RQF LEVEL 3

SWDVF301

SOFTWARE DEVELOPMENT

Vue.JS Framework

TRAINEE'S MANUAL

October 2024

RWANDA TVET BOARD

VUE.JS FRAMEWORK

Korea International Cooperation Agency

TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

The competent development body of this manual is Rwanda TVET Board ©, reproduce with permission.

All rights reserved.

- This work has been produced initially with the Rwanda TVET Board with the support from KOICA through TQUM Project
- This work has copyright, but permission is given to all the Administrative and Academic Staff of the RTB and TVET Schools to make copies by photocopying or other duplicating processes for use at their own workplaces.
- This permission does not extend to making of copies for use outside the immediate environment for which they are made, nor making copies for hire or resale to third parties.
- The views expressed in this version of the work do not necessarily represent the views of RTB. The competent body does not give warranty nor accept any liability.
- RTB owns the copyright to the trainee and trainer's manuals. Training providers may reproduce these training manuals in part or in full for training purposes only. Acknowledgment of RTB copyright must be included on any reproductions. Any other use of the manuals must be referred to the RTB.

© Rwanda TVET Board

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 and trainee’s manuals for the TVET Certificate III in Software Development, specifically for the module **"SWDVF301: Vue.JS Framework."**

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

RTB | RWANDA TVET BOARD

Under Financial and Technical support of

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Usher Emmanuel

PRODUCTION TEAM

Authoring and Review

SEKABANZA Jean de la Paix

IRARORA Jonah

Validation

HARERIMANA Love

MINANI Gervais

KWIZERA Emmanuel

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
HAKIZAYEZU Adrien

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENTS

ACRONYMS

CDN: Content delivery network

CI/CD: Continuous Integration/Continuous Deployment

CLI: Command Line Interface

CMS: Content Management Systems

CSS: Cascading Style Sheets

GUI: Graphical user interface

HTML: Hypertext Markup Language

IDE: Integrated Development Environment

MVVM: Model-View-View Model

RTB: Rwanda TVET Board

SPA: Single Page Application

SWD: Software Development

TQUM Project: TVET Quality Management Project

VS Code: Visual Studio Code

INTRODUCTION

This trainer's manual includes all the methodologies required to effectively deliver the module titled **"Vue.JS Framework."** Trainees 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 trainer'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 trainer, you will begin by asking questions related to the activities to encourage critical thinking and guide trainees toward real-world applications in the labor market. The manual also outlines essential information such as learning hours, didactic materials, and suggested methodologies.

This manual outlines the procedures and methodologies for guiding trainees through various activities as detailed in their respective trainee manuals. The activities included in this training manual are designed to offer students opportunities for both individual and group work. Upon completing all activities, you will assist trainees in conducting a formative assessment known as the end learning outcome assessment. Ensure that students review the key reading and the points to remember section.

