server that you set up yesterday! Some examples of things you can try:

- use `ll` and `cd` to move around the file system
- add a new user with `adduser`
- install a package with `apt install`

Once you're done playing around, exit the container with CTRL+D.

Some basic Docker commands

Now that we've been playing around with Docker for a while, let's have a look at some tools to use to help you work with Docker on your machine. Docker commands always start with `docker`, which is helpful.

Docker commands

- `docker run {image_name}`: this commands takes an image and spins up a container based on that image!
- `docker image ls`: shows a list of available images downloaded or built on your machine. You can remove an image using `docker image rm`.
- `docker stop {container_name}`: this spins down an active container
- `docker ps`: one of the most useful commands, it shows the list of currently running Docker containers on your system. Adding the `-a` argument will also show all the containers that have been spun up since the last system prune.
- `docker system prune`: this command cleans up your Docker. Make sure you adhere to it's warning messages!

Docker syntax is very friendly! You should notice that these commands are structured to align with the definitions of bash commands with the same name. For example, as `ls` lists the contents of a directory, `docker image ls` lists all our saved images. This makes working with Docker pretty easy to learn.

Extra credit question: why is the Ubuntu container that we just spun up show a `STATUS` of `Exited`?

Building our own docker image

There is a vast array of Docker images available from Docker Hub for you to use. However, a prebuilt docker image will **never** meet your needs forever. So we need to learn how to customise these images to be fit for whatever our purpose happens to be.

A Docker image is created using a `Dockerfile`. In essence, this is a docker base layer combined with a small set of basic commands that outlines a custom Docker image. Let's have a look at the basic building blocks of a `Dockerfile`.

- `FROM`: this specifies the base image that the rest of the customisation will be built on. This can be as simple as `ubuntu`.
- `COPY`: add additional pre written files, scripts or data to the image as it is **built**.
- `RUN`: add supporting lines of code that will be run **during the image build only**
- `CMD`: commands/operations that should be run **each time** a container using this image starts up

Let's try out an example of how this might work. First, let's keep things organised: make a new folder in your home directory called `docker`, and within that another dir called `01-basic`.

Building our own Docker image

Outlining an image

Within your `~/docker/01-basic` directory, use `vim` to create a new file called `Dockerfile`, and add these lines:

```
FROM ubuntu:latest
CMD echo 'Docker is nuts!'
```

That's it. What are these lines doing?

- `FROM ubuntu:latest`: this tells the Docker engine that this image will be based on the latest ubuntu image layer that is available on Docker Hub.
- `CMD echo 'Docker is nuts!'`: each time this image is called into action, the command `echo 'Docker is nuts!'` should be run.

Build the image, making sure that you are in the `~/docker/01-basic` directory: Note that the `.` is refers to the current working directory!

```
docker build .
```

Building our own Docker image

Outlining an image

If all goes well, you should see some output from docker which is downloading image layers and compiling it all together. You can verify that your image has been built using `docker image ls`. You should see an unnamed and untagged image! You can give your image a name using the `-t` (tag) argument in the `docker build` function:

```
docker build -t 01-basic .
```

You can see now that our image has a name and a tag (and is now much easier to manage!)

What's left to do but use our image to spin up a container? Before you do, think about what you expect to happen.

P.S. - don't overthink it!

```
docker run 01-basic
```

Building our own Docker image

Adding a `RUN` statement

How about some customisation to try out some of the other basic components of a Dockerfile. Edit your Dockerfile so that it looks like this:

```
FROM ubuntu:latest
RUN adduser james
CMD id james
```

What is this image specifying? Remember that there is an important difference between `RUN` and `CMD`, and that commands in `Dockerfile` are evaluated **one after the other**.

Rebuild the image and run it using the commands from earlier. What happens? Is it what you expected?

```
docker build -t 01-basic .
docker run 01-basic
```

Building our own Docker image

Adding a `COPY` statement

Now let's add in some external data that we create. Make a new file with some information in it (whatever you want to say - just feel it). Call it copy_test.

Now add a line to your Dockerfile so that it looks like this:

```
FROM ubuntu:latest
COPY copy_test .
RUN adduser james
CMD id james && cat copy_test
```

Again, rebuild your image and then run it. You should see that exactly what you entered into `copy_test` displayed. This is how you can get external information into your image!

