CBIT Open Source Community

In Association With **CDC**

Presents

REACTJS & FASTAPI BOOTCAMP

FAST API

A Python Web framework

Backend Server Basics

- A backend server is the **core component** of a web application that processes client requests, performs necessary computations, interacts with **databases**, and returns the appropriate **responses**.

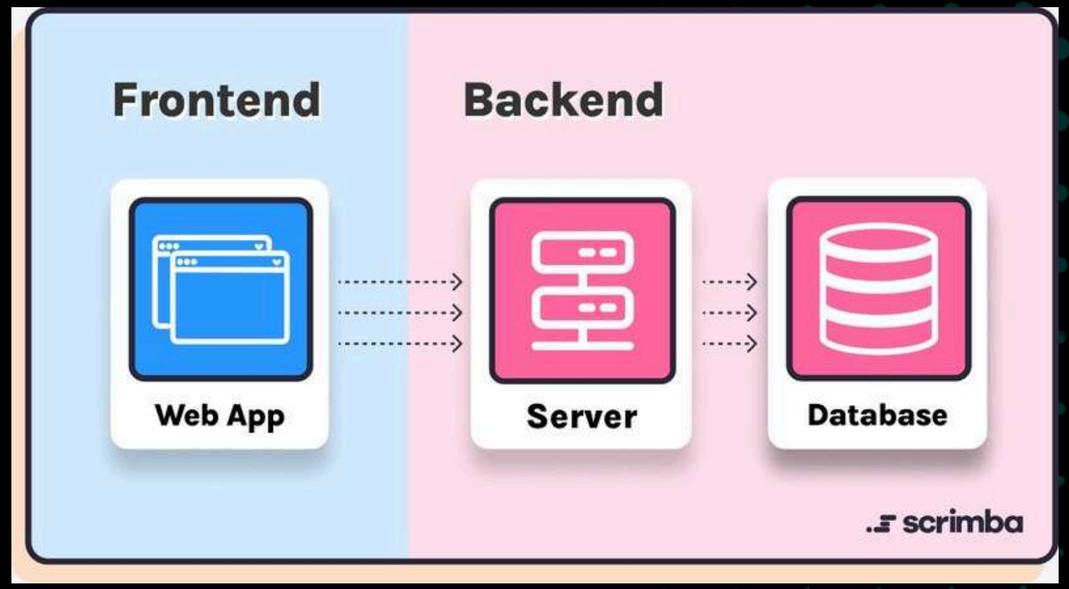
- It serves as the intermediary between the user-facing **frontend** and the server-side **databases** or external services, ensuring secure, reliable, and efficient communication.

Backend Server Basics

- A server is a **centralized computer program** that provides resources, services, or data to other computers, known as clients, over a network.
- It plays a vital role in **hosting websites**, managing databases, processing client requests, and ensuring secure communication.
- Servers can be specialized for different tasks, such as web servers for hosting websites, database servers for managing data, hardware accelerated servers for computationally intensive tasks like ML and application servers for running software applications.

The Magic behind “The Backend”

- User interacts with the Frontend.
- Frontend sends a request to the Backend.
- Backend queries the Database.
- Database returns data to the Backend.
- Backend sends processed data back to the Frontend.
- Frontend displays updated data to the User.



The Need for a Backend

- **Database Interactions and Processing and Storing Data:** Efficiently handles data retrieval and updates.
- **Business Logic:** Implements core functionality and rules.
- **User Authentication:** Manages user identities and access control.
- **API Integrations:** Connects with third-party services and APIs.
- **Scalability:** Ensures the application can grow with demand.
- **Consistency:** Provides a unified experience across platforms.
- **Security:** Protects sensitive information and maintains data integrity.

Types of APIs

SOAP APIs

- These APIs use Simple Object Access Protocol. Client and server exchange messages using XML. This is a less flexible API that was more popular in the past.

RPC APIs

- These APIs are called Remote Procedure Calls. The client completes a function (or procedure) on the server, and the server sends the output back to the client.

