Web Development Fundamentals

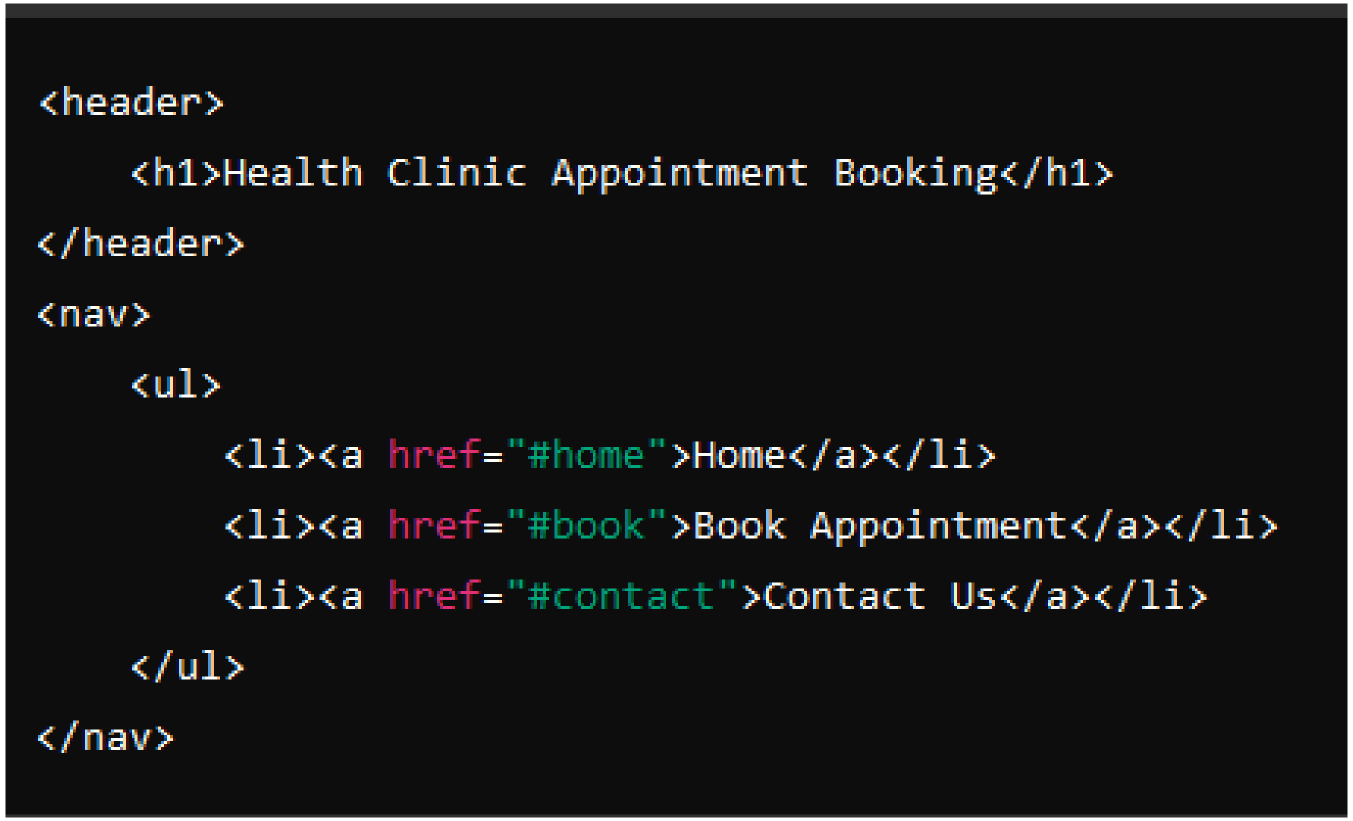
Peter Khisa Wakholi, Ph.D

Content

- Overview of Core Technologies:
  - HTML5: Structure your web pages.
  - CSS3: Style and layout your content.
  - JavaScript ES6+: Add interactivity and logic.
  - Responsive Design: Ensure your web app works on any device.
  - Server-side Programming: Manage backend processes.
  - RESTful APIs: Enable communication between frontend and backend.

The Role of HTML5 in Web Development

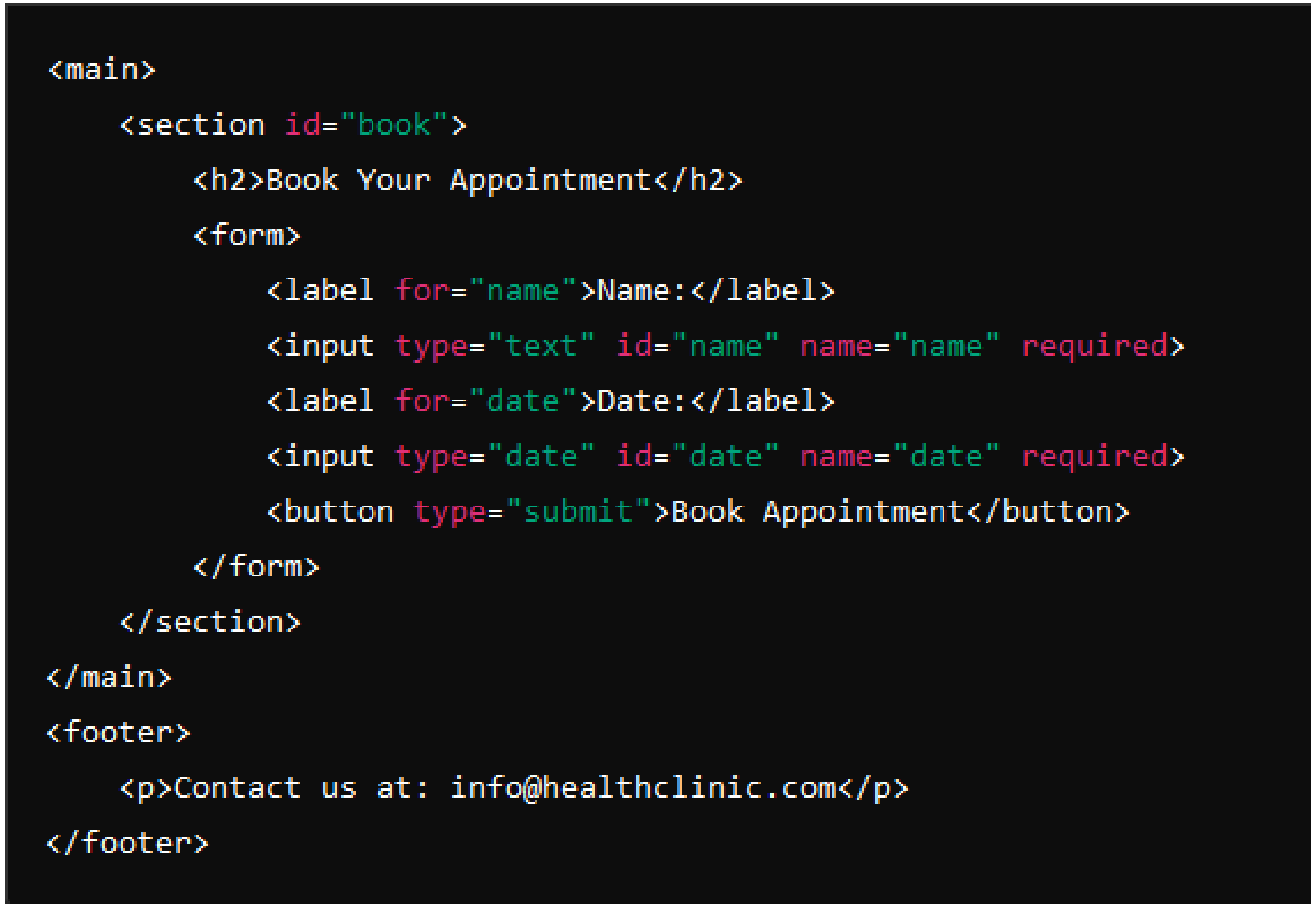
- Key Features of HTML5:
  - Semantic Elements (<header>, <nav>, <article>, <footer>)
  - Multimedia Elements (<audio>, <video>)
  - Form Enhancements (Input types: date, email, number, etc.)



```html
<header>
    <h1>Health Clinic Appointment Booking</h1>
</header>
<nav>
    <ul>
        <li><a href="#home">Home</a></li>
        <li><a href="#book">Book Appointment</a></li>
        <li><a href="#contact">Contact Us</a></li>
    </ul>
</nav>
```

