RQF LEVEL 4

NITIW401

NETWORKING AND INTERNET TECHNOLOGIES

IoT Web Application Development Using PHP

TRAINEE'S MANUAL

October, 2024

IoT WEB APPLICATION DEVELOPMENT USING PHP

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate RQF Level IV in Networking and Internet Technology, specifically for the module "**NITIW401: IoT Web Application Development Using PHP**".

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

Production Team

Authoring and Review

BAYISENGE Jean Bosco

UMULISA Gaston

Validation

UWIMBABAZI Phénias

KWIZERA Emmanuel

HABUKUBAHO Gad

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

BIZIMANA Eric

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

MANIRAKORA Alexis

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

AUTHOR'S NOTE PAGE (COPYRIGHT)---iii

ACKNOWLEDGEMENTS---iv

TABLE OF CONTENT ---vii

ACRONYMS---viii

INTRODUCTION ---1

MODULE CODE AND TITLE: NITIW401 IoT WEB APPLICATION DEVELOPMENT ---2

Learning Outcome 1: Develop API using PHP ---3

Key Competencies for Learning Outcome 1: Develop API using PHP ---4

Indicative content 1.1: Analysis of IoT Web Application Requirements. ---6

Indicative content 1.2: Prepare PHP Environment---19

Indicative content 1.3: Identification of PHP Concepts. ---27

Indicative content 1.4: Develop Application Programming Interface CRUD Endpoints.---53

Indicative content 1.5: Secure API Endpoints. ---96

Indicative content 1.6: Test Application Programming Interface Endpoint.--- 112

Indicative content 1.7: Documentation of the API.--- 122

Learning outcome 1 end assessment --- 129

References --- 132

Learning Outcome 2: Develop User Interface --- 133

Key Competencies for Learning Outcome 2: Develop User Interface. --- 134

Indicative content 2.1: Identification of UI Requirements --- 136

Indicative content 2.2: Use HTML Tags --- 151

Indicative content 2.3: Application of CSS --- 206

Learning outcome 2 end assessment --- 236

References--- 238

Learning Outcome 3: Integrate API Endpoint with user interface --- 239

Key Competencies for Learning Outcome 1 : Integrate API Endpoints with User Interface. ------- 240

Indicative content 3.1: Interpret API Endpoint --- 242

Indicative content 3.2: Use API data on User Interface--- 255

Indicative content 3.3: Document IoT Web Application--- 280

Learning outcome 3 end assessment --- 293

References--- 297

ACRONYMS

2FA: Two-Factor Authentication

API: Application Programming Interface

CBT/A: Competency-Based Training and Assessment

CD: Continuous Delivery

CORS: Cross-Origin Resource Sharing

CRUD: Create Read Update Delete

CSS: Cascading Style Sheets

HTML: Hypertext Mark-up Language

HTTP: Hypertext Transfer Protocol

IDE: Integrated Development Environment

IoT: Internet of Things

JSON: JavaScript Object Notation

KOICA: Korea International Cooperation Agency

PDO: PHP Data Object

PHP: Hypertext Pre-processor

RBAC: Role Based Access Control

REST: Representational State Transfer

RQF: Regulated Qualifications Framework

RTB: Rwanda TVET Board

SOAP: Simple Object Access Protocol

SQL: Structured Query Language

TQUM Project: TVET Quality Management Project

TQUM: TVET Quality Management Project

TVET: Technical and Vocational Education and Training

UI: User Interface

WI-FI: Wireless Fidelity

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in food processing specifically for the module of **"IoT Web Application Development Using PHP".** Students enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labour market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: NITIW401 IoT WEB APPLICATION DEVELOPMENT

Learning Outcome 1: Develop Application programming interface (API) using PHP
Learning Outcome 2: Develop user interface
Learning Outcome 3: Integrate API Endpoints with User Interface


API
</>

Indicative contents

1.1 Analysis of IoT Web application requirements

1.2 Prepare PHP environment

1.3 Identification of PHP concepts

1.4 Develop Application Programming Interface CRUD Endpoints

1.5 Secure API endpoints

1.6 Test API endpoints

1.7 Documentation of the developed API

Key Competencies for Learning Outcome 1: Develop API using PHP

Knowledge	Skills	Attitudes
• Description of IoT web application requirements • Description of PHP Key terms • Description of PHP concept • Description of API concepts • Description of CRUD • Description of API end points	• Collecting data used in IoT Web application • Selecting IoT Web Application devices • Installing and configuring PHP environment tools used in IoT web application • Developing Application Programming Interface CRUD Endpoints. • Developing User Interface • Securing API endpoints • Testing API Endpoints • Documenting the developed API Endpoints and Versioning	• Being Exploratory • Having Critical thinking • Being Innovative when developing user interface • Being committed Developing Application Programming Interface CRUD Endpoints • Being confidence • Having Team work spirit • Having Adaptability

Duration: 40 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Analyse properly IoT web application requirements based on organization requirements
2. Describe properly PHP Key terms, PHP key concept based on IoT system requirement
3. Describe properly API concept, API Endpoint based on IoT Web Application Development
4. Describe properly CRUD based on IoT Web Application Development requirement
5. Analyse properly data collected based on IoT Web application requirements.
6. Select correctly IoT Web Application devices based on system requirement.
7. Secure properly API endpoints based on PHP data security standards.
8. Test effectively API endpoints based on IoT system requirements.
9. . Document properly the developed API according to its functionality.

 **Resources**

Equipment	**Tools**	**Materials**
● Computer ● Sensors ● Arduino boards ● Node MCU boards ● Raspberry pi	● XAMMP ● DBMS, MYSQL ● Web browser ● Text editor (Sublime text, ● Notepad++)	● Electricity ● Internet

Indicative content 1.1: Analysis of IoT Web Application Requirements.

Duration: 5 hrs

Theoretical Activity 1.1.1: Description of IoT Web Application system

Tasks:

1: Answer the following questions:

i. Define IoT Web Application
ii. What are IoT Web Application requirement
iii. Describe IoT system requirement
iv. Identify the source of data
v. Describe code management

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.1.1.

Key readings 1.1.1.: Description of IoT Web application requirements

1. Description of IoT Web application requirements

1.1 IoT Web Application is a web-based software platform designed to interact with and manage IoT devices. These applications collect, process, and display data from various connected devices and sensors, enabling users to monitor and control these devices remotely. The primary goal of an IoT web application is to provide real-time insights and facilitate the management of IoT ecosystems.

1.2 IoT Web Application Requirements

- **Scalability:** Ability to handle a growing number of devices and data points.
- **Real-Time Data Processing:** Capable of processing and displaying data in real-time.
- **Security:** Strong security measures to protect data and device integrity.
- **Interoperability:** Support for various communication protocols and device types.
- **User Interface:** Intuitive and user-friendly interface for monitoring and controlling devices.

- **Data Storage:** Efficient storage solutions for large volumes of data.
- **APIs:** Robust APIs for device integration and data exchange.
- **Reliability:** High availability and reliability to ensure continuous operation.
- **Analytics:** Tools for analyzing data and generating insights.
- **Compliance:** Adherence to relevant regulatory and industry standards.

1.3 Identify System Requirements

Collect the Data: Determine what data the IoT application needs to collect, monitor, or control. This can include sensor data (temperature, humidity, motion, etc.), user input, or external data sources.

Devices Required While Developing and Displaying IoT web application: Decide on the specific IoT devices such as: sensors, actuators, and displays that will be used in the application. Consider compatibility, scalability, and cost-effectiveness.

1.4 Identify Source of Data:

Understand where the data will come from. It could be from physical sensors, a database, external APIs, or a combination of these sources. Knowing the data sources is essential for data integration and processing.

1.4.1: IoT Devices/Sensors:

- Description: These are the primary sources of data, including various sensors and actuators that collect information from the environment or perform actions based on commands.

- Examples: Temperature sensors, humidity sensors, motion detectors, cameras, smart meters, and wearable devices.

1.4.2. External APIs:

- Description: Integration with external services or APIs can provide additional data that complements the information collected from IoT devices.

- Example: Weather APIs (for environmental data), traffic data APIs, or social media feeds that might influence IoT device behavior.

1.4.3. Databases:

- Description: Historical data that has been collected and stored in databases can serve as a source for analytics and reporting.

- Examples: SQL or NoSQL databases that store historical sensor data, user interactions, or device configurations

Functional Requirements

Functional requirements for an IoT web application define what the system should do and how it should behave. These requirements focus on the specific functionalities that the application must support to meet user needs.

Non-functional Requirements

Non-functional requirements for an IoT web application define the quality attributes, system performance, and constraints that the application must meet. These requirements focus on how the system performs its functions rather than what functions it performs.

Here's some non-functional requirements for an IoT web application:

Performance: Define performance expectations, such as response times, throughput, and scalability. Ensure the system can handle the expected number of devices and data volume.

Security: Specify security measures to protect data, devices, and communication. This includes encryption, authentication, authorization, and secure device management.

Reliability: Determine the system's availability, fault tolerance, and backup strategies. IoT applications often need to work continuously without downtime.

Scalability: Address how the system will scale as the number of devices and users grows. Consider load balancing and resource allocation.

Usability: Ensure that the user interface is intuitive and user-friendly. Consider accessibility and user experience design.

1.5. Code Management

Code Management in the context of IoT web applications involves practices and tools to manage the software development lifecycle efficiently. Key aspects include:

Version Control and collaboration: Implement a version control system (e.g., Git) to track changes in the codebase, collaborate with team members, and manage code branches.

Continuous Delivery (CD):

Automate the deployment of IoT updates to devices and cloud services.

Implement blue-green deployments or canary releases to minimize downtime and the impact of potential issues.

Code Minimization and Optimization:

Modularize Your Code: Break your code into modular components, making it easier to manage and test. This also allows you to load only the necessary modules, reducing memory usage.

Remove Unused Code: Regularly review and remove any code that is no longer necessary.

Code Size Analysis: Use tools to analyze your code's size and identify areas for optimization.

1.6 IoT System architecture

IoT system architecture typically consists of several layers, each playing a crucial role in the overall functionality of IoT solutions. Here's a breakdown of the common layers in IoT system architecture:

1.6.1. Device Layer (Perception Layer):

This is the foundational layer where the physical devices and sensors reside. It includes:

- Sensors: Devices that collect data from the environment (e.g., temperature sensors, humidity sensors, motion detectors).

- Actuators: Components that perform actions based on commands received (e.g., turning on a light, opening a valve).

- Embedded Systems: Microcontrollers or processors that manage the sensors and actuators and facilitate communication.

1.6.2. Network Layer (Transport Layer):

This layer is responsible for transmitting the data collected by the devices to the cloud or data processing systems. It includes:

- Communication Protocols: Various protocols such as MQTT, CoAP, HTTP, and WebSocket that enable communication between devices and the server.

- Network Infrastructure: Different types of networks such as Wi-Fi, cellular, LoRaWAN, Zigbee, or Bluetooth that facilitate connectivity.

1.6.3. Edge Computing Layer:

In this layer, data processing occurs closer to the source (the devices) rather than sending all data to the cloud. It includes:

- Edge Devices: Local servers or gateways that process data, reducing latency and bandwidth usage.

- Data Filtering and Aggregation: Initial analysis and processing of data to extract relevant information before sending it to the cloud.

1.6.4. Cloud Layer (Backend Layer):

This layer handles the storage, processing, and analysis of data collected from IoT devices. It includes:

- Data Storage: Databases or data lakes where large volumes of data are stored for further analysis.

- Data Analytics: Tools and algorithms that analyze the data to derive insights, generate reports, and support decision-making.

- Application Hosting: The infrastructure for hosting web applications or services that users interact with to monitor and control IoT devices.

1.6.5. Application Layer:

This is the topmost layer where end-user applications reside. It includes:

- User Interfaces: Dashboards, mobile apps, or web applications that allow users to visualize data, control devices, and receive alerts.

- APIs: Application programming interfaces that enable integration with other services or systems, facilitating interoperability.

1.6.6. Security Layer:

Although security is a concern across all layers, this layer focuses specifically on protecting the entire IoT architecture. It includes:

- Authentication and Authorization: Mechanisms to ensure that only authorized devices and users can access the system.

- Data Encryption: Protecting data in transit and at rest to prevent unauthorized access.

- Regular Updates and Monitoring: Keeping the system secure through updates and monitoring for vulnerabilities.

Theoretical Activity 1.1.2: Description on data collection

Tasks:

1: Answer the following questions:

i. What is data collection?
ii. What are the required devices use while developing and displaying data?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.1.2.

Key readings 1.1.2: Description on data collection

1.1 Data collection

Data collection is the process of gathering and measuring information on variables of interest in a systematic way that enables one to answer research questions, test hypotheses, and evaluate outcomes. In the context of IoT, data collection involves acquiring data from various sensors and devices that are part of the IoT network. This data can include anything from temperature readings, motion detection, GPS coordinates, to more complex data like video feeds or biometric information.

2. Methods of Data Collection

2.1. Surveys and Questionnaires

- Purpose: Collect quantitative and qualitative data from stakeholders, including building managers, maintenance staff, and occupants.

- Implementation: Distribute online surveys or paper questionnaires to gather insights on user needs, preferences, and pain points regarding energy usage, lighting, and HVAC systems.

2.2. Interviews

- Purpose: Conduct in-depth discussions with key stakeholders to understand their requirements and expectations.

- Implementation: Schedule one-on-one or group interviews with building managers, maintenance teams, and end-users to gather detailed information about their experiences and needs.

2.3. Observations

- Purpose: Gain firsthand insights into the building's current operations and user interactions with existing systems.

- Implementation: Perform site visits to observe how energy usage, lighting, and HVAC systems are currently managed. Note any inefficiencies or areas for improvement.

2.4. Focus Groups

- Purpose: Facilitate discussions among a group of stakeholders to explore their thoughts and opinions on specific features or functionalities.

- Implementation: Organize focus group sessions with representatives from different user groups to brainstorm ideas and gather feedback on proposed solutions.

2.5. Existing Data Analysis

- Purpose: Leverage historical data from existing building management systems or utility bills to understand energy consumption patterns.

- Implementation: Analyze past energy usage records, maintenance logs, and occupancy data to identify trends and areas where the IoT application can provide value.

2.6. Sensor Data Collection

- Purpose: Utilize IoT sensors to gather real-time data on environmental conditions and system performance.

- Implementation: Install sensors for temperature, humidity, occupancy, and energy usage to collect data that can inform system design and functionality.

1.2 The required devices used while collecting, developing, and displaying data.

- **Sensors and IoT Devices**: These are the primary data collection tools that capture various forms of data. For images, this might include cameras or image sensors that can capture still images or video footage.
- **Microcontrollers and Microprocessors:** Devices like Arduino or Raspberry Pi are often used to process data from sensors and manage communication between devices.

- **Data Storage Solutions:** Systems like cloud storage or local databases are used to store the collected data. For large image files or video data, efficient storage solutions are necessary to handle the data volume.

- **Data Processing Units:** These can include servers or cloud-based services that process the raw data into a usable format. This might involve image processing software or machine learning algorithms to analyse images.

- **Networking Equipment:** Routers, gateways, and other networking devices are essential for transmitting data from IoT devices to storage and processing units.

- **Display Devices:** Monitors, dashboards, or mobile devices can be used to visualize the data. This might involve specialized software for rendering images and displaying data analytics.

- **Development Tools**: Software development tools and platforms, such as integrated development environments (IDEs) and version control systems, are used to build applications that process and display data.

Practical Activity 1.1.3: Analysing IoT web application requirements

Task:

1: Read key reading 1.1.3

2: Referring to key reading 1.1.3, As web developer (IoT), you are asked to go to a computer lab to Analyse IoT web application requirement.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.1.3: Analysing IoT web application requirementsAalysing

Analyzing the requirements for an IoT (Internet of Things) web application involves a systematic approach to gather, evaluate, and document the necessary components to ensure the application functions effectively. Here are the steps to follow for analyzing the requirements:

1. Identify and Analyze IoT Devices and Sensors

List the devices and sensors: Identify all IoT devices, sensors, or gateways that will communicate with the web application. Define how they connect (e.g., via Wi-Fi, Bluetooth, or cellular networks).

Data types and formats: Understand what types of data (e.g., temperature, humidity, motion) the devices will generate and how this data will be structured (e.g., JSON, XML).

Communication protocols: Define the communication protocols used by the devices (e.g., MQTT, CoAP, and HTTP/HTTPS). Ensure the application can handle those protocols effectively.

2. Data Management Requirements

Data collection frequency: Determine how often data will be sent from the IoT devices to the web application.

Data storage: Analyze the requirements for storing data, including real-time data and historical data. Will the data be stored on a cloud server, local server, or hybrid?

Data processing: Understand if the data needs to be processed in real-time or near-real-time and how it will be used (e.g., for analytics, decision-making, or visualizations).

Data security: Identify requirements for securing data, including encryption in transit and at rest. Ensure compliance with regulations (e.g., GDPR, HIPAA, etc.).

3. Define User Requirements

User roles and permissions: Define different user roles (e.g., administrators, operators, viewers) and their access levels. Who needs access to what data or features?

User interface (UI) design: Analyze how users will interact with the web application. This includes designing an intuitive dashboard, ensuring responsiveness for mobile access, and enabling device control from the web interface.

Alerts and notifications: Identify if users need real-time alerts (e.g., via email or SMS) for specific events like equipment failures or threshold breaches.

4. Define Functional Requirements

Device management: Specify how the web application will manage and control IoT devices (e.g., adding/removing devices, configuring settings, firmware updates).

Data visualization: Define how the application will display data (e.g., charts, graphs, dashboards) and what real-time metrics or KPIs users will monitor.

Reporting: Determine the need for generating reports based on collected data, including the format and frequency of reports.

Device control capabilities: Analyze how users will interact with and control IoT devices through the web application (e.g., turning devices on/off, adjusting settings remotely).

5. Non-Functional Requirements

Scalability: Assess how the web application will scale as the number of IoT devices increases. The system should handle growing data volumes and support a larger user base without performance degradation.

Performance: Identify the required speed and efficiency of data collection, processing, and display. Define acceptable response times for data retrieval, real-time updates, and user interactions.

Reliability: Define uptime requirements and redundancy measures to ensure the system remains available and reliable even during high loads or failures.

Security: Evaluate the security requirements, such as secure communication (SSL/TLS), user authentication (OAuth, token-based), data encryption, and protection against cyberattacks.

Compliance: Ensure that the application meets regulatory requirements, such as data privacy and industry-specific standards (e.g., ISO, NIST).

6. Integration with Third-Party Services

APIs and interoperability: Analyze whether the application needs to integrate with third-party APIs (e.g., cloud services, external databases) or other software systems like CRMs or ERP systems.

Cloud services: Determine if the IoT web application will integrate with cloud platforms such as AWS IoT, Google Cloud IoT, or Azure IoT for data storage and processing.

Third-party tools: Identify any tools or libraries required for specific functionalities (e.g., data analytics, visualization tools).

7. Security and Privacy Requirements

Authentication and authorization: Define how users will log in and authenticate themselves (e.g., via username/password, multi-factor authentication). Set up role-based access control.

Encryption: Ensure data is encrypted both in transit and at rest, and that sensitive data such as user credentials and device information is securely stored.

Privacy concerns: Understand any legal and privacy obligations related to handling users' data, particularly if dealing with sensitive information or data from personal devices.

Steps to select required devices and collect data used in IoT

Selecting the right devices and collecting data for Internet of Things (IoT) applications involves a systematic approach. Here are the key steps:

1. Define Your Objectives: Clearly outline what you want to achieve with your IoT project. This could include monitoring, automation, data collection, etc.

2. Identify Use Cases: Specify the use cases that your IoT solution will address. This helps in determining the types of devices needed.

3. Assess Environmental Factors: Consider the environment where the devices will be deployed. Factors such as temperature, humidity, and connectivity options (Wi-Fi, cellular, etc.) can influence your device selection.

4. Select Device Types: Based on your use cases, choose the appropriate types of devices. These can include sensors (temperature, humidity, motion), actuators (for automation), gateways, and communication devices.

5. Evaluate Device Specifications: Look at the specifications of potential devices, including:

- Range and connectivity options
- Power consumption and battery life
- Data processing capabilities
- Security features

6. Consider Scalability: Ensure that the devices you choose can scale with your project's growth. This might involve selecting devices that can handle increased data loads or additional connections.

7. Prototype and Test: Before full deployment, create a prototype to test the selected devices in a controlled environment. This will help you identify any issues with data collection or connectivity.

8. Data Collection Framework: Establish a framework for how data will be collected, stored, and processed. This includes selecting appropriate cloud services or on-premises solutions for data storage.

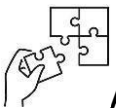
9. Implement Security Measures: Ensure that your devices and data collection methods are secure. This includes using encryption, secure communication protocols, and regular updates.

10. Monitor and Optimize: After deployment, continuously monitor the performance of your devices and the data collected. Use this information to optimize your setup and make necessary adjustments.

Points to Remember

- IoT Web Application is a web-based software platform designed to interact with and manage IoT devices. These applications collect, process, and display data from various connected devices and sensors, enabling users to monitor and control these devices remotely.
- There is different source of data in IoT web application which are: IoT Devices/Sensors, External APIs etc.
- Code Management in the context of IoT web applications involves practices and tools to manage the software development lifecycle efficiently that include continuous delivery, code minimization and optimisation
- They functional requirements define what the system should do, while non-functional requirements define how well the system performs those functions
- Analyzing the requirements for an IoT (Internet of Things) web application involves a systematic approach to gather, evaluate, and document the necessary components to ensure the application functions effectively. Here are the steps to follow for analyzing the requirements:

    1. Identify and Analyze IoT Devices and Sensors
    2. Data Management Requirements
    3. Define User Requirements
    4. Define Functional Requirements
    5. Non-Functional Requirements
    6. Integration with Third-Party Services
    7. Security and Privacy Requirements



Application of learning 1.1.

XYZ Company wants to develop an IoT web application for managing a smart building. This system will enable building managers to monitor and control energy usage, Lighting, HVAC (Heating, Ventilation, Air conditioning) through a web-based interface. you are requested to analyse the requirements by collecting data and Selecting the required devices for developing and displaying.

Indicative content 1.2: Prepare PHP Environment

Duration: 5 hrs

Theoretical Activity 1.2.1: Description of PHP key terms

Tasks:

1: Answer the following questions:

i. What do you understand by the following terms: PHP, Interpreter, Compiler, Ope n source, Web server, browser, Text editor, Integrated Development Environmen t (IDE)?
ii. What is the purpose and characteristics of PHP?
iii. Describe the types of Database

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.2.1.

Key readings 1.2.1.: Description of PHP key terms

1.Description of PHP Key Terms:

- **PHP**: PHP (Hypertext Pre-processor) is a popular open-source scripting language that is especially suited for web development. It is embedded within HTML and is used to create dynamic web pages. PHP scripts are executed on the server, and the result is returned to the client as plain HTML.
- **Interpreter:** An interpreter is a program that directly executes instructions written in a programming or scripting language without requiring them to be compiled into machine language. It reads the code line-by-line and executes it on the fly.
- **Compiler:** A compiler translates code written in a high-level programming language into machine code (binary code) that a computer's processor can execute. Unlike interpreters, compilers translate the entire program at once before execution.

- **Open Source:** Open source refers to software with source code that anyone can inspect, modify, and enhance. Open-source software is often developed collaboratively and is free to use and distribute.
- **Web Server:** A web server is a system that hosts websites and delivers web content to users over the internet. It processes incoming network requests over HTTP and several other related protocols.
- **Browser:** A web browser is a software application used to access and view websites. Browsers interpret HTML, CSS, and JavaScript code and render it into the web pages you see.
- **Text Editor:** A text editor is a program that allows users to write and edit plain text. They are often used for coding and can be simple (like Notepad) or feature-rich (like Sublime Text or Visual Studio Code).
- **Integrated Development Environment (IDE):** An IDE is a software application that provides comprehensive facilities to programmers for software development. It typically includes a code editor, debugger, and build automation tools, all integrated into a single interface.

2.Purpose and Characteristics of PHP:

Purpose: PHP is designed primarily for server-side scripting, enabling developers to create dynamic and interactive web applications. It can be used to manage databases, session tracking, and even build complete e-commerce websites.

Characteristics:

- ✓ Easy to Learn: PHP has a straightforward syntax that is easy for beginners to pick up.
- ✓ Flexibility: It supports a wide range of databases and is compatible with various platforms.
- ✓ Efficiency: PHP is optimized for web development and can handle large amounts of data efficiently.
- ✓ Community Support: Being open source, PHP has a large community that contributes to its development and provides extensive documentation and resources.
- ✓ Integration: PHP can easily integrate with other technologies and services, such as HTML, CSS, JavaScript, and various database systems.

3.Description of database

Database: is an organized collection of structured information or data that is stored and accessed electronically. Databases are designed to manage large amounts of information efficiently, allowing for easy retrieval, manipulation, and management of data.

3.1 Types of Databases:

- **Relational Databases:** These databases store data in tables with rows and columns. They use Structured Query Language (SQL) for defining and manipulating data. Examples include MySQL, PostgreSQL, and Oracle.
- **NoSQL Databases:** Designed for unstructured data, NoSQL databases are useful for large sets of distributed data. They come in various types, such as document stores (e.g., MongoDB), key-value stores (e.g., Redis), column-family stores (e.g., Cassandra), and graph databases (e.g., Neo4j).
- **In-Memory Databases:** These databases store data in the main memory rather than on disk to provide faster data access. Examples include Redis and Memcached.
- **Object-Oriented Databases:** These databases store data in the form of objects, similar to object-oriented programming. They are useful for applications that require complex data representations.
- **Distributed Databases:** These databases are spread across multiple sites or nodes, providing redundancy and fault tolerance. They can be relational or NoSQL.
- **Cloud Databases:** Hosted on cloud platforms, these databases offer scalability and flexibility, allowing users to pay for only the resources they use. Examples include Amazon RDS and Google Cloud SQL.

Practical Activity 1.2.2: Preparing PHP environment

Task:

1. Read the key readings 1.2.2
2. Referring to the key readings 1.2.2. you are asked to go to the computer lab to prepare PHP environmental tools:
3. Present your work to the trainer and whole class
4. Ask clarification where necessary

Key readings 1.2.2: Preparing PHP environment

Steps to Install and Configure PHP Environment Tools for IoT Web Application Development:

Step 1: Install a Web Server

You can choose between different web servers, but the most common options are Apache and Nginx. Here's how to install Apache:

For Windows:

1. Download XAMPP:

- Go to the [XAMPP website] (https://www.apachefriends.org/index.html) and download the XAMPP installer.

2. Install XAMPP:

- Run the installer and follow the prompts to install XAMPP. Make sure to select Apache and PHP during the installation.

For macOS:

1. Download MAMP:

- Visit the [MAMP website] (https://www.mamp.info/en/downloads/) and download MAMP.

2. Install MAMP:

- Open the downloaded file and drag MAMP to your Applications folder.

For Linux (Ubuntu):

1. Open Terminal.

2. Update Package List:

sudo apt update

3. Install Apache:

sudo apt install apache2

Step 2: Install PHP

For Windows (XAMPP):

- PHP comes bundled with XAMPP, so no additional installation is needed.

For macOS (MAMP):

- PHP is included in MAMP, so no additional installation is needed.

For Linux (Ubuntu):

1. Install PHP:

sudo apt install PHP libapache2-mod-PHP

2. Install Additional PHP Extensions (optional):

- Depending on your project, you may need additional extensions. Common ones include:

sudo apt install PHP-mysql PHP-xml PHP-mbstring PHP-curl PHP-zip

Step 3: Install a Database Server (Optional)

If your application requires a database, you can install MySQL or MariaDB.

For Windows (XAMPP):

- MySQL is included in XAMPP.

For macOS (MAMP):

- MySQL is included in MAMP.

For Linux (Ubuntu):

1. Install MySQL:

sudo apt install mysql-server

2. Secure MySQL Installation:

sudo mysql_secure_installation

Step 4: Configure the Environment

1. Start the Web Server:

- XAMPP (Windows): Open the XAMPP Control Panel and start Apache and MySQL.

- MAMP (macOS): Open MAMP and click "Start Servers."

- Linux: Start Apache with:

sudo systemctl start apache2

2. Check PHP Installation:

- Create a PHP file named `info.PHP` in your web server's root directory (usually `C:\xampp\htdocs` for XAMPP, `/Applications/MAMP/htdocs` for MAMP, or `/var/www/html` for Linux).

- Add the following code to `info.PHP`:

PHP

```
<? PHP
    PHPinfo ();
```

?>

- Open a web browser and navigate to `http://localhost/info.PHP`. You should see the PHP information page.

Step 5: Install a Code Editor

Choose a code editor that suits your needs. Some popular options include:

- Visual Studio Code: A powerful and customizable code editor.

- Sublime Text: A lightweight and fast editor.

- PHPStorm: A commercial IDE specifically designed for PHP development.

Step 6: Set Up a Version Control System (Optional)

Consider using Git for version control. You can install Git using:

For Windows:

- Download and install Git from the [official website] (https://git-scm.com/).

For macOS:

- Use Homebrew to install Git:

brew install git

For Linux (Ubuntu):

sudo apt install git

Step 7: Test Your Environment

Create a simple PHP script to ensure everything is working correctly. For example, create a file named `test.PHP` in your web server's root directory with the following content:

PHP

<?PHP

echo "Hello, World!";

?>

Navigate to `http://localhost/test.PHP` in your browser to see if it displays "Hello, World!".

Here are the common steps to install a web browser, which can be applied to popular browsers like Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari. The steps

may vary slightly depending on the operating system you are using, but the general process is similar.

Step8: Installing a Browser

1. Download the Browser:

Google Chrome: Go to the [Google Chrome website](https://www.google.com/chrome/) and click on the "Download Chrome" button.

- Mozilla Firefox: Visit the [Mozilla Firefox website] (https://www.mozilla.org/firefox/) and click on "Download Now."

- Microsoft Edge: Edge is usually pre-installed on Windows, but you can download it from the [Microsoft Edge website] (https://www.microsoft.com/edge) if needed.

- Safari: Safari is typically pre-installed on macOS. If you need to reinstall it, you can download it from the Apple website.

2. Run the Installer:

- Windows: Locate the downloaded installer file (usually in your Downloads folder) and double-click it to run.

- macOS: Open the downloaded `.dmg` file and follow the prompts to install the browser.

- Linux: Depending on the browser, you may download a `.deb` or `.rpm` file, or use a package manager. For example, for Firefox, you can run:

sudo apt install firefox

3. Follow the Installation Prompts:

- For Windows and macOS, follow the on-screen instructions to complete the installation. This may include agreeing to terms and conditions and selecting installation options.

- On Linux, if using a package manager, the installation will proceed automatically after you run the command.

4. Complete the Installation:

- Once the installation is complete, you may receive a notification or see an option to launch the browser immediately.

5. Launch the Browser:

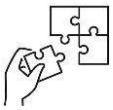
- Open the browser from the Start menu (Windows), Applications folder (macOS), or

your applications menu (Linux). You can also create a desktop shortcut for easier access.

These steps provide a general guideline for installing a web browser on various operating systems. After installation, you can customize your browser settings and start browsing the internet!

Points to Remember

- PHP, which stands for "Hypertext Pre-processor," is a widely-used open-source scripting language primarily designed for web development. There are purposes of PHP such as: PHP is designed primarily for server-side scripting, enabling developers to create dynamic and interactive web applications and characteristics are: Easy to Learn, Flexibility and Integration.
- There are different types of database such as: Relational Databases, NoSQL Databases, In-Memory Databases and Cloud Databases etc…
- These are the steps to follow when prepare PHP environment tools in IoT web application Development.
  Step 1: Install a Web Server
  Step2: Install PHP
  Step3: Install a Database Server (Optional)
  Step4: Configure the Environment
  Step5: Install a Code Editor
  Steps6.to install browser
  Step 7: Test Your Environment



Application of learning 1.2.

XYZ Company wants to set up the server-side components necessary to connect IoT devices with a central server and ensure data is stored and processed effectively, you are requested to prepare a PHP-based web environment for an IoT application.

Indicative content 1.3: Identification of PHP Concepts.

Duration: 10 hrs

Theoretical Activity 1.3.1: Description of PHP concepts

Tasks:

1: Answer the following questions:

i. Define the following terms used in PHP and give an example per each:

a) Tags
b) Super global variable
c) Operators
d) Data types
e) Variable scope
f) Constant
g) Comment
h) String concatenation

ii. What are the types of variables?

iii. What are the naming rule of variable?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.3.1.

Key readings 1.3. 1.Description of PHP concepts

1.Definitions and Examples of terms related to PHP web development

1.1 Tags: PHP tags are used to indicate the start and end of PHP code within a file. The most common tag is `<?PHP ... ?>`.naming

Example:

PHP

```
<?PHP
echo "Hello, World!";
?>
```

1.2 Variable: A variable in PHP is a container for storing data, such as numbers, strings, or arrays. Variables in PHP start with a dollar sign `$`.

Example:

PHP

```
<?PHP
$name = "Alice";
echo $name;
?>
```

1.2.1 Rules for naming variable in PHP programming

- Variable names must start with a dollar sign
- Variable names must not start with numbers
- Variable names include letters
- numbers, and underscores

Variable names are case-sensitive

1.2.2 Types of Variables: In PHP, variables can be categorized based on their data types and scope. Here are the main types:

1. Based on Data Types:

Integer: Whole numbers, e.g., `10`

Float (Double): Decimal numbers, e.g., `10.5`

String: Sequence of characters, e.g., `"Hello"`

Boolean: True or false values, e.g., `true`

Array: Collection of values, e.g., `array(1, 2, 3)`

Object: Instances of classes

NULL: Special type for a variable with no value

2. Based on Scope:

Local Variables: Declared within a function and accessible only within that function.

Global Variables: Declared outside of any function and accessible globally. To access them inside a function, the `global` keyword or `$GLOBALS` array is used.

Static Variables: Retain their value between function calls and are declared using the `static` keyword.

3. Super Global Variable: Super global variables are built-in variables in PHP that are always accessible, regardless of scope. They are used to collect data from forms, session data, etc.

Example:

PHP

```
<?PHP
echo $_SERVER['HTTP_USER_AGENT'];
?>
```

4. Operators: Operators are symbols used to perform operations on variables and values. Types of operators are: arithmetic, assignment, comparison, logical, and more operators.

1. Arithmetic Operators

These operators perform basic mathematical operations.

Addition (`+`): Adds two operands.

```
PHP
$sum = 5 + 3; // $sum is 8
```

Subtraction (`-`): Subtracts the second operand from the first.

```
PHP
$difference = 5 - 3; // $difference is 2
```

Multiplication (`*`): Multiplies two operands.

```
PHP
$product = 5 * 3; // $product is 15
```

Division (`/`): Divides the first operand by the second.

```
PHP
$quotient = 5 / 2; // $quotient is 2.5
```

Modulus (`%`): Returns the remainder of a division operation.

```
PHP
$remainder = 5 % 2; // $remainder is 1
```

2. Comparison Operators

These operators compare two values and return a boolean result.

Equal (`==`): Checks if two values are equal.

```
PHP
$isEqual = (5 == 5); // $isEqual is true
```

Identical (`===`): Checks if two values are equal and of the same type.

```
PHP
$isIdentical = (5 === '5'); // $isIdentical is false
```

Not Equal (`!=`): Checks if two values are not equal.

```
PHP
$isNotEqual = (5 != 3); // $isNotEqual is true
```

Greater Than (`>`): Checks if the left operand is greater than the right.

```
PHP
$isGreater = (5 > 3); // $isGreater is true
```

Less Than (`<`): Checks if the left operand is less than the right.

```
PHP
$isLess = (5 < 3); // $isLess is false
```

Greater Than or Equal To (`>=`): Checks if the left operand is greater than or equal to the right.

```
PHP
$isGreaterOrEqual = (5 >= 5); // $isGreaterOrEqual is true
```

Less Than or Equal To (`<=`): Checks if the left operand is less than or equal to the right.

```
PHP
```

$isLessOrEqual = (5 <= 3); // $isLessOrEqual is false

3. Logical Operators

These operators are used to combine conditional statements.

Logical AND (`&&`): Returns true if both operands are true.

PHP

$isBothTrue = (true && false); // $isBothTrue is false

Logical OR (`||`): Returns true if at least one of the operands is true.

PHP

$isEitherTrue = (true || false); // $isEitherTrue is true

Logical NOT (`!`): Reverses the boolean value.

PHP

$isNotTrue = !true; // $isNotTrue is false

4. Assignment Operators

These operators are used to assign values to variables.

Assignment (`=`): Assigns the value on the right to the variable on the left.

PHP

$x = 5; // $x is 5

Addition Assignment (`+=`): Adds the right operand to the left operand and assigns the result to the left operand.

PHP

$x += 3; // $x is now 8

Subtraction Assignment (`-=`): Subtracts the right operand from the left operand and assigns the result to the left operand.

PHP

$x -= 2; // $x is now 6

Multiplication Assignment (`*=`): Multiplies the left operand by the right operand and assigns the result to the left operand.

PHP

$x *= 2; // $x is now 12

Division Assignment (`/=`): Divides the left operand by the right operand and assigns the result to the left operand.

PHP

$x /= 3; // $x is now 4

5.Data Types: Data types specify the type of data a variable can hold. PHP supports several data types like integers, floats, strings, arrays, and objects.

1. Primitive Data Types

These are the basic data types provided by a programming language.

a. Integer

- Represents whole numbers, both positive and negative.

PHP

$age = 25; // Integer

b. Float (or Double)

- Represents numbers with decimal points.

PHP

$price = 19.99; // Float

c. String

- Represents a sequence of characters, enclosed in quotes.

PHP

$name = "John Doe"; // String

d. Boolean

- Represents a value that can be either true or false.

PHP

$isRegistered = true; // Boolean

2.Composite Data Types

These types are composed of multiple values.

a. Array

- A collection of values, which can be of the same or different data types.

PHP

$fruits = array("Apple", "Banana", "Cherry"); // Indexed array

$person = array("name" => "John", "age" => 25); // Associative array

b. Object

- An instance of a class containing properties and methods.