Building our own Docker image

Adding a `COPY` statement

Usually, you'd replace `copy_test` with a script of some kind. This would be a dashboard or an API that you have already developed! Modify `copy_test` so that it turns into an executable file that echoes a string - change it up a little bit so that you can tell the difference in your next output.

```
#! /bin/bash
echo 'James is my [least] favourite Uncle'
```

Test that your file is working properly by executing it in the terminal first. Add execute permissions first:

```
chmod 700 copy_test
./copy_test
```

You should see your expect string echoed in the terminal! If not, check your file permissions to make sure that you can actually execute `copy_test`.

Building our own Docker image

Adding a `COPY` statement

Now that you are confident that your script will execute, return to editing `Dockerfile` and update your `CMD` command to **execute** `copy_test` rather than just echoing the contents:

```
FROM ubuntu:latest
COPY copy_test .
RUN adduser james
CMD id james && ./copy_test
```

For one last time:

```
docker build -t 01-basic .
docker run 01-basic
```

Even though the output is only slightly different to what we've seen before, the underlying mechanics are vastly different and have **HUGE** implications for how we can use Docker! As long as you have the right dependencies installed, you can run any script within a container that you can on your machine.

Practical Docker

Now that we are familiar with how to work with Docker, it's time to head for the deep end and put it into practice. We're going to do this by setting up a MariaDB database that runs in a container. Let's recap what the benefits of running a db inside a container rather than natively are:

- **Security**: since a container is separate from the rest of your server, there is an extra layer of security between users of the database and the rest of the server
- **Stability**: you never have to worry about the database breaking because of inadvertent changes to dependencies. The container will also spring right back up on a server reboot so it doesn't need to be manually restarted either!
- **Portability and reproducibility**: if you need to transfer your db onto another server, you can do so with minimal worries about setup. Just install Docker on the new machine, and the container will run as normal.

The MariaDB Image

There is a lot to be thankful for in life. One of them is the extent to which friendly programmers and developers make easy-to-use and high-quality software available to us **for free**.

Case in point: MariaDB. High-quality documentation is available to assist us in installing and running MariaDB in a container. The Docker Hub documentation here and the MariaDB docs here give us information on a prebuilt MariaDB image!

Looking through the Docker Hub documentation, the most minimal configuration to run is the following:

```
docker run --detach --name mariadb --env MARIADB_ROOT_PASSWORD=password --volume ./db_data:/var/lib/mysql maria
```

`docker run` should be familiar to you by now, but there are some additional parameters and arguments here that we haven't seen before. In order to understand what this is doing, we need to cover some additional functionality that Docker provides.

Advanced Docker parameters

`--detach`

The `--detach` parameter takes no argument. It tells Docker to spin up the container in **detached mode**, which essentially means that the Docker will run in the background. Try it out with our first image `01-basic`.

```
docker run --detach 01-basic
```

The only thing you should see is a random string of digits and letters, which is actually the container ID that is assigned to that container. We don't see any of the output that we saw previously, because the container has been run in the background without you having to see any of the output or messages created. This is what you would use in production.

Advanced Docker parameters

`--name`

This is a fairly simple one. The `--name` parameter takes one argument, which is the name that you want for your container! This overwrites the random name that docker will assign to a container when you spin it up. Run the following:

```
docker run --name my_first_docker 01-basic
```

Now, run `docker ps -a` and compare the names of the containers that you have spun up to check that this works. The `--name` parameter will become important in keeping track of your containers as your use of Docker gets more complex!

Advanced Docker parameters

`--env`

This is a very useful Docker function that allows you to pass custom variables into the `CMD` command of the Dockerfile. Let's have a look at how this works. We are going to do this by once again editing our good old `01-basic` image to include an environment variable called `HELLO_NAME`, which allows us to customise the name that the container spits out.

```
FROM ubuntu:latest
COPY copy_test .
RUN adduser james
ENV HELLO_NAME=james
CMD id james && ./copy_test && echo "Hello, $HELLO_NAME"
```

Rebuild the image, run the docker as usual, then add the the additional `--env` parameter:

```
docker build -t 01-basic .
docker run 01-basic
docker run --env HELLO_NAME=hanjo 01-basic
```

Advanced Docker parameters

`--volume`

One of the most powerful functions of Docker is the ability to share files between a container and the native file system. This is called **mounting a volume**. The `--volume` parameter specifies a common directory between the container and the server and mirrors the contents. This serves two functions:

- **Persistency**: as long as the mount is consistent, a container will be able to pull from a stable source of data regardless of how many times it has started and stopped
- **Extraction**: mounting a volume to the container allows you to pull out anything generated by the container's processes

Let's have a look at how this works.