Structuring Web Pages with HTML5

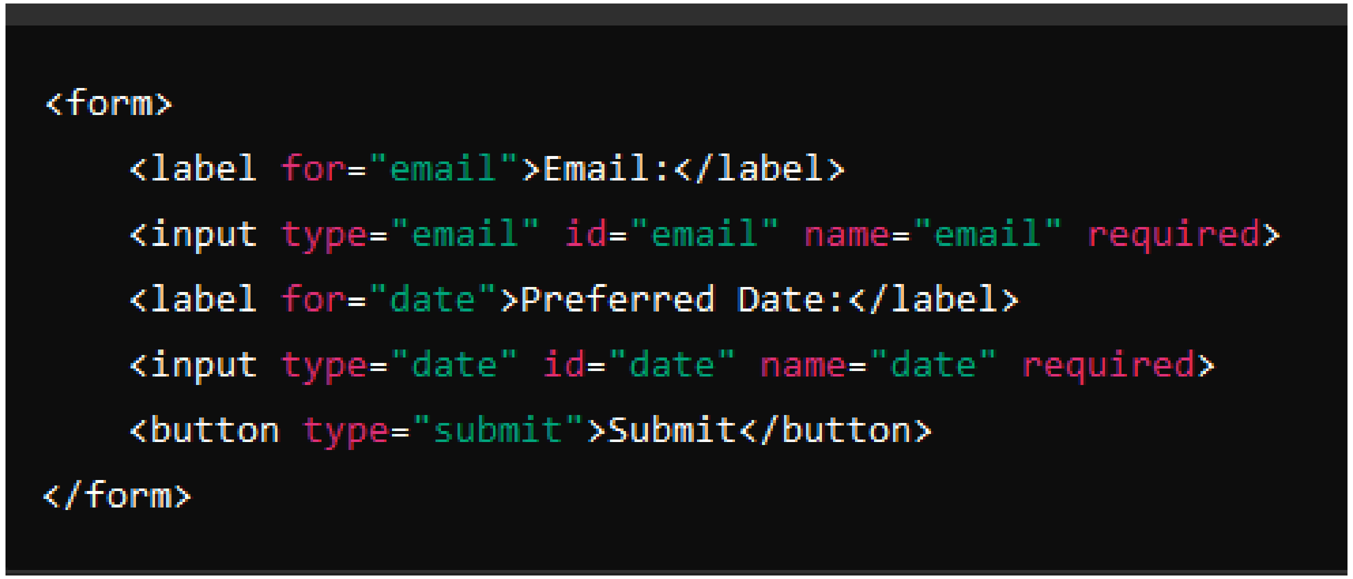
- Page Structure:
  - Header (<header>): Title, logo, navigation.
  - Main Content (<main>): Forms, content sections.
  - Footer (<footer>): Contact information, links.



```html
<main>
    <section id="book">
        <h2>Book Your Appointment</h2>
        <form>
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" required>
            <label for="date">Date:</label>
            <input type="date" id="date" name="date" required>
            <button type="submit">Book Appointment</button>
        </form>
    </section>
</main>
<footer>
    <p>Contact us at: info@healthclinic.com</p>
</footer>
```

HTML5 Forms - Enhancing User Interaction

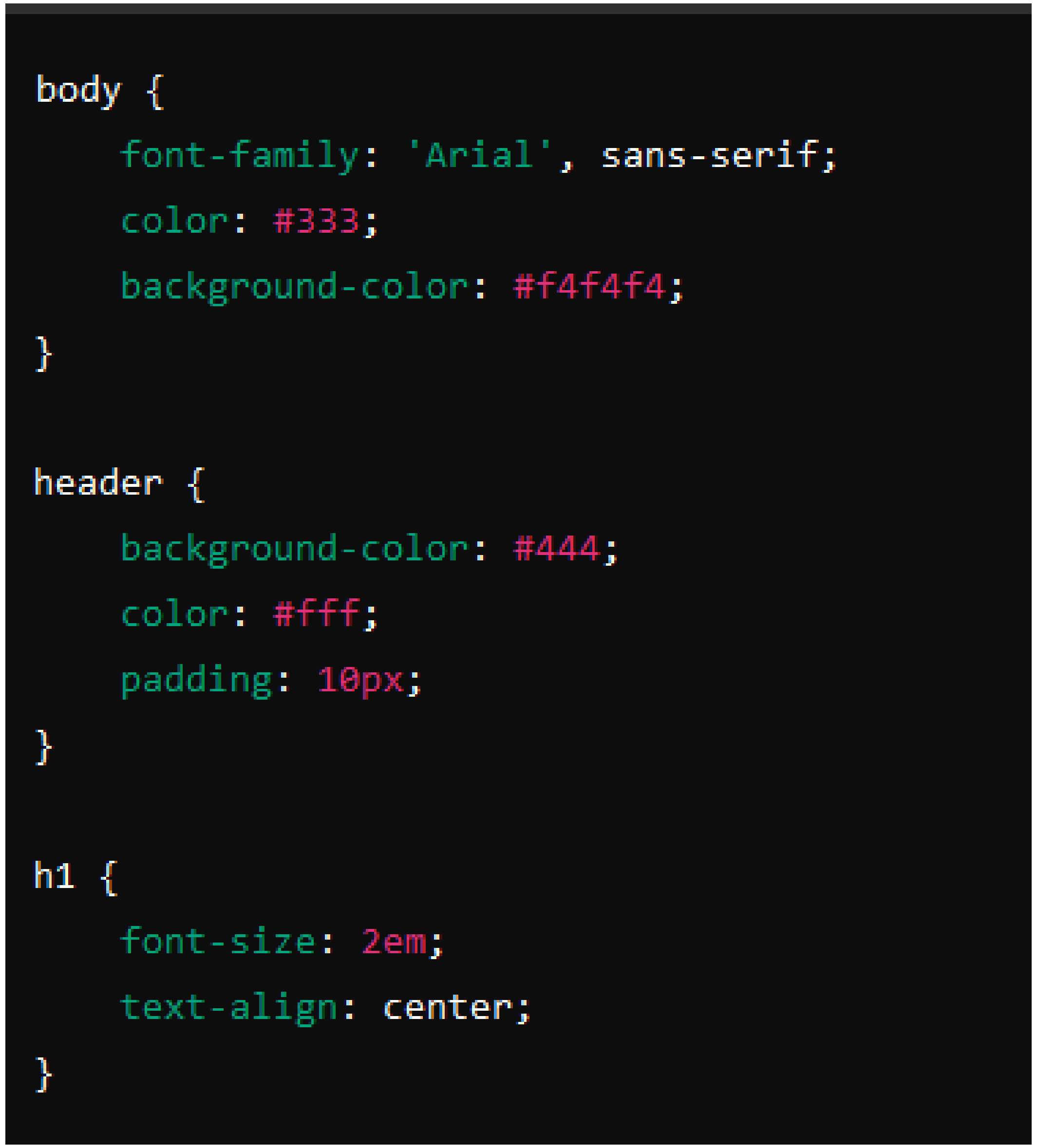
- Form Elements:
- Input Types: text, email, date, number, range.
- Validation Attributes: required, min, max, pattern.
- New HTML5 Elements: datalist, output, progress.



```html
<form>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
    <label for="date">Preferred Date:</label>
    <input type="date" id="date" name="date" required>
    <button type="submit">Submit</button>
</form>
```

Styling with CSS3 - Basics and Beyond

- Core Concepts:
  - Selectors: Element, class, ID selectors.
  - Box Model: Margin, border, padding, content.
  - Styling Text: Fonts, colors, text alignment.



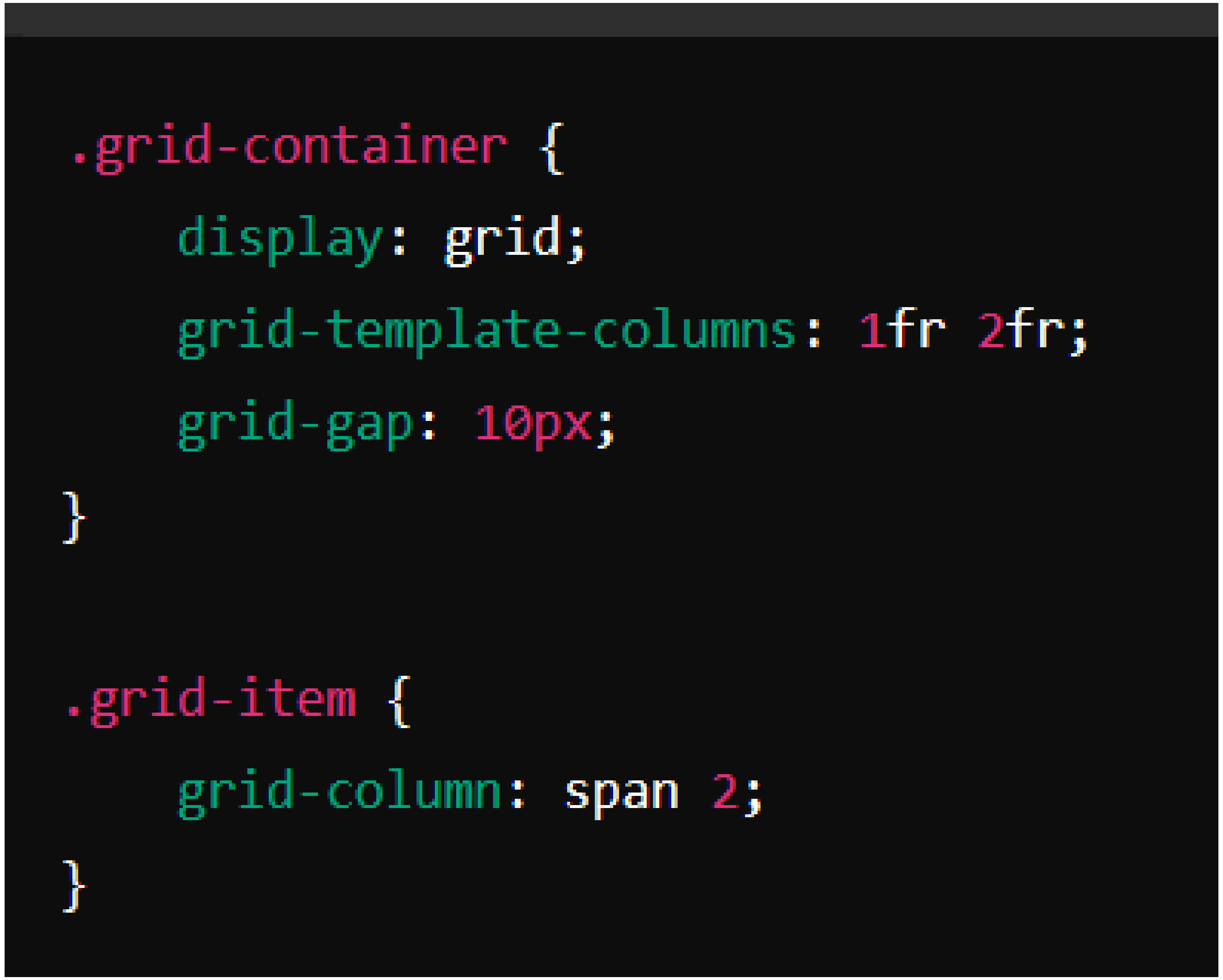
```css
body {
    font-family: 'Arial', sans-serif;
    color: #333;
    background-color: #f4f4f4;
}

header {
    background-color: #444;
    color: #fff;
    padding: 10px;
}

h1 {
    font-size: 2em;
    text-align: center;
}
```

