ENSF 381
Full Stack Web Development

Lecture 35:
Recap: React and Flask

Ahmad Abdellatif, PhD

UNIVERSITY OF CALGARY

Final Exam

- **The final exam is closed book.**
- **Students must pass the final exam in order to pass the course**
- **The passing mark for the exam is 50%.**
- **All materials covered throughout the semester will be included in the final exam. The material covered in review classes is intended for review purposes only.**
- **Each student is required to bring their UCID.**
- Students who arrive late will not be admitted after **thirty minutes** have elapsed from the start of the examination.
- Practice questions provided are intended for training purposes **and may or may not appear in the final exam.**
- The final exam may consist of a diverse range of question formats, including but not limited to coding questions, and questions requiring detailed responses. **These formats may differ from those used in the practice questions.**

Recap: React

- React is a free and open-source front-end JavaScript library.
- Created by Facebook/Meta.
- Allows to create reusable components.
- Used to create user interfaces (UI) through the composition of components.

React application folder structure

- Node_modules:
  - This directory houses all the Node packages installed through npm.
  - Due to the use of create-react-app, several node modules are already present in this folder.
  - This directory is typically managed by npm commands on the command line, involving installation and uninstallation.

- Public: contains development files including "public/index.html"

- package.json: presents the node package dependencies and various project configurations such as project name, version, entry point, and scripts.

React application folder structure

- Package-lock.json: provides a detailed description of the exact versions of packages and their dependencies in a Node.js project. Used to lock down the versions of packages to ensure consistent installations across different environments.

- README.md: provides instructions and useful information about the project. It is a markdown file.

How React works?

- To work with React, we **need to include React library** for creating views: `import React from 'react';`

- React establishes a virtual DOM in memory, where it performs all required manipulations before applying changes to the actual browser DOM.

- React does not **directly** manipulate the browser's DOM.

How React works?

- A mechanism that allows React to efficiently manage updates to the user interface by first making changes in a virtual representation and then selectively applying those changes to the actual DOM.
- When changes occur in a React application, React first makes these changes in the virtual DOM rather than directly manipulating the browser's DOM.
- React then performs a process called "reconciliation" to identify the differences between the current virtual DOM and its previous state.
- Only the specific changes or differences identified in the virtual DOM are then applied to the actual DOM.
- This strategy contributes to a more responsive and performant web application.

React JSX

- JSX stands for JavaScript XML.

- JSX allows us to write HTML elements and components within JavaScript code:

```
const element = <h1>Hello, World!</h1>;
```

- Under the hood, React transforms this JSX code into JavaScript code using a process called **transpilation**, **making it compatible with browsers**.

React Component

- A reusable and self-contained building block for building user interfaces.

- They can represent anything from simple UI elements, such as buttons or input fields, to more complex structures like entire sections of a webpage or even entire pages.

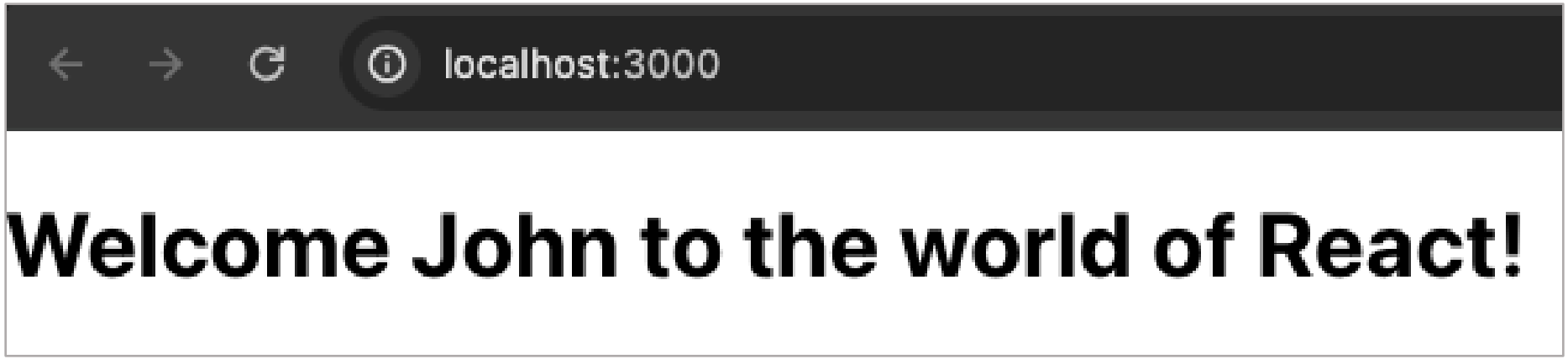
- Functions that return HTML elements.

Component – example 2

```
import React from 'react';
function App() {

const name = "John"

return (
<div>
<h1>Welcome {name} to the world of React!</h1>
</div>
);
}
export default App;
```

Embed JavaScript expressions within JSX.

Component – example 2



Lists in React

- A collection of elements or components rendered in a specific order.

- Commonly used to **display dynamic data**, where the number of items may vary, and you want to render each item in a repetitive structure.

Map function in JavaScript

- An array method that is used to iterate over each element of an array and apply a given function to each element.

- The result is a new array where each element is the result of applying the provided function to the corresponding element of the original array.

- The original array **remains unchanged**.

