COS30049 – Computing Technology Innovation Project

Week 11 – Building a Full-Stack Application

(Lecture – 01)

Ningran Li (Icey)
ningranli@swin.edu.au

Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

- **Recap Front-end & Back-end**
- **React API Call**
- **Data Visualization**
- **Full-stack integration**
- **Debugging and Troubleshooting**
- **Best Practices in Code Development**

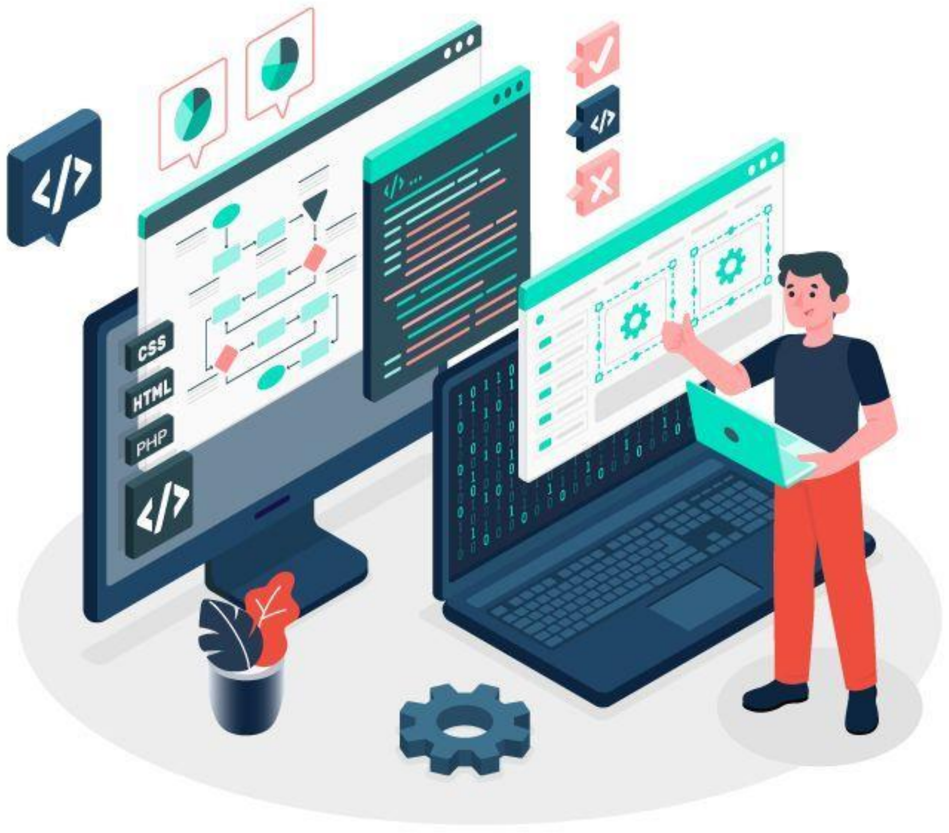


Image from [Pinterest]

React Lifecycle

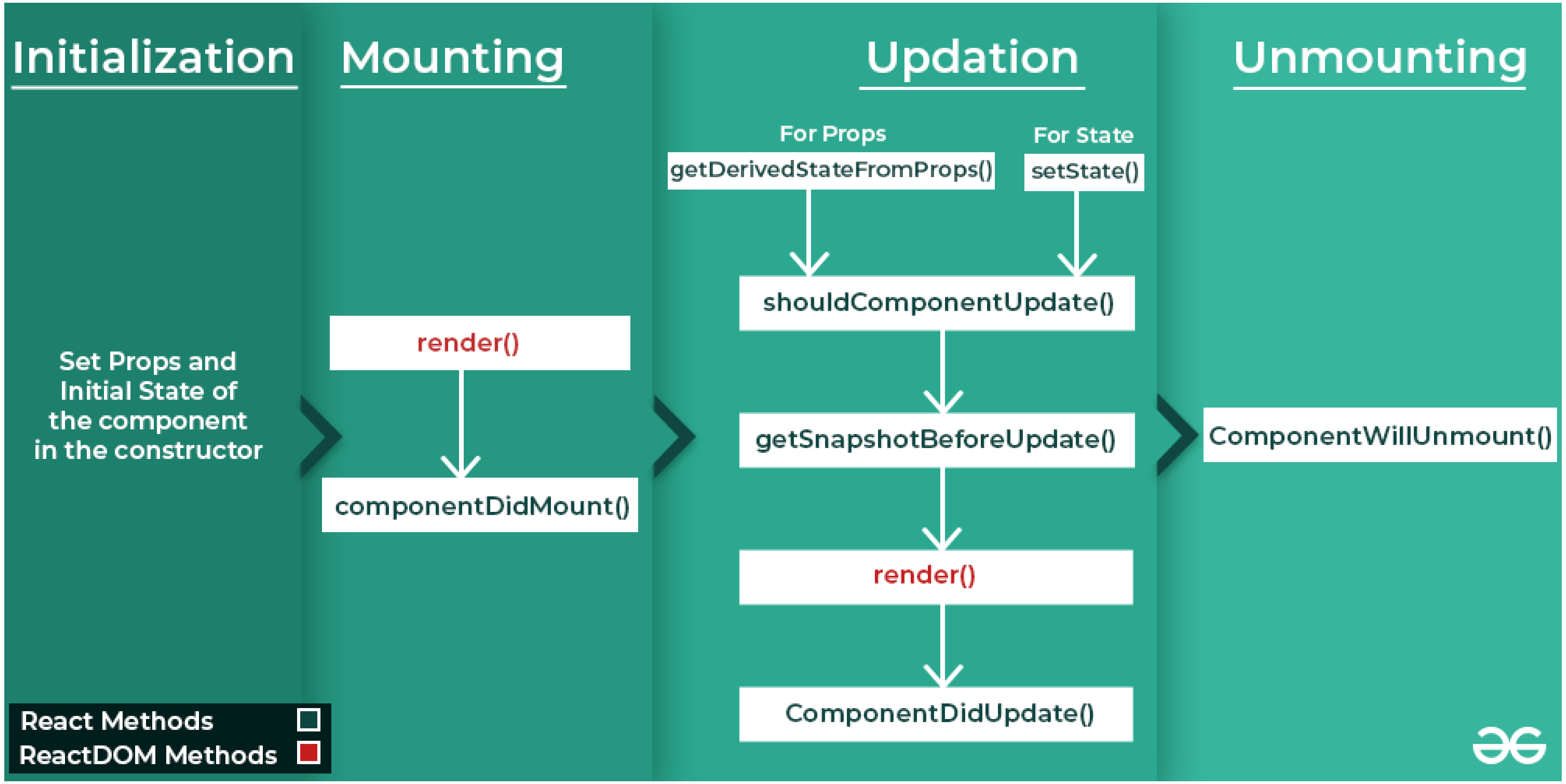


Image from [GFG]

FastAPI

What is FastAPI?

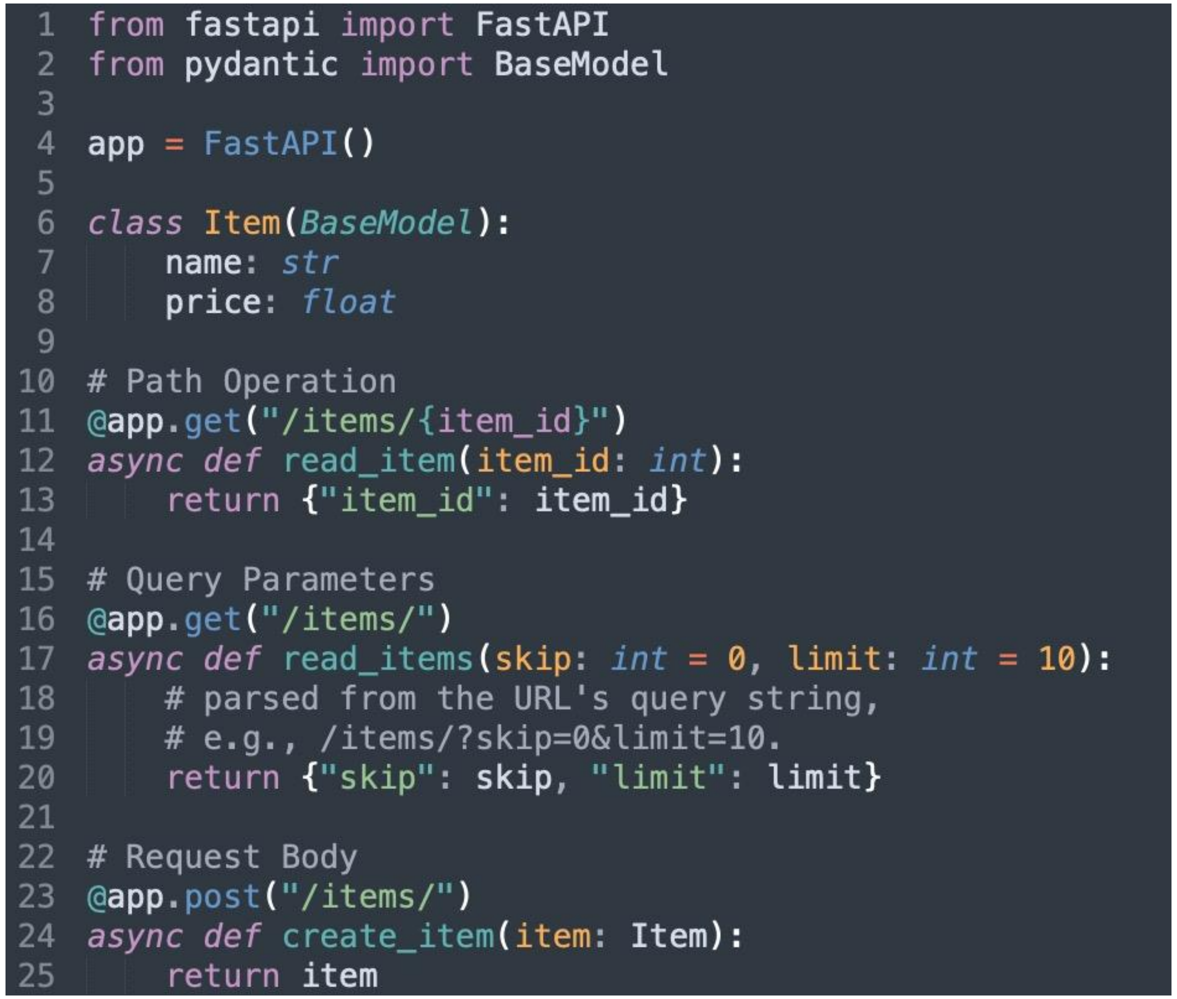
- Modern, fast (high-performance) web framework for building APIs with Python 3.6+
- Based on Starlette and Pydantic

Key Features

Fast to code | Fewer bugs | Intuitive | Easy to use and learn

Basic Operations

- Path Operations: Define API endpoints using decorators
- Query Parameters: Automatically parsed from the URL
- Request Body: Use Pydantic models to declare the structure of request bodies



```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float

# Path Operation
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}

# Query Parameters
@app.get("/items/")
async def read_items(skip: int = 0, limit: int = 10):
    # parsed from the URL's query string,
    # e.g., /items/?skip=0&limit=10.
    return {"skip": skip, "limit": limit}

# Request Body
@app.post("/items/")
async def create_item(item: Item):
    return item
```

- Recap Front-end & Back-end
- **React API Call**
- Data Visualization
- Full-stack integration
- Debugging and Troubleshooting
- Best Practices in Code Development

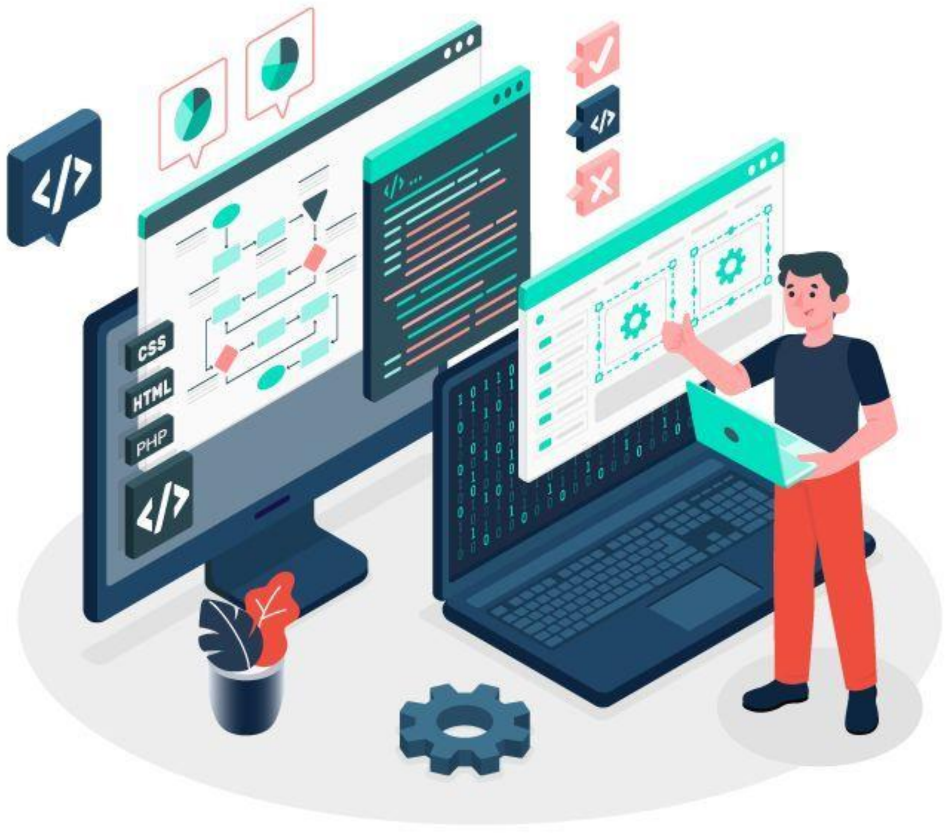
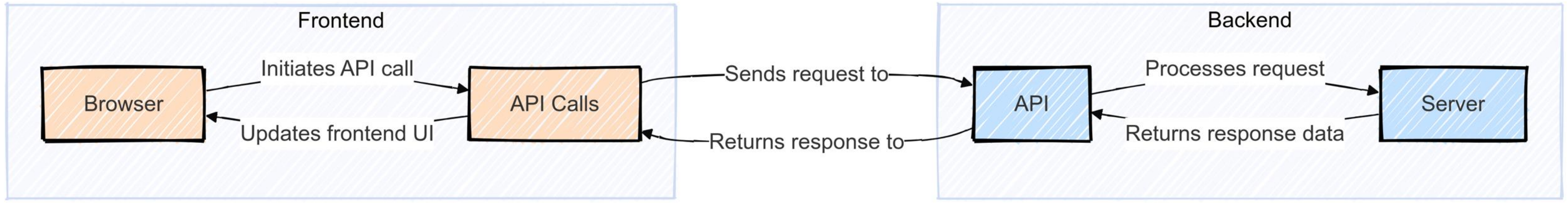


Image from [Pinterest]

Frontend API Calls vs Backend API Interfaces



Location

- Frontend API Call: Client-side (Browser)
- Backend API Interface: Server-side

Language

- Frontend: JavaScript/TypeScript
- Backend: Python, Java, etc.

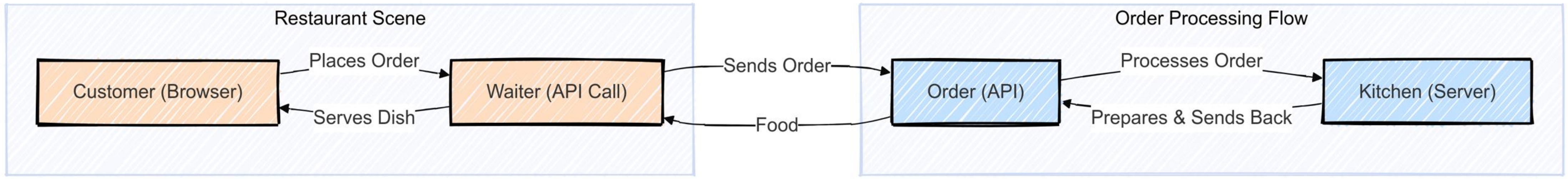
Purpose

- Frontend: Requests data/services
- Backend: Processes requests, returns responses

Relationship

- Frontend initiates communication
- Backend responds with data/results
- They work together to create a complete application

Analogy: Restaurant Order System



Make React API Calls (XMLHttpRequest)

1. useState Hook

- Initializes a state variable data with a default value of null and provides a function setData to update the value of data
- The useState hook is used to handle state within functional components in React

2. handleClick Function

- This function is triggered when the user clicks a button (defined in 3)
- an XMLHttpRequest object (xhr) is created to make an HTTP request to the API endpoint **https://api.example.com/data**
- xhr.open('GET', ...) sets up the request method (GET) and the target URL
- The xhr.onload callback is triggered when the request is complete
- If the request is successful (status 200), it parses the received JSON response (xhr.responseText) and updates the data state using setData
- xhr.send() sends the request to the server.