Advanced Docker parameters

`--volume`

First, let's edit our familiar friend `01-basic` to output something.

```
FROM ubuntu:latest
COPY copy_test .
RUN adduser james && mkdir output
ENV HELLO_NAME=james
CMD echo "Hello, $HELLO_NAME" && ./copy_test > /output/output_file
```

Rebuild the image:

```
docker build -t 01-basic .
docker run 01-basic
```

Now, let's spin up the container remembering to include the `--volume` parameter

```
mkdir container_output
docker run --volume ./container_output:/output 01-basic
```

Return to MariaDB

Now we should be well-placed to fully understand the `docker run` command that is used to spin up our MariaDB container.

```
docker run --detach --name mariadb --env MARIADB_ROOT_PASSWORD=password --volume ./db_data:/var/lib/mysql maria
```

What's left to do but run it?

Using MariaDB

We now have a running MariaDB instance - but how do we start working with it? The `docker exec` command will help us.

```
docker exec -it mariadb bash
```

This opens an **i**nteractive **t**erminal for us to use to administrate our db.

```
mariadb -u root -p
```

We are now in the MariaDB monitor and can go ahead with the usual database operations that we are familiar with!

```
CREATE DATABASE warehouse;
SHOW TABLES;
CREATE TABLE test(number FLOAT);
INSERT INTO test SELECT RAND();
```

A sneak peak into docker compose

Our `docker run` command is not looking pretty hefty. You certainly won't be remembering that without a lot of effort. In general, constructing these commands becomes exponentially more complicated especially when you want to spin up multiple containers that are all talking to each other!

Fortunately, there is a solution! It is called `docker compose`, and it allows you to replace a long, cumbersome `docker run` command with a concise `yaml` file that outlines all the different parameters that are required to spin up our container. To demonstrate this, create a new directory in `~/docker/` called `02-maridb` and create a file called `docker-compose.yaml` in it with the following contents:

```yaml
services:
  db:
    image: mariadb
    container_name: mariadb
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: password
    ports:
      - "3601:3601"
    volumes:
      - ./db_data:/var/lib/mysql
```

A sneak peak into docker compose

Remember to also create a directory called `db_data` to use as a mapped volume!

We can now easily spin up a container using `docker compose up --detach`, which is much more friendly. Hanjo will showcase some more advanced uses of `docker compose` in the next sessions.

FastAPI and Docker

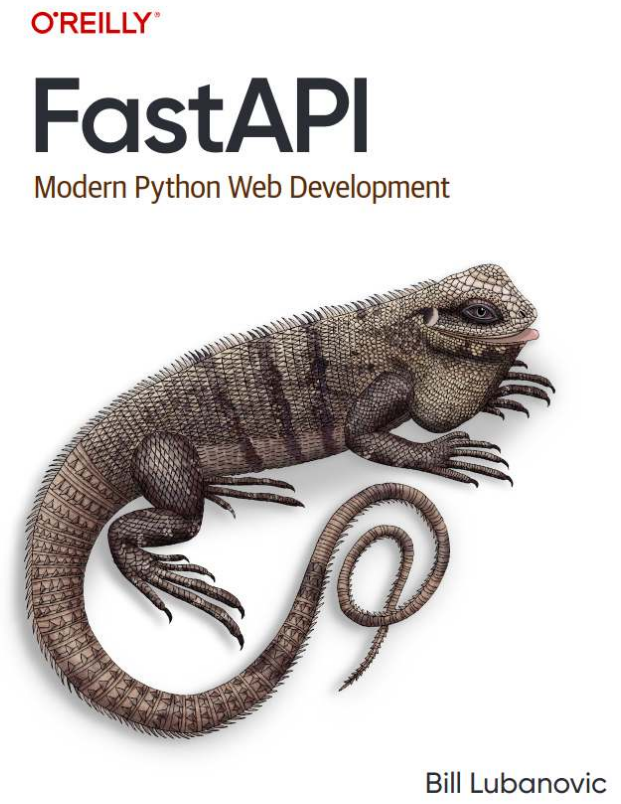
```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Learning outcomes

> Now and then, the bright and shiny objects that we stumble across can turn out to be very useful.

We want to take you on a high-level tour of FastAPI. Getting you started with the basics! The rest of the journey is up to you:

- Hello World
- Hello World in Docker
- Making DB calls while in Docker



Building basic API ⚙

SETUP

Another important term to know is operation, which is used in reference to any of the HTTP request methods:

- POST
- GET
- PUT
- DELETE
- OPTIONS
- HEAD
- PATCH
- TRACE

Create an environment