List - Example

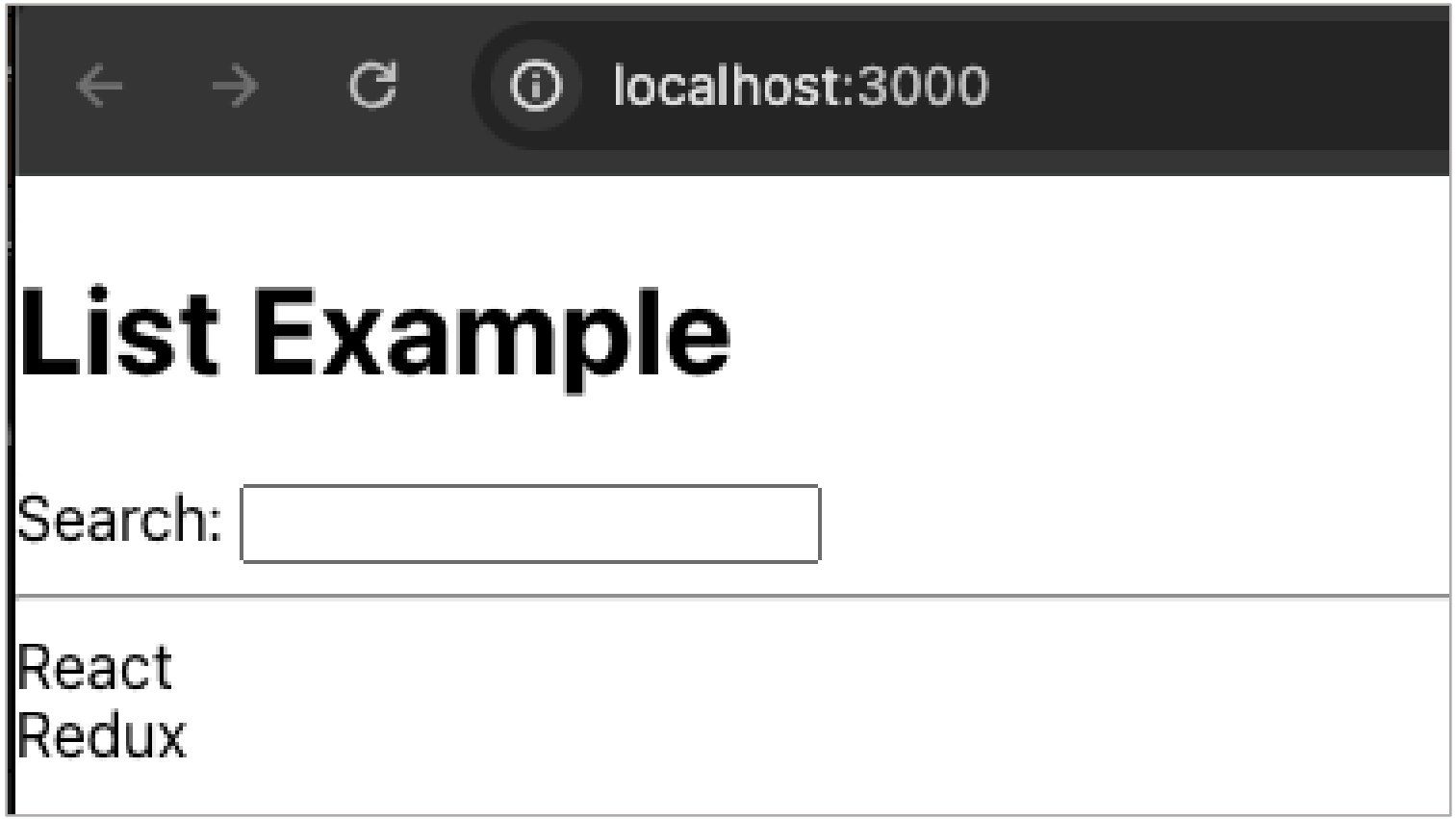
```
import React from 'react';
const list = [
{
title: 'React',
url: 'https://reactjs.org/',
author: 'Jordan Walke',
num_comments: 3,
points: 4,
objectID: 0,
},
{
title: 'Redux',
url: 'https://redux.js.org/',
author: 'Dan Abramov, Andrew Clark',
num_comments: 2,
points: 5,
objectID: 1,
},
];
```

List - Example

```
function App() {
return (
<div>
<h1>List Example</h1>
<label htmlFor="search">Search: </label>
<input id="search" type="text" />
<hr />
{list.map(function(item) {
return <div>{item.title}</div>;
})}
</div>
);
}

export default App;
```


Src: The Road to React: The React.js with Hooks in JavaScript Book

List - Example



Customizing component behavior and appearance with Props

- Pass data from a parent component to a child component.
- A set of arguments that are passed to a React component.
- These arguments are **similar to parameters in a function**, providing a way to customize the behavior and appearance of a component.
- **Props are read-only**: components can not modify the props they receive; they are considered immutable.
- "props" is an abbreviation for "properties".

React Router

- A popular library for handling navigation and routing in React applications.

- It enables the creation of **single-page applications** by allowing developers to define different "routes" within their application and rendering the appropriate components based on the current URL.

- We need to install the React Router Library*:

```
npm install react-router-dom
```

***Ensure that you are located at the root of your React project.**

Router – Example (App)

```
import React from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Home from './Home';
import AboutUs from './AboutUs';
import ContactUs from './ContactUs';

function App() {
return(
<BrowserRouter>
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/About" element={<AboutUs />} />
  <Route path="/ContactUs" element={<ContactUs />} />
</Routes>
</BrowserRouter>
);
}

export default App;
```

Router – Example (AboutUs)

```
import React from 'react';

function AboutUs() {
  return
    <h1>
      This is About Us page!
    </h1>;
};

export default AboutUs;
```

Router – Navigate to another page using Event (ContactUs)

```
import React from 'react';
import {useNavigate } from 'react-router-dom';

function ContactUs() {

const navigate = useNavigate(); // This hook is used for programmatic navigation in a React application

function handleButtonClick(){
  navigate("/About")
}
  return (
    <div>
       <h1>For any question, please contact us at: info@info.com</h1>
       <button onClick={handleButtonClick}>Go to About Us page!</button>
    </div>
  );
};

export default ContactUs;
```

Router – Navigate to another page using Event (ContactUs)