Responsive Design - Principles and Techniques

- Key Principles:
  - Mobile-First Approach.
  - Fluid Grids and Flexible Images.
  - Media Queries.
- Responsive Techniques:
  - Viewport Meta Tag.
  - Scaling Text and Elements.
  - Using Flexbox and Grid for responsive layouts.

Creating Responsive Layouts with CSS Grid

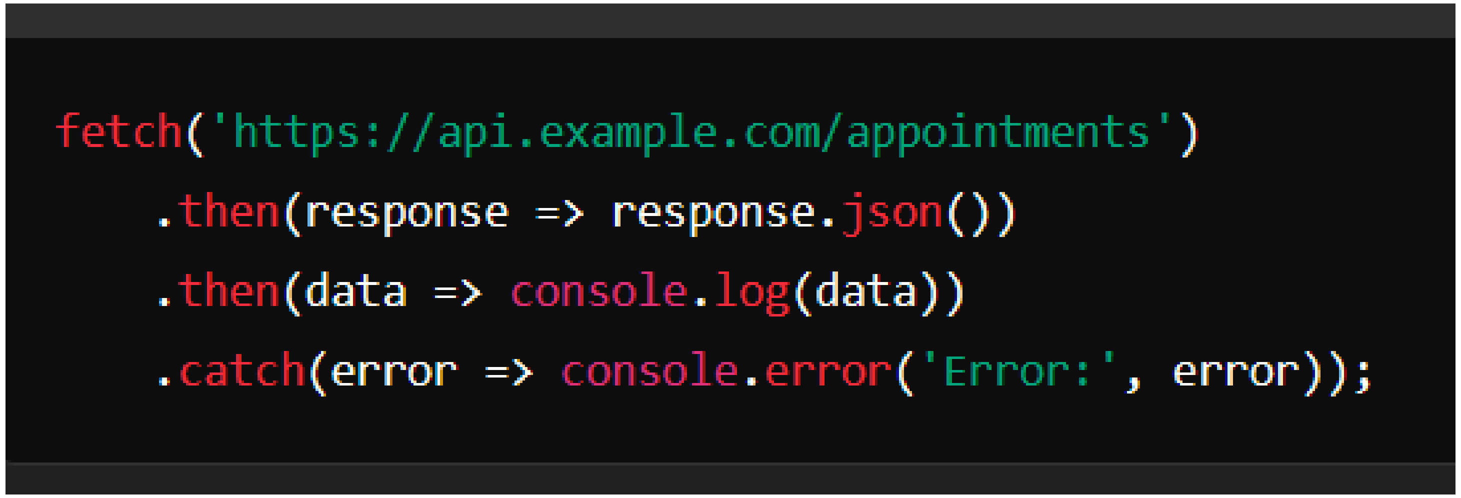
- Defining a Grid:
  - Grid Container, Rows, and Columns.
  - Explicit vs. Implicit Grid.
- Placing Items:
  - Using Grid Lines and Areas.
  - Spanning Rows and Columns.



```
.grid-container {
    display: grid;
    grid-template-columns: 1fr 2fr;
    grid-gap: 10px;
}

.grid-item {
    grid-column: span 2;
}
```

Using Fetch API for HTTP Requests

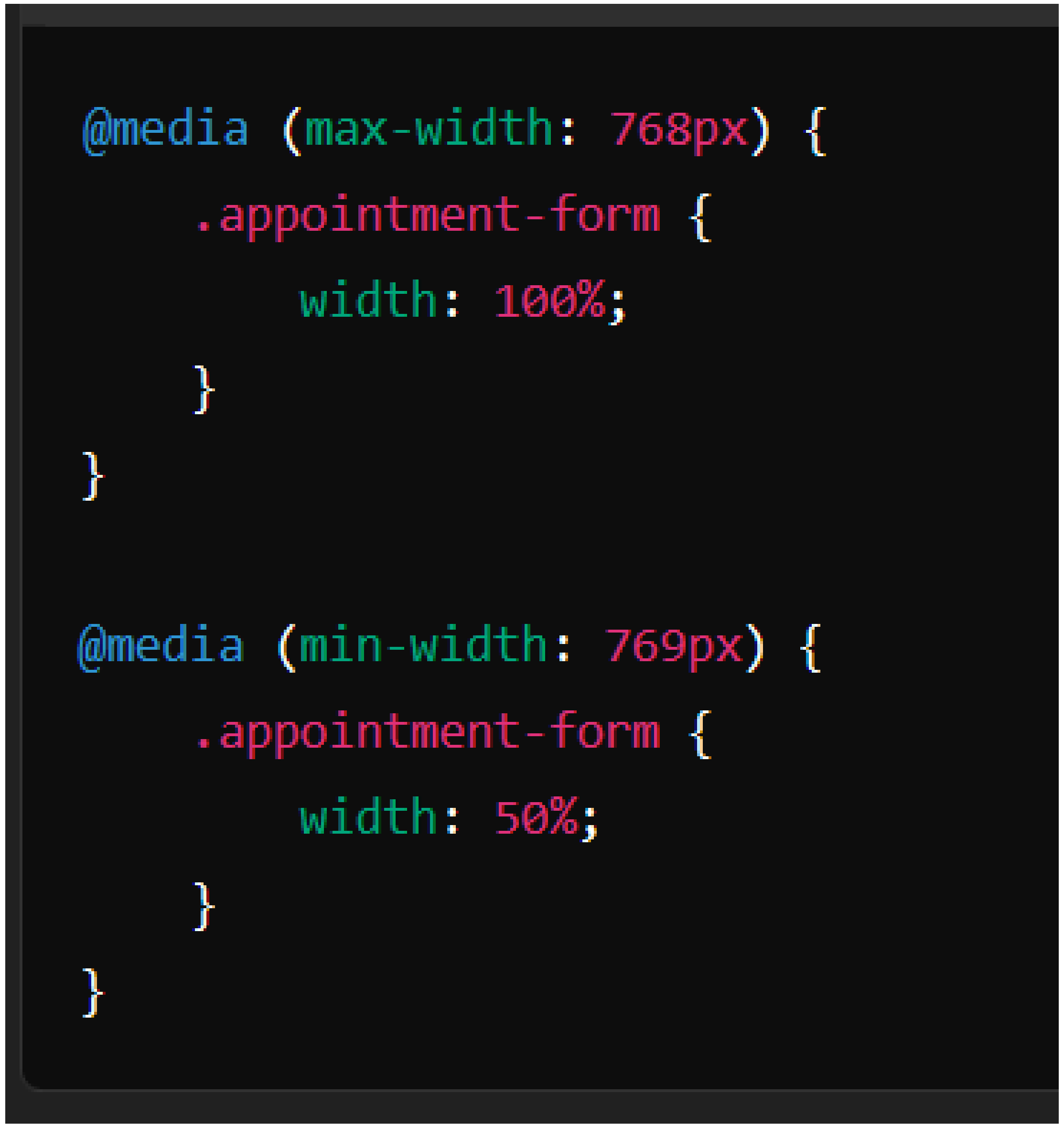
- Key Concepts:
  - Making GET and POST Requests.
  - Handling Responses and Errors.
  - Parsing JSON Data.



```
fetch('https://api.example.com/appointments')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

Responsive Design - Media Queries

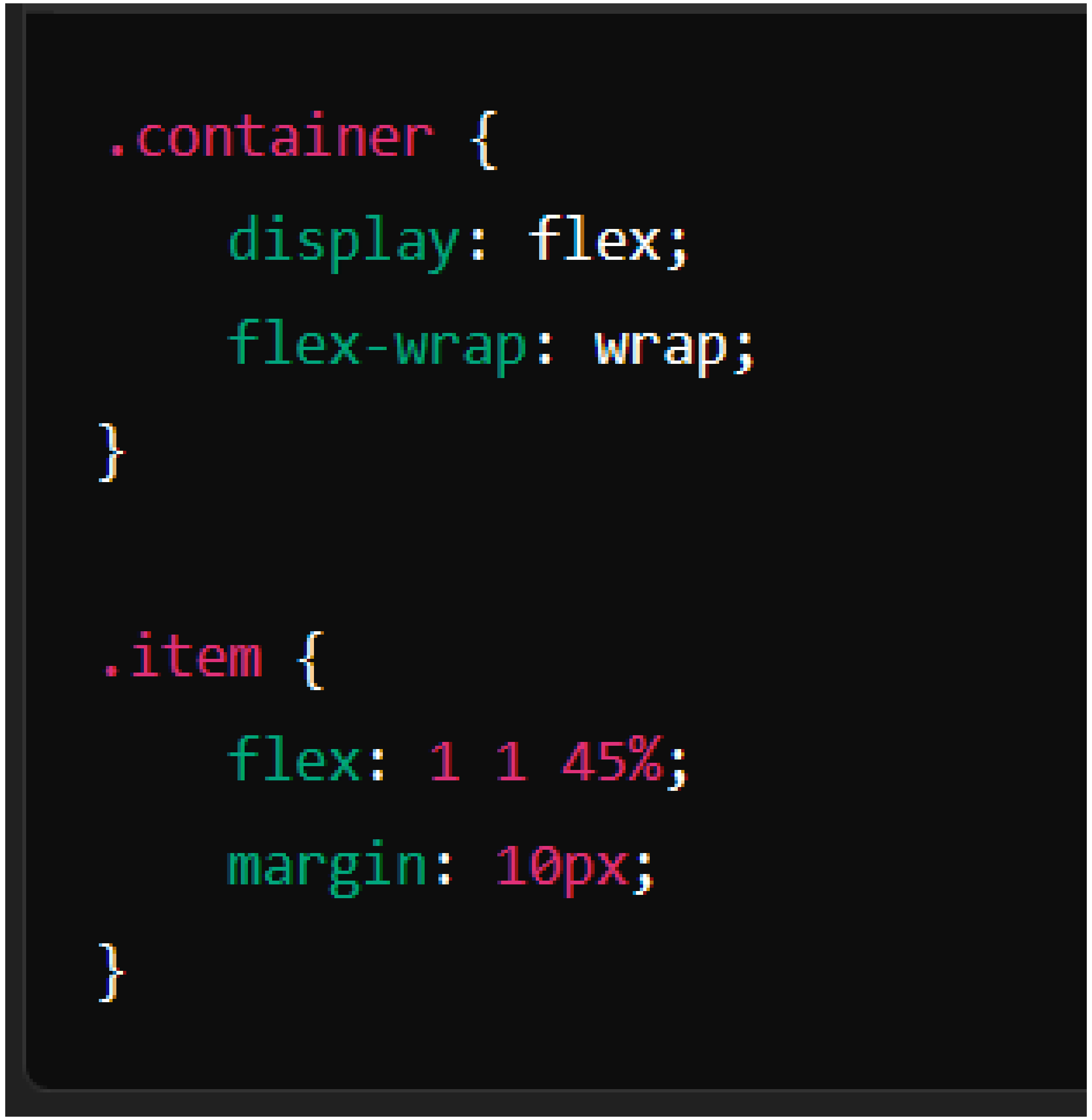
- Defining Media Queries:
    - Syntax and Usage.
    - Targeting Different Devices.
- Examples:
    - Mobile-First Approach.
    - Adjusting Layouts Based on Screen Size.