```
mkdir ~/.venv
python3 -m venv myapi
source ~/venv/myapi/bin/activate
```

Create a coherent folder for API

Its always good practice to organise your folders, distinguishing between production and development.

- My Suggestion would be to have the following folder structure:

```
cd ~
mkdir -p {docker/fastapi,production,projects,packages/{r,python}}
```

Create a coherent folder for API

Its always good practice to organise your folders, distinguishing between production and development.

- My Suggestion would be to have the following folder structure:

```
(base) hanjo@optimus:~$ tree
.
├── docker
│   └── fastapi
├── packages
│   ├── python
│   └── r
├── production
└── projects

cd packages/python
```

Ready for the basics!

Requirements

On to creating your first using API `fastapi`. Create a file in VScode under the `~/docker/fastapi/` folder. Create the requirements.txt file: `requirements.txt`.

`requirements.txt`

```
fastapi
uvicorn[standard]
pydantic
typing
```

Install Linux dependencies:

```
sudo apt update
sudo apt install python3-pip
sudo apt install uvicorn
sudo apt install net-tools
sudo apt  install httpie
```

In the same folder execute the requirements file, making sure you are in the right environment!

```
pip3 install -r requirements.txt
```

main.py

To start your first using API `fastapi`, create a file in VScode under the `~/docker/fastapi/app` folder. Call this file `main.py`

A basic FastAPI file looks like this:

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/hi")
def greet():
    return {"message": "Hello World"}
```

- `app` is the top-level FastAPI object that represents the whole web application.
- `@app.get("/hi")` is a *path decorator*. It tells FastAPI the following:
  - A request for the URL "/hi" on this server should be directed to the following function.
  - This decorator applies only to the HTTP `GET` verb. You can also respond to a `/hi` URL sent with the other HTTP verbs (PUT, POST, etc.), each with a separate function.
- `def greet()` is a path function

main.py

To start your first using API `fastapi`, create a file in VScode under the `~/docker/fastapi/app` folder. Call this file `main.py`

A basic FastAPI file looks like this:

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/hi")
def greet():
    return {"message": "Hello World"}
```

Run the First API App With Uvicorn 🦄:

```
uvicorn main:app --port 8000 --host 0.0.0.0 --reload
```

main.py

What can we put in `FastAPI()`? https://fastapi.tiangolo.com/reference/fastapi/

```
description = """
Football API helps you get football data. ⚽

## Users

You will be able to:

  **Authenticate**
  **Get World Cup Football Data**
"""

tags_metadata = [
    {
        "name": "basics",
        "description": "Basics operations.",
    },
    {
        "name": "data",
        "description": "Data is legit.",
        "externalDocs": {
            "description": "Items external docs",
            "url": "https://fastapi.tiangolo.com/",
        },
    },
]
```

```
app = FastAPI(
    title = "WorldCupApi",
    description = description,
    summary = "Messi's favorite app. Nuff said.",
    version = "0.0.1",
    terms_of_service = "http://example.com/terms/",
    contact = {
        "name": "Ronaldinio",
        "url": "http://football.com",
        "email": "football@worldcup.com",
    },
    license_info={
        "name": "Apache 2.0",
        "url": "https://www.apache.org/licenses/LICENSE-2.0.html",
    },
    openapi_tags=tags_metadata
)

@app.get("/hi", tags=["basics"])
def greet():
    return {"message": "Hello World"}
```

Interacting with the endpoint

- `netstat -nltp`
- Endpoint
  - `http://{IP}:8000`
- Docs
  - `http://{IP}:8000/docs`
  - `http://{IP}:8000/redoc`

Its also good to understand terminology around requests, especially curl commands! In our case using `http` from `httpie`:

```
http localhost:8000/hi
http -b localhost:8000/hi
http -v localhost:8000/hi

# HTTP/1.1 200 OK
# content-length: 25
# content-type: application/json
# date: Sun, 10 Dec 2023 17:54:08 GMT
# server: uvicorn
#
# {
#     "message": "Hello World"
# }
```

Interacting with the endpoint

Its also good to understand terminology around requests, especially curl commands! In our case using `http` from `httpie`:

```
http -v localhost:8000/hi

# GET /hi HTTP/1.1
# Accept: */*
# Accept-Encoding: gzip, deflate
# Connection: keep-alive
# Host: localhost:8000
# User-Agent: HTTPie/2.6.0
#
#
#
# HTTP/1.1 200 OK
# content-length: 25
# content-type: application/json
# date: Sun, 10 Dec 2023 18:04:55 GMT
# server: uvicorn
#
# {
#     "message": "Hello World"
# }
```

This request contains the following:

· The verb (`GET`) and path (`/hi`) · Any query parameters (text after any ? in this case, none) · Other HTTP headers · No request body content

Exercise

1) Write an app that returns your name using path (or endpoint) `name`:

- The returned object has to say "Hello, my name is {name}"

2) Pick an IP from this list and tell me whose machine is whose. Do this using commandline and ThunderClient

20:00

How to add parameters

All the arguments that you need can be declared and provided directly inside the `path function`, using the definitions in the preceding list(Path, Query, etc.), and by functions that you write. This uses a technique called dependency injection and is one of the major advantages of FastAPI above other frameworks

Lets add parameters in four different ways:

- In the URL path
- As a query parameter in the URL using `?`
- In the HTTP `body`
- As an HTTP header
    - My preferred option for authentication

How to add parameters: URL Path

Once you save this change from your editor, Uvicorn should restart! Remember we set it up to assist with autorelad using `--reload`.

Adding that `{who}` in the URL (after `@app.get`) tells FastAPI to expect a variable named who at that position in the URL. FastAPI then assigns it to the who argument in the following `greet()` function.

> This shows coordination between the path decorator and the path function.

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/hi/{who}")
def greet(who):
    msg = f"Hello {who}!"
    return {"message": msg}
```

```
ubuntu@ip-172-31-11-91:~/docker/fastapi$ http localhost

# HTTP/1.1 200 OK
# content-length: 24
# content-type: application/json
# date: Sun, 10 Dec 2023 19:11:24 GMT
# server: uvicorn
#
# {
#     "message": "Hello Mom!"
# }
```

How to add parameters: Query Parameters

Query parameters are the `name=value` strings after the `?` in a URL, separated by `&` characters. This is a common way to specify and build basic APIs.

The endpoint function is defined as `greet(who)` again, but `{who}` isn't in the URL on the previous decorator line this time, so FastAPI now assumes that who is a query parameter!

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/hi")
def greet(who):
    msg = f"Hello {who}!"
    return {"message": msg}
```

Note the double `==` in the second example:

```
http -v localhost:8000/hi?who=Mom
http -v localhost:8000/hi who==Mom
```

How to add parameters: Body

Note that `GET` is suppose to be `idempotent` - *ask the same question, get the same answer*. If we want to "send" data to an endpoint, we use `POST`!

```python
from fastapi import FastAPI, Body
import uvicorn

app = FastAPI()

@app.post("/hi")
def greet(who:str = Body(embed=True)):
    msg = f"Hello {who}!"
    return {"message": msg}
```

Note the single `=`

```
http -v localhost:8000/hi who=Mom

# POST /hi HTTP/1.1
# Accept: application/json, */*;q=0.5
# Accept-Encoding: gzip, deflate
# Connection: keep-alive
# Content-Length: 14
# Content-Type: application/json
# Host: localhost:8000
# User-Agent: HTTPie/2.6.0
#
# {
#     "who": "Mom"
# }
#
#
# HTTP/1.1 200 OK
# content-length: 24
```

How to add parameters: HTTP Header

Probably my favourite way to include parameters (mostly because I work a lot with authentication).