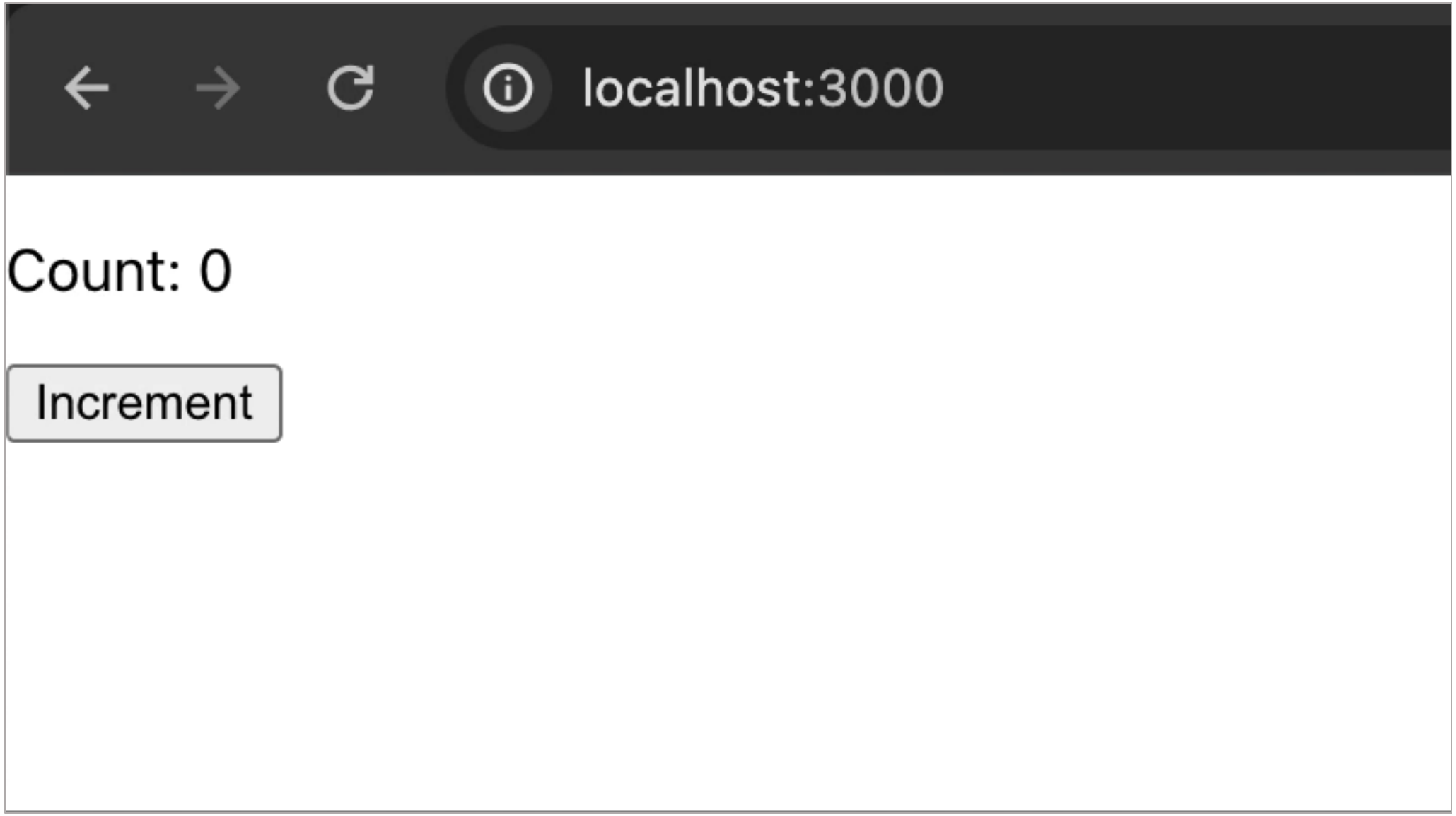
useState

- One of the fundamental React Hooks used to manage state in components.
- It allows developers to add state to the components, making them more powerful and versatile.
- Track of data that may change over time, causing the component to re-render.
- Hooks can **only be called inside components**.
- The useState hook returns an array with two elements:
  - The current state value.
  - A function that allows to update the state.

Example on incrementing the count when the button is clicked using UseState

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

function Counter() {
const [count, setCount] = useState(0);

// Define a function to handle incrementing the count
function handleIncrement() {
  setCount(count + 1);
};

return (
  <div>
    <p>Count: {count}</p>
    <button onClick={handleIncrement}>
      Increment
    </button>
  </div>
);
}
export default Counter;
```

Example on incrementing the count when the button is clicked using UseState



Question….

What are some specific scenarios or types of applications where useState proves to be particularly useful?

- **Dynamic UI Updates**: facilitate dynamic updates in the UI based on user interactions. State changes **trigger re-renders**, ensuring the UI reflects the latest user input.

- **Form Handling**: each form input (like text fields, checkboxes, etc.) can have its state managed independently.

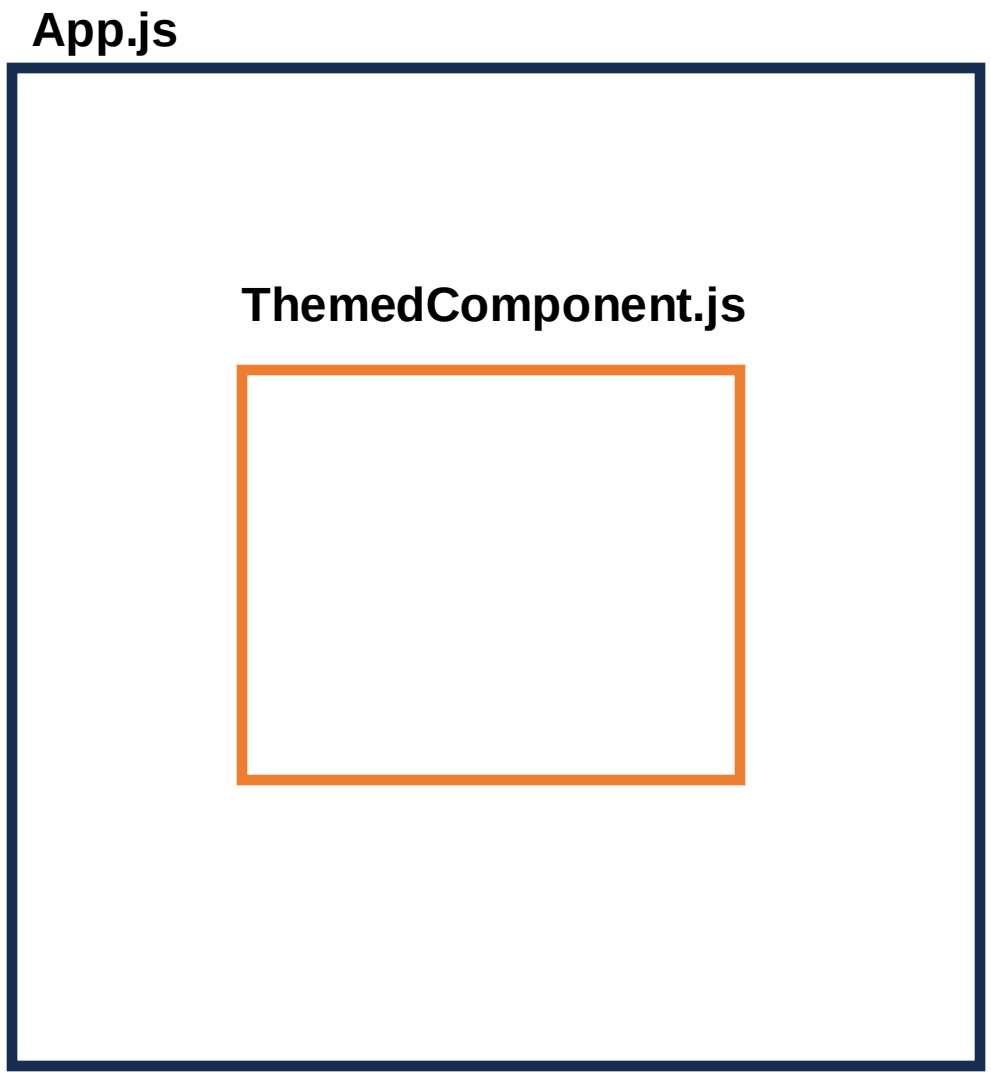
useContext

- Provides a way to access the values from a React context directly within a component.

- Making it easier to share data across **different parts of an application without prop drilling**.

- Allows developers to efficiently share and consume context values in a React application.

UseContext - Example



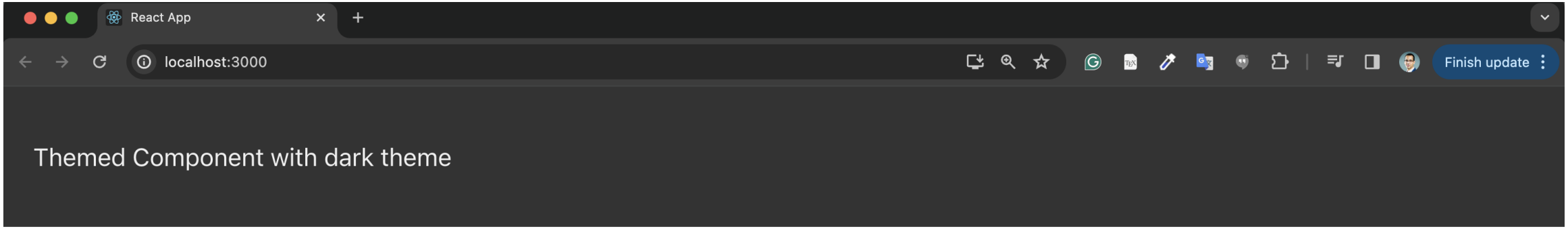
UseContext – Example (App)