MODULE CODE AND TITLE: SWDVF301 DEVELOP SIMPLE GAME IN VUE FRAMEWORK

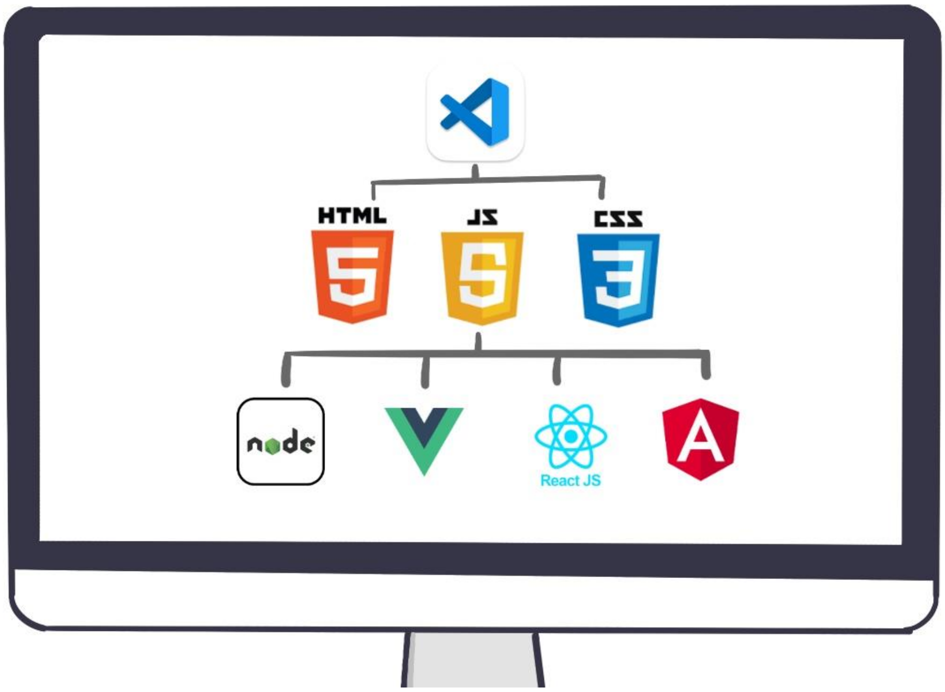
Learning Outcome 1: Set Up Environment

Learning Outcome 2: Apply Vue Framework

Learning Outcome 3: Plan game

Learning Outcome 4: Develop Game

Learning Outcome 1: Set Up Environment



Indicative contents

1.1 Description of key concepts

1.2 Vue project installation

1.3 Description of Vue project folder & files

Key Competencies for Learning Outcome 1: Set Up Environment

Knowledge	**Skills**	**Attitudes**
• Differentiate command line from Interface from IDE • Differentiate frontend from backend • Description of Javascript framework • Explanation of single page application • Explanation of dependencies and the Vue development environment. • Description of Vue project folder and files	• Installing NodeJS. • Using command line interface (cmd) • Configuring NPM • Installing of Vue framework • Testing Javascript file using NodeJs • Initialising Vue project using terminal • Running Vue project • Interpretating vue project folder and files	• Having Team work spirit ability • Being critical thinker • Being Innovative • Being attentive. • Being creative • Problem solving • Being Practical oriented • Being Detail oriented

Duration: 20 hrs

Learning outcome 1 objectives:

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

1. Describe correctly key concepts that are used in Vuejs framework.
2. Describe properly folders and files as used in Vuejs framework project
3. Explain correctly the term dependencies and single page application as applied in vue js framework.
4. Install properly Node js and Vuejs framework as used in single page application development.
5. Run properly Vuejs projects as applied in the software Development process.
6. Test properly Vuejs projects as applied in the software Development process.
7. Initialize correctly vue project as applied in frontend development
8. Describe correctly javascript frameworks as used in single application development.
9. Differentiate correctly frontend from backend as applied in software Development.
10. Interpret correctly vue project folder and files as used in project creation.
11. Configure properly NPM as used in installation of packages and libraries.

Resources

Equipment	**Tools**	**Materials**
• Computer	• Text editor (VSCode) • Node js • Web browser	• Internet

Indicative content 1.1: Description of Key Concepts

Duration: 6 hrs

Theoretical Activity 1.1.1: Explanation of the key concepts related to Vue js Framework.

Tasks:

1. You are requested to answer the following questions related to the key concepts in Vue Js framework:

   I. Differentiate Command Line Interface (CLI) from Integrated Development Environment (IDE).

   II. What are the key differences between backend and frontend development?

   III. Provide explanations for the following terms:

      a) Node.js and NPM

      b) Single Page Application (SPA)

      c) Dependencies

2. Provide the answer for the asked questions and write them on papers.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 1.1.1. And ask questions where necessary.

Key readings 1.1.1.: Explanation of the key concepts related to Vue js Framework.

CLI stands for Command Line Interface. It is a text-based interface that allows users to interact with a computer program or operating system by entering commands. Instead of using a graphical user interface (GUI) with windows, icons, and menus, a CLI relies on typed commands and text-based responses.

