# Sustainable Software Engineering Workshop

Saif Latifi

24th November 2025

DRIVE-Health, King’s College London

# Motivation

- Writing software goes beyond *just coding*.
- We want to write code that can be used *beyond the lifetime* of the project.
- Sustainable software is easier to **maintain, extend, and share**.

So how do we achieve this?

# Agenda

Today we will broadly cover a range of techniques, including:

- **Version control** with Git and GitHub
- **Testing** and CI/CD
- Building **APIs** with Flask
- **Building and packaging** with Docker

# Version Control

# What is Version Control?

- It is used to *track changes* to files over time.
- Helpful if you want to *revert* to a previous version.
- Enables *collaboration* among multiple developers by providing tools to combine changes.

# Git and GitHub

What is Git?

- A specific flavour of version control.
- Comes installed with most operating systems (expect Windows of course).
- Command line tool with a rich ecosystem.

What is GitHub?

- A web-based platform for hosting Git repositories.
- Facilitates collaboration by easily sharing code with others.
- Provides additional features like issue tracking, pull requests, and project management tools.

# Basic Git Concepts: Repository

A **repository** (or *repo*) is a directory that contains your project files and the entire history of changes made to those files.

It also contains a hidden subdirectory called `.git` which stores all the *metadata* for your project.

You can create a new repository using the command:
`git init`.
Or, clone an existing one using `git clone <url>`.

# Basic Git Concepts: Commit

A **commit** is a snapshot of your repository at a specific point in time. It allows you to see the changes made since the last commit, known as the *diff*.

A commit contains a **message** that describes the changes made, along with metadata such as a unique ID, author, and timestamp.

There are a few steps to creating a commit:

1. "Stage" (i.e. select) the changes using `git add <path>`.
2. Create the commit using `git commit -m "Your commit message"`.

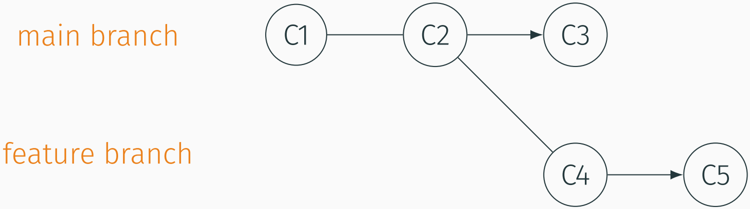
# Basic Git Concepts: Branch

A **branch** is a separate version of your files within the same repository.

It allows you to make changes *without affecting* the main codebase (usually called `main` or `master`).

You can switch branches using `git switch <branch-name>` and create new branches using the same command with the `-c` flag.

# Branch Visualisation



main branch

feature branch

# Basic Git Concepts: Branch

A **branch** is a separate version of your files within the same repository.

It allows you to make changes *without affecting* the main codebase (usually called `main` or `master`).

You can switch branches using `git switch <branch-name>` and create new branches using the same command with the `-c` flag.

# Basic Git Concepts: Remote

A **remote** is a version of your repository that is hosted on the internet.

It allows you to *collaborate* with others by sharing your code and changes. Typically, you would have a single remote called `origin` that points to your GitHub repository.

You can add a remote using `git remote add <name> <url>` and push changes to it using `git push <remote> <branch>`.

# Visualising Git

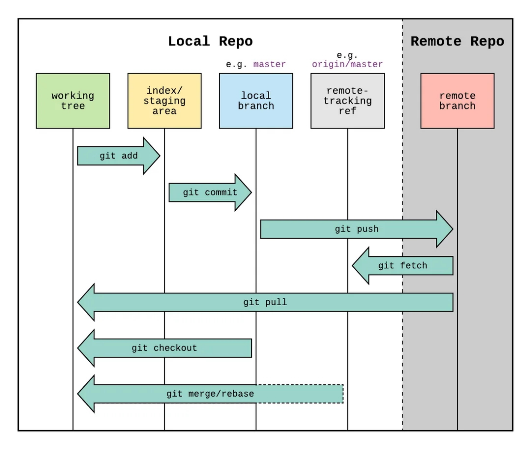


**Figure 1:** Git workflow. Source: `https://imgur.com/oodiCnB`

# Exercise: Git Basics

See the `exercise/instructions/git.md` for instructions on how to complete this exercise.

# Testing

# Why Test?

- Tests catch **bugs** early in the development process.
- In addition to this, they can act as a specification for your code.
- They provide **confidence** when making changes or adding new features.

# Types of Tests

- **Unit tests**: Test individual functions or components in isolation.
- **Integration tests**: Test how different components work together.
- **End-to-end tests**: Test the entire application from start to finish.

In this workshop, we will focus primarily on **unit tests**.