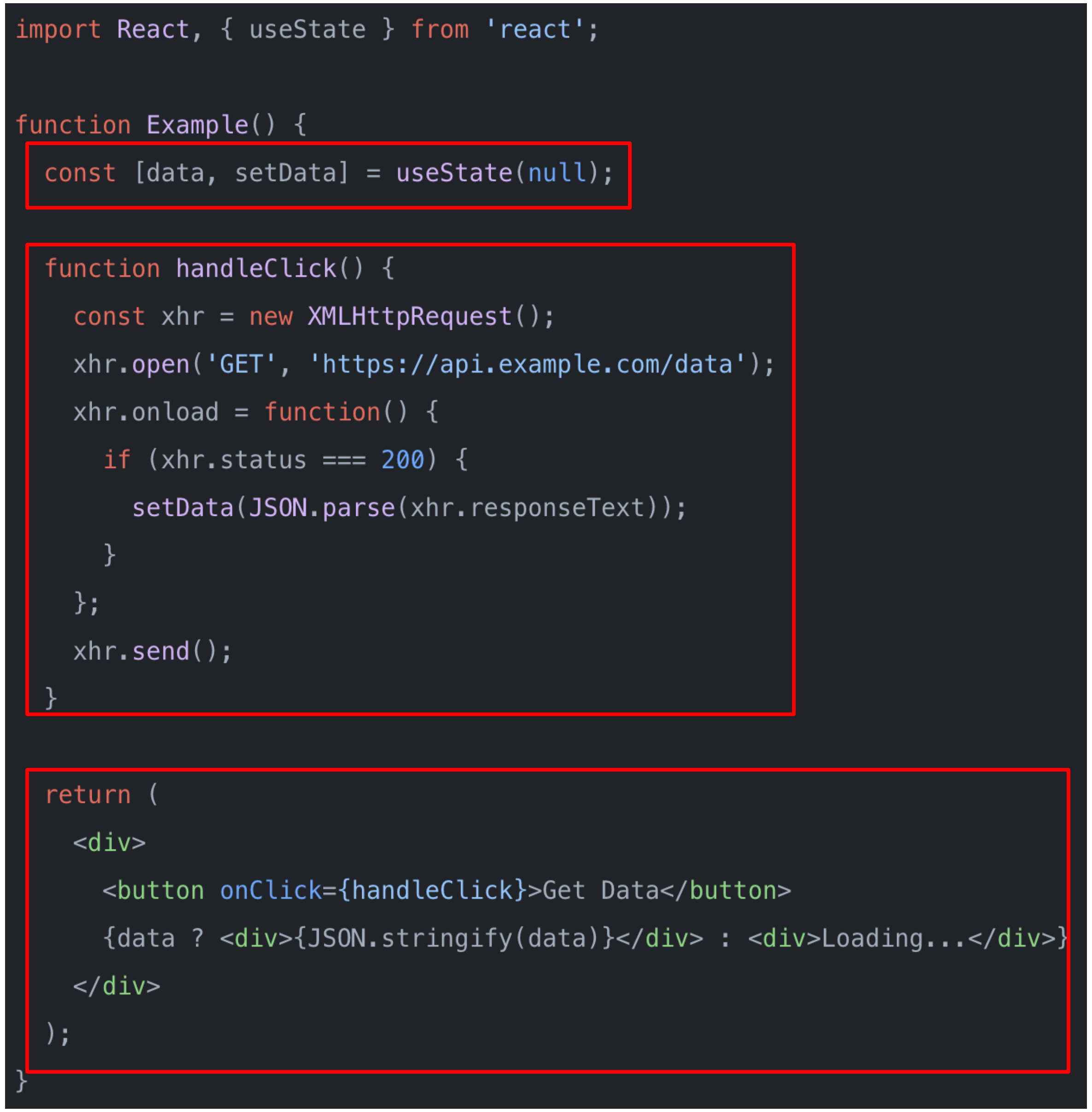
3. JSX (UI Rendering)

- This part of the code defines what will be rendered on the screen.
- This part of the code defines what will be rendered on the screen.
- The code also conditionally renders the response data
- If data is not null, it displays the data inside a <div>. The data is converted to a string using JSON.stringify(data)
- If data is still null (i.e., before the data is fetched), it shows a "Loading..." message instead

1

2

3



```
import React, { useState } from 'react';

function Example() {
  const [data, setData] = useState(null);

  function handleClick() {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://api.example.com/data');
    xhr.onload = function() {
      if (xhr.status === 200) {
        setData(JSON.parse(xhr.responseText));
      }
    };
    xhr.send();
  }

  return (
    <div>
      <button onClick={handleClick}>Get Data</button>
      {data ? <div>{JSON.stringify(data)}</div> : <div>Loading...</div>}
    </div>
  );
}
```

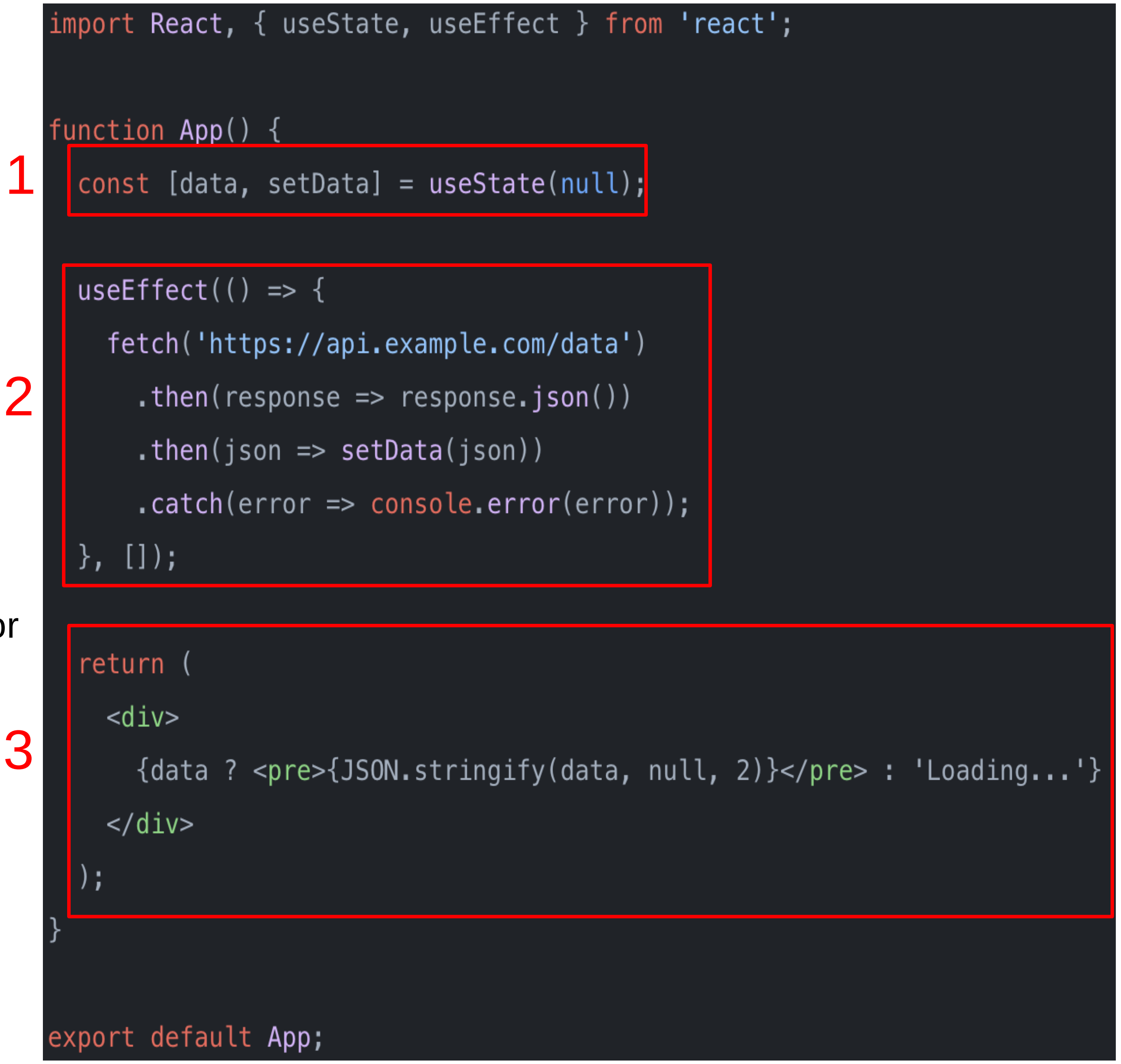
Make React API Calls (Fetch API)

1. useState Hook

2. useEffect Hook

- a React hook that performs side effects in functional components, like fetching data
- The fetch function is used to make a GET request to https://api.example.com/data
- When the response is received, it's converted into JSON format using response.json()
- After parsing the JSON, the result is passed to setData, which updates the data state
- The empty array [] as the second argument to useEffect ensures that this effect runs only once, when the component mounts, mimicking the behavior of componentDidMount in class-based components
- If an error occurs during fetching, it is caught by the catch block, which logs the error to the console.

3. JSX (UI Rendering)



```
import React, { useState, useEffect } from 'react';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(json => setData(json))
      .catch(error => console.error(error));
  }, []);

  return (
    <div>
      {data ? <pre>{JSON.stringify(data, null, 2)}</pre> : 'Loading...'}
    </div>
  );
}

export default App;
```

Make React API Calls (Axios)

1. useState Hook

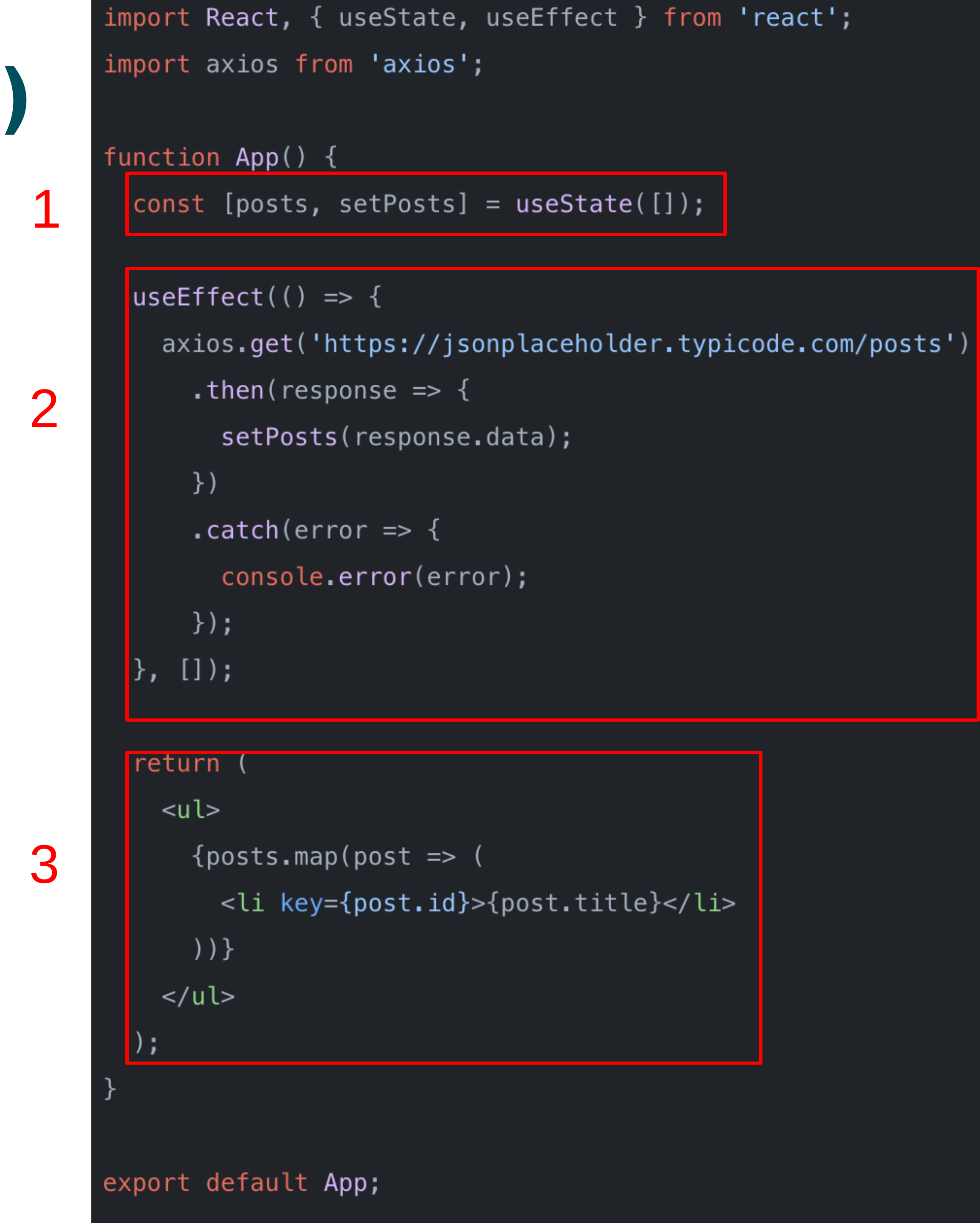
- posts is a state variable initialized as an empty array. It will store the list of posts retrieved from the API
- setPosts is a function used to update the state when the posts are fetched

2. useEffect Hook

- The useEffect hook is used to run side effects, like fetching data, after the component has been rendered
- Inside useEffect, an axios.get request is made to the https://jsonplaceholder.typicode.com/posts API to retrieve posts
- axios.get returns a Promise which, upon resolution, contains the API response. The .then() method processes this response
- response.data contains the array of posts from the API, which is passed to setPosts to update the state
- If there's an error during the request, it's caught by the .catch() method, and the error is logged to the console
- The empty dependency array [] ensures this effect only runs once, mimicking the behavior of componentDidMount in class components.

3. JSX (UI Rendering)

- The component renders an unordered list (<ul>)
- Inside the list, posts.map() is used to iterate over each post from the posts array
- For each post, a list item (<li>) is created displaying the post.title. The key attribute is set to post.id, ensuring each list item has a unique identifier (important for React's rendering optimization)



```
import React, { useState, useEffect } from 'react';
import axios from 'axios';


function App() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    axios.get('https://jsonplaceholder.typicode.com/posts')
      .then(response => {
        setPosts(response.data);
      })
      .catch(error => {
        console.error(error);
      });
  }, []);

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}


export default App;
```

Make React API Calls (Key Differences)

XMLHttpRequest:

- An older API that is callback-based and has verbose syntax. It supports older browsers but requires manual handling for most tasks.

Fetch API:

- A modern, Promise-based API that is simpler to use but lacks progress monitoring and requires additional handling for features like timeouts.

Axios:

- A Promise-based library that simplifies HTTP requests with built-in features like JSON handling and request timeouts. Requires installation but provides more advanced features out of the box.

Let‘s continue learning and mastering the skills in workshop!

- Recap Front-end & Back-end
- React API Call
- **Data Visualization**
- Full-stack integration
- Debugging and Troubleshooting
- Best Practices in Code Development

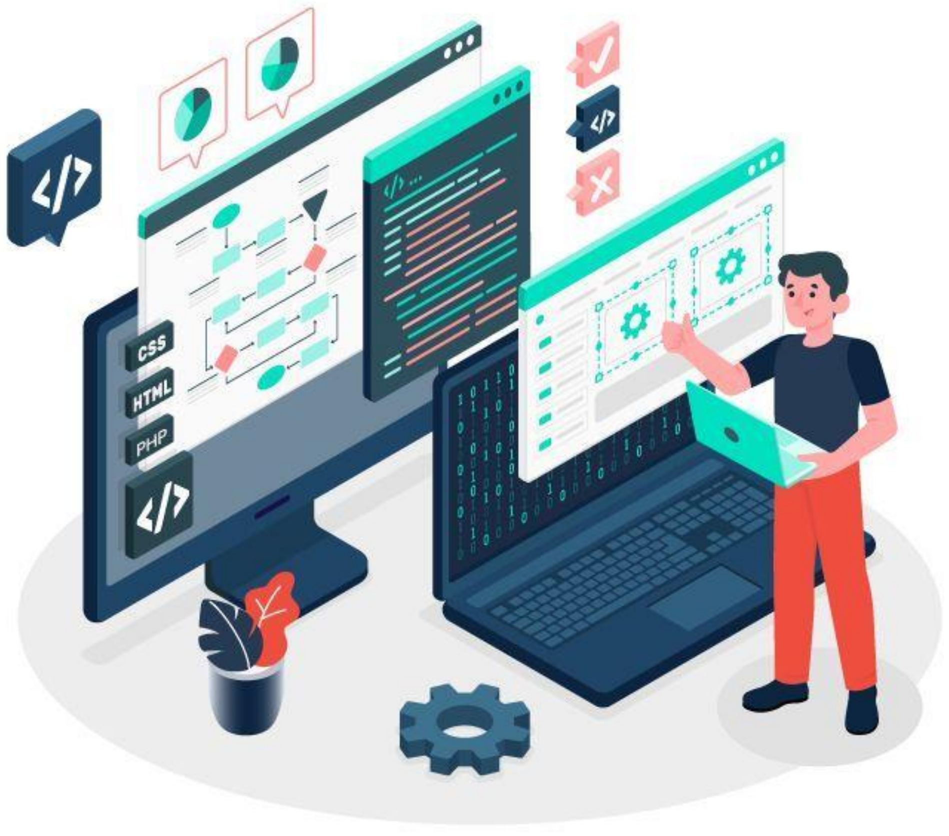


Image from [Pinterest]

Three I's of Data Visualization

1. Interactivity: The visualization should allow users to explore and interact with the data, such as filtering, zooming, or adjusting parameters to uncover deeper insights.

2. Information: The data should be presented clearly and accurately, allowing the viewer to understand the insights and story behind the data.

3. Interpretation: The visual should help the audience easily interpret and draw meaningful conclusions from the data, making complex information more accessible and actionable.