PHP

```
class Car {
    public $color;
    public $model;
    function __construct($color, $model) {
        $this->color = $color;
        $this->model = $model;
    }
}
$myCar = new Car("red", "Toyota"); // Object
```

3. Special Data Types

These types are used for specific purposes.

a. NULL

- Represents a variable with no value or a non-existent object.

PHP

$value = NULL; // NULL

b. Resource

- A special variable that holds a reference to an external resource, such as a database connection or file handle.

PHP

```
$connection = mysqli_connect("localhost", "username", "password", "database");
// Resource
```

4. Enumerated Data Types (Enums)

- A special data type that enables a variable to be a set of predefined constants.

PHP

```
enum Status {
    case Pending;
    case Approved;
    case Rejected;
}
$currentStatus = Status::Approved; // Enum
```

6. Variable Scope: Variable scope refers to the context within which a variable is defined and can be accessed. PHP has local, global, and static scopes.

Example:

PHP

```
<?PHP
$globalVar = "I'm global";
function testScope() {
   $localVar = "I'm local";
   echo $localVar;
}
testScope();
echo $globalVar;
?>
```

7. Constant: A constant is a name or an identifier for a simple value. The value cannot be changed during the script execution. Constants are defined using the `define ()` function.

1. Integer Constants: These are whole numbers without any fractional or decimal part.
   - Example: `42`, `-7`, `0`
2. Floating-Point Constants: These are numbers that have a decimal point or are expressed in scientific notation.
   - Example: `3.14`, `-0.001`, `2.5e3` (which is 2500)
3. Character Constants: These represent single characters and are usually enclosed in single quotes.
   - Example: `'A'`, `'z'`, `'!'`
4. String Constants: These are sequences of characters enclosed in double quotes.

- Example: `"Hello, World!"`, `"Rwanda"`, `"12345"`

5. Boolean Constants: These represent truth values and are typically used in logical operations.
   - Example: `true`, `false`
6. Null Constants: This represents a null value, indicating the absence of any value or object.
   - Example: `null` (in languages like Java, C, etc.)
7. Enumeration Constants: These are named constants defined in an enumeration, which is a special data type that consists of a set of named values.
   - Example: In an enum for days of the week, you might have `Monday`, `Tuesday`, etc.
8. Final Constants: In some programming languages, constants can be defined using keywords that prevent them from being changed.
   - Example: In Java, `final int MAX_VALUE = 100;`

8 Comment: Comments are used to describe the code and are ignored by the PHP engine. PHP supports single-line (`//` or ``) and multi-line (`/* ... */`) comments.

Example:

PHP

```
<?PHP
// This is a single-line comment
 Another single-line comment
/*
This is a
multi-line comment
*/
?>
```

9. String Concatenation: String concatenation is the process of joining two or more strings together. In PHP, the dot (`.`) operator is used for concatenation.

Example:

PHP

```
<?PHP
$firstName = "John";
$lastName = "Doe";
$fullName = $firstName . " " . $lastName;
echo $fullName;
?>
```

Theoretical Activity 1.3.2: Description of fundamental programming concepts in PHP

Tasks:

1: Answer the following questions:

i. What do you understand by the following terms?
   a) Control structure
   b) Array
   c) Functions
   d) File handling
   e) Data validation

ii. What are the components of Control structure?

iii. What are the Types of Arrays?

iv. What are the Types of Functions?

v. What are file operations in PHP programming?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.3.2.

Key readings 1.3.2: Theoretical Activity 1.3.2: Description of fundamental programming concepts in PHP

1.Description of fundamental programming concepts in PHP

1.1. Control structure: Control structures in PHP are essential for directing the flow of execution in a program. They allow you to make decisions, repeat actions, and control the order of operations based on certain conditions. Here are the main types of control structures in PHP:

1.1.1. Conditional Statements

Conditional statements execute different blocks of code based on specific conditions.

a. `if` Statement

The `if` statement executes a block of code if the specified condition is true.

PHP

```
$age = 20;

if ($age >= 18) {

   echo "You are an adult.";

}
```

b. `if...else` Statement

The `if...else` statement allows you to execute one block of code if the condition is true and another block if it is false.

PHP

```
$age = 16;

if ($age >= 18) {

   echo "You are an adult.";

} else {

   echo "You are a minor.";

}
```

c. `else if` Statement

You can chain multiple conditions using `else if`.

PHP

```
$age = 65;

if ($age < 18) {

   echo "You are a minor.";

} elseif ($age >= 18 && $age < 60) {

   echo "You are an adult.";

} else {

   echo "You are a senior citizen.";

}
```

d. `switch` Statement

The `switch` statement is an alternative to using multiple `if...else` statements, especially when dealing with multiple possible values for a single variable.

PHP

```
$day = 3;
switch ($day) {
   case 1:
      echo "Monday";
      break;
   case 2:
      echo "Tuesday";
      break;
   case 3:
      echo "Wednesday";
      break;
   default:
      echo "Not a valid day.";
}
```

1.1.2. Looping Structures

Looping structures allow you to execute a block of code multiple times.

a. `for` Loop

The `for` loop is used when you know in advance how many times you want to execute a statement or a block of statements.

PHP

```
for ($i = 0; $i < 5; $i++) {
   echo "Iteration: $i\n";
}
```

b. `while` Loop

The `while` loop executes a block of code as long as the specified condition is true.

PHP

```
$i = 0;

while ($i < 5) {

   echo "Iteration: $i\n";

   $i++;

}
```

c. `do...while` Loop

The `do...while` loop is similar to the `while` loop, but it guarantees that the block of code will be executed at least once.

PHP

```
$i = 0;

do {

   echo "Iteration: $i\n";

   $i++;

} while ($i < 5);
```

3. Control Flow Statements

Control flow statements allow you to alter the execution of loops.

a. `break`

The `break` statement is used to exit a loop or switch statement prematurely.

PHP

```
for ($i = 0; $i < 10; $i++) {

   if ($i == 5) {

      break; // Exit the loop when $i equals 5

   }

   echo "Iteration: $i\n";

}
```

b. `continue`

The `continue` statement skips the current iteration of a loop and moves to the next iteration.

PHP

```
for ($i = 0; $i < 10; $i++) {
   if ($i % 2 == 0) {
      continue; // Skip even numbers
   }
   echo "Odd number: $i\n";
}
```

3. `goto`

The `goto` statement allows you to jump to a specific point in your code marked by a label. While it can be useful in certain situations, it is generally discouraged because it can lead to less readable and harder-to-maintain code.

Example of `goto`:

PHP

```
goto label;
echo "This will be skipped.";
label:
echo "Jumped to the label.";
```

N.B: In PHP, the term "jump statement" refers to control flow statements that alter the normal execution flow of a program. The primary jump statements in PHP are `break`, `continue`, and `goto`. Each of these statements serves a different purpose in controlling the flow of execution.

2.arrays:

Arrays in PHP are a powerful and versatile data structure that allows you to store multiple values in a single variable. PHP supports both indexed arrays (numerically indexed) and associative arrays (key-value pairs). Here's a detailed overview of arrays in PHP:

2.1.1. Indexed Arrays

Indexed arrays use numeric indices to access their elements. You can create an indexed array using the `array()` function or the short array syntax `[]`.

Example of Indexed Array:

PHP

```
// Using array() function
$fruits = array("Apple", "Banana", "Cherry");
// Using short array syntax
$vegetables = ["Carrot", "Potato", "Tomato"];
// Accessing elements
echo $fruits[0]; // Outputs: Apple
echo $vegetables[1]; // Outputs: Potato
```

2.2.2. Associative Arrays

Associative arrays use named keys that you assign to them. This allows you to access values using meaningful keys instead of numeric indices

Example of Associative Array:

PHP

```
$person = array(
   "name" => "John",
   "age" => 30,
   "city" => "Kigali"
);
// Accessing elements using keys
echo $person["name"]; // Outputs: John
echo $person["age"]; // Outputs: 30
```

2.2.3. Multidimensional Arrays

Multidimensional arrays are arrays that contain other arrays. This allows you to create complex data structures.

Example of Multidimensional Array:

PHP

```
$contacts = array(
   "John" => array(
```

```
        "email" => "john@example.com",
        "phone" => "123-456-7890"
    ),
    "Jane" => array(
        "email" => "jane@example.com",
        "phone" => "987-654-3210"
    )
);
// Accessing elements in a multidimensional array
echo $contacts["John"]["email"]; // Outputs: john@example.com
```

3. Functions

Functions in PHP are blocks of code that perform a specific task and can be reused throughout your program. They help organize your code, make it more readable, and reduce redundancy. Here's a detailed overview of functions in PHP:

3.1 types of function

1. Built-in Functions

PHP comes with a vast library of built-in functions that perform various tasks, such as string manipulation, array handling, mathematical operations, and more. These functions are readily available and can be used without any additional definition.

Example:

PHP

```
echo strlen("Hello, World!"); // Outputs: 13
```

2. User-defined Functions

User-defined functions are functions that you create to perform specific tasks. These functions can accept parameters and return values as needed.

Example:

PHP

```
function multiply($a, $b) {
    return $a * $b;
```

```
}

echo multiply(5, 3); // Outputs: 15
```

2.1. Defining a Function

You can define a function using the `function` keyword, followed by the function name and parentheses. Any parameters the function takes are defined within the parentheses.

Example of a Simple Function:

PHP

```
function sayHello() {
   echo "Hello, World!";
}

// Calling the function
sayHello(); // Outputs: Hello, World!
```

2.2. Function with Parameters

Functions can accept parameters (also known as arguments) that allow you to pass values into the function.

Example with Parameters:

PHP

```
function greet($name) {
   echo "Hello, " . $name . "!";
}

// Calling the function with an argument
greet("Alice"); // Outputs: Hello, Alice!
```

2.3. Returning Values

Functions can return values using the `return` statement. When a function returns a value, you can capture it in a variable.

Example of a Function with Return Value:

PHP

```
function add($a, $b) {
   return $a + $b;
}
// Capturing the return value
$sum = add(5, 10);
echo $sum; // Outputs: 15
```

2.4. Default Parameter Values

You can set default values for function parameters. If the caller does not provide a value for that parameter, the default value is used.

Example with Default Parameter:

PHP

```
function greet($name = "Guest") {
   echo "Hello, " . $name . "!";
}
// Calling the function without an argument
greet(); // Outputs: Hello, Guest!
```

2.5 Using Functions in Arrays

You can use functions as values in arrays, allowing for flexible programming patterns.

Example of Functions in Arrays:

PHP

```
$functions = [
   'add' => function($a, $b) { return $a + $b; },
   'subtract' => function($a, $b) { return $a - $b; },
];
echo $functions['add'](10, 5); // Outputs: 15
echo $functions['subtract'](10, 5); // Outputs: 5
```

3. Anonymous Functions

3.1 Closures

PHP also supports anonymous functions, which are functions without a name. These can be assigned to variables or passed as arguments to other functions.

Example of an Anonymous Function:

PHP

```
$multiply = function($a, $b) {
   return $a * $b;
};
echo $multiply(3, 4); // Outputs: 12
```

3.2 Recursive Functions

Recursive functions are functions that call themselves to solve a problem. They are often used for tasks that can be broken down into smaller, similar tasks, such as calculating factorials or traversing trees.

Example:

PHP

```
function factorial($n) {
   if ($n <= 1) {
      return 1;
   }
   return $n * factorial($n - 1);
}
echo factorial(5); // Outputs: 120
```

3.3. Callback Functions

A callback function is a function that is passed as an argument to another function. This allows for custom behavior to be defined in the context of that function.

Example:

PHP

```
function processArray($arr, $callback) {
   foreach ($arr as $value) {
      echo $callback($value) . "\n";
   }
}
processArray([1, 2, 3], function($num) {
   return $num * 2; // Callback function that doubles the value
});
```

4.File handling in PHP

File handling in PHP allows you to read from and write to files on the server. PHP provides a range of functions to perform various file operations, such as creating, opening, reading, writing, and deleting files. Here's an overview of file handling in PHP:

4.1. Opening a File

You can open a file using the `fopen()` function. This function requires the file path and the mode in which you want to open the file.

Common Modes:

- `'r'`: Read-only. The file pointer is placed at the beginning of the file.

- `'w'`: Write-only. Opens the file for writing; if the file already exists, it is truncated to zero length.

- `'a'`: Write-only. Opens the file for writing; the file pointer is placed at the end of the file.

- `'r+'`: Read and write. The file pointer is placed at the beginning of the file.

Example:

PHP

```
$file = fopen("example.txt", "r"); // Opens the file for reading
```

4.2. Reading from a File

Once a file is opened, you can read its contents using various functions:

- `fgets()`: Reads a line from the file.

- `fread()`: Reads a specified number of bytes from the file.

- `file_get_contents()`: Reads the entire file into a string.

Example:

PHP

```
$handle = fopen("example.txt", "r");
while (($line = fgets($handle)) !== false) {
   echo $line; // Outputs each line of the file
}
fclose($handle); // Always close the file after use
```

4.3. Writing to a File

You can write data to a file using the `fwrite()` function or by using `file_put_contents()` for simpler use cases.

Example of `fwrite()`:

PHP

```
$handle = fopen("example.txt", "w"); // Opens the file for writing
fwrite($handle, "Hello, World!\n");
fclose($handle);
```

Example of `file_put_contents()`:

PHP

```
file_put_contents("example.txt", "Hello, World!\n"); // Writes to the file
```

4.4. Deleting a File

You can delete a file using the `unlink()` function.

Example:

PHP

```
if (file_exists("example.txt")) {
   unlink("example.txt"); // Deletes the file
```

```
    echo "File deleted.";
} else {
    echo "File does not exist.";
}
```

4.5 Closing a File

Always close a file after you are done with it using the `fclose()` function to free up system resources.

Example:

PHP

```
$handle = fopen("example.txt", "r");
// Perform file operations...
fclose($handle); // Close the file
```

5.data validation

Data validation in PHP is an essential process that ensures the data provided by users is accurate, complete, and secure before it is processed or stored. Validating data helps prevent issues such as incorrect data entry, security vulnerabilities, and application errors. Here's an overview of data validation techniques in PHP:

5.1. Types of data validation

1. Client-side Validation: Performed using JavaScript before data is submitted to the server. This provides immediate feedback to the user but should not be relied upon solely.

2. Server-side Validation: Performed using PHP or other server-side languages to ensure data integrity and security. This is crucial as client-side validation can be bypassed.

5.2 Techniques used for Data Validation

1. Checking Required Fields

You can check if a field is empty using the `empty()` function.

Example:

PHP

```
if (empty($_POST['username'])) {
```

```
    echo "Username is required.";
}
```

2. Validating String Length

You can validate the length of a string using `strlen()`.

Example:

PHP

```
if (strlen($_POST['password']) < 6) {
    echo "Password must be at least 6 characters long.";
}
```

3. Validating Email Addresses

You can use the `filter_var()` function with the `FILTER_VALIDATE_EMAIL` filter to validate email addresses.

Example:

PHP

```
$email = $_POST['email'];
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    echo "Invalid email format.";
}
```

4. Validating URLs

Similarly, you can validate URLs using `filter_var()` with the `FILTER_VALIDATE_URL` filter.

Example:

PHP

```
$url = $_POST['website'];
if (!filter_var($url, FILTER_VALIDATE_URL)) {
    echo "Invalid URL format.";
}
```

5. Validating Numbers

You can check if a value is a number using `is_numeric()` or validate integers using `filter_var()` with the `FILTER_VALIDATE_INT` filter.

Example:

PHP

```
$age = $_POST['age'];

if (!is_numeric($age)) {

   echo "Age must be a number.";

}

// Using filter_var

if (!filter_var($age, FILTER_VALIDATE_INT)) {

   echo "Age must be a valid integer.";

}
```

6. Validating Regular Expressions

You can use regular expressions with `preg_match()` to perform custom validation.

Example:

PHP

```
$username = $_POST['username'];

if (! preg_match("/^[a-zA-Z0-9_]{5,} $/", $username)) {

   echo "Username must be at least 5 characters long and contain only letters, numbers, and underscores.";

}
```

7. Sanitizing Input Data

In addition to validation, it's important to sanitize input data to prevent security issues like SQL injection or XSS (Cross-Site Scripting). You can use functions like `htmlspecialchars()` and `strip_tags()`.

Example:

PHP

```
$name = htmlspecialchars($_POST['name']); // Converts special characters to HTML entities
```

$comment = strip_tags($_POST['comment']); // Removes HTML and PHP tags

8. Using PHP Data Validation Libraries

For more complex validation scenarios, you can use libraries like Respect\Validation or Laravel's validation if you are working within a framework. These libraries provide a more structured approach to validation.

9. Server-side vs Client-side Validation

While server-side validation is crucial for security, you can also implement client-side validation using JavaScript to improve user experience. However, always validate on the server side, as client-side validation can be bypassed.

Practical Activity 1.3.2: applying programming fundamentals concepts in PHP

Task:

1: Read key readings 1.3.2
2: Referring to the key readings 1.3.2, you are requested to apply programming fundamentals in PHP
3: Present you work to the trainer or colleagues
4: Ask questions where necessary

Key readings 1.3.2: applying programming fundamentals concepts in PHP

Steps to apply programming fundamentals concepts in PHP

- Applying PHP fundamentals effectively involves a structured approach to learning and implementing the core concepts of the language. Here's a step-by-step guide to help you get started with PHP fundamentals:

Step 1: Set Up Your Development Environment

1. Install a Web Server: Use software like XAMPP, WAMP, or MAMP, which includes Apache, PHP, and MySQL.
2. Choose a Code Editor: Use a code editor or IDE like Visual Studio Code, Sublime Text, or PHPStorm for writing your PHP code.

3. Create a Project Directory: Set up a folder in your web server's root directory (e.g., `htdocs` in XAMPP) for your PHP projects.

Step 2: Learn Basic Syntax

1. PHP Tags: Understand how to start and end PHP code with `<?PHP` and `?>`.
2. Variables: Learn how to declare variables and understand variable types (strings, integers, arrays, etc.).
3. Comments: Use single-line (`//`) and multi-line (`/* ... */`) comments for documentation.

Step 3: Control Structures

1. Conditional Statements: Learn how to use `if`, `else`, and `switch` statements to control the flow of your program.
2. Loops: Understand different types of loops (`for`, `while`, `foreach`) for iterating over data.

Step 4: Functions

1. Define Functions: Learn how to create and call functions to organize your code.
2. Parameters and Return Values: Understand how to pass parameters to functions and return values.

Step 5: Arrays

1. Creating Arrays: Learn how to create indexed and associative arrays.
2. Array Functions: Familiarize yourself with built-in array functions like `count()`, `array_push()`, `array_merge()`, and `foreach` for iterating over arrays.

Step 6: File Handling

1. Opening Files: Use `fopen()` to open files for reading or writing.
2. Reading and Writing: Learn how to read from files using `fgets()` or `fread()`, and write to files using `fwrite()`.
3. Closing Files: Always close files using `fclose ()` after operations.

Step 7: Data Validation and Sanitization

1. Input Validation: Learn how to validate user inputs using conditional statements and functions like `filter_var()`.
2. Sanitization: Understand how to sanitize inputs using functions like `htmlspecialchars()` and `strip_tags()` to prevent security issues.

1. Control Structures

Control structures allow you to control the flow of execution in your code. Common control structures include `if`, `else`, `switch`, and loops like `for`, `while`, and `foreach`.

Example: If-Else Statement

PHP

```
$age = 20;
if ($age >= 18) {
   echo "You are an adult.";
```

```
} else {
   echo "You are a minor.";
}
```

2. Arrays

Arrays in PHP are used to store multiple values in a single variable. You can create indexed arrays and associative arrays.

Example: Indexed Array

PHP

```
$fruits = array("Apple", "Banana", "Cherry");
echo $fruits[1]; // Outputs: Banana
```

Example: Associative Array

PHP

```
$person = array("name" => "John", "age" => 30);
echo $person["name"]; // Outputs: John
```

3. Functions

Functions are reusable blocks of code that perform a specific task. You can define your own functions in PHP.

Example: Function Definition and Call

PHP

```
function greet($name) {
   return "Hello, " . $name . "!";
}
echo greet("Alice"); // Outputs: Hello, Alice!
```

4. File Handling

PHP provides functions to handle files, allowing you to read from and write to files.

Example: Writing to a File

PHP

```
$file = fopen("example.txt", "w");
fwrite($file, "Hello, World!");
fclose($file);
```

Example: Reading from a File

PHP

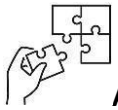
```
$file = fopen("example.txt", "r");
$content = fread($file, filesize("example.txt"));
fclose($file);
echo $content; // Outputs: Hello, World!
```

5. Data Validation

Data validation is essential to ensure that the data being processed meets specific criteria. You can use functions to validate user input.

Example: Validating an Email Address

```
PHP
function validateEmail($email) {
   if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
      return true;
   } else {
      return false;
   }
}
$email = "test@example.com";
if (validateEmail($email)) {
   echo "Valid email address.";
} else {
   echo "Invalid email address.";
}
```



Application of learning 1.3.

You have been hired to develop a small system for collecting and managing customer feedback for a local business. The system will allow users to submit their feedback, store it in a file, display all feedback entries, and delete outdated or inappropriate feedback. Additionally, the system will validate the inputs to ensure only appropriate feedback is accepted.

Indicative content 1.4: Develop Application Programming Interface CRUD Endpoints.

Duration: 10 hrs

Theoretical Activity 1.4.1: Description of API CRUD Endpoints

Tasks:

1: Answer the following questions:

i. What is Application Programming Interface (API) and Endpoints?
ii. Explain the types of Representational State Transfer REST) and Simple Object A ccess Protocol (SOAP)
iii. What are the API working Principles?
iv. Describe the API Request methods
v. Describe endpoint used in API?
vi. What is the application of data formats (JSON)?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.4.1.

Key readings 1.4.1.: Description of API CRUD Endpoints

1. What is Application Programming Interface (API) and Endpoints

Application Programming Interface (API): An API is a set of rules and protocols for building and interacting with software applications. It defines the methods and data formats that applications can use to communicate with each other, enabling different software systems to interact and share data.

1.1 Types of APIs:

- Web APIs: These are accessed over the internet using HTTP/HTTPS protocols. They allow applications to communicate with web services and are commonly used in web and mobile app development.

- Library APIs: These provide a set of functions and procedures for interacting with a software library. Developers can use these APIs to perform specific tasks without needing to understand the underlying implementation.

- Operating System APIs: These allow applications to interact with the operating system, enabling tasks like file management, memory allocation, and hardware interaction.

-Database APIs-: These APIs facilitate communication between applications and databases, allowing for operations like querying, updating, and managing data.

2 .Endpoints

In the context of APIs, endpoints are specific URLs or URIs where an API can be accessed by a client. Each endpoint corresponds to a specific function or resource in the API, allowing users to perform operations like retrieving data, updating information, or executing a command.

In the context of web development and APIs (Application Programming Interfaces), endpoints refer to specific URLs or URIs (Uniform Resource Identifiers) that allow clients to interact with a server. Each endpoint represents a specific function or resource that can be accessed or manipulated through HTTP requests. Here's a breakdown of what endpoints are and their significance:

2.1. HTTP Methods:

Endpoints are usually associated with HTTP methods that define the type of operation to be performed. Common HTTP methods include:

- GET: Retrieve data from the server.
- POST: Send data to the server to create a new resource.
- PUT: Update an existing resource on the server.
- DELETE: Remove a resource from the server.

2.2. Resource Representation:

Each endpoint typically represents a resource, such as a user, product, or article. For example:

- `GET /users`: Retrieve a list of users.
- `POST /users`: Create a new user.
- `GET /users/{id}`: Retrieve details of a specific user by their ID.
- `PUT /users/{id}`: Update a specific user's information.
- `DELETE /users/{id}`: Delete a specific user.

2.3. Parameters:

Endpoints can accept parameters, which may be included in the URL path, query string, or request body. These parameters help filter, sort, or specify the data being requested or sent. For example:

- `GET /products? category=electronics`: Retrieve products in the electronics category.

2.4. Response Format:

When a client makes a request to an endpoint, the server responds with data, usually in a structured format such as JSON or XML. The response may include the requested data, status codes, and error messages.

6. RESTful APIs: In REST (Representational State Transfer) architecture, endpoints are designed to be stateless and follow standard conventions. RESTful APIs use endpoints to represent resources and enable CRUD (Create, Read, Update, Delete) operations.

Importance of Endpoints:

- Client-Server Communication: Endpoints facilitate communication between client applications (like web or mobile apps) and server-side applications or services.
- Modularity: By organizing functionality into distinct endpoints, developers can create modular and maintainable APIs, making it easier to manage and update specific parts of the application.
- Scalability: Well-defined endpoints allow for easier scaling of applications, as different parts of the system can be developed and maintained independently.

2. What is the Meaning of Terms: REST, SOAP, JSON

REST (Representational State Transfer): REST is an architectural style for designing networked applications. It uses a stateless communication protocol, typically HTTP, and involves standard operations like GET, POST, PUT, and DELETE. RESTful APIs are designed around resources, which are identified by URLs.

SOAP (Simple Object Access Protocol): SOAP is a protocol used for exchanging structured information in web services. It relies on XML as its message format and typically uses HTTP or SMTP for message negotiation and transmission. SOAP is known for its robustness and ability to handle complex operations.

SOAP APIs: Simple Object Access Protocol (SOAP) is a protocol for exchanging structured information in web services. SOAP APIs use XML for message formatting and typically rely on HTTP or SMTP for message transmission. They are known for their robustness and strict standards.

JSON (JavaScript Object Notation): JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is commonly used to transmit data between a server and a web application as an alternative to XML.

3. Explain the Types of REST and SOAP

3.1 Types of REST:

RESTful APIs: These adhere to REST principles, using standard HTTP methods and focusing on statelessness, resource-based operations, and a uniform interface. RESTful APIs are typically lightweight and easily scalable.

3.2 Types of SOAP:

SOAP 1.1 and SOAP 1.2: These are versions of the SOAP protocol, with SOAP 1.2 being the more recent and standardized version. Both versions use XML for messaging and can handle complex transactions and security features.

4. API Working Principles

- **Uniform Interface:** APIs should have a consistent and standardized way of communicating, which simplifies interaction and understanding.
- **Statelessness:** Each API request from a client contains all the information needed to understand and process the request, without relying on stored context on the server.
- **Catchability:** Responses should define themselves as cacheable or non-cacheable to optimize performance and scalability.
- **Layered System:** An API can have multiple layers, which can improve scalability and security.
- **Code on Demand (optional):** Servers can extend client functionality by transferring executable code, though this is optional.

5. API Request Methods

GET: Retrieves data from a server at the specified resource.
POST: Sends data to the server to create a new resource.
PUT: Updates an existing resource on the server with new data.
DELETE: Removes a specified resource from the server.
PATCH: Partially updates an existing resource on the server.

6.Application of Data Format JSON

JSON is widely used for:

Web APIs: JSON is the preferred data format for exchanging data between clients and servers in web applications due to its simplicity and ease of use.

Configuration Files: JSON is used to store configuration settings in a human-readable format.

Data Serialization: JSON is used to serialize and deserialize data in programming languages, making it easy to store and transmit structured data.

Mobile Applications: JSON is used for data exchange in mobile applications due to its lightweight nature, which is important for performance and efficiency.

Practical Activity 1.4.2: Applying of data format (JSON)

Task:

1: Read key reading 1.4.2

2: Referring to key reading 1.4.2, As web developer (IoT), you are asked to go to the computer lab to Apply JSON data format in PHP.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.4.2: Applying of data format(JSON)

- **Steps to Apply JSON in IoT Web Application Development Using PHP**

Step 1. Define the Data Structure

Determine the data structure you will be using for your IoT devices. This includes defining the attributes (e.g., device ID, name, type, status) that will be sent and received as JSON.

Step 2. Set Up the PHP Environment

Ensure you have a PHP development environment set up, including a web server (like Apache or Nginx) and a database (like MySQL).

Step 3. Create the Database Schema

Design your database schema to store IoT device information. Ensure you have tables that can accommodate the data you will be sending and receiving.

Step 4. Develop API Endpoints

Create RESTful API endpoints in PHP to handle CRUD operations (Create, Read, Update, Delete) for your IoT devices. Each endpoint should be capable of processing JSON data.

Step 5. Handle Incoming JSON Data

Use `file_get_contents ('PHP://input')` to retrieve raw JSON data sent in the body of HTTP requests (typically for POST and PUT requests).

Decode the JSON data into a PHP array or object using `json_decode()`.

```
PHP

$jsonInput = file_get_contents('PHP://input');

$data = json_decode($jsonInput, true);
```

Step 6. Validate the Data

Validate the incoming JSON data to ensure that it contains all necessary fields and that the data types are correct. Respond with appropriate error messages if validation fails.

```
PHP

if (!isset($data['device_id']) || !isset($data['device_name'])) {
    http_response_code(400);
    echo json_encode(["error" => "Invalid data. Device ID and name are required."]);
    exit;
}
```

Step7. Process the Data

- Depending on the endpoint, process the data accordingly:
  - For CREATE operations, insert the data into the database.
  - For READ operations, query the database and return the data as JSON.
  - For UPDATE operations, modify existing records in the database.
  - For DELETE operations, remove records from the database.

Step 8. Send JSON Responses

After processing requests, send responses back to the client in JSON format. Use `json_encode()` to convert PHP arrays or objects into JSON strings.

```
PHP

$response = ["status" => "success", "message" => "Device registered successfully."];

header('Content-Type: application/json');
```

```
echo json_encode($response)
```

Step 9. Error Handling

- Implement error handling to manage exceptions and unexpected behavior. Return meaningful error messages in JSON format to help clients understand what went wrong.

```
PHP

try {
    // Database operations
} catch (Exception $e) {
    http_response_code(500);
    echo json_encode(["error" => "Server error: " . $e->getMessage()]);
}
```

step10.Testing the API

- Use tools like Postman or cURL to test your API endpoints. Ensure that they correctly handle JSON data and respond appropriately.

step11. Documentation

- Document your API endpoints, including the expected request and response formats, to help other developers understand how to interact with your IoT application.

Demonstration of data formt JSON

JSON (JavaScript Object Notation) is widely used in PHP applications, especially when developing APIs or handling data interchange between the client and server. Below are key applications of JSON in PHP, including encoding and decoding JSON data, and practical examples.

Applications of JSON in PHP

1. Data Interchange

JSON is commonly used to exchange data between a server and a client. It is lightweight and easy to read, making it ideal for APIs.

2. Storing and Retrieving Data

JSON can be used to store complex data structures in a single field in a database or a file, allowing for easy retrieval and manipulation.

3. Configuration Files

JSON files are often used to store configuration settings for applications due to their simplicity and readability.

Working with JSON in PHP

PHP provides built-in functions to encode and decode JSON data:

- `json_encode ()`: Converts a PHP array or object into a JSON string.

- `json_decode ()`: Converts a JSON string into a PHP array or object.

Example Usa 1. Encoding Data to JSON

PHP array

```
$data = [
   "device_id" => "12345",
   "device_name" => "Temperature Sensor",
   "device_type" => "sensor",
   "status" => "active"
];
// Convert array to JSON
$jsonData = json_encode($data)
// Output JSON string
header('Content-Type: application/json');
echo $jsonData;
?>
```

2. Decoding JSON Data

PHP

```
<?PHP
// Sample JSON string (usually received from a client)
```

```
$jsonString = '{"device_id":"12345","device_name":"Temperature Sensor","device_type":"sensor","status":"active"}';

// Convert JSON string to PHP array

$dataArray = json_decode($jsonString, true); // true to get associative array

// Accessing data

echo "Device ID: " . $dataArray['device_id'] . "\n";

echo "Device Name: " . $dataArray['device_name'] . "\n";

?>
```

3. Handling JSON in API Requests

When developing an API, you might receive JSON data in a POST request. Here's how you can handle it:

PHP

```
<?PHP

// Get JSON input from request body

$jsonInput = file_get_contents('PHP://input');

// Decode JSON to PHP array

$data = json_decode($jsonInput, true);

// Validate data

if (isset($data['device_id']) && isset($data['device_name'])) {

   // Process the data (e.g., store in database)

   // Example of creating a response

   $response = [

      "status" => "success",

      "message" => "Device registered successfully.",

      "device_id" => $data['device_id']

   ];

   header('Content-Type: application/json');
```

Practical Activity 1.4.3: Execution of CREATE Query

Task:

1: Read key reading 1.4.3

2: Referring to key reading 1.3.2, As web developer (IoT), you are asked to go to the computer lab to Execute CREATE Query.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.4.3: Execution of CREATE Query

These are the steps of executing CREATE query

Step 1: Set Up Your Environment

1. Install a Web Server: You can use Apache or Nginx. If you're using a local development environment, tools like XAMPP, WAMP, or MAMP can simplify the setup.

2. Install PHP: Ensure PHP is installed on your server. Most web server packages come with PHP pre-installed.

3. Install a Database: MySQL or MariaDB is commonly used for IoT applications. Ensure it is installed and running.

Step 2: Create a Database and Table

1. Access MySQL: Use a tool like PHPMyAdmin or the MySQL command line to create your database.

2. Create a Database:

sql

CREATE DATABASE iot_database;

3. Create a Table:

sql

USE iot_database;

CREATE TABLE sensor_data (

```
    id INT AUTO_INCREMENT PRIMARY KEY,
    sensor_id VARCHAR(50) NOT NULL,
    temperature FLOAT,
    humidity FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Connect to the Database in PHP

1. Create a PHP File: Create a file named `db.PHP` to handle the database connection.

```
PHP
<? PHP
$host = 'localhost';
$db = 'iot_database';
$user = 'your_username';
$pass = 'your_password';
// Create a connection
$conn = new mysqli($host, $user, $pass, $db);
// Check the connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
?>
```

Step 4: Create the Query to Insert Data

1. Create Another PHP File: Create a file named `insert_data.PHP` to handle the data insertion.

```
<? PHP
include 'db.PHP'; // Include the database connection
// Sample data from an IoT device
```

```
$sensor_id = 'sensor_1';

$temperature = 25.5; // Example temperature value

$humidity = 60; // Example humidity value

// Prepare the SQL query

$sql = "INSERT INTO sensor_data (sensor_id, temperature, humidity) VALUES (?, ?, ?)";

// Prepare statement

$stmt = $conn->prepare($sql);

$stmt->bind_param("sdd", $sensor_id, $temperature, $humidity); // s = string, d = double

// Execute the query

if ($stmt->execute()) {

    echo "Data inserted successfully.";

} else {

    echo "Error: " . $stmt->error;

}

// Close the statement and connection

$stmt->close ();

$conn->close ();

?>
```

Execute the Query

- Execute the prepared statement:

PHP

```
$stmt->execute ();
```

. Create a PHP File for Querying: Create a file named `fetch_data.PHP` to retrieve data from the database.

PHP

```
<? PHP
```

```
include 'db.PHP'; // Include the database connection

// Prepare the SQL query

$sql = "SELECT * FROM sensor_data ORDER BY created_at DESC";

// Execute the query

$result = $conn->query($sql);

if ($result->num_rows > 0) {

// Output data of each row

while ($row = $result->fetch_assoc()) {

echo "ID: " . $row["id"] . " - Sensor ID: " . $row["sensor_id"] . " - Temperature: " . $row["temperature"] . " - Humidity: " . $row["humidity"] . " - Timestamp: " . $row["created_at"] . "<br>";

}

} else {

echo "No data found.";

}

// Close the connection

$conn->close ();

?>
```

Step5. Prepare the Statement

- Prepare the SQL statement to prevent SQL injection:

PHP

```
$stmt = $pdo->prepare($sql);
```

Step6. Bind Parameters

- Bind the parameters to the SQL query. This is where you will insert the actual values:

PHP

```
$stmt->bindParam(':sensor_id', $sensorId);

$stmt->bindParam(':value', $value);
```

$stmt->bindParam(':timestamp', $timestamp);

step7: Handle the Response

- Depending on the type of query, handle the response accordingly. For example, if you are selecting data, you can fetch the results:

PHP

$result = $stmt->fetchAll(PDO: FETCH_ASSOC);

step8. Close the Connection

- Although PHP automatically closes the connection at the end of the script, you can explicitly set it to null:

PHP

$pdo = null;

step9. Error Handling

- Implement error handling to manage any issues that arise during the database operations.

Step10: Test the Insertion

1. Run the Script: Open your web browser and navigate to `http://localhost/your_project_folder/insert_data.PHP`. This should execute the script, and you should see a success message if everything is set up correctly.

Below are examples demonstrating how to register an IoT device, validate data from IoT devices, and store IoT data in a database using PHP.

a) Registering an IoT Device using PHP

To register an IoT device, you typically collect details about the device and store them in a database. Here's a simple example:

<? PHP

// Database connection details

$host = 'localhost';

$dbname = 'iot_database';

$username = 'root';

$password = '';

```
try {
   // Create a new PDO connection
   $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   // Device details
   $deviceName = 'Temperature Sensor';
   $deviceId = '12345';
   $deviceType = 'Sensor';
   // Insert device into the database
   $sql = "INSERT INTO devices (device_name, device_id, device_type) VALUES (:device_name, :device_id, :device_type)";
   $stmt = $pdo->prepare($sql);
   $stmt->execute([
      ':device_name' => $deviceName,
      ':device_id' => $deviceId,
      ':device_type' => $deviceType
   ]);
   echo "Device registered successfully.";
} catch (PDOException $e) {
   echo "Error: " . $e->getMessage();
}
?>
```

b) Validating Data from IoT Devices using PHP

Validation ensures that the data received from IoT devices is accurate and safe to process. Here's an example of validating data:

```
<? PHP
// Sample data from IoT device
```

```
$data = [
   'temperature' => 25.5, // in Celsius
   'humidity' => 60,     // in percentage
];
// Function to validate data
function validateData($data) {
   if (!is_numeric($data['temperature']) || $data['temperature'] < -50 || $data['temperature'] > 150) {
      return "Invalid temperature value.";
   }
   if (!is_numeric($data['humidity']) || $data['humidity'] < 0 || $data['humidity'] > 100) {
      return "Invalid humidity value.";
   }
   return true;
}
$validationResult = validateData($data);
if ($validationResult === true) {
   echo "Data is valid.";
} else {
   echo $validationResult;
}
?>
```

c) Storing IoT Data in the Database using PHP

Once the data is validated, you can store it in a database. Here's how you can do it:

```
<?PHP
// Database connection (reuse from the registration example)
$host = 'localhost';
```

```
$dbname = 'iot_database';

$username = 'root';

$password = '';

try {

  // Create a new PDO connection

  $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);

  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  // Sample validated data from IoT device

  $data = [

    'device_id' => '12345',

    'temperature' => 25.5,

    'humidity' => 60,

    'timestamp' => date('Y-m-d H:i:s')

  ];

  // Insert data into the database

  $sql = "INSERT INTO device_data (device_id, temperature, humidity, timestamp) VALUES (:device_id, :temperature, :humidity, :timestamp)";

  $stmt = $pdo->prepare($sql);

  $stmt->execute([

    ':device_id' => $data['device_id'],

    ':temperature' => $data['temperature'],

    ': humidity' => $data
```

Practical Activity 1.4.4: Execution of READ Query

Task:

1: Read key reading 1.4.4

2: Referring to key reading 1.4.4, As web developer (IoT), you are asked to go to the computer lab to Execute READ Query.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.4.4: Practical Activity 1.4.4: Execution of READ Query

These are the steps of executing READ query

Step 1: Set Up Your Environment

1. Install a Web Server: Use a web server like Apache or Nginx. For local development, you can use packages like XAMPP, WAMP, or MAMP.

2. Install PHP: Ensure that PHP is installed on your server. Most web server packages include PHP.

3. Install a Database: MySQL or MariaDB is commonly used. Ensure it is installed and running.

Step 2: Create a Database and Table

1. Access MySQL: Use PHPMyAdmin or the MySQL command line to create your database.

2. Create a Database:

sql

```
CREATE DATABASE iot_database;
```

3. Create a Table:

sql

```
USE iot_database;
```

```
CREATE TABLE sensor_data (
    id INT AUTO_INCREMENT PRIMARY KEY,
    sensor_id VARCHAR(50) NOT NULL,
    temperature FLOAT,
    humidity FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Connect to the Database in PHP

1. Create a PHP File: Create a file named `db.PHP` to handle the database connection.

```
PHP
<?PHP
$host = 'localhost';
$db = 'iot_database';
$user = 'your_username'; // Replace with your MySQL username
$pass = 'your_password'; // Replace with your MySQL password
// Create a connection
$conn = new mysqli($host, $user, $pass, $db);
// Check the connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
?>
```

Step 4: Create the Read Query

1. Create Another PHP File: Create a file named `read_data.PHP` to handle the data retrieval.

```
PHP
```

```
<? PHP

include 'db.PHP'; // Include the database connection

// Prepare the SQL query to read data

$sql = "SELECT * FROM sensor_data ORDER BY created_at DESC"; // Fetch all data, ordered by timestamp

// Execute the query

$result = $conn->query($sql);

// Check if there are results

if ($result->num_rows > 0) {

    // Output data of each row

    while ($row = $result->fetch_assoc()) {

        echo "ID: " . $row["id"] . " - Sensor ID: " . $row["sensor_id"] . " - Temperature: " . $row["temperature"] . "°C - Humidity: " . $row["humidity"] . "% - Timestamp: " . $row["created_at"] . "<br>";

    }

} else {

    echo "No data found.";

}

// Close the connection

$conn->close();

?>
```

Step 5: Test the Read Query

1. Run the Script: Open your web browser and navigate to `http://localhost/your_project_folder/read_data.PHP`. This should execute the script, and you should see the retrieved data displayed. If there are no entries in the database, you will see "No data found."