```python
from fastapi import FastAPI, Header
import uvicorn

app = FastAPI()

@app.get("/hi")
def greet(who:str = Header()):
    msg = f"Hello {who}!"
    return {"message": msg}
```

HTTPie uses `name:value` to specify an HTTP header.

```
http -v localhost:8000/hi who:Mom
```

How to add parameters: HTTP Header

FastAPI converts HTTP header keys to lowercase, and converts a hyphen (-) to an underscore (_). So you could print the value of the HTTP User-Agent header like this:

```python
from fastapi import FastAPI, Header
import uvicorn

app = FastAPI()

@app.get("/agent")
def get_agent(user_agent:str = Header()):
    return user_agent
```

- Get agent:

```
http -v localhost:8000/agent
```

Exercise

1) Write an app that returns your name and surname using endpoint `fullname`

- Use Headers and Query Parameters
- The returned object has to say "Hello {name} {suname}"

2) Write a function that returns the host address

> Your amazing app should have three endpoints!

30:00

Dockerize

Now that you have an amazing app, lets put it into production! In the folder `~/docker/fastapi`, create a `Dockefile`:

```
# BUILD: docker build -t myapi .
# DEV: docker run --rm --name myapi -p 8000:80 myapi

FROM python:3.9

WORKDIR /code

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./app /code/app

EXPOSE 80

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

check: `http://{IP}:8000/`.

Homework: Football API

> Disclaimer, we do not cover SSL certifications - without it, any BASIC auth like this is kind of useless

1) Using the `request` library in `python` make a request to get data from a csv.

- Authenticate using headers with the name and password you inserted
  - 💡 Combine headers and if statement in order to check for correct auth
  - Use {name} {suname} (use your own)
- Return all data from `~/data/worldcup.csv`
  - For optimised performance, read in at the top of the app

2) Adjust your dockerfile

45:00

Back to interactions with DB!

Lets create a user database

It is likely that we certain users to have certain levels of access. Especially if we interacting with APIs that are open to the public.

The basic first step is to create a DB in MySQL:

```
CREATE TABLE api.keys AS(
  user VARCHAR(100),
  apikey VARCHAR(32),
  api_loaded TIMESTAMP,
  PRIMARY KEY(apikey)
)
```

Upload some data.

- Load the following dataset

```
LOAD DATA LOCAL INFILE '/home/ubuntu/data/worldcup.csv'
INTO TABLE data.worldcup
FIELDS TERMINATED BY ','
IGNORE 1 LINES
;
```

- What is wrong with this if our `MariaDB` is in a container?
- Create the correct `CREATE TABLE` and then upload. Use `head` to get the correct column names.

20:00

Using dbutils example

Start off by creating two files in the `app/` folder called `dev.py` and `.env`. Most important, in the `.env` file, add your credentials that were used when setting up the MariaDB docker:

```
db_user = X
db_pass = X
db_host = X
db_port = X
```

Next install the `dbutils.whl` package using:

```
pip install dbutils.whl
```

Using dbutils example

The `dev.py` file will be broken down into three distinct sections: (1) Imports, (2) Logger setup and (3) Main:

```python
import logging
from decouple import config
from dbutils import Query
import pandas as pd

def setup_logger():
    # create logger
    logger = logging.getLogger('dbutils')
    # logger.setLevel(logging.DEBUG)
    logger.setLevel(logging.INFO)

    # create console handler and set level to debug
    ch = logging.StreamHandler()
    ch.setLevel(logging.DEBUG)

    # create formatter
    formatter = logging.Formatter('%(asctime)s [%(levelname)s] %(name)s: %(message)s')

    # add formatter to ch
    ch.setFormatter(formatter)

    # add ch to logger
    logger.addHandler(ch)
```

Using dbutils example

The final piece of the script contains our `main` function:

```
def main():
    setup_logger()

    api_db = Query(
        db_type = 'mysql',
        db_name = 'api',
        db_user = config('db_user'),
        db_pass = config('db_pass'),
        db_host = config('db_host'),
        db_port = config('db_port')
    )

    print(api_db.sql_query(sql = "SELECT * FROM keys", limits = 5))

if __name__ == '__main__' and __package__ is None:
    print(f"Running main file {__name__}")
    main()
```

Add keys for auth into DB

```sql
INSERT INTO api.keys (user, apikey)
VALUES ('hanjo', 'secret')
```

Move to Production in docker

What do we change when we move this configuration to docker?