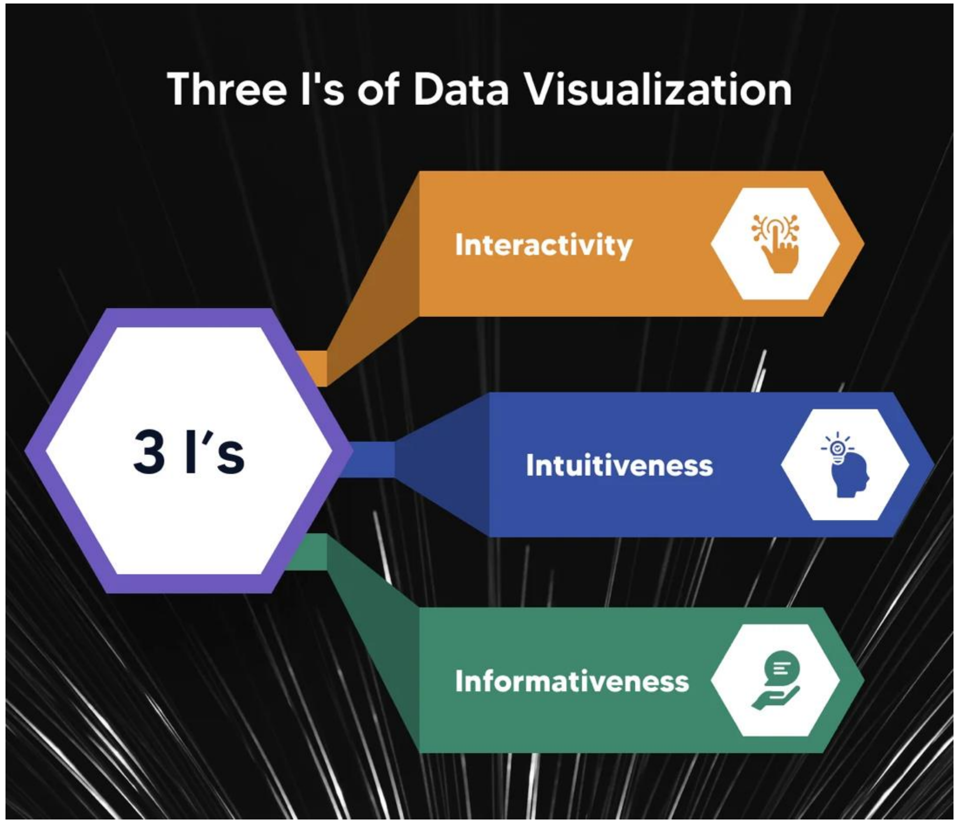
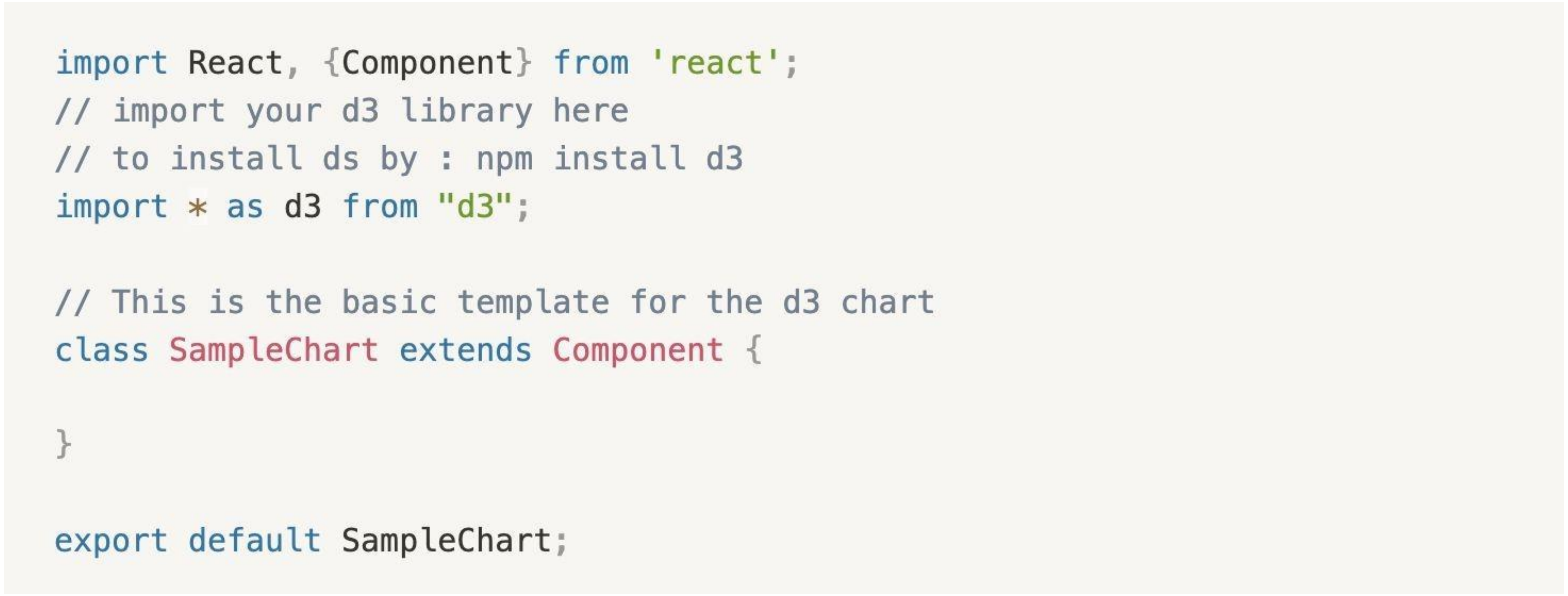


Image from [kellton]

Introduction of D3.js

Usage of D3.js

D3.js：Bar Chart **npm install d3**

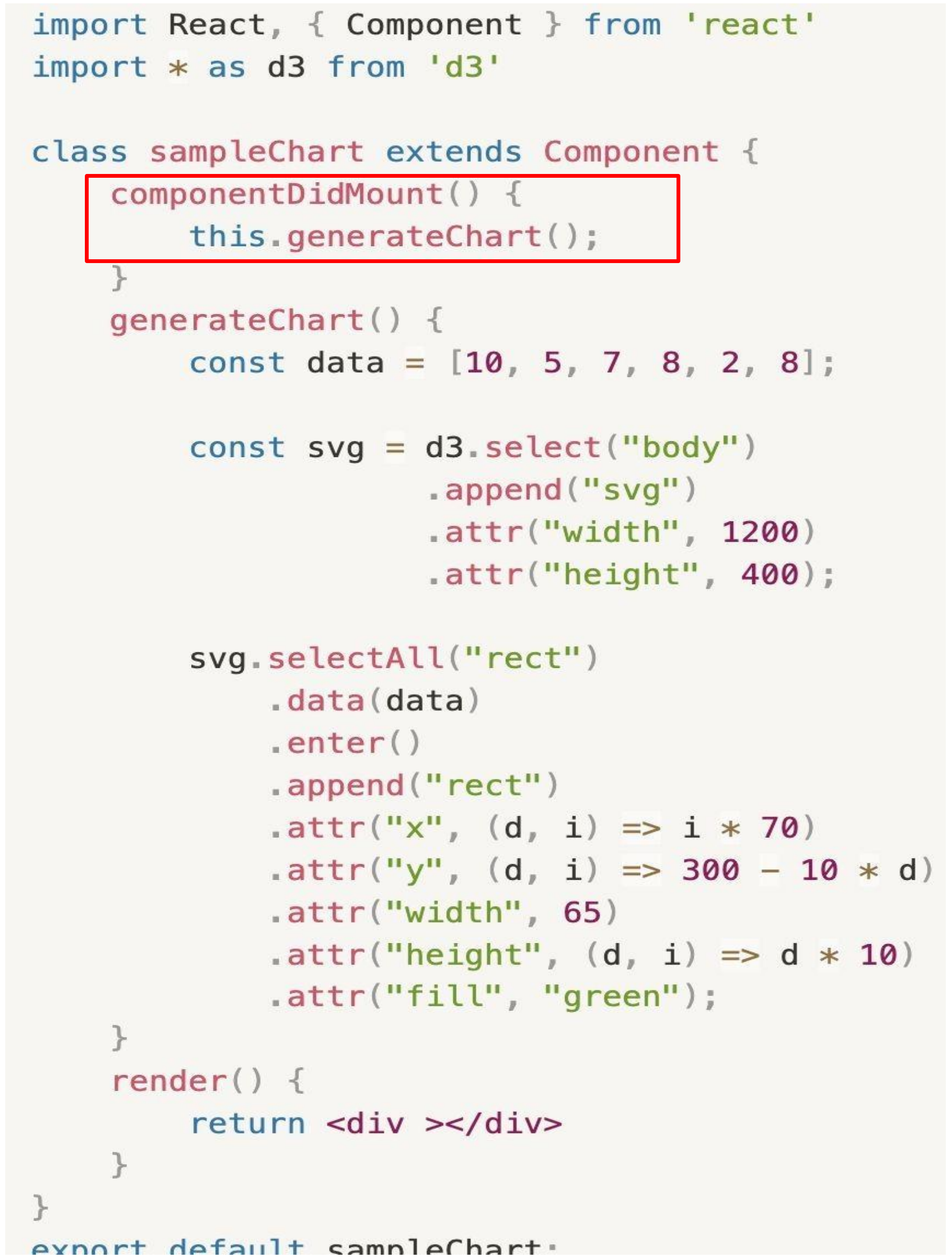


```
import React, {Component} from 'react';
// import your d3 library here
// to install ds by : npm install d3
import * as d3 from "d3";

// This is the basic template for the d3 chart
class SampleChart extends Component {

}

export default SampleChart;
```

Usage of D3.js

In order to display the bar chart when the sampleChart component is mounted to the DOM, we will utilize the ComponentDidMount lifecycle.

In D3.js, loading the visualization logic within the **ComponentDidMount** lifecycle method is a common practice in React applications.

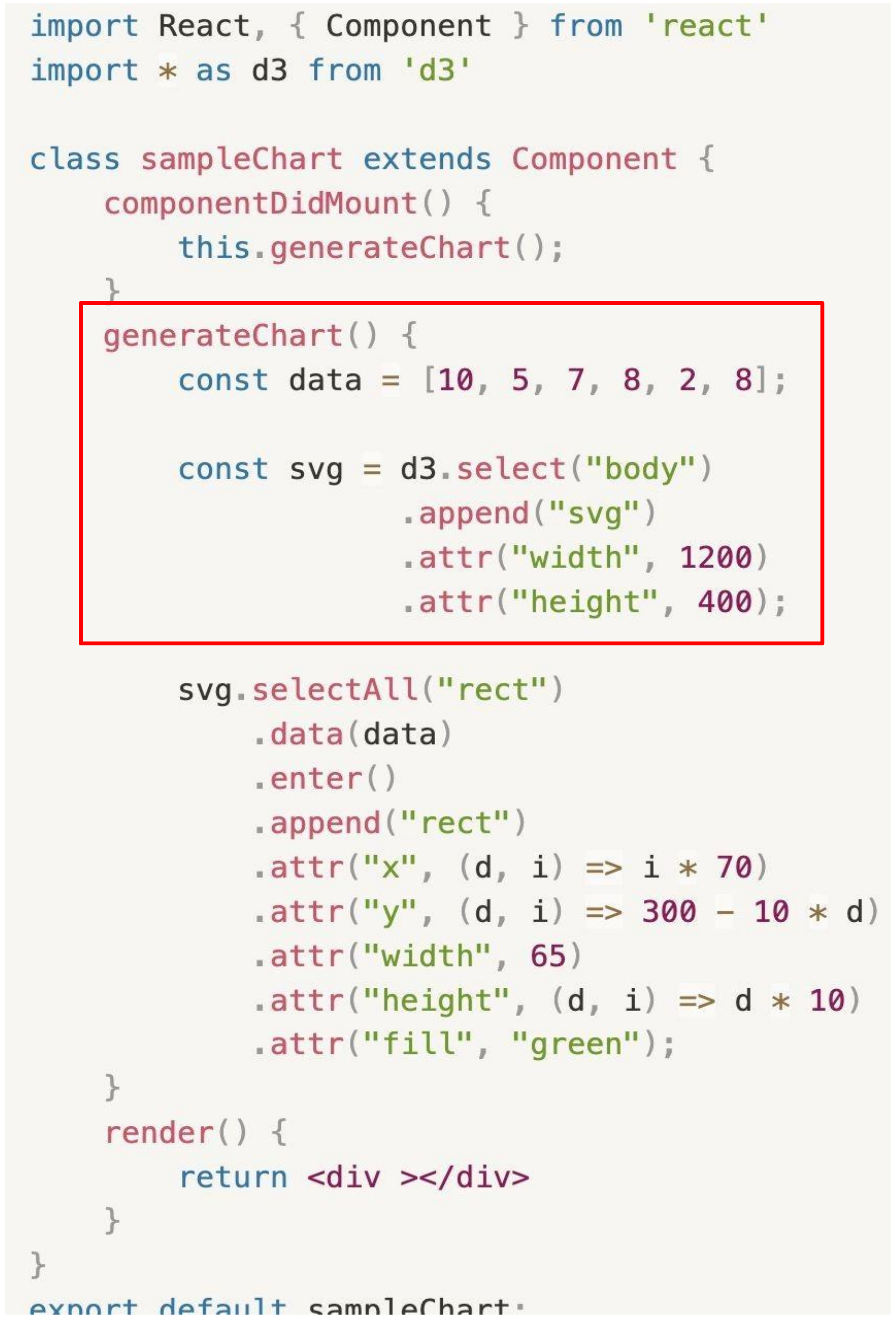


```
import React, { Component } from 'react'
import * as d3 from 'd3'

class sampleChart extends Component {
    componentDidMount() {
        this.generateChart();
    }
    generateChart() {
        const data = [10, 5, 7, 8, 2, 8];

        const svg = d3.select("body")
                      .append("svg")
                      .attr("width", 1200)
                      .attr("height", 400);

        svg.selectAll("rect")
            .data(data)
            .enter()
            .append("rect")
            .attr("x", (d, i) => i * 70)
            .attr("y", (d, i) => 300 - 10 * d)
            .attr("width", 65)
            .attr("height", (d, i) => d * 10)
            .attr("fill", "green");
    }
    render() {
        return <div ></div>
    }
}
export default sampleChart;
```

Usage of D3.js

generateChart() is the method we use to create D3.js charts. In React, this step is crucial as it ensures that the chart is only displayed when the component is mounted to the DOM.

Firstly, we define a parameter which contains the data for the chart we want to visualise.
Next, we define an image in **SVG** format using the D3.js method. SVG is used because it is scalable and the data will not appear pixelated regardless of how the screen size is scaled or broadened.

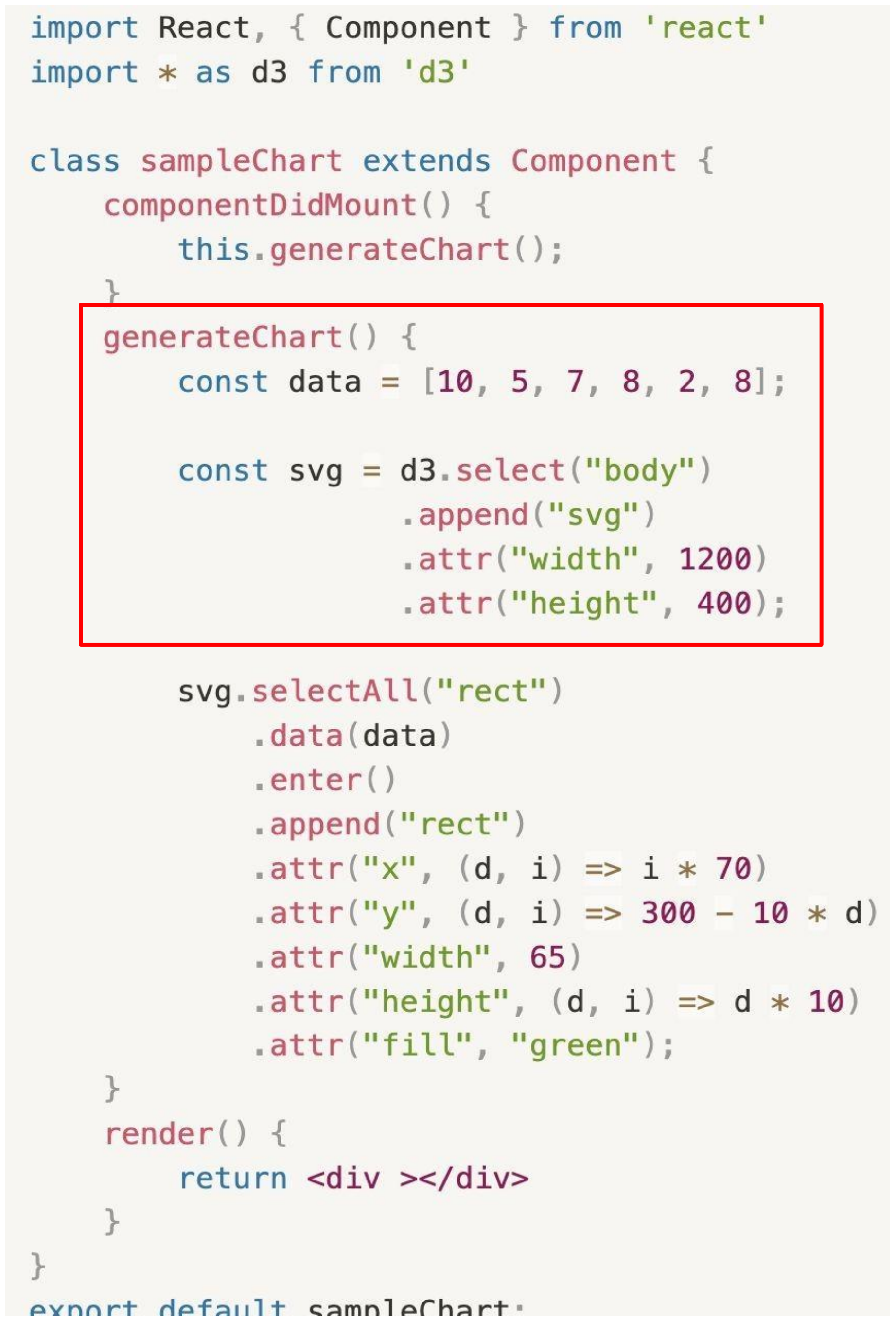


```
import React, { Component } from 'react'
import * as d3 from 'd3'

class sampleChart extends Component {
    componentDidMount() {
        this.generateChart();
    }
    generateChart() {
        const data = [10, 5, 7, 8, 2, 8];

        const svg = d3.select("body")
                    .append("svg")
                    .attr("width", 1200)
                    .attr("height", 400);

        svg.selectAll("rect")
            .data(data)
            .enter()
            .append("rect")
            .attr("x", (d, i) => i * 70)
            .attr("y", (d, i) => 300 - 10 * d)
            .attr("width", 65)
            .attr("height", (d, i) => d * 10)
            .attr("fill", "green");
    }
    render() {
        return <div ></div>
    }
}
export default sampleChart;
```

Usage of D3.js

The ***d3.select()*** method selects an HTML element. It selects the first element that matches the passed parameter and creates a node for it. Here we have passed the body element.

The ***attr()*** method is responsible for adding attributes to the element, which can be any attribute of the HTML element, such as class, height, width, etc.



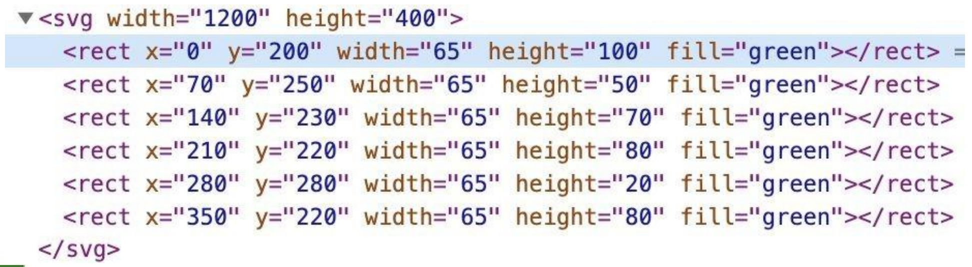
```
import React, { Component } from 'react'
import * as d3 from 'd3'

class sampleChart extends Component {
    componentDidMount() {
        this.generateChart();
    }
    generateChart() {
        const data = [10, 5, 7, 8, 2, 8];

        const svg = d3.select("body")
                    .append("svg")
                    .attr("width", 1200)
                    .attr("height", 400);

        svg.selectAll("rect")
            .data(data)
            .enter()
            .append("rect")
            .attr("x", (d, i) => i * 70)
            .attr("y", (d, i) => 300 - 10 * d)
            .attr("width", 65)
            .attr("height", (d, i) => d * 10)
            .attr("fill", "green");
    }
    render() {
        return <div ></div>
    }
}
export default sampleChart;
```

Usage of D3.js

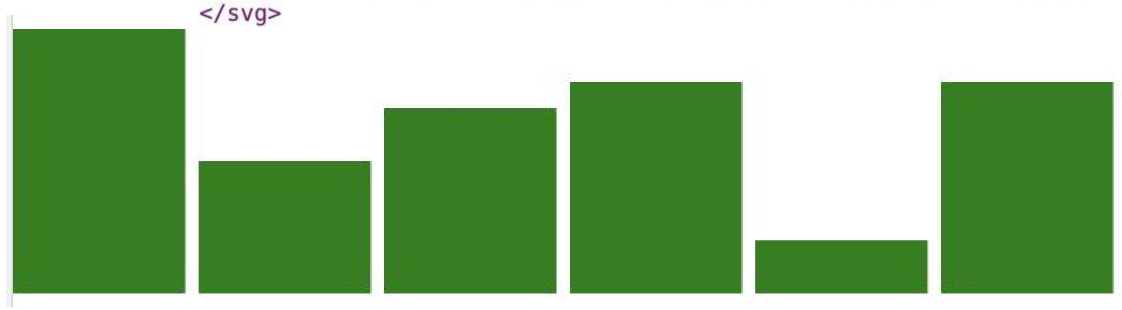
.selectAll("rect") selects all rectangular elements in the SVG.