Step 6: Handling Errors and Enhancements

1. Error Handling: You can enhance the script by adding error handling for the SQL query execution.

```
PHP

// Execute the query and check for errors

if ($result = $conn->query($sql)) {

    if ($result->num_rows > 0) {

        while ($row = $result->fetch_assoc()) {

            echo "ID: " . $row["id"] . " - Sensor ID: " . $row["sensor_id"] . " - Temperature: " . $row["temperature"] . "°C - Humidity: " . $row["humidity"] . "% - Timestamp: " . $row["created_at"] . "<br>";

        }

    } else {

        echo "No data found.";

    }

} else {

    echo "Error executing query: “. $conn->error;

}
```

a) How Device Information Can Be Retrieved Using PHP

To retrieve device information from a database, you can execute a SELECT query. Here's an example:

```
PHP

<? PHP

// Database connection details

$host = 'localhost';

$dbname = 'iot_database';

$username = 'root';

$password = '';

try {
```

```
    // Create a new PDO connection
    $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    // Retrieve device information
    $deviceId = '12345';
    $sql = "SELECT * FROM devices WHERE device_id = :device_id";
    $stmt = $pdo->prepare($sql);
    $stmt->execute([':device_id' => $deviceId]);
    $device = $stmt->fetch(PDO::FETCH_ASSOC);
    if ($device) {
        echo "Device Name: " . $device['device_name'] . "<br>";
        echo "Device Type: " . $device['device_type'] . "<br>";
    } else {
        echo "Device not found.";
    }
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```

b) How to Fetch Data from an External API

To fetch data from an external API, you can use PHP's `cURL` or `file_get_contents()`. Here's an example using cURL

```
<? PHP
// URL of the external API
$apiUrl = 'https://api.example.com/data';
// Initialize cURL session
$ch = curl_init();
```

```
// Set cURL options
curl_setopt($ch, CURLOPT_URL, $apiUrl);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
// Execute cURL request
$response = curl_exec($ch);
// Check for errors
if ($response === false) {
   echo "cURL Error: " . curl_error($ch);
} else {
   // Decode JSON response
   $data = json_decode($response, true);
   print_r($data); // Display data
}
// Close cURL session
curl_close($ch);
?>
```

c) How Device Status Can Be Checked Using PHP

Assuming device status is stored in the database, you can retrieve and check it like this:

PHP

```
<?PHP
// Database connection (reuse from the previous examples)
$host = 'localhost';
$dbname = 'iot_database';
$username = 'root';
$password = '';
try {
```

```
    // Create a new PDO connection
    $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    // Check device status
    $deviceId = '12345';
    $sql = "SELECT status FROM devices WHERE device_id = :device_id";
    $stmt = $pdo->prepare($sql);
    $stmt->execute([':device_id' => $deviceId]);
    $device = $stmt->fetch(PDO::FETCH_ASSOC);
    if ($device) {
        echo "Device Status: " . $device['status'];
    } else {
        echo "Device not found.";
    }
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```

d) How Alert Records Can Be Accessed Using PHP

To access alert records, you can query the database for alerts related to a specific device or condition:

```
<? PHP
// Database connection (reuse from the previous examples)
$host = 'localhost';
$dbname = 'iot_database';
$username = 'root';
$password = '';
```

```
try {
   // Create a new PDO connection
   $pdo = new PDO ("mysql:host=$host;dbname=$dbname", $username, $password);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   // Retrieve alert records
   $deviceId = '12345';
   $sql = "SELECT * FROM alerts WHERE device_id = :device_id ORDER BY timestamp DESC";
   $stmt = $pdo->prepare($sql);
   $stmt->execute([':device_id' => $deviceId]);
   $alerts = $stmt->fetchAll(PDO::FETCH_ASSOC);
   if ($alerts) {
      foreach ($alerts as $alert) {
         echo "Alert ID: " . $alert['alert_id'] . "<br>";
         echo "Alert Message: " . $alert['message'] . "<br>";
         echo "Timestamp: " . $alert['timestamp'] . "<br><br>";
      }
   } else {
      echo "No alerts found for this device.";
   }
} catch (PDOException $e) {
   echo "Error: " . $e->getMessage();
}
?>
```

Practical Activity 1.4.5: Executing UPDATE Query

Task:

1: Read key reading 1.4.5

2: Referring to key reading 1.3.2, As web developer (IoT), you are asked to go to the computer lab to Execute UPDATE Query.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.4.5: Executing UPDATE Query

Steps to Execute an UPDATE Query in PHP

step1. Set Up Your Environment

- Ensure you have a web server (like Apache or Nginx) and PHP installed.

- Ensure you have access to a database (like MySQL or MariaDB) where you want to perform the update.

step 2. Connect to the Database

- Use the following code to establish a connection to your database:

PHP

```
try {
$pdo = new PDO('mysql:host=your_host;dbname=your_database', 'username', 'password');
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}
```

step 3. Prepare the UPDATE SQL Statement

- Write the SQL statement for the update operation. For example, if you want to update a device's name and type in a `devices` table:

PHP

```
$sql = "UPDATE devices SET device_name = :device_name, device_type = :device_type WHERE device_id = :device_id";
```

step 4. Prepare the Statement

- Prepare the SQL statement to prevent SQL injection attacks:

PHP

```
$stmt = $pdo->prepare($sql);
```

step 5. Bind Parameters

- Bind the parameters to the SQL query. This step is crucial to ensure that the values are safely inserted into the query:

PHP

```
$deviceId = 'your_device_id'; // Replace with the actual device ID
$newDeviceName = 'Updated Device Name';
$newDeviceType = 'Updated Device Type';
$stmt->bindParam(':device_name', $newDeviceName);
$stmt->bindParam(':device_type', $newDeviceType);
$stmt->bindParam(':device_id', $deviceId);
```

step 6. Execute the Query

- Execute the prepared statement to perform the update:

PHP

```
try {
    $stmt->execute();
    echo "Device information updated successfully.";
} catch (PDOException $e) {
    echo 'Error updating device information: ' . $e->getMessage();
}
```

step 7. Close the Connection

- After completing the operation, it's good practice to close the database connection $pdo = null;

Complete Example Code

Here's a complete example of how the code might look for updating device information:

PHP

<? PHP

try {

// Connect to the database

Demonstration of update query in PHP

1: Set Up Your Environment

1. Install a Web Server: Use a web server like Apache or Nginx. For local development, you can use packages like XAMPP, WAMP, or MAMP.

2. Install PHP: Ensure PHP is installed on your server. Most web server packages come with PHP pre-installed.

3. Install a Database: MySQL or MariaDB is commonly used. Ensure it is installed and running.

2: Create a Database and Table

1. Access MySQL: Use PHPMyAdmin or the MySQL command line to create your database.

2. Create a Database:

sql

CREATE DATABASE iot_database;

3. Create a Table:

sql

USE iot_database;

CREATE TABLE sensor_data (

id INT AUTO_INCREMENT PRIMARY KEY,

sensor_id VARCHAR(50) NOT NULL,

```
    temperature FLOAT,
    humidity FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3: Connect to the Database in PHP

1. Create a PHP File: Create a file named `db.PHP` to handle the database connection.

```
PHP
<?PHP
$host = 'localhost';
$db = 'iot_database';
$user = 'your_username'; // Replace with your MySQL username
$pass = 'your_password'; // Replace with your MySQL password
// Create a connection
$conn = new mysqli($host, $user, $pass, $db);
// Check the connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
?>
```

4: Create the Update Query

1. Create Another PHP File: Create a file named `update_data.PHP` to handle the data update.

```
<?PHP
include 'db.PHP'; // Include the database connection
// Sample data to update
$id = 1; // The ID of the record you want to update
$new_temperature = 28.5; // New temperature value
```

```
$new_humidity = 65; // New humidity value

// Prepare the SQL update query

$sql = "UPDATE sensor_data SET temperature = ?, humidity = ? WHERE id = ?";

// Prepare statement

$stmt = $conn->prepare($sql);

$stmt->bind_param("ddi", $new_temperature, $new_humidity, $id); // d = double, i = integer

// Execute the query

if ($stmt->execute ()) {

   echo "Record updated successfully.";

} else {

   echo "Error updating record: " . $stmt->error;

}

// Close the statement and connection

$stmt->close ();

$conn->close ();

?>
```

5.Modify Sensor Settings

Assuming you have a `sensor_settings` table, you can update sensor settings like this:

```
$sensorId = 'your_sensor_id'; // Example sensor ID

$newSettingValue = 'New Setting Value';

$sql = "UPDATE sensor_settings SET setting_value = :setting_value WHERE sensor_id = :sensor_id";

$stmt = $pdo->prepare($sql);

$stmt->bindParam(':setting_value', $newSettingValue);

$stmt->bindParam(':sensor_id', $sensorId);

try {
```

```
    $stmt->execute();
    echo "Sensor settings updated successfully.";
} catch (PDOException $e) {
    echo 'Error updating sensor settings: ' . $e->getMessage();
}
```

6.Update Device Firmware

To update firmware information, you might have a `firmware` table:

```
$deviceId = 'your_device_id';
$newFirmwareVersion = '1.0.1';
$sql = "UPDATE firmware SET version =: version WHERE device_id = :device_id";
$stmt = $pdo->prepare($sql);
$
```

7. Update Access Permission

Assuming you have a table called `user_permissions` with fields like `user_id`, `permission_level`, and `device_id`, here's how you can update access permissions:

```
<?PHP
try {
   // Connect to the database
   $pdo = new PDO('mysql:host=your_host;dbname=your_database', 'username', 'password');
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   // Prepare the SQL statement
   $sql = "UPDATE user_permissions SET permission_level = :permission_level WHERE user_id = :user_id AND device_id = :device_id";
   $stmt = $pdo->prepare($sql);
   // Bind parameters
   $userId = 'example_user_id'; // Replace with actual user ID
   $deviceId = 'example_device_id'; // Replace with actual device ID
```

```
    $newPermissionLevel = 'admin'; // Replace with the new permission level
    $stmt->bindParam(':permission_level', $newPermissionLevel);
    $stmt->bindParam(':user_id', $userId);
    $stmt->bindParam(':device_id', $deviceId);
    // Execute the query
    try {
        $stmt->execute ();
        echo "Access permission updated successfully.";
    } catch (PDOException $e) {
        echo 'Error updating access permission: ' . $e->getMessage();
    }
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
} finally {
    $pdo = null; // Close the connection
}
?>
```

8. Update Alert and Notification Settings

Assuming you have a table called `notification_settings` with fields like `user_id`, `alert_type`, and `is_enabled`, here's how you can update alert and notification settings.

```
<?PHP
try {
    // Connect to the database
    $pdo = new PDO('mysql:host=your_host;dbname=your_database', 'username', 'password');
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    // Prepare the SQL statement
```

```
    $sql = "UPDATE notification_settings SET is_enabled = :is_enabled WHERE user_id
= :user_id AND alert_type = :alert_type";
    $stmt = $pdo->prepare($sql);
    // Bind parameters
    $userId = 'example_user_id'; // Replace with actual user ID
    $alertType = 'email'; // Replace with the alert type (e.g., email, SMS)
    $isEnabled = 1; // 1 for enabled, 0 for disabled
    $stmt->bindParam(':is_enabled', $isEnabled);
    $stmt->bindParam(':user_id', $userId);
    $stmt->bindParam(':alert_type', $alertType);
    // Execute the query
    try {
        $stmt->execute();
        echo "Alert and notification settings updated successfully.";
    } catch (PDOException $e) {
        echo 'Error updating alert settings: ' . $e->getMessage();
    }
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
} finally {
    $pdo = null; // Close the connection
}
?>
```

9: Test the Update Query

1. Run the Script: Open your web browser and navigate to `http://localhost/your_project_folder/update_data.PHP`. This should execute the script, and you should see a success message if the update was successful. If the record with the specified ID does not exist, you will receive an error message.

Practical Activity 1.4.6: Executing Delete Query and error handling

Task:

1: Read key reading 1.4.6

2: Referring to key reading 1.4.6, As web developer (IoT), you are asked to go to the computer lab to Execute Delete Query and error handling.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.4.6: Executing Delete Query and error handling

In IoT web application development using PHP here are the procedures of Delete Query and error handling:

Step 1: Set Up Your Environment

1. Install a Web Server: Use a web server like Apache or Nginx. For local development, you can use packages like XAMPP, WAMP, or MAMP.
2. Install PHP: Ensure PHP is installed on your server. Most web server packages come with PHP pre-installed.
3. Install a Database: MySQL or MariaDB is commonly used. Ensure it is installed and running.

Step 2: Create a Database and Table

1. Access MySQL: Use PHPMyAdmin or the MySQL command line to create your database.
2. Create a Database:

```
sql
CREATE DATABASE iot_database;
```

3. Create a Table:

```
sql
USE iot_database;
CREATE TABLE sensor_data (
    id INT AUTO_INCREMENT PRIMARY KEY,
    sensor_id VARCHAR(50) NOT NULL,
    temperature FLOAT,
    humidity FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Connect to the Database in PHP

1. Create a PHP File: Create a file named `db.PHP` to handle the database connection

```
<?PHP
$host = 'localhost';
$db = 'iot_database';
$user = 'your_username'; // Replace with your MySQL username
$pass = 'your_password'; // Replace with your MySQL password
// Create a connection
$conn = new mysqli($host, $user, $pass, $db);

// Check the connection
if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}
?>
```

Step 4: Create the Delete Query

1. Create Another PHP File: Create a file named `delete_data.PHP` to handle the data deletion.

PHP

```
<?PHP
include 'db.PHP'; // Include the database connection
// ID of the record to delete
$id = 1; // Replace with the ID of the record you want to delete
// Prepare the SQL delete query
$sql = "DELETE FROM sensor_data WHERE id = ?";
// Prepare statement
$stmt = $conn->prepare($sql);
$stmt->bind_param("i", $id); // i = integer
// Execute the query
if ($stmt->execute()) {
   if ($stmt->affected_rows > 0) {
      echo "Record deleted successfully.";
   } else {
      echo "No record found with the specified ID.";
   }
} else {
   echo "Error deleting record: " . $stmt->error;
}
// Close the statement and connection
$stmt->close ();
$conn->close ();
?>
```

Step 5: Prepare the DELETE SQL Statement

- Write the SQL statement for the delete operation. For example, if you want to delete a device from a `devices` table:

PHP

```
$sql = "DELETE FROM devices WHERE device_id = :device_id";
```

Step6.Prepare the Statement

- Prepare the SQL statement to prevent SQL injection attacks:

PHP

```
$stmt = $pdo->prepare($sql);
```

Step7.Bind Parameters

- Bind the parameters to the SQL query. This step is crucial to ensure that the values are safely inserted into the query:

PHP

```
$deviceId = 'your_device_id'; // Replace with the actual device ID
$stmt->bindParam(':device_id', $deviceId);
```

Step8.Execute the Query

- Execute the prepared statement to perform the deletion:

PHP

```
try {
    $stmt->execute();
    if ($stmt->rowCount() > 0) {
        echo "Device deleted successfully.";
    } else {
        echo "No device found with the given ID.";
    }
} catch (PDOException $e) {
    echo 'Error deleting device: '. $e->getMessage();
}
```

Step9.Close the Connection

- After completing the operation, it's good practice to close the database connection:

PHP

```
$pdo = null;
```

Complete Example Code

Here's a complete example of how the code might look for deleting a device:

PHP

```
<? PHP
try {
   // Connect to the database
```

$pdo = new PDO ('mysql:host=your_host;dbname=your_database', 'username', 'password');

Step 10: Test the Delete Query

1. Run the Script: Open your web browser and navigate to `http://localhost/your_project_folder/delete_data.PHP`. This should execute the script, and you should see a success message if the deletion was successful. If there is no record with the specified ID, you will receive a message indicating that.

Step 11: Error Handling

Implementing error handling is essential for a robust application. Here are some strategies:

1. Check Connection Errors: Ensure that the database connection is established correctly, as shown in the `db.PHP` file.
2. Check for Affected Rows: After executing the delete query, check if any rows were affected using `$stmt->affected_rows`. This helps determine if the deletion was successful or if the specified ID was not found.
3. Use Try-Catch Blocks: For more complex applications, consider using try-catch blocks for error handling. This can help catch exceptions and handle them gracefully.

```
PHP
try {
   // Execute the query
   if ($stmt->execute()) {
      if ($stmt->affected_rows > 0) {
         echo "Record deleted successfully.";
      } else {
         echo "No record found with the specified ID.";
      }
   } else {
      Throw
```

i. Examples of Execute DELETE Query

a) Remove Sensor Data in IoT Web Application Using PHP

To remove sensor data, you would typically use a DELETE SQL query. Here's an example:

PHP

<?PHP

// Database connection details

```
$host = 'localhost';

$dbname = 'iot_database';

$username = 'root';

$password = '';

try {

   // Create a new PDO connection

   $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);

   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

   // Remove sensor data

   $sensorId = '67890';

   $sql = "DELETE FROM sensor_data WHERE sensor_id = :sensor_id";

   $stmt = $pdo->prepare($sql);

   $stmt->execute ([':sensor_id' => $sensorId]);

   echo "Sensor data removed successfully.";

} catch (PDOException $e) {

   echo "Error: " . $e->getMessage();

}

?>
```

b) Delete Command History and Delete Alert Configurations Using PHP

To delete command history and alert configurations, you can use similar DELETE queries:

```
<? PHP

// Database connection (reuse from the previous example)

$host = 'localhost';

$dbname = 'iot_database';

$username = 'root';

$password = '';
```

```
try {
   // Create a new PDO connection
   $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   // Delete command history
   $commandId = '123';
   $sqlCommand = "DELETE FROM command_history WHERE command_id = :command_id";
   $stmtCommand = $pdo->prepare($sqlCommand);
   $stmtCommand->execute([':command_id' => $commandId]);
   // Delete alert configurations
   $alertId = '456';
   $sqlAlert = "DELETE FROM alert_configurations WHERE alert_id = :alert_id";
   $stmtAlert = $pdo->prepare($sqlAlert);
   $stmtAlert->execute([':alert_id' => $alertId]);
   echo "Command history and alert configurations deleted successfully.";
} catch (PDOException $e) {
   echo "Error: " . $e->getMessage();
}
?>
```

c) Purge Inactive Devices in IoT Web Application Using PHP

To purge inactive devices, you might check for devices that haven't been active for a certain period and delete them:

```
<?PHP
// Database connection (reuse from the previous example)
$host = 'localhost';
$dbname = 'iot_database';
```

```
$username = 'root';

$password = '';

try {

   // Create a new PDO connection

   $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);

   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

   // Purge inactive devices

   $inactivePeriod = '2023-01-01'; // Example date, adjust as needed

   $sql = "DELETE FROM devices WHERE last_active < :inactive_period";

   $stmt = $pdo->prepare($sql);

   $stmt->execute([':inactive_period' => $inactivePeriod]);

   echo "Inactive devices purged successfully.";

} catch (PDOException $e) {

   echo "Error: " . $e->getMessage();

}

?>
```

ii. Define Error Handling and Give Its Code for Execution Using PHP

Error handling in PHP involves managing errors gracefully so that the application can continue running or fail safely. PHP provides several mechanisms for error handling, including try-catch blocks, custom error handlers, and logging.

Here's an example using try-catch blocks for error handling:

```
<?PHP

function handleDatabaseOperation() {

   // Database connection details

   $host = 'localhost';

   $dbname = 'iot_database';

   $username = 'root';
```

```
    $password = '';
    try {
        // Create a new PDO connection
        $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        // Example operation
        $sql = "SELECT * FROM devices";
        $stmt = $pdo->query($sql);
        // Fetch and process data
        $devices = $stmt->fetchAll(PDO::FETCH_ASSOC);
        foreach ($devices as $device) {
            echo "Device ID: " . $device['device_id'] . "<br>";
        }
    } catch (PDOException $e) {
        // Handle database-related errors
        error_log("Database error: " . $e->getMessage(), 3, 'errors.log');
        echo "An error occurred while accessing the database. Please try again later.";
    } catch (Exception $e) {
        // Handle general errors
        error_log("General error: " . $e->getMessage(), 3, 'errors.log');
        echo "An unexpected error occurred. Please try again later.";
    }
}
// Execute the function
handleDatabaseOperation();
?>
```

Points to Remember

- An API is a set of rules and protocols for building and interacting with software applications. It defines the methods and data formats that applications can use to communicate with each other, enabling different software systems to interact and share data.
- In IoT Web Application Development using PHP, the API Request Methods used are GET, POST, PUT, DELETE, PATCH.
- The REST(Representational State Transfer) is an architectural style for designing networked applications, SOAP (Simple Object Access Protocol): is a protocol used for exchanging structured information in web services and JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate.
- In development of Application Programming Interface CRUD Endpoints, it is required to execute CREATE query, READ query, UPDATE query and DELETE query.

1. Install a Web Server

2. Install PHP

Step 2: Create a Database and Table

Step 3: Connect to the Database in PHP

Step 4: Create the Query to Insert Data and execute

Step5: Prepare the Statement

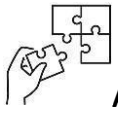
Step6: Bind Parameters

Step7: Handle the Response

Step8: Close the Connection

Step9: Error Handling

Step10: Testing



Application of learning 1.4.

XYZ Company is developing an IoT Device Management System to monitor and manage connected devices remotely. They require an Application Programming Interface **(**API**)** to handle device registration, data retrieval, updates to device settings, and device

removal. The API will follow REST principles and use JSON as the primary data format. The API must also handle errors efficiently.

As a developer, you are tasked with building this API using PHP and MySQL to perform CRUD (Create, Read, Update, Delete) operations on the IoT device database.

Indicative content 1.5: Secure API Endpoints.

Duration: 4 hrs

Theoretical Activity 1.5.1: Description of API Endpoints Secure

Tasks:

1: Answer the following questions:

i. What is authentication
ii. What are API security best practices?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.5.1.

Key readings 1.5.1.: Description of API Endpoints Secure

1. What is Authentication?

Authentication is the process of verifying the identity of a user, device, or entity attempting to access a system, service, or resource. It ensures that only authorized individuals or entities can access sensitive information or perform certain actions. Authentication typically involves the presentation of credentials, such as a username and password, biometric data, or a security token, which are then validated against the system's records.

Common authentication methods include:

Password-based Authentication: Using a combination of username and password.

Two-Factor Authentication (2FA): Requires two different types of credentials, such as a password and a one-time code sent to a mobile device.

Biometric Authentication: Uses unique biological characteristics, like fingerprints or facial recognition.

Token-based Authentication: Uses a token, often generated by an authentication server, to verify identity.

2. API Security Best Practices?

API security is crucial to protect data and ensure that only authorized users have access to the API's functionality. Here are some best practices for securing APIs:

1. Use HTTPS: Always use HTTPS to encrypt data in transit, preventing eavesdropping and man-in-the-middle attacks.

2. Implement Authentication: Require authentication to access API endpoints. Use secure methods like OAuth 2.0, API keys, or JWTs (JSON Web Tokens).

3. Use Rate Limiting: Implement rate limiting to prevent abuse and denial-of-service attacks by limiting the number of requests a client can make in a given time frame.

4. Validate Input: Always validate and sanitize user inputs to prevent injection attacks, such as SQL injection or cross-site scripting (XSS).

5. Use Strong Authentication Methods: Implement strong, multifactor authentication methods to enhance security.

6. Limit Data Exposure: Only expose necessary data fields and endpoints. Use filtering and pagination to limit the amount of data returned in responses.

7. Implement Access Control: Use role-based access control (RBAC) to ensure that users have access only to the API resources they need.

8. Monitor and Log API Activity: Keep detailed logs of API access and usage to detect and respond to suspicious activity.

9. Keep APIs Updated: Regularly update API components and dependencies to patch security vulnerabilities.

10. Use Security Headers: Implement security headers like Content Security Policy (CSP) and X-Content-Type-Options to protect against common web vulnerabilities.

Practical Activity 1.5.2: Implementing API authentication

Task:

1: Read key reading 1.5.2

2: Referring to key reading 1.5.2, As web developer (IoT), you are asked to go to the computer lab to implement API Authentication.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.5.2: Implementing API authentication

Steps for implementing API Authentication

1. Define Authentication Method

Decide on the authentication method you will use. Common methods include:

Token-based authentication (e.g., JWT - JSON Web Tokens)

OAuth 2.0

API Keys

For IoT applications, token-based authentication is often preferred due to its scalability and security.

2. Set Up Database for User/Device Management

Create a database to store user and device information, including credentials and tokens.

Example Schema:

sql

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
CREATE TABLE devices (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    device_id VARCHAR(50) NOT NULL UNIQUE,
    token VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

3. Create Registration Endpoint

Develop an API endpoint to register users/devices. This endpoint should:

- Validate input data.

- Hash the password using a secure hashing algorithm (e.g., `password_hash()`).

- Store user/device information in the database.

Example:

PHP

```
<?PHP
// registration.PHP
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
   $username = $_POST['username'];
   $password = password_hash($_POST['password'], PASSWORD_DEFAULT);
   // Insert into database
   // Use prepared statements to prevent SQL injection
   // ...
}
?>
```

4. Create Login Endpoint

Develop an API endpoint for users/devices to log in. This endpoint should:

- Validate credentials.

- Generate a token (e.g., JWT) upon successful authentication.

- Send the token back to the client for future requests.

Example:

```
<?PHP
// login.PHP
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
   $username = $_POST['username'];
```

```
    $password = $_POST['password'];
    // Retrieve user from database
    // Verify password
    if (password_verify($password, $hashed_password_from_db)) {
        // Generate JWT token
        $token = generate_jwt($user_id); // Implement JWT generation
        echo json_encode(['token' => $token]);
    } else {
        echo json_encode(['error' => 'Invalid credentials']);
    }
}
?>
```

5. Implement Token Generation

Use a library (like `firebase/PHP-jwt`) to generate JWT tokens. The token should include claims such as user ID, expiration time, and any other relevant information.

Example:

PHP

```
require 'vendor/autoload.PHP';
use \Firebase\JWT\JWT;
function generate_jwt($user_id) {
    $key = "your_secret_key";
    $payload = [
        'iat' => time(),
        'exp' => time() + (60 * 60), // Token valid for 1 hour
        'sub' => $user_id,
    ];
    return JWT: encode ($payload, $key);}
```

Practical Activity 1.5.3: Applying API Security Best Practices

Task:

1: Read key reading 1.5.3.

2: Referring to key reading 1.5.3., As web developer (IoT), you are asked to go to the computer lab to apply security best practices and the Steps and codes of starting using Raspberry Pi, Arduino and Node MCU in IoT web Application Development using PHP.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 1.5.3: Applying API Security Best Practices

Steps for API Security Best Practices in PHP

Step 1. Use HTTPS

- Always use HTTPS instead of HTTP to encrypt data transmitted between the client and server. This protects against man-in-the-middle attacks and ensures data confidentiality.

Step 2. Authentication and Authorization

- Implement robust authentication mechanisms (e.g., OAuth2, JWT) to verify the identity of users or devices accessing the API.
- Ensure that users have appropriate permissions to access specific resources by implementing role-based access control (RBAC).

Step 3. Input Validation and Sanitization

- Validate and sanitize all incoming data to prevent SQL injection, XSS (Cross-Site Scripting), and other injection attacks. Use prepared statements for database queries.

PHP

```
$stmt = $pdo->prepare("SELECT * FROM devices WHERE device_id = :device_id");
$stmt->bindParam(':device_id', $deviceId);
$stmt->execute();
```

Step 4. Rate Limiting

- Implement rate limiting to prevent abuse and denial-of-service attacks. This can be done by limiting the number of requests a user or device can make within a specified time frame.

Step 5. Error Handling

- Avoid exposing sensitive information in error messages. Use generic error messages for clients and log detailed errors on the server for debugging purposes.

```
PHP
try {
    // API logic
} catch (Exception $e) {
    http_response_code(500);
    echo json_encode(["error" => "An error occurred."]);
    // Log $e->getMessage() for debugging
}
```

Step 6. CORS (Cross-Origin Resource Sharing) Configuration

- Configure CORS policies to control which domains can access your API. This helps prevent unauthorized domains from making requests to your API.

```
PHP
header("Access-Control-Allow-Origin: https://yourdomain.com");
header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE");
```

Step 7. Use API Keys

- Generate and require API keys for clients to access your API. This adds an additional layer of security and allows you to track usage.
- Store API keys securely and rotate them periodically.

Step 8. Data Encryption

- Encrypt sensitive data both in transit (using HTTPS) and at rest (using database encryption). This ensures that even if data is compromised, it cannot be easily read.

Step 9. Implement Logging and Monitoring

- Log API requests and responses, including errors, to monitor for suspicious activity. Use monitoring tools to track API usage patterns and detect anomalies.

Step 10. Keep Software Updated

- Regularly update PHP, libraries

Developing an IoT web application using Raspberry Pi, Arduino, and NodeMCU with PHP involves several steps. Here's a general guide to help you get started:

1. Set Up Your Hardware:

- Raspberry Pi: Install the latest version of Raspberry Pi OS. You can use it as a server or a bridge between your IoT devices and the web application.

- Arduino: Use it to control sensors and actuators. You'll need the Arduino IDE to write and upload code.

- NodeMCU: This is a microcontroller with built-in Wi-Fi, perfect for IoT applications. Use the Arduino IDE to program it as well.

2. Connect Sensors and Actuators:

- Attach the necessary sensors (e.g., temperature, humidity) and actuators (e.g., motors, LEDs) to your Arduino or NodeMCU.

- Write the code to read from sensors and control actuators. For NodeMCU, you can use the ESP8266 or ESP32 libraries.

3. Set Up Communication:

- Raspberry Pi: You can set up a server using PHP and a web server like Apache or Nginx. Use PHP to handle HTTP requests and interact with a database.

- Arduino/NodeMCU: Use protocols like HTTP, MQTT, or WebSockets to send data to the Raspberry Pi. For HTTP, you can use simple GET or POST requests.

4. Develop the PHP Web Application:

- Create a web interface using HTML, CSS, and JavaScript to display sensor data and control actuators.

- Use PHP to handle backend logic, process incoming data from your IoT devices, and store it in a database like MySQL.

- Implement APIs using PHP to allow communication between your web application and IoT devices.

5. Database Integration:

- Set up a MySQL or SQLite database on your Raspberry Pi to store sensor data and logs.

- Use PHP's PDO or MySQLi extension to interact with the database.

6. Testing and Deployment:

- Test the entire setup locally to ensure all components communicate properly.

- Once tested, you can deploy your application to a cloud server if needed or use the Raspberry Pi as your main server.

7. Security Considerations:

- Make sure to secure your web application and IoT devices. Use HTTPS for encrypted communication and implement authentication mechanisms.

8. Additional Enhancements:

- Consider using frameworks like Laravel for PHP to streamline development.

- Implement real-time data updates using technologies like WebSockets.

Integrating APIs with Raspberry Pi, Arduino, and NodeMCU using PHP involves setting up a server on the Raspberry Pi to handle requests and sending data from the microcontrollers. Here's a step-by-step guide:

1. Raspberry Pi as a PHP Server

Set Up Apache and PHP:

1. Install Apache and PHP:

bash

sudo apt update

sudo apt install apache2 PHP libapache2-mod-PHP

2. Create a PHP API Endpoint:

Create a file named `api.PHP` in the `/var/www/html/` directory:

PHP

```
<?PHP
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $json = file_get_contents('PHP://input');
    $data = json_decode($json, true);
    // Process the data (e.g., save to a file or database)
    file_put_contents('data.txt', print_r($data, true), FILE_APPEND);
    echo json_encode(["status" => "success", "data_received" => $data]);
} else {
    echo json_encode(["status" => "error", "message" => "Only POST requests are allowed"]);
}
```

```
?>
```

2. Arduino to Collect Sensor Data

Use Arduino to Read Sensor Data:

1. Connect a Sensor:

Attach a sensor to the Arduino (e.g., a temperature sensor).

2. Arduino Sketch:

```
cpp
int sensorPin = A0; // Example sensor pin
int sensorValue = 0;
void setup() {
  Serial.begin(9600);
}
void loop() {
  sensorValue = analogRead(sensorPin);
  Serial.println(sensorValue);
  delay(2000); // Read every 2 seconds
}
```

3. NodeMCU to Send Data to Raspberry Pi API

Program NodeMCU to Make HTTP Requests:

1. Arduino IDE Sketch for NodeMCU:

```
include <ESP8266WiFi.h>
include <ESP8266HTTPClient.h>
const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";
const char* serverName = "http://your_rpi_ip/api.PHP";
void setup() {
  Serial.begin(115200);
```

```
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");
}
void loop() {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin(serverName);
    http.addHeader("Content-Type", "application/json");
    int sensorValue = analogRead(A0);  // Example sensor reading
    String jsonData = "{\"sensor_value\":" + String(sensorValue) + "}";
    int httpResponseCode = http.POST(jsonData);
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println(httpResponseCode);
      Serial.println(response);
    } else {
      Serial.print("Error on sending POST: ");
      Serial.println(httpResponseCode);
    }
    http.end();
  }
  delay(60000); // Send data every 60 seconds
}
```

- Raspberry Pi: Hosts a PHP server to handle incoming POST requests from NodeMCU.

- Arduino: Collects sensor data and can communicate with NodeMCU if needed.

- NodeMCU: Connects to Wi-Fi and sends sensor data as a JSON payload to the Raspberry Pi's PHP API.

Practical Activity 1.5.4: Managing API security

Task:

1: Read key reading 1.5.4.

2: Referring to key reading 1.5.4, As web developer (IoT), you are asked to go to the computer lab to manage API security.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary.

Key readings 1.5.4: Managing API security

Here are steps for managing API Security:

1. Use HTTPS

Always use HTTPS to encrypt data in transit. This prevents eavesdropping and man-in-the-middle attacks.

Example:

- Obtain an SSL certificate for your server.

- Ensure your web server is configured to redirect HTTP traffic to HTTPS.

2. Implement Authentication

Use secure authentication methods to verify the identity of users and devices. Common methods include:

Token-based authentication (e.g., JWT)

API keys

OAuth 2.0

Example:

- Implement a login endpoint that issues a JWT upon successful authentication.

3. Use Strong Password Policies

Enforce strong password policies for user accounts. This includes:

- Minimum length (e.g., 8 characters)

- Use of uppercase, lowercase, numbers, and special characters

- Regular password changes

Example:

PHP

```
if (strlen($password) < 8 || !preg_match('/[A-Z]/', $password) || !preg_match('/[0-9]/', $password)) {
   echo "Password must be at least 8 characters long and include uppercase letters and numbers.";
}
```

4. Rate Limiting

Implement rate limiting to prevent abuse and brute-force attacks. Limit the number of requests a user or device can make in a specified time frame.

Example:

PHP

```
// Simple rate limiting example
session_start();
if (!isset($_SESSION['requests'])) {
   $_SESSION['requests'] = 0;
}
$_SESSION['requests']++;
if ($_SESSION['requests'] > 100) { // Limit to 100 requests
   http_response_code(429); // Too Many Requests
   echo "Rate limit exceeded. Please try again later.";
```

exit;

}

5. Input Validation and Sanitization

Validate and sanitize all inputs to prevent SQL injection and XSS attacks. Use prepared statements for database queries.