Types of APIs

WebSocket APIs

- Websocket API is another modern web API development that uses JSON objects to pass data. A WebSocket API supports two-way communication between client apps and the server.
- The server can send callback messages to connected clients, making it more efficient than REST API.

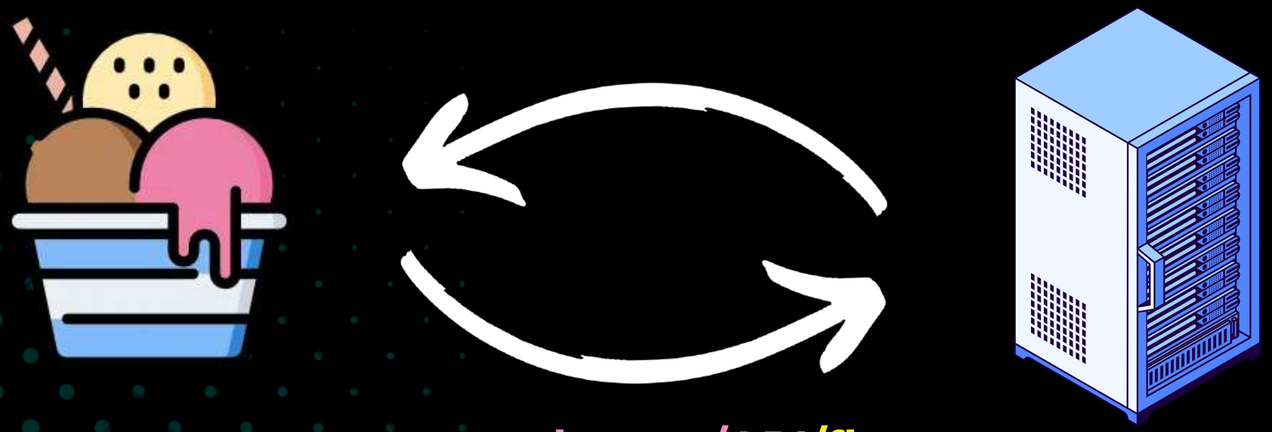
REST APIs

- These are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client. Let’s look at REST APIs in more detail below.

REST APIs

- REST stands for **Representational State Transfer**. REST defines a set of functions like **GET, PUT, DELETE,** etc. that clients can use to access server data.

- The main feature of **REST API** is **statelessness**. Statelessness means that servers do not save client data between requests. It is **simple** and **standardised** and also **scalable**



www.example.com/API/flavours

HTTP Request

What It Is: An HTTP request is a message sent from the client (e.g., your browser or an app) to the server asking for some action to be performed. This could be to retrieve data, send data, or perform some other operation.

CRUD.txt — Python

```
Create  --> POST method
Read    --> GET method
Update  --> PUT method
Delete  --> DELETE method
```

HTTP Response

What It Is: An HTTP response is the server's reply to the HTTP request. It contains the requested data, or a message indicating the result of the request.

Components:

- **Status Code:** Indicates the outcome of the request (e.g., 200 for success, 404 for not found, 500 for server error).
- **Headers:** Provide information about the response, like the content type (e.g., JSON) or caching policies.
- **Body:** Contains the data requested or a message, usually in a format like JSON or HTML.

Adding Banana Flavoured Ice Cream

Adding_Banana.py — Python

```
#POST Method
  {
  'id': 2,
  'flavour': 'Banana'
}
```

API Status Codes

When an API responds to a request, it usually contains 3 main components:

- Status Code
- Header
- Body

The status code: It is a three-digit number sent by the server to indicate the result of the request.

Common status codes include:

- **2xx OK**: They are used to indicate success of requested operation
- **3xx Redirection**: They are used to indicate that requested resource has been moved
- **4xx Client Errors**: The request was invalid or malformed.
- **5xx Server Errors:** Errors occurred at the server side while processing the request

API Headers

Headers: These provide additional information about the response. They include metadata such as content type, length, server information, etc.

Examples of headers include:

- **Content-Type:** Indicates the media type of the response body (e.g., application/json, text/html).
- **Content-Length:** Specifies the size of the response body in bytes.

