Web Programming

Testing and CI/CD
What's next?

Prof. Josué Obregón

Department of Industrial Engineering- ITM

Seoul National University of Science and Technology

Objectives

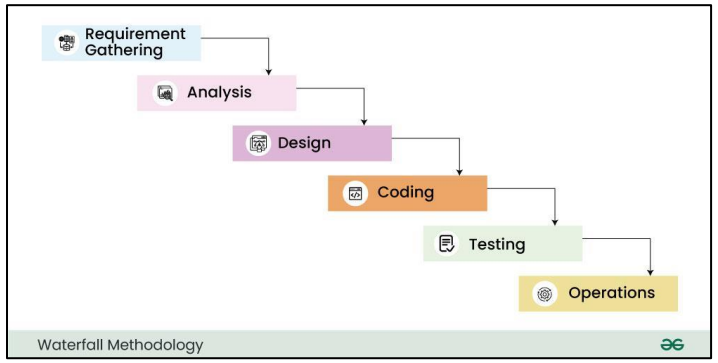
- Identify and describe common best practices that aid in developing modern web applications
- Wrap-up and summarize all the content of the course and get a glance on what's next

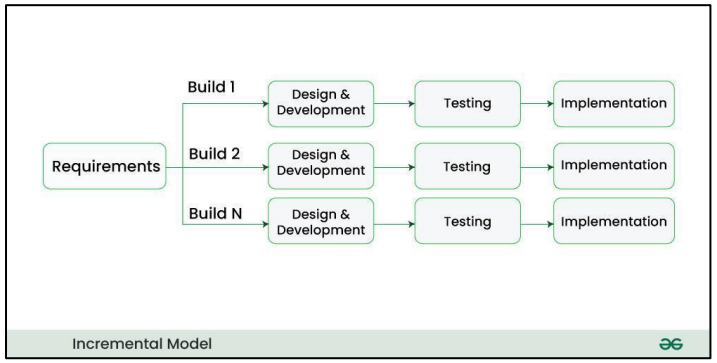
Agenda

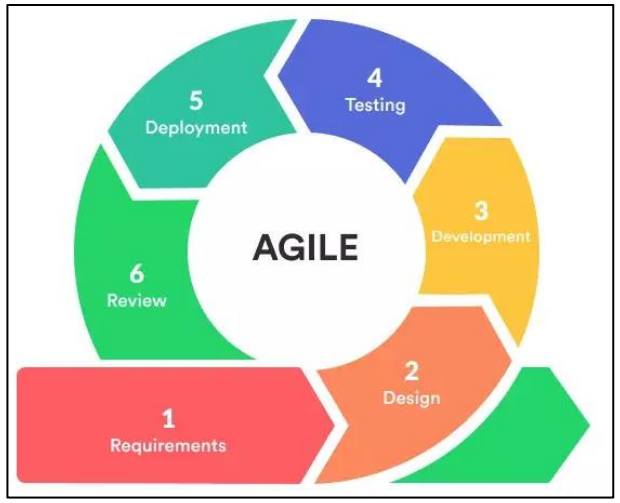
- Testing
- Continuous Integration (CI)
- Continuous Delivery (CD)
- Containerization with Docker
- What's next?

Introduction to Software Best Practices

- Software Development Life Cycle (SDLC)







https://www.geeksforgeeks.org/software-development-process/

Testing in Web Development

- Ensures and verifies code correctness
  - Functions work as they are supposed to
  - Webpages behave as intended
- Improves Code Quality
- Facilitates Continuous Integration and Delivery
  - Efficiently and effectively test code as our applications grow large
  - Code changes are continuously tested, leading to faster and safer releases
- Types of Testing: Unit, Integration, System

Basic Testing with Python's `assert`

- Simple way to verify code correctness
- We can run tests in Python by using the `assert` command
- This command is followed by some expression that should be True
- If the expression is `True`, nothing will happen, and if it is `False`, an exception will be thrown

```
def square(x):
    return x*x

assert square(10) == 100

""" Output:

""
```

```
def square(x):
    return x + x

assert square(10) == 100

""" Output:
Traceback (most recent call last):
  File "assert.py", line 4, in <module>
    assert square(10) == 100
AssertionError
"""
```

Test-Driven Development

- Development style where every time you add a new functionality or fix a bug, you add a test that checks everything works correctly
- The test include a growing set of tests that are run every time you make changes
- When used for adding new functionality (TDD methodology)
  - It emphasizes writing a failing test case first, then writing the minimum amount of code necessary to pass the test, and finally refactoring the code for optimization and clarity.
- When encountering a bug
  - Fix the bug, and then add set of tests

Unit Testing with `unittest`

- Definition of Unit Testing
  - A software testing technique where individual components or functions of an application are tested in isolation to ensure that each part works as expected.
- Purpose of Unit Testing
  - validate that each unit of the software performs as designed.
- Python `unittest` Library
  - A built-in Python library that provides a framework for writing and running unit tests.
- Central concept of a test case, which is a single unit of testing
  - The `unittest.TestCase` class provides methods to set up test conditions and validate outcomes.