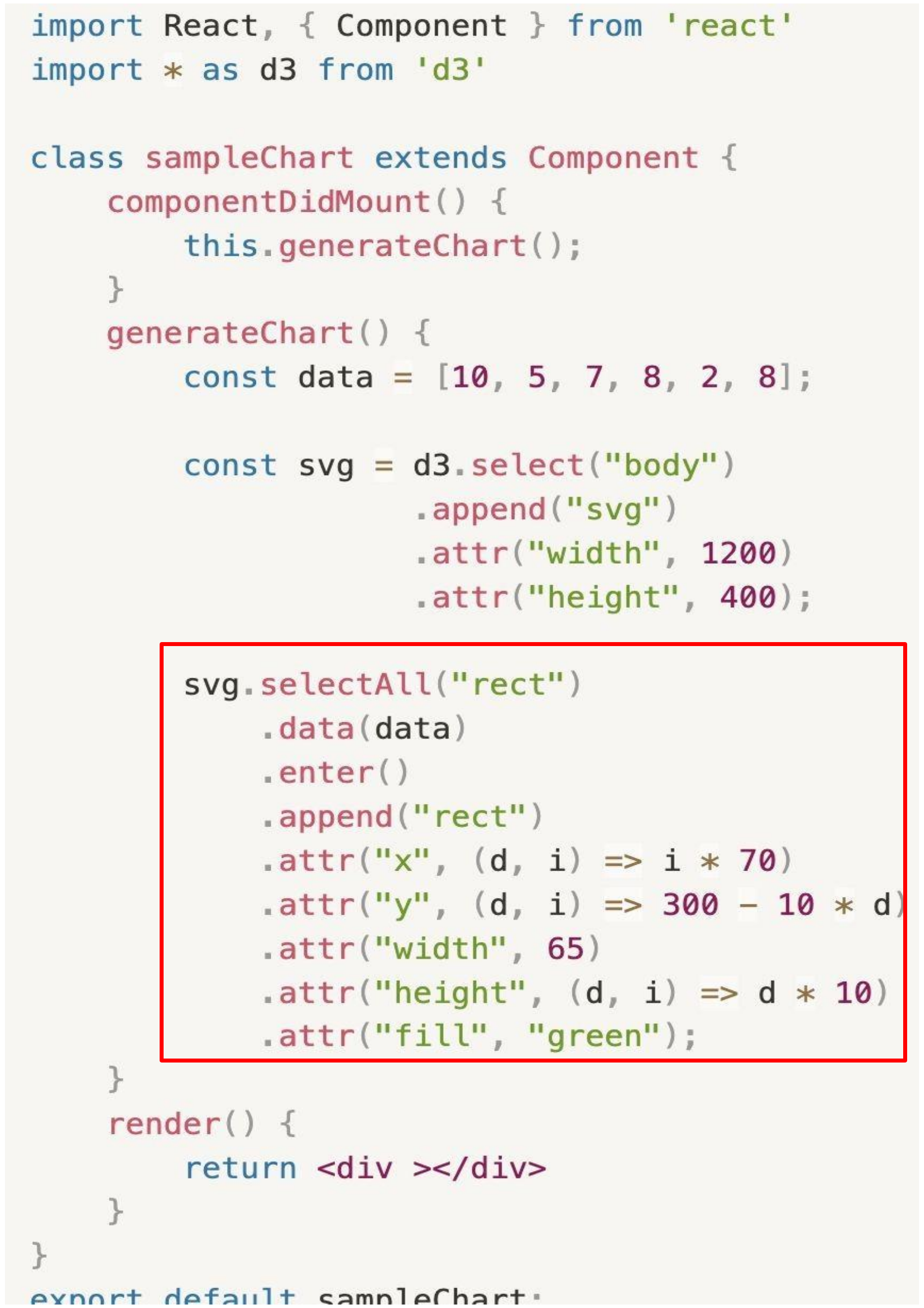
.data(data) binds the data to the selected rectangle element.

.attr("x", (d, i) => i * 70): here the x-coordinate of the rectangles is set, and the x-coordinate interval of each rectangle is set to 70, according to the index i of the data in the array



```
▼<svg width="1200" height="400">
    <rect x="0" y="200" width="65" height="100" fill="green"></rect> =
    <rect x="70" y="250" width="65" height="50" fill="green"></rect>
    <rect x="140" y="230" width="65" height="70" fill="green"></rect>
    <rect x="210" y="220" width="65" height="80" fill="green"></rect>
    <rect x="280" y="280" width="65" height="20" fill="green"></rect>
    <rect x="350" y="220" width="65" height="80" fill="green"></rect>
  </svg>
```





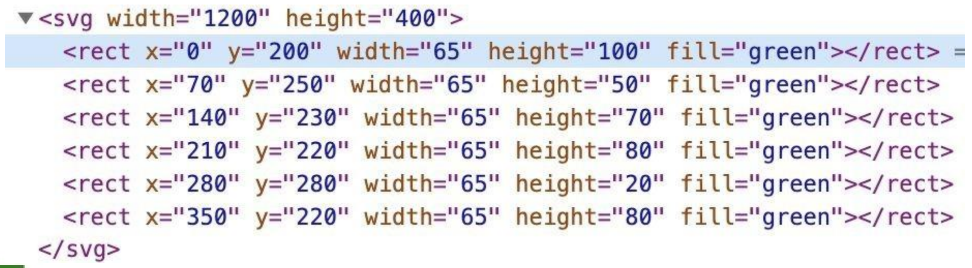
```
import React, { Component } from 'react'
import * as d3 from 'd3'

class sampleChart extends Component {
    componentDidMount() {
        this.generateChart();
    }
    generateChart() {
        const data = [10, 5, 7, 8, 2, 8];

        const svg = d3.select("body")
                      .append("svg")
                      .attr("width", 1200)
                      .attr("height", 400);

        svg.selectAll("rect")
            .data(data)
            .enter()
            .append("rect")
            .attr("x", (d, i) => i * 70)
            .attr("y", (d, i) => 300 - 10 * d)
            .attr("width", 65)
            .attr("height", (d, i) => d * 10)
            .attr("fill", "green");
    }
    render() {
        return <div ></div>
    }
}
export default sampleChart;
```

Usage of D3.js

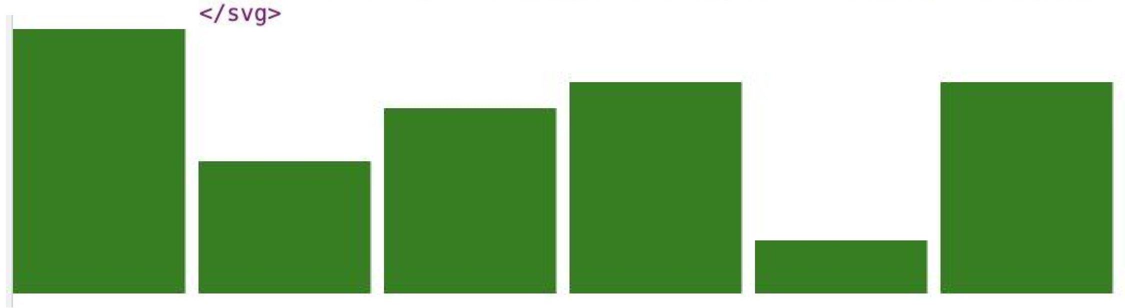
.attr("width", 65): sets the width of the rectangle to a fixed value 65

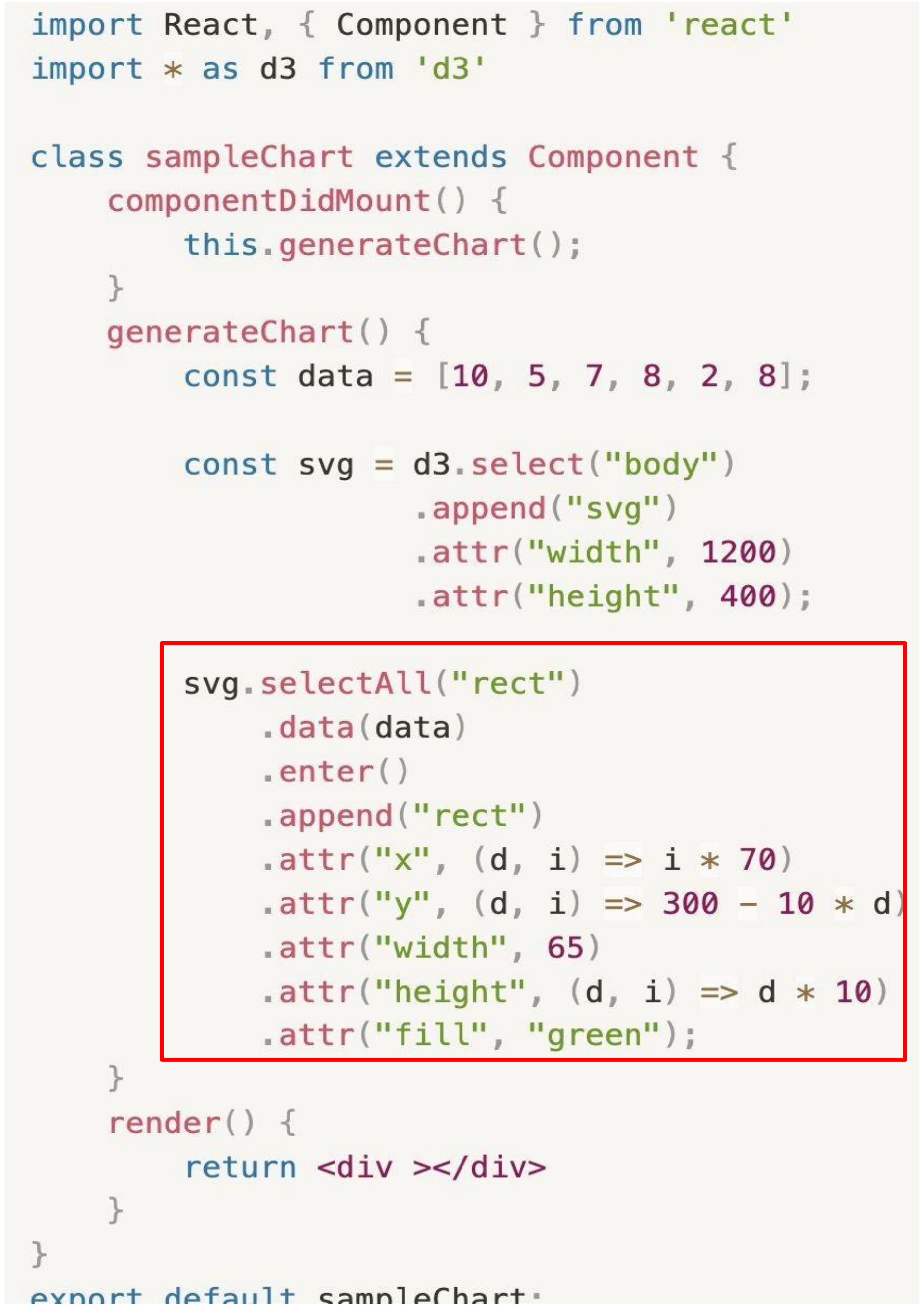
.attr("height", (d, i) => d * 10): set the height of the rectangle, according to the data d, set the height of the rectangle to 10 times the data value d

.attr("fill", "green"): set the fill colour of the rectangle to green



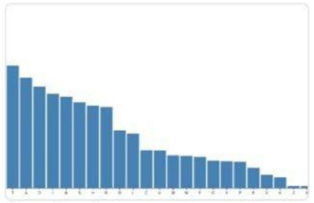
```
▼<svg width="1200" height="400">
    <rect x="0" y="200" width="65" height="100" fill="green"></rect> =
    <rect x="70" y="250" width="65" height="50" fill="green"></rect>
    <rect x="140" y="230" width="65" height="70" fill="green"></rect>
    <rect x="210" y="220" width="65" height="80" fill="green"></rect>
    <rect x="280" y="280" width="65" height="20" fill="green"></rect>
    <rect x="350" y="220" width="65" height="80" fill="green"></rect>
  </svg>
```

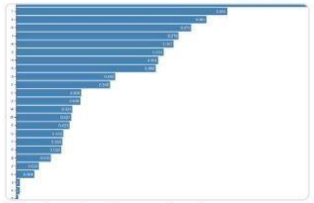


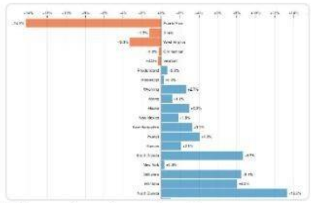


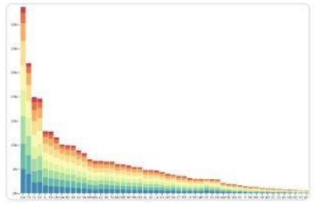
```
import React, { Component } from 'react'
import * as d3 from 'd3'

class sampleChart extends Component {
    componentDidMount() {
        this.generateChart();
    }
    generateChart() {
        const data = [10, 5, 7, 8, 2, 8];

        const svg = d3.select("body")
                    .append("svg")
                    .attr("width", 1200)
                    .attr("height", 400);

        svg.selectAll("rect")
            .data(data)
            .enter()
            .append("rect")
            .attr("x", (d, i) => i * 70)
            .attr("y", (d, i) => 300 - 10 * d)
            .attr("width", 65)
            .attr("height", (d, i) => d * 10)
            .attr("fill", "green");
    }
    render() {
        return <div ></div>
    }
}
export default sampleChart;
```

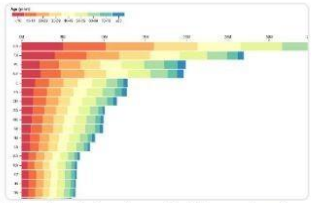
More Charts of D3.js

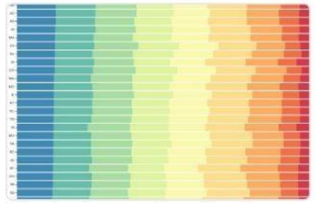

Bar chart

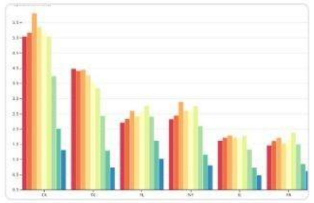

Horizontal bar chart

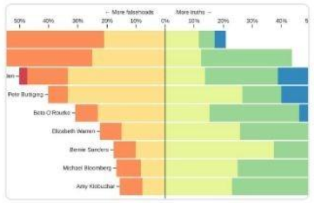

Diverging bar chart

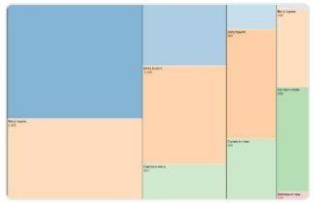

Stacked bar chart

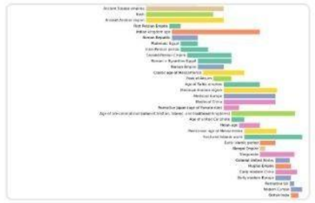

Stacked horizontal bar chart

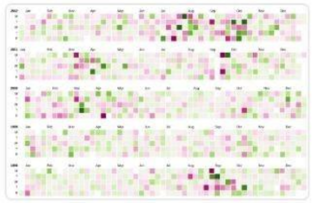

Stacked normalized horizon...

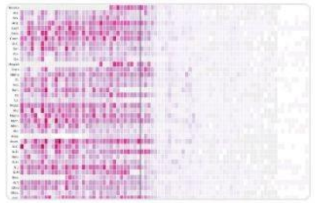

Grouped bar chart

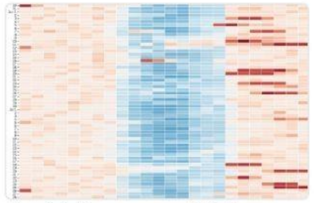

Diverging stacked bar chart

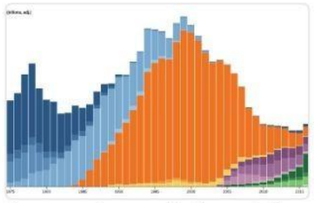

Marimekko chart

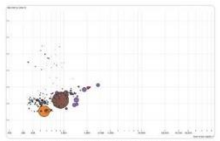

World history timeline

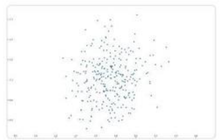

Calendar

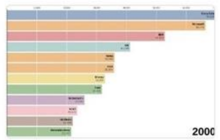

The impact of vaccines


Electricity usage, 2019


Revenue by music format, 1...


The wealth & health of natio...


Scatterplot tour

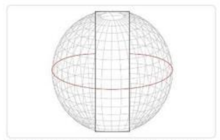

Bar chart race

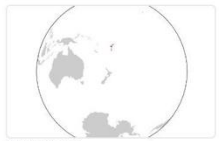
Treemap

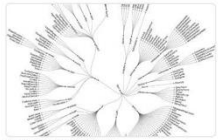
Cascaded treemap

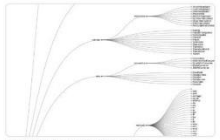
Nested treemap

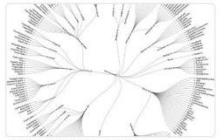
Zoom to bounding box

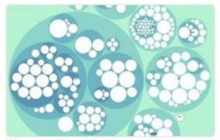

Orthographic to equirectang...