```
@media (max-width: 768px) {
    .appointment-form {
        width: 100%;
    }
}

@media (min-width: 769px) {
    .appointment-form {
        width: 50%;
    }
}
```

Flexbox and Grid for Responsive Design

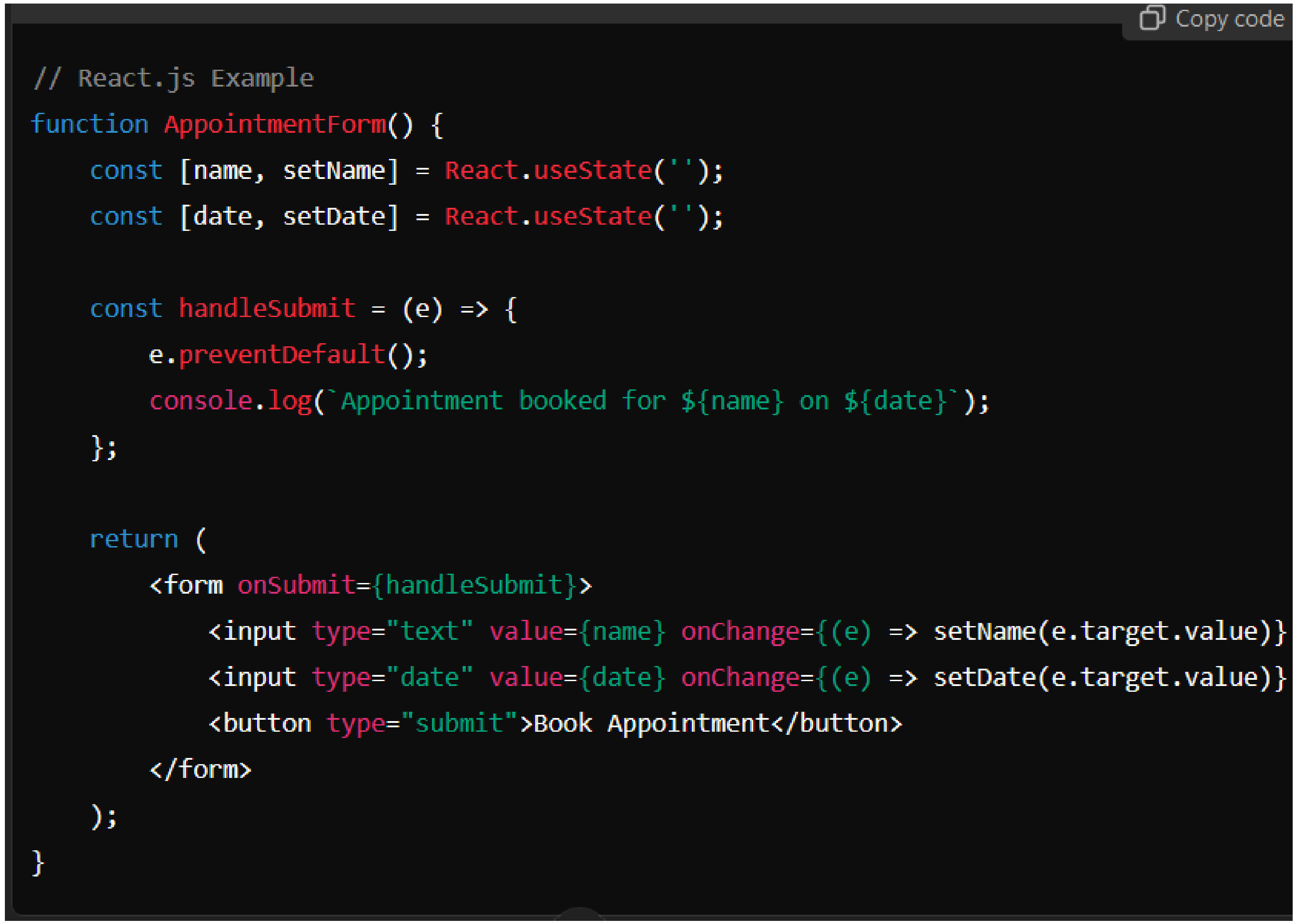
- Flexbox:
  - Aligning and Distributing Space in a Container.
  - Flexbox vs. CSS Grid.
- Grid Layout:
  - Defining Grids with Media Queries.
  - Creating Responsive Layouts with Grid.



```css
.container {
    display: flex;
    flex-wrap: wrap;
}

.item {
    flex: 1 1 45%;
    margin: 10px;
}
```

Front-End Frameworks

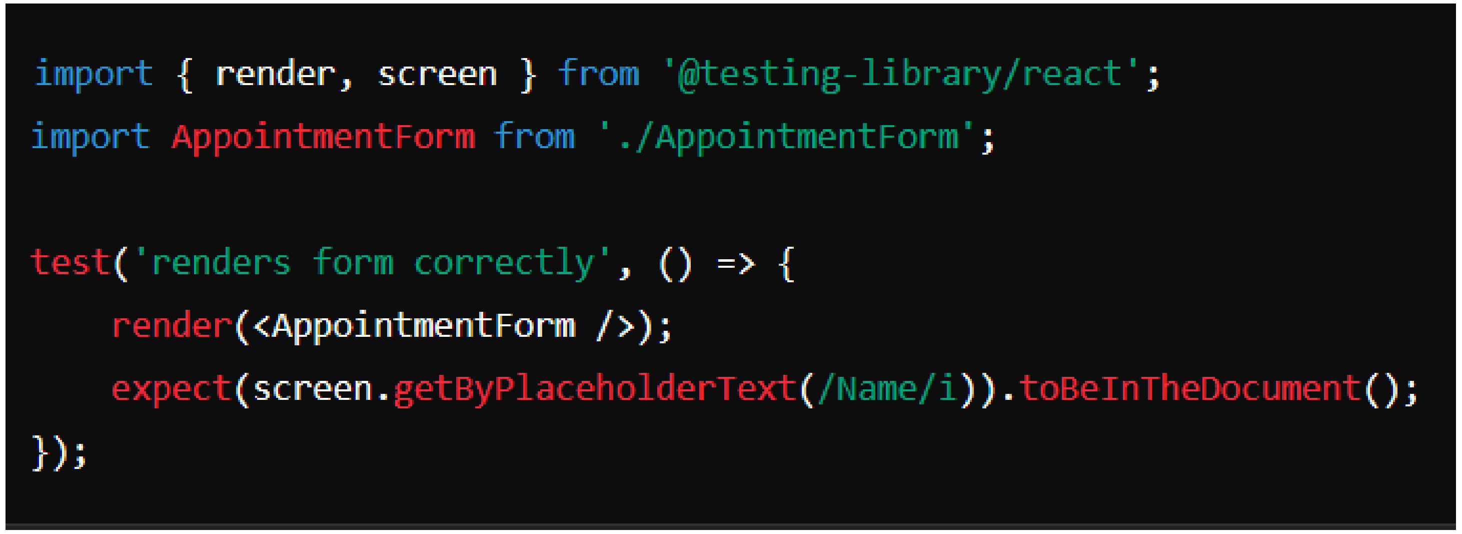
- Popular Frameworks:
  - React.js: Component-Based UI Development.
  - Angular: Full-Featured Framework for Web Applications.
  - Vue.js: Progressive Framework for Building UIs.
- Key Features:
  - Component Architecture.
  - State Management.
  - Virtual DOM (for React.js and Vue.js).