```
import {React, createContext} from 'react';
import ThemedComponent from './ThemedComponent';

export const ThemeContext = createContext(null);

function App () {
const theme = 'dark';

return (
  <ThemeContext.Provider value={{theme}}>
    <ThemedComponent />
  </ThemeContext.Provider>
);
};

export default App;
```

UseContext – Example (ThemedComponent)

```
import React from 'react';
import { useContext } from 'react';
import { ThemeContext } from './App';

function ThemedComponent() {
const { theme } = useContext(ThemeContext);

return (
  <div style={{ background: theme === 'light' ? '#f0f0f0' : '#333', padding: '20px'
}}>
    <p style={{ color: theme === 'light' ? '#333' : '#f0f0f0' }}>
      Themed Component with {theme} theme
    </p>
  </div>
);
}

export default ThemedComponent;
```

UseContext - Example



useEffect

- A hook that allows components to perform side effects in a React application.

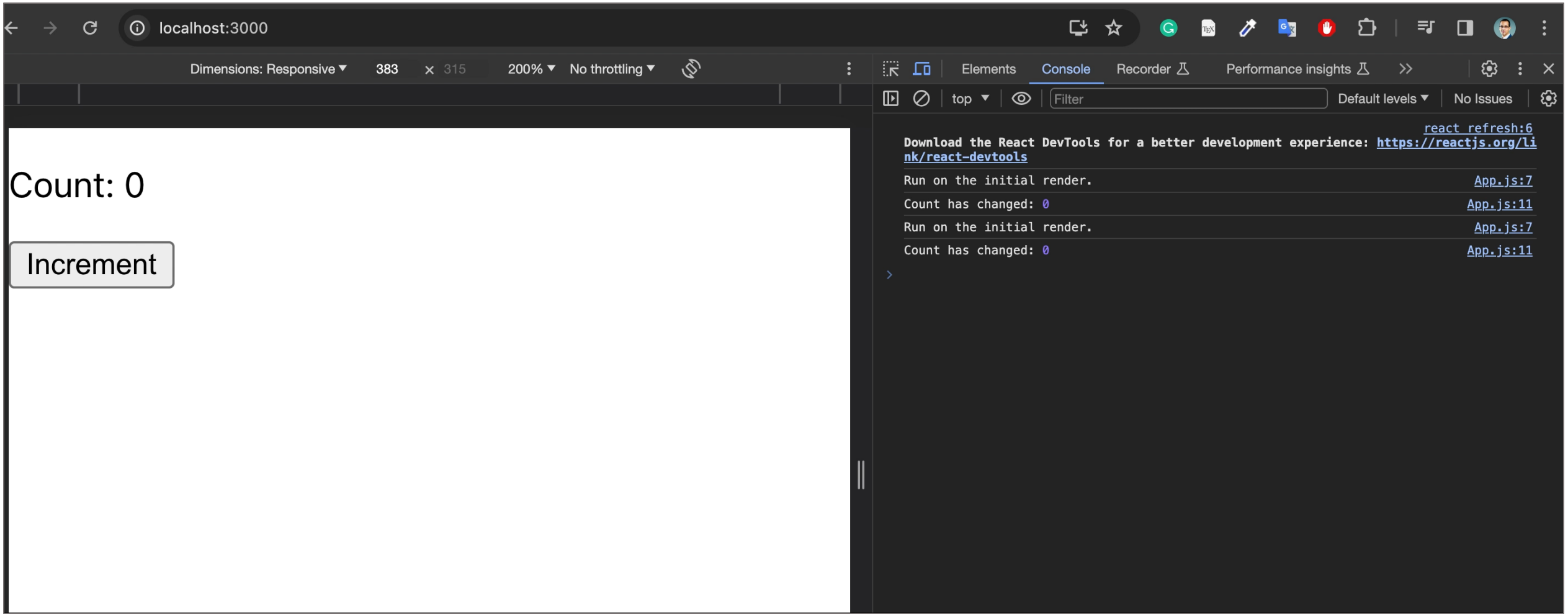
- Side effects in this context refer to operations that are not directly related to rendering the user interface.

- Side effects can include things like:
  - Fetching data from an API.
  - Subscribing to external data sources.
  - Any other asynchronous or imperative operations.

useEffect - Example

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

function ExampleComponent() {
const [count, setCount] = useState(0);

useEffect(() => { // Effect for running code on the initial render
  console.log('Run on the initial render.');
}, []);

useEffect(() => {// Effect for running code on the first render when the 'count' state changes
  console.log('Count has changed:', count);
}, [count]);

return (
  <div>
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
);};
export default ExampleComponent;
```

useEffect - Example



**The app rendered twice due to the use of StrictMode. In other words, StrictMode causes the app to render twice during development (but not in production).

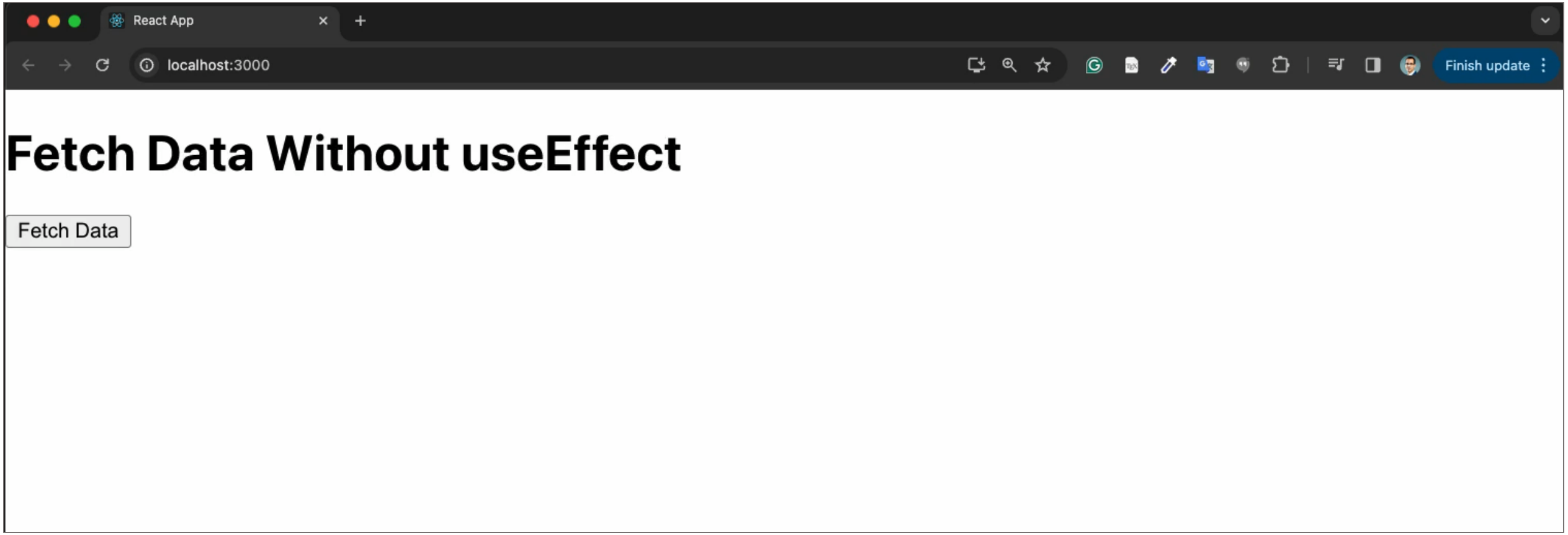
Fetching data using .then()