- Ensure we have the correct `.env` folder
    - Integrated DNS allows us to use the name!
- Adjust Dockerfile in order to install `dbutils.whl`
- Adjust Dockerfile in order to install

Final Docker-compose

What do we change when we move this configuration to docker?

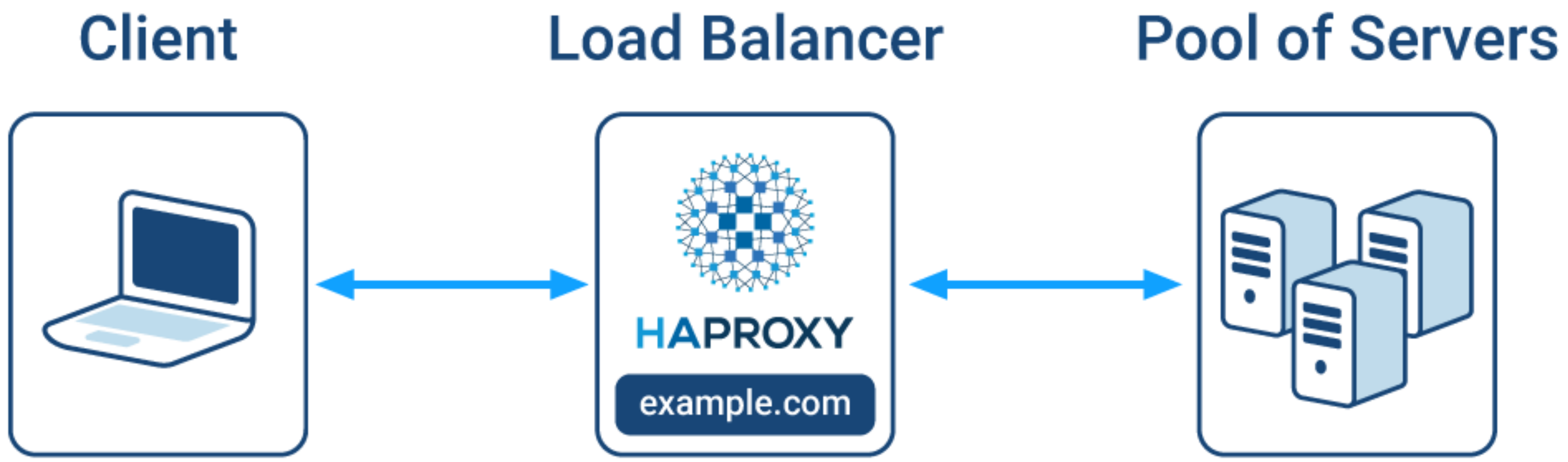
```
version: "3.4"

# RUN: docker-compose -p myapi up -d
# TEST: siege -t1s 'http://127.0.0.1:8000/'

services:
  api:
   restart: always
   image: myapi
   ports:
    - "8000:80"
   networks:
     - mariadb

networks:
  mariadb:
    external: true
    name: mariadb
```

Everyone's favourite part: documentation

https://fastapi.tiangolo.com/tutorial/metadata/

How do we scale?

How do we scale?



Scale your API using Gunicorn

You can use Gunicorn to manage Uvicorn and run multiple of these concurrent processes. That way, you get the best of concurrency and parallelism.

```
pip3 install "uvicorn[standard]" gunicorn
sudo apt install siege
```

- RUN!

```
gunicorn \
main:app --workers 4 --worker-class \
uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000
```

- Siege!

```
siege -C
siege -t1s 'http://127.0.0.1:8000/'
```

Dockerize

Adjust the Dockerfile to use `Gunicorn`:

```
# BUILD: docker build -t myapi .
# DEV: docker run --rm --name myapi -p 8000:80 myapi

FROM python:3.9

WORKDIR /code

RUN apt update

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

RUN pip3 install "uvicorn[standard]" gunicorn

COPY ./app /code/app

EXPOSE 80

CMD [ \
    "gunicorn", "app.main:app", "--workers", "4", \
    "--worker-class","uvicorn.workers.UvicornWorker", "--bind", "0.0.0.0:80" \
    ]
```