Response Body

Body: The response contains the actual data sent back from the server. Depending on the request and the response, the body can include:

- **Data**: For successful **GET** or **READ** requests, this could be the requested resource (e.g., a JSON object, file, etc).
- **Confirmation Messages**: For **POST** or **PUT** requests, this might include a confirmation message or the details of the created/updated resource.
- **Error Details**: In the case of errors, the body might contain a description of what went wrong or instructions on how to correct it.

FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.
Important features:

1. Based on Open Standards
2. Automatic Docs
3. Modern Python
4. Editor Support
5. Validation
6. Security and Authentication
7. Dependency Injection
8. Unlimited Plug-Ins and Community support
9. Concise

FastAPI

FastAPI is fully compatible with (and based on) Starlette (a lightweight Asynchronous Server Gateway Interface (ASGI) framework/toolkit, which is ideal for building async web services in Python) and Pydantic (widely used data validation library for Python)

Installation

FastAPI needs Starlette and Pydantic as dependencies. It installs them if not already installed. To install FastAPI, we get it from the Python Package Index (PyPI) using pip (Python's package-management tool):

Installation

```
$ pip install fastapi
```

FastAPI Basics

We write some python functions to run on each type of request for a specific route and FastAPI takes care of the rest, right from starting up a server to managing incoming requests, routing them to the right function and returning the response back to the client.

FastAPI uses decorators to make a Python Function act as an API Endpoint, it does all the heavy-lifting to manage the server and create the routing map. As FastAPI is built based on PyDantic, we will also explore a bit of validation here.

Getting Started

We will need a Python file with a FastAPI app instance and one or more route-handlers. For example, a file named server.py containing the route-handlers.

Importing FastAPI and creating an application instance

We can then import the FastAPI class from the fastapi module and create an application instance using it.

Installation

```
// import the FastAPI class
from fastapi import FastAPI

app = FastAPI() // creates a new FastAPI application instance
```

Getting Started

FastAPI provides decorators that can be used to define our route-handlers for specific request methods and we can also pass in the route-URL as an argument to it to specify the endpoint corresponding to this function.

Hello world API

```
// use a decorator of the form app.<method>
@app.get("/")
def read_root():
    return {"message": "Hello, World!"}
```

The handler can return various forms of output, like strings, dictionaries, Pydantic schema objects, and more

Running the FastAPI Application with Uvicorn

To run your FastAPI application, you typically use Uvicorn, a fast ASGI server for Python.

Installation
To install Uvicorn, you can use pip:

Installation

```
$ pip install uvicorn
```

Running FastAPI with Uvicorn

After you've created your FastAPI application, you can run it using Uvicorn. For example, if your FastAPI app is defined in a file named main.py, you can start your server by running:

Installation

```
$ uvicorn main:app --reload
```

- main refers to the file name (without the .py extension).
- app refers to the FastAPI instance in your code.
- --reload tells Uvicorn to automatically reload the server when code changes, which is useful during development.

Running FastAPI with Uvicorn

This will spin-up a server on the localhost port 8000 (or another port if port 8000 is already in use). The routes on the server will be as per the route-handlers defined in the server.py file.

We can check it by going to http://localhost:8000 on a web browser and we should see a JSON response with the object we returned from the function {"Hello": "World"}

To view the **interactive documentation** for API routes, go to http://localhost:8000/docs on a web browser.

There is also an Alternative documentation format supported by FastAPI, we can view that by going to http://localhost:8000/redoc

Route Parameters

Sometimes it is not possible to specify all the paths separately but it is possible to specify a pattern for the routes and in such cases, we can add parameters to our Route-URLs to create dynamic routes. There are 2 types of parameters:

Path parameters - They are a part of the endpoint-url
Example: /users/{id} represents a parameter id which can take any value.
Sample routes that match this pattern would be: /users/41, /users/103ab

Query parameters - They are added after the endpoint-url.
They follow the ?key1=value1&key2=value2 notation.
Example: /cars?color=blue&type=sedan