Example

```
import unittest

# The function to be tested
def add(a, b):
    return a + b

# Test case for the add function
class TestAddFunction(unittest.TestCase):
    def test_add_positive_numbers(self):
        """Check that 2 + 5 = 5."""
        self.assertEqual(add(2, 3), 5)

    def test_add_negative_numbers(self):
        """Check the sign rule of addition."""
        self.assertEqual(add(-1, -1), -2)

    def test_add_zero(self):
        """Check the Identity Property of addition."""
        self.assertEqual(add(5, 0), 5)

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

- Inherit from `TestCase`
- `TestAddFunction` functions pattern:
  - The name of the functions begin with test_
  - The first line of each function contains a docstring surrounded by three quotation marks
  - Other assert functions https://docs.python.org/3/library/unittest.html#assert-methods

Example output

```
FFF
======================================================================
FAIL: test_add_negative_numbers
(__main__.TestAddFunction.test_add_negative_numbers)
Check the sign rule of addition.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 15, in test_add_negative_numbers
    self.assertEqual(add(-1, -1), -2)
AssertionError: -1 != -2

======================================================================
FAIL: test_add_positive_numbers
(__main__.TestAddFunction.test_add_positive_numbers)
Check that 2 + 5 = 5.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 11, in test_add_positive_numbers
    self.assertEqual(add(2, 3), 5)
AssertionError: 6 != 5

======================================================================
FAIL: test_add_zero (__main__.TestAddFunction.test_add_zero)
Check the Identity Property of addition.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 19, in test_add_zero
    self.assertEqual(add(5, 0), 5)
AssertionError: 6 != 5

----------------------------------------------------------------------
Ran 3 tests in 0.002s
```

```
...
----------------------------------------------------------------------
---------------
Ran 3 tests in 0.000s

OK
```

Testing Django Applications

- Apply the ideas of automated testing when creating Django applications
- Creating test cases for models and views
- Django provide automatically the `tests.py` file when creating a project
  - Django test library is automatically imported

Example: Django Tests Airline app

models.py

```
class Flight(models.Model):
    origin = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="departures")
    destination = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="arrivals")
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id} : {self.origin} to {self.destination}"

    def is_valid_flight(self):
        return self.origin != self.destination and self.duration >= 0
```

Example: Setting up our tests

test.py

```
from django.test import Client, TestCase

from .models import Airport, Flight, Passenger

# Create your tests here.
class FlightTestCase(TestCase):

    def setUp(self):

        # Create airports.
        a1 = Airport.objects.create(code="AAA", city="City A")
        a2 = Airport.objects.create(code="BBB", city="City B")

        # Create flights.
        Flight.objects.create(origin=a1, destination=a2, duration=100)
        Flight.objects.create(origin=a1, destination=a1, duration=200)
        Flight.objects.create(origin=a1, destination=a2, duration=-100)
```

Example: Creating test cases

test.py

```
    def test_arrivals_count(self):
        a = Airport.objects.get(code="AAA")
        self.assertEqual(a.arrivals.count(), 1)

    def test_valid_flight(self):
        a1 = Airport.objects.get(code="AAA")
        a2 = Airport.objects.get(code="BBB")
        f = Flight.objects.get(origin=a1, destination=a2, duration=100)
        self.assertTrue(f.is_valid_flight())

    def test_invalid_flight_destination(self):
        a1 = Airport.objects.get(code="AAA")
        f = Flight.objects.get(origin=a1, destination=a1)
        self.assertFalse(f.is_valid_flight())