```
// React.js Example
function AppointmentForm() {
    const [name, setName] = React.useState('');
    const [date, setDate] = React.useState('');

    const handleSubmit = (e) => {
        e.preventDefault();
        console.log(`Appointment booked for ${name} on ${date}`);
    };

    return (
        <form onSubmit={handleSubmit}>
            <input type="text" value={name} onChange={(e) => setName(e.target.value)}
            <input type="date" value={date} onChange={(e) => setDate(e.target.value)}
            <button type="submit">Book Appointment</button>
        </form>
    );
}
```

Testing React Components

- Testing Tools:
  - Jest for Unit Testing.
  - Enzyme for Component Testing.
  - React Testing Library.
- Writing Test Cases:
  - Testing Component Rendering.
  - Simulating User Events.



```
import { render, screen } from '@testing-library/react';
import AppointmentForm from './AppointmentForm';

test('renders form correctly', () => {
    render(<AppointmentForm />);
    expect(screen.getByPlaceholderText(/Name/i)).toBeInTheDocument();
});
```

Introduction to Web Application Security

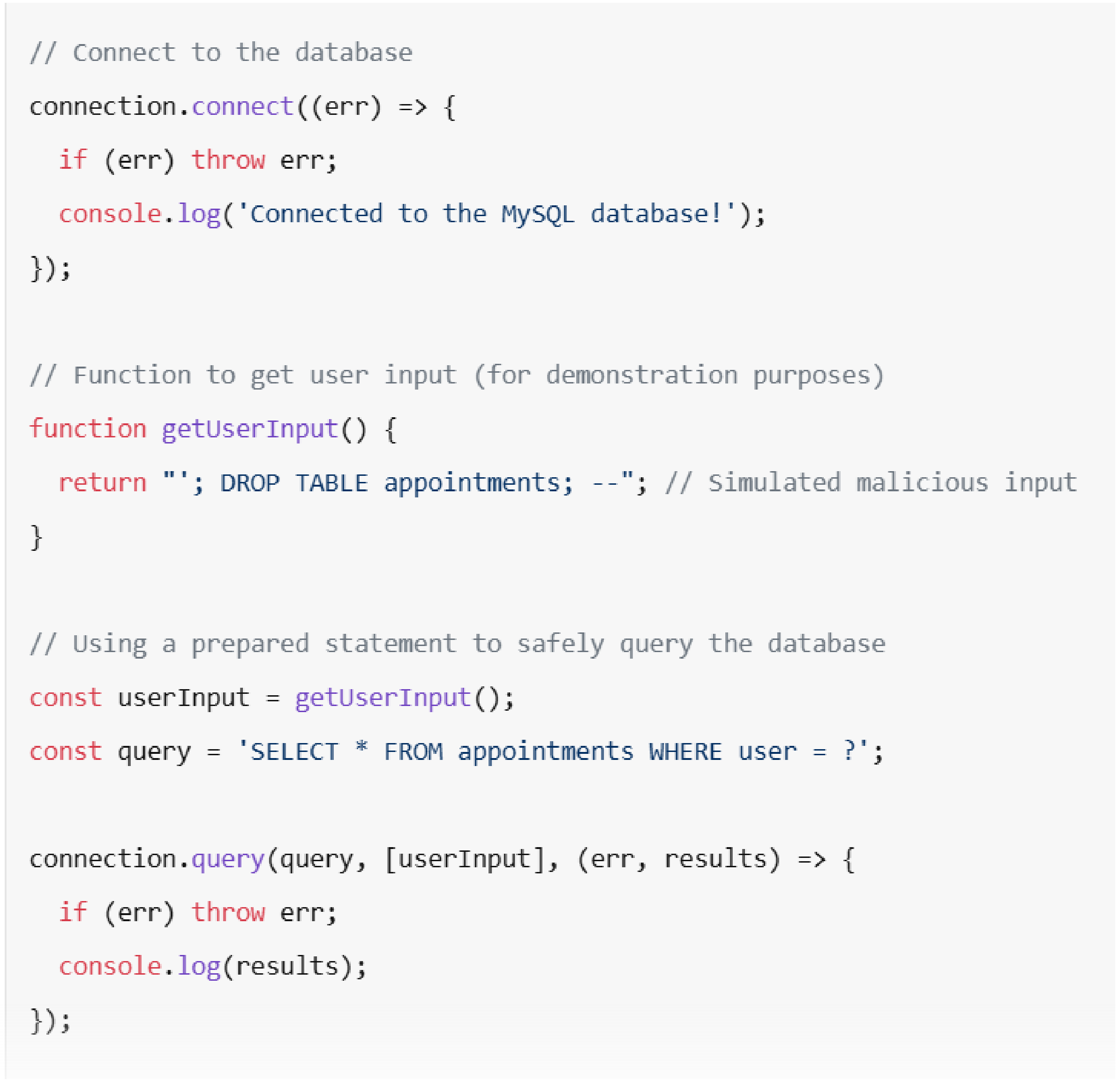
- **Security Importance:** Web applications, especially those handling sensitive data (e.g., patient information in a health clinic), must be secure to protect against attacks and data breaches.
- **Common Threats:** Discuss the most common security threats that web applications face, particularly SQL Injection, XSS, and CSRF.

Key Security Concerns -

Concept: SQL Injection occurs when an attacker is able to execute arbitrary SQL queries on the database through user input. For example, a malicious user could enter SQL code into a form field that interacts with the database.

Example: If an appointment booking form allows for unvalidated user input, an attacker could input something like '; DROP TABLE appointments; --, which could delete the entire appointments table if the input is directly inserted into an SQL query.

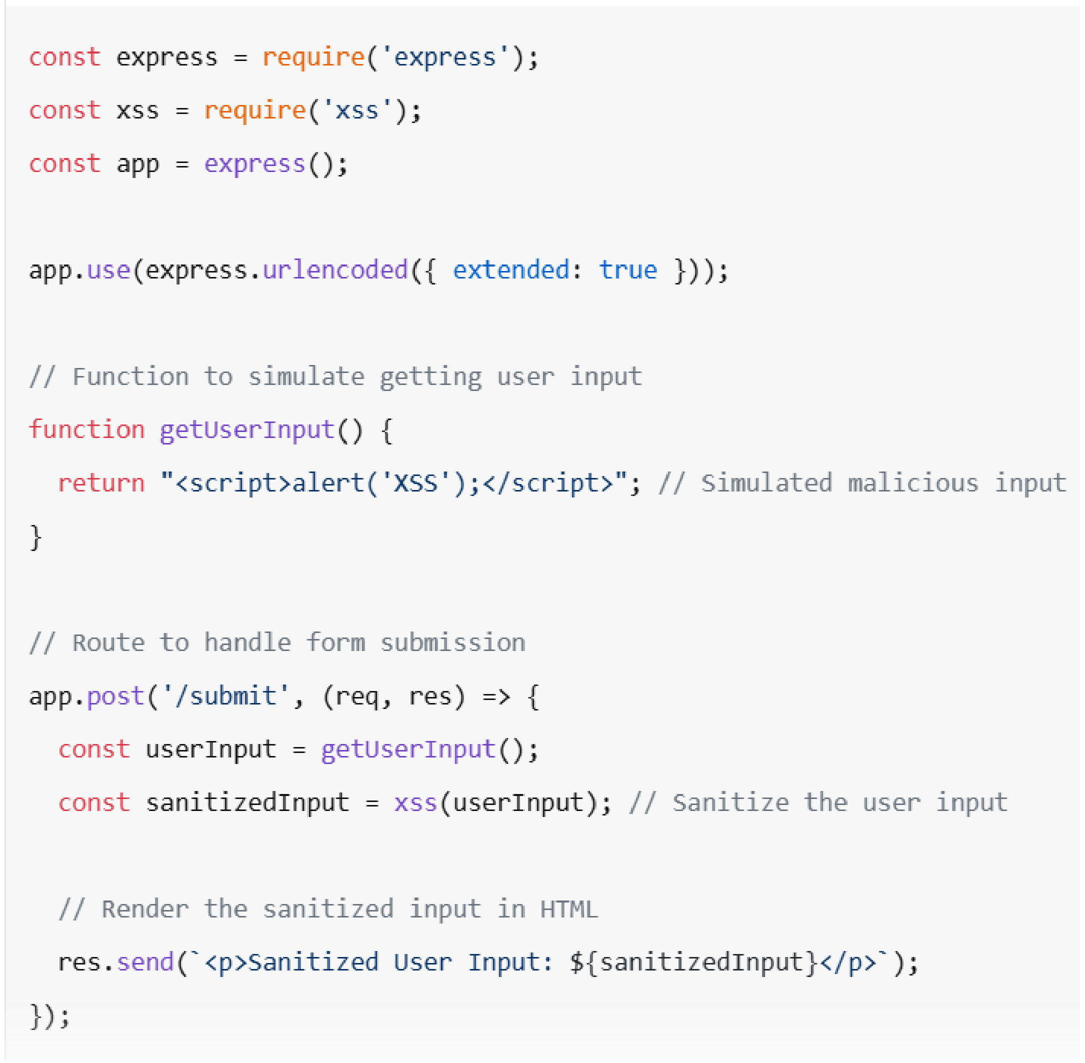
Mitigation: Always use prepared statements and parameterized queries to prevent SQL injection.