# An Example Unit Test

Here is a simple example of a unit test using the `unittest` framework in Python:

```python
import unittest
def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative(self):
        self.assertEqual(add(-1, 1), 0)

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

# What Should Tests Cover?

- **Normal cases**: Test the expected behavior of your functions.
- **Edge cases**: Test the boundaries of your functions (e.g., empty inputs, large inputs).
- **Error cases**: Test how your functions handle invalid inputs or exceptions.

# Example of a Failing Test

From the previous example, here is a test that is expected to fail:

```python
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add_fails_when_not_numbers(self):
        self.assertRaises(TypeError, add("a", True)) # FAILS!
```

# Fixing the Failing Test

To fix the failing test, we need to modify the `add` function to raise a `TypeError` when the inputs are not numbers:

```
import unittest

def add(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Both arguments must be numbers")
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add_fails_when_not_numbers(self):
        with self.assertRaises(TypeError):
            add("a", True)
```

# Exercise: Writing Unit Tests

See the `exercise/instructions/testing.md` for instructions on how to complete this exercise.

# Intro to CI/CD

- **Continuous Integration (CI)**: Automatically running tests and checks on your code whenever changes are made.
- **Continuous Deployment (CD)**: Automatically deploying your code to a production environment.

GitHub provides a built-in CI/CD service called **GitHub Actions** which we will explore next.

# Structure of a GitHub Actions Workflow

A GitHub Actions workflow is defined in a YAML file located in the `.github/workflows` directory of your repository.

Here is a simple example of a workflow that echoes “Hello, World!” whenever code is pushed to the repository:

```yaml
name: Hello World

on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Say Hello
        run: echo "Hello, World!"
```

# Exercise: Writing a CI Workflow

See the `exercise/instructions/ci.md` for instructions on how to complete this exercise.

# APIs with Flask

# What is an API?

An **API** (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other.

The APIs we discuss today will operate over the HTTP protocol and can be used to access data or perform operations on a server.

# Advantages of Using APIs for Research

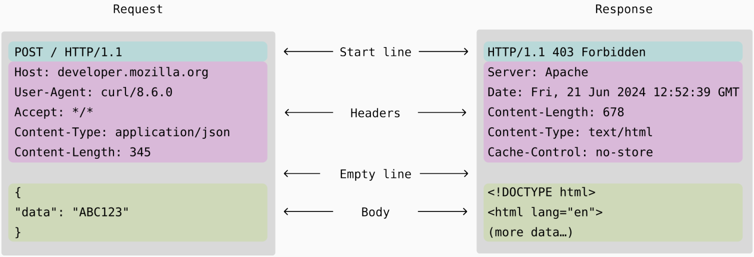
- **Modularity**: APIs allow different parts of a system to be developed and maintained independently.
- **Reusability**: APIs can be reused across different applications or services.
- **Interoperability**: APIs enable different systems to work together, regardless of their underlying technologies.
- **Standardisation**: APIs provide a consistent interface for accessing data and services.

# Basics of HTTP

HTTP (*Hypertext Transfer Protocol*) is the foundation of data communication on the web. There are a few key concepts to understand:

- **Requests**: Clients send requests to servers to access resources. Common HTTP methods include `GET`, `POST`, and `DELETE`.
- **Responses**: Servers respond to requests with status codes (e.g., 200 OK, 404 Not Found) and data (e.g., JSON, HTML).
- **Endpoints**: Specific URLs that represent resources or actions in an API.

# What is in a HTTP Request?



```
POST / HTTP/1.1
Host: developer.mozilla.org
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
"data": "ABC123"
}
```

```
HTTP/1.1 403 Forbidden
Server: Apache
Date: Fri, 21 Jun 2024 12:52:39 GMT
Content-Length: 678
Content-Type: text/html
Cache-Control: no-store

<!DOCTYPE html>
<html lang="en">
(more data…)
```

**Figure 2:** Anatomy of a HTTP Request. Source: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages`

# Basics of Flask

**Flask** is a lightweight web framework for Python that makes it easy to build APIs.

Key features of Flask include:

- Simple routing system (mapping URLs to functions).
- Built-in development server (so you can run the API locally).
- Support for various extensions to add functionality (e.g., database integration, authentication).

# A Simple Flask API Example

Here is a simple example of a Flask API that has one endpoint which returns a greeting message:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'


if __name__ == '__main__':
    app.run(debug=True)
```

# URL Paramters

The below example builds on the previous one by adding a URL parameter to the endpoint:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/greet/<name>')
def greet(name):
    return f'Hello, {name}!'


if __name__ == '__main__':
    app.run(debug=True)
```

# POST Request Example

If we want to pass data to the API (more than just URL parameters), we can use a POST requests. POST requests typically include a body containing the data.

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/square', methods=['POST'])
def square():
    data = request.get_json()
    number = data.get('number', 0)
    return {'result': int(number) ** 2}


if __name__ == '__main__':
    app.run(debug=True)
```

# Exercise: Building a Simple Flask API

See the `exercise/instructions/api.md` for instructions on how to complete this exercise.