Example: Path Parameters in FastAPI

We can pass a path-string with {param} syntax to the decorator to add path parameter(s) as below time_of_day and name are Route Paramters

Path parameters

```python
@app.get("/greet/{time_of_day}/{name}")
def read_path_params(time_of_day: str, name: str):
    return {"message": f"Good {time_of_day}, {name}!"}
```

Usage

```
http://127.0.0.1:8000/greet/morning/SaiKiran
```

Example: Query Parameters in FastAPI

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

Query Parameters

```python
@app.get("/greet")
def read_query_params(time_of_day: str, name: str):
    return {"message": f"Good {time_of_day}, {name}!"}
```

Usage

```
http://127.0.0.1:8000/greet?time_of_day=morning&name=SaiKiran
```

Pydantic

Pydantic is the most widely used validation library for Python. It is written in Rust and is fast and extensible, building upon Pythons type hints, it is an easy to use robust way to validate data.

One of the primary ways of defining schema in Pydantic is via models. Models are simply classes which inherit from pydantic.BaseModel and define fields as annotated attributes.

Models share many similarities with Python's dataclasses

Pydantic:Basic Usage

request.py Python

```python
from pydantic import BaseModel, Field #import the base class

class User(BaseModel): #create our model using a class definition
    id: int    #defining fields with type hints and additionally
   name: str = Field('Jane Doe') #a default value (similar to Python dataclasses)
```

Pydantic: Basic Usage

Then, we can instantiate an object of this model (class) with the values for each field and two things can happen (broadly):

- The fields satisfy validation constraints and we successfully create an instance of the model which is guaranteed to follow the defined schema.
  - Eg: user = User(id='123')

- Or, the validation process results in a ValidationError showing that the input data is not and can not be coerced to the schema provided by the Model and object creation fails.
  - Eg: user = User(id='1a')

Pydantic: Basic Usage

- The model's instance is guaranteed to have data that adheres to the Model Schema but the input itself may or may not be so, this is a subtle difference but it is important to understand.

- The data from the model's instance can be converted to a dictionary using the model_dump() method on the instance. This is different from dict(user) in terms that dic() doesn't recursively convert nested models to dictionaries while .model_dump() does.

CRUD Operations in FastAPI

Create: Add a new resource.

Read: Retrieve existing resources.

Update: Modify existing resources.

Delete: Remove existing resources.

Use Cases:

-Managing items in an inventory system.

-Handling user accounts in an application.

-Working with blog posts or articles in a content management system.

Setting Up the Backend

FastAPI and SQLite

```
from fastapi import FastAPI
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

app = FastAPI()
Base = declarative_base()
engine = create_engine("sqlite:///./test.db")
SessionLocal = sessionmaker(bind=engine)
```

SQLite is a lightweight, disk-based database that doesn’t require a separate server.

Database Model and Pydantic Schemas

SQLAlchemy: Defines the database schema.

Pydantic: Validates and parses the data..

```
class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True)
    title = Column(String)
    description = Column(String)

class ItemCreate(BaseModel):
    title: str
    description: str
```

Setting Up the Frontend

Initialize the React Project

```
npm create vite@latest . -- --template react
npm install
```

Install Axios

```
npm install axios
```

Create state variables

```
// Inside App.jsx
const [items, setItems] = useState([]);
const [title, setTitle] = useState("");
const [description, setDescription] = useState("");
const [editId, setEditId] = useState(null);
```

Create Operation in FastAPI

The "Create" operation uses the POST method to add new resources to the server. For example, you might add a new item to a database by sending the necessary data as a JSON object. FastAPI uses Pydantic models to validate and structure this data before processing it.