```
// Connect to the database
connection.connect((err) => {
  if (err) throw err;
  console.log('Connected to the MySQL database!');
});

// Function to get user input (for demonstration purposes)
function getUserInput() {
  return "'; DROP TABLE appointments; --"; // Simulated malicious input
}

// Using a prepared statement to safely query the database
const userInput = getUserInput();
const query = 'SELECT * FROM appointments WHERE user = ?';

connection.query(query, [userInput], (err, results) => {
  if (err) throw err;
  console.log(results);
});
```

Cross-Site Scripting (XSS):

- **Concept:** XSS occurs when an attacker injects malicious scripts into content that is later served to other users. This can lead to session hijacking, defacement of websites, or redirection to malicious sites.
- **Example:** In the context of your appointment booking app, an attacker could enter a script in the "name" field of a form, which, if not sanitized, could be executed when viewed by a staff member.
- **Mitigation:** Always escape and sanitize user input, especially before rendering it in HTML.



```
const express = require('express');
const xss = require('xss');
const app = express();

app.use(express.urlencoded({ extended: true }));

// Function to simulate getting user input
function getUserInput() {
  return "<script>alert('XSS');</script>"; // Simulated malicious input
}

// Route to handle form submission
app.post('/submit', (req, res) => {
  const userInput = getUserInput();
  const sanitizedInput = xss(userInput); // Sanitize the user input

  // Render the sanitized input in HTML
  res.send(`<p>Sanitized User Input: ${sanitizedInput}</p>`);
});
```

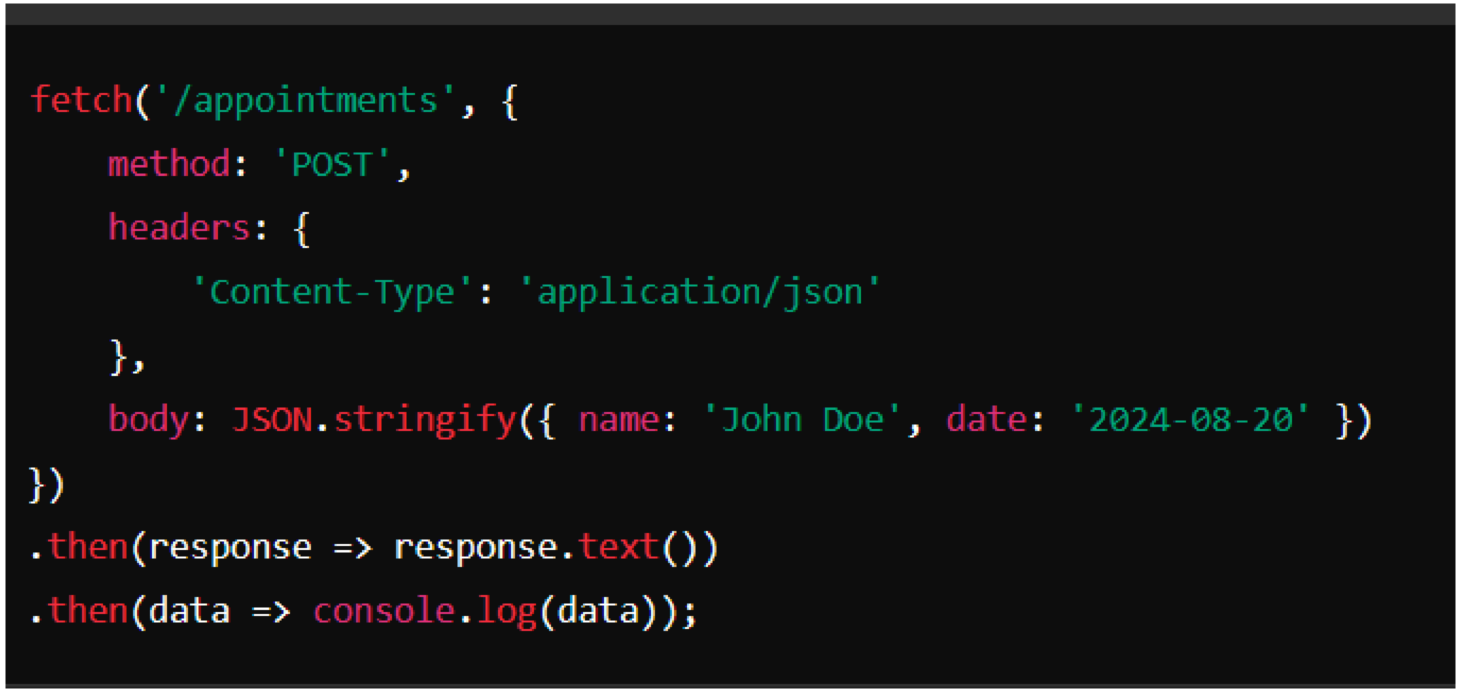
Cross-Site Request Forgery (CSRF)

- **Concept:** CSRF tricks the user's browser into making unwanted requests to a site they are authenticated on, potentially performing unauthorized actions like changing user settings or making a fraudulent booking.
- **Example**: A user might be tricked into clicking on a hidden form submission button on a different site, which submits a request to your appointment booking app using their session.
- **Mitigation:** Use anti-CSRF tokens and ensure that critical operations require the user to re-authenticate or confirm their action.

```
// Setup CSRF protection middleware
const csrfProtection = csurf({ cookie: true });

// Route to display the form
app.get('/form', csrfProtection, (req, res) => {
  // Send the form with the CSRF token
  res.send(`
    <form action="/submit" method="POST">
      <input type="hidden" name="_csrf" value="${req.csrfToken()}">
      <input type="text" name="name" placeholder="Enter your name">
      <button type="submit">Submit</button>
    </form>
  `);
});

// Route to handle form submission
app.post('/submit', csrfProtection, (req, res) => {
  // Process the form submission
  res.send(`Form submitted successfully! Name: ${req.body.name}`);
});
```

Connecting Frontend and Backend with RESTful APIs

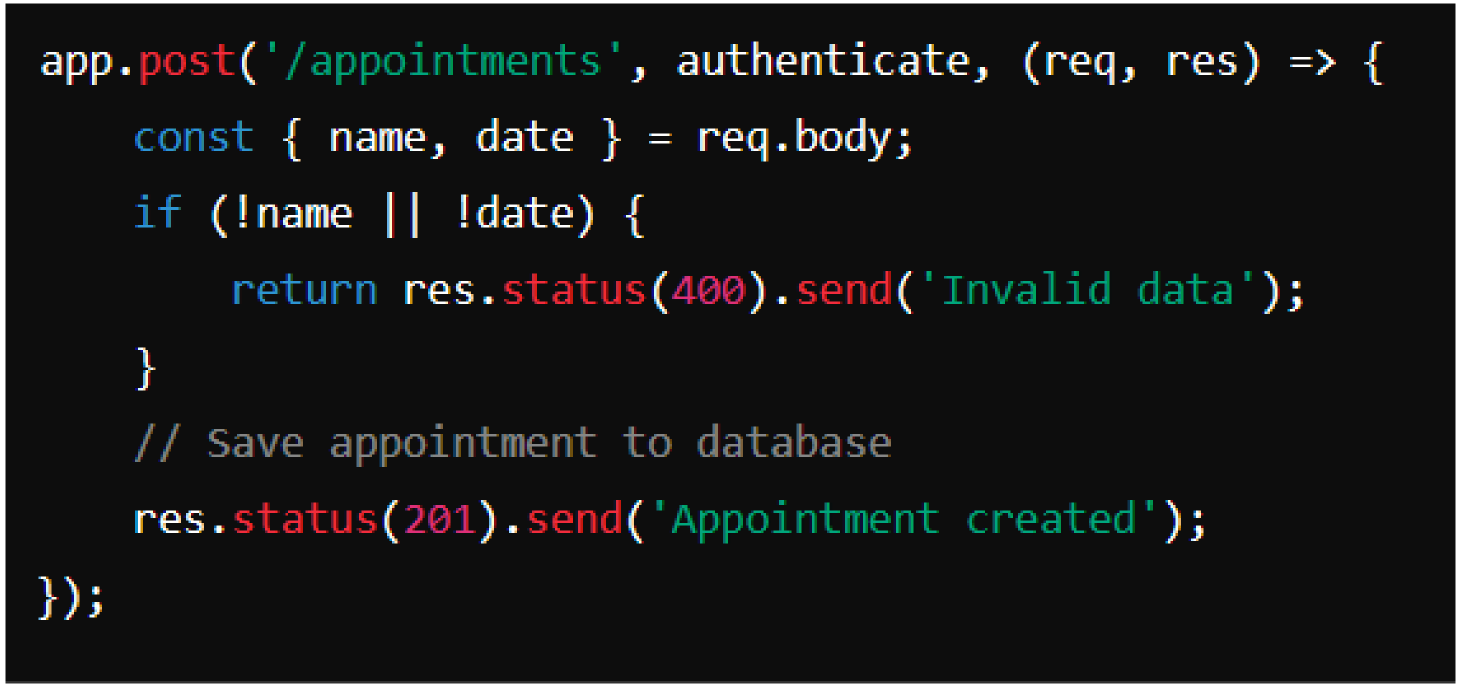
- RESTful API Basics:
    - HTTP Methods: GET, POST, PUT, DELETE.
    - RESTful Routes: /appointments, /patients.
- Making API Calls from the Frontend:
    - Using Fetch API to Interact with Backend.
    - Handling JSON Responses.