In the context of software development, many frameworks and tools provide a Command Line Interface to streamline various tasks.

Here are a few examples:

1. **Vue CLI:** Vue CLI is a command-line tool specifically designed for Vue.js applications. It helps developers scaffold new projects, manage dependencies, run development servers, build production-ready bundles, and more.

2. **Angular CLI:** Angular CLI is a command-line interface for developing Angular applications. It offers a set of commands to generate components, services, modules, and other Angular-specific entities. It also provides features for testing, building, and serving Angular projects.

3. **Create React App:** Create React App is a popular command-line tool used for quickly setting up React applications. It sets up a development environment with all the necessary configurations and dependencies, allowing developers to start building React apps without manually configuring build tools.

4. **Git cmd:** Git, a widely used version control system, is primarily operated through the command line. Developers use commands like git init, git add, git commit, and git push to manage and collaborate on code repositories.

Basic commands

cd: Changes the current directory.

dir: Lists files and directories in the current directory.

mkdir: Creates a new directory.

rmdir: Removes an empty directory.

del: Deletes a file.

copy: Copies files and directories.

move: Moves files and directories.

ren: Renames a file or directory

Using a CLI provides several benefits for developers, such as:

1. Efficiency: CLI tools often offer shortcuts and automation, allowing developers to perform repetitive tasks more efficiently.

2. Scripting and Automation: CLI commands can be scripted and combined to automate complex workflows and build processes.

3. Flexibility: Command-line interfaces are generally platform-independent and can be used on various operating systems, making them accessible to a wide range of developers.

4. Control and Precision: CLI tools provide fine-grained control over configurations and options, allowing developers to customize their environment and workflows to their specific needs.

IDE

IDE stands for Integrated Development Environment. It is a software application that provides comprehensive tools and features to facilitate software development. An IDE typically combines a code editor, build automation tools, debugging capabilities, and other features into a single integrated package, providing developers with a unified environment for writing, testing, and deploying their code.

Here are some key components and features commonly found in IDEs:

1. Code Editor: IDEs include a sophisticated code editor that offers features like syntax highlighting, code completion, code navigation, and formatting. These features help developers write code faster, catch errors, and improve code readability.

2. Build Tools: IDEs often integrate build tools such as compilers, interpreters, and build automation systems. These tools help in compiling, executing, and packaging the code into a distributable form.

3. Debugging: IDEs provide debugging capabilities, allowing developers to set breakpoints, inspect variables, step through code execution, and analyze runtime behavior. This helps in identifying and fixing issues in the code.

4. Version Control Integration: Many IDEs offer built-in support for version control systems like Git, allowing developers to manage code repositories, commit changes, and collaborate with others directly from the IDE.

5. Integrated Terminal: IDEs often include an integrated terminal that allows developers to run command-line tools, execute scripts, and interact with the operating system without leaving the IDE.

6. Project Management: IDEs provide tools for managing projects, organizing files and directories, and handling dependencies. They may offer features like project templates, file navigation, and project-wide search.

7. Code Refactoring: IDEs offer automated code refactoring capabilities, enabling developers to safely modify their code structure and improve its quality without introducing bugs. Refactoring tools can rename variables, extract methods, and perform other code transformations.

8. Integration with External Tools: IDEs can integrate with various external tools and frameworks, such as testing frameworks, code analysis tools, and deployment platforms. This integration streamlines the development workflow and allows developers to seamlessly work with third-party tools.

Some popular IDEs used for different programming languages and frameworks include:

●**Visual Studio Code:** A highly popular and extensible IDE developed by Microsoft, known for its wide language support and rich ecosystem of extensions.

●**IntelliJ IDEA**: A powerful IDE primarily used for Java development, but also supports other languages like Kotlin, Groovy, and Scala.

●**PyCharm:** An IDE specifically designed for Python development, offering advanced features for coding, debugging, and testing Python applications.

●**Eclipse:** A widely used IDE known for its extensive support for Java development, but it also supports other languages through plugins.