Example:

PHP

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");

$stmt->execute(['username' => $username]);
```

` 6. Use CORS (Cross-Origin Resource Sharing)

If your IoT devices communicate with a web application hosted on a different domain, configure CORS to control which domains can access your API.

Example:

PHP

```
header("Access-Control-Allow-Origin: https://yourdomain.com");

header("Access-Control-Allow-Methods: GET, POST, OPTIONS");

header("Access-Control-Allow-Headers: Content-Type, Authorization");
```

7. Implement Logging and Monitoring

Log API requests and responses to monitor for suspicious activity. Use tools to analyze logs and set up alerts for unusual patterns.

Example:

PHP

```
file_put_contents('api_log.txt', date('Y-m-d H:i:s') . " - " . $_SERVER['REQUEST_URI'] . "\n", FILE_APPEND);
```

8. Use Security Headers

Implement security headers to protect against various attacks, such as XSS and clickjacking.

Example:

PHP

header("X-Content-Type-Options: nosniff");

header("X-Frame-Options: DENY");

header("X-XSS-Protection: 1; mode=block");

9. Regular Security Audits

Conduct regular security audits and penetration testing to identify and fix vulnerabilities in your API.

10. Keep Software Updated

Regularly update PHP, libraries, and server software to patch known vulnerabilities.

11. Implement Device Authentication

For IoT applications, ensure that devices authenticate themselves securely. Use unique device IDs and tokens.

Example:

- Generate and store unique tokens for each device during registration.

12. Use Firewalls and Security Groups

Configure firewalls and security groups to restrict access to your API server. Allow only trusted IP addresses or ranges.

Points to Remember

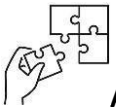
- the best practice for API security are implementing Authentication methods, use HTTPS, validate input, use rate limit, limiting data exposure, monitor and log API activities, keep API updated, Use Strong Authentication Methods.
  API authentication can be implemented in IoT web application development using the following steps:
  1. Define Authentication Method
  2. Set Up Database for User/Device Management
  3. Create Registration Endpoint
  4. Create Login Endpoint
  5. Implement Token Generation

  Step 1. Use HTTPS
  Step 2. Authentication and Authorization
  Step 3. Input Validation and Sanitization
  Step 4. Rate Limiting

Step 5. Error Handling
Step 6. CORS (Cross-Origin Resource Sharing) Configuration
Step 7. Use API Keys
Step 8. Data Encryption
Step 9. Implement Logging and Monitoring
Step 10. Keep Software Updated

- The steps for managing API security in IoT web application development using PHP are:

1. Use HTTPS
2. Implement Authentication
3. Use Strong Password Policies
4. Rate Limiting
5. Input Validation and Sanitization
6. Use CORS (Cross-Origin Resource Sharing)
7. Implement Logging and Monitoring
8. Use Security Headers
9. Regular Security Audits
11. Implement Device Authentication
12. Use Firewalls and Security Groups



Application of learning 1.5.

XYZ Company wants to create API endpoint necessary to connect IoT devices with a central server and ensure data is stored and processed effectively, you are requested to manage API security for an IoT web application.

Indicative content 1.6: Test Application Programming Interface Endpoint.

Duration: 4 hrs

Theoretical Activity 1.6.1: Description of API testing concept

Tasks:

1: Answer the following questions:

i. What are the API testing Tools?
ii. What are API testing best practices?
iii. Give the benefits of Application Programming Interface testing
iv. What are types of application programming interface endpoints testing

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.6.1.

Key readings 1.6.1.: Description of API testing concept

Certainly! Let's explore API testing tools, best practices, benefits, and the types of API endpoint testing.

1.API Testing Tools

There are numerous tools available for API testing, each with its own strengths. Here are some popular ones:

Postman: A widely used tool for testing APIs, offering a user-friendly interface for sending requests and analyzing responses. It supports automation and collaboration features.

SoapUI: A tool that supports both REST and SOAP APIs, providing comprehensive testing capabilities, including functional, security, and load testing.

REST Assured: A Java library for testing RESTful services, allowing developers to write readable and maintainable tests.

JMeter: Primarily used for performance and load testing, JMeter can also be configured for functional API testing.

Karate: A framework that combines API testing, mocking, and performance testing, allowing you to write tests in a domain-specific language.

Swagger (OpenAPI): While primarily a documentation tool, Swagger can be used to test APIs by generating mock servers and client SDKs.

2.API Testing Best Practices and Testing Steps

Best Practices:

1. Define Clear Objectives: Understand what you want to achieve with your testing, such as functionality, performance, or security.
2. Use Test Cases: Develop comprehensive test cases that cover all possible scenarios, including edge cases and error conditions.
3. Automate Tests: Automate repetitive tests to save time and reduce human error, especially for regression testing.
4. Mock External Dependencies: Use mocks to simulate external systems and isolate the API for more focused testing.
5. Validate All Aspects: Check not only the response data but also status codes, headers, and response times.

3. Benefits of Application Programming Interface Testing

1. Early Detection of Issues: API testing allows for earlier identification of defects, often before the GUI is developed, reducing the cost and effort of fixing them later.
2. Faster Testing: API tests are generally faster than UI tests, allowing for quicker feedback and more efficient testing cycles.
3. Improved Test Coverage: APIs often have complex logic, and testing them ensures that all functionalities are covered, leading to more robust applications.

4. Language Independence: APIs can be tested independently of the platform or programming language, providing flexibility in testing approaches.

5. Enhanced Security: By testing APIs, you can identify potential security vulnerabilities, such as unauthorized access or data leaks.

4. What are types of application programming interface endpoints testing.

1. Unit Testing

- Unit testing involves testing individual components or functions of the API in isola tion. The goal is to verify that each unit of the code performs as expected.
- Typically focuses on specific functions or methods that handle requests or respon ses.
- PHPUnit is commonly used for unit testing in PHP.
- Testing a function that processes input data or a method that formats a response.

2. Integration Testing

- **Definition:** Integration testing evaluates how different components of the API work together. This includes testing interactions between the API and external services, such as databases or third-party APIs.
- **Focus:** Ensures that the various modules or services interact correctly and that data flows as expected between them.
- Tools: PHPUnit, Guzzle, or Codeception can be used for integration testing.
- **Example:** Testing an API endpoint that retrieves data from a database and returns it in a specified format.

3. End-to-End Testing

- Definition: End-to-end testing simulates real user scenarios by testing the complete workflow of the application from start to finish. This includes making requests to the API and verifying the entire process, including data retrieval and storage.
- Focus: Ensures that the API, along with its front-end and back-end components, works together as intended.
- Tools: Tools like Behat or Codeception can be utilized for end-to-end testing.
- Example: Testing a user registration flow where a user submits their information, and the API processes the request and stores the data.

4. Load Testing

- Definition: Load testing assesses how the API performs under various load conditions. The goal is to determine the API's behavior when handling a specific number of requests over a period of time.
- Focus: Evaluates response times, throughput, and resource utilization when the API is subjected to high traffic.
- Tools: Tools like JMeter, Gatling, or Loader.io can be used for load testing.
- Example: Simulating multiple users making requests to the API simultaneously to see how it handles the increased load.

Practical Activity 1.6.2: Testing API accessibility

Task:

1: Read key reading 1.6.2.

2: Referring to key reading 1.6.2, you are required to go in a computer lab where the trainees should test API accessibility.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary.

Key readings 1.6.2: Testing API accessibility

Steps to test API accessibility

Testing API accessibility in PHP involves several steps to ensure that your API endpoints are reachable, respond correctly, and provide the expected data. Below are the key steps to test API accessibility:

Step 1: Set Up Your Testing Environment

1. Install Required Tools: Ensure you have PHP installed along with any necessary libraries or frameworks. You may also want to install tools like PHPUnit for testing.

2. Create a Testing Script: Create a PHP script that will handle the API requests and responses.

Step 2: Define the API Endpoints

Identify the API endpoints you want to test. For example:

- `GET /api/users`
- `POST /api/users`
- `GET /api/users/{id}`

Step 3: Write Test Cases

Create test cases for each of the API endpoints. You can use PHPUnit to structure your tests. Below is an example of how to test API accessibility using PHPUnit.

PHP

```
<?PHP
use PHPUnit\Framework\TestCase;

class ApiAccessibilityTest extends TestCase
{
   protected $apiBaseUrl = 'http://yourapi.com/api'; // Replace with your API base URL
   public function testGetUsers()
   {
      $response = file_get_contents($this->apiBaseUrl . '/users');
      $this->assertNotFalse($response, 'API is not accessible');

      $data = json_decode($response, true);
      $this->assertIsArray($data, 'Response is not in JSON format');
   }
   public function testPostUser()
   {
      $data = json_encode(['name' => 'John Doe', 'email' => 'john@example.com']);
      $options = [
         'http' => [
```

```
            'header' => "Content-Type: application/json\r\n",
            'method' => 'POST',
            'content' => $data,
        ],
    ];
    $context = stream_context_create($options);
    $response = file_get_contents($this->apiBaseUrl . '/users', false, $context);

    $this->assertNotFalse($response, 'API is not accessible');
    $this->assertEquals(201, http_response_code(), 'Expected HTTP status code 201');
  }
  public function testGetUserById()
  {
    $userId = 1; // Replace with a valid user ID
    $response = file_get_contents($this->apiBaseUrl . '/users/' . $userId);
    $this->assertNotFalse($response, 'API is not accessible');
    $data = json_decode($response, true);
    $this->assertArrayHasKey('id', $data, 'User ID not found in response');
  }
}
```

Step 4: Run the Tests

1. Run PHPUnit: Execute the test script using PHPUnit in your terminal or command prompt:

```
bash
./vendor/bin/PHPunit ApiAccessibilityTest.PHP
```

2. Check Results: Review the output to see if all tests passed. If any tests failed, investigate the reasons and make necessary adjustments to your API or tests.

Step 5: Validate Responses.

For each API endpoint, ensure that:

- The correct HTTP status codes are returned (e.g., 200 for success, 201 for resource creation, 404 for not found).
- The response format is as expected (e.g., JSON).
- The data returned matches the expected structure and content.

Step 6: Test Edge Cases

Consider testing edge cases to ensure robustness:

- Test with invalid or missing parameters.
- Test with unauthorized access (if applicable).
- Test for rate limiting if implemented.

Step 7: Automate Tests (Optional)

Testing API endpoints is crucial for ensuring that they function correctly and efficiently. Here's a breakdown of how to approach each type of testing:

a) Unit Testing

Purpose: Verify that individual components or functions of the API work as expected in isolation.

Tools: Use frameworks such as JUnit (Java), NUnit (C), or unittest/pytest (Python).

Approach:

Mock External Dependencies: Use mocking libraries (e.g., Mockito for Java, unittest.mock for Python) to simulate interactions with external systems like databases or external services.

Test Logic: Focus on the internal logic of the API, ensuring methods return the correct outputs for given inputs.

Edge Cases: Include tests for edge cases and handle exceptions to ensure robustness.

b) Integration Testing

Purpose: Test the interaction between different components or systems, ensuring they work together correctly.

Tools: Tools like Postman, REST Assured (Java), or pytest with integration testing capabilities are useful.

Approach:

Real Environment: Conduct tests in an environment that closely resembles production, using actual databases and services.

Data Flow Verification: Ensure data flows correctly between components and that integrations are functioning as intended.

End-to-End Scenarios: Test scenarios that involve multiple components interacting with each other.

c) End-to-End Testing

Purpose: Validate the entire application workflow, ensuring all components work together seamlessly.

Tools: Use tools like Cypress, Selenium, or TestCafe for web applications; Postman can also be used for API-centric end-to-end tests.

Approach:

Simulate User Journeys: Test complete workflows that a user might perform, involving multiple API endpoints.

System Behavior: Verify that the application behaves correctly from a user's perspective, covering all interactions with the API.

Cross-Component Testing: Ensure integration between frontend and backend components, checking that they communicate correctly.

d) Load Testing

Purpose: Assess how the API performs under heavy load to ensure it can handle high traffic without performance degradation.

Tools: Use tools like Apache JMeter, Gatling, or Locust.

Approach:

Simulate High Traffic: Generate a large number of requests to test how the API handles increased load.

Monitor Performance: Measure response times, throughput, and resource utilization to identify any bottlenecks.

Scalability: Ensure the API can scale and maintain performance under peak load conditions.

e) Endpoint Accessibility

Purpose: Ensure that API endpoints are accessible and respond correctly from different locations or environments.

Tools: Use tools like Postman, curl, or HTTPie for basic accessibility checks; services like Pingdom or Uptrends can monitor endpoint availability.

Approach:

Basic Connectivity Checks: Verify that endpoints are reachable and respond with the correct status codes (e.g., 200 OK).

Geographic Testing: Test accessibility from different geographic

Testing API accessibility is crucial for ensuring that your IoT web application functions correctly and that devices can communicate with the server as intended. Here are the steps to effectively manage and test API accessibility using PHP:

1. Set Up a Testing Environment

Create a separate testing environment that mimics your production environment. This helps to ensure that tests do not affect live data or services.

- Use tools like Docker to create isolated environments.

- Set up a staging server where you can deploy your application for testing.

2. Define API Endpoints

Clearly define the API endpoints that need to be tested. Document each endpoint's purpose, request methods (GET, POST, etc.), required parameters, and expected responses.

3. Use API Testing Tools

Utilize API testing tools to automate and streamline the testing process. Some popular tools include:

Postman: For manual testing and automation.

cURL: Command-line tool for making requests.

PHPUnit: For writing unit tests in PHP.

4. Write Test Cases

Create test cases for each API endpoint based on the defined specifications. Include positive and negative scenarios, such as:

- Valid requests

- Invalid requests (missing parameters, incorrect data types)

- Authentication failures

- Rate limiting scenarios

5. Manual Testing

Perform manual testing using tools like Postman or cURL to ensure that each API endpoint is accessible and behaves as expected.

Example using cURL:

bash

curl -X GET https://yourapi.com/endpoint -H "Authorization: Bearer your_token"

6. Automated Testing with PHPUnit

Write automated tests using PHPUnit for your PHP application. This allows you to run tests easily and repeatedly.

Example Test Case:

PHP

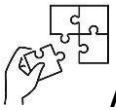
```
use PHPUnit\Framework\TestCase;

class ApiTest extends TestCase {
    public function testGetEndpoint() {
        $response = file_get_contents("https://yourapi.com/endpoint");
        $this->assertNotEmpty($response);
        $this->assertJson($response);
    }
    public function testPostEndpoint() {
        $data = json_encode(['key' => 'value']);
        $options = [
            'http' => [
                'header'  => "Content-type: application/json\r\n" .
                             "Authorization: Bearer your_token\r\n",
                'method'  => 'POST',
                'content' => $data,
            ],
        ];
        $context  = stream_context_create($options);
        $result = file_get_contents("https://yourapi.com/endpoint", false, $context);
        $this->assertEquals(200, http_response_code());
    }
}
```

Points to Remember

- There are numerous tools available for API testing, each with its own strengths. Here are some popular ones such as postman, soapUI, RESTassued, JMeter, swagger etc.
- Testing steps that are used set up environment, Define Endpoints, Create Test Scenarios etc.
- The types of application programming interface endpoint testing used is unit testing, integration testing, end to end testing, load testing.

  Step 1: Set Up Your Testing Environment
  Step 2: Define the API Endpoints
  Step 3: Write Test Cases
  Step 4: Run the Tests
  Step 5: Validate Responses
  Step 6: Test Edge Cases
  Step 7: Automate Tests (Optional)



Application of learning 1.6.

XYZ Company wants to create API endpoint necessary to connect IoT devices with a central server and ensure data is stored and processed effectively, you are requested to test API accessibility for an IoT web application.

Indicative content 1.7: Documentation of the API.

Duration: 2 hrs

Theoretical Activity 1.7.1: Identification of documentation tools for developed API requirements

Tasks:

1: Answer the following questions:

a) Define API versioning
b) What are the documentation tools for developed API
c) What is API specification (information) documentation?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 1.7.1.

Key readings 1.7.1.: Identification of documentation tools for developed API requirements

a) **Define API Versioning**

API Versioning is the practice of managing changes to an API in a way that allows developers to introduce new features, fix bugs, or make improvements without disrupting existing clients that rely on the current API. Versioning ensures backward compatibility and provides a clear path for clients to adopt new functionality at their own pace.

Common approaches to API versioning include:

URL Path Versioning: Including the version number in the URL path (e.g., `/v1/resource`).

Query Parameter Versioning: Specifying the version as a query parameter (e.g., `/resource? version=1`).

Header Versioning: Indicating the version in a custom HTTP header (e.g., `X-API-Version: 1`).

Content Negotiation: Using the `Accept` header to specify the version (e.g., `Accept: application/vnd.example.v1+json`).

b) What are the Documentation Tools for Developed API

There are several tools available that can help document APIs effectively, providing both static and interactive documentation. Some popular tools include:
Swagger (OpenAPI): A widely used tool that allows you to define your API in a standardized format and generate interactive documentation. Swagger UI provides a web-based interface for exploring the API.
Postman: A collaboration platform for API development that includes features for documenting APIs. Postman can generate documentation based on your API collections.

Apiary: A platform that offers API design, development, and documentation tools. It uses the API Blueprint format to create interactive documentation.
Redoc: A tool for generating beautiful, customizable API documentation from OpenAPI (Swagger) definitions.
Slate: An open-source tool that generates static API documentation. It uses Markdown to create clean and readable documentation.
Docusaurus: A static site generator that can be used to create comprehensive documentation sites, including API documentation.

c) What is API Specification (Information) Documentation

API Specification Documentation is a detailed description of an API's structure, behavior, and functionality. It serves as a blueprint for how the API operates and what developers can expect when interacting with it. The specification typically includes:

Endpoint Definitions: A list of all available API endpoints, including their paths and methods (e.g., GET, POST, PUT, DELETE).
Request Parameters: Details about the parameters that can be included in requests, including their names, types, and whether they are required or optional.
Response Structures: Descriptions of the data returned by the API

Practical Activity 1.7.2: Documentation of API endpoint

Task:

1: Read key reading 1.7.2.

2: Referring to key reading 1.7.2, As web developer (IoT), you are asked to go to the computer lab to document API Endpoint.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary.

Key readings 1.7.2: Documentation of API endpoint

- **Steps to document API Endpoint.**

Documenting your Application Programming Interface (API) endpoints is crucial for ensuring that developers understand how to use your API effectively. Below are the steps to create comprehensive API documentation, including authentication information, request parameters, response structure, error codes, error response formats, and usage guidelines.

Step 1: Choose a Documentation Format

Decide on the format for your documentation. Common formats include:

- Markdown files (e.g., README.md)

- Swagger/OpenAPI specifications

- HTML documentation

- PDF documents

Step 2: Provide an Overview of the API

Start with a brief introduction to your API, its purpose, and its intended audience. This section should include:

- A summary of what the API does.

- Key features and functionalities.

Step 3: Include Authentication Information

Detail the authentication methods required to access the API. This could include:

- Authentication Type: Describe the authentication method (e.g., API keys, OAuth 2.0, JWT).

- How to Obtain Credentials: Provide instructions on how users can obtain the necessary credentials (e.g., signing up for an account, generating API keys).

- Example Header: Include an example of how to include authentication in API requests.

Example:

plaintext

Authentication

To access the API, you must include an API key in the request header.

Header Example:

Authorization: Bearer YOUR_API_KEY

Step 4: Document Request Parameters

For each API endpoint, list the request parameters, including:

- Parameter Name: The name of the parameter.

- Type: The data type (e.g., string, integer, boolean).

- Required/Optional: Specify whether the parameter is required or optional.

- Description: A brief description of what the parameter does.

Example:

plaintext

Endpoint: GET /api/users

Request Parameters:

Parameter Type Required Description

1. page integer Optional The page number for pagination.
2. per_page integer Optional Number of users per page.
3. search string Optional Search term to filter users.

Step 5: Document Response Structure

Describe the structure of the response for each endpoint, including:

-Response Format.Specify the format (e.g., JSON).

- Fields: List the fields returned in the response, including their types and descriptions.

Example

plaintext

Response Structure:

json

{

```
 "data": [
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com"
  }
 ],
 "meta": {
  "current_page": 1,
  "total_pages": 10,
  "total_users": 100
 }
}Fields
 - `id`: integer - The unique identifier for the user.
 - `name`: string - The name of the user.
 - `email`: string - The email address of the user.
```

Step 6: Document Error Codes and Error Response Formats

List the possible error codes that the API may return, along with their meanings and example responses.Example

plaintext

Error Codes

Code: 400 Description: Bad Request, Example Response: {“error": "Invalid input”}

Code: 401 Description: Unauthorized, Example Response: {“error": "Authentication failed”}

Code: 404 Description: Not Found, Example Response: { "error": "User not found" }

Code: 500 Description: Internal Server Error, Example Response: { "error": "Unexpected error" }`

Step 7: Add Usage Guidelines

Provide guidelines on how to use the API effectively. This can include:

- Best practices for making requests.

- Rate limiting information (if applicable).

- Tips for handling errors and retries.

- Sample code snippets in various programming languages to demonstrate how to interact with the API.

Example

plaintext

Usage Guidelines

- Rate Limiting: Each API key is limited to 1000 requests per hour. Exceeding this limit will result in a 429 Too Many Requests error.

- Error Handling: Always check the response status code and handle errors gracefully. Implement retries for transient errors.

- Sample Code:

PHP

```
$apiKey = 'YOUR_API_KEY';

$response = file_get_contents('http://yourapi.com/api/users', false, stream_context_create ([

  'http' => [

    'header' => "Authorization: Bearer $apiKey"

  ]

]));
```

Step 8: Review and Update Regularly

Ensure that the documentation is reviewed for accuracy and updated regularly as the API evolves. Encourage feedback from users to improve clarity and usability.

Step 9: Publish the Documentation

Make the documentation accessible to users. You can host it on your website, use tools like GitHub Pages, or publish it as part of your API management platform.

Points to Remember

- API Versioning is the practice of managing changes to an API in a way that allows developers to introduce new features, fix bugs, or make improvements without disrupting existing clients that rely on the current API. Versioning ensures backward compatibility and provides a clear path for clients to adopt new functionality at their own pace
- Documentation tools used in development of API such as Swagger (OpenAPI), Postman, Apiary etc.
- The API Specification (Information) Documentation include Endpoint (GET, POST, PUT, DELETE), Request Parameters, Response Structures.

  Step 1: Choose a Documentation Format

  Step 2: Provide an Overview of the API

  Step 3: Include Authentication Information

  Step 4: Document Request Parameters

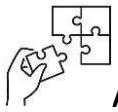
  Step 5: Document Response Structure

  Step 6: Document Error Codes and Error Response Formats

  Step 7: Add Usage Guidelines

  Step 8: Review and Update Regularly

  Step 9: Publish the Documentation



Application of learning 1.7.

XYZ Company wants to create API endpoint necessary to connect IoT devices with a central server and ensure data is stored and processed effectively, you are requested to document API endpoint for an IoT web application.

Learning outcome 1 end assessment

Written assessment

I. Multiple Choice Questions

Question 1:What is the primary purpose of an API?

A) To store data

B) To enable communication between different software systems

C) To create user interfaces

D) To manage databases

Question 2:Which HTTP method is typically used to retrieve data from an API?

A) POST

B) PUT

C) GET

D) DELETE

Question 3: What PHP function is commonly used to encode data into JSON format?

A) json_decode()

B) json_encode()

C) json_format()

D) json_stringify()

Question 4:Which of the following is a common way to secure an API?

A) Using plain text for passwords

B) Implementing API keys or tokens

C) Allowing all IP addresses to access the API

D) Disabling authentication

Question 5:What is the purpose of using a framework like Laravel or Slim for API development in PHP?

A) To avoid using PHP

B) To provide a structured way to build applications and APIs

C) To create HTML pages

D) To manage databases directly

Question 6:Which status code indicates that a request was successful and the server returned the requested data?

A) 404

B) 500

C) 200

D) 403

Question 7:What is the purpose of using middleware in a PHP API application?

A) To handle database connections

B) To process requests before reaching the main application logic

C) To render HTML pages

D) To manage user sessions

Question 8:Which of the following is a common tool for testing APIs?

A) MySQL

B) Postman

C) HTML

D) CSS

Question 9: When developing an API, what does CORS stand for?

A) Cross-Origin Resource Sharing

B) Common Object Resource System

C) Centralized Open Resource Service

D) Cross-Origin Resource Security

Question 10: What is the primary benefit of using RESTful principles in API design?

A) It allows for real-time communication.

B) It provides a standardized way to interact with resources.

C) It eliminates the need for authentication.

D) It allows for complex data queries.

II. Read the following statement related to developing APIs using PHP, and answer by True if the statement is correct or False if the statement is wrong

Question 1: APIs can only be developed using JavaScript and cannot be created using PHP.

Question 2: The HTTP method POST is typically used to create new resources in a RESTful API.

Question 3:JSON (JavaScript Object Notation) is the only format that can be used for data exchange in APIs.

Question 4:Using HTTPS instead of HTTP is important for securing API communication.

Question 5:API keys are a form of authentication that can be included in the request headers.

Question 6:It is unnecessary to validate user input when developing an API because the API is only accessed by trusted clients.

Question 7:The status code 404 indicates that the requested resource was not found on the server.

Question 8:Middleware in a PHP API can be used to handle tasks such as authentication, logging, and request modification.

Question 9:A RESTful API must always use the same URL structure for all endpoints.

Question 10:It is a good practice to document your API endpoints for better usability and integration by other developers.

Open questions

Question 1: Define IoT Web Application

Question 2: Identify Source of Data

Question 3: What is an API, and why is it important in web development?

Question 4: Explain the difference between RESTful APIs and SOAP APIs.

Question 5: Describe the different HTTP methods (GET, POST, PUT, DELETE) and their typical uses in RESTful APIs.

Question 6: How can PHP be used to create an API? What are the advantages of using PHP for API development?

Question 7: Discuss common security practices for API development, such as authentication and data validation.

Practical assessment

XYZ company want to creating endpoints that allow clients (like web applications or IoT devices) to interact with their server. As an IoT web application developer you are requested to create a simple RESTful API for managing tasks which will allow users to create, read, update, and delete.

References

Prabowo, Muhamad Cahyo Ardi, et al. "Development of an IoT-Based Egg Incubator with PID Control System and Web Application." *JOIV: International Journal on Informatics Visualization* 8.1 (2024): 465-472.

https://ori.hhs.gov/education/products/n_illinois_u/datamanagement/dctopic.html

Mohammad El-Basioni, Basma M., and Sherine M. Abd El-Kader. "Designing and modeling an IoT-based software system for land suitability assessment use case." *Environmental Monitoring and Assessment* 196.4 (2024): 380.

https://www.geeksforgeeks.org/introduction-to-internet-of-things-iot-set-1/

Azkiya, Khoirul, Muhamad Irsan, and Muhammad Faris Fathoni. "Implementation of App Engine and Cloud Storage as REST API on Smart Farm Application." *Sinkron: jurnal dan penelitian teknik informatika* 8.2 (2024): 902-910.


CSS
HTML

Indicative Contents

2.1 Identification of UI requirements

2.2 Use HTML tags

2.3 Application of CSS

Key Competencies for Learning Outcome 2: Develop User Interface.

Knowledge	Skills	Attitudes
• Identification of user interface requirements • Description of HTML tag • Description of CSS	• Conducting system Stakeholder to gather requirement • Creating User Personas from Data • Developing User Interface • Understanding html and PHP Basics • Installing and Configuring a html and PHP Environment • Setting Up a Local Development Environment • Connecting html and PHP to a Database • Understanding HTML Structure • Writing and Understanding HTML Tags • Understanding CSS Syntax • Applying Styling Properties	• Being Open-Mindedness from stakeholders • Having communication skills to clearly articulate CSS concepts • Having User-Centric Focus and behaviors of the end users. • Having commitment for Learning • Having Attention to Detail for installation and configuration steps • Having Curiosity to Maintain and understanding how PHP interacts with databases • Having Logical Thinking to understand how HTML tags fit into the overall document structure

Duration: 25 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Identify properly user interface requirement Used to develop user interface in IOT web application
2. Describe clearly HTML tag Used to develop user interface in IOT web application
3. Describe properly css Used to develop user interface in IOT web application
4. Installing and Configuring a HTML and PHP Environment Used to develop user interface in IOT web application
5. Applying Styling Properties Used to develop user interface in IOT web application
6. Understanding HTML and PHP Basics Used to develop user interface in IOT web application
7. Understanding CSS Syntax Used to develop user interface in IOT web application

Resources

Equipment	Tools	Materials
• Computer • Sensors • Arduino boards • Node MCU boards • Raspberry pi	• XAMMP • DBMS, MYSQL, postman etc • Web browser • Text editor (Sublime text, • Notepad++)	• Electricity • Internet

Indicative content 2.1: Identification of UI Requirements

Duration: 5 hrs

Theoretical Activity 2.1.1: Identification of User Interface Requirement

Tasks:

1: Answer the following questions:

a. What is user personas
b. What are Review System requirements?
c. Provide List key user services in develop user interface
d. Describe user experience (UX) best practices
e. What are the Features of a good user interface?

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 2.1.1.

Key readings 2.1.1: Identification of User Interface Requirement

What is the user interface in IoT?

This is done via user interface (UI). The user interface consists of the features by which a user interacts with a computer system. This includes screens, pages, buttons, icons, forms, etc. The most obvious examples of user interfaces are softwares and applications on computers and smartphones

Review System requirements

System requirements is a statement that identifies the functionality that is needed by a system in order to satisfy the customer's requirements. System requirements are a broad and also narrow subject that could be implemented to many items

The three requirements of the Internet of Things are connectivity, sensors, and intelligence. These enable IoT devices to be helpful and valuable but also bring specific challenges and considerations to address

When designing a user interface for an application, consider the following system requirements to ensure functionality, performance, and compatibility:

1. Platform Compatibility

- Ensure the UI works across different operating systems (e.g., Windows, macOS, Linux) and devices (e.g., mobile, desktop, tablets).

2. Hardware Specifications

- Design for varying device capabilities such as screen size, resolution, processing power, and memory.

3. Performance Optimization

- Ensure the UI does not cause performance bottlenecks. Use lightweight images, minimize animations, and optimize load times.

4. Browser and Device Support

- Test UI on multiple browsers (Chrome, Firefox, Safari) and device types to ensure accessibility and compatibility.

5. Security Features

- Implement secure authentication, encryption, and privacy measures, especially in applications handling sensitive data.

6. Scalability

- Design the UI to handle increased user loads without sacrificing performance or usability.

7. Accessibility Compliance

- Ensure the UI meets accessibility standards (e.g., WCAG) for users with disabilities (screen readers, color contrast, etc.).

2. List key user services

In an IoT (Internet of Things) web application, the user interface plays a crucial role in providing a seamless and intuitive experience for users interacting with connected devices and data. Here are key user services that a well-designed UI in an IoT web application should offer:

1.Device management

Add, remove, and configure IoT devices.

Monitor the status and health of connected devices.

View a list of all connected devices with relevant details.

2.data visualization

Real-time visualization of sensor data.

Historical data representation through charts, graphs, and other visualizations.

Ability to customize and filter data views.

3. alerts and notification

Set up alerts for specific device conditions or thresholds.

Receive real-time notifications for critical events.

View a log of past alerts and notifications.

4.remote control

Control IoT devices remotely (e.g., turning devices on/off, adjusting settings).

Implement two-way communication for devices that support it.

5.user authentication and authorization

Secure user authentication to access the IoT web application.

Granular user roles and permissions to control access to specific devices and features.

6.location tracking

Display the geographical location of devices on a map.

Track the movement of devices over time.

7.energy management

Monitor and manage the power consumption of connected devices.

Set power-saving preferences and schedules.

8.firmware updates

Initiate and track firmware updates for IoT devices.

Notify users about available updates and their importance.

9.integration with third-party services:

Support integration with other IoT platforms or external services.

Enable data sharing or interoperability with other smart devices and systems.

10.user preferences and testings

Allow users to customize their dashboard and preferences.

Provide settings for language, units, and other personalization options.

11.security features

Implement secure data transmission and storage practices.

Allow users to configure security settings for their devices.

Regularly update and patch software to address security vulnerabilities.

12.usage analytics

Provide insights into device usage patterns.

Analytics on energy consumption, user interactions, and device performance.

13.support and documentation

Offer help and support resources within the UI.

Provide documentation for users to troubleshoot issues.

14.scalability and performance

Ensure the UI can handle a large number of connected devices and users.

Optimize performance for responsiveness and speed.

15.mobile responsiveness:

Design the UI to be responsive and accessible on various devices, including smartphones and tablets.

3. Define user personas

Creating user personas is a valuable step in developing a user interface for an IoT web application. User personas help in understanding the needs, goals, and pain points of different user groups, allowing designers and developers to tailor the interface to meet those specific requirements.

4. Description of user experience (UX) best practices

User Experience (UX) best practices for defining a **user interface (UI)** focus on creating an intuitive, efficient, and enjoyable experience for users when interacting with application. When designing an interface, especially for web or mobile applications, these best practices ensure the UI is user-centered and aligns with the users' needs and behaviors.

Intuitive navigation is crucial for a positive user experience in web development. PHP, being a server-side scripting language, is commonly used in conjunction with HTML, CSS, and JavaScript to create dynamic and interactive web applications

Here's a detailed description of the key UX best practices for defining a user interface:

1. Understand User Needs (User-Centered Design)

User Research: Start by understanding who your users are, their goals, needs, and pain points. Use techniques like surveys, interviews, and user personas to create a clear picture of your target audience.

Empathy: Put yourself in the users' shoes by mapping out their journeys and identifying what tasks they need to complete. A user-centered design ensures the UI is focused on solving real problems in the most efficient and enjoyable way.

User Personas: Develop personas that represent your key user groups. These personas will guide your design decisions by reflecting the behaviors, frustrations, and preferences of your audience.

2. Simplicity and Clarity

Minimalism: Keep the UI simple by avoiding unnecessary elements. Focus only on the features and information that are essential to the user.

Clear Visual Hierarchy: Arrange content and elements in a way that naturally guides users to the most important actions or information. Use size, color, and placement to create a clear hierarchy.

Whitespace: Allow for sufficient whitespace (or negative space) between elements. This improves readability and reduces visual clutter, making it easier for users to focus on tasks.

Plain Language: Use simple, clear language for instructions, buttons, and notifications. Avoid jargon or overly technical terms that may confuse users.

3. Consistency

Design Patterns: Use consistent UI patterns throughout the application. This includes buttons, forms, navigation menus, and icons. Consistency helps users learn the interface faster and reduces cognitive load.

Uniform Layout and Colors: Maintain a uniform layout, color scheme, and typography throughout the app to ensure a cohesive look and feel. Inconsistent design elements can disorient users.

Platform Conventions: Adhere to established design guidelines for specific platforms (e.g., Android, iOS, web browsers). Users are familiar with certain conventions, and breaking them can lead to confusion.

4.2 intuitive navigation examples

Examples: 1. Navigation Bar with PHP Include: Use PHP include to create a modular navigation bar that can be included on all pages. This ensures consistency across the website.

2.Active Page Highlighting: Highlight the current page in the navigation bar to give users a visual cue of their location.

3.Breadcrumb Navigation: Implement breadcrumb navigation to show users the path from the homepage to the current page.

4.3 Consistent Design

Maintaining a consistent design in web development is crucial for providing a cohesive and user-friendly experience.

practices and examples for ensuring consistent design in web development using PHP

a) Separation of Concerns
b) Reusable Components with PHP Include
c) Consistent Styling with CSS
d) Responsive Design
e) Consistent Naming Conventions

4.4 Use Familiar UI Patterns

essential for providing a seamless and intuitive user experience

example:

a) Navigation Drawer or Sidebar Menu
b) Tabs for Content Organization
c) Form Validation and Feedback

4.5 Choose an attractive colour scheme

Choosing an attractive color scheme is subjective and depends on the specific context of your website or web application, including its purpose, target audience, and branding.

Such as: Primary Color, Secondary Color, Background Color, Text Color, Accent Color.
7.Features of a good user interface

A good user interface (UI) is crucial for a positive user experience in web development using PHP.

1. Intuitive Navigation:

- Navigation should be clear, logical, and easy to understand.

2. Consistency:

- Maintain a consistent design across all pages to provide a unified and cohesive user experience.

3. Responsive Design:

- Ensure your UI is responsive to different screen sizes and devices.

4. Clear Call-to-Action (CTA):

- Use distinct and visually appealing buttons for important actions.

5. Readable Typography:

- Choose readable fonts and font sizes.

6. Interactive Elements:

- Incorporate interactive elements like buttons, forms, and sliders to engage users.

7. Visual Hierarchy:

- Use visual cues such as color, size, and placement to create a clear hierarchy of information.

8. Error Handling and Validation:

- Clearly communicate errors and provide helpful feedback during form submissions.

9. Loading Feedback:

- Provide visual feedback during page loading to inform users that the system is processing their request.

10. Accessibility:

- Ensure your UI is accessible to users with disabilities.

8.Draw Wireframe

Creating a wireframe is a fundamental step in web development to outline the basic structure and layout of a webpage. Wireframes are simple, low-fidelity sketches that focus on the arrangement of elements without getting into detailed design or styling

II. How Do UX Best Practices Improve the Overall User Experience in a System?

UX best practices focus on creating intuitive and satisfying experiences for users, which leads to higher engagement, usability, and satisfaction.

1. Good UX Practices:

- User-Centered Design: Involve users in the design process to ensure their needs and pain points are addressed.
- Intuitive Navigation: Clear menus, easily recognizable icons, and logical flow help users find what they need quickly.
- Responsiveness: Ensure smooth performance and quick response to user interactions, reducing frustration.
- Feedback Mechanisms: Provide immediate feedback (e.g., loading animations, confirmation messages) to inform users their actions are recognized.

Examples:

- Good UX: A shopping website with a simple checkout process that gives feedback on each step (e.g., Amazon's one-click checkout).
- Bad UX: A website where users can't easily find key features, causing them to leave in frustration (e.g., slow-loading sites or overly complicated menus).

2. Benefits of Following UX Best Practices:

- Improved user satisfaction.
- Higher engagement and longer interaction time.
- Reduced bounce rates.
- Positive brand reputation through usability.

III. What Makes a UI Design Consistent Across Multiple Pages or Devices? Why Is Consistency Important?

1. Key Elements for Consistency:

- Design Language: Use a unified design language (e.g., colors, fonts, spacing, and layouts) across all pages or devices.
- Navigation Structure: Maintain consistent navigation across screens or devices so users don't get lost.
- Interaction Patterns: Keep interaction elements (e.g., buttons, forms) uniform in style and behavior.
- Responsive Design: Ensure your design adapts fluidly to different screen sizes without losing functionality or usability.

2. Importance of Consistency:

- User Familiarity: Users are more likely to understand and use the interface when it behaves in a predictable way.
- Reduced Cognitive Load: Consistency reduces the mental effort required to learn new UI patterns, leading to faster navigation and task completion.
- Professional Appearance: A consistent UI gives a polished, professional look, enhancing the brand's credibility.

IV. Features of a Good User Interface

A good user interface should balance aesthetics, usability, and functionality to enhance the user experience. The following are some features that define a good UI:

1. Intuitive Navigation:
   a. Navigation should be clear, logical, and easy to understand.
   2. Consistency:
   a. Maintain a consistent design across all pages to provide a unified and cohesive user experience.
   3. Responsive Design:
   a. Ensure your UI is responsive to different screen sizes and devices.
   4. Clear Call-to-Action (CTA):
   a. Use distinct and visually appealing buttons for important actions.

5. Readable Typography:

a. Choose readable fonts and font sizes.

Practical Activity 2.1.2: draw wireframe

Task:

1: Read key reading 2.1.2

2: Referring to key reading 2.1.2, As web developer (IoT), you are asked to go to the computer lab to provide steps to draw wireframe.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary.

Key readings 2.1.2: draw wireframe

Creating wireframes is an essential step in the user interface (UI) development process, as it helps you visualize the layout and functionality of your application before diving into the coding phase.

Here are the steps to draw a wireframe for developing a user interface in PHP:

Step 1: Define the Purpose and Scope

1. Identify Goals: Determine the primary goals of the application and what problems it aims to solve.

2. Target Audience: Understand who your users are and what their needs and expectations are from the application.

Step 2: Gather Requirements

1. List Features: Collaborate with stakeholders to list all the features that need to be included in the application.

2. User Stories: Create user stories to understand how different users will interact with the application.

Step 3: Choose Wireframing Tools

Select a wireframing tool that fits your needs. Some popular options include:

- Online Tools: Figma, Balsamiq, Sketch, or Adobe XD.

- Desktop Applications: Axure RP or Mockplus.

- Paper and Pen: Sometimes, sketching on paper can be the quickest way to brainstorm ideas.

Step 4: Create a Site Map

1. Outline the Structure: Create a site map that outlines the main sections and pages of your application.

2. Hierarchy: Define the hierarchy of information and how different pages' link to each other

Step 5: Start with Low-Fidelity Wireframes

1. Basic Layout: Begin with low-fidelity wireframes that focus on layout rather than details. Use simple shapes and placeholders to represent elements like headers, footers, buttons, and content areas.

2. Grid System: Consider using a grid system to maintain alignment and consistency across the wireframe.

Step 6: Focus on Key UI Elements

1. Navigation: Design the main navigation menu (e.g., top navigation, sidebar) to ensure intuitive access to different sections.

2. Content Areas: Define where content will be displayed, including text, images, and any interactive elements.

3. Forms and Inputs: Include forms for user input, such as login forms, search bars, or review submission forms

Step 7: Add Annotations

1. Explain Features: Add notes or annotations to explain the functionality of each element, such as buttons or links.

2.User Interactions: Describe how users will interact with specific elements (e.g., clicking a button will lead to a new page).

Step 8: Review and Iterate

1. Gather Feedback: Share your wireframe with stakeholders, team members, or potential users to gather feedback.

2. Make Revisions: Based on the feedback, make necessary revisions to improve usability and functionality.

Step 9: Create

Demonstration on how to draw wire frame while developing user interface in PHP

Creating a wireframe for a user interface in PHP typically involves using HTML, CSS, and possibly some JavaScript to visually represent the layout without getting into the actual design details. While PHP is primarily a server-side language, you can use it to generate the HTML structure for your wireframe. Here's a simple approach to get you started:

Step 1: Set Up Your Environment

Make sure you have a web server (like Apache or Nginx) and PHP installed on your machine. You can use tools like XAMPP or MAMP for an easy setup.

Step 2: Create a Basic PHP File

Create a new PHP file, e.g., `wireframe.PHP`, and start with a basic HTML structure.