- Promises are a way to handle asynchronous operations in JavaScript.
- .then() executes the code after a Promise is successfully resolved.
- Multiple .then() methods can be chained to handle sequential steps in asynchronous operations.
- Enhances readability and makes it easier to handle success and error scenarios.

Fetching data using .then() - Example

```
function fetchData() {
    // Set loading to true while data is being fetched
    setIsLoading(true);

    // Fetch data from an API using .then()
    fetch('https://api.randomuser.me/?nat=US&results=1')
      .then((response) => response.json())
      .then((data) => {
        // Parse the JSON data from the response
        let { email, cell } = data.results[0];

        // Set the fetched data to the state
        setEmail(email);
        setCellphone(cell);
      })
      .catch((error) => {
        // Handle error if the request was not successful
        console.error('Failed to fetch data:', error.message);
      })
      .finally(() => {
        // Set loading to false once data fetching is complete
        setIsLoading(false);
      }); }
```

Fetching data using .then() - Example



Client-Side data storage

Sometimes we need to store data on the client-side:

- Storing user preferences and settings to provide a personalized experience.

- Persisting authentication tokens to keep users logged in across page reloads.

Local Storage

- Allows web applications to store data persistently on a user's device.

- A simple key-value storage mechanism and is designed to retain data even when the user closes the browser or navigates away from the page.

Example on storing and retrieving user preferences

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

function App() {
const [theme, setTheme] = useState('light'); // State to track the current theme

// Function to toggle between light and dark themes
function toggleTheme() {
  const newTheme = theme === 'light' ? 'dark' : 'light';
  setTheme(newTheme);
  localStorage.setItem('theme', newTheme); // Save the theme (case-sensitive) preference to local storage
};

// Effect to retrieve the theme preference from local storage on component render
useEffect(() => {
  const savedTheme = localStorage.getItem('theme'); // Retrieve the theme preference from local storage
  setTheme(savedTheme ? savedTheme : 'light'); // Set the theme based on the stored preference or default to 'light'
}, []);

return (
  <div>
    <h1>Current Theme: {theme}</h1>
    <button onClick={toggleTheme}>Toggle Theme</button>
  </div>
);
};
export default App;
```

Example on storing and retrieving user preferences

React CSS styling

There are many ways to style React with CSS. The most common methods include:

- **CSS stylesheet**: a CSS stylesheet is an external file containing styles written in Cascading Style Sheets (CSS).

- **Inline styling**: involves applying styles directly within HTML elements using the style attribute.

- **CSS modules**: are a CSS file organization technique in React that locally scopes styles to specific components. **Each component imports its own CSS module**, preventing style conflicts and allowing for encapsulation of styles within the component.

CSS modules

- We need to use styled-components library.
- Enable developers to write CSS in JS while building custom components in React.
- To install the library, run the following command:

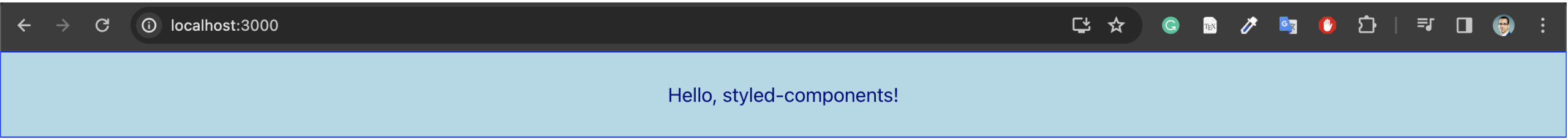
```
npm install styled-components
```

CSS modules - Example

```
import React from 'react';
import styled from 'styled-components';

// Create a styled component using the styled() function
const StyledDiv = styled.div`
background-color: lightblue;
padding: 10px;
border: 1px solid blue;
text-align: center;
`;
const StyledText = styled.p`
color: navy;
font-size: 18px;
`;