```
fetch('/appointments', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({ name: 'John Doe', date: '2024-08-20' })
})
.then(response => response.text())
.then(data => console.log(data));
```

Building RESTful APIs - Best Practices

- RESTful API Design Principles:
    - Use of HTTP Methods and Status Codes.
    - Resource Naming Conventions.
    - Versioning APIs.
- Security Considerations:
    - Implementing Authentication and Authorization.
    - Using HTTPS.
    - Rate Limiting and Throttling.



```
app.post('/appointments', authenticate, (req, res) => {
    const { name, date } = req.body;
    if (!name || !date) {
        return res.status(400).send('Invalid data');
    }
    // Save appointment to database
    res.status(201).send('Appointment created');
});
```

Introduction to RESTful API Best Practices

- RESTful APIs are a crucial part of modern web applications, allowing different systems to communicate. Following best practices ensures that your API is efficient, secure, and easy to use.
- There are essential principles for designing robust RESTful APIs and the security considerations to protect them from abuse.

RESTful API Design Principles

- **Use of HTTP Methods and Status Codes:**
  - **HTTP Methods:**
    - **GET:** Retrieve data from the server (e.g., fetching appointment details).
    - **POST:** Submit data to the server (e.g., creating a new appointment).
    - **PUT/PATCH:** Update existing data on the server (e.g., rescheduling an appointment).
    - **DELETE:** Remove data from the server (e.g., canceling an appointment).
  - **Status Codes:**
    - **200 OK:** The request was successful.
    - **201 Created:** A new resource has been created (e.g., a new appointment).
    - **400 Bad Request:** The server could not understand the request due to invalid syntax.
    - **401 Unauthorized:** Authentication is required and has failed or has not been provided.
    - **404 Not Found:** The requested resource could not be found.
    - **500 Internal Server Error:** The server encountered an unexpected condition.

Resource Naming Conventions

- **Consistency:** Use plural nouns for resource names (e.g., /appointments, /patients).
- **Hierarchical Structure:** Organize resources in a way that reflects their relationships (e.g., /patients/{patientId}/appointments to list a patient’s appointments).
- **Example:**
  - **GET** /appointments: Fetch all appointments.
  - **GET** /appointments/{id}: Fetch a specific appointment by ID.
  - **POST** /appointments: Create a new appointment.

Security Considerations

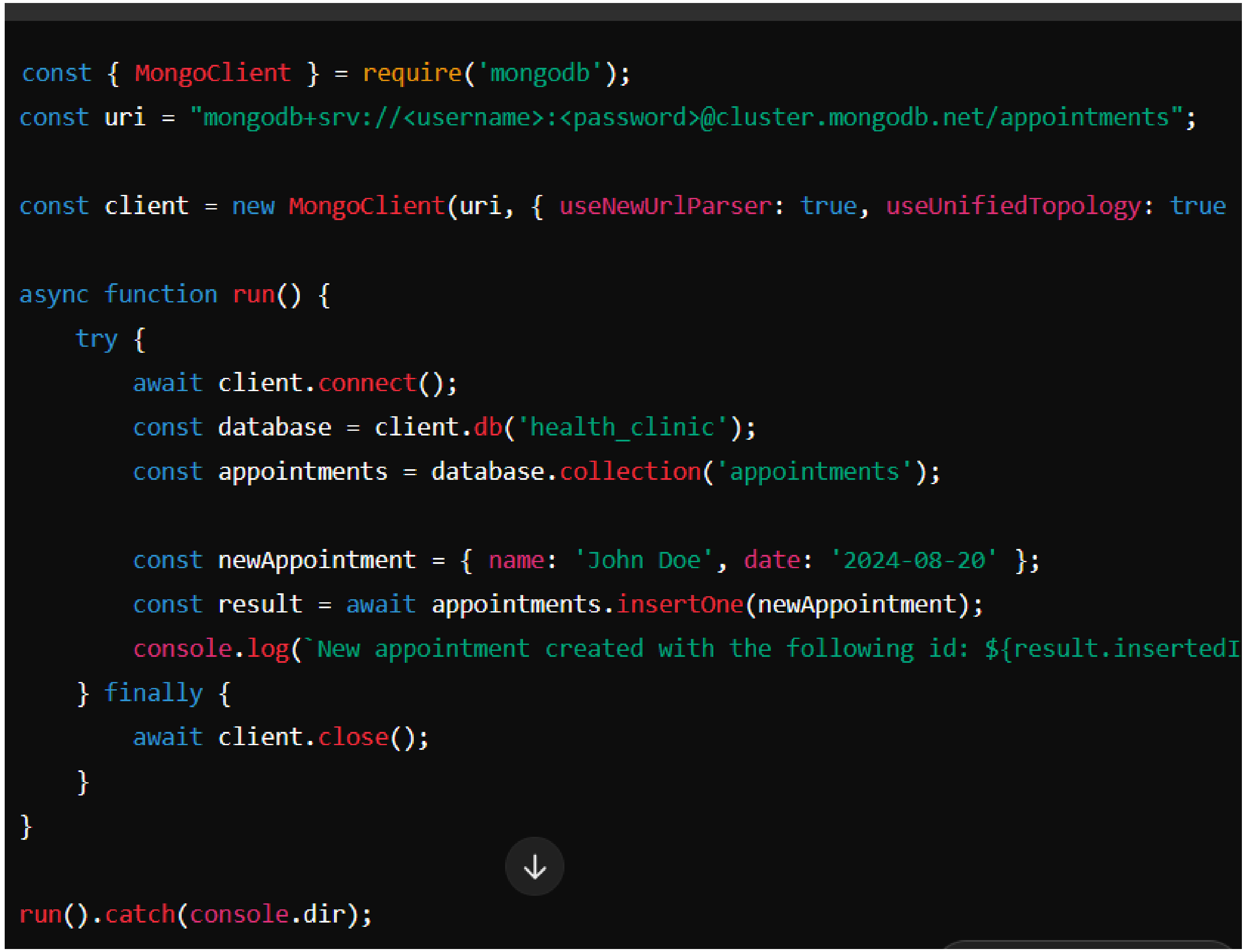
- **Implementing Authentication and Authorization:**
  - **Authentication:** Verify the identity of the user (e.g., using JWT tokens).
  - **Authorization:** Determine what actions an authenticated user is allowed to perform (e.g., only a doctor can modify an appointment).
  - **Example:** Use middleware in Express.js to verify JWT tokens and protect routes.
- **Using HTTPS:**
  - **Why HTTPS:** Encrypts data in transit to protect against eavesdropping and man-in-the-middle attacks.
  - **Implementation:** Ensure your API is served over HTTPS by obtaining and configuring an SSL/TLS certificate.
- **Rate Limiting and Throttling:**
  - **Rate Limiting:** Restrict the number of requests a client can make in a given time period to prevent abuse (e.g., DDoS attacks).
  - **Throttling:** Slow down the rate at which requests are processed to prevent server overload.
  - **Example:** Use middleware like express-rate-limit in Node.js to implement rate limiting.

Versioning APIs

- **Importance:** API versioning allows you to make changes to your API without breaking existing clients.
- **Methods:**
  - **URI Versioning:** Include the version number in the URL (e.g., /v1/appointments).
  - **Header Versioning:** Specify the version in the request header.
- **Best Practice:** Start versioning early to avoid complications later as your API evolves.

Introduction to MongoDB for Data Storage

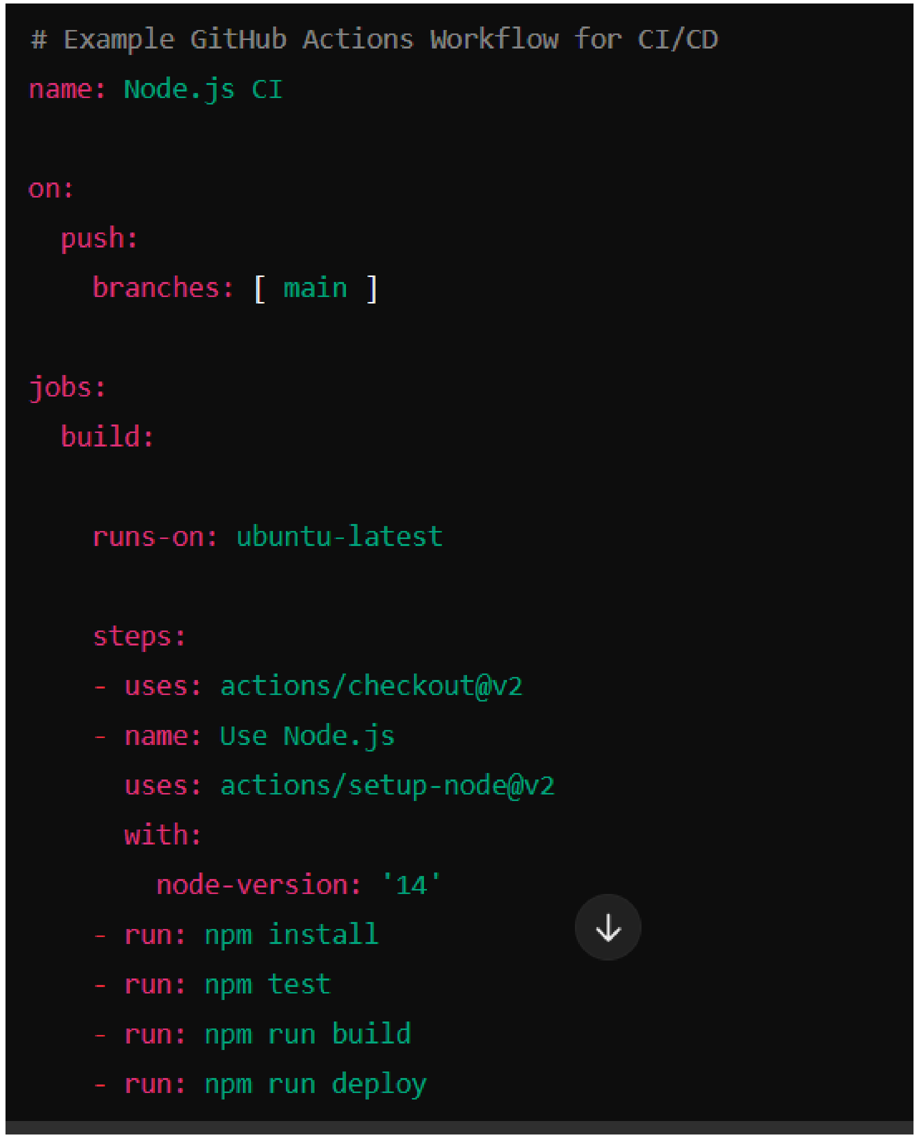
- NoSQL Database Concepts:
  - Document-Oriented Database.
  - Collections and Documents.
  - Schema-Less Data Model.
- Setting Up MongoDB:
  - Installing and Connecting to MongoDB.
  - Basic CRUD Operations.