```
<! DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Wireframe Example</title>
   <link rel="stylesheet" href="styles.css">
</head>
<body>
<header>
<h1>Header Area</h1>
</header>
<nav>
<ul>
<li>Home</li>
<li>About</li>
<li>Contact</li>
</ul>
```

```
</nav>
<main>
<section>
<h2>Main Content Area</h2>
<p>This is where the main content will go.</p>
</section>
<aside>
<h2>Sidebar</h2>
<p>This is the sidebar area.</p>
</aside>
</main>
<footer>
<p>Footer Area</p>
</footer>
</body>
</html>
```

Step 3: Add Some Basic Styles

Create a `styles.css` file to give your wireframe a simple layout. You can use borders and background colors to distinguish different sections.

```
css
body {
font-family: Arial, sans-serif;
margin: 0;
padding: 0;
}
header, nav, main, footer {
border: 1px solid ccc;
```

padding: 10px;

margin: 10px;

}

nav ul {

list-style-type: none;

padding: 0;

}

nav li {

display: inline;

margin-right: 10px;

}

main {

display: flex;

}

section {

flex: 3;

background-color: f0f0f0;

margin-right: 10px;

}

aside {

flex: 1;

background-color: e0e0e0;

}

Step 4: View Your Wireframe

Open your `wireframe.PHP` file in a web browser. You should see a simple layout with sections for the header, navigation, main content, sidebar, and footer.

Step 5: Iterate and Improve

You can continue to modify and expand your wireframe by adding more sections, adjusting styles, or including more interactive elements using JavaScript.

Points to Remember

- There are user experience (UX) best practices which are: Define user interface, Clear, Intuitive Navigation, Consistent Design, Use Familiar UI Patterns, Choose an attractive colour scheme.
- Here are the steps to draw a wireframe for developing a user interface in PHP

  Step 1: Define the Purpose and Scope
  Step 2: Gather Requirements

  Step 3: Choose Wireframing Tools

  Step 4: Create a Site Map

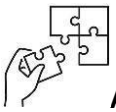
  Step 5: Start with Low-Fidelity Wireframes

  Step 6: Focus on Key UI Elements

  Step 7: Add Annotations

  Step 8: Review and Iterate

  Step 9: Create



Application of learning 2.1.

after understanding Identification of UI requirements: You are tasked with designing the user interface for a library management application. The app will be used by librarians, library members, and admin staff to manage book lending, returns, inventory, and user accounts.

Indicative content 2.2: Use HTML Tags

Duration: 10hrs

Theoretical Activity 2.2.1: Description of HTML Tag

Tasks:

1: Answer the following questions:

I. Provide the description of:
a) Html
b) Tag
c) Version of html?
d) Tag attribute
II.What are html tag categories.

2: Address any questions or concerns.

3: Present your findings to the whole class.

4: Ask trainee to read the key readings on activity 2.2.1

Key readings 2.2.1: Description of HTML Tag

1.Description of HTML

1. HTML, or Hypertext Mark-up Language, is the standard mark-up language used to create and design the structure of web pages. It is an essential component of web development and serves as the backbone for presenting content on the World Wide Web. HTML consists of a series of elements, each represented by tags, which define the structure and semantics of the content such as:(Elements: Tags, Attributes, Document Structure, Headings, Paragraphs, Lists, Links, Images, Forms)

2.Versions of html

- HTML evolves over time, and different versions have been released to introduce new features, improve existing ones, and enhance the overall capabilities of web development such as: (HTML 4(It included features like tables for layout and the introduction of forms.
-),XHTML(XHTML (Extensible Hypertext Mark-up Language) 1.0, introduced in 2000,

aimed to bring HTML to the world of XML.

-) 1.0,HTML5: introduced in 2014,It brought numerous new features, including semantic elements, native multimedia support, and APIs for enhanced functionality.

In HTML (HyperText Markup Language), a **tag** is a basic building block used to create elements that structure and present content on a web page. Tags are enclosed in angle brackets (< >) and typically come in pairs: an opening tag and a closing tag. The content placed between the tags is affected or displayed according to the tag's purpose.

Basic Structure of a Tag:

A tag consists of:

Opening tag: <tagname>

Closing tag: </tagname>

Some tags are self-closing and don't need a closing tag.

Example of a Paired Tag:

```html
<p>This is a paragraph.</p>
```

In this example:

 is the opening tag for a paragraph.

 is the closing tag for a paragraph.

The content between the opening and closing tags is the paragraph text: "This is a paragraph."

3.Tag attributes

In HTML, tag attributes provide additional information about HTML elements.

Attributes are always included in the opening tag of an HTML element and are specified as name-value pairs. common HTML tag attributes are:

1. class Attribute:

- Defines one or more class names for an HTML element.

2. id Attribute:

- Specifies a unique identifier for an HTML element.

3. style Attribute:

- Defines inline CSS styles for an HTML element.

4. src Attribute:

- Specifies the source URL or file path for external resources like images or scripts.

5. href Attribute:

- Defines the URL for hyperlinks, specifying the destination of the link.

6. alt Attribute:

- Provides alternative text for images, which is displayed if the image cannot be loaded.

7. width and height Attributes:

- Sets the width and height of an image (in pixels).

8. colspan and rowspan Attributes:

- Used in table cells to define the number of columns or rows a cell should span.

9. disabled Attribute:

- Disables an input, button, or select element.

10. placeholder Attribute:

- Provides a hint or example text for the expected input in a form field.

11. required Attribute:

- Specifies that an input field must be filled out before submitting a form.

12. checked Attribute:

- Pre-checks a checkbox or radio button by default.

4.HTML tag categories

categories of HTML tags are(Document Structure,Headings,Paragraph and Text Formatting,Lists,Links andAnchors,Images and Multimedia, Forms and Input Elements)

1.HTML Structural tags

HTML structural tags play a crucial role in organizing the content and defining the structure of a web page.

Here are some key HTML structural tags:

1. <header>

- Defines the header of a document or a section.

2. <footer>

- Represents the footer of a document or a section.

3. <nav>

- Defines a container for navigation links.

4. <main>

- Represents the main content of the document.

5. <section>

- Defines a section of content within a document

6.

- A generic container that does not carry any semantic meaning on its own.

2.Formatting tags

Formatting tags in HTML are used to control the visual presentation of text and elements on a web page

Here are some commonly used formatting tags:

a) <b> and :
b) <i> and :
c) <u>:
d) <s>:
e) <sub> and <sup>:

3. Table tags

Tables in HTML are used to organize and display data in rows and columns. Here are the key table-related tags in HTML

a) <table>
b) <tr>
c) <td>
d) <th>
e) <thead>

3.Form tags

HTML forms are used to collect user input on a webpage. The following are the key form-related tags in HTML

a) <form>
b) <input>
c) <label>
d) <textarea>
e) <select>

4. Heading tags

Heading tags in HTML are used to define headings and subheadings on a webpage.

a) <h1> to <h6>:
b) Semantic Usage

c) Styling with CSS
d) Heading Structure

5. List tags

HTML provides several tags for creating lists. There are two main types of lists: ordered lists (<ol>) and unordered lists (<ul>).

1. Ordered List (<ol>)
2. Unordered List (<ul>)

6. Media tags

HTML provides several tags for embedding media content such as images, audio, and video. Here are the key media-related tags:

a) Image (<img>)
b) Audio (<audio>)
c) Video (<video>)

7. Code tags

In HTML, the <code> tag is used to define a piece of computer code. It is typically used to display code snippets within the text of a document. Here's how you can use the <code> tag: like

a) Inline Code (<code>):
b) Code Blocks (<code> inside <pre>):
c) Styling with CSS

8. HTML frame tags

The use of frames in web development is for using CSS for layout and employing techniques like flexbox or grid.

a) <frameset>
b) <frame>
c) <noframes>

9. HTML Comment

In HTML, comments are used to include explanatory notes within the code that are not displayed on the webpage such as inline comment and a multi-line comment

10. Grouping tags (div, span)

In HTML, and are container elements used for grouping and styling content.

1. (Division):
   - Represents a generic container that divides the content into block-level sections.
2. :
   - Represents an inline container, often used to apply styles to smaller portions of text or elements within a larger block of content.

10. Hyperlink tag

In HTML, the hyperlink tag is represented by the <a> (anchor) element. The <a> element is used to create hyperlinks, allowing users to navigate to other pages or resources.

11. Semantic tags

Semantic HTML tags are elements that carry meaning about the structure and content of a webpage such as

a) <header>
b) <nav>
c) <main>
d) <section>

2. Application of CSS

CSS (Cascading Style Sheets) is a powerful styling language used to control the presentation and layout of HTML documents. various applications of CSS

a) Styling Text
b) Box Model and Layout
c) Backgrounds and Borders
d) Positioning and Flexbox/Grid
e) Transitions and Animations
f) Responsive Design
g) Fonts and Icons
h) Form Styling

18. Versions OF CSS

1. CSS1 (Cascading Style Sheets Level 1):

- The first version of CSS, published in 1996.
- Introduced basic styling properties and selectors.

2. CSS2 (Cascading Style Sheets Level 2):

- Published in 1998 and later revised in 2011 as CSS 2.1.
- Expanded the capabilities of CSS1 with new properties, positioning, and media types.
- Introduced features like absolute, relative, and fixed positioning.

3. CSS2.1:

- A revision of CSS2, aiming to clarify and correct errors in the specification.
- It became a stable recommendation by the W3C in 2011.

4. CSS3 (Cascading Style Sheets Level 3):

- CSS3 is not a single monolithic specification but a collection of separate modules, each covering specific features.

19 Style types

In CSS, style types refer to the ways in which styles can be applied to HTML elements are:

a) Inline Styles
b) Internal Styles (Embedded Styles)
c) External Styles
d) CSS Selectors
e) CSS @import

20. Use CSS Visual rules

CSS visual rules are styles and properties that control the appearance and layout of HTML elements, influencing how they are displayed on a webpage. Here are some common CSS visual rules and examples of their application:

a) Color:
b) Font Properties:
c) Margin and Padding:

d) Borders:
e) Background:
f) Box Shadow:
g) Text Alignment and Decoration:

21. Font

In CSS, you can control the font properties of text on your web page using various font-related properties. Here are some commonly used font-related CSS properties:

a) font-family
b) fontsize
c) font-weight
d) font-style
e) font-variant
f) line-height
g) text-align
h) text-decoration

22. Colors and background colors

In CSS, colors and background colors are specified using various properties such as

a) Text Color (color)
b) Background Color (background-color)
c) Opacity (opacity)
d) RGBA Colors
e) HSL Colors
f) Background Image (background-image)
g) Gradient Background
h) Text Shadow (text-shadow)

23. Opacity

In CSS, the opacity property is used to control the transparency of an element. It takes a value between 0 and 1, where 0 is fully transparent, 1 is fully opaque, and values in between represent varying levels of transparency.

24. Background image

In CSS, you can set a background image for an element using the background-

image property. This property allows you to specify the URL of an image file that will be used as the background for the selected element.

25. Use CSS Display and positioning

CSS display and position properties play a crucial role in controlling the layout and positioning of HTML elements on a webpage

a) relative:

- Positioned relative to its normal position in the document flow.

b) absolute:

- Positioned relative to its nearest positioned ancestor (if any), otherwise relative to the initial containing block.

c) fixed:

- Positioned relative to the browser window.

d) sticky:

- Acts like relative within its container until a given offset threshold is met, then it is treated as fixed.

e) Block

- Renders the element as a block-level element

26. Use CSS Box and Grid model

The CSS Box Model and Grid Layout are essential concepts for designing the layout of web pages. They provide a structured way to handle the positioning, sizing, and spacing of elements on a webpage

four main components:

1. Content:

- The actual content of the element, such as text, images, or other elements.

2. Padding:

- The space between the content and the element's border.

3. Border:

- A border surrounding the padding (if any).

4. Margin:

- The space between the border and adjacent elements

5. Border Radius:

The border-radius property is used to create rounded corners for an element.

6. Visibility:

The visibility property is used to control the visibility of an element. It can take values like visible, hidden, collapse, etc.

7. Auto:

The auto value is used in various properties to allow the browser to automatically compute a value. For example, it can be used with margin, width, height, etc.

An HTML tag is a special code used in HTML to define the structure and content of a webpage. HTML tags are enclosed in angle brackets (< >) and usually come in pairs (an opening tag and a closing tag). For example, is the opening tag for a paragraph, and is the closing tag. Tags tell the browser how to display the content inside them.

Structural Tags and Formatting Tags in HTML

- Structural Tags: Structural tags define the layout of the webpage and how content is organized. They are used to build the basic framework or structure of a webpage.

Examples:

- <header>: Defines the header section of a webpage.
- <nav>: Specifies navigation links.
- <main>: Represents the main content of the document.

- <footer>: Defines the footer section of a webpage.

Formatting Tags: Formatting tags are used to style the content inside the tags, such as making text bold, italic, or underlined.

Examples:

- <b>: Makes text bold.
- <i>: Italicizes text.
- <u>: Underlines text.

Main Attributes Used in HTML Tags

HTML attributes provide additional information about elements. They are placed inside the opening tag and have a name and a value.

- Common Attributes:

- class: Assigns a class name to an element, used for styling with CSS.
- id: Provides a unique identifier for an element.
- style: Adds inline CSS styles to an element.
- href: Used in <a> tags to specify the link destination.

● How to Create a Form in HTML?

A form in HTML allows users to enter and submit data to a server.

- <form>: Defines the form element.
- action: Specifies where to send the form data.
- method: Specifies the HTTP method (GET or POST).
- <input>: Collects user input (e.g., text, email).
- <label>: Associates a label with an input element.
- <submit>: Adds a submit button.

Why are Semantic Tags Important in HTML, and How Do They Differ from Non-Semantic Tags?

- Semantic Tags: These tags clearly define the meaning of the content within them. They improve the readability of code for both developers and search engines, helping to create more accessible and SEO-friendly websites.

Examples of Semantic Tags:

- <article>: Represents an independent piece of content, such as a blog post.
- <section>: Defines a section of the document.
- <aside>: Represents content aside from the main content, like a sidebar.
- <footer>: Defines the footer of a webpage.

Non-Semantic Tags: These tags do not convey meaning about the content they contain and are primarily used for layout purposes.

Examples of Non-Semantic Tags:

- : A generic container for block-level content.
- : A generic inline container for text or other elements.

Practical Activity 2.2.2: Using Html Tag

Task:

1: Read key reading 2.2.2.
2: Referring to key reading 2.2.2, In a computer lab setting, you are tasked with structuring a basic webpage using appropriate HTML tags.
3: Present your work to the trainer and whole class
4: Ask clarification where necessary.

Key readings 2.2.2: Using Html Tag

Steps of html structure tags

Using HTML structural tags is essential for creating a well-organized and semantically correct web page. Here's a step-by-step guide to help you understand and implement HTML structural tags effectively:

Step 1: Set Up Your HTML Document

Start by creating a basic HTML document structure. Open a text editor and create a file named `index.html`.

html

```
<! DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Your Web Page Title</title>
  <link rel="stylesheet" href="styles.css"> <!-- Optional CSS file -->
</head>
<body>
  <! -- Content will go here -->
</body>
</html>
```

Step 2: Use Structural Tags

Add structural tags within the `<body>` section to define the layout of your web page. Here are the main structural tags you should use:

1. Header (`<header>`): Contains introductory content or navigational links.

```
html
  <header>
    <h1>Website Title</h1>
    <nav>
      <ul>
        <li><a href="home">Home</a></li>
        <li><a href="about">About</a></li>
        <li><a href="contact">Contact</a></li>
      </ul>
    </nav>
  </header>
```

2. Main (`<main>`): Represents the main content of the document.

html

```
<main>
  <section id="home">
    <h2>Welcome to Our Website</h2>
    <p>This is the home section.</p>
  </section>
  <section id="about">
    <h2>About Us</h2>
    <p>Information about us goes here.</p>
  </section>
  <section id="contact">
    <h2>Contact Us</h2>
    <p>Contact details go here.</p>
  </section>
</main>
```

3.Footer (`<footer>`): Contains footer information, such as copyright and contact details.

html

```
<footer>
  <p>&copy; 2023 Your Website Name. All rights reserved.</p>
</footer>
```

Step 3: Add Additional Structural Tags

You can enhance your web page's structure using additional semantic tags

- Article (`<article>`): Represents a self-contained piece of content.

html

```
<article>
```

<h2>Blog Post Title</h2>

This is a blog post summary.

</article>

- Aside (`<aside>`): Contains content related to the main content, like sidebars or related links.

`html

<aside>

<h2>Related Links</h2>

Link 1

Link 2

</aside>

- Section (`<section>`): Defines sections in your document, useful for grouping related content.

Step 4: Final HTML Structure

Here's how your complete `index.html` might look:

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Your Web Page Title</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<header>

```
<h1>Website Title</h1>
<nav>
<ul>
<li><a href="home">Home</a></li>
<li><a href="about">About</a></li>
<li><a href="contact">Contact</a></li>
</ul>
</nav>
</header>
<main>
<section id="home">
<h2>Welcome to Our Website</h2>
<p>This is the home section.</p>
</section>
<section id="about">
<h2>About Us</h2>
<p>Information about us goes here.</p>
</section>
<section id="contact">
<h2>Contact Us</h2>
<p>Contact details go here.</p>
</section>
</main>
<footer>
<p>&copy; 2023 Your Website Name. All rights reserved.</p>
</footer>
</body>
```

```
</html>
```

`Step 5: Style Your Page

You can add a `styles.css` file to style your page and make it visually appealing. Here's an example of some basic CSS you might include:

```
css

body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
}

header, footer {
    background-color: f8f9fa;
    padding: 20px;
    text-align: center;
}

nav ul {
    list-style-type: none;
    padding: 0;
}

nav li {
    display: inline;
    margin-right: 15px;
}

main {
    padding: 20px;
}

section {
```

```
    margin-bottom: 20px;
}
```

Step 6: Test Your Web Page

Open your `index.html` file in a web browser to see your structured web page in action. You can make further adjustments and enhancements as needed.

Steps of Formatting tags tags

Formatting tags in HTML are used to control the appearance of text and other elements on a web page. These tags help you create visually appealing content by applying styles such as bold, italic, underline, and more. Here's a step-by-step guide to using formatting tags while developing a web page:

tep 1: Set Up Your HTML Document

Create a new HTML file named `formatting-example.html` and set up the basic HTML structure.

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formatting Tags Example</title>
    <link rel="stylesheet" href="styles.css"> <!-- Optional CSS file -->
</head>
<body>
    <!-- Content will go here -->
</body>
</html>
```

Step 2: Add Text Content with Formatting Tags

Within the `<body>` section, you can use various formatting tags to style your text. Here are some commonly used formatting tags:

1. Bold Text (`<strong>` or `<b>`): Use `<strong>` for semantically important text and `<b>` for visual styling.

html

```
<p>This is a <strong>strong</strong> statement. </p>

<p>This is <b>bold</b> text.</p>
```

2. Italic Text (`<em>` or `<i>`): Use `<em>` for emphasized text and `<i>` for visual styling.

html

```
<p>This is an <em>emphasized</em> statement.</p>

<p>This is <i>italic</i> text.</p>
```

3. Underlined Text (`<u>`): Use this tag to underline text.

html

```
<p>This is <u>underlined</u> text.</p>
```

4. Strikethrough Text (`<s>` or `<del>`): Use `<s>` for text that is no longer accurate or relevant, and `<del>` for deleted text.

html

```
<p>This is <s>strikethrough</s> text.</p>

<p>This is <del>deleted</del> text.</p>
```

5. Small Text (`<small>`): Use this tag to display smaller text.

html

```
<p>This is <small>small</small> text.</p>
```

6. Superscript (`<sup>`) and Subscript (`<sub>`): Use these tags for chemical formulas or mathematical expressions.

html

```
<p>Water is represented as H<sub>2</sub>O.</p>

<p>Einstein's theory is represented as E=mc<sup>2</sup>.</p>
```

Step 3: Combine Formatting Tags

You can also combine multiple formatting tags to achieve the desired effect. Here's an example:

html

This is a bold and emphasized statement.

This is <u>underlined and bold</u> text.

Step 4: Final HTML Structure

Here's how your complete `formatting-example.html` might look:

html

<! DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Formatting Tags Example</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<h1>Formatting Tags Example</h1>

This is a strong statement.

This is an emphasized statement.

This is <u>underlined</u> text.

This is <s>strikethrough</s> text.

This is <small>small</small> text.

Water is represented as H₂O.

Einstein's theory is represented as E=mc².

This is a bold and emphasized statement.

This is <u>underlined and bold</u> text.

</body>

</html>

Step 5: Style Your Page (Optional)

You can add a `styles.css` file to customize the appearance of your text. Here's an example of some basic CSS you might include:

```
css

body {
   font-family: Arial, sans-serif;
   margin: 20px;
}
h1 {
   color: 333;
}
p {
   font-size: 16px;
   line-height: 1.5;
}
strong {
   color: d9534f; /* Bootstrap primary color */
}
em {
   color: 5bc0de; /* Bootstrap info color */
}
```

Step 6: Test Your Web Page

Open your `formatting-example.html` file in a web browser to see the formatted text in action. You can adjust the styles and formatting as needed.

Step for table tags

Creating a table in a web page using HTML involves a few straightforward steps. Here's a simple guide to help you get started:

Step 1: Set Up Your HTML Document

Start by creating a basic HTML structure. You can use any text editor to write your HTML code.

html

<! DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My Table Page</title>

</head>

<body>

</body>

</html>

Step 2: Add the Table Tags

Inside the `<body>` tag, you can insert your table. Use the `<table>` tag to create the table, and inside it, you can define rows and cells.

html

<table>

<tr>

<th>Header 1</th>

<th>Header 2</th>

<th>Header 3</th>

</tr>

<tr>

<td>Row 1, Cell 1</td>

<td>Row 1, Cell 2</td>

<td>Row 1, Cell 3</td>

</tr>

```
    <tr>
        <td>Row 2, Cell 1</td>
        <td>Row 2, Cell 2</td>
        <td>Row 2, Cell 3</td>
    </tr>
</table>
```

Step 3: Style Your Table (Optional)

You can add CSS to style your table. You can do this either in a `<style>` tag in the `<head>` section or in an external stylesheet.

html

```
<style>
table {
width: 100%;
border-collapse: collapse;
}
th, td {
  border: 1px solid black;
padding: 8px;
text-align: left;
    }
    th {
        background-color: f2f2f2;
    }
</style>
```

Step 4: Complete Example

Here's how your complete HTML document might look:

html

```
<! DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>My Table Page</title>
   <style>
      table {
         width: 100%;
         border-collapse: collapse;
      }
      th, td {
         border: 1px solid black;
         padding: 8px;
         text-align: left;
      }
      th {
         background-color: f2f2f2;
      }
   </style>
</head>
<body>
   <table>
      <tr>
         <th>Header 1</th>
         <th>Header 2</th>
         <th>Header 3</th>
```

```
    </tr>
    <tr>
      <td>Row 1, Cell 1</td>
      <td>Row 1, Cell 2</td>
      <td>Row 1, Cell 3</td>
    </tr>
    <tr>
      <td>Row 2, Cell 1</td>
      <td>Row 2, Cell 2</td>
      <td>Row 2, Cell 3</td>
    </tr>
  </table>
</body>
</html>
```

Step 5: View Your Table

Save your HTML file and open it in a web browser to see your table displayed.

Step for Form tags

Creating a form in a web page using HTML is an essential skill for collecting user input. Here's a step-by-step guide to help you set up a basic form:

Step 1: Set Up Your HTML Document

Start with a basic HTML structure. You can use any text editor to create your HTML file.

```
html
<! DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

<title>My Form Page</title>

</head>

<body>

</body>

</html>

Step 2: Add the Form Tag

Inside the `<body>` tag, you can create a form using the `<form>` tag. You can specify attributes like `action` (where the form data will be sent) and `method` (how the data will be sent).

html

<form action="/submit" method="post">

<!-- Form elements will go here -->

</form>

Step 3: Add Form Elements

Inside the `<form>` tag, you can add various input elements such as text fields, radio buttons, checkboxes, and buttons.

Here's an example with several common input types:

html

<form action="/submit" method="post">

<label for="name">Name:</label>

<input type="text" id="name" name="name" required>

<label for="email">Email:</label>

<input type="email" id="email" name="email" required>

<label for="age">Age:</label>

<input type="number" id="age" name="age">

<label for="gender">Gender:</label>

<input type="radio" id="male" name="gender" value="male">

<label for="male">Male</label>

```
    <input type="radio" id="female" name="gender" value="female">
    <label for="female">Female</label><br><br>
    <label for="interests">Interests:</label>
    <input type="checkbox" id="coding" name="interests" value="coding">
    <label for="coding">Coding</label>
    <input type="checkbox" id="music" name="interests" value="music">
    <label for="music">Music</label><br><br>
    <input type="submit" value="Submit">
</form>
```

Step 4: Complete Example

Here's how your complete HTML document might look:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Form Page</title>
</head>
<body>
    <h1>Contact Form</h1>
    <form action="/submit" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>
        <label for="age">Age:</label>
```

```
        <input type="number" id="age" name="age"><br><br>
        <label for="gender">Gender:</label>
        <input type="radio" id="male" name="gender" value="male">
        <label for="male">Male</label>
        <input type="radio" id="female" name="gender" value="female">
        <label for="female">Female</label><br><br>
        <label for="interests">Interests:</label>
        <input type="checkbox" id="coding" name="interests" value="coding">
        <label for="coding">Coding</label>
        <input type="checkbox" id="music" name="interests" value="music">
        <label for="music">Music</label><br><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Step 5: View Your Form

Save your HTML file and open it in a web browser to see your form displayed. When you fill it out and click the "Submit" button, the data will be sent to the specified action URL**.**

Step for heading tags

Using heading tags in HTML is essential for structuring your web page content. They help organize information hierarchically and improve accessibility and SEO. Here's a step-by-step guide to using heading tags effectively:

Step 1: Set Up Your HTML Document

Start by creating a basic HTML structure. You can use any text editor to write your HTML code.

html

```
<!DOCTYPE html>
<html lang="en">
```

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My Heading Page</title>

</head>

<body>

</body>

</html>

Step 2: Understand Heading Tags

HTML provides six levels of heading tags, from `<h1>` to `<h6>`. The `<h1>` tag is the most important, while `<h6>` is the least important. Use them to create a hierarchy in your content.

- `<h1>`: Main title of the page (usually one per page)

- `<h2>`: Subheading under `<h1>`

- `<h3>`: Subheading under `<h2>`

- `<h4>`: Subheading under `<h3>`

- `<h5>`: Subheading under `<h4>`

- `<h6>`: Subheading under `<h5>`

Step 3: Add Heading Tags

Inside the `<body>` tag, you can add your heading tags to structure your content. Here's an example:

html

<body>

<h1>Welcome to My Web Page</h1>

<h2>About Me</h2>

This section contains information about me.

<h2>My Interests</h2>

<h3>Technology</h3>

```
    <p>I love exploring new technologies.</p>
<h3>Music</h3>
    <p>Music is a huge part of my life.</p>
<h2>Contact Information</h2>
    <h3>Email</h3>
    <p>You can reach me at example@example.com.</p>
</body>
```

Step 4: Complete Example

Here's how your complete HTML document might look with heading tags:

html

```
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Heading Page</title>
</head>
<body>
    <h1>Welcome to My Web Page</h1>
<h2>About Me</h2>
    <p>This section contains information about me.</p>
    <h2>My Interests</h2>
    <h3>Technology</h3>
    <p>I love exploring new technologies.</p>
    <h3>Music</h3>
    <p>Music is a huge part of my life.</p>
    <h2>Contact Information</h2>
```

```
    <h3>Email</h3>
    <p>You can reach me at example@example.com.</p>
</body>
</html>
```

Step 5: View Your Page

Save your HTML file and open it in a web browser to see how the headings are displayed. You should see a clear hierarchy of headings that makes your content easy to read and navigate.

Steps for List tags

Creating lists in HTML is a great way to organize information clearly and concisely. There are three main types of lists in HTML: ordered lists, unordered lists, and description lists. Here's a step-by-step guide to using list tags effectively in a web page:

Step 1: Set Up Your HTML Document

Start with a basic HTML structure. You can use any text editor to create your HTML file.

```
html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My List Page</title>
</head>
<body>
</body>
</html>
```

Step 2: Add an Unordered List

An unordered list is used when the order of items does not matter. Use the `<ul>`

tag for the list and `<li>` tags for each list item.

html

```
<h2>My Favorite Fruits</h2>
<ul>
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

Step 3: Add an Ordered List

An ordered list is used when the order of items is important. Use the `<ol>` tag for the list and `<li>` tags for each list item.

html

```
<h2>Steps to Make a Smoothie</h2>
<ol>
  <li>Gather ingredients</li>
  <li>Blend until smooth</li>
  <li>Pour into a glass</li>
</ol>
```

Step 4: Add a Description List

A description list is used for terms and their descriptions. Use the `<dl>` tag for the list, `<dt>` for the term, and `<dd>` for the description.

html

```
<h2>HTML List Tags</h2>
<dl>
  <dt>Unordered List</dt>
  <dd>A list where the order of items does not matter.</dd>
  <dt>Ordered List</dt>
  <dd>A list where the order of items is important.</dd>
```

```
    <dt>Description List</dt>
    <dd>A list of terms and their descriptions.</dd>
</dl>
```

Step 5: Complete Example

Here's how your complete HTML document might look with all three types of lists:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My List Page</title>
</head>
<body>
    <h1>Welcome to My List Page</h1>
    <h2>My Favorite Fruits</h2>
    <ul>
        <li>Apple</li>
        <li>Banana</li>
        <li>Cherry</li>
    </ul>
    <h2>Steps to Make a Smoothie</h2>
    <ol>
        <li>Gather ingredients</li>
        <li>Blend until smooth</li>
        <li>Pour into a glass</li>
    </ol>
```

```
    <h2>HTML List Tags</h2>
    <dl>
      <dt>Unordered List</dt>
      <dd>A list where the order of items does not matter.</dd>
      <dt>Ordered List</dt>
      <dd>A list where the order of items is important.</dd>
      <dt>Description List</dt>
      <dd>A list of terms and their descriptions.</dd>
    </dl>
</body>
</html>
```

Step 6: View Your Lists

Save your HTML file and open it in a web browser to see how the lists are displayed. You should see clearly organized content that is easy to read.

Steps for Media tags

Incorporating media elements such as images, audio, and video into your web page is a great way to enhance user experience. Here's a step-by-step guide on how to use media tags in HTML:

Step 1: Set Up Your HTML Document

Start with a basic HTML structure. You can use any text editor to create your HTML file.

```
html
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Media Page</title>
```

</head>

<body>

</body>

</html>

Step 2: Adding Images

To add an image, use the `<img>` tag. You need to specify the `src` attribute for the image source and the `alt` attribute for alternative text.

html

<h2>My Favorite Landscape</h2>

<img src="landscape.jpg" alt="A beautiful landscape" width="600">

Step 3: Adding Audio

To add audio, use the `<audio>` tag. You can include controls for play, pause, and volume. The `src` attribute specifies the audio file.

html

<h2>Listen to My Favorite Song</h2>

<audio controls>

<source src="favorite-song.mp3" type="audio/mpeg">

Your browser does not support the audio tag.

</audio>

Step 4: Adding Video

To add a video, use the `<video>` tag. Similar to the audio tag, you can include controls. The `src` attribute specifies the video file.

html

<h2>Watch My Favorite Video</h2>

<video width="600" controls>

<source src="favorite-video.mp4" type="video/mp4">

Your browser does not support the video tag.

</video>

Step 5: Complete Example

Here's how your complete HTML document might look with images, audio, and video:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My Media Page</title>
</head>
<body>
  <h1>Welcome to My Media Page</h1>
  <h2>My Favorite Landscape</h2>
  <img src="landscape.jpg" alt="A beautiful landscape" width="600">
  <h2>Listen to My Favorite Song</h2>
  <audio controls>
    <source src="favorite-song.mp3" type="audio/mpeg">
    Your browser does not support the audio tag.
  </audio>
  <h2>Watch My Favorite Video</h2>
  <video width="600" controls>
    <source src="favorite-video.mp4" type="video/mp4">
    Your browser does not support the video tag.
  </video>
</body>
</html>
```

Step 6: View Your Media

Save your HTML file and open it in a web browser to see how the media elements are displayed. You should be able to view the image, play the audio, and watch the video directly on the page.

Steps for Code tags

Using code tags in HTML is essential for displaying code snippets or technical content on your web page. The `<code>` tag is specifically designed to represent a fragment of computer code. To enhance the display of code, you can also use the `<pre>` tag to maintain formatting and whitespace. Here's a step-by-step guide on how to use code tags effectively in a web page:

Step 1: Set Up Your HTML Document

Start with a basic HTML structure. You can use any text editor to create your HTML file.

```
html

<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>My Code Page</title>

</head>

<body>

</body>

</html>
```

Step 2: Using the `<code>` Tag

To display a single line of code or a short code snippet, use the `<code>` tag.

```
html

<h2>Displaying a Code Snippet</h2>

<p>To create a variable in JavaScript, you can use the following code:</p>
```

<code>let myVariable = 10;</code>

Step 3: Using the `<pre>` Tag

If you want to display a block of code with preserved formatting (like indentation and line breaks), use the `<pre>` tag. You can also combine it with the `<code>` tag.

html

```
<h2>Displaying a Block of Code</h2>
<pre><code>
function greet(name) {
   console.log("Hello, " + name + "!");
}
greet("World");
</code></pre>
```

Step 4: Complete Example

Here's how your complete HTML document might look with code tags:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>My Code Page</title>
</head>
<body>
   <h1>Welcome to My Code Page</h1>
   <h2>Displaying a Code Snippet</h2>
   <p>To create a variable in JavaScript, you can use the following code:</p>
   <code>let myVariable = 10;</code>
```

```
    <h2>Displaying a Block of Code</h2>
    <pre><code>
function greet(name) {
    console.log("Hello, " + name + "!");
}
greet("World");
</code></pre>
</body>
</html>
```

Step 5: View Your Code

Save your HTML file and open it in a web browser to see how the code snippets are displayed. You should see the inline code and the block of code formatted clearly.

Steps for HTML frame tags

Using HTML frame tags allows you to create a web page that can display multiple HTML documents within a single browser window. However, it's important to note that frames are considered outdated in modern web design, and the use of `<iframe>` is more common. Nevertheless, if you're interested in learning how to use frames, here's a step-by-step guide:

Step 1: Set Up Your HTML Document

Create a new HTML file and start with a basic structure. Use the `<frameset>` tag instead of the `<body>` tag.

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Frame Page</title>
</head>
```

<frameset cols="25%, 75%">

<!-- Frame definitions will go here -->

</frameset>

</html>

Step 2: Define the Frameset

The `<frameset>` tag defines how to divide the browser window into frames. You can specify either rows or columns.