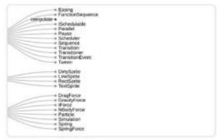

World tour


Radial tidy tree

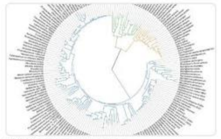

Cluster dendrogram


Radial dendrogram


Zoomable circle packing


Collapsible tree

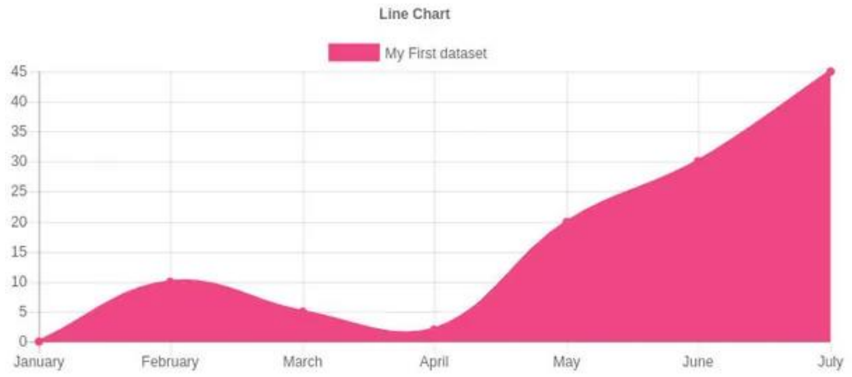
Zoomable icicle

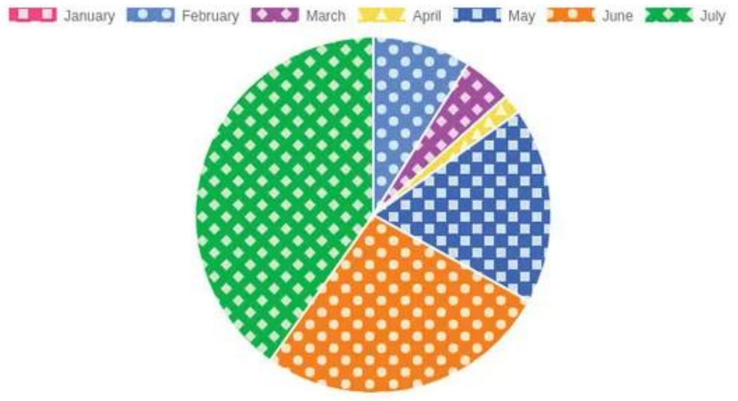

Phylogenetic tree

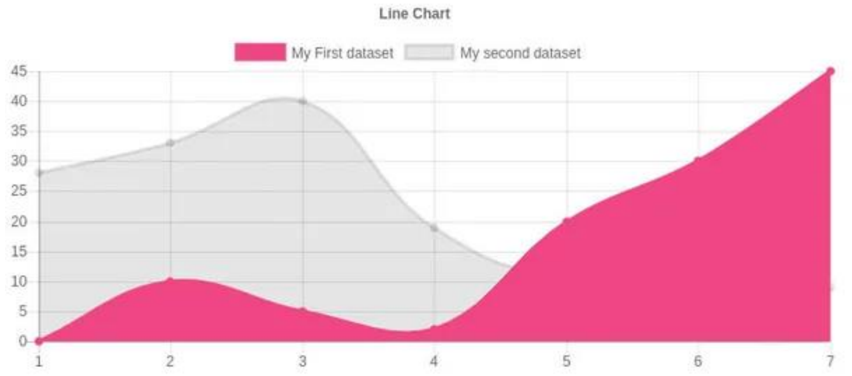
Force-directed tree

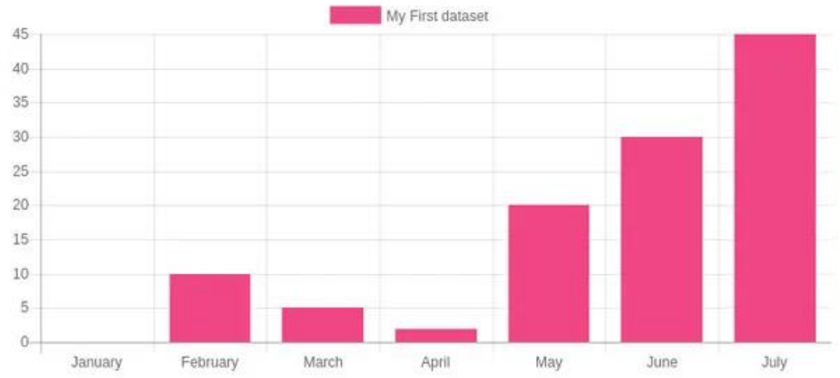
Introduction of Chart.js

Chart.js provides a set of frequently used chart types, plugins, and customization options. In addition to a reasonable set of built-in chart types

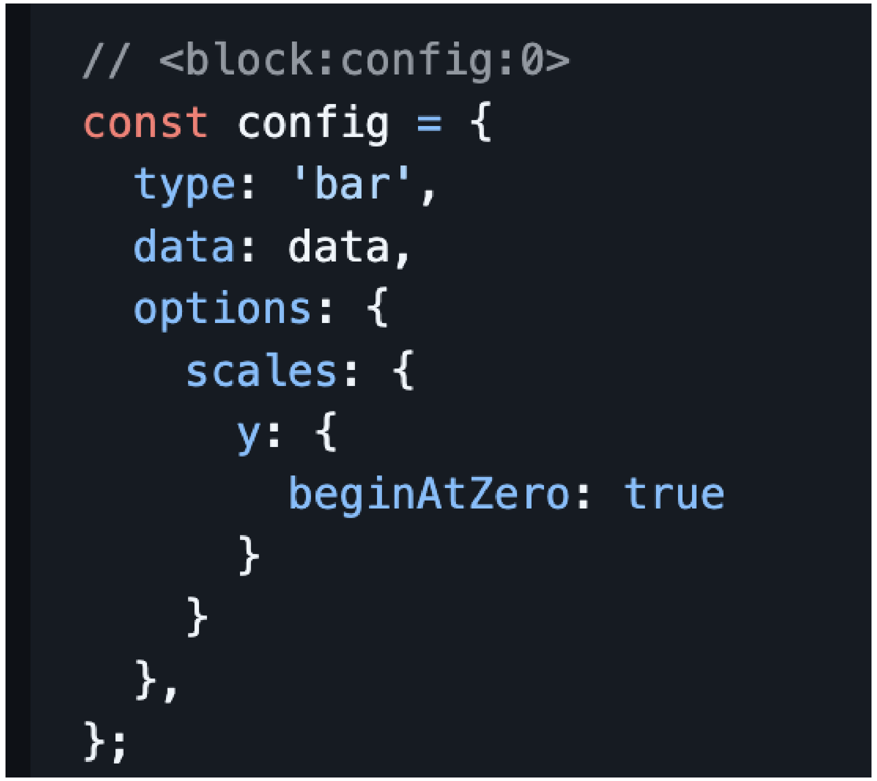




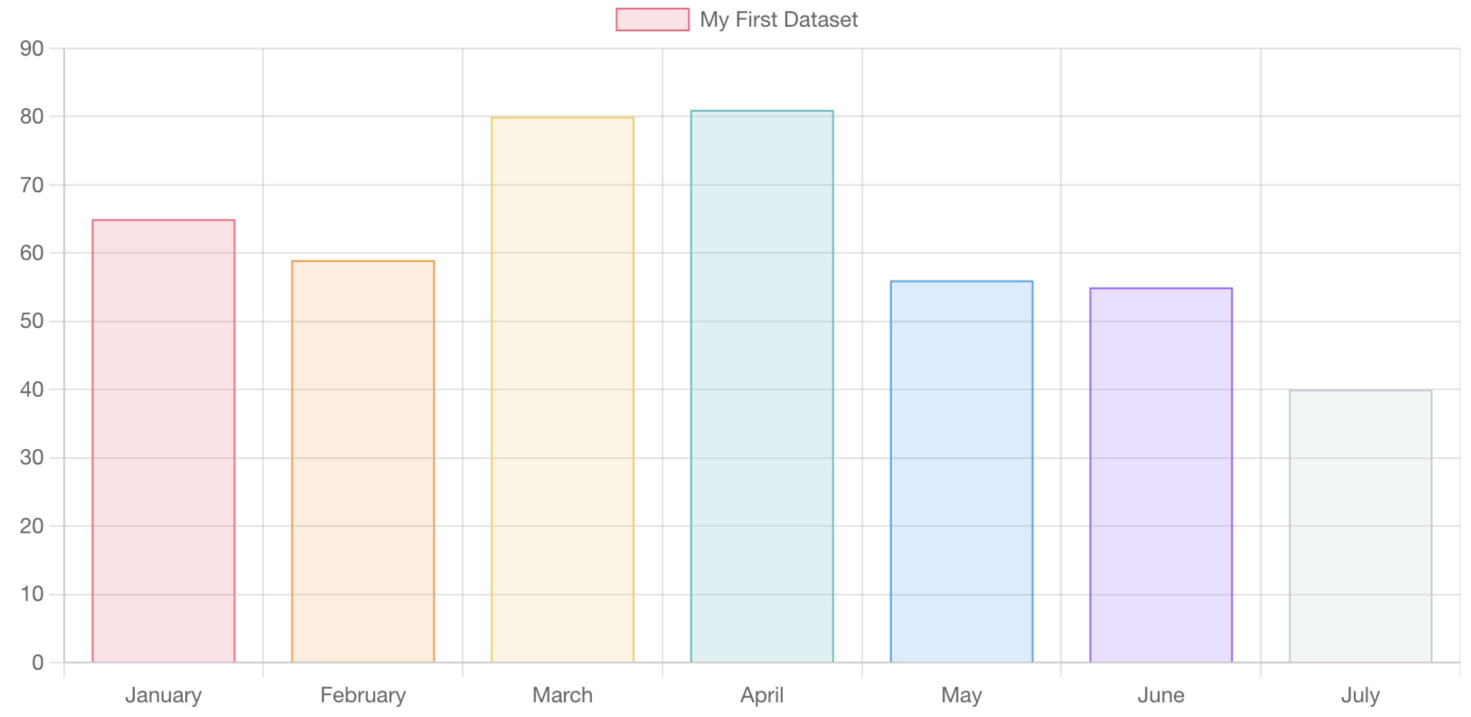




Usage of Chart.js



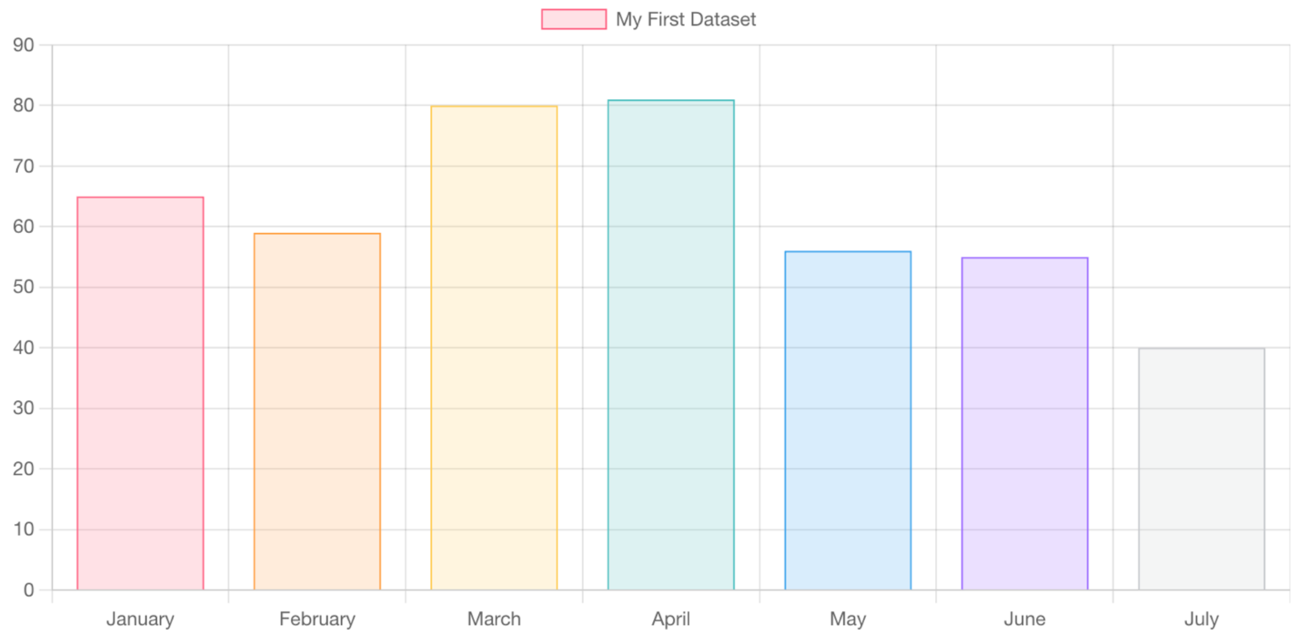
```
// <block:config:0>
const config = {
  type: 'bar',
  data: data,
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  },
};
```

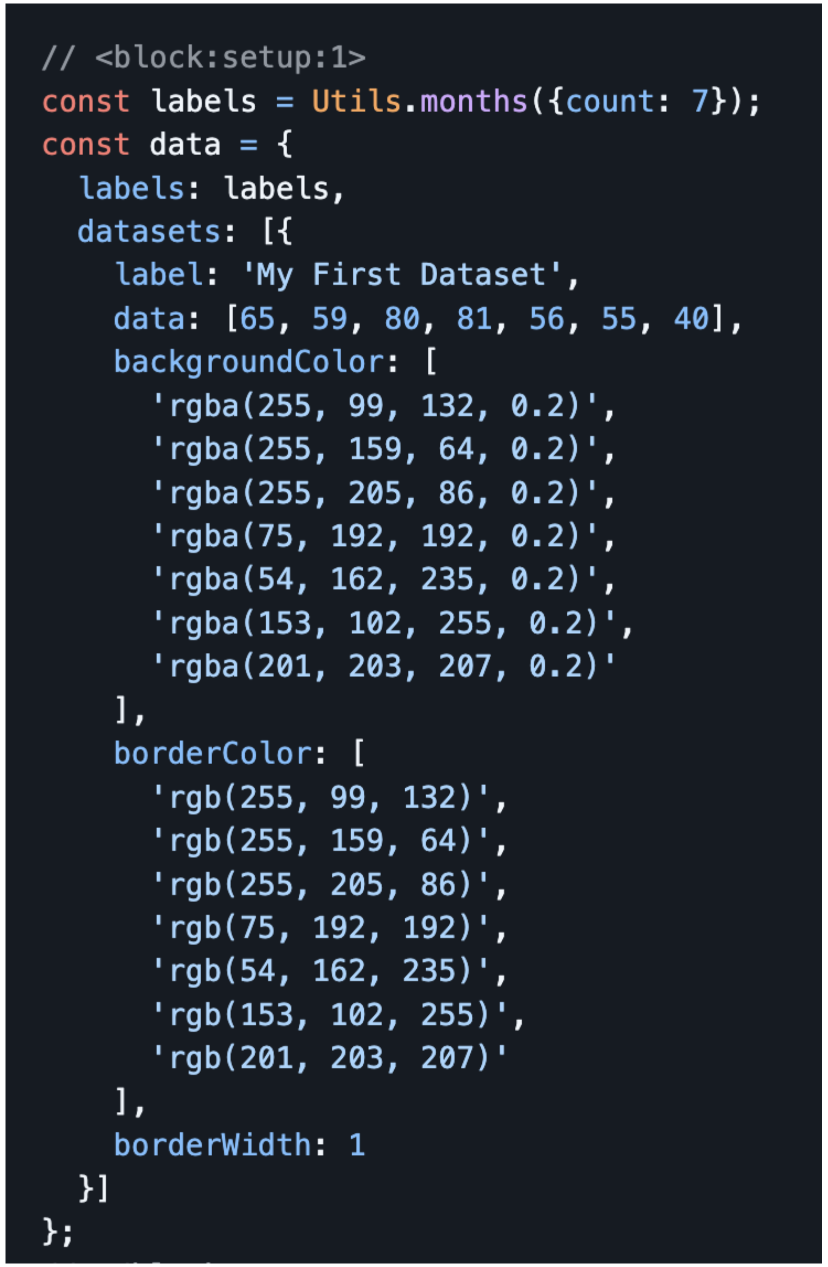


Usage of Chart.js

Const data = { … }:

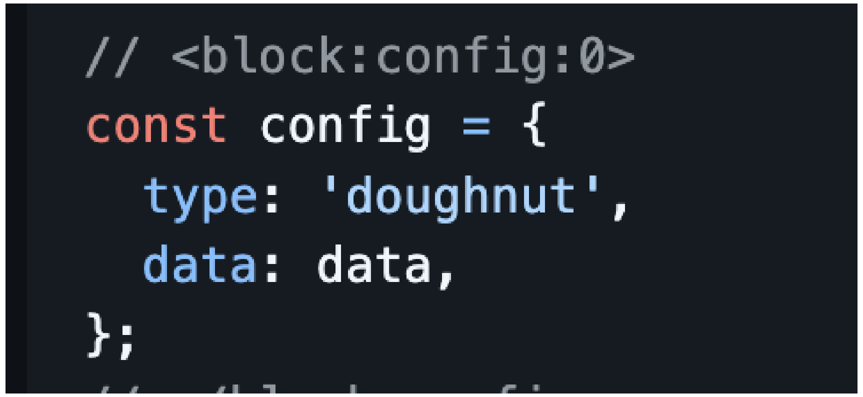
data: [65,59,80,81,56,55,40]:





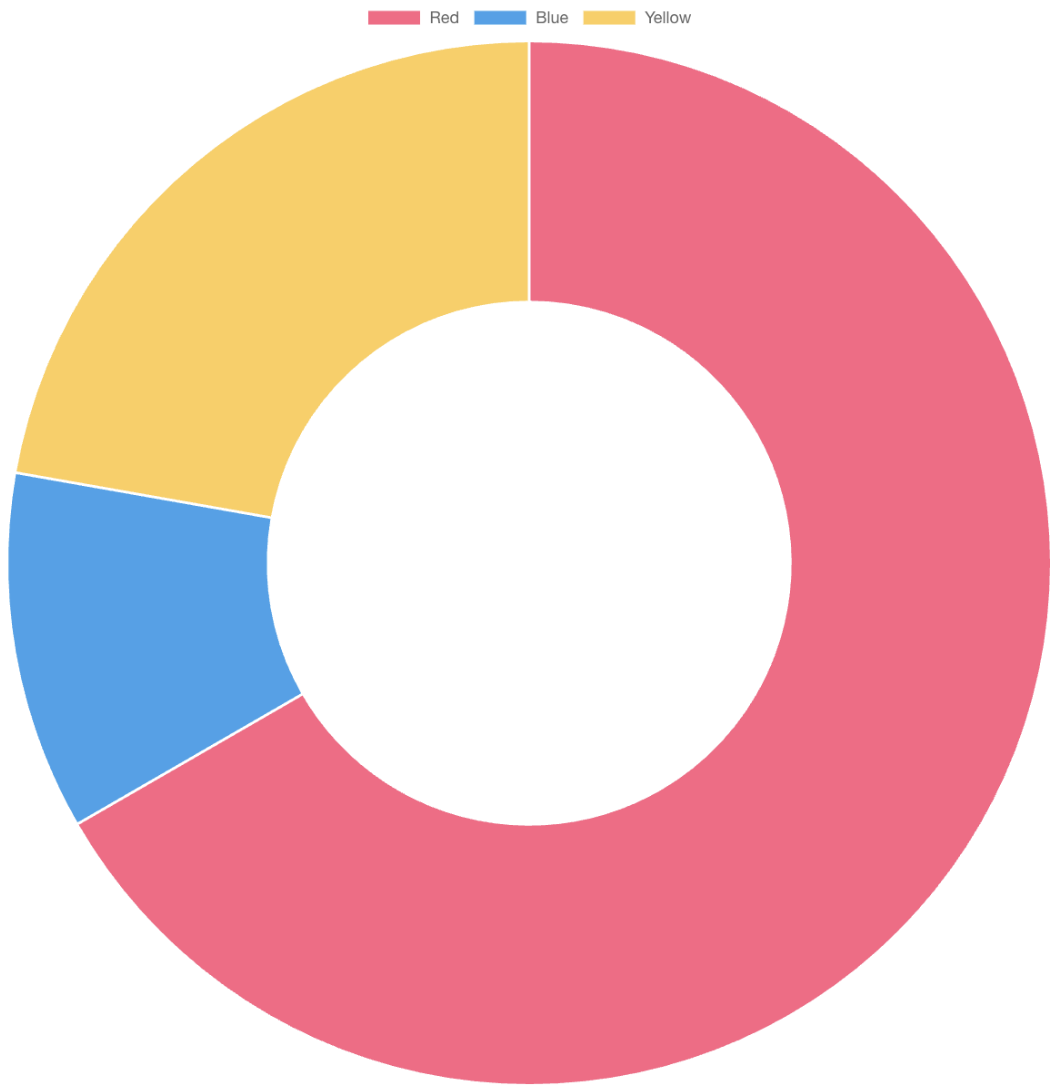
```
// <block:setup:1>
const labels = Utils.months({count: 7});
const data = {
  labels: labels,
  datasets: [{
    label: 'My First Dataset',
    data: [65, 59, 80, 81, 56, 55, 40],
    backgroundColor: [
      'rgba(255, 99, 132, 0.2)',
      'rgba(255, 159, 64, 0.2)',
      'rgba(255, 205, 86, 0.2)',
      'rgba(75, 192, 192, 0.2)',
      'rgba(54, 162, 235, 0.2)',
      'rgba(153, 102, 255, 0.2)',
      'rgba(201, 203, 207, 0.2)'
    ],
    borderColor: [
      'rgb(255, 99, 132)',
      'rgb(255, 159, 64)',
      'rgb(255, 205, 86)',
      'rgb(75, 192, 192)',
      'rgb(54, 162, 235)',
      'rgb(153, 102, 255)',
      'rgb(201, 203, 207)'
    ],
    borderWidth: 1
  }]
};
```

Usage of Chart.js

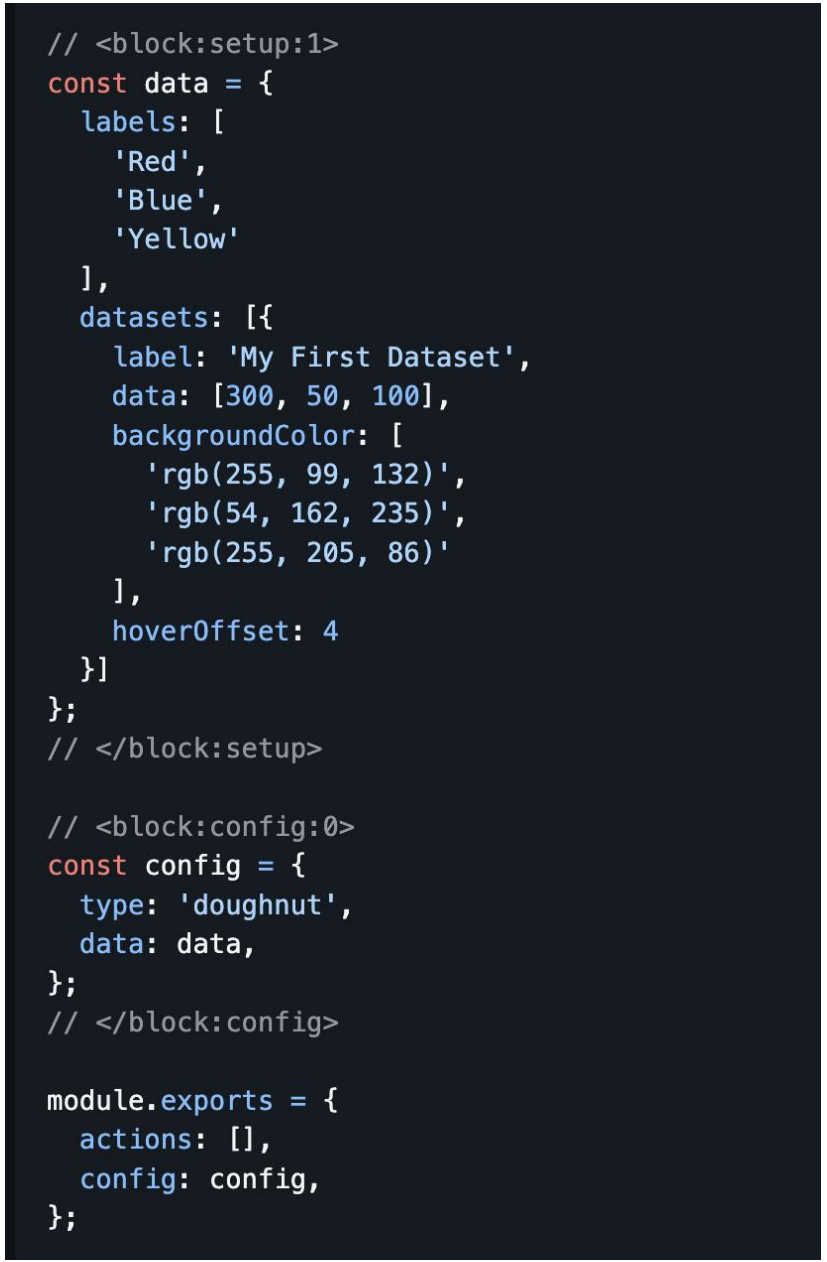


```
// <block:config:0>
const config = {
  type: 'doughnut',
  data: data,
};
```

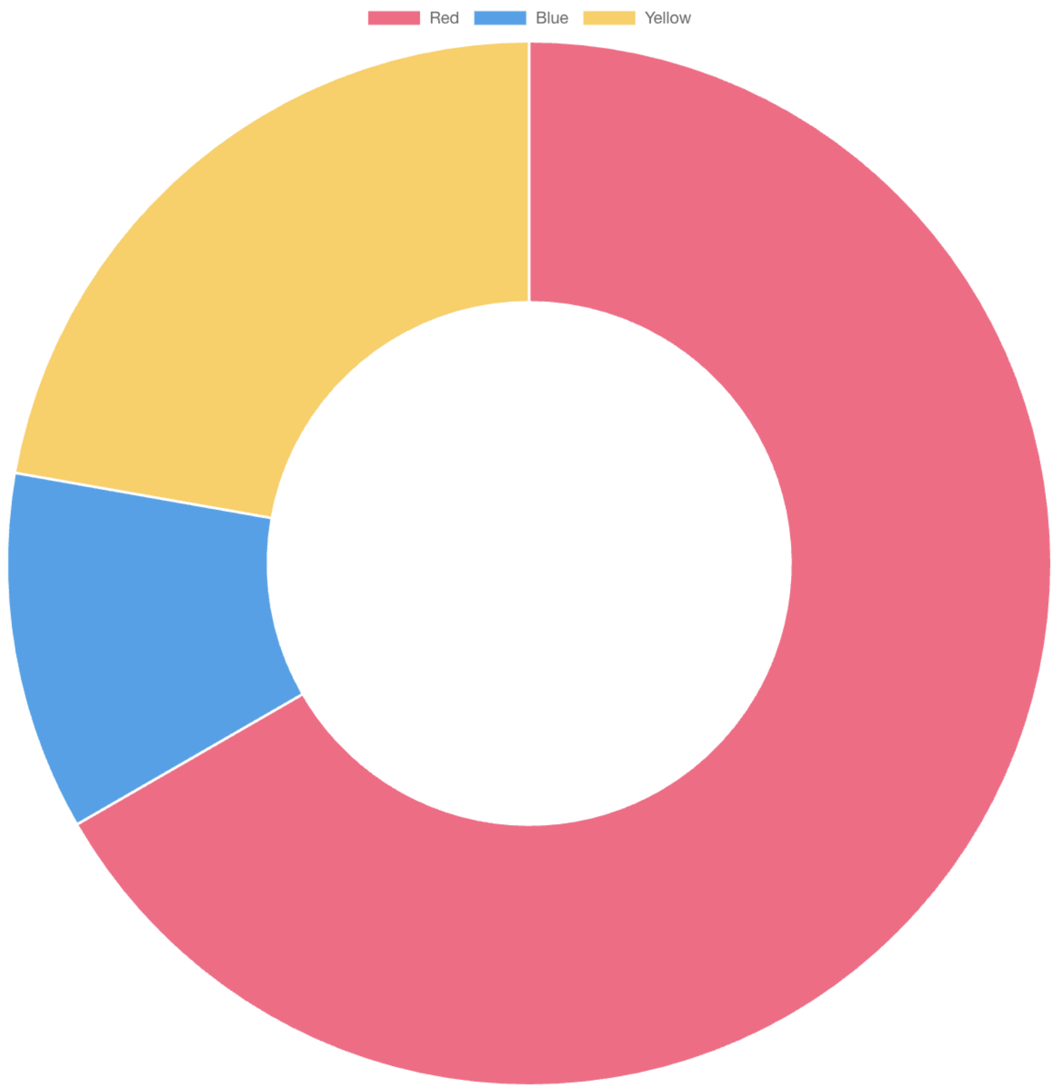
Config object specifies the type of chart you want to create



Usage of Chart.js



```
// <block:setup:1>
const data = {
  labels: [
    'Red',
    'Blue',
    'Yellow'
  ],
  datasets: [{
    label: 'My First Dataset',
    data: [300, 50, 100],
    backgroundColor: [
      'rgb(255, 99, 132)',
      'rgb(54, 162, 235)',
      'rgb(255, 205, 86)'
    ],
    hoverOffset: 4
  }]
};
// </block:setup>

// <block:config:0>
const config = {
  type: 'doughnut',
  data: data,
};
// </block:config>

module.exports = {
  actions: [],
  config: config,
};
```



- Recap Front-end & Back-end
- React API Call
- Data Visualization
- **Full-stack integration**
- Debugging and Troubleshooting
- Best Practices in Code Development

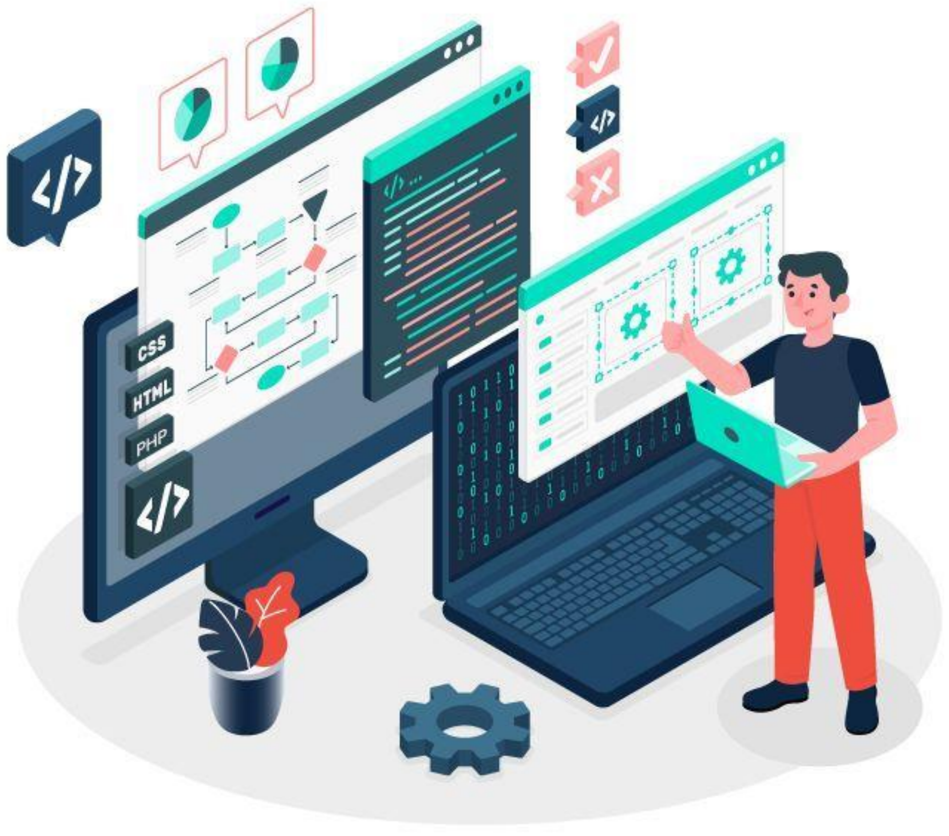
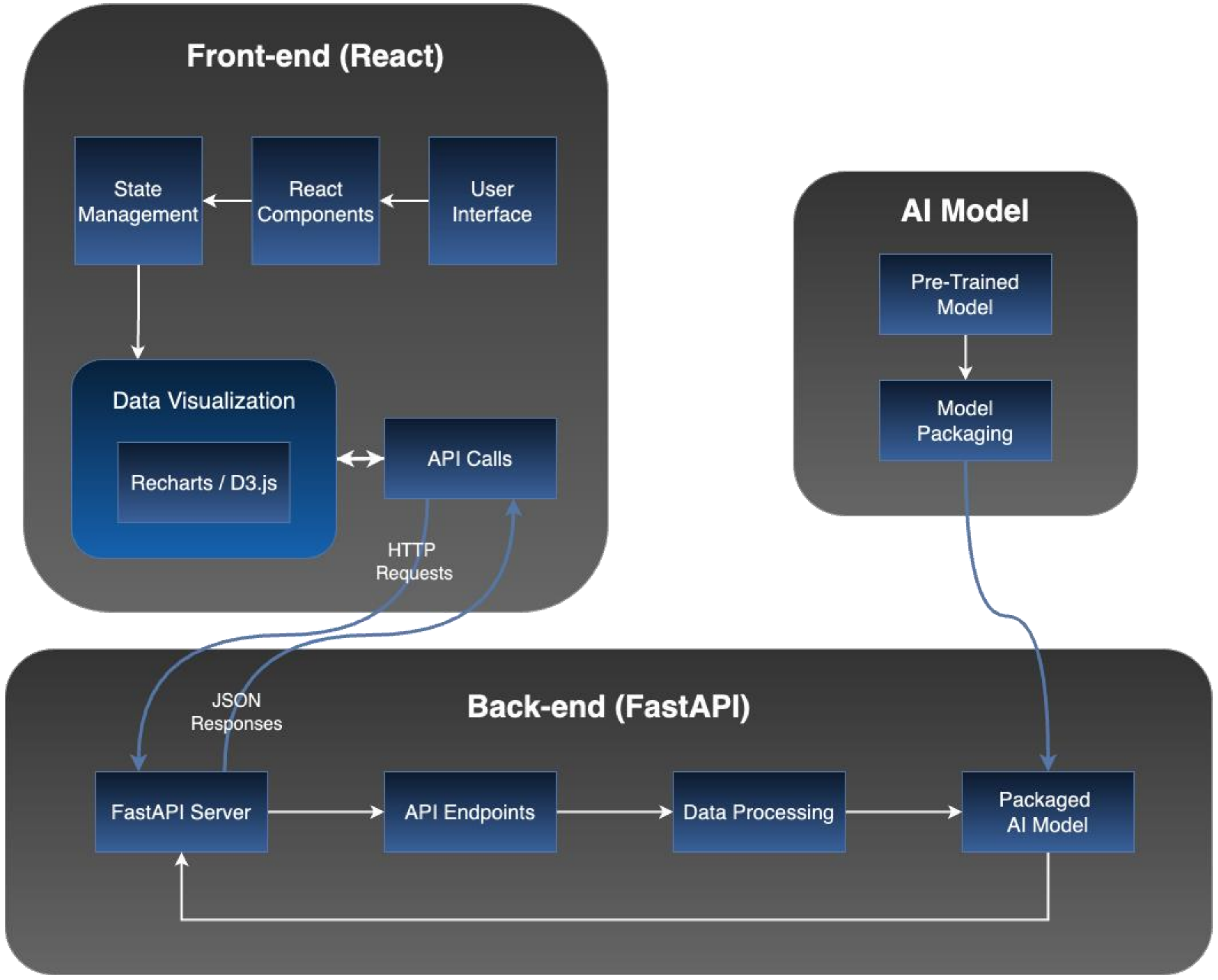


Image from [Pinterest]

Architecture Diagram



Architecture Overview

Full-stack Architecture with AI Integration

- React (Frontend): Handles UI, user interactions, and data visualization
- FastAPI (Backend): Manages API endpoints, data processing, and AI model integration
- Packaged AI Model: Pre-trained model integrated into the backend for real-time predictions or analysis

Communication Flow

- User interacts with React frontend
- React sends HTTP requests to FastAPI backend
- FastAPI processes requests, interacts with the AI model
- Backend sends responses back to React
- React updates UI and visualizes data based on responses

AI Model Integration

- AI model is packaged as a Python module
- Backend imports and utilizes the AI model for data processing or predictions
- Results are sent back to the frontend for visualization

Setting Up the Development Environment

React Setup

- Create React App: `npx create-react-app my-app`
- Key dependencies: `axios` for API calls

FastAPI Setup

- Virtual environment | Install FastAPI
- CORS middleware for allowing cross-origin requests

AI Model Integration

- Package pre-trained AI model as a Python module
- Install the AI model package in the FastAPI environment

Development Servers on Local Host

- React: `npm start` (usually on localhost:3000)
- FastAPI: `uvicorn main:app --reload` (usually on localhost:8000)