●**Xcode:** Apple's integrated development environment for macOS and iOS development, offering tools and features tailored for building applications for Apple platforms.

Frontend and backend

Frontend and backend are two distinct parts of a web application that work together to deliver a complete user experience. Here's an overview of each:

Frontend: The frontend refers to the client-side of a web application, which is what the users interact with directly in their web browsers. It encompasses the presentation layer and user interface (UI). The frontend technologies are responsible for creating an engaging and user-friendly experience.

Some key aspects of frontend development include:

1. HTML (Hypertext Markup Language): It defines the structure and content of web pages.

2. CSS (Cascading Style Sheets): It handles the visual styling of HTML elements, such as layout, colors, typography, and animations.

3. JavaScript: It enables interactivity and dynamic behaviour on web pages. It allows developers to manipulate the DOM, handle user events, make AJAX requests, and build rich user interfaces.

4. Frontend Frameworks and Libraries: Frameworks like React, Angular, or Vue.js provide tools and abstractions to simplify frontend development, manage state, and build reusable components.

5. Responsive Design: Frontend developers strive to create websites that adapt to different screen sizes and devices, ensuring a consistent experience across desktop, tablets, and mobile devices.

6. Browser Compatibility: Frontend developers must consider cross-browser compatibility to ensure their websites work well across different web browsers like Chrome, Firefox, Safari, and Edge.

Backend: The backend, also known as the server-side, refers to the behind-the-scenes operations that power a web application. It handles the logic, data processing, and storage. Backend technologies enable the communication between the frontend and various external resources, such as databases, APIs, and file systems.

Some key aspects of backend development include:

1. Server-Side Programming Languages: Popular backend programming languages include Python, Java, JavaScript (with Node.js), Ruby, and PHP. These languages handle the server-side logic and data processing.

2. Web Frameworks: Backend frameworks, such as Django (Python), Ruby on Rails (Ruby), Express.js (Node.js), and Laravel (PHP), provide pre-built tools and structures to handle common backend tasks, like routing, database integration, and authentication.

3. Databases: Backend developers work with databases like MySQL, PostgreSQL, MongoDB, or Redis to store and retrieve data required by the application.

4. APIs (Application Programming Interfaces): Backend developers design and build APIs that expose data and functionality to the frontend or other external services. APIs allow different applications to communicate and exchange data in a standardized way.

5. Security: Backend developers handle security concerns such as user authentication, data validation, encryption, and protection against common web vulnerabilities like cross-site scripting (XSS) or SQL injection.

6. Scaling and Performance: Backend developers need to ensure that the server-side infrastructure can handle increased traffic and perform efficiently as the user base grows. Techniques like load balancing, caching, and optimizing database queries are commonly used.

A Single Page Application

A Single Page Application (SPA) is a web application architecture where the entire application runs within a single HTML page. Unlike traditional multi-page applications, SPAs do not require page reloads or navigation to different pages for every user action. Instead, SPAs dynamically update the content on the page, providing a more fluid and responsive user experience.

Dependencies

In Vue.js, dependencies refer to the external libraries, packages, or modules that are required by your Vue project to add specific functionality or features. Dependencies are managed through a package manager, such as npm (Node Package Manager) or Yarn, which allows you to easily install, update, and remove dependencies.

There are two types of dependencies in Vue.js:

Runtime Dependencies: These are the dependencies required for your Vue.js application to run properly in the browser. They include the Vue.js framework itself and other runtime-specific libraries.

Development Dependencies: These are the dependencies that are necessary during the development process, but are not required in the final production build of your application. They include tools, testing frameworks, bundlers, and other development-specific libraries.

Theoretical Activity 1.1.2: Explanation of environment in Vue js Framework

Tasks:

1. You are requested to answer the following questions related to the environments in Vue Js Framework:

   I. How could you describe the following environments?

   a) Development environment
   b) Testing environment
   c) Production environment

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 1.1.2. Ask questions where necessary.