- Columns: Use the `cols` attribute to define the width of each frame.

- Rows: Use the `rows` attribute to define the height of each frame.

For example, to create a layout with two columns (one taking up 25% of the width and the other 75%):

html

<frameset cols="25%, 75%">

<frame src="left.html" name="leftFrame">

<frame src="right.html" name="rightFrame">

</frameset>

Step 3: Specify the Frame Source

Each `<frame>` tag specifies the content to be displayed in that frame using the `src` attribute.

html

<frameset cols="25%, 75%">

<frame src="left.html" name="leftFrame">

<frame src="right.html" name="rightFrame">

</frameset>

Step 4: Complete Example

Here's how your complete HTML document might look using frames:

html

<!DOCTYPE html>

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>My Frame Page</title>
</head>
<frameset cols="25%, 75%">
   <frame src="left.html" name="leftFrame">
   <frame src="right.html" name="rightFrame">
</frameset>
</html>
```

Step 5: Create Frame Content

You will need to create the `left.html` and `right.html` files that will be displayed in the respective frames. Here's a simple example for each.

left.html:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Left Frame</title>
</head>
<body>
   <h1>Welcome to the Left Frame</h1>
   <p>This is the content of the left frame.</p>
</body>
</html>
```

right.html:

```
html

<!DOCTYPE html>

<html lang="en">

<head>

   <title>Right Frame</title>

</head>

<body>

   <h1>Welcome to the Right Frame</h1>

   <p>This is the content of the right frame.</p>

</body>

</html>
```

Step 6: View Your Frames

Save your HTML files and open the main HTML file in a web browser. You should see two frames: the left frame displaying content from `left.html` and the right frame displaying content from `right.html`.

Steps for Html comment

HTML comments are a useful feature that allows you to leave notes or explanations in your code without affecting how the HTML is rendered in the browser. Comments can help you document your code, making it easier for you or others to understand it later. Here's a step-by-step guide on how to use HTML comments in your web page development:

Step 1: Set Up Your HTML Document

Start by creating a basic HTML structure. You can use any text editor to create your HTML file.

```
html

<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">
```

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My Comment Example</title>

</head>

<body>

</body>

</html>

Step 2: Add HTML Comments

To add a comment in your HTML code, use the following syntax:

html

<!-- This is a comment -->

Comments can be placed anywhere in your HTML document, and they will not be displayed in the browser. Here are a few examples:

Example 1: Commenting in the Head Section

You can add comments in the `<head>` section to explain the purpose of certain elements.

html

<head>

<meta charset="UTF-8">

<title>My Comment Example</title>

<!-- This title is displayed in the browser tab -->

</head>

Example 2: Commenting in the Body Section

You can also add comments in the `<body>` section to clarify different parts of your content.

html

<body>

<h1>Welcome to My Web Page</h1>

<!-- This is the main heading of the page -->

```
    <p>This is a paragraph of text.</p>
    <!-- This paragraph provides information about the topic -->
<div>
        <h2>Subheading</h2>
        <!-- This section contains additional information -->
    </div>
</body>
```

Step 3: Complete Example

Here's how your complete HTML document might look with comments included:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Comment Example</title>
    <!-- This title is displayed in the browser tab -->
</head>
<body>
    <h1>Welcome to My Web Page</h1>
    <!-- This is the main heading of the page -->
    <p>This is a paragraph of text.</p>
    <!-- This paragraph provides information about the topic -->
    <div>
        <h2>Subheading</h2>
        <!-- This section contains additional information -->
    </div>
```

</body>

</html>

Step 4: View Your Comments

Save your HTML file and open it in a web browser. You will not see the comments displayed on the page, but they will be present in the source code. You can view the source code by right-clicking on the page and selecting "View Page Source" or "Inspect."

Steps for Grouping tags (div, span)

Grouping tags like `<div>` and `<span>` are fundamental in web development for organizing content and applying styles. Here's a guide on how to effectively use these tags in your web page development:

1. Understand the Purpose:

- `<div>`: A block-level element used to group larger sections of content. It typically contains other block-level elements (like paragraphs, headings, etc.) and can be styled using CSS.

- `<span>`: An inline element used to group small portions of text or other inline elements. It's useful for applying styles to a part of the text without breaking the flow.

2. Set Up Your HTML Document: Start with a basic HTML structure.

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Web Page Title</title>
    <link rel="stylesheet" href="styles.css"> <!-- Link to your CSS file -->
</head>
<body>
    <!-- Content will go here -->
```

```
</body>
</html>
```

3. Using `<div>` for Block-Level Grouping:

- Use `<div>` to create sections of your webpage. This can include headers, footers, main content areas, and sidebars.

html

```
<body>
    <div class="header">
        <h1>Website Title</h1>
    </div>
    <div class="main-content">
        <p>This is the main content area.</p>
    </div>
    <div class="footer">
        <p>&copy; 2023 Your Website. All rights reserved.</p>
    </div>
</body>
```

4. Using `<span>` for Inline Grouping:

- Use `<span>` for styling or manipulating small pieces of text within a paragraph or other inline elements.

html

```
<div class="main-content">
    <p>This is a <span class="highlight">highlighted</span> word in the main content area.</p>
</div>
```

5. Apply CSS Styles: Use CSS to style your `<div>` and `<span>` elements. For example:

css

```
.header {
    background-color: f8f9fa;
    padding: 10px;
    text-align: center;
}
.main-content {
    margin: 20px;
    font-family: Arial, sans-serif;
}
.footer {
    background-color: f1f1f1;
    text-align: center;
    padding: 10px;
    position: relative;
    bottom: 0;
    width: 100%;
}
.highlight {
    background-color: yellow;
    font-weight: bold;
}
```

6. Test Your Layout: Open your HTML file in a web browser to see how the `<div>` and `<span>` elements are displayed. Check for proper alignment, spacing, and styling.

7. Ensure Responsiveness: Use CSS media queries to ensure your layout is responsive and looks good on different screen sizes.

css

```
@media (max-width: 600px) {
```

```
    .main-content {
        margin: 10px;
    }
}
```

8. Validate Your HTML and CSS: Use online validators to check for errors in your HTML and CSS to ensure they comply with web standards.

Steps for Hyperlink tag

Creating hyperlinks is a fundamental aspect of web development. Hyperlinks allow users to navigate between different pages or sections of a website, or even to external sites. Here are the steps to effectively use hyperlink tags in your web page development:

Steps for Using the Hyperlink Tag (`<a>`)

1. Understand the Hyperlink Tag:

- The `<a>` tag (anchor tag) is used to create hyperlinks. The `href` attribute specifies the URL of the page the link goes to.

2. Set Up Your HTML Document: Start with a basic HTML structure.

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Web Page Title</title>
</head>
<body>
    <!-- Hyperlinks will go here -->
</body>
</html>
```

3. Creating a Basic Hyperlink:

- Use the `<a>` tag to create a hyperlink. Here's a simple example:

html

<body>

<h1>Welcome to My Website</h1>

Visit my favorite website for more information.

</body>

4. Linking to Different Types of URLs:

- You can link to external websites, internal pages, or specific sections within a page

- External Link:

html

Visit Example

- Internal Link: Linking to another page within your website.

html

About Us

- Anchor Link: Linking to a specific section within the same page.

html

Contact Us

<!-- Somewhere else on the same page -->

<h2 id="contact">Contact Us</h2>

5. Opening Links in a New Tab:

- To open a link in a new tab, use the `target="_blank"` attribute.

html

Visit Example in a new tab

6. Adding Link Titles:

- You can provide additional information about the link using the `title` attribute.

This text appears as a tooltip when the user hovers over the link.

html

Visit Example

7. Styling Hyperlinks with CSS:

- Use CSS to style your hyperlinks. You can change colors, text decoration, and more.

css

```
a {
    color: blue; /* Normal link color */
    text-decoration: none; /* Remove underline */
}
a:hover {
    color: red; /* Color when hovered */
    text-decoration: underline; /* Underline on hover */
}
```

8. Testing Your Links:

- After adding hyperlinks, open your HTML file in a web browser to test that all links work correctly and navigate to the intended pages.

9. Validate Your HTML:

- Use an HTML validator to ensure your code is error-free and adheres to web standards.

Example of a Complete HTML Document with Hyperlinks

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Web Page Title</title>
    <style>
      a {
        color: blue;
        text-decoration: none;
      }
      a:hover {
        color: red;
        text-decoration: underline;
      }
    </style>
</head>
<body>
    <h1>Welcome to My Website</h1>
    <p>Visit my <a href="https://www.example.com" target="_blank" title="Go to Example Website">favorite website</a> for more information.</p>
    <p>Learn more about us on our <a href="about.html">About Us</a> page.</p>
    <p>For inquiries, <a href="contact">Contact Us</a>.</p>
    <h2 id="contact">Contact Us</h2>
    <p>Email: contact@example.com</p>
</body>
</html>
```

Steps for semantic tags

1. Understand Semantic HTML: Learn the purpose of semantic HTML, which helps to clearly define the structure and meaning of your content.

2. Identify Content Types: Analyze the content of your webpage and determine the different types of content you have, such as headers, navigation, articles, sections,

and footers.

3. Choose Appropriate Semantic Tags: Select the right semantic tags based on your content types. Common semantic tags include:

- `<header>`: For introductory content or navigational links.
- `<nav>`: For navigation links.
- `<main>`: For the main content of the document.
- `<article>`: For independent, self-contained content.
- `<section>`: For thematic grouping of content.
- `<aside>`: For content related to the main content, like sidebars.
- `<footer>`: For footer content at the bottom of the page or section.

4. Create the Basic HTML Structure: Start with a basic HTML template:

```
html
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Page Title</title>
</head>
<body>
    <!-- Add semantic tags here -->
</body>
</html>
```

5. Implement Semantic Tags: Replace generic `<div>` and `<span>` tags with the appropriate semantic tags based on the content structure. For example:

```
html
<header>
    <h1>Website Title</h1>
```

```
    <nav>
      <ul>
        <li><a href="home">Home</a></li>
        <li><a href="about">About</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <article>
      <h2>Article Title</h2>
      <p>This is the main content of the article.</p>
    </article>
    <aside>
      <h3>Related Information</h3>
      <p>This is some additional information related to the article.</p>
    </aside>
  </main>
  <footer>
    <p>&copy; 2023 Your Website</p>
  </footer>
```

6. Validate Your HTML: Use HTML validators to check for any syntax errors and ensure that your semantic structure is correctly implemented.

7. Test for Accessibility: Use accessibility tools to ensure your webpage is usable for individuals with disabilities. Semantic HTML enhances accessibility by providing clear structure.

8. Optimize for SEO: Ensure that your semantic tags are used to enhance search engine optimization. Properly structured content helps search engines understand the context of your pages.

9. Review and Iterate: After implementing semantic tags, review your webpage for

any improvements and iterate on your design and structure as needed.

Points to Remember

- HTML, or Hypertext Mark-up Language, is the standard mark-up language used to create and design the structure of web pages. An HTML tag is a special code used in HTML to define the structure and content of a webpage. HTML tags are enclosed in angle brackets (< >) and usually come in pairs (an opening tag and a closing tag). And the can be used to create a website
- The main attribute is class, id, style, href etc.
- Html categories are (HTML Structural tags, Formatting tags, Table tags, Form tags ,Heading tags ,List tags, Media tags ,Code tags ,HTML frame tags ,HTML Comment, Grouping tags (div, span) Hyperlink tag, Semantic tags).
  Step 1: Set Up Your HTML Document

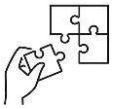
  Step 2: Use Structural Tags

  Step 3: Add Additional Structural Tags

  Step 4: Final HTML Structure

  Step 5: Style Your Page

  Step 6: Test Your Web Page



Application of learning 2.2.

You are part of a student organization at your schools and have been tasked with creating a webpage for an upcoming student fair. The page will showcase event details, a registration form for student groups, a schedule of activities, a gallery of previous events, and links to social media pages.

Indicative content 2.3: Application of CSS

Duration: 10 hours

Theoretical Activity 2.3.1: **Description of CSS**

Tasks:

1:

1: Answer the following questions:

i. What do you understand about CSS?
ii. What are version of CSS.
iii. What are Types of CSS Styles

IV.What are the Use of CSS Visual rules.

V.What are the Use of CSS Display and positioning?

VI.What are the Use of CSS Box and Grid model?

2: Address any questions or concerns.

3: Present your findings to the whole class.

4: Ask trainee to read the key readings on activity 2.3.1

Key readings 2.3.1: Description of CSS

1.Description of CSS

CSS (Cascading Style Sheets) is a style sheet language used to describe the presentation and visual appearance of HTML elements on a webpage. While HTML is used to structure content (e.g., text, images, and links), CSS controls the layout, design, and style of those elements, such as colors, fonts, margins, and positioning.

1.2 Versions OF CSS

As of my last knowledge update in January 2022, here are the major versions of CSS that have been released:

1. CSS1 (Cascading Style Sheets Level 1):
   - The first version of CSS, published in 1996.
   - Introduced basic styling properties and selectors.

2. CSS2 (Cascading Style Sheets Level 2):
   - Published in 1998 and later revised in 2011 as CSS 2.1.
   - Expanded the capabilities of CSS1 with new properties, positioning, and media types.
   - Introduced features like absolute, relative, and fixed positioning.
3. CSS2.1:
   - A revision of CSS2, aiming to clarify and correct errors in the specification.
   - It became a stable recommendation by the W3C in 2011.
4. CSS3 (Cascading Style Sheets Level 3):
   - CSS3 is not a single monolithic specification but a collection of separate modules, each covering specific features.
   - Introduced new properties, selectors, and capabilities, allowing for more advanced styling and design.
   - Modules include Selectors, Color, Backgrounds and Borders, Text Effects, Transforms, Transitions, Animations, and more.

Notable CSS3 modules include:

- CSS Selectors Level 3
- CSS Color Module Level 3
- CSS Backgrounds and Borders Module Level 3
- CSS Transitions
- CSS Animations

CSS3 is an ongoing effort, and new modules may be added over time.

5. CSS4:
   - As of my last update in January 2022, there is no formal CSS4 specification. The CSS Working Group has moved towards the concept of "Level" rather than version numbers. New features are added as separate modules and updates to existing modules.

Some of the ongoing and proposed features include:

- Flexbox Level 2
- Grid Layout Level 2
- Scroll Snap
- CSS Variables (Custom Properties)

It's important to note that the development and adoption of CSS features depend on browser support. Browsers may implement parts of CSS3 independently, and

feature support can vary between browsers.

For the latest information on CSS specifications and modules, you may want to check the official W3C (World Wide Web Consortium) website or other reliable sources for web standards.

1.3 Different Types of CSS Styles

CSS styles can be categorized into three main types:

Inline CSS: The style rules are applied directly within the HTML element using the style attribute.

Example: <h1 style="color: blue;">This is a heading</h1>

Advantage: Quick, specific to a single element.

Limitation: Not reusable; mixes content with styling.

Internal (Embedded) CSS: Styles are placed inside the <style> tag within the <head> section of an HTML document.

Example:

html

Copy code

<style>

h1 { color: blue; }

</style>

Advantage: Styles apply to that specific HTML page.

Limitation: Affects only one page, making it harder to maintain for larger websites.

External CSS: Styles are written in a separate .css file and linked to the HTML document.

Example:

html

Copy code

<link rel="stylesheet" href="styles.css">

Advantage: Can apply consistent styles across multiple HTML pages and makes the HTML file cleaner.

Limitation: Requires loading an additional file, which may affect page load time.

3.Compare and Contrast the Various CSS Positioning Methods

CSS positioning defines how elements are placed in relation to other elements or their parent containers. The key positioning methods include:

Static Positioning (default):

Elements are positioned in the normal document flow.

Example: Text paragraphs in a document.

Limitation: Cannot be adjusted with top, left, right, or bottom properties.

Relative Positioning:

The element is positioned relative to its normal position in the document flow.

Example: position: relative; top: 10px; shifts the element 10px down from its original position.

Advantage: Can be moved without affecting the layout of surrounding elements.

Absolute Positioning:

The element is positioned relative to the nearest positioned ancestor or the initial containing block (if no ancestor is positioned).

Example: position: absolute; top: 20px; moves the element 20px from the top of its parent.

Limitation: Removed from the document flow, so other elements will not adjust around it.

Fixed Positioning:

The element is positioned relative to the viewport and stays in place when the page is scrolled.

Example: A sticky navigation bar.

Limitation: Also removed from the document flow and can overlap other elements.

Sticky Positioning:

The element toggles between relative and fixed positioning depending on the scroll position.

Example: position: sticky; top: 0; keeps the element in place after the user scrolls past it.

Advantage: Useful for making elements like headers "stick" as you scroll.

Comparison:

Static is the default flow with no special positioning.

Relative modifies an element's position without affecting others.

Absolute and fixed remove elements from the flow, making them independent but potentially overlapping other content.

Sticky combines the benefits of both relative and fixed, depending on the scroll position.

1.4 Key Concepts of CSS in HTML

Separation of Content and Style:

HTML focuses on the structure and content of the webpage, while CSS defines how that content should be styled and displayed. This separation allows for cleaner code and easier maintenance.

Selectors:

CSS uses **selectors** to target HTML elements that you want to style. These selectors can be based on element types, classes, IDs, or attributes.

Cascading:

CSS follows the **cascading** principle, meaning that the style rules are applied based on a hierarchy of importance. Styles are applied in order, and if there are conflicts, the most specific rule wins.

Reusability:

With CSS, you can define styles once and apply them to multiple elements throughout the webpage, ensuring consistency and reducing the need to repeat the same styling code.

CSS Syntax

A CSS rule is made up of:

Selector: Specifies which HTML element(s) to style.

Declaration Block: Contains one or more declarations separated by semicolons, each consisting of a property and its corresponding value.

```css
p {
    color: blue;
    font-size: 16px;
}
```

In this example:

p is the selector that targets all <p> (paragraph) elements.

Inside the curly braces {}, there are two declarations: one setting the color property to blue, and another setting the font-size to 16 pixels.

1.4.1 Ways to Include CSS in HTML

There are three primary ways to add CSS to an HTML document:

1. Inline CSS:

CSS styles can be applied directly to an HTML element using the style attribute. This is useful for quick styling but not recommended for large-scale projects due to poor maintainability.

Example:

```html
<p style="color: blue; font-size: 16px;">This is a blue paragraph.</p>
```

2. Internal CSS (Embedded Styles):

CSS can be included within the <style> tag inside the <head> section of the HTML document. This applies styles only to the current page.

Example:

```html
<head>
    <style>
        p {
            color: blue;
            font-size: 16px;
        }
    </style>
</head>
```

3. External CSS:

CSS is written in a separate .css file and linked to the HTML document using the <link> tag. This method is the best practice for larger websites, allowing you to style multiple pages from a single stylesheet.

Example (HTML):

```html
<head>
    <link rel="stylesheet" href="styles.css">
</head>
```

Example (styles.css):

```css
p {
    color: blue;
    font-size: 16px;
}
```

1CSS Selectors

CSS selectors define which HTML elements the styles will apply to. Some common selectors include:

Element Selector: Targets all elements of a specified type.

Example: p { color: blue; } targets all <p> elements.

Class Selector: Targets elements with a specific class. Classes are reusable across multiple elements.

HTML:

CSS: .box { border: 1px solid black; }

ID Selector: Targets an element with a unique ID. IDs should be used for elements that appear only once on a page.

- ✓ Only once on a page.

  HTML:

  CSS: #main-header { background-color: gray; }

- ✓ Universal Selector: Targets all elements.

  Example: * { margin: 0; padding: 0; }

- ✓ Attribute Selector: Targets elements with a specific attribute or attribute value.

Example: input[type="text"] { background-color: yellow; }

1.5.CSS Visual rules.

CSS visual rules define the way HTML elements are displayed and styled on a webpage. These rules control layout, positioning, spacing, and overall visual presentation. By applying various visual properties in CSS, developers can create more organized, visually appealing, and user-friendly web designs.

1.5.1 CSS visual rules for fonts

are a set of properties used to control the appearance and style of text on a webpage. These rules allow developers to specify font types, sizes, weights, and other visual aspects of typography, helping to create a consistent and aesthetically pleasing design.

Font Properties in CSS:

1. font-family

The font-family property defines the typeface that will be used for the text. You can

specify multiple fonts as a **font stack**, where the browser will use the first available font.

Syntax:

```css
font-family: "Arial", "Helvetica", sans-serif;
```

Example:

```css
body {
    font-family: "Times New Roman", Georgia, serif;
}
```

2. font-size

The font-size property controls the size of the text. The value can be specified in several units, including px (pixels), em, %, and rem.

Units:

px: A fixed size in pixels (e.g., 16px).

em: A relative unit based on the parent element's font size (e.g., 1em is equal to the current font size).

rem: Similar to em, but relative to the root element (usually <html>).

%: A percentage of the parent element's font size.

Syntax:

```css
font-size: 18px;
```

Example:

```css
p {
    font-size: 1.5em;  /* 1.5 times the size of the parent
}
```

3. font-weight

The font-weight property controls the **boldness** or thickness of the font. You can use keywords or numerical values to define the weight of the text.

Values:

Normal: The default weight (equivalent to 400).

Bold: Bold text (equivalent to 700).

Lighter: Lighter text compared to the parent.

Bolder: Bolder text compared to the parent.

Numeric values: You can use values between 100 (thin) to 900 (extra-bold).

Syntax:

```css
font-weight: bold;
```

Example:

```css
h1 {
    font-weight: 700;
}

p {
    font-weight: 300;
}
```

4. font-style

The font-style property controls the style of the font, typically used for **italicizing** text.

Values:

Normal: Regular text.

italic: Italicized text.

oblique: Similar to italic but with a slight slant.

Syntax:

Syntax:

```css
font-style: italic;
```

Example:

```css
em {
    font-style: italic;
}

p.quote {
    font-style: oblique;
}
```

Text-align

The text-align property controls the **horizontal alignment** of text within its container.

Values:

- **left**: Aligns text to the left.
- **right**: Aligns text to the right.
- **center**: Centers the text.
- **justify**: Stretches text so that each line has equal width, often used in newspaper-like layouts.

Syntax:

Syntax:

```css
text-align: center;
```

Example:

```css
h1 {
    text-align: center;
}

p {
    text-align: justify;
}
```

2.Colors and background colors

CSS visual rules for colors and background colors help define the color scheme and overall visual aesthetics of text and elements on a webpage. These rules control the appearance of fonts (text color) and the background behind elements, allowing developers to create visually engaging designs.

CSS Properties for Colors and Background Colors:

1. **Color**

The color property in CSS defines the **text color**. You can set the text color using:

Named colors (e.g., red, blue, green)

Hexadecimal codes (e.g., #ff0000)

RGB (e.g., rgb(255, 0, 0))

RGBA (for colors with transparency) (e.g., rgba(255, 0, 0, 0.8))

HSL (Hue, Saturation, Lightness) (e.g., hsl(0, 100%, 50%))

HSLA (HSL with transparency) (e.g., hsla(0, 100%, 50%, 0.8))

Syntax:

```css
color: blue;
```

Example:

```css
h1 {
    color: #ff6600;  /* Orange color */
}

p {
    color: rgb(34, 139, 34);  /* ForestGreen color */
}
```

Tip: You can combine color with font rules like font-size, font-family, and font-weight to define the overall typography and color scheme.

2. background-color

The background-color property defines the **background color** of an element. Similar to the color property, it can use named colors, hex codes, RGB, RGBA, HSL, or HSLA.

Syntax:

```css
background-color: yellow;
```

Example:

```css
div {
    background-color: #333333;  /* Dark gray background */
    color: white;               /* Text color is white */
}
```

You can apply background-color to any block or inline element, such as a , <section>, <p>, or .

3. Opacity

The opacity property controls the **transparency** of an element and its background. The value ranges from 0 (completely transparent) to 1 (fully opaque).

Syntax:

Syntax:

```css
opacity: 0.8;
```

Example:

```css
.box {
    background-color: rgba(0, 0, 255, 0.5);
    color: white;
}
```

Note: When using opacity, it affects both the background color and the text inside the element. If you only want a transparent background without affecting the text, use rgba or hsla for the background color.

4. background-image

The background-image property allows you to set an **image** as the background of an element. It can be used in conjunction with background-color as a fallback in case the image is unavailable.

Syntax:

```css
background-image: url('image.jpg');
```

Example:

```css
body {
    background-color: #f0f0f0;
    background-image: url('pattern.png');
}
```

You can combine background-color and background-image for layering effects, but keep in mind that the background image will cover the color unless the image has transparency.

2.CSS Display and Positioning

CSS Display and Positioning are fundamental concepts that control how elements are arranged on a webpage. They define how elements are displayed in relation to one another and how they are positioned on the page.

1. CSS Display Property

The display property specifies how an element is displayed in the document flow. It determines whether an element behaves as a block-level or inline element and controls the overall visibility and formatting model of the element.

Common **display** values:

block: Displays the element as a block-level element, which takes up the full width available and begins on a new line.

```css
display: block;
```

Example: , <p>, <h1> are block-level elements by default.

Example:

```css
div {
    display: block;
    width: 100%;
}
```

inline: Displays the element inline, meaning it takes up only as much width as necessary and does not start a new line.

```css
display: inline;
```

Example: , <a>, are inline elements by default.

Example:

```css
span {
    display: inline;
}
```

Inline-block: Acts like an inline element but allows setting width, height, and padding, similar to block-level elements.

```css
display: inline-block;
```

Example:

```css
.box {
    display: inline-block;
    width: 100px;
    height: 100px;
    padding: 10px;
}
```

2. CSS Positioning

CSS positioning controls how elements are positioned in relation to their parent, siblings, or the document window. The position property allows for absolute, relative, or fixed positioning of elements.

Common position values:

Static: The default position value for all elements. Elements are positioned according to the normal document flow (i.e., no specific positioning). Top, right, bottom, and left properties do not apply.

```css
position: static;
```

Example:

```css
p {
    position: static;
}
```

relative: The element is positioned relative to its normal position in the document flow. You can use the top, right, bottom, and left properties to adjust its position.

Example:

```css
.box {
    position: relative;
    top: 20px;    /* Move down by 20px */
    left: 10px;   /* Move right by 10px */
}
```

absolute: The element is positioned relative to its nearest positioned ancestor (i.e., the closest ancestor with a position value of relative, absolute, or fixed). If there is no such ancestor, it is positioned relative to the initial containing block (usually the <html>).

Example:

```css
.absolute-box {
    position: absolute;
    top: 50px;
    right: 0;
}
```

Top, Right, Bottom, and Left:

- The top, right, bottom, and left properties specify the **offset** for positioned elements (those with relative, absolute, fixed, or sticky values).

  These properties determine how far an element is moved from the reference point (either its normal position or its containing block).

Example:

```css
.element {
    position: absolute;
    top: 20px;
    left: 50px;
}
```

fixed: The element is positioned relative to the browser window, meaning it will not move when the page is scrolled. It remains in the same position even when the user scrolls.

Example:

```css
.navbar {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
}
```

The **CSS Box Model** and the **CSS Grid Model** are two fundamental concepts in web layout design that control how elements are sized, spaced, and arranged on a webpage.

CSS Box Model

The **CSS Box Model** is the fundamental concept that describes how HTML elements are rendered as rectangular boxes. It includes the element's content, padding, border, and margin, all of which affect the total size of the element on a webpage.

Components of the Box Model:

Content:

The actual content of the box, such as text, images, or other media.

The size of the content area is controlled by properties like width and height.

Example:

```css
.box {
    width: 300px;
    height: 200px;
}
```

Padding:

The space between the content and the border.

It adds space inside the element but within the element's border.

Padding is affected by the padding property, which can be applied uniformly or for each side (top, right, bottom, left).

Example:

Example:

```css
.box {
    padding: 20px; /* Adds 20px space inside the box on all sides */
}
```

Border:

The line surrounding the padding (if any) and the content.

You can control its width, style, and color using the border property (or more specific ones like border-width, border-style, border-color).

Example:

```css
.box {
    border: 2px solid black; /* A solid black border of 2px */
}
```

Margin:

- The space outside the border, which separates the element from other elements.
- The margin property controls the outer spacing around the element, and you can specify margins for each side.

Example:

```css
.box {
    margin: 10px; /* Adds 10px space outside the box */
}
```

Practical Activity 2.3.2: Applying CSS

Task: 1

1: Read key reading 2.3. 2

2: Referring to key reading 2.3.2, As an IOT developper, you are asked to go to the computer lab for applying the use of CSS.

3: Present your work to the trainer and whole class

4: Ask clarification where necessary.

Key readings 2.3.1: Applying CSS

Step-by-step guide for applying CSS to your web page, focusing on visual rules,

display and positioning, and the box and grid model.

Step 1: Set Up Your HTML Document

1. Create Your HTML File: Ensure you have an HTML file (e.g., `index.html`).

2. Link Your CSS File: In the `<head>` section of your HTML, link to your CSS file (e.g., `styles.css`).

```
 html

<! DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Your Web Page Title</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<! -- Your content goes here -->

</body>

</html>
```

Step 2: Use CSS Visual Rules

2.1 Font, Colors, and Background Colors

1. Font Styles: Define font properties for text elements.

```
 css
```

```
  body {
font-family: 'Arial', sans-serif; /* Font family */
font-size: 16px; /* Font size */
color: 333; /* Text color */
  }
  h1 {
font-weight: bold; /* Bold font weight */
color: 4CAF50; /* Green color */
  }
```

2. Background Colors: Set background colors for elements.

```
 css
body {
background-color: f0f0f0; /* Light gray background */
}
header {
background-color: 4CAF50; /* Green background for header */
color: white; /* White text color */
  }
```

2.2 Opacity

1. Set Opacity: Adjust the transparency of an element.

```
css
  transparent {
```

```
  opacity: 0.5; /* 50% opacity */
}
```

2.3 Background Image

1. Add a Background Image: Use a background image for the body or a specific element.

```
css
body {
    background-image: url('background.jpg'); /* Path to your image */
    background-size: cover; /* Cover the entire area */
    background-position: center; /* Center the image */
}
```

Step 3: Use CSS Display and Positioning

3.1 Display Types

1. Block Elements: Use `display: block;` for block-level elements.

```
css
div {
    display: block; /* Makes the div a block element */
}
```

2. Inline Elements: Use `display: inline;` for inline elements.

```
css
span {
    display: inline; /* Makes the span an inline element */
```

```
}
```

3.2 Positioning

1. Relative Positioning: Position elements relative to their normal position.

```
css
.relative {
    position: relative; /* Position relative to its original location */
    top: 10px; /* Move down 10 pixels */
}
```

2. Absolute Positioning: Position elements relative to the nearest positioned ancestor.

```
css
.absolute {
    position: absolute; /* Position absolute to the nearest positioned ancestor */
    top: 20px; /* 20 pixels from the top */
    left: 30px; /* 30 pixels from the left */
}
```

3. Fixed Positioning: Position elements relative to the viewport, unaffected by scrolling.

```
css
. fixed {
    position: fixed; /* Fixed position relative to the viewport */
    bottom: 0; /* Stick to the bottom */
    right: 0; /* Stick to the right */
```

```
}
```

4. Sticky Positioning: A mix of relative and fixed positioning.

```
css
.sticky {
    position: sticky; /* Sticky position */
    top: 0; /* Stick to the top of the viewport */
}
```

Step 4: Use CSS Box and Grid Model

4.1 Height and Width

1. Set Height and Width: Define the dimensions of elements.

```
css
.box {
    width: 300px; /* Set width */
    height: 200px; /* Set height */
}
```

4.2 Borders and Border Radius

1. Add Borders: Define borders around elements.

```
css
.border-box {
    border: 2px solid 000; /* Solid black border */
}
```

2. Add Border Radius: Create rounded corners.

```
css

.rounded {

    border-radius: 15px; /* Rounded corners */

}
```

4.3 Padding and Margin

1. Set Padding: Add space inside an element, between the content and the border.

```
css

.padded {

    padding: 20px; /* 20 pixels of padding */

}
```

2.Set Margin: Add space outside an element, between it and other elements.

```
css

.margined {

    margin: 15px; /* 15 pixels of margin */

}
```

4.4 Visibility and Auto

1. Control Visibility: Use the `visibility` property to show or hide elements.

```
css

.hidden {

    visibility: hidden; /* Hide the element but still occupy space */

}
```

2. Auto Margin: Use auto margins for centering elements.

```
`css
.centered {
    margin: 0 auto; /* Center the element horizontally */
    width: 50%; /* Set a width */
}
```

Step 5: Use CSS Grid Model (Optional)

1. Create a Grid Layout: Use CSS Grid for layout.

```
css
.grid-container {
    display: grid; /* Use grid layout */
    grid-template-columns: repeat (3, 1fr); /* 3 equal columns */
   gap: 10px; /* Space between grid items */
}
```

Points to Remember

- CSS (Cascading Style Sheets) is a powerful styling language used to control the presentation and layout of HTML documents. It is applied to enhance the visual appearance, layout, and responsiveness of web pages. Here are various applications of CSS.
- The version of CSS are CSS1 (Cascading Style Sheets Level 1, CSS2 (Cascading Style Sheets Level 2, CSS2.1, CSS3 (Cascading Style Sheets Level 3), CSS4).
- The types of CSS style is (Inline Styles, Internal Styles (Embedded Styles), External Styles, CSS Selectors etc.
- Step-by-step guide for applying CSS to your web page, focusing on visual rules, display and positioning, and the box and grid model.

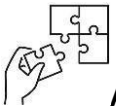
Step 1: Set Up Your HTML Document

Step 2: Use CSS Visual Rules

Step 3: Use CSS Display and Positioning

Step 4: Use CSS Box and Grid Model

Step 5: Use CSS Grid Model (Optional)



Application of learning 2.3.

You are tasked with developing a **smart home IoT dashboard** that allows users to control and monitor their home devices such as lights, thermostats, and security cameras. The goal is to make the interface user-friendly, visually appealing, and responsive across various devices (smartphones, tablets, desktops).

Learning outcome 2 end assessment

Section 1: Read the following statement related to IoT Web application using PHP, and answer by True if the statement is correct or False if the statement is wrong

1. User personas are fictional representations of the ideal users for a system.
2. A consistent design is not necessary for a good user interface.
3. The tag in HTML is used to create inline elements.
4. CSS stands for Cascading Style Sheets and is used to style HTML elements.
5. The CSS position: fixed; property allows an element to stay in the same position relative to the viewport, even when the page is scrolled.

Section 2: Multiple Choice Questions (MCQs)

1. Which of the following is not a feature of a good user interface?

A. Clear and intuitive navigation
B. Consistent design
C. Overloaded information on a single page
D. Use of familiar UI patterns

2. Which HTML tag is used to create an ordered list?

A.
B.
C.
D. <dl>

3. Which of the following CSS properties is used to control the transparency of an element?

A. opacity
B. display
C. position
D. border-radius

4. What is the main purpose of creating wireframes during UI development?

A. To add color and styles to the interface
B. To outline the basic structure and layout of the user interface
C. To write the backend logic for the application
D. To finalize the content for the UI

5. Which CSS property is used to change the font size of an element?

A. font-size
B. font-weight
C. text-align
D. font-style

Section 3: Matching Questions

Match the following HTML tags with their appropriate use:

1. <table>	A) Used to display tabular data.
2. <h1>	C) Used to create a form for user input.
3. <form>	B) Used to define the most important heading.
4. <a>	E) Used to display images.
5. <img>	D) Used to create hyperlinks.
6.<tr>	

Section 4: Open Questions

1. What is the purpose of using wireframes in UI development?

2. Explain the difference between padding and margin in CSS.

Practical assessment

You are developing a **smart agriculture monitoring dashboard** that allows farmers to track various environmental parameters (e.g., soil moisture, temperature, humidity) across their farm using IoT sensors. The goal is to create a user-friendly, visually appealing, and responsive interface that can be accessed on multiple devices like desktops, tablets, and smartphones.

References

Mohammad El-Basioni, Basma M., and Sherine M. Abd El-Kader. "Designing and modeling an IoT-based software system for land suitability assessment use case." *Environmental Monitoring and Assessment* 196.4 (2024): 380.