    def test_invalid_flight_duration(self):
        a1 = Airport.objects.get(code="AAA")
        a2 = Airport.objects.get(code="BBB")
        f = Flight.objects.get(origin=a1, destination=a2, duration=-100)
        self.assertFalse(f.is_valid_flight())
```

Client testing

- We also need to check whether individual web pages load as intended.
- We can do this by creating a `Client` object in our Django testing class, and then making requests using that object.
- Add `Client` to our imports
- We can create test cases that make sure we get the right HTTP response code
  - 200 for success responses, 404 for invalid responses, etc.
- We can also check test that the context is generated correctly

Example: Creating Client test cases

test.py

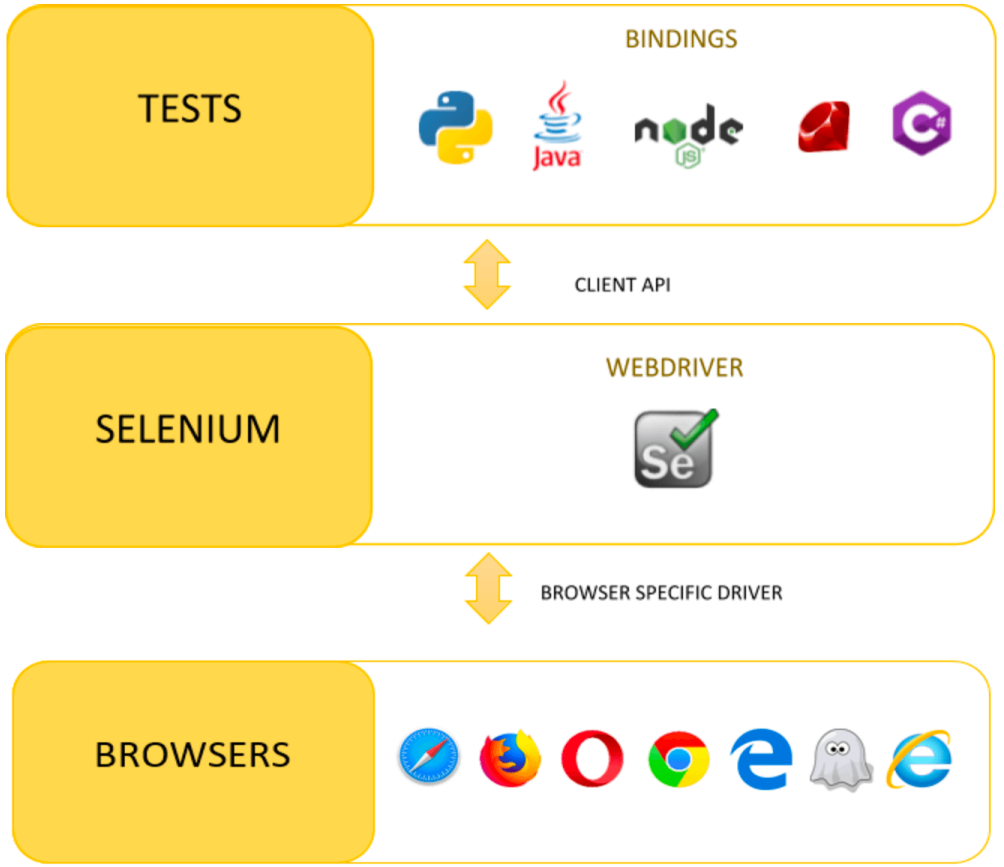
```
    def test_index(self):
        c = Client()
        response = c.get("/flights/")
        print(response)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context["flights"].count(), 3)

    def test_valid_flight_page(self):
        a1 = Airport.objects.get(code="AAA")
        f = Flight.objects.get(origin=a1, destination=a1)

        c = Client()
        response = c.get(f"/flights/{f.id}")
        self.assertEqual(response.status_code, 200)

    def test_invalid_flight_page(self):
        max_id = Flight.objects.all().aggregate(Max("id"))["id__max"]

        c = Client()
        response = c.get(f"/flights/{max_id + 1}")
        self.assertEqual(response.status_code, 404)
```

What about client-side testing?

- Client-side testing verifies that all interactive elements of the user interface (UI) work as intended.
  - This includes buttons, forms, links, and dynamic content.
- Ensures that users can interact with the application as expected, preventing issues like broken buttons, non-responsive forms, and other interaction problems.
- It enhances user experience, identify browser-specific issues and validate client-side logic

Technologies for Client-Side Testing

- **Selenium**: A widely-used open-source tool for automating web browsers. It allows you to write scripts in various programming languages (such as Python, Java, and JavaScript) to simulate user interactions with web applications.
- Ideal for end-to-end testing of web applications, verifying that the application works as expected in a real browser environment.



https://www.selenium.dev/

Example: Selenium Tests