function App() {
return (
  <StyledDiv>
    <StyledText>Hello, styled-components!</StyledText>
  </StyledDiv>
);};
export default App;
```

Create a styled version of a specific HTML element, in this case, a div.

CSS modules - Example



Use cases

CSS Stylesheet

- **Reusability**: when styles need to be shared across multiple components or pages.
- **Consistent Styling**: when consistency in styling across the application is a high priority.

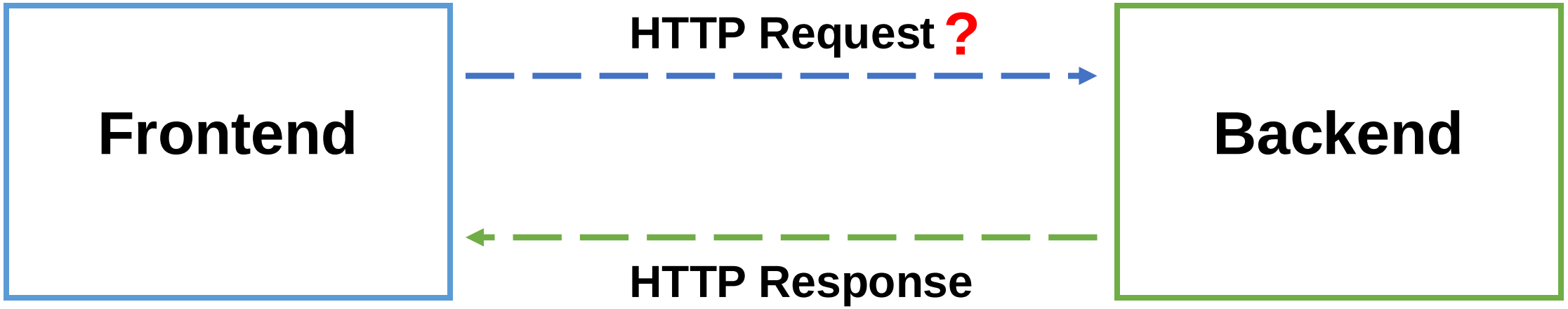
Inline Styling:

- **Small Components**: for smaller, self-contained components where encapsulating styles is not a high priority.
- **Dynamic Styles**: when styles need to be computed dynamically based on component state or props.

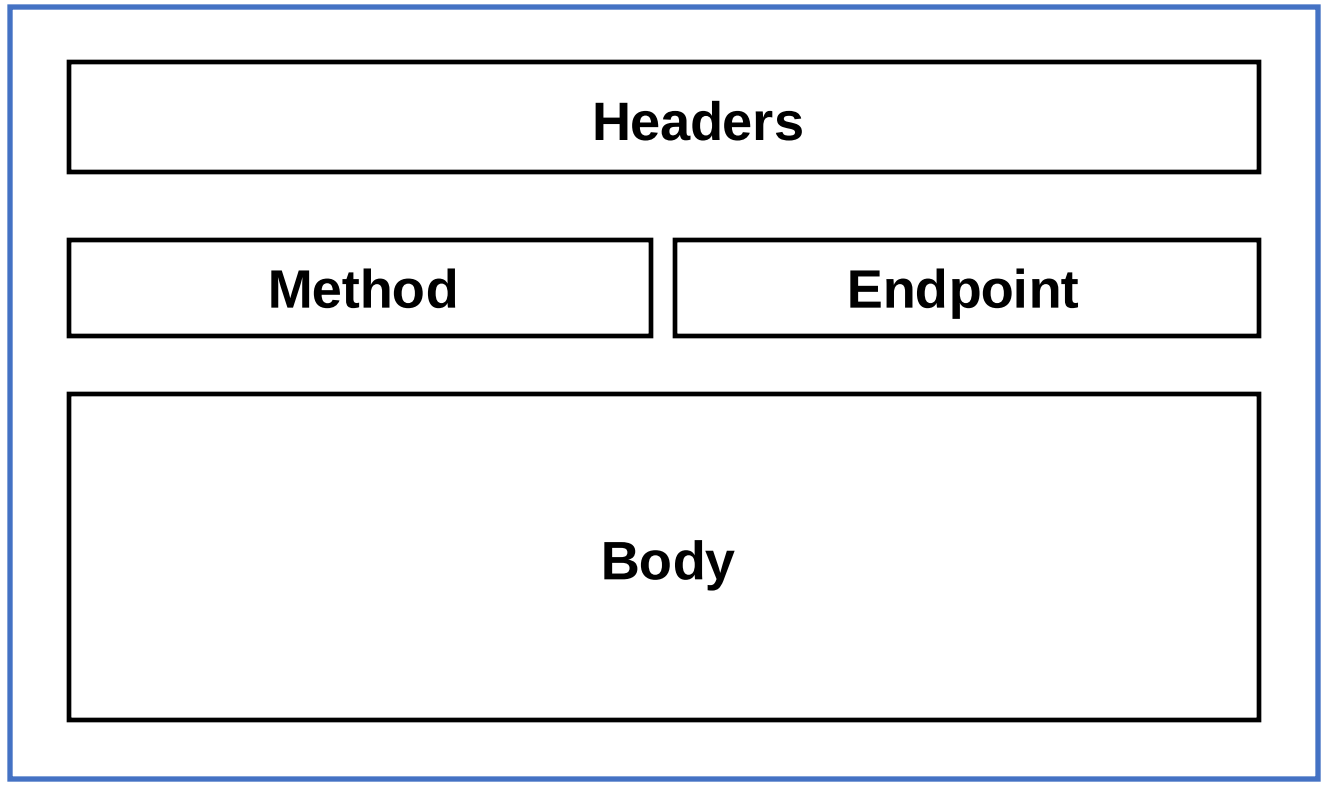
CSS Modules:

- **Component-Level Styling**: when emphasizing component-level styling and encapsulation.
- **Dynamic Styling**: convenient for dynamic styles using props or theme variables.

HTTP - Example



HTTP Request



Headers: contains information related to some authentication data, and browser type.

Endpoint: the URL of the API endpoint.

Body: contains data that needs to be sent to the server.

Method: Indicates the HTTP method used (e.g., GET, POST, PUT, DELETE).

CRUD

- Represents basic operations that can be performed on data in a database or a persistent storage system.

- These operations are fundamental in the context of data management and are commonly associated with database systems and web applications.

- CRUD is an acronym that stands for Create, Read, Update, and Delete.
  - Create (C): involves creating new records or entries in a database.
  - Read (R): involves retrieving or reading data from a database.
  - Update (U): involves modifying existing records or entries in a database.
  - Delete (D): involves removing records or entries from a database.

HTTP methods - GET

GET: retrieve data from a specified resource.

- Example: fetching a webpage, image, or any resource without modifying the server's state.
- GET is the **default** method.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log('GET response:', data);
    // Handle the retrieved data
  })
  .catch(error => console.error('GET error:', error));
```

HTTP methods - POST

POST: submit data to be processed to a specified resource.

- Example: submitting a form, uploading a file, or making a request that results in the creation of a new resource on the server.

```
const postData = { key: 'value' };

fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(postData),
})
  .then(response => response.json())
  .then(data => {
    console.log('POST response:', data);
    // Handle the response data
  })
  .catch(error => console.error('POST error:', error));
```

HTTP methods - PUT

PUT: update or modify a resource on the server.

- Example: updating user profile information, uploading a new version of a file.

```
const putData = { updatedKey: 'updatedValue' };

fetch('https://api.example.com/data/123', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(putData),
})
  .then(response => response.json())
  .then(data => {
    console.log('PUT response:', data);
    // Handle the response data
  })
  .catch(error => console.error('PUT error:', error));
```

HTTP methods - DELETE

DELETE: delete a specified resource.

- Example: deleting a user account, removing a file from a server.

```
fetch('https://api.example.com/data/123', {
  method: 'DELETE',
 })
  .then(response => {
    if (!response.ok) {
      throw new Error('DELETE request failed');
    }
    console.log('DELETE request successful');
    // Handle success
  })
  .catch(error => console.error('DELETE error:', error));
```

Python

- Created by Guido van Rossum in the late 1980s.
- The language was designed with the idea that code should be easy to write and understand.
- An open-source language, meaning its source code is freely available to the public.
- Python has gone through several major versions, with the most notable being Python 2 and Python 3.

Variables

- Variables are used to store and reference data values in the program.
- Python is dynamically typed, eliminating the need to declare the data type of variables explicitly.
- Variable names are **case-sensitive**.
- Comments starts with #, and Python will ignore them.

String useful methods

- `format`: a method used for string formatting, allowing the insertion of values into a string template.

- `split`: a method applied to strings that divides the string into a list of substrings based on a specified delimiter.

- `strip`: a method used to remove leading and trailing whitespaces (including newline characters) from a string.