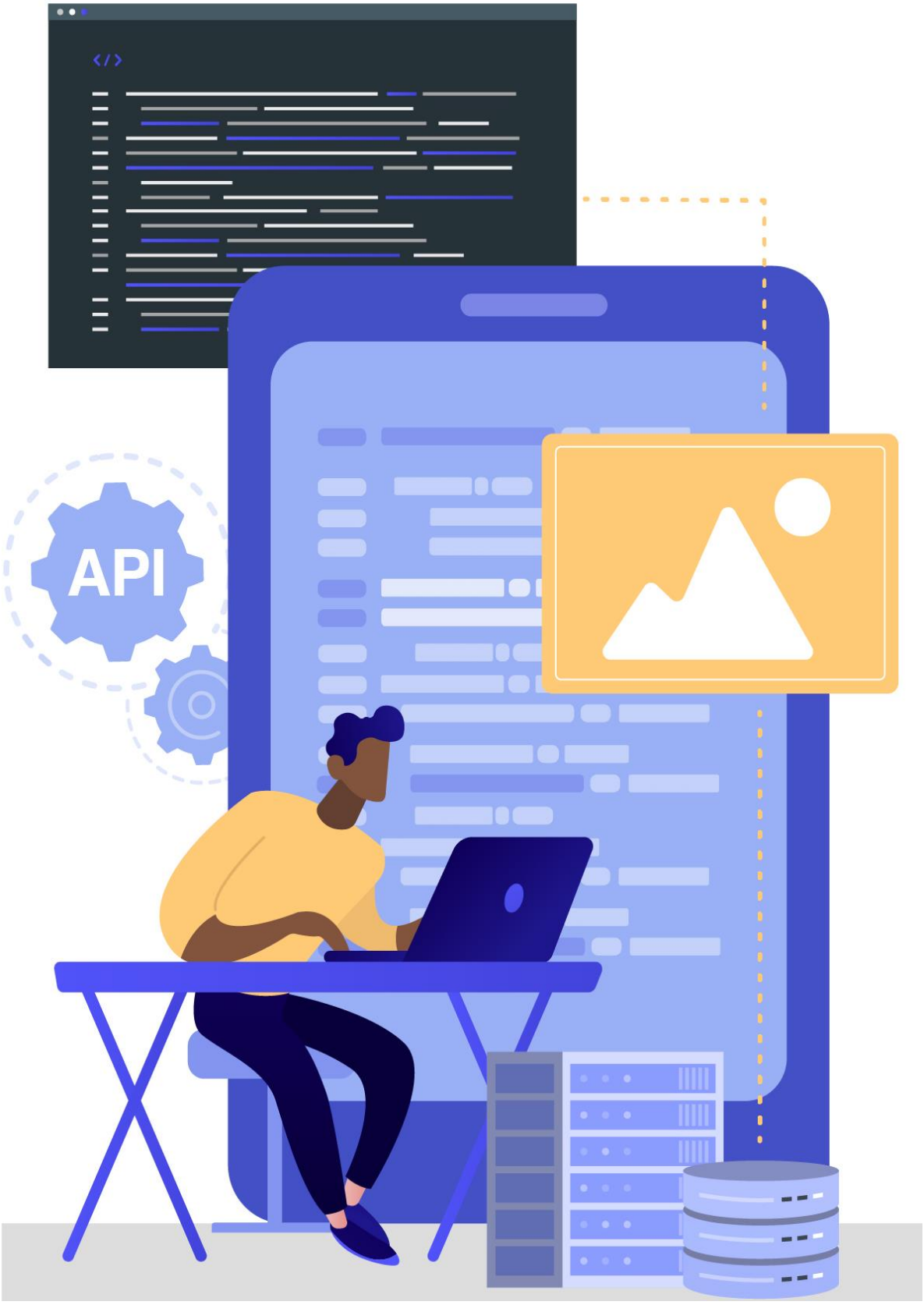
https://www.w3schools.com/html/html_intro.asp

Sinlae, Fried, et al. "Penggunaan Framework Laravel dalam Membangun Aplikasi Website Berbasis PHP." *Jurnal Siber Multi Disiplin* 2.2 (2024): 119-132.

https://www.w3schools.com/css/css_intro.asp

Duy, Le. "Web Application Development." (2024).

Sinlae, Fried, et al. "Penggunaan Framework Laravel dalam Membangun Aplikasi Website Berbasis PHP." *Jurnal Siber Multi Disiplin* 2.2 (2024): 119-132.


API

Indicative Contents
3.1 Interpret API Endpoints
3.2 Use API data on User Interface
3.3 Document IoT Web Application

Key Competencies for Learning Outcome 1 : Integrate API Endpoints with User Interface.

Knowledge	Skills	Attitudes
• Description of endpoint HTTP Methods • Description the typical structure of an API response	• Interpreting API Endpoints • Using API data on User Interface • Documenting IoT Web Application	• Being meticulous in checking the structure of API requests • Having a problem-solving mindset to enables you effectively analyse and resolve API issues. • Adaptability and Continuous Learning for integrating APIs and enhancing user interfaces as technologies • Having collaboration and feedback when documenting API Endpoints with User Interface.

Duration: 20hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe clearly endpoints HPPT methods based on the user interface in line with http communication protocol
2. Interpret correctly interface (API) endpoints based on to API documentation.
3. Use properly Interface (API) data the user interface in line with http communication protocol

4.Document effectively the report based on system functionality

Resources

Equipment	Tools	Materials
• Computer • Projector	• Text editor • Webserver • Web browser • DBMS	• Electricity • Internet

Indicative content 3.1: Interpret API Endpoint

Duration: 4 hrs

Theoretical Activity 3.1.1: Description of Endpoint HTTP method

Tasks:

1: Answer the following questions:

i. What are HTTP method API endpoint.
ii. How can you check URL structure in API endpoint?
iii. What are the types of request parameter used in API endpoint?
iv. Provide the typical structure of an API response
v. What are different types of API authentication
vi. What are error codes and error response format?.

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4. Ask question for clarification if any.

5. Read the key readings 3.1.1.

Key readings 3.1.1: : Description of Endpoint HTTP method

1.Description of each endpoint HTTP method

Sure, here's a breakdown of the typical meanings of HTTP methods used in API endpoints:

GET: This method is used to request data from a specified resource. It should only retrieve data and should not have any other effect on the data. GET requests can be cached and remain in the browser history, meaning they can be bookmarked and shared.

POST: This method is used to submit data to be processed to a specified resource. It often results in the creation of a new resource or the updating of an existing one. POST requests are not cached, and they do not remain in the browser history. They are commonly used for actions like submitting a form or uploading a file.

PUT: This method is used to update data on the server. It replaces the entire resource with the new data provided in the request. If the resource does not exist, PUT can

create a new resource with the specified data. PUT requests are idempotent, meaning that making the same request multiple times will have the same effect as making it once.

PATCH: Similar to PUT, PATCH is used to update data on the server. However, instead of replacing the entire resource, PATCH applies partial modifications to it. It is typically used when you want to update only specific fields of a resource without affecting the rest. Like PUT, PATCH requests are also idempotent.

DELETE: This method is used to delete a specified resource from the server. After a successful DELETE request, the resource is removed from the server. DELETE requests are idempotent, meaning that making the same request multiple times will have the same effect as making it once.

OPTIONS: This method is used to describe the communication options for the target resource. It is often used to request information about the supported methods, headers, or other capabilities of a server without actually fetching any data.

HEAD: This method is similar to GET but is used to request the headers of a resource instead of the actual content. It is often used to check the validity or existence of a resource without downloading its entire content.

TRACE: This method is used to echo the received request back to the client. It is mainly used for debugging or diagnostic purposes to see how the request changes as it passes through intermediate servers.

CONNECT: This method is used to establish a tunnel to the server identified by the target resource. It is typically used in conjunction with the HTTP CONNECT method to create a secure connection to a web server over HTTPS.

2.checking request parameters

Understanding the meanings and use cases of these HTTP methods is crucial for designing and implementing effective API endpoints.

When checking request parameters in the context of API endpoints, you typically examine the parameters sent along with the HTTP request. These parameters can provide additional information to the server about the action the client is requesting or the data it wants to manipulate. Here's how you can check request parameters:

Query Parameters: These are key-value pairs sent in the URL after the ? symbol. For example, in the URL https://api.example.com/resource?key1=value1&key2=value2, key1 and key2 are query parameters. In most web frameworks, you can access query parameters directly from the request object.

Path Parameters: Sometimes, parameters are included directly in the URL path itself. These are often used to specify a unique resource identifier. For example, in the URL https://api.example.com/resource/{id}, {id} is a path parameter. The value of the path parameter can be extracted from the URL path in server-side code.

Request Body: Parameters can also be sent in the body of a POST, PUT, or PATCH request. This is common when sending larger amounts of data or when the parameters are complex objects. In this case, you need to parse the request body to extract the parameters.

Headers: While not strictly request parameters in the traditional sense, headers can also convey important information to the server. For example, authentication tokens or content type information may be sent in headers. You can access headers from the request object in most web frameworks.

HTTP Method: The HTTP method used in the request (GET, POST, PUT, DELETE, etc.) can also be considered a parameter, as it indicates the action the client wants to perform. You can access the HTTP method from the request object in most web frameworks.

Cookies: Cookies are another way to send data with a request. While they are typically used for maintaining session state, they can also contain parameters relevant to the request. You can access cookies from the request object in most web frameworks.

When checking request parameters, it's important to validate and sanitize them to prevent security vulnerabilities such as injection attacks or malformed input. Additionally, ensure that all required parameters are present and handle any missing parameters gracefully.

3.API response structure

The API response structure should be consistent and easy to understand. It should include a status code, a message, and the actual data being returned. The message should provide a brief explanation of the status code, and the data should be in a well-defined format, such as JSON or XML

4.Check if the API requires authentication

To check if an API requires authentication, you typically need to refer to its documentation or explore its endpoints and configurations. Here are some common methods to determine if authentication is required:

Documentation Review: Most APIs provide comprehensive documentation that outlines their endpoints, request parameters, and authentication requirements.

Look for sections in the documentation that discuss authentication methods, such as API keys, OAuth tokens, or other forms of authentication.

Endpoint Access: Attempt to access an API endpoint that requires authentication without providing any credentials. If the API requires authentication, it will likely respond with an error message indicating that authentication is required or that the request is unauthorized.

Response Headers: Examine the response headers returned by API endpoints. Some APIs include headers like WWW-Authenticate or Authorization to indicate authentication requirements or status. However, not all APIs provide this information in response headers.

Error Messages: When making requests to the API without proper authentication, pay attention to the error messages returned by the API. Some APIs explicitly state that authentication is required in their error responses.

Community Forums or Support: If you're unsure about the authentication requirements of an API, you can also seek help from community forums, developer support channels, or contact the API provider directly for clarification.

Testing with Authentication: Attempt to access an authenticated endpoint using valid credentials. If you receive a successful response, it confirms that the API requires authentication.

Once you've determined that the API requires authentication, you'll need to implement the appropriate authentication mechanism in your integration code. This may involve obtaining API keys, OAuth tokens, or other credentials, and including them in your requests to authenticate with the API.

5.Check error codes and error response format.

To check error codes and error response formats for an API, you typically need to refer to its documentation. Here's how you can typically find this information:

Documentation Review: The API documentation should provide detailed information about the error codes and error response formats that the API might return. Look for sections in the documentation that cover error handling, status codes, and examples of error responses.

HTTP Status Codes: Most APIs use standard HTTP status codes to indicate the success or failure of a request. Common HTTP status codes for errors include:

4xx series: Client errors, such as 400 Bad Request, 401 Unauthorized, 404 Not Found, etc.

5xx series: Server errors, such as 500 Internal Server Error, 503 Service Unavailable, etc.

Error Response Body: Along with the HTTP status code, APIs often return a structured error response body containing additional details about the error. This response body may include:

Error code: A specific error code to identify the type of error.

Error message: A human-readable error message providing more information about the error.

Error details: Additional details or metadata about the error, such as invalid input parameters or missing authentication credentials.

Error documentation: Links or references to further documentation or resources for troubleshooting the error.

Example Responses: The API documentation may provide examples of error responses, showing the expected format of error messages and how to interpret them. These examples can be valuable for understanding how to handle errors in your integration code.

Testing: In addition to reviewing the documentation, you can also perform testing by intentionally triggering errors in your requests and observing the responses. This can help verify that your code correctly handles various error scenarios.

Community Forums or Support: If you're unable to find the information you need in the documentation, you can also seek help from community forums, developer support channels, or contact the API provider directly for assistance.

Understanding the error codes and error response formats of an API is essential for building robust and reliable integrations, as it enables you to handle errors gracefully and provide meaningful feedback to users when things go wrong.

Practical Activity 3.1.2: Interpreting API HTTP Method

Task: 1

1: Read key reading 3.1.2

2: Referring to key reading 3.1.2, As an IoT developper, you are asked to go to the computer lab to Interpret API Endpoints

3: Present your work to the trainer and whole class

4: Ask clarification where necessary

Key readings 3.1.2: Interpreting API HTTP Method

step to Interpret API Each Endpoint HTTP Method

Interpreting an API involves understanding the various endpoints it offers, the HTTP methods used for each endpoint, and how to interact with them. Here's a step-by-step guide to help you interpret an API:

Step 1: Understand API Documentation

- Read the Documentation: Start by reviewing the API documentation thoroughly. This will provide you with an overview of the available endpoints, their purposes, and required parameters.

Step 2: Identify Endpoints

- List Endpoints: Identify all the available endpoints in the API. An endpoint is typically a URL that corresponds to a specific resource or action.

Step 3: Determine HTTP Methods

- Understand HTTP Methods: Each endpoint will be associated with one or more HTTP methods. Common methods include:

- GET: Retrieve data from the server.

- POST: Send data to the server to create a new resource.

- PUT: Update an existing resource.

- DELETE: Remove a resource from the server.

Step 4: Analyze Request Parameters

- Check Required Parameters: For each endpoint, determine what parameters are needed. This can include path parameters, query parameters, and request body data.

- Understand Response Format: Know what format the API uses for responses (e.g., JSON, XML) and the structure of the response data.

Step 5: Test the Endpoints

- Use API Testing Tools: Utilize tools like Postman or curl to test the endpoints. This

allows you to send requests and observe the responses.

- Check Status Codes: Pay attention to HTTP status codes returned by the server (e.g., 200 for success, 404 for not found, 500 for server error).

Step 6: Handle Authentication

- Authentication Requirements: Determine if the API requires authentication (e.g., API keys, OAuth tokens) and how to include these in your requests.

Step 7: Implement Error Handling

- Understand Error Responses: Familiarize yourself with common error responses

Step 8: Review Rate Limits

- Check Rate Limits: Some APIs impose limits on the number of requests you can make in a given time frame. Be aware of these limits to avoid being blocked.

Step 9: Build Your Application

- Integrate with Your Application: Once you understand the endpoints and methods, you can start integrating the **API** into your application, ensuring to follow best practices for API usage.

To ensure that the URL structure, request parameters, and HTTP methods of an IoT API are correctly implemented, follow these systematic steps:

1. Review and Understand API Documentation

Before interacting with the API, the **API documentation** is essential for understanding how the API works. Documentation will typically provide:

Base URL of the API.

Endpoint paths for specific resources (e.g., devices, sensors).

Request parameters: Path, query, and body parameters.

HTTP methods to use (GET, POST, PUT, DELETE, etc.).

Response formats (usually JSON or XML).

2. Understand the URL Structure

The **URL structure** of an API typically follows this format:

```arduino
https://{api-domain}/{version}/{resource}/{id}
```

Check URL structure by verifying:

Protocol: Should be HTTPS for secure communication.

Base URL: Should match the domain of the API.

Versioning: Should be clearly defined, e.g., /v1/.

Resource endpoint: The resource should be relevant to what you're accessing, e.g., /devices/ or /sensors/.

ID (optional): Resource identifier (optional but required for specific requests targeting individual resources).

Steps to Check URL Structure:

Validate base URL: Ensure it follows the correct protocol and domain.

Check resource path: Ensure it correctly references the resources (e.g., /devices/ for device-related calls).

Verify resource identifiers: For endpoints targeting specific resources, ensure correct IDs are used.

Example URL to Check:

```bash
https://api.smartfarm.com/v1/devices/12345
```

v1: API version
devices: Resource path
12345: Unique device ID

3. Check Request Parameters

Request parameters can be sent in three ways:

Path parameters: Included directly in the URL.

Query parameters: Appended to the URL after a ?.

Body parameters: Included in the request body (for POST, PUT methods).

Steps to Check Request Parameters:

Path Parameters:

Ensure the resource ID or required parameter is correctly included in the path.

Example: /devices/67890 targets device 67890.

Query Parameters:

Verify the key-value pairs are correctly formatted and appended after a ?.

Multiple parameters are joined using &.

Example: /devices?status=active&location=farm1.

Body Parameters (for POST/PUT requests):

Check if the parameters are being passed in the request body (JSON, XML, etc.).

Ensure that required fields are included in the correct format.

Example Query Parameters to Check:

```
bash

GET https://api.smartfarm.com/v1/devices?status=active&location=farm1
```

- ✓ status=active: Filters active devices.
- ✓ location=farm1: Filters devices located at "farm1".

Steps to Check the Structure of API Responses in IoT API (HTTP Methods)

When interacting with an IoT API, ensuring the response structure is correct and meets the expected format is crucial for effective integration. Follow these steps to check the structure of API responses:

1. Review the API Documentation

- **Start by reviewing the API documentation** to understand the expected structure of the responses.
- Look for:
  - **Response format** (usually JSON or XML).
  - **Response codes** for different HTTP methods (GET, POST, PUT, DELETE).
  - **Schema or sample response** for specific endpoints

Example Documentation Information:

- **Format**: JSON

- **Response codes**: 200 OK, 201 Created, 400 Bad Request
- **Sample Response**:

```json
{
  "device_id": "12345",
  "status": "active",
  "last_updated": "2024-10-14T10:45:00Z"
}
```

2. Send an API Request

Use an API client like **Postman**, **cURL**, or **Insomnia** to send a request to the IoT API endpoint. For example, to retrieve device information:

Example GET Request:

```bash
GET https://api.smartfarm.com/v1/devices/12345
```

You can also use the following for other methods:

- ✓ POST: To create a resource.
- ✓ PUT: To update a resource.
- ✓ DELETE: To remove a resource.

3. Analyze the Response Format

- **Ensure the response format** matches what is specified in the documentation (e.g., JSON, XML).
- Common formats:
  - **JSON**: Data is structured as key-value pairs.
  - **XML**: Data is structured with tags.

Example JSON Response:

```json
{
  "device_id": "12345",
  "device_name": "Soil Sensor",
  "status": "active",
  "location": "Farm1",
  "last_updated": "2024-10-14T10:45:00Z"
}
```

4. Check the Status Code

- ✓ Every API response includes an **HTTP status code** that indicates the success or failure of the request. Make sure the status code aligns with the expected outcome.
- ✓ Common Status Codes:
- ✓ **200 OK**: Success for GET, PUT, or DELETE requests.
- ✓ **201 Created**: Success for POST requests.
- ✓ **204 No Content**: Success for DELETE requests (no content returned).
- ✓ **400 Bad Request**: Invalid request, often due to incorrect parameters.
- ✓ **404 Not Found**: The resource was not found (e.g., invalid device ID).
- ✓ **500 Internal Server Error**: Server-side error.

5. Validate the Structure of the Response Body

- ✓ Check that all the **required fields** are present in the response.
- ✓ Ensure the **data types** of the fields (string, integer, boolean, etc.) match the documentation.
- ✓ Verify that **nested data structures** (arrays, objects) are correctly formatted.

Steps to Validate the Response Structure:

- ✓ **Compare the response structure** with the API documentation's sample response or schema.
- ✓ **Ensure required fields** (like device_id, status, etc.) are present.
- ✓ **Check for missing or extra fields** that are not mentioned in the documentation.
- ✓ **Verify the data types** (e.g., string for device_name, boolean for status, etc.)

Example of Properly Structured Response:

```json
{
  "device_id": "12345",
  "device_name": "Irrigation Pump",
  "status": "active",
  "location": "Farm1",
  "last_updated": "2024-10-14T10:45:00Z",
  "sensors": [
    {
      "sensor_id": "56789",
      "type": "moisture",
      "value": 34.7
    }
  ]
}
```

Validate Error Handling in the Response

- ✓ Ensure that the API handles errors properly by testing invalid requests (e.g., providing incorrect parameters or device IDs). The response should include an appropriate error message and a status code.
- ✓ Example Error Response:

```json
{
  "error": {
    "code": 404,
    "message": "Device not found."
  }
}
```

Points to Remember

- The different HTTP methods used in API endpoints are
- (GET: Retrieves data from the server. POST: Sends new data to the server to create a new resource.PUT: Updates an existing resource with new data.PATCH: Partially updates an existing resource.DELETE: Removes a resource from the server.)

- the types of request parameters used in API endpoints are
    - (Path Parameters, Query Parameters, Body Parameters, Header Parameters)
- A typical API response consists of (Status, Data, Message, Error)
- The different types of API authentication are (API Key Authentication, OAuth, Basic Authentication, Bearer Token, Digest Authentication)
- to check error codes and error response format use (Check the Status Code, Inspect the Response Body, Test Edge Cases).

Step 1: Understand API Documentation

Step 2: Identify Endpoints

Step 3: Determine HTTP Methods

Step 4: Analyze Request Parameters

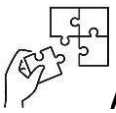
Step 5: Test the Endpoints

Step 6: Handle Authentication

Step 7: Implement Error Handling

Step 8: Review Rate Limits

Step 9: Build Your Application



Application of learning 3.1.

Your parent wants to find nearby restaurants using a location-based service, such as the Google Places API**,** while planning a weekend outing. You are required to do that task

Indicative content 3.2: Use API data on User Interface

Duration: 4 hrs

Theoretical Activity 3.2.1: Use API data on user interface

Tasks:

1. Answer the following questions: .
    i. What are the URL requests?
    ii. what are the HTTP request?
    iii. what are Testing and Debugging techniques used to identify the Use of API data on User Interface
2. Explain the task and provide clear work instructions.
3. make an explanation on URL requests and HTTP request. While explaining give the steps to follow.
4. Write your answers on paper, blackbord, flipchart or white board
5. Present your findings to the trainer or your classmates
6. Ask question for clarification if any.
7. Read the key readings 3.2.1.

Key readings 3.2.1: Use API data on user interface

- **Use API data on User Interface**

Using API data on a user interface involves fetching data from an API and displaying it to users in a meaningful way within the UI. Here's a basic guide on how to achieve this:

- ✓ **Fetch Data from API:** Use client-side code (such as JavaScript) to make HTTP requests to the API endpoints. You can use libraries like `fetch ()` or frameworks like Axios to simplify this process. Make sure to include any required authentication tokens or headers if the API requires them.
- ✓ **Handle Responses:** Once the API responds to your request, handle the response data appropriately. This may involve parsing JSON or XML data returned by the API and extracting the relevant information that you want to display on the UI.
- ✓ **Update UI:** Update the UI elements with the data retrieved from the API. This could involve populating tables, lists, charts, or other UI components with the fetched data. Use HTML, CSS, and JavaScript to dynamically update the UI based on the received data.

- ✓ **Error Handling:** Implement error handling mechanisms to deal with situations where API requests fail or encounter errors. Display meaningful error messages to users and provide options for retrying the request or contacting support if necessary.
- ✓ **Loading States:** While waiting for API responses, display loading indicators or placeholders to indicate to users that data is being fetched. This helps improve the user experience by providing feedback that the application is working in the background.
- ✓ **Pagination and Filtering:** If the API returns large amounts of data, consider implementing pagination or filtering options to improve performance and usability. Allow users to navigate through different pages of data or filter results based on specific criteria.
- ✓ **Real-time Updates:** If the API supports real-time updates (e.g., through WebSocket connections), implement mechanisms to update the UI in real-time as data changes on the server. This provides users with timely information and enhances the interactivity of the application.
- ✓ **Caching and Performance Optimization:** To improve performance and reduce unnecessary API calls, consider implementing client-side caching mechanisms to store previously fetched data. Only make API requests when necessary or when the cached data becomes stale.
- ✓ **Accessibility:** Ensure that the UI is accessible to all users, including those using screen readers or other assistive technologies. Use semantic HTML, provide alternative text for images, and ensure that interactive elements are keyboard accessible.

By following these steps, you can effectively integrate API data into your user interface, providing users with dynamic and interactive experiences that leverage the power of external data sources.

- **URL requests**

URL requests, also known as HTTP requests, are the backbone of communication between a client (such as a web browser or a mobile app) and a server. They are used to send a request to a specific URL (Uniform Resource Locator) to retrieve data or perform actions on a web server. Here's an overview of the different types of URL requests:

- **Endpoints**

Endpoints in the context of web development typically refer to specific URLs that are exposed by a web server to perform certain actions or retrieve specific resources. These endpoints are the entry points into the web application's functionality and are

accessed via HTTP requests. Here's a breakdown of the common types of endpoints:

- ✓ **Resource Endpoints:** These endpoints are used to interact with resources (data entities) in the system. They typically correspond to CRUD (Create, Read, Update, Delete) operations on resources. For example:

  /users: Endpoint to manage user resources.

  /posts: Endpoint to manage post resources.

  /products: Endpoint to manage product resources.

  Action Endpoints: These endpoints perform specific actions or operations that are not strictly CRUD operations on resources. They may trigger business logic, perform calculations, or initiate processes. For example:

  /login: Endpoint to authenticate users and generate a session token.

  /logout: Endpoint to invalidate a user's session token.

  /search: Endpoint to perform a search operation.

  /upload: Endpoint to handle file uploads.

- ✓ **Webhook Endpoints:** Webhooks are HTTP callbacks that are triggered by specific events. These endpoints are used to receive notifications or data from external services or systems. For example:

  /webhook/order: Endpoint to receive notifications about new orders.

  /webhook/payment: Endpoint to receive notifications about payment events.

- ✓ **Utility Endpoints:** These endpoints provide utility or auxiliary functionality that supports the main features of the application. They may return metadata, perform health checks, or provide configuration information. For example:

  /ping: Endpoint to check if the server is alive.

  /status: Endpoint to retrieve system status information.

  /config: Endpoint to retrieve application configuration settings.

  Nested Endpoints: Some APIs use nested endpoints to represent hierarchical relationships between resources. For example:

  /users/{userId}/posts: Endpoint to retrieve posts created by a specific user.

  /products/{productId}/reviews: Endpoint to retrieve reviews for a specific

product.

Versioned Endpoints: APIs may use versioning in their endpoints to manage changes and updates to the API over time. For example:

/v1/users: Version 1 of the user management endpoint.

/v2/users: Version 2 of the user management endpoint.

Endpoints are the building blocks of RESTful APIs, providing a structured way for clients to interact with the server and access its functionality. Properly defining and organizing endpoints is essential for creating a well-designed and intuitive API.

- **Method**

In web development, the term "method" typically refers to the HTTP method used in a request to an API endpoint. HTTP methods define the action that the client (such as a web browser or a mobile app) wants to perform on the server. Here are the most common HTTP methods:

- ✓ **GET:** Retrieves data from the server specified by the URL. It should only retrieve data and should not have any other effect on the data. GET requests can be cached and remain in the browser history, meaning they can be bookmarked and shared.
- ✓ **POST:** Submits data to be processed to a specified resource. It often results in the creation of a new resource or the updating of an existing one. POST requests are not cached, and they do not remain in the browser history. They are commonly used for actions like submitting a form or uploading a file.
- ✓ **PUT:** Updates data on the server specified by the URL. It replaces the entire resource with the new data provided in the request. If the resource does not exist, PUT can create a new resource with the specified data. PUT requests are idempotent, meaning that making the same request multiple times will have the same effect as making it once.
- ✓ **PATCH:** Similar to PUT, PATCH is used to update data on the server. However, instead of replacing the entire resource, PATCH applies partial modifications to it. It is typically used when you want to update only specific fields of a resource without affecting the rest. Like PUT, PATCH requests are also idempotent.
- ✓ **DELETE:** Deletes the resource specified by the URL on the server. After a successful DELETE request, the resource is removed from the server. DELETE requests are idempotent, meaning that making the same request multiple times will have the same effect as making it once.
- ✓ **OPTIONS:** Requests information about the communication options for the target resource. It is often used to determine the supported methods,

headers, or other capabilities of a server.

- ✓ **HEAD:** Similar to GET, but only retrieves the headers of the resource specified by the URL without fetching the actual content. It is often used to check the validity or existence of a resource.
- ✓ **TRACE:** Echoes the received request back to the client. It is primarily used for debugging or diagnostic purposes to see how the request changes as it passes through intermediate servers.
- ✓ **CONNECT:** Establishes a tunnel to the server specified by the URL for use with protocols like HTTPS. It is typically used in conjunction with the HTTP CONNECT method to create a secure connection to a web server.

These HTTP methods, combined with the URL and any additional request headers or parameters, determine the action that the client wants to perform on the server and are fundamental to the HTTP protocol.

- **Data (or body)**

The "data" or "body" in the context of HTTP requests refers to the payload sent by the client to the server in the request message. It contains the information or content that the client wants to transmit to the server, such as form data, JSON objects, or files. The structure and format of the data depend on the type of request and the application's requirements. Here are the common scenarios for including data in HTTP requests:

- ✓ **GET Requests**: Typically, GET requests do not have a request body, as they are used to retrieve data from the server, and any parameters are usually included in the URL query string.
- ✓ **POST Requests**: POST requests often include data in the request body. This data can be in various formats, such as form data (sent with application/x-www-form-urlencoded or multipart/form-data content types), JSON (sent with application/json content type), or XML.
- ✓ **PUT and PATCH Requests**: PUT and PATCH requests are used to update resources on the server. They also include data in the request body, typically in JSON format, representing the new state of the resource.
- ✓ **DELETE Requests**: DELETE requests may include data in the request body for specifying additional parameters or criteria for the deletion operation. However, it's less common to include data in DELETE requests compared to other HTTP methods.

To include data in the request body, clients typically serialize the data into a format that can be transmitted over the network, such as JSON or form-encoded data. The server then parses the request body and processes the data according to the

application's logic.

Here's an example of a JSON request body for a POST request:

```
{
  "username": "john_doe",
  "email": "john.doe@example.com",
  "password": "securepassword"
}
```

And here's an example of form data for a POST request:

username=john_doe&email=john.doe@example.com&password=securepassword

- **Status (code and message)**

The "status code" and "message" in the context of HTTP responses refer to the information returned by the server to the client after processing a request. They provide feedback about the outcome of the request and any relevant information or instructions for the client. Here's how they typically work:

- ✓ **Status Code:** The status code is a three-digit numerical code that indicates the outcome of the HTTP request. It is included in the response header and provides information about whether the request was successful, encountered an error, or requires further action. Some common status code ranges and their meanings include:

  1xx Informational: Indicates that the server has received the request and is continuing the process. Example: 100 Continue.

  2xx Success: Indicates that the request was successful. Example: 200 OK.

  3xx Redirection: Indicates that further action needs to be taken to complete the request. Example: 301 Moved Permanently.

  4xx Client Error: Indicates that the request contains bad syntax or cannot be fulfilled. Example: 404 Not Found.

  5xx Server Error: Indicates that the server failed to fulfill an apparently valid request. Example: 500 Internal Server Error.

- ✓ **Message:** The message is a short human-readable description associated with the status code. It provides additional context or details about the outcome of the request. For example, for a 404 Not Found status code, the message might be "The requested resource could not be found." The

message is often included in the response body, but it can also be found in the response header or status line.

Here's an example of an HTTP response containing status code and message:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
 "status": "success",
 "data": {
  "message": "Resource retrieved successfully",
  "user": {
   "id": 123,
   "username": "john_doe"
  }
 }
}
```

In this example, the status code 200 indicates a successful response, and the message "Resource retrieved successfully" provides additional information about the outcome of the request. The actual data associated with the request (in this case, information about a user) is included in the response body.

- **HTTP requests**

HTTP (Hypertext Transfer Protocol) requests are the foundation of communication between clients (such as web browsers, mobile apps, or other servers) and servers on the internet. They are used to send requests for resources or perform actions on a server, and they consist of several components, including the request method, headers, and optional body. Here's a breakdown of **the main components of an HTTP request:**

- ✓ **Request Line:** The request line is the first line of an HTTP request and contains the following components:
- ✓ **HTTP Method:** Specifies the action to be performed by the server. Common HTTP methods include GET, POST, PUT, DELETE, etc.
- ✓ **Request Target (URL):** Specifies the URL of the resource or endpoint on the server that the client wants to access or interact with.

- ✓ **HTTP Version:** Indicates the version of the HTTP protocol being used, such as HTTP/1.1 or HTTP/2.

  Example:

  GET /api/users HTTP/1.1

- ✓ **Headers:** HTTP headers provide additional information about the request, such as the client's user agent, content type, authentication credentials, and more. Headers are key-value pairs separated by a colon (:), and multiple headers are separated by line breaks.

  Example:

  Host: example.com

- ✓ **User-Agent:** Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36

  Accept: application/json

  Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

- ✓ **Body (Optional):** Some HTTP requests, such as POST or PUT requests, may include a message body containing data to be sent to the server. The body can contain any type of data, such as form data, JSON, XML, or binary data, depending on the request's purpose.

Example:

```
POST /api/users HTTP/1.1
Content-Type: application/json
{
 "username": "john_doe",
 "email": "john.doe@example.com",
 "password": "securepassword"
}
```

HTTP requests allow clients to interact with servers by retrieving resources, submitting data, updating information, or performing other actions. The

server processes the request and returns an HTTP response, which contains the requested data or indicates the outcome of the request.

- **POST requests to create records**

POST requests are commonly used to create new records or resources on a server. When making a POST request to create a record, you typically include the data for the new record in the request body, usually in JSON format. **Here's an example of how you might structure a POST request to create a new record:**

- ✓ **HTTP Method:** Use the POST method to indicate that you want to create a new record.
- ✓ **Request URL:** Specify the URL of the endpoint where the new record should be created. This URL typically corresponds to a resource-specific endpoint.
- ✓ **Headers:** Include any necessary headers in the request, such as the Content-Type header to specify the format of the request body (e.g., application/json).
- ✓ **Request Body:** Include the data for the new record in the request body. This data should be formatted according to the requirements of the server's API. For example, if the server expects JSON data, you would include a JSON object representing the new record.

Here's an example of a POST request to create a new user record:

```
POST /api/users HTTP/1.1

Host: example.com

Content-Type: application/json

Authorization: Bearer <access_token>

{

 "username": "john_doe",

 "email": "john.doe@example.com",

 "password": "securepassword"

}
```

In this example:

The request method is POST.

The request URL is /api/users, indicating that the new user record should be created in the users resource.

The Content-Type header specifies that the request body contains JSON data.

The request body contains the data for the new user record, including the username, email, and password fields.

When the server receives this request, it processes the data in the request body and creates a new user record accordingly. The server then returns an HTTP response indicating the outcome of the request, typically with a status code of 201 (Created) if the record was successfully created.

- **GET request to read or get a resource from the server**

A GET request is used to retrieve data or resources from a server. When making a GET request, you specify the URL of the resource you want to retrieve, and the server responds with the requested data. **Here's how you might structure a GET request to read or get a resource from the server:**

- ✓ **HTTP Method:** Use the GET method to indicate that you want to retrieve data from the server.
- ✓ **Request URL:** Specify the URL of the resource you want to retrieve. This URL typically corresponds to a specific endpoint that serves the requested resource.
- ✓ **Headers:** Include any necessary headers in the request. For a simple GET request, you may not need to include any additional headers.

Here's an example of a GET request to retrieve a user record:

GET /api/users/123 HTTP/1.1

Host: example.com

In this example:

The request method is GET.

The request URL is /api/users/123, indicating that the user record with the ID 123 should be retrieved from the users resource.

There are no additional headers included in the request.

When the server receives this request, it looks for the user record with the ID 123 in the users resource. If the record exists, the server responds with an HTTP response containing the requested data, typically with a status code of 200 (OK). The response body contains the data for the user record, which the client can then use as needed. If the record does not exist or there is an error, the server will respond with an appropriate status code and possibly an error message in the response body.

PUT and PATCH requests to update records

PUT and PATCH requests are both used to update existing records on a server, but they differ in their behavior and the extent of the modifications they make. **Here's how you might structure PUT and PATCH requests to update records:**

- ✓ **PUT Request:** Use a PUT request when you want to replace an existing record with a new one. When making a PUT request, you typically include the entire updated record in the request body.
- ✓ **HTTP Method:** Use the PUT method to indicate that you want to replace the entire record with a new one.
- ✓ **Request URL:** Specify the URL of the resource you want to update, typically including the ID of the specific record.
- ✓ **Headers:** Include any necessary headers in the request, such as the Content-Type header to specify the format of the request body (e.g., application/json).
- ✓ **Request Body:** Include the data for the updated record in the request body. This data should represent the entire updated record, including any fields that are being modified.

Example PUT request:

```
PUT /api/users/123 HTTP/1.1

Host: example.com

Content-Type: application/json

Authorization: Bearer <access_token>

{

 "username": "new_username",

 "email": "new_email@example.com",

 "password": "new_secure_password"

}
```

- ✓ **PATCH Request:** Use a PATCH request when you want to make partial modifications to an existing record. When making a PATCH request, you typically include only the specific fields that are being updated in the request body.
- ✓ **HTTP Method:** Use the PATCH method to indicate that you want to apply partial modifications to the record.
- ✓ **Request URL:** Specify the URL of the resource you want to update, typically

including the ID of the specific record.

- ✓ **Headers:** Include any necessary headers in the request, such as the Content-Type header to specify the format of the request body (e.g., application/json).
- ✓ **Request Body:** Include only the data for the specific fields that are being updated in the request body. The server will apply these modifications to the existing record.

Example PATCH request:

```
PATCH /api/users/123 HTTP/1.1

Host: example.com

Content-Type: application/json

Authorization: Bearer <access_token>

{

  "email": "new_email@example.com"