Create Operation

Backend

```
@app.post("/items/", response_model=ItemResponse)
def create_item(item: ItemCreate):
    db = SessionLocal()
    db_item = Item(**item.dict())
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

Create Operation

```
const createItem = async () => {
    await axios.post("http://127.0.0.1:8000/items/", {
      title,
      description,
    });
    setTitle("");
    setDescription("");
    fetchItems();
  };
```

```
<form onSubmit={handleSubmit}>
     <input
        type="text"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
     />
     <input
        type="text"
        value={description}
        onChange={(e) => setDescription(e.target.value)}
     />
     <button type="submit">Create</button>
</form>
```

Read Operation in FastAPI

The "Read" operation relies on the GET method, allowing you to retrieve one or more resources. You can fetch all available items or filter them based on specific criteria, such as an item ID. Path parameters are often used to dynamically retrieve specific resources.

Read Operation

Backend

```
@app.get("/items/", response_model=List[ItemResponse])
def read_items():
  db = SessionLocal()
  items = db.query(Item).all()
  db.close()
  return items
```

Read Operation

```
const [items, setItems] = useState([]);
const fetchItems = async () => {
    const response = await axios.get("http://127.0.0.1:8000/items/");
    setItems(response.data);
  };

useEffect(() => {
    fetchItems();
  }, []);
```

```
<ul>
    {items.map((item) => (
        <li
            key={item.id}
            <div className=" flex flex-row items-center gap-2">
              <strong>{item.title}</strong>: {item.description}
            </div>
         </li>
        ))}
</ul>
```

Update Operation in FastAPI

When you need to modify existing data, the "Update" operation comes into play. Using the PUT or PATCH methods, you can either fully replace a resource or update only specific fields. This operation also leverages Pydantic models to ensure data consistency.

Update Operation

Backend

```
@app.put("/items/{item_id}", response_model=ItemResponse)
def update_item(item_id: int, item: ItemCreate):
  db = SessionLocal()
  db_item = db.query(Item).filter(Item.id == item_id).first()
  if db_item is None:
    raise HTTPException(status_code=404, detail="Item not found")
  db_item.title = item.title
  db_item.description = item.description
  db.commit()
  db.refresh(db_item)
  db.close()
  return db_item
```

Update Operation

```
{/* Add this to item */}
<button
     onClick={() => {
         setEditId(item.id);
         setTitle(item.title);
         setDescription(item.description);
      }}>
         Edit
</button>
```

```
const updateItem = async () => {
    await axios.put(`http://127.0.0.1:8000/items/${editId}`, {
      title,
      description,
    });
    setTitle("");
    setDescription("");
    setEditId(null);
    fetchItems();
  };
```

Delete Operation in FastAPI

Finally, the "Delete" operation removes a resource from the server using the DELETE method. This operation typically requires specifying the resource by an identifier, such as an item ID. It's essential to handle this operation carefully, potentially implementing soft deletes to preserve data integrity.

Delete Operation

Backend

```
@app.delete("/items/{item_id}")
def delete_item(item_id: int):
  db = SessionLocal()
  db_item = db.query(Item).filter(Item.id == item_id).first()
  if db_item is None:
      raise  HTTPException(status_code=404,  detail="Item  not
found")
  db.delete(db_item)
  db.commit()
  db.close()
  return {"ok": True}
```

Delete Operation

```
{/* Add this to item */}
<button onClick={() => deleteItem(item.id)}>
    Delete
</button>
```

```
const deleteItem = async (id) => {
    await axios.delete(`http://127.0.0.1:8000/items/${id}`);
    fetchItems();
  };
```

Middleware

- A middleware runs between the reception of all requests and the execution of the function that actually runs at the given route’s functionality.
- Middlewares can be used to perform many things including but not limited to handling authentication, body parsing, CORS handling etc.
- Multiple middlewares can be chained together to perform multiple tasks sequentially.

Why Middleware?

- Middlewares allow programmers to reduce repetitive code and allow programmers to process all incoming requests uniformly without much effort.

- The need also arises due to the fact that data formats that are efficient to send across a network are not necessarily human readable or easily malleable in the backend.

How to use Middlewares in FastAPI?