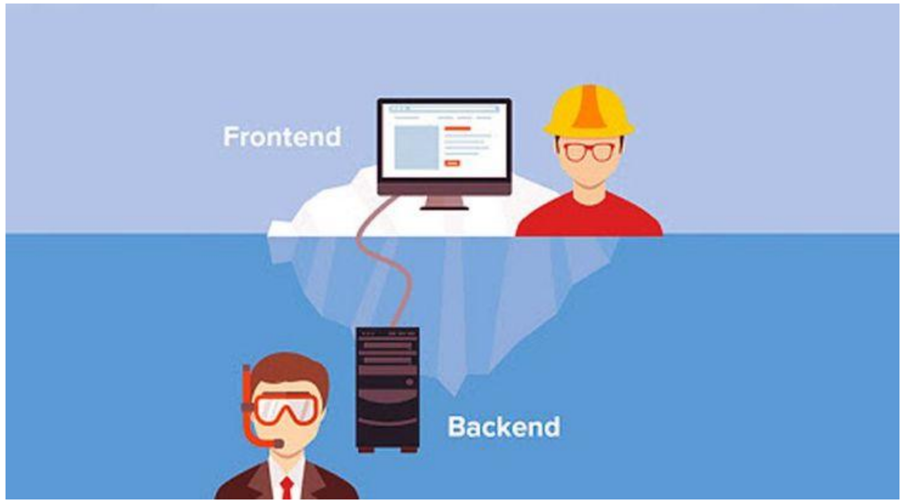


Image from [Pinterest]

Method 1 - Packaging AI Models for Integration

Creating a Python Package

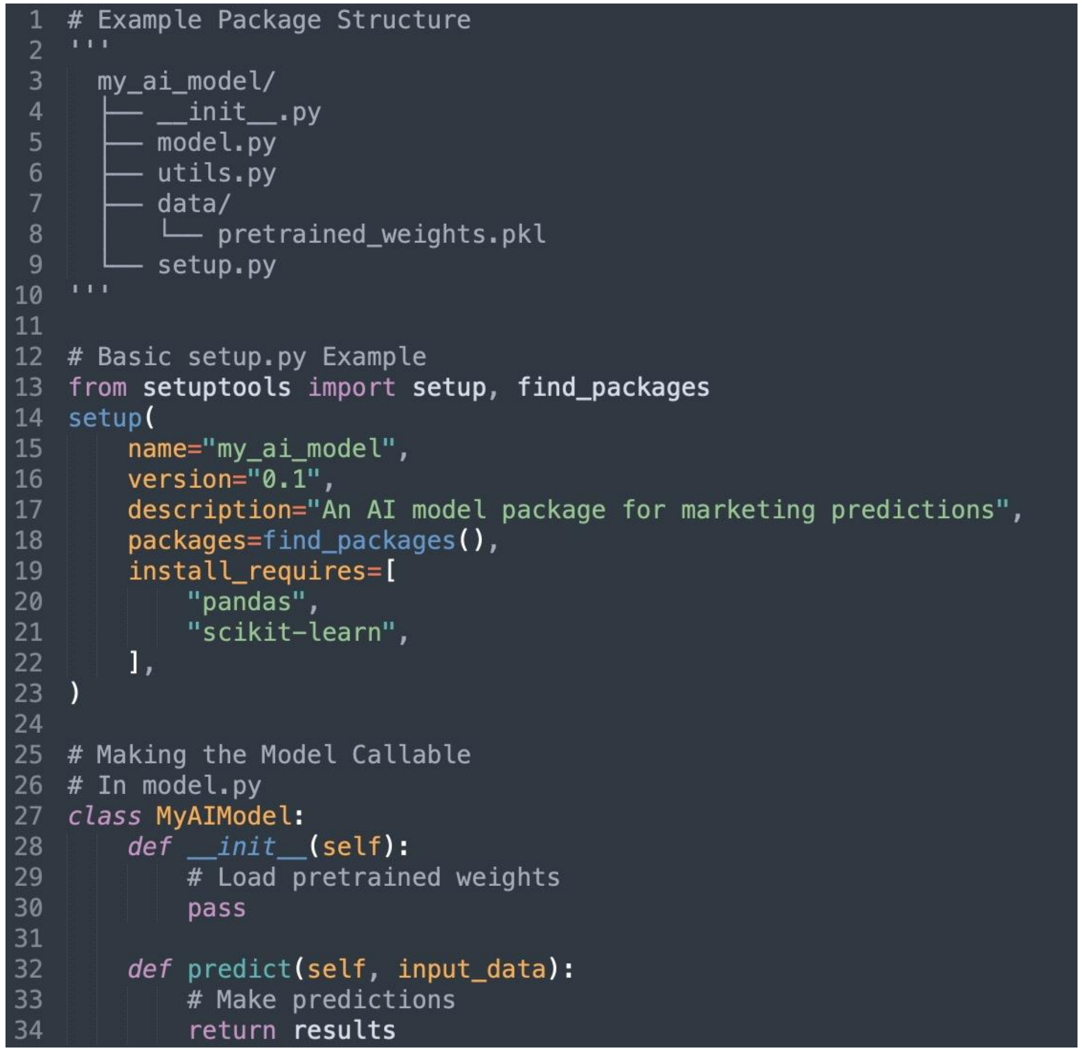
- Structure the AI model code
- Create `setup.py` file
- Use `__init__.py` to define public interface

Example Package Structure

Basic setup.py Example

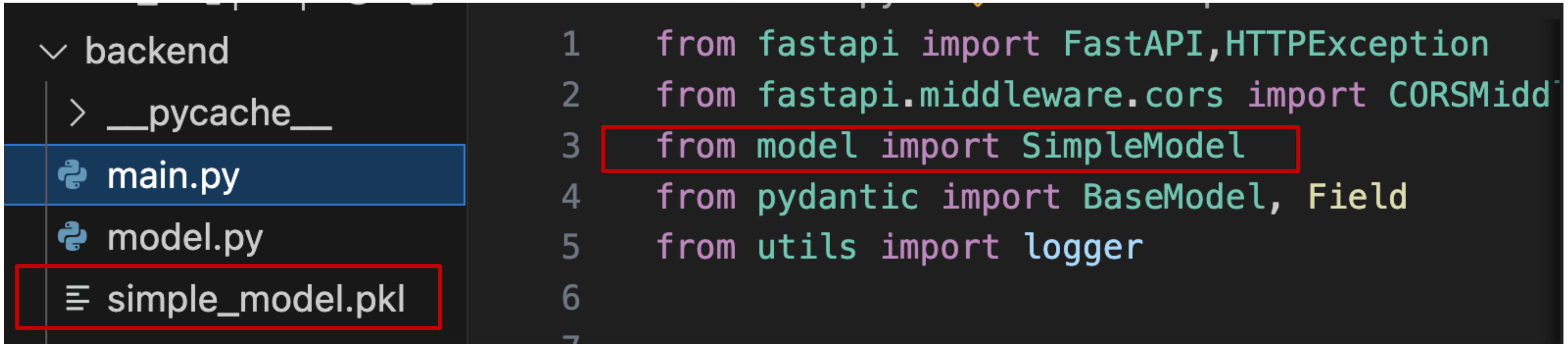
- Defining Metadata
- Package Discovery
- Specifying Dependencies

Making the Model Callable



```
# Example Package Structure
'''
  my_ai_model/
  ├── __init__.py
  ├── model.py
  ├── utils.py
  ├── data/
  │   └── pretrained_weights.pkl
  └── setup.py
'''

# Basic setup.py Example
from setuptools import setup, find_packages
setup(
    name="my_ai_model",
    version="0.1",
    description="An AI model package for marketing predictions",
    packages=find_packages(),
    install_requires=[
        "pandas",
        "scikit-learn",
    ],
)

# Making the Model Callable
# In model.py
class MyAIModel:
    def __init__(self):
        # Load pretrained weights
        pass

    def predict(self, input_data):
        # Make predictions
        return results
```

Method 2 – AI Models Integration by Joblib



```
from fastapi import FastAPI,HTTPException
from fastapi.middleware.cors import CORSMidd
from model import SimpleModel
from pydantic import BaseModel, Field
from utils import logger

```

Simpler joblib Integration:

- Use this for quick integrations, smaller projects, and when you don’t need to distribute the model as a Python package.

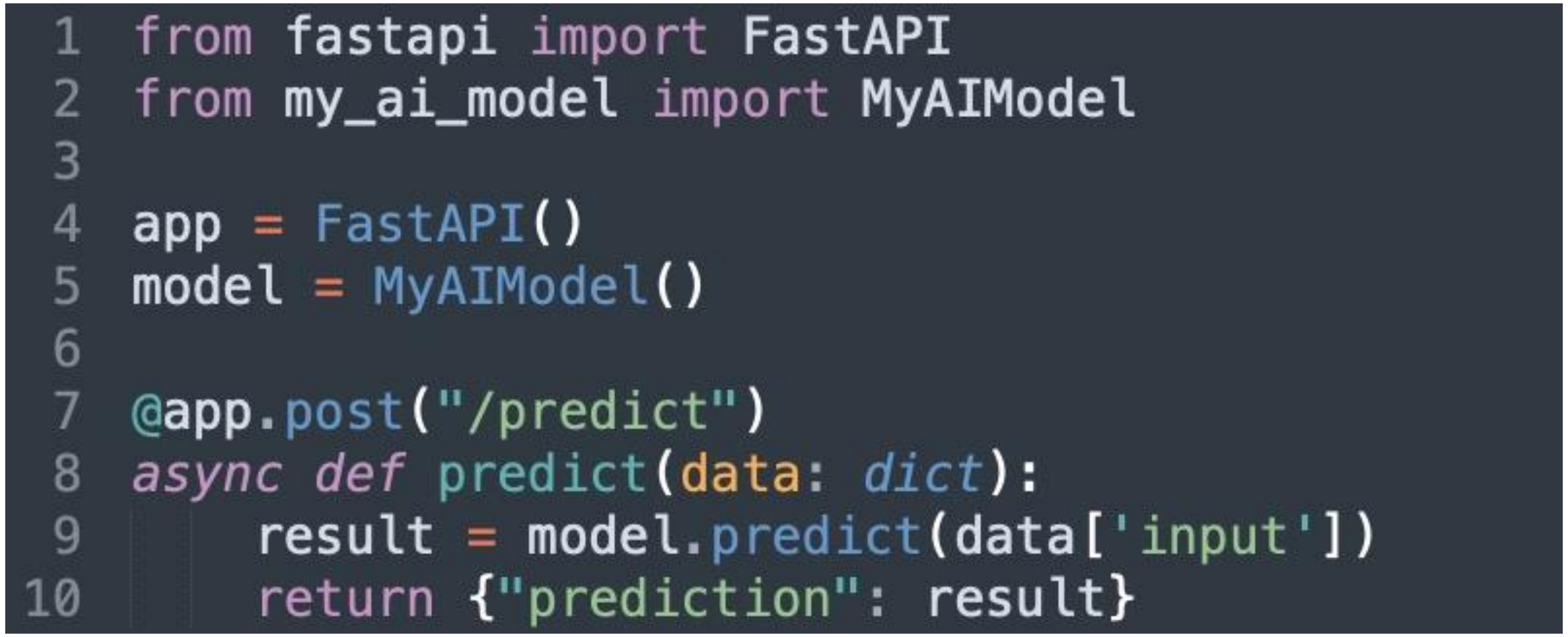
Let‘s continue learning and mastering the skills in workshop!

Integrating AI Model with FastAPI

Installing the AI Model Package

pip install -e path/to/my_ai_model

Using the Model in FastAPI



```python
from fastapi import FastAPI
from my_ai_model import MyAIModel

app = FastAPI()
model = MyAIModel()

@app.post("/predict")
async def predict(data: dict):
    result = model.predict(data['input'])
    return {"prediction": result}
```

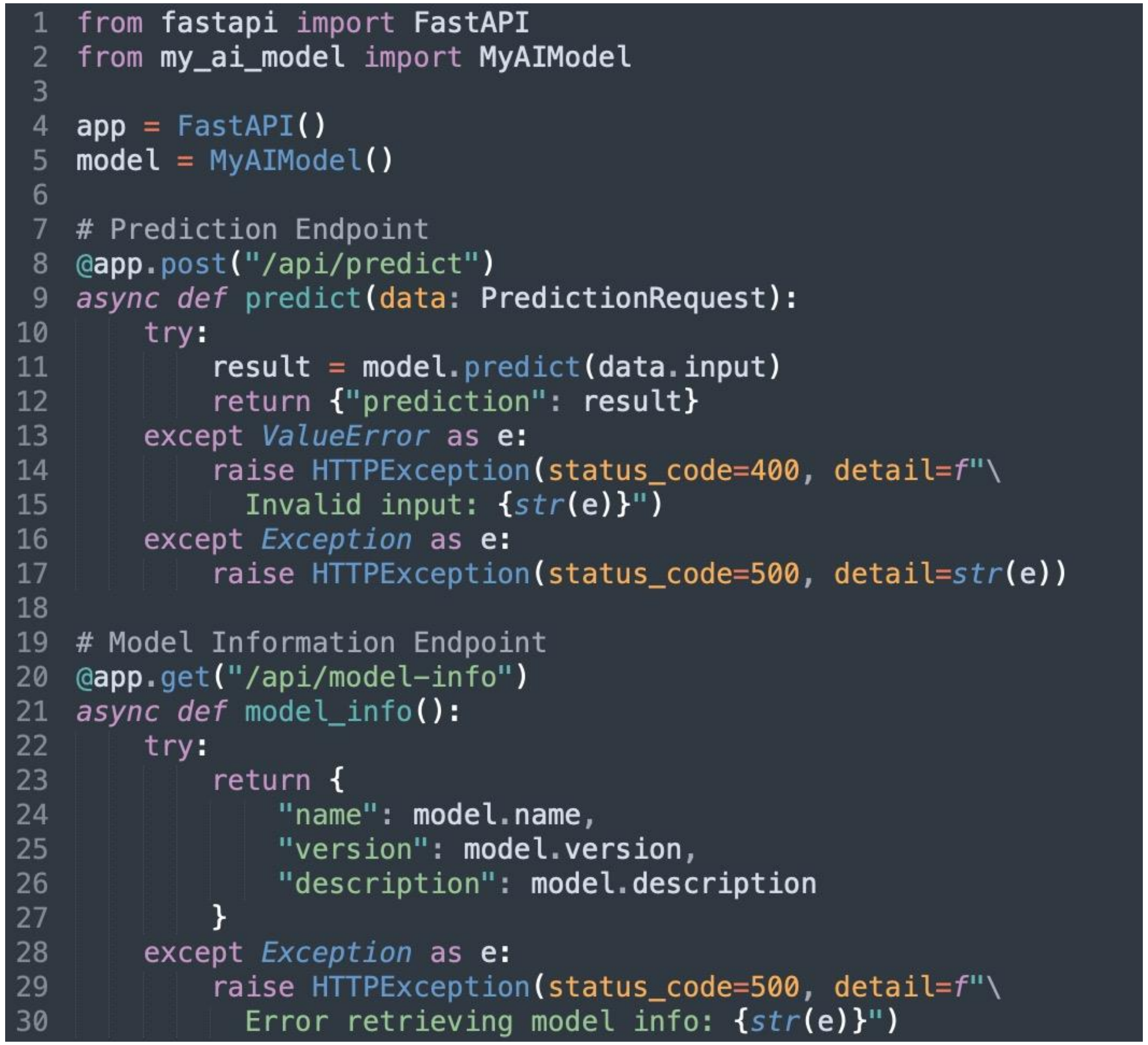
Implementing API Endpoints for AI Functionality

Prediction Endpoint

Model Information Endpoint

Error Handling

- Implement try-except blocks for robust error management



```python
from fastapi import FastAPI
from my_ai_model import MyAIModel

app = FastAPI()
model = MyAIModel()

# Prediction Endpoint
@app.post("/api/predict")
async def predict(data: PredictionRequest):
    try:
        result = model.predict(data.input)
        return {"prediction": result}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=f"\
          Invalid input: {str(e)}")
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

# Model Information Endpoint
@app.get("/api/model-info")
async def model_info():
    try:
        return {
            "name": model.name,
            "version": model.version,
            "description": model.description
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"\
          Error retrieving model info: {str(e)}")
```

Using Axios in React for API Calls

What is Axios?

- Promise-based HTTP client for browser and Node.js
- Simplifies making HTTP requests from JS

Key Features

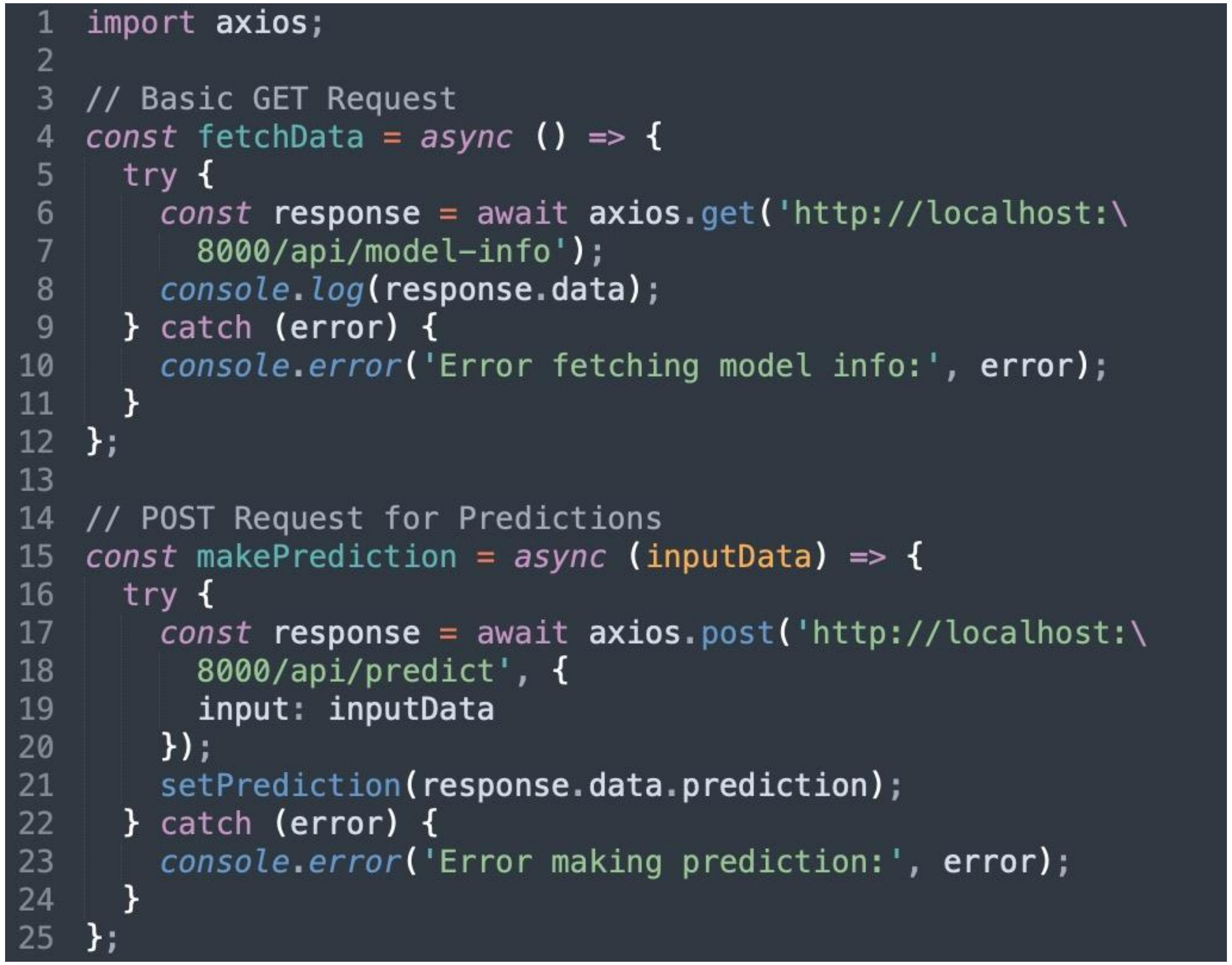
- Automatic transforms for JSON data
- Client-side protection against XSRF
- Easy to use API

Installation

- npm install axios

Basic GET Request

POST Request for Predictions



```
import axios;

// Basic GET Request
const fetchData = async () => {
  try {
    const response = await axios.get('http://localhost:\
      8000/api/model-info');
    console.log(response.data);
  } catch (error) {
    console.error('Error fetching model info:', error);
  }
};

// POST Request for Predictions
const makePrediction = async (inputData) => {
  try {
    const response = await axios.post('http://localhost:\
      8000/api/predict', {
      input: inputData
    });
    setPrediction(response.data.prediction);
  } catch (error) {
    console.error('Error making prediction:', error);
  }
};
```

Understanding CORS

Image from [Medianova]

What is CORS?