- `find`: a method that searches for a specified substring within a string and returns the index of the first occurrence. If not found, it returns -1.

- `replace`: a method that replaces occurrences of a specified substring with another substring in a string.

String useful methods

Code Snippet

```
name = "John"
age = 25
message = "My name is {} and I am {} years old.".format(name, age)
print(message)
```

→ My name is John and I am 25 years old.

```
sentence = "Hello, world!"
words = sentence.split(", ")
print(words)
```

→ ['Hello', 'world!']

```
text = "   This is a string with whitespace.   "
clean_text = text.strip()
print(clean_text)
```

→ This is a string with whitespace.

```
sentence = "Python is powerful and Python is easy to learn."
index = sentence.find("Python")
print(index)
```

→ 0

```
sentence = "Python is fun!"
new_sentence = sentence.replace("fun", "awesome")
print(new_sentence)
```

→ Python is awesome!

Output

What are Arrays?

- Arrays are data structures that store multiple items in a single variable.
- In Python, arrays are represented using lists.
- Accessing elements using index (indexing starts from 0).
- Syntax:

```
my_array = [element1, element2, element3]
```

Arrays - methods

- `len`: is a built-in function used to get the length (the number of elements) of a sequence, such as a list, tuple, or string.

- `append`: adds a specified element to the end of the list.

- `insert`: inserts a specified element at a given position (index) in the list.

- `remove`: removes the first occurrence of a specified value from the list.

- `extend`: appends the elements of an iterable (e.g., list, tuple) to the end of the current list, effectively extending its length.

Arrays - methods

Code snippet	Output

`my_list = [1, 2, 3, 4, 5]`
`length = len(my_list)`
`print(length)`
	5

`my_list = [1, 2, 3, 4, 5]`
`my_list.append(6)`
`print(my_list)`
	[1, 2, 3, 4, 5, 6]

`my_list = [1, 2, 3, 4, 5]`
`my_list.insert(2, 7)`
`print(my_list)`
	[1, 2, 7, 3, 4, 5]

`my_list = [1, 2, 3, 4, 5]`
`my_list.remove(4)`
`print(my_list)`
	[1, 2, 3, 5]

`my_list = [1, 2, 3, 4, 5]`
`additional_elements = [6, 7, 8]`
`my_list.extend(additional_elements)`
`print(my_list)`
	[1, 2, 3, 4, 5, 6, 7, 8]

Dictionary

- Dictionary is versatile and powerful data structures in Python.

- They are unordered collections of key-value pairs, providing fast and efficient data retrieval.

- Use Cases:
  - Ideal for scenarios where data retrieval based on a unique identifier (key) is crucial.
  - Used to represent real-world entities and their associated attributes.

Dictionary - example

```python
# Define a dictionary
student_info = {
    'name': 'Alice',
    'age': 20,
    'grade': 'A+',
    'courses': ['Math', 'Physics', 'English']
}

# Accessing elements
student_name = student_info['name']
courses_taken = student_info['courses']

print("Original Dictionary:")
print("Student Name:", student_name)
print("Courses Taken:", courses_taken)

# Modifying elements
student_info['age'] = 21  # Update the age
student_info['courses'].append('History')  # Add a new course

print("Modified Dictionary:")
print("Updated Age:", student_info['age'])
print("Updated Courses:", student_info['courses'])
```

Dictionary - example

```
Original Dictionary:
Student Name: Alice
Courses Taken: ['Math', 'Physics', 'English']
Modified Dictionary:
Updated Age: 21
Updated Courses: ['Math', 'Physics', 'English', 'History']
```

Functions

- A function in Python is a reusable and self-contained block of code designed to perform a specific task.
- Functions help **modularize code**, making it more organized, readable, and easier to maintain.
- Define a function using the `def` keyword followed by the function name and parameters.
- Use indentation to define the block of code within the function.

Functions - example

```python
def add_numbers(x, y):
    """Add two numbers and return the result."""
    result = x + y
    return result

# Calling the function
sum_result = add_numbers(3, 5)
print("Sum:", sum_result)
```

```
Sum: 8
```

What is Flask?

- A lightweight, versatile and extensible web framework for Python.

- Provides essential features and functionality for building web applications, while extensions provide the rest.

- To use Flask, we need to install it by:

```
pip install Flask
```

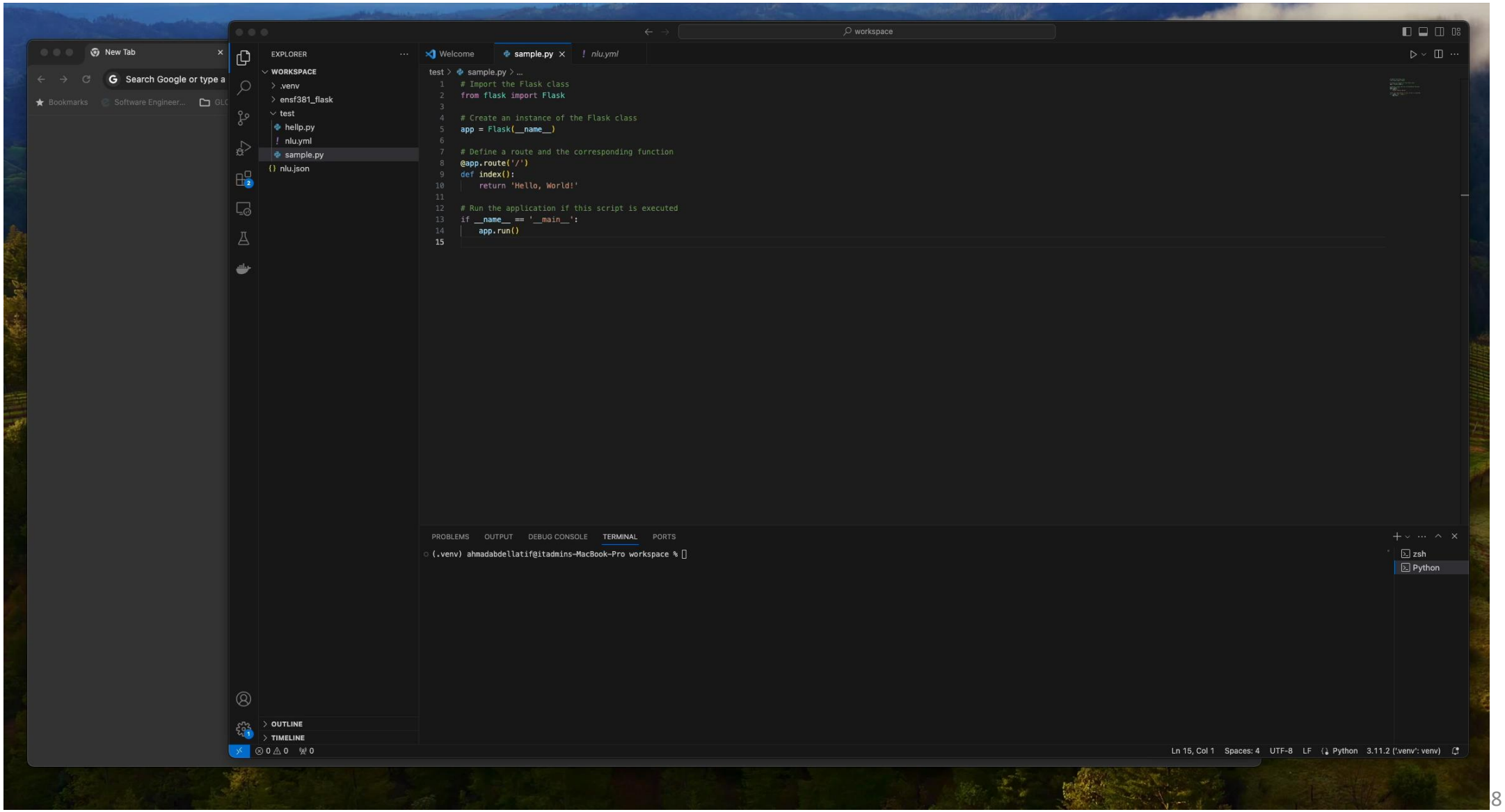
Flask – example

```python
# Import the Flask class
from flask import Flask

# Create an instance of the Flask class
app = Flask(__name__)

# Define a route and the corresponding function
@app.route('/')
def index():
    return 'Hello, World!'

# Run the application if this script is executed
if __name__ == '__main__':
    app.run()
```