Key readings 1.1.2.: Explanation of environment in Vue js Framework

The development environment for Vue.js involves the setup of tools and configurations that enable developers to efficiently develop, debug, and test Vue.js applications.

Here are the key components of a Vue.js development environment:

Node.js: Vue.js development typically requires Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine. It provides the environment to execute JavaScript code outside the browser and is essential for running build tools, package managers, and development servers.

Package Manager: A package manager, such as npm (Node Package Manager) or Yarn, is used to manage project dependencies and install necessary libraries. It allows you to easily add, update, and remove packages required for your Vue.js project.

Vue CLI: Vue CLI (Command Line Interface) is a development tool specifically designed for Vue.js projects. It provides a standardized project structure, build configurations, and a set of helpful commands to scaffold, develop, and build Vue.js applications. Vue CLI also supports features like Hot Module Replacement (HMR) for instant code updates during development.

Development Server: Vue CLI includes a development server that allows you to run your Vue.js application locally during development. The server provides automatic page reloading or Hot Module Replacement (HMR), which updates only the modified components or modules without reloading the entire page.

Code Editor: A code editor is essential for writing Vue.js code. Popular options include Visual Studio Code, Sublime Text, Atom, or Web Storm. Code editors often provide features like syntax highlighting, code formatting, linting, and code suggestions that enhance the development experience.

Vue Devtools: Vue Devtools is a browser extension that enhances Vue.js development by providing a set of developer tools for debugging and inspecting Vue components.

It allows you to inspect component hierarchy, view component data and props, monitor component performance, and modify component state in real-time.

Browser Developer Tools: Modern browsers, such as Google Chrome or Firefox, come with built-in developer tools that provide a range of features for debugging, inspecting the DOM, monitoring network requests, and profiling JavaScript code.

These tools are essential for troubleshooting and optimizing Vue.js applications.

Testing Tools: Vue.js provides tools and libraries for testing your Vue components and application. Popular options include Jest, Mocha, Karma, and Vue Test Utility.

These tools enable unit testing, component testing, and end-to-end testing of your Vue.js application.

Build Tools: Vue.js applications often require build tools like webpack or Vue CLI service to bundle and optimise the application's code, assets, and dependencies for production. These tools handle tasks like code transpilation, minification, CSS preprocessing, and asset optimization.

Production environment

In Vue.js, the production environment refers to the setup and configuration necessary for deploying and running your Vue.js application in a production environment. The production environment is optimized for performance, security, and scalability.

Here are the key components and considerations for a Vue.js production environment:

Build Process: Before deploying a Vue.js application to production, it needs to be built for optimal performance and reduced file sizes. Vue CLI provides build configurations and tools that bundle and optimize your application's code, assets, and dependencies.

Static Asset Hosting: To serve your Vue.js application in a production environment, you need a web server capable of hosting static assets. Vue.js applications are typically static files (HTML, CSS, JavaScript, and assets) that can be served by any web server, such as Apache, Nginx, or a content delivery network (CDN). You can configure the web server to handle requests and serve the built files from the different directory.

Application Deployment: To deploy your Vue.js application to a production environment, you need to transfer the built files to the hosting server. The deployment process may vary depending on your hosting provider or infrastructure. It can involve using FTP, SSH, Git, or CI/CD pipelines to transfer files to the server and ensure the latest version of the application is deployed.

Security Considerations: In a production environment, it's crucial to consider security best practices to protect your Vue.js application and its data. This includes securing your web server with SSL/TLS certificates to enable HTTPS, implementing authentication and authorization mechanisms, validating user input, protecting against cross-site scripting (XSS) and other common vulnerabilities, and following security guidelines for third-party libraries and dependencies.

Performance Optimization: Optimizing the performance of your Vue.js application in a production environment is essential to deliver a fast and smooth user experience. Techniques like code splitting, lazy loading, caching, compression, and using a content delivery network (CDN) for static assets can significantly improve performance. Consider leveraging Vue.js features like asynchronous component loading and dynamic imports to optimize the loading and rendering of your application.