# Docker

# What is Docker?

**Docker** is a platform that allows you to package your application and its dependencies into a **container**.

Containers are lightweight, portable, and consistent environments that can run on any system with Docker installed.

# How Does Docker Work?

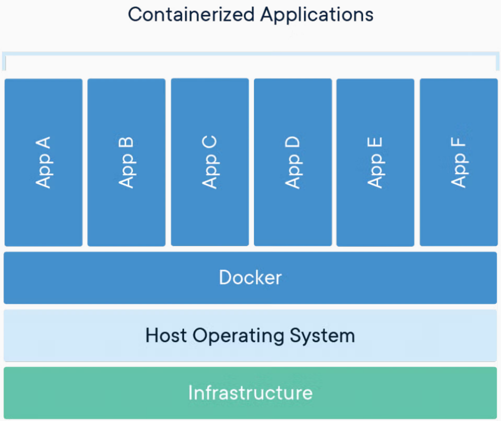


**Figure 3:** Docker in relation to the host system. Source: `https://www.docker.com/resources/what-container/`

# Why Use Docker?

- **Consistency**: Ensures that your application runs the same way across different environments (development, testing, production).
- **Isolation**: Containers isolate your application from the host system and other containers, reducing conflicts.
- **Portability**: Containers can be easily shared and deployed on different systems.
- **Scalability**: Docker makes it easy to scale applications by running multiple containers.

# So How Do We Build a Docker Container?

To build a Docker container, we need to create a `Dockerfile` that specifies the instructions for building the container image. It is every similar to running commands in a terminal.

A Dockerfile typically includes:

- A base image (e.g., Python, Ubuntu) with the `FROM` instruction.
- Instructions to install dependencies (e.g., using `RUN`).
- Commands to copy your application code into the container (using `COPY`).
- The command to run your application (using `CMD`).

# Docker Commands

Here are some common Docker commands:

- `docker build -t <image-name> .`: Build a Docker image from the Dockerfile in the current directory.
- `docker run -p <host-port>:<container-port> <image-name>`: Run a Docker container from the specified image, mapping ports.
- `docker ps`: List running containers.
- `docker stop <container-id>`: Stop a running container.

# Visualising Docker

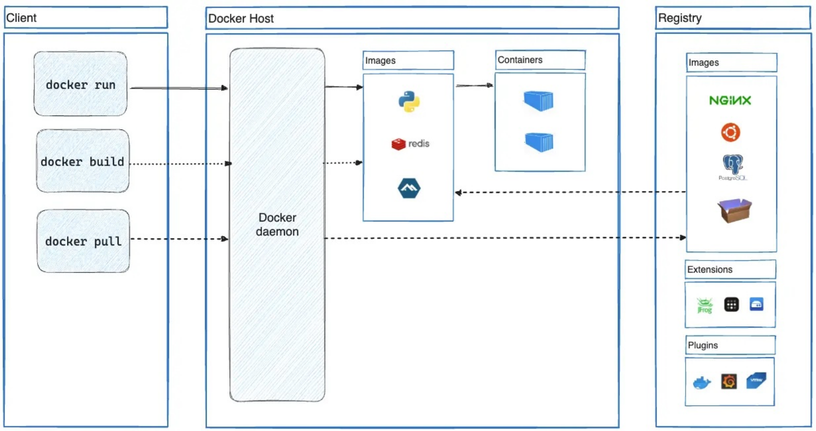


**Figure 4:** Docker architecture. Source: `https://docs.docker.com/get-started/docker-overview/`

# A Simple Dockerfile Example

Here is a simple example of a Dockerfile that creates a container for a Python application:

```dockerfile
# Base image with node preinstalled
FROM node:22-alpine

# Set working directory where commands will be run
WORKDIR /app

# Copy over files
COPY . .

# Install dependencies
RUN npm install

# Run the application when the container starts
CMD ["node", "index.js"]
```

# Exercise: Building a Docker Container

See the `exercise/instructions/docker.md` for instructions on how to complete this exercise.

# What Happens When We Have Multiple Containers?

When you have multiple containers, you can use **Docker Compose** to define and manage multi-container applications.

Docker Compose uses a YAML file (typically called `docker-compose.yml`) to define the different containers (known as *services*) and their configurations. It can also be used to provide a persistent and shared storage between containers using *volumes*.

# Example Docker Compose File

Here is a simple example of a Docker Compose file that defines two services: a API service and a database.

```yaml
services:
  app:
    image: myapp:latest
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres:13
    # Env variables go here
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

# Docker Compose Commands

Here are some common Docker Compose commands:

- `docker compose up`: Start the services defined in the Docker Compose file.
- `docker compose down`: Stop and remove the services.

# Conclusion

# Summary

In this workshop, we covered several key concepts for sustainable software engineering:

- **Version Control** with Git and GitHub
- **Testing** and CI/CD
- Building **APIs** with Flask
- **Building and packaging** with Docker

Remember, sustainable software engineering is an ongoing process that requires continuous learning and adaptation.