```
from fastapi import FastAPI, Request
app = FastAPI()
@app.middleware("http")
    async def authenticate_user(request: Request, call_next):
        ... # Authentication Logic
        ...
        response = await call_next(request)
        return response
```

Async Operations in FastAPI

- Asynchronous programming is a core feature in FastAPI, allowing you to handle many requests simultaneously without blocking the server.
- By defining routes with async def, FastAPI can pause operations with await, freeing the server to handle other requests in the meantime.
- This is especially useful for I/O-bound operations like database queries or external API calls, where waiting for a response could otherwise hold up your entire application.
- Using async operations ensures that your FastAPI application remains highly performant and scalable, capable of serving many users concurrently without slowdowns.

BackgroundTasks() in FastAPI

Non-Blocking Execution: FastAPI allows for background task execution with the BackgroundTasks class, ensuring that time-consuming tasks don't block the response to the client.

Use Cases:

- Sending emails
- Logging data
- Processing files

How It Works:

1. **Immediate Response:** When a request is received, FastAPI sends the response to the client right away.
2. **Background Task:** The task continues to run in the background after the response is sent.

BackgroundTasks Example


```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

def write_log(message: str):
    with open("log.txt", "a") as log:
        log.write(message + "\n")

@app.post("/send-notification/")
async def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "Notification sent")
    return {"message": "Notification sent, logging in the background"}
```

Request Class in FastAPI

The Request class is designed for handling HTTP requests in FastAPI. It inherits from HTTPConnection, adding additional functionality specific to HTTP requests.

Importing Request:

```
from fastapi import Request
```

Request Class Example

request.py — Python

```python
from fastapi import FastAPI, Request
app = FastAPI()
@app.get("/items/{item_id}")
async def read_item(item_id: int, request: Request):
    headers = request.headers
    query_params = request.query_params
    body = await request.body()
    return {
        "item_id": item_id,
        "headers": headers,
        "query_params": query_params,
        "body":body.decode()
    }
```

HTTPConnection Class in FastAPI

The HTTPConnection class is a more general class that can be used for both HTTP and WebSocket connections. It provides functionalities common to both connection types.

Importing HTTPConnection:

request.py — Python

```python
from fastapi.requests import HTTPConnection
```

HTTPConnection Class Example

request.py — Python

```python
from fastapi import FastAPI
from fastapi.requests import HTTPConnection

app = FastAPI()

@app.get("/info")
async def get_info(conn: HTTPConnection):
    url = conn.url
    headers = conn.headers
    return {"url": url, "headers": headers}
```

Dependencies

- In FastAPI, managing dependencies and security is crucial for building robust applications.
- Two key functions for handling these aspects are Depends() and Security().
- Both functions allow you to declare dependencies, but they cater to different needs.

Depends()

- The Depends() function is used to declare dependencies in FastAPI. It enables you to inject reusable components or functions into your route handlers.
- Import Depends from fastapi and use it to specify a callable that FastAPI will invoke.
- Allows you to declare a dependency that is automatically resolved by FastAPI.
- By default, dependencies are cached within a single request. Set use_cache to False if you need the dependency to be re-evaluated if used multiple times in the same request.

Example

request.py — Python