Monitoring and Error Tracking: Monitoring your Vue.js application in production helps identify performance issues, errors, and user behaviour. Tools like Google Analytics, Sentry, or New Relic can be integrated into your Vue.js application to collect data, track errors, and monitor application performance. These tools provide insights into user interactions, performance metrics, and error reports, enabling you to detect and address issues promptly.

Continuous Deployment: To ensure a smooth deployment process and quick iteration, you can leverage continuous deployment practices. By integrating your Vue.js application with a continuous integration/continuous deployment (CI/CD) pipeline, you can automate the build, testing, and deployment processes. CI/CD tools like Jenkins, Travis CI, or GitLab CI/CD enable you to automate the steps from code changes to production deployment, ensuring consistency and reducing manual errors.

Theoretical Activity 1.1.3: Introduction to Vue JS Framework

Tasks:

1. You are requested to discuss on the following as related to introduction to vue js framework:

I. History of Vue.js Framework
II. Advantages of Vue.js Framework
III. Purpose of Vue.js Framework
IV. Requirements for Learning Vue.js Framework
V. Specific areas where Vue.js can be effectively utilized.

2. Write your findings on papers or flipcharts.

3. Present the findings to the whole class

4. For more clarification, read the key readings 1.1.2. Ask questions where necessary.

Key readings 1.1.3.: Introduction to Vue JS Framework

A. History of Vue.js Framework

Vue.js was created by **Evan You** and first released in February 2014. It was developed to address the need for a more flexible and efficient front-end framework that could compete with Angular and React. Over the years, Vue.js has gained popularity for its simplicity, ease of integration, and robust performance, becoming one of the most widely used JavaScript frameworks.

B. Advantages of Vue.js Framework

Easy to Learn and Use: Vue.js has a gentle learning curve, making it accessible for beginners while still offering advanced features for more experienced developers.

Versatility: It can be used for building single-page applications (SPAs), complex web interfaces, and even native mobile apps with frameworks like NativeScript.

Component-Based: Vue.js follows a component-based architecture similar to React, making it easy to reuse components and manage complex UIs.

Flexibility: Vue.js can be integrated into existing projects seamlessly, allowing incremental adoption. It also supports building large-scale applications efficiently.

Performance: Vue.js is lightweight and fast, with optimized rendering and minimal overhead.

Official Tooling: Vue.js provides official libraries and tools (like Vue Router for routing and Vuex for state management) that work seamlessly with the framework.

Active Community: Vue.js has a large and active community, contributing to its growth, support, and availability of plugins and extensions.

C. Purpose of Vue.js Framework

Vue.js is designed to simplify the development of interactive web interfaces. Its primary purposes include:

Building User Interfaces: Vue.js allows developers to create dynamic and responsive user interfaces efficiently.

SPA Development: It excels in building single-page applications where fast rendering and seamless user interactions are crucial.

Component Reusability: Vue.js promotes modular development through components, enhancing code maintainability and reusability.

Progressive Framework: Vue.js can be incrementally adopted into existing projects, making it suitable for both small and large-scale applications.

D. Requirements for Learning Vue.js Framework

To effectively learn Vue.js, you should be familiar with:

HTML/CSS: Basic knowledge of HTML for markup and CSS for styling is essential.

JavaScript (ES6+): Understanding of JavaScript fundamentals, especially ES6 features like arrow functions, classes, and modules.

Basic Understanding of MVVM Architecture: Vue.js follows the Model-View-View Model (MVVM) pattern, so familiarity with this architecture is beneficial.

Node.js and npm: Basic understanding of Node.js and npm (Node Package Manager) for installing and managing Vue.js and its dependencies.

Asynchronous Programming: Knowledge of asynchronous programming concepts, Promises, and async/await syntax is helpful when dealing with Vue.js components that fetch data asynchronously.

E. Specific areas where Vue.js can be effectively utilized.

1. Single-Page Applications (SPAs)

Vue.js is ideal for building SPAs where dynamic content is loaded without refreshing the entire page, providing a smooth user experience.