}
```

In both examples:

The request method is either PUT or PATCH, depending on the desired behavior.

The request URL specifies the URL of the resource to be updated, including the ID of the specific record.

The request body contains the data for the updated fields, either representing the entire updated record (for PUT requests) or only the specific fields being modified (for PATCH requests).

When the server receives these requests, it processes the data in the request body and updates the corresponding record accordingly. The server then returns an HTTP response indicating the outcome of the request, typically with a status code of 200 (OK) if the update was successful.

- **DELETE request to delete a resource from a server**

A DELETE request is used to delete a resource from a server. When making a DELETE request, you specify the URL of the resource you want to delete, and the server responds by deleting the resource if it exists. **Here's how you might structure a DELETE request to delete a resource from the server:**

- ✓ **HTTP Method:** Use the DELETE method to indicate that you want to delete a resource from the server.

- ✓ **Request URL:** Specify the URL of the resource you want to delete. This URL typically corresponds to the specific endpoint that serves the resource you want to delete.
- ✓ **Headers:** Include any necessary headers in the request. For a simple DELETE request, you may not need to include any additional headers.

Here's an example of a DELETE request to delete a user record:

DELETE /api/users/123 HTTP/1.1

Host: example.com

Authorization: Bearer <access_token>

In this example:

The request method is DELETE.

The request URL is /api/users/123, indicating that the user record with the ID 123 should be deleted from the user's resource.

The Authorization header contains an access token, which may be required for authentication and authorization purposes.

When the server receives this request, it looks for the user record with the ID 123 in the users resource. If the record exists, the server deletes it from the resource and responds with an HTTP response indicating that the deletion was successful, typically with a status code of 204 (No Content). If the record does not exist or there is an error, the server will respond with an appropriate status code and possibly an error message in the response body.

- **Testing and Debugging**

Testing and debugging are critical aspects of software development, ensuring that applications function correctly, efficiently, and reliably. Here's how you can approach testing and debugging in your development process:

- ✓ **Unit Testing:** Write unit tests to verify that individual components (functions, methods, classes) of your code behave as expected. Use testing frameworks like JUnit (for Java), pytest (for Python), or Jasmine (for JavaScript) to automate the execution of tests and report any failures.
- ✓ **Integration Testing:** Conduct integration tests to verify interactions between different components of your application. Test how these components work together and whether they integrate smoothly. Tools like Selenium (for web applications) or Postman (for APIs) can help automate integration tests.
- ✓ **End-to-End (E2E) Testing:** Perform end-to-end tests to validate the entire

flow of your application from start to finish. E2E tests simulate real user interactions and verify that the application behaves correctly under various scenarios. Frameworks like Cypress or Protractor are commonly used for E2E testing.

- ✓ **Manual Testing:** While automated testing is essential, manual testing by human testers is also valuable. Manual testing can uncover usability issues, edge cases, and user experience problems that automated tests might miss.
- ✓ **Debugging Tools:** Use debugging tools provided by your programming language or development environment to identify and resolve issues in your code. Debuggers allow you to step through your code line by line, inspect variables, and track the flow of execution.
- ✓ **Logging:** Incorporate logging statements into your code to capture important information, such as variable values, function calls, and error messages. Logging can help you understand the behavior of your application and diagnose problems during runtime.
- ✓ **Code Reviews:** Conduct code reviews with peers to get feedback on your code and identify potential issues early in the development process. Code reviews help ensure code quality, maintainability, and adherence to best practices.
- ✓ **Error Handling:** Implement robust error handling mechanisms in your code to gracefully handle unexpected situations and provide meaningful error messages to users or developers. Proper error handling can prevent crashes and improve the user experience.
- ✓ **Performance Testing:** Evaluate the performance of your application under various load conditions to ensure it can handle expected traffic levels efficiently. Use tools like Apache JMeter or Gatling to simulate heavy traffic and measure response times, throughput, and resource utilization.
- ✓ **Continuous Integration/Continuous Deployment (CI/CD):** Implement CI/CD pipelines to automate the testing, building, and deployment of your application. CI/CD helps catch issues early, maintain code quality, and deliver updates to users quickly and reliably.

By incorporating these testing and debugging practices into your development process, you can build robust, reliable, and high-quality software that meets user expectations and delivers a positive user experience.

- **DATA RENDERING**

Data rendering in the context of using API data on a user interface (UI) refers to the process of taking data retrieved from an API and displaying it in a meaningful and user-friendly way on a web or mobile application. Here's a breakdown of how this

works:

1. **Fetching Data from an API**

- API Call: The first step involves making a request to an API endpoint using an HTTP method (like GET) to fetch data. This can be done using tools like `fetch` in JavaScript or libraries like Axios.

- Receiving Data: The API responds with data, usually in formats like JSON or XML.

2. Processing the Data

- Parsing Data: Once the data is received, it often needs to be parsed. For example, if the data is in JSON format, it can be converted into a JavaScript object for easier manipulation.

- Data Transformation: Sometimes, the raw data requires transformation or filtering to fit the needs of the UI. This might involve sorting, aggregating, or formatting the data.

3. Rendering Data in the UI

- Dynamic Rendering: The processed data is then rendered in the UI. This can involve creating HTML elements (like tables, lists, or charts) and populating them with the API data.

- Libraries and Frameworks: Many developers use libraries or frameworks (like React, Angular, or Vue.js) to facilitate dynamic rendering. These tools allow for efficient updates to the UI when the underlying data changes.

- Static vs. Dynamic Content: Depending on the application, some data may be rendered statically (fixed content) while other parts may be dynamic (updating in real-time based on user actions or new data).

4. User Interaction

- Interactive Elements: Data rendered in the UI can include interactive components, such as buttons, dropdowns, or charts that users can click on to explore more details or filter the data.

- Real-Time Updates: In some applications, data may update in real-time (e.g., through WebSockets or periodic API calls), necessitating re-rendering of certain UI elements to reflect the latest data.

5. Error Handling and Loading States

- Loading Indicators: When fetching data, it's common to show loading indicators to inform users that data is being retrieved.

- Error Handling: If the API call fails (due to network issues or server errors), proper error handling should be implemented to inform users and possibly provide options to retry.

6. Accessibility and Usability

- Ensuring Accessibility: When rendering data in the UI, it's crucial to consider accessibility features, ensuring that all users can interact with the data effectively, including those using screen readers.

Example Use Case

Imagine a weather application that fetches weather data from an API. The process would look like this:

1. API Call: The app makes a call to a weather API to retrieve current weather data.

2. **Data Processing:** The app receives and processes the data, extracting relevant information like temperature, humidity, and conditions.

3. Rendering: The app dynamically updates the UI to display the current weather, perhaps in a card format with icons and text.

4. User Interaction: Users can click to see a forecast for the next few days, which may involve another API call to fetch that data.

- **Show Loading States and Progress Indicators**

Loading states and progress indicators are essential elements in user interfaces, indicating to users that the application is processing data or performing an action. They provide feedback and reassure users that their request is being handled, especially when dealing with tasks that may take some time to complete. Here are some common ways to implement loading states and progress indicators in web and mobile applications:

1. **Spinner or Loading Animation:**

Display a spinner or loading animation to indicate that the application is loading data or performing an action.

Use CSS animations or pre-built spinner libraries to create visually appealing loading indicators.

Position the spinner prominently on the screen to catch the user's attention and convey that something is happening.

2. **Progress Bar:**

Show a progress bar to visualize the progress of a task, such as downloading a file or uploading data.

Update the progress bar dynamically as the task progresses, providing real-time feedback to users.

Consider using determinate or indeterminate progress bars based on whether the duration or completion of the task is known.

3. **Skeleton Screens:**

Use skeleton screens to provide a placeholder layout while content is being loaded.

Display empty containers or wireframe-like representations of the content's structure to give users a sense of what to expect.

Gradually replace skeleton screens with actual content as data becomes available, creating a smooth transition.

4. **Overlay or Modal:**

Overlay the loading indicator or progress bar on top of the current screen content to prevent user interactions during loading.

Use a modal dialog to focus the user's attention on the loading process and prevent them from interacting with other parts of the application.

5. **Text or Message:**

Display text messages or notifications to inform users about the progress of a task or the reason for the delay.

Provide descriptive messages, such as "Loading..." or "Please wait while we process your request," to keep users informed and reduce frustration.

6. **Throttling or Debouncing:**

Implement throttling or debouncing techniques to manage the frequency of loading indicators and prevent them from appearing unnecessarily.

Throttle requests to backend servers to avoid overwhelming them with too many simultaneous requests.

Debounce user input events to wait for a short period of inactivity before triggering

a loading indicator, reducing visual clutter.

7. **Accessibility Considerations:**

Ensure that loading indicators are accessible to users with disabilities by providing alternative text or aria attributes.

Use semantic HTML elements and ARIA roles to convey the purpose and state of the loading indicator to screen readers and assistive technologies.

By incorporating loading states and progress indicators into your user interface design, you can enhance the user experience and improve usability by providing clear feedback and managing user expectations during data processing and interactions.

- **Preserve Filtering and Sorting**

Preserving filtering and sorting states is crucial for providing a seamless user experience, especially in applications where users frequently interact with large datasets. Here are some strategies to preserve filtering and sorting states:

1. **URL Parameters:** Encode filtering and sorting criteria into the URL as query parameters. This allows users to bookmark or share specific views of the data and provides a consistent way to navigate back to a particular state.

Example:

https://example.com/products?category=electronics&sort=price_asc

Browser History API: Use the browser's History API (e.g., pushState in JavaScript) to update the URL and preserve filtering and sorting states when users apply filters or sort options. This enables users to navigate back and forth through the application's history while maintaining their selected preferences.

2. **Session Storage or Local Storage:** Store filtering and sorting criteria in the browser's session storage or local storage. This allows the application to persist user preferences across page reloads and browser sessions.
3. **Cookies:** Use cookies to store filtering and sorting preferences on the client side. This approach is suitable for applications where user preferences need to be preserved even if the user closes the browser and returns later.
4. **Backend Storage:** Store filtering and sorting states on the server side, associated with the user's session or account. This approach allows users to access their preferences across different devices and browsers, maintaining consistency in their data views.
5. **Query Parameters in API Requests:** If the data is fetched from an API, include

filtering and sorting criteria as query parameters in API requests. This ensures that the server-side data retrieval reflects the user's selected preferences.

6. **URL Hash Fragments:** Use URL hash fragments to encode filtering and sorting states in the URL. While similar to URL parameters, hash fragments are not sent to the server, making them suitable for preserving states purely on the client side.
7. **State Management Libraries:** If using a JavaScript framework like React or Vue.js, utilize state management libraries (e.g., Redux, Vuex) to manage filtering and sorting states centrally across components. This ensures consistency and simplifies state management logic.

By implementing one or a combination of these strategies, you can provide users with a seamless experience by preserving their filtering and sorting preferences across interactions with your application, resulting in improved usability and satisfaction.

- **Preserve Pagination and Infinite Scrolling**

Preserving pagination and infinite scrolling states is essential for maintaining the user's context and providing a seamless browsing experience, especially in applications with large datasets. Here are strategies to preserve pagination and infinite scrolling states:

1. **URL Parameters for Pagination:**

Encode pagination parameters (e.g., page number, page size) into the URL as query parameters.

Example: https://example.com/products?page=2&pageSize=20

This allows users to bookmark or share specific pages and provides a consistent way to navigate back to a particular page.

2. **Browser History API:**

Use the browser's History API (e.g., pushState in JavaScript) to update the URL and preserve pagination states when users navigate between pages.

This enables users to navigate back and forth through the application's history while maintaining their current page.

Session Storage or Local Storage:
Store pagination states (e.g., current page number) in the browser's session storage or local storage.

This allows the application to persist pagination states across page reloads and browser sessions.

- ✓ **Cookies:**

Use cookies to store pagination preferences on the client side.

This approach is suitable for applications where users need to maintain their pagination settings across different sessions or devices.

- ✓ **Backend Storage:**

Store pagination states on the server side, associated with the user's session or account.

This allows users to access their pagination preferences across different devices and browsers.

- ✓ **Scroll Position for Infinite Scrolling:**

Record the scroll position of the page when users scroll through content in an infinite scrolling interface.

When users navigate away from the page and return later, restore the scroll position to where they left off.

This provides a seamless browsing experience, allowing users to pick up where they left off without losing their place in the content.

- ✓ **State Management Libraries:**

If using a JavaScript framework like React or Vue.js, utilize state management libraries (e.g., Redux, Vuex) to manage pagination and scrolling states centrally across components.

This ensures consistency and simplifies state management logic, especially in complex applications with multiple components.

By implementing these strategies, you can preserve pagination and infinite scrolling states in your application, providing users with a seamless browsing experience and allowing them to navigate through large datasets efficiently.

- **Ensure responsiveness design**

Ensuring responsive design is crucial for providing a consistent and user-friendly experience across different devices and screen sizes. Here are some best practices to achieve responsive design:

- ✓ **Use a Mobile-First Approach:** Design the user interface with mobile devices in mind first, and then progressively enhance the layout for larger screens. This ensures that the application is optimized for smaller screens and can

gracefully scale up to larger ones.

- ✓ **Fluid Layouts:** Use fluid or flexible layouts that adapt to the available screen space. Avoid fixed-width layouts that may cause horizontal scrolling on smaller screens. Use percentages or relative units (e.g., percentages, ems, rems) for sizing elements rather than pixels.
- ✓ **Media Queries:** Use CSS media queries to apply different styles based on the device's screen size, resolution, and orientation. Define breakpoints where the layout or design changes to accommodate different screen sizes. Common breakpoints include mobile (up to 767px), tablet (768px to 1023px), and desktop (1024px and above).
- ✓ **Responsive Images and Media:** Optimize images for different screen resolutions and sizes. Use responsive image techniques such as srcset and sizes attributes in HTML or CSS background images with media queries to serve appropriate images based on the device's capabilities.
- ✓ **Flexible Typography:** Use relative units (e.g., ems, rems) for font sizes and line heights to ensure that text scales proportionally across different screen sizes. Consider using viewport-relative units (e.g., vw, vh) for font sizes to maintain readability on all devices.
- ✓ **Touch-Friendly Design:** Design interactive elements (e.g., buttons, links, menus) to be easily tappable on touch devices. Use larger touch targets and provide ample spacing between elements to prevent accidental taps.
- ✓ **Content Prioritization:** Prioritize content based on its importance and relevance to users. On smaller screens, focus on essential content and actions, and consider hiding or collapsing less critical elements. Use progressive disclosure techniques to reveal additional content as needed.
- ✓ **Viewport Meta Tag:** Include the viewport meta tag in the HTML head to control the viewport behavior on mobile devices. Set the viewport width to device-width and enable scaling to ensure that the content fits within the device's screen and is not zoomed in by default.
- ✓ **Test Across Devices and Viewports:** Test the application on various devices, screen sizes, and orientations to ensure compatibility and responsiveness. Use browser developer tools or online testing services to simulate different devices and viewports during development and testing.
- ✓ **Accessibility Considerations:** Ensure that the responsive design is accessible to users with disabilities. Use semantic HTML, provide descriptive alternative text for images, and ensure that interactive elements are keyboard accessible and usable with screen readers.

By following these best practices, you can create a responsive design that adapts seamlessly to different devices and screen sizes, providing an optimal user

experience for all users.

Practical Activity 3.2.2: Use API data on User Interface

Task: 1

1: Read key reading 3.2.2.
2: Referring to key reading 3.2.2., You are requested to go to the computer lab where trainees Use API data on User Interface.
3: Present your work to the trainer and whole class
4: Ask clarification where necessary.

Key readings 3.2.2: Use API data on User Interface

- **Step to Use API data on User Interface**

Integrating API endpoints with a user interface (UI) involves several steps to ensure that data is fetched, processed, and displayed effectively. Here's a structured approach to achieve this:

Step 1: Understand the API

- Read the Documentation: Familiarize yourself with the API documentation to understand the available endpoints, required parameters, authentication methods, and response formats.

- Identify Endpoints: Determine which endpoints you will need for your application based on the data you want to display.

Step 2: Set Up Your Development Environment

- Choose a Technology Stack: Decide on the front-end framework or library you will use (e.g., React, Angular, Vue.js) and set up your development environment.

- Install Necessary Libraries: If using JavaScript, consider installing libraries like Axios or Fetch for making API calls.

Step 3: Create the User Interface

- Design the UI: Sketch or use design tools to create a layout of the UI that will display the API data. Consider how users will interact with the data.

- Build UI Components: Create the necessary UI components (e.g., buttons, tables, cards) that will hold the data fetched from the API.

Step 4: Implement API Calls

- Make API Requests: Use the chosen library to make HTTP requests to the API endpoints. This is typically done in lifecycle methods or hooks (like `useEffect` in React).

```
javascript

// Example using Fetch API

fetch('https://api.example.com/data')

  .then(response => response.json())

  .then(data => {

    // Process and render the data

  })

  .catch(error => {

    // Handle errors

  });
```

Step 5: Handle Responses

- Process Data: Once the data is received, parse it and extract the information needed for the UI.

- Error Handling: Implement error handling to manage scenarios where the API call fails (e.g., displaying an error message to the user)

Step 6: Render Data in the UI

- Update the UI: Use the processed data to update the UI dynamically. This may involve setting the state in your application to trigger a re-render.

```
javascript

// Example in React

const [data, setData] = useState([]);

useEffect(() => {

  fetch('https://api.example.com/data')
```

```
    .then(response => response.json())
    .then(data => setData(data))
    .catch(error => console.error('Error fetching data:', error));
}, []);
```

Step 7: Implement User Interactions

- Add Interactivity: Implement features that allow users to interact with the data, such as filtering, sorting, or pagination. This may involve additional API calls based on user actions.

- Update UI Based on Interaction: Ensure that the UI updates accordingly when users interact with it (e.g., changing filters or selecting items).

Step 8: Optimize Performance

- Loading States: Show loading indicators while data is being fetched to improve user experience.

- Pagination and Caching: For large datasets, consider implementing pagination or caching strategies to enhance performance and reduce API calls.

Step 9: Test the Integration

- Testing: Test the integration thoroughly to ensure that data is fetched correctly, the UI displays data as expected, and interactions work without issues.

- Cross-Browser Compatibility: Ensure that the UI works well across different browsers and devices.

Step 10: Deploy the Application

- Deployment: Once everything is working as intended, deploy your application to a hosting platform so users can access it.

- Monitor Performance: After deployment, monitor the application for performance issues and user feedback to make necessary improvements.

Points to Remember

- The URL requests (Endpoints Method Headers Data (or body), Status (code and message))

- HTTP request (POST requests to create records, GET request to read or get a resource from the server, PUT and PATCH requests to update records, DELETE request to delete a resource from a server)
- Testing and Debugging techniques used to identify the Use of API data on User Interface used unit testing, intergration testing, end to end testing, manual testing
- Data rendering in the context of using API data on a user interface (UI) refers to the process of taking data retrieved from an API and displaying it in a meaningful and user-friendly way on a web or mobile application.

  Step 1: Understand the API

  Step 2: Set Up Your Development Environment

  Step 3: Create the User Interface

  Step 4: Implement API Calls

  Step 5: Handle Responses

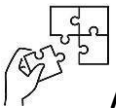
  Step 6: Render Data in the UI

  Step 7: Implement User Interactions

  Step 8: Optimize Performance

  Step 9: Test the Integration

  Step 10: Deploy the Application



Application of learning 3.2.2.

You are tasked with building a weather dashboard that displays the current weather and forecast for a selected city. The app retrieves data from a public weather API, allows users to search for cities, and displays the weather data dynamically. You will handle various API requests, ensure correct data rendering, and make sure the dashboard is responsive

Indicative content 3.3: Document IoT Web Application

Duration: 2 hrs

Theoretical Activity 3.3.1: Document IoT Web Application

Tasks:

1: Answer the following questions:

i. What is User Guide Documentation in IoT web application?
ii. How to do Report of Work Done in IoT web application?
iii. Provide Bills of quantities

2: Write your answers on paper, blackboard, flipchart or white board

3: Present your findings to the trainer or your classmates

4.Ask question for clarification if any.

5. Read the key readings 3.3.1

Key readings 3.2.2: Document IoT Web Application

- **Document IoT Web Application**

Documenting an IoT (Internet of Things) web application involves creating comprehensive documentation that covers various aspects of the application, including its architecture, components, functionality, APIs, data flows, and deployment. Here's a suggested outline for documenting an IoT web application:

✓ **Introduction**

Overview: Brief introduction to the IoT web application, its purpose, and key features.

Audience: Description of the intended audience for the documentation.

Architecture

- **High-Level Architecture:** Overview of the application's architecture, including client-side components (e.g., web interface), server-side components (e.g., backend APIs, databases), and IoT devices (e.g., sensors, actuators).

- **Data Flow:** Description of how data flows through the system, from IoT devices to the backend server and to the client interface.
- **Scalability and Performance:** Considerations for scaling the application to handle a large number of IoT devices and users.

✓ **Components**

- **Client-Side Components:** Description of the web interface, including user interface elements, navigation, and interaction flows.
- **Server-Side Components:** Overview of the backend server, APIs, databases, and other server-side components responsible for processing data from IoT devices and serving client requests.
- **IoT Devices:** Description of the IoT devices used in the application, including their types, capabilities, and communication protocols.

✓ **Functionality**

- **User Roles and Permissions:** Description of user roles (e.g., administrators, regular users) and their permissions within the application.

✓ **Features:** Detailed description of the application's features, including device management, data visualization, alerts, notifications, and automation.

- **API Documentation**

✓ **Overview:** Description of the APIs exposed by the backend server, including endpoints, request/response formats, authentication mechanisms, and usage examples.

✓ **Authentication and Authorization:** Explanation of how authentication and authorization are handled in the API, such as OAuth, JWT tokens, or API keys.

- **Data Model**

✓ **Data Schema:** Description of the data model used in the application, including entities, attributes, relationships, and data types.

✓ **Data Storage:** Overview of the databases or data stores used to store application data, including relational databases, NoSQL databases, or cloud storage services.

- **Deployment**

✓ **Deployment Architecture:** Description of the deployment architecture, including hosting environments (e.g., cloud platforms, on-premises servers),

deployment models (e.g., monolithic, microservices), and deployment tools (e.g., Docker, Kubernetes).

- ✓ **Deployment Steps:** Step-by-step instructions for deploying the application, including setting up servers, configuring databases, deploying code, and managing dependencies.

- **Monitoring and Maintenance**

- ✓ **Monitoring:** Description of monitoring tools and practices used to monitor the health, performance, and security of the application and IoT devices.
- ✓ **Maintenance:** Guidelines for maintaining and updating the application, including patch management, version control, and backup procedures.

- **Troubleshooting**

- ✓ **Common Issues:** Description of common issues and troubleshooting steps for resolving them, such as connectivity issues with IoT devices, server errors, or performance bottlenecks.
- ✓ **Logging:** Explanation of logging mechanisms used in the application to track errors, warnings, and informational messages.

- **Appendix**

- ✓ **Glossary**: Definitions of key terms and concepts used throughout the documentation.
- ✓ **References:** Links to relevant resources, documentation, APIs, and libraries used in the development of the application.

Creating thorough documentation for an IoT web application is essential for facilitating development, deployment, and maintenance tasks, as well as for onboarding new team members and supporting end-users.

- **User guide documentation**

Creating user guide documentation for an IoT web application involves providing users with comprehensive instructions and guidelines on how to use the application effectively. Here's a suggested outline for a user guide documentation:

- ✓ **Introduction**

- **Welcome Message:** A brief welcome message introducing users to the user guide and the IoT web application.
- **Purpose:** Explanation of the purpose of the user guide and how it can help

users navigate and utilize the application efficiently.

✓ **Getting Started**

- **System Requirements:** Information on the minimum system requirements needed to access and use the application, including supported web browsers and devices.
- **Account Creation:** Step-by-step instructions on how to create a user account, including registration, account verification, and password setup.

- **Navigating the Interface**

✓ **Overview of the Interface:** Description of the main components of the user interface, including menus, navigation bars, and dashboard elements.
✓ **User Roles:** Explanation of different user roles within the application (e.g., administrators, regular users) and their permissions.

- **Managing IoT Devices**

✓ **Adding Devices:** Instructions on how to add IoT devices to the application, including device registration, pairing, and configuration.
✓ **Device Management:** Overview of device management features, such as viewing device status, renaming devices, and updating device settings.

- **Data Visualization**

✓ **Dashboard Overview**: Explanation of the dashboard interface and its components, including widgets, charts, and graphs.
✓ **Viewing Data:** Instructions on how to view real-time data from IoT devices, including temperature, humidity, and other sensor readings.
✓ **Customizing Views:** Guidance on customizing dashboard views, such as adding or removing widgets, resizing elements, and rearranging layouts.

- **Alerts and Notifications**

✓ **Setting Up Alerts:** Instructions on how to set up alerts and notifications for specific events or thresholds, such as temperature exceeding a certain value.
✓ **Managing Alerts:** Explanation of how to manage and respond to alerts, including acknowledging alerts, dismissing notifications, and adjusting alert settings.

- **Automation and Rules**

- ✓ **Creating Automation Rules:** Step-by-step instructions on how to create automation rules to automate actions based on specific conditions or triggers.
- ✓ **Managing Rules:** Overview of rule management features, including enabling, disabling, editing, and deleting automation rules.

- **Account Settings**

- ✓ **Profile Management**: Guidance on managing user profiles, including updating personal information, changing passwords, and managing notification preferences.
- ✓ **Account Security:** Tips for maintaining account security, such as enabling two-factor authentication and reviewing login activity.

- **Troubleshooting and Support**

- ✓ **Common Issues**: Description of common issues users may encounter and troubleshooting steps to resolve them, such as connectivity issues with devices or errors in data visualization.
- ✓ **Support Resources:** Information on where users can find additional help and support, including FAQs, documentation, and contact information for technical support.

Conclusion

Summary: Recap of key points covered in the user guide and encouragement for users to explore and utilize the application effectively.

Creating a user guide documentation that is clear, concise, and user-friendly can help users navigate the IoT web application with confidence and make the most of its features and capabilities.

Report on work done

Creating a report on work done is essential for tracking progress, communicating accomplishments, and sharing insights with stakeholders. Here's a suggested outline for a work done report:

Introduction

Overview: Brief overview of the report, including the reporting period and the purpose of the report.

Objective: Explanation of the objectives and goals accomplished during the

reporting period.

Project Overview

Project Name: Name of the project or initiative being reported on.

Project Description: Description of the project scope, objectives, and key deliverables.

Work Accomplished

Tasks Completed: List of tasks, activities, or milestones achieved during the reporting period.

Progress Summary: Summary of progress made towards project goals and objectives.

Challenges Faced: Description of any challenges, obstacles, or issues encountered during the work period and how they were addressed.

Key Achievements

Major Milestones: Highlight significant milestones or achievements reached during the reporting period.

Deliverables Completed: Description of key deliverables completed, such as reports, prototypes, or code releases.

Metrics Achieved: Quantitative data or metrics demonstrating progress and success, such as project milestones met, deadlines achieved, or performance indicators reached.

Tasks in Progress

Ongoing Work: Description of tasks, activities, or projects that are currently in progress and their status.

Future Plans: Outline of planned activities, tasks, or objectives for the next reporting period.

Contributions and Collaborations

Team Contributions: Recognition of team members' contributions and achievements during the reporting period.

Collaborations: Description of collaborations with other teams, departments, or

stakeholders and their impact on project outcomes.

Lessons Learned

Success Factors: Identification of factors contributing to success or positive outcomes.

Lessons Learned: Reflection on lessons learned, challenges faced, and areas for improvement.

Recommendations

Actionable Insights: Recommendations for future actions, strategies, or improvements based on lessons learned and experiences gained.

Opportunities Identified: Identification of opportunities for growth, innovation, or optimization in future work.

Conclusion

Summary: Recap of key points covered in the report.

Appreciation: Expression of gratitude to team members, stakeholders, or contributors for their efforts and support.

Attachments

Additional Documentation: Optional attachments, such as charts, graphs, or supplementary reports, to provide further context or detail.

By following this outline, you can create a comprehensive and informative report on work done that effectively communicates progress, achievements, challenges, and recommendations to stakeholders.

Provide Bills of quantities

Creating Bills of Quantities (BoQ) involves detailing the quantities of materials, labor, equipment, and other resources required for the construction or implementation of a project. Here's a general outline of how you might structure a BoQ:

Introduction

Project Information: Name of the project, location, client name, project manager, and any other relevant details.

BoQ Reference Number: A unique identifier for the BoQ document.

Summary of Work

Overview: Brief description of the scope of work covered by the BoQ.

Total Quantities: Summary of the total quantities of materials, labor, and other resources included in the BoQ.

Sections

Breakdown of the BoQ into sections based on the different aspects of the project (e.g., civil works, electrical works, plumbing works, etc.).

Each section should include a detailed list of items and quantities required for that specific aspect of the project.

Item Details

Item Code/ID: A unique identifier for each item in the BoQ.

Description: Description of the item, including specifications, materials, and any other relevant details.

Quantity: The quantity of each item required for the project, usually measured in standard units (e.g., meters, kilograms, cubic meters).

Unit: The unit of measurement for the quantity (e.g., meters, kilograms, pieces).

Unit Rate: The cost per unit of the item, if applicable.

Total Cost: The total cost of each item, calculated by multiplying the quantity by the unit rate.

Labor

Breakdown of labor requirements, including the types of labor (e.g., skilled, unskilled) and the number of workers required for each task.

Hourly Rates: Rates for different types of labor, if applicable.

Total Labor Costs: Calculation of the total cost of labor based on the number of workers and hourly rates.

Equipment

List of equipment required for the project, including machinery, tools, and vehicles.

Rental Rates: Rates for renting or leasing equipment, if applicable.

Total Equipment Costs: Calculation of the total cost of equipment based on rental rates and duration of use.

Miscellaneous Costs

Other costs not covered by materials, labor, or equipment, such as transportation, permits, and overhead expenses.

Breakdown of miscellaneous costs and their associated expenses.

Total Cost Estimate

Summation of all costs, including materials, labor, equipment, and miscellaneous expenses.

Grand Total: Total estimated cost for the entire project.

Practical Activity 3.3.2: Document IoT web application

Task: 1

1: Read key reading 3.3.2.
2: Referring to key reading 3.3.2., You are requested to go to the computer lab where trainees to make documentation of IoT Web Application.
3: Present your work to the trainer and whole class
4: Ask clarification where necessary.

Key readings 3.3.2: Document IoT web application

Creating documentation for an IoT web application that integrates API endpoints with a user interface (UI) is essential for ensuring that developers, users, and stakeholders understand how to use and interact with the application.:

Step 1: Introduction

- Overview of the Application: Provide a brief description of the IoT web application, its purpose, and its key features.

- Target Audience: Specify who the documentation is intended for (e.g., developers, users, stakeholders).

Step 2: Prerequisites

- Technical Requirements: List any technical requirements needed to run the application, such as hardware, software, and network configurations.

- Access to APIs: Provide information on how to obtain API keys or access credentials if necessary.

Step 3: System Architecture

- Architecture Diagram: Include a diagram that illustrates the overall architecture of the application, showing how the IoT devices, APIs, and front-end interface interact.

- Component Descriptions: Briefly describe each component in the architecture, including IoT devices, servers, databases, and the user interface.

Step 4: API Endpoints Documentation

- Endpoint Overview: List all the API endpoints used in the application, along with a brief description of their functionality.

- Endpoint Details: For each endpoint, provide the following information:

- URL: The endpoint URL.

- HTTP Method: The method used (GET, POST, PUT, DELETE).

- Parameters: List required and optional parameters, including their types and descriptions.

- Request Example: Provide an example of a request, including headers and body if applicable.

- Response Format: Describe the structure of the response, including status codes and example response data.

- Error Handling: Document common errors and their meanings.

Step 5: User Interface Integration

- UI Components: List the UI components that interact with the API data (e.g., dashboards, charts, forms).

- Data Flow: Explain how data flows between the API and the UI, including:

- How API calls are made from the UI.

- How data is processed and rendered in the UI.

- User interactions that trigger API calls.

Step 6: Setup and Installation

- Installation Instructions: Provide step-by-step instructions for setting up the application, including cloning the repository, installing dependencies, and configuring environment variables.

- Running the Application: Explain how to start the application and access the UI.

Step 7: Usage Instructions

- User Guide: Offer a guide on how to use the application, including common tasks and features.

- Examples: Provide examples of how users can interact with the application and what results they can expect.

Step 8: Troubleshooting

- Common Issues: List common issues users may encounter and provide solutions or workarounds.

- FAQs: Include a frequently asked questions section to address common concerns.

Step 9: Maintenance and Updates

- Versioning: Explain how versioning is handled for the API and the application.

- Updating the Application: Provide guidelines for updating the application and its dependencies.

Step 10: Conclusion

- Summary: Summarize the key points covered in the documentation.

- Contact Information: Provide contact details for support or further inquiries.

Step 11: Appendices

- Glossary: Include a glossary of terms and acronyms used in the documentation.

- References: List any additional resources, such as links to relevant articles, tutorials, or official documentation.

Step 12: Review and Feedback

- Feedback Mechanism: Encourage users to provide feedback on the documentation for continuous improvement.

- Version Control: Keep track of changes made to the documentation over time.

Points to Remember

- In IoT web application User Guide Documentation is to Create detailed user manuals or guides that explain how to set up, use, and troubleshoot the IoT web application. This should include step-by-step instructions, screenshots, and any technical requirements
- To do Report on Work Done You Should Prepare a comprehensive report documenting the development process of the IoT web application. It should include the project scope, milestones achieved, challenges encountered, and solutions implemented
- The things that Provide Bills of Quantities (BoQ) required hardware, software, and other materials used in the development and deployment of the IoT system, along with their costs and quantities.

Step 1: Introduction

Step 2: Prerequisites

Step 3: System Architecture

Step 4: API Endpoints Documentation

Step 5: User Interface Integration

Step 6: Setup and Installation

Step 7: Usage Instructions

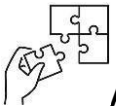
Step 8: Troubleshooting

Step 9: Maintenance and Updates

Step 10: Conclusion

Step 11: Appendices

Step 12: Review and Feedback



Application of learning 3.3.

A student buys a new smart home device (like a smart light bulb or thermostat) and is responsible for creating a user guide for their family. You are required to Set Up their Smart Home Device so, you are able to Write a simple user guide to help others set up and control the device through the app, Documenting the steps taken during setup, noting any issues and solutions as a report on work done and estimating the cost of all

devices and accessories involved in the setup, like Wi-Fi extenders or smart plugs, as a basic Bills of Quantities

Learning outcome 3 end assessment

A. Multiple Choice Questions (MCQs):

1. Which HTTP method is used to retrieve data from a server?
    a. POST
    b. PUT
    c. GET
    d. DELETE
2. What does an API response typically contain?
    a. Status Code, Headers
    b. Status, Data, Error
    c. Request Body, Query Parameters
    d. URL, Endpoint
3. Which HTTP method is used to update an existing resource?
    a. A) GET
    b. B) POST
    c. C) DELETE
    d. D) PUT
4. What does the status code "404" represent in an API response?
    a. A) Resource not found
    b. B) Server error
    c. C) Unauthorized access
    d. D) OK
5. Which part of an API request contains data to be sent to the server?
    a. A) Headers
    b. B) Body
    c. C) URL
    d. D) Endpoint
6. What is the purpose of an API key?
    a. A) It defines the structure of the response
    b. B) It authenticates requests made to the API
    c. C) It sets the timeout for a request
    d. D) It specifies the data type of the response
7. Which method is used to delete a resource from the server?
    a. A) PATCH
    b. B) GET
    c. C) POST
    d. D) DELETE

B. Read the following statement related to IoT Web application using PHP, and answer by True if the statement is correct or False if the statement is wrong

1. A DELETE request is used to create new resources.
2. API keys are a form of API authentication.
3. A PATCH request is used to partially update an existing resource.
4. An API with status code 200 means the request has failed.
5. API endpoints usually require authentication to secure access to data.
6. In a REST API, POST is the preferred method for retrieving data.
7. The HTTP DELETE method is used to modify an existing resource.

C. Read the following statement related to IoT Web application using PHP and write the letter of description that corresponding to the correct http method:

Answer	http method	Description
.....	1. GET	a) Retrieves data from the server
.....	2. POST	b) Creates new resources
.....	3. PUT	c) Updates an existing resource
.....	4. DELETE	d) Deletes a resource

D. Open-Ended Questions

1. Explain the steps to integrate API data into a user interface, and how you would handle errors during the process
2. Explain the role of authentication in securing API endpoints. How would you implement API key authentication in a project?
3. Describe the steps for handling API errors in a user interface. How would you display different error messages to users based on the error codes (e.g., 400, 401, 404, 500)? 500: "Server error. Please try again later."
4. Walk through the process of making a GET request in a user interface. Include all necessary steps such as setting up the endpoint, sending the request, handling the response, and rendering the data on the page.
5. Explain how you would implement pagination and infinite scrolling when rendering data retrieved from an API. What would be the key considerations to ensure a smooth user experience?

6. Discuss the importance of testing and debugging API requests. What tools would you use, and how would you identify and resolve issues in both request and response handling?

Practical assessment

QUESTION: Building a Weather Application

You are tasked with developing a simple weather application that fetches current weather data from a public weather API. The application will display weather information based on the user's location and allow users to search for weather data by city. Your application should handle API requests, render the data on the user interface, and provide appropriate error messages. You will also document the process and provide a report on your work.

Steps to Complete the Task:

1. Interpret API Endpoints:

- Research a public weather API (e.g., OpenWeatherMap, WeatherAPI).
- Identify the endpoint to retrieve current weather data by city.
- Check the required HTTP method (GET), URL structure, request parameters (e.g., city name, API key), and response format.

2. Implement API Data Fetching:

- Create a user interface with a search bar for entering the city name.
- Implement the HTTP GET request to fetch weather data when the user submits a city name.
- Ensure that the request includes necessary headers (e.g., API key) and parameters.

3. Render Data on User Interface:

- Display the retrieved weather data, such as temperature, humidity, weather conditions, and an icon representing the weather.
- Implement loading states to show the user that data is being fetched.
- Handle potential errors (e.g., city not found) and display appropriate error messages.

4. Testing and Debugging:

- Test the application with valid and invalid city names to ensure it behaves as expected.

- Debug any issues related to API requests, data rendering, or error handling.

5. Documentation:

- Create a user guide explaining how to use the weather application.
- Write a report detailing the API integration process, including challenges faced and how they were overcome.
- Provide a Bill of Quantities (BoQ) outlining any costs associated with using the API or hosting the application.

END

References

Kumar, S. P. ((April 2018)). 2. S Praveen Kumar. (April 2018). IoT-based temperature and humidity monitoring system using Raspberry Pi ResearchGate

https://www.researchgate.net/publication/325172192_IoTbased_Temperature_and_Humidity_Monitoring_System_using_Raspberry_pi

Sinlae, Fried, et al. "Penggunaan Framework Laravel dalam Membangun Aplikasi Website Berbasis PHP." *Jurnal Siber Multi Disiplin* 2.2 (2024): 119-132.

Sinlae, Fried, et al. "Penggunaan Framework Laravel dalam Membangun Aplikasi Website Berbasis PHP." *Jurnal Siber Multi Disiplin* 2.2 (2024): 119-132.

https://documentation.mindsphere.io/MindSphere/howto/howto-integrate-api.html#:~:text=A%20user%20can%20develop%20backend,functional%20logic%20in%20multiple%20applications.

RTB | RWANDA TVET BOARD

October 2024