```
import unittest

from selenium import webdriver

def file_uri(filename):
    return pathlib.Path(os.path.abspath(filename)).as_uri()

driver = webdriver.Chrome()

class WebpageTests(unittest.TestCase):

    def test_title(self):
        driver.get(file_uri("counter.html"))
        self.assertEqual(driver.title, "Counter")

    def test_increase(self):
        driver.get(file_uri("counter.html"))
        increase = driver.find_element_by_id("increase")
        increase.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "1")


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

Technologies for Client-Side Testing

- **Jest**: A JavaScript testing framework developed by Facebook, commonly used for testing React applications.
- It focuses on simplicity and supports a variety of testing needs, including unit, integration, and snapshot testing.
- Ensures that individual components and functionalities of a web application work as expected.

https://jestjs.io/

Example: Jest tests

Testing a simple function

```
function sum(a, b) {
    return a + b;
  }

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Testing an asynchronous function

```
test('the data is peanut butter', () => {
    return fetchData().then(data => {
      expect(data).toBe('peanut butter');
    });
  });
```

CI/CD

- Stands for Continuous Integration and Continuous Delivery or Deployment aims to streamline and accelerate the software development lifecycle
- CI refers to the practice of automatically and frequently integrating code changes into a shared source code repository
  - Frequent merges to main branch
  - Automated unit testing
- CD is a 2-part process that refers to the integration, testing, and delivery of code changes
  - Short release cycles

Benefits of CI/CD

- Tackles Small Conflicts Incrementally
  - Many conflicts may arise when multiple features are combined at the same time
- Easier Isolation of Code Issues
  - Unit tests are run with each merge
- Isolates Problems Post-Launch
  - Frequent releasing of new versions help to quickly isolate problems
- Gradual Introduction of New Features
  - Releasing small, incremental changes allows users to slowly get used to new app features
- Competitive Advantage with Rapid Releases

Technologies for CI/CD

GitHub Actions

- GitHub tool integrated into GitHub that automates workflows for building, testing, and deploying code.
- In order to set up a GitHub action, we'll use a configuration language called YAML.
  - YAML structures its data around key-value pairs (like a JSON object or Python Dictionary). Here's an example of a simple YAML file:

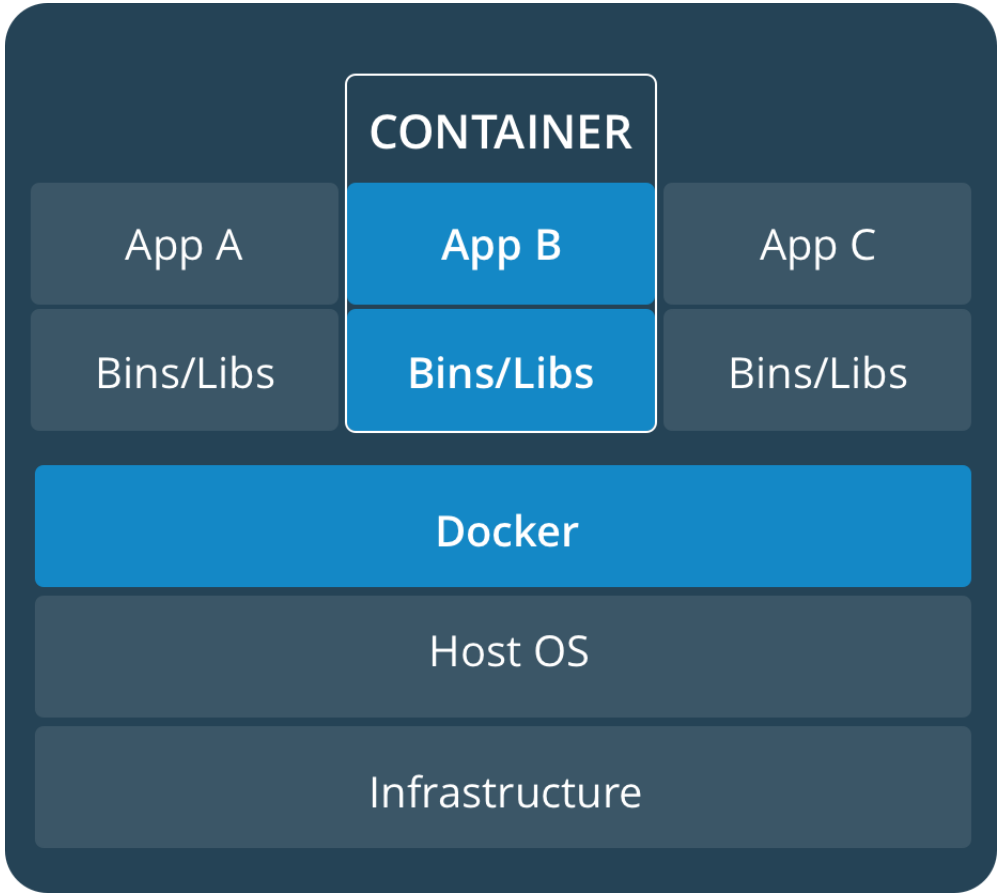
```
name: Testing
on: push