```
const { MongoClient } = require('mongodb');
const uri = "mongodb+srv://<username>:<password>@cluster.mongodb.net/appointments";

const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true

async function run() {
    try {
        await client.connect();
        const database = client.db('health_clinic');
        const appointments = database.collection('appointments');

        const newAppointment = { name: 'John Doe', date: '2024-08-20' };
        const result = await appointments.insertOne(newAppointment);
        console.log(`New appointment created with the following id: ${result.insertedI
    } finally {
        await client.close();
    }
}

run().catch(console.dir);
```

Deployment Considerations for Web Applications

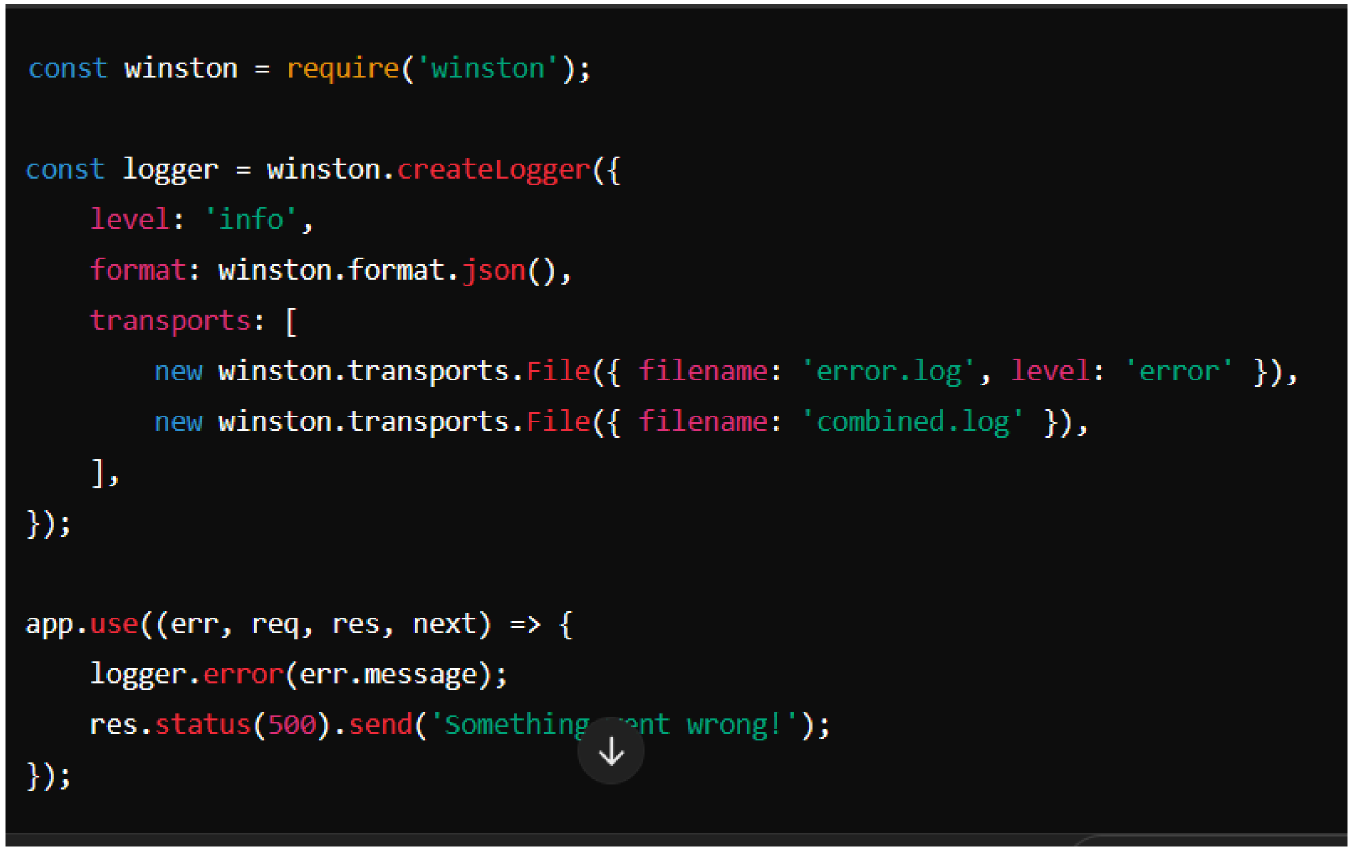
- Deployment Platforms:
  - Heroku, AWS, Netlify, Vercel.
  - Differences Between PaaS and IaaS.
- Deployment Steps:
  - Preparing Your Application for Deployment.
  - Setting Up Environment Variables.
  - Continuous Integration and Deployment (CI/CD) Pipelines.



```yaml
# Example GitHub Actions Workflow for CI/CD
name: Node.js CI

on:
  push:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '14'
    - run: npm install
    - run: npm test
    - run: npm run build
    - run: npm run deploy
```

Monitoring and Maintaining Web Applications

- Key Areas of Monitoring:
  - Application Performance.
  - Error Tracking and Logging.
  - User Analytics.
- Tools for Monitoring:
  - New Relic, Datadog, Sentry.
  - Logging with Winston in Node.js.



```
const winston = require('winston');

const logger = winston.createLogger({
    level: 'info',
    format: winston.format.json(),
    transports: [
        new winston.transports.File({ filename: 'error.log', level: 'error' }),
        new winston.transports.File({ filename: 'combined.log' }),
    ],
});

app.use((err, req, res, next) => {
    logger.error(err.message);
    res.status(500).send('Something went wrong!');
});
```

Recap and Next Steps

- Key Takeaways:
  - Understanding the Full Web Development Stack.
  - Building and Styling Responsive Web Pages.
  - Implementing Frontend Interactivity with JavaScript.
  - Developing Backend Services with Node.js and Express.js.
  - Connecting Frontend and Backend via RESTful APIs.
  - Deploying and Maintaining Web Applications.
- Next Steps:
  - Start Building Your Own Appointment Booking App.
  - Explore Advanced Topics: Authentication, Performance Optimization, Advanced State Management.
  - Continue Learning: "Learning Web Design" by Jennifer Niederst Robbins, "Eloquent JavaScript" by Marijn