- Cross-Origin Resource Sharing
- Security feature implemented by web browsers
- Restricts web pages from making requests to a different domain

Why is it Necessary?

- Prevents malicious scripts on one page from obtaining access to sensitive data on another domain
- Helps protect against attacks like Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF)

How CORS Works?

- Browser sends a preflight request to check if cross-origin access is allowed
- The server responds with CORS headers defining what’s permitted

Common CORS Headers

- Access-Control-Allow-Origin: Specifies allowed origins
- Access-Control-Allow-Methods: Lists allowed HTTP methods
- Access-Control-Allow-Headers: Specifies allowed custom headers

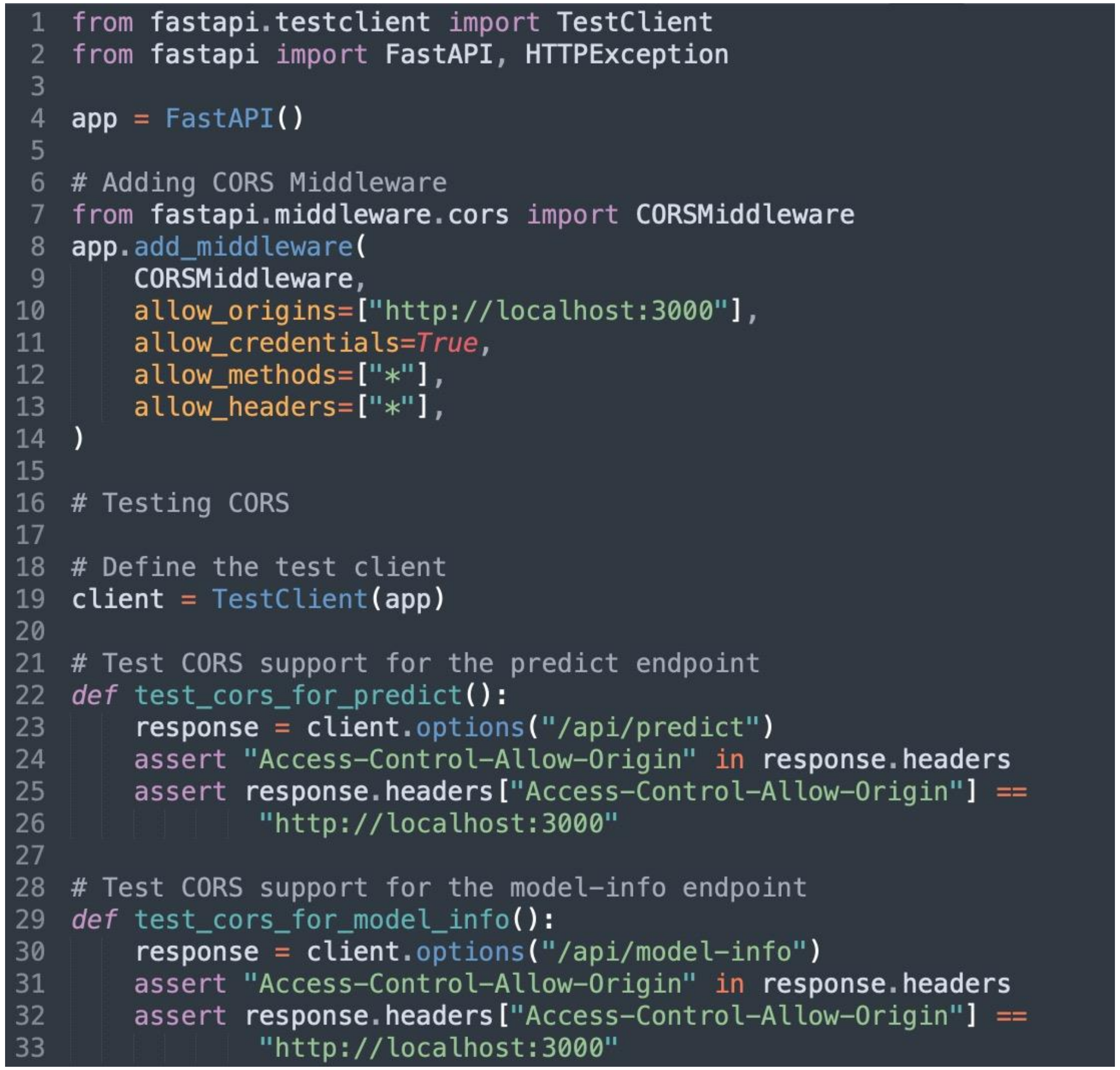
Implementing CORS in FastAPI

Adding CORS Middleware

Security Considerations

- Only allow necessary origins in production
- Restrict methods and headers as needed

Testing CORS



```python
from fastapi.testclient import TestClient
from fastapi import FastAPI, HTTPException

app = FastAPI()

# Adding CORS Middleware
from fastapi.middleware.cors import CORSMiddleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Testing CORS

# Define the test client
client = TestClient(app)

# Test CORS support for the predict endpoint
def test_cors_for_predict():
    response = client.options("/api/predict")
    assert "Access-Control-Allow-Origin" in response.headers
    assert response.headers["Access-Control-Allow-Origin"] ==
            "http://localhost:3000"

# Test CORS support for the model-info endpoint
def test_cors_for_model_info():
    response = client.options("/api/model-info")
    assert "Access-Control-Allow-Origin" in response.headers
    assert response.headers["Access-Control-Allow-Origin"] ==
            "http://localhost:3000"
```

Visualizing AI Model Results in React

Choosing Visualization Libraries

- Recharts: Simple and customizable charts (e.g., line, bar, pie).
- D3.js: Highly flexible, low-level control over visualizations.
- Chart.js: Easy-to-use, supports various chart types, good for quick implementation.
- Victory: A React-specific charting library focused on data visualization

Key Design Principles

- Keep the charts simple and intuitive, focus on clarity over complexity.
- Use consistent color schemes to make data easy to distinguish.
- Add tooltips and legends for better user interaction and understanding.

Responsive Design Tips

- Ensure visualizations are responsive to different screen sizes (desktop, tablet, mobile).
- Use CSS media queries or built-in library features for responsive layouts.

Aesthetic Improvements

- Use smooth transitions. Consider animations to make charts feel more dynamic.

Visualizing AI Model Results (Assignment3)

Basic Requirements

- AI Model results are processed on the backend
- Frontend receives ready-to-display images and data
- Simple, effective display of results

Advanced Implementation (Optional)

- Implement interactive elements for higher scores
- Examples of advanced features:

    1. Clickable data points for detailed view
    2. Zoom functionality for complex visualizations
    3. Toggleable data layers or filters

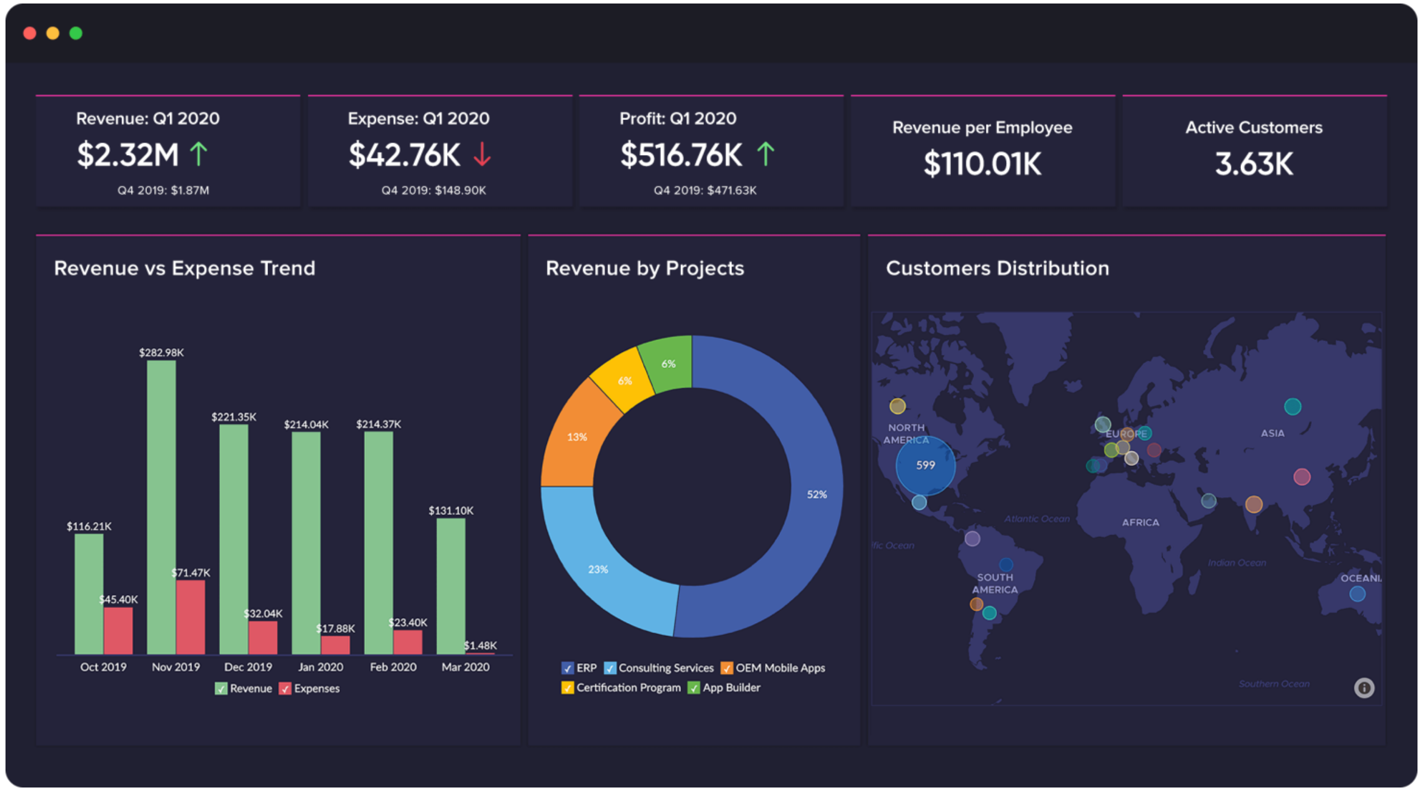


Image from [Zoho]

Handling Long-Running Predictions

Implementing Websockets for Real-time Updates

- Set up a FastAPI WebSocket endpoint to send prediction progress updates.
- Use a React WebSocket client to receive and display progress in real time.

Progress Indicators

- Display loading spinners or progress bars while waiting for predictions.
- If possible, provide an estimated time remaining for the prediction to complete.

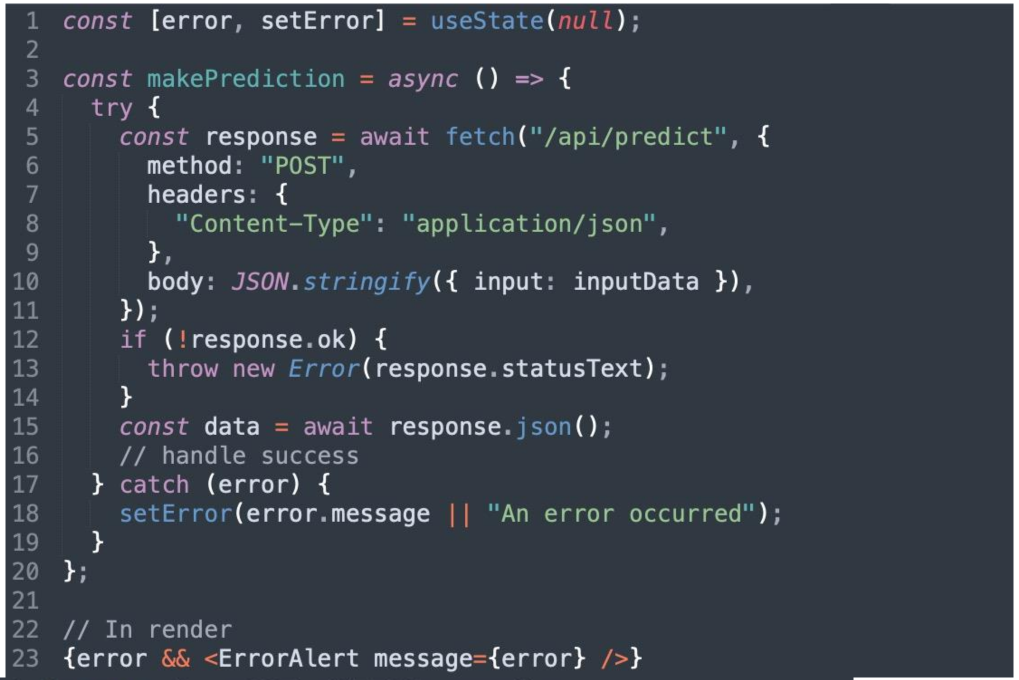
Async Predictions

- Use async processing in FastAPI to handle long-running predictions without blocking other requests.
- Consider sending intermediate results or status updates to keep users informed.

User Experience Enhancements

- Notify the user when the prediction is done, either via a notification or by automatically updating the results on the UI.
- Avoid freezing the UI during the prediction by keeping it responsive and interactive.

Error Handling in Practice



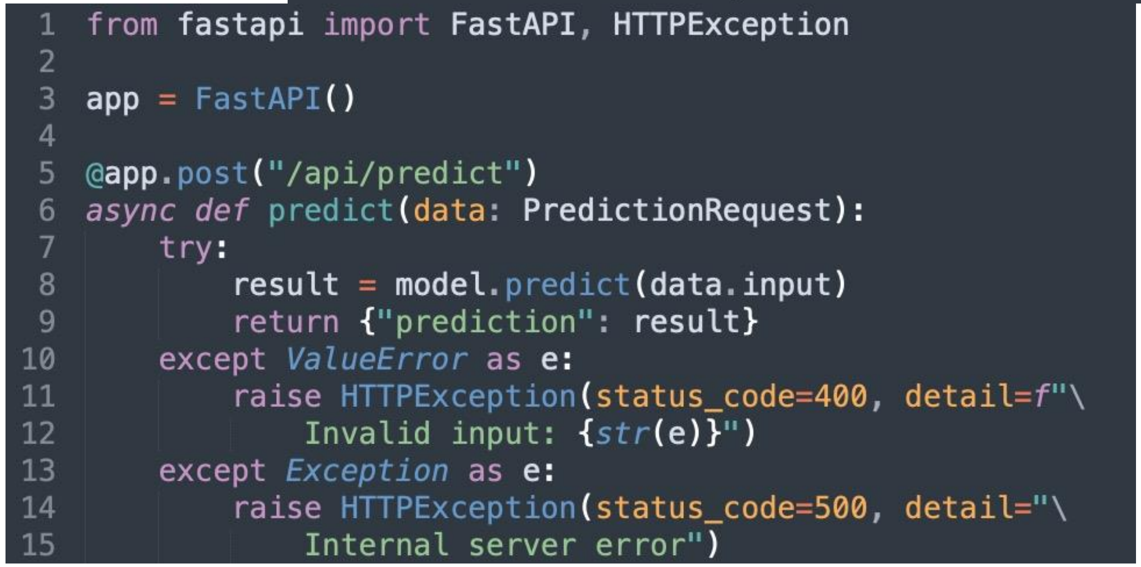
```javascript
const [error, setError] = useState(null);

const makePrediction = async () => {
  try {
    const response = await fetch("/api/predict", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ input: inputData }),
    });
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    const data = await response.json();
    // handle success
  } catch (error) {
    setError(error.message || "An error occurred");
  }
};

// In render
{error && <ErrorAlert message={error} />}
```

Client-Side Error Handling in React

- Handling API Errors: Ensure API call errors are caught and displayed in a user-friendly manner.

Server-Side Error Handling in FastAPI

- Handling Predictive Errors: Ensure any errors during model prediction are caught and meaningful error messages are returned.



```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.post("/api/predict")
async def predict(data: PredictionRequest):
    try:
        result = model.predict(data.input)
        return {"prediction": result}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=f"\
          Invalid input: {str(e)}")
    except Exception as e:
        raise HTTPException(status_code=500, detail="\
          Internal server error")
```

Error Handling and User Feedback

Improving User Feedback

- User-Friendly Error Messages: Avoid showing technical details to the user; instead, translate errors into understandable messages.

Example: Instead of showing “500 Internal Server Error,” display “Something went wrong. Please try again later.”

- Differentiate Between Error Types: Handle different error statuses (e.g., 400 for bad requests, 500 for server issues) and provide context to users.

For a 400 error: “It looks like there was an issue with your input. Please check and try again.”

Best Practices for Error Handling

- Frontend: Always catch errors from API calls and ensure the user gets clear feedback. Consider adding retry mechanisms for transient errors.
- Backend: Return descriptive and non-technical error messages while logging technical details for developers to diagnose.
- General UX: Show a loading spinner while waiting for the prediction, and a clear error message if something goes wrong, so users aren’t left in uncertainty.

Have a Break and Let's continue with Lecture 02

Image from [Pinterest]

SWINBURNE UNIVERSITY OF TECHNOLOGY