```python
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

async def common_parameters(q: str | None = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(commons: Annotated[dict, Depends(common_parameters)]):
    return commons
```

In this example, common_parameters is a dependency injected into the read_items endpoint. It provides query parameters and pagination options.

Machine Learning

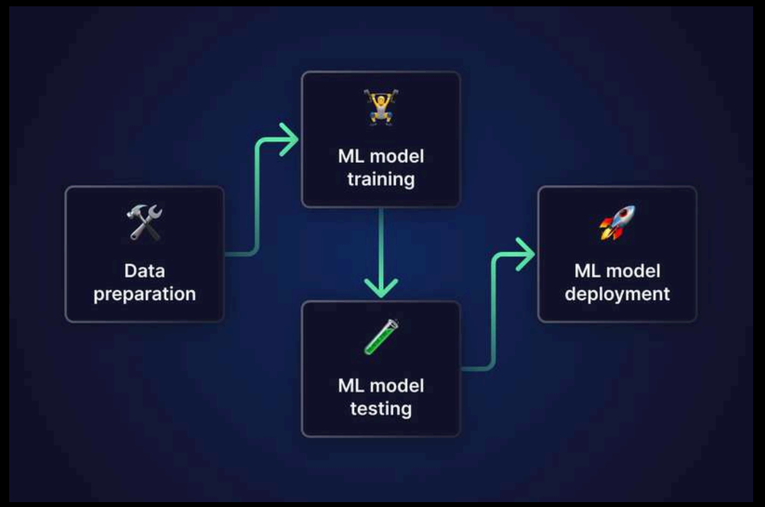
Exporting and Integrating an ML model with FastAPI

Machine Learning

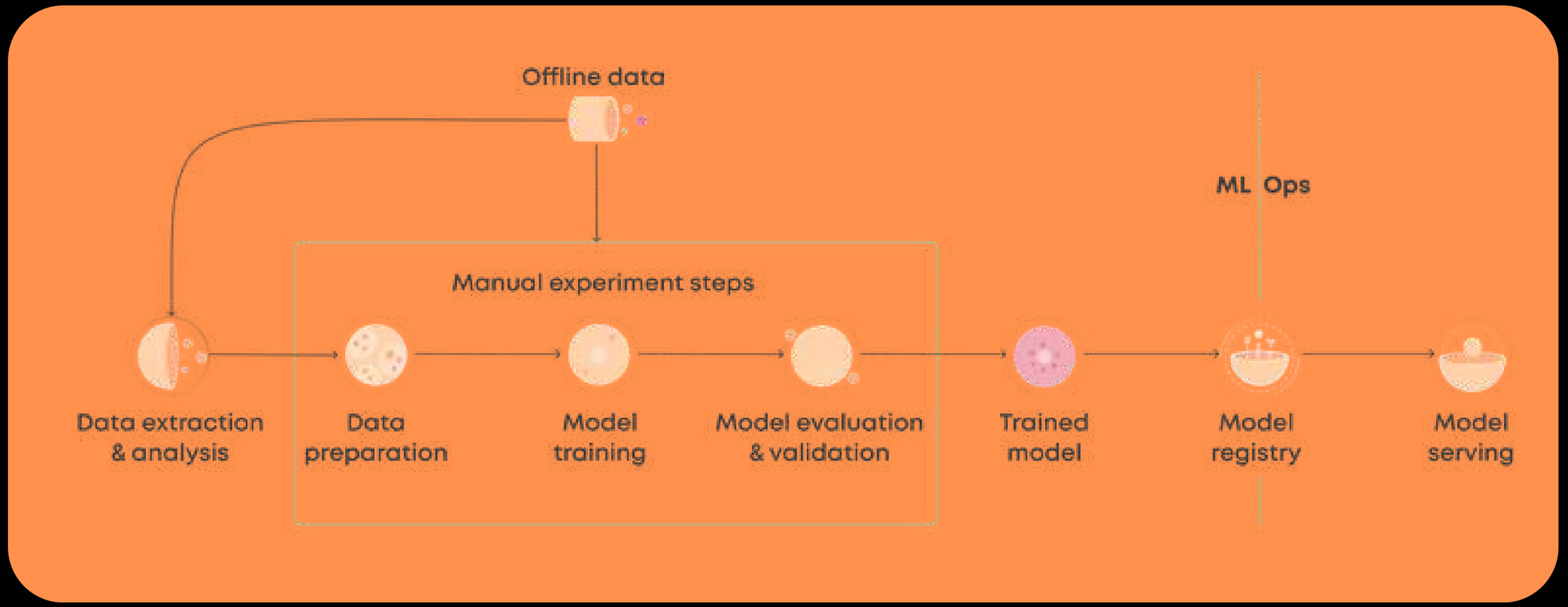
Machine Learning is a subset of Artificial Intelligence, where a “machine” learns without explicitly being told to do so.

Types of Machine Learning:

- Supervised Learning
- Unsupervised Learning



Pipeline



Upon obtaining data to create a model for, split the data to train the model and use the remaining to test it. You can evaluate your model's accuracy and perfirmance on the basis of how close it is to the expected output.

Classification vs Regression

Classification

- **Definition:** Classification involves predicting a discrete label or category for a given input. The model assigns the input data to one of several predefined classes or categories.
- **Output:** The output of a classification model is categorical, meaning it belongs to a specific class or label, such as *spam* or *not spam*, *cat* or *dog*, etc.

Regression

- **Definition:** Regression involves predicting a continuous numeric value for a given input. The model estimates the relationship between the input variables (features) and the output, which is typically a real number.
- **Output:** The output of a regression model is continuous, meaning it can take any value within a range, such as predicting a price, temperature, or salary.