1. app instance: create an instance of the Flask class, typically passing __name__ as an argument.

2. @app.route() decorator: associate a URL route with a Python function that corresponds to the route.

Flask – example



What is Git?

- Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

- It was created by Linus Torvalds in 2005 for the development of the Linux kernel.

- Enable multiple developers to work on the same project simultaneously.

Key features of Git

- **Distributed system**: each developer can work on their local copy of the project, make changes, and then synchronize those changes with others.

- **Branching and merging**: makes it easy to create branches for separate lines of development. Changes can be made in isolation, tested, and then merged back into the main project.

- **Commit history**: maintains a detailed record of every change made to the project, providing a comprehensive history. Each change is recorded as a commit.

- **Fast and efficient**: designed to be fast and efficient, making it suitable for both small projects and large-scale, complex software development.

- **Compatibility**: compatible with various operating systems, including Windows, macOS, and Linux.

Basic concepts

- **Repository (Repo)**: is a directory or storage space where your projects can live. It can be local (on your computer) or remote (on a server).

- **Commit**: is a snapshot of changes at a specific point in time. It includes modifications, additions, or deletions of files.

- **Branch**: is an independent line of development. It allows for the creation of isolated environments to work on specific features or fixes without affecting the main project.

- **Merge**: combines changes from different branches into a single branch, often the main branch.

Advantages of distinct repositories

- Clear separation allows for modular development. Frontend and backend can evolve independently, making it easier to maintain and update each component.
- Different technology stacks for frontend and backend can be easily accommodated without cluttering a single repository.
- If different teams or individuals are responsible for frontend and backend, separate repositories offer more autonomy.
- Easier scalability as you can scale frontend and backend independently based on requirements.

Advantages of same repository

- A single repository provides a clear, unified history of changes, making it easier to track the evolution of the entire application.
- Versioning is simplified, and it is easier to ensure that frontend and backend versions are compatible.
- Changes that span both frontend and backend can be managed in a more coordinated manner.
- Developers can clone a single repository and have everything they need to work on the project.

Continuous Integration and Delivery (CI/CD)

- A set of best practices and automated processes in software development.
- Emerged as a response to challenges in traditional software development.
- The growing adoption of agile methods and the increasing demand for faster and more reliable software delivery are accelerating the process.

CI/CD benefits

- **Reduced Manual Errors**: automated processes minimize human errors in tasks like testing and deployment.

- **Continuous Feedback Loop**: automated testing provides prompt feedback on code changes. Developers receive continuous insights, allowing for quick adjustments and improvements.

- **Faster Time-to-Market**: streamlines development workflows, enabling quicker releases, which ensures products reach the market ahead of competitors.

- **Improved Collaboration**: CI/CD fosters collaboration among developers through continuous integration. Early issues resolution swift identification and resolution of integration issues strengthen teamwork.

CI/CD practices

CI/CD comprises the integrated practices of:

1. Continuous Integration (CI).

2. Continuous Delivery (CD).

3. Continuous Deployment.

Semantic Versioning (SemVer)

A standardized way for developers and systems to understand the nature of changes between different versions of a software library or application.

The version number is structured as three numbers separated by dots:

$$\underbrace{10}_{\text{MAJOR version}}.\underbrace{5}_{\text{MINOR version}}.\underbrace{7}_{\text{PATCH version}}$$

Major Version: indicates significant and backward-incompatible changes that may require modifications in the way software interacts with the new release.

Minor version: introduce new features or enhancements while maintaining compatibility with previous versions.

Patch Version: Indicates of issues or bugs resolution without introducing new features or breaking changes.

SemVer

- Example Progression:
  - 1.0.0 (Initial stable release)
  - 1.1.0 (Minor feature additions)
  - 1.1.1 (Patch for bug fix)
  - 2.0.0 (Major release with breaking changes)

- Pre-release Versions:
  - Used for versions in development or testing.
  - Indicated by a hyphen, e.g., 1.0.0-alpha.

- Build Metadata:
  - Identifies specific builds for internal purposes.
  - Indicated by a plus sign, e.g., 1.0.0+build123.

What is Web Application Security?

- The process of securing websites and online services against various threats and vulnerabilities.
- It involves protecting data, users, and systems from unauthorized access, attacks, and misuse.
- Ensures the confidentiality, integrity, and availability of web applications and their data.

Securing your full-stack application is paramount

- **Protecting Sensitive Data:** full-stack applications often handle sensitive user data, such as personal information, financial details, or proprietary business data.

- **Preventing Unauthorized Access:** without proper security measures, attackers can gain unauthorized access to your application, its databases, or backend systems.

- **Maintaining User Trust:** users expect their data to be handled securely by the applications they use.

- **Mitigating Downtime and Damage:** security incidents can lead to downtime, data loss, and damage to your application and infrastructure.

Securing Full Stack Application

- Frontend
- Backend
- Deployment

Hashing

- Hashing is the process of converting input data (such as passwords or other sensitive information) into a fixed-size string of characters, typically through a mathematical algorithm called a hash function.

- Secure Hash Algorithm 256 (SHA-256): SHA-256 is part of the SHA-2 family and is widely used for secure hashing. It produces a 256-bit hash value.
  - Example: e59e31e90a0d68e0d15c866edf167bc4e6f8b6255188122284fda2fc640dee7c

- bcrypt: bcrypt is a password-hashing function designed specifically for secure password storage.
  - Example: $2b$12$luLwZ25eKhbVHeG5n9H7VOu8b76Ddw4N9dU9bue9ZpH4W5Npf9wHm

Questions