jobs:
  test_project:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Run Django unit tests
      run: |
        pip3 install --user django
        python3 manage.py test
```

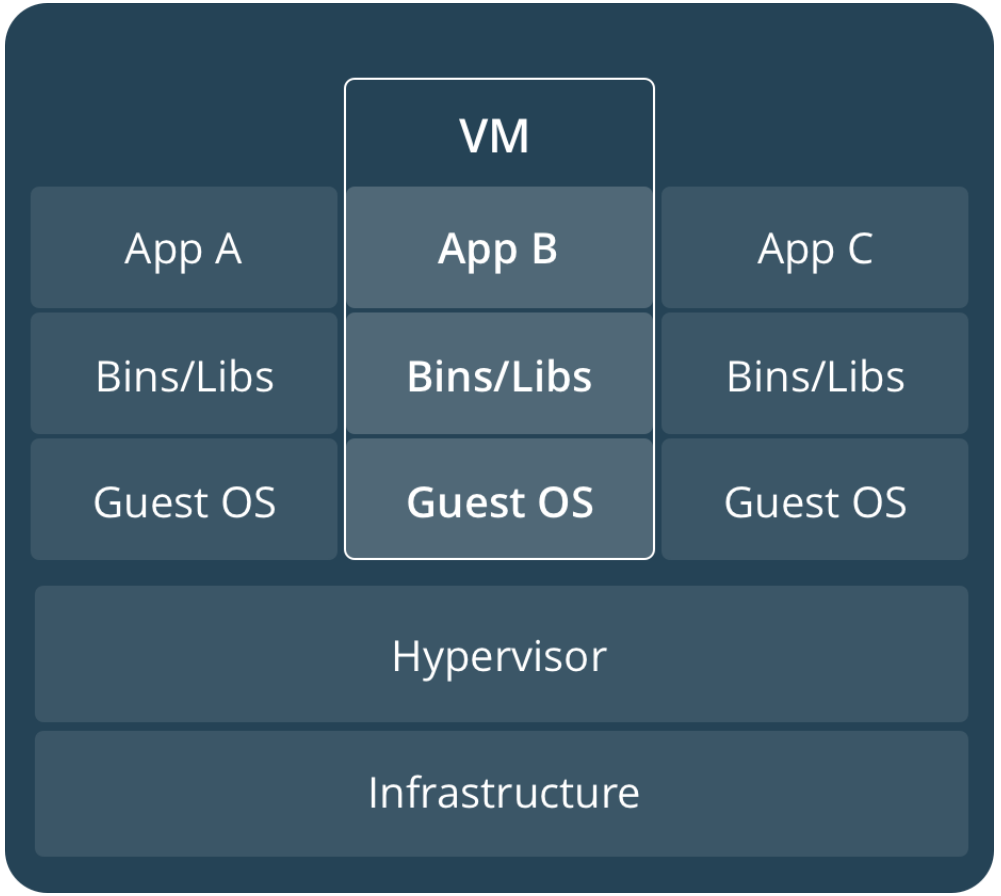
Technologies for CI/CD

Docker

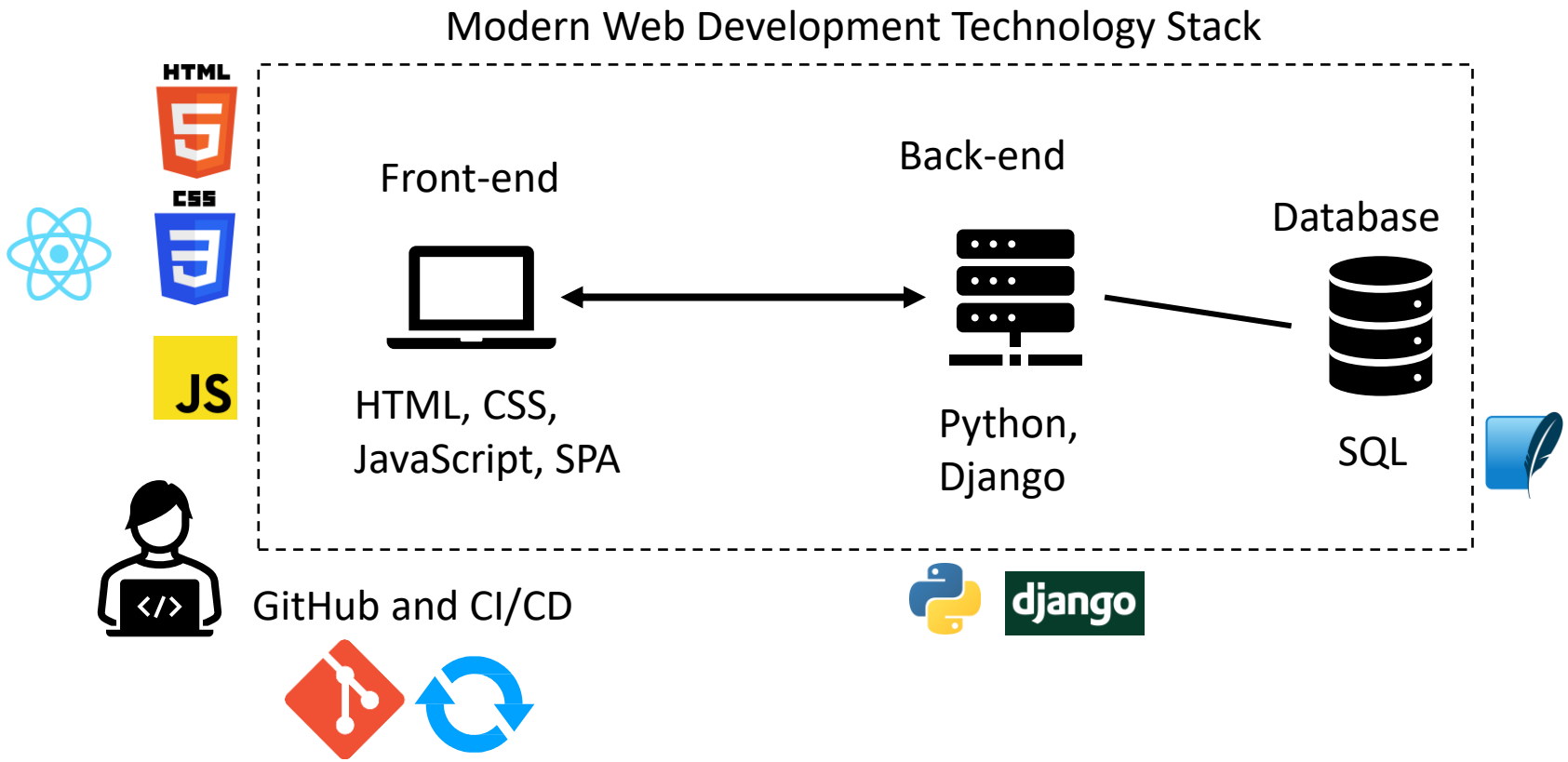
- Platform that uses containerization to create, deploy, and run applications in isolated environments, ensuring consistency across multiple development and deployment environments.
- Docker ensures that applications run in the same environment from development to production by encapsulating everything needed to run the software, including the code, runtime, libraries, and dependencies.
- Docker containers are lightweight and share the host system's kernel, leading to lower overhead compared to traditional virtual machines.

Docker


CONTAINER
App A
App B
App C
Bins/Libs
Bins/Libs
Bins/Libs
Docker
Host OS
Infrastructure


VM
App A
App B
App C
Bins/Libs
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Guest OS
Hypervisor
Infrastructure

Wrap-up


Modern Web Development Technology Stack
HTML
CSS
JS
Front-end
Back-end
Database
HTML, CSS,
JavaScript, SPA
Python,
Django
SQL
GitHub and CI/CD
django

Other Web Frameworks

- Server-side
  - Python → Flask, FastAPI
  - JavaScript → Express.js, Next.js
  - Java → Spring
  - PHP → Laravel
  - Ruby → Ruby on Rails
- Client-side (JavaScript-based)
  - Angular
  - Vue.js
  - React (actually a library)
    - Next.js
  - Backbone.js

Web Programming