Basics of Linear Regression

Linear regression is a statistical method that models the relationship between a dependent variable (target) and one or more independent variables (features) by fitting a linear equation to the observed data.

The simplest form is simple linear regression, which uses one independent variable to predict a dependent variable. The equation of a simple linear regression line is:

$$y=mx+c$$

where:

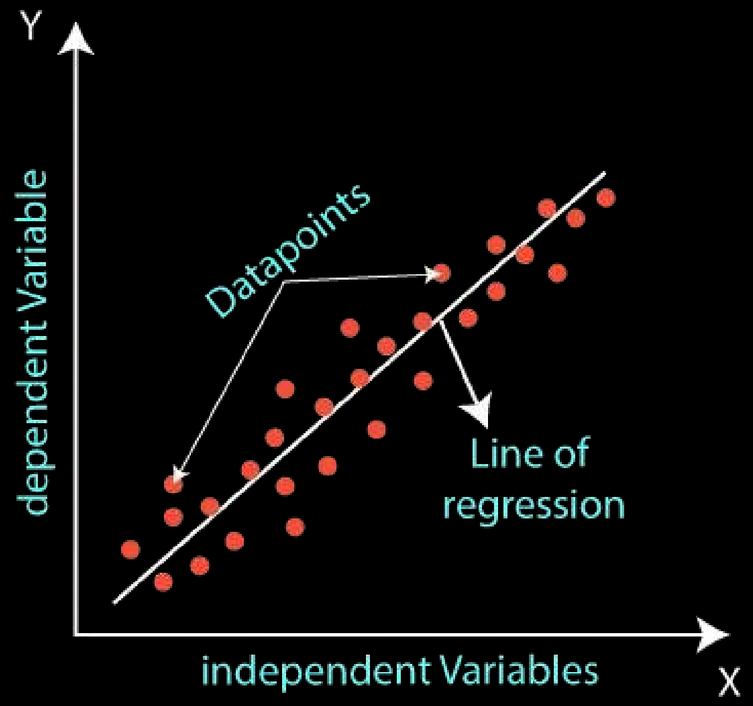
y is the dependent variable (target)

x is the independent variable (feature)

m is the slope of the line (coefficient)

c is the y-intercept (constant)

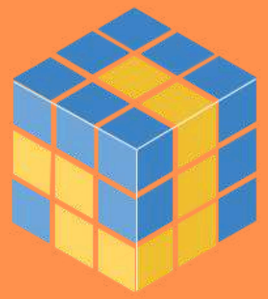
Basics of Linear Regression



The goal of linear regression is to find the values of m and c that minimize the error between the predicted values and the actual values.

Tools Used

scikit learn



NumPy

pandas

Building Our Model

We'll use a dataset, where the Price of a Car is predicted based on its Year of Manufacture.

ML.py

```python
import pandas as pd
from sklearn.linear_model import LinearRegression
import pickle

# Example dataset
data = {
    'year': [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009],
    'price': [2000, 2100, 2150, 2200, 2300, 2400, 2450, 2500, 2600, 2700]
}
```

Building Our Model

ML.py

```
# Loading the dataset into a DataFrame
df = pd.DataFrame(data)

# Getting features and target from the dataset
X = df[['year']]
y = df['price']

# Creating and fitting our Linear Regression model
model = LinearRegression()
model.fit(X, y)

# Saving the model to a file
with open('car_price_model.pkl', 'wb') as f:
    pickle.dump(model, f)
```