Examples: Social networks, dashboards, and project management tools.

2. Progressive Web Apps (PWAs)

Vue.js can be used to develop PWAs that offer a native app-like experience in the browser, with offline support and fast load times.

Examples: E-commerce platforms, news websites, and blogs.

3. Dynamic User Interfaces

Vue.js is great for creating dynamic and responsive user interfaces with components that can be easily reused and customized.

Examples: Interactive forms, modals, and widgets.

4. **Content Management Systems (CMS)**

   Vue.js can be used to build the front-end of content management systems, providing a seamless content editing and management experience.

   Examples: Custom CMS platforms, headless CMS front-ends.

5. **E-commerce Websites**

   Vue.js can power the front-end of e-commerce websites, handling product displays, user authentication, and payment integration.

   Examples: Online stores, product catalogs, and shopping carts.

6. **Enterprise Applications**

   Vue.js can be used to develop complex enterprise-level applications that require robust user interfaces and integration with backend systems.

   Examples: CRM systems, ERP systems, and HR management tools.

7. **Real-Time Applications**

   With its reactivity and integration capabilities, Vue.js is suitable for developing real-time applications that require instant updates and interactions.

   Examples: Chat applications, live collaboration tools, and stock trading platforms.

8. **Mobile Applications**

   Vue.js can be used to build mobile applications through frameworks like NativeScript-Vue, enabling the development of native mobile apps using Vue components.

   Examples: Mobile apps using frameworks like NativeScript or Ionic.

9. **Data-Driven Dashboards**

   Vue.js is effective for creating data-driven dashboards with real-time data visualization, charts, and graphs.

   Examples: Analytical tools, business intelligence platforms, and monitoring systems.

10. **Interactive Learning Platforms**

    Vue.js can be used to develop interactive learning platforms that provide engaging and interactive experiences for users.

    Examples: Online courses, quiz systems, and educational games.

11. Blogs and Portfolio Websites

Vue.js can power the front-end of blogs and portfolio websites, offering smooth navigation and interactive features.

Examples: Personal blogs, artist portfolios, and resume websites.

12. Games

Vue.js can be used to develop simple browser-based games or integrate game mechanics into web applications.

Examples: Browser-based games, educational games.

13. Cross-Platform Desktop Applications

Vue.js can be combined with Electron to develop cross-platform desktop applications, allowing developers to use web technologies for desktop software.

Examples: Productivity tools, note-taking apps.

14. Custom Web Components

Vue.js can be used to create custom web components that can be reused across different projects or integrated into other frameworks.

Examples: Custom UI libraries, reusable component sets.

Practical Activity 1.1.4: Use command prompt (CMD)

Task:

1. Read key reading 1.1.3 and ask clarification where necessary
2. Referring to the previous theoretical activities (1.1.1) you are requested to go to the computer lab to use command prompt for creating and managing folders. This task should be done individually.
3. Apply safety precautions
4. Present out the steps to use the command prompt.
5. Referring to the steps provided on task 3, use command prompt
6. Present your work to the trainer and whole class

Key readings 1.1.4.: Use command prompt (CMD)

Using the Command Prompt in Windows allows you to execute various commands to interact with the operating system. Here are the steps to use the Command Prompt:

1. **Open Command Prompt:**

- Press `Win + R` on your keyboard to open the "Run" dialog.

- Type `cmd` and press Enter to open the Command Prompt window.

2. **Navigate Folders:**

- Use the `cd` command to change directories. For example, `cd Desktop` will move you to the Desktop directory.

3. **List Files and Folders:**

- Use the `dir` command to list files and folders in the current directory.

4. **Run Programs:**

- You can run programs by typing their name and pressing Enter. For example, type `notepad` and press Enter to open Notepad.

5.Execute Commands:

- Type commands and press Enter to execute them. For example, `ipconfig` will display network configuration information.

6. Create and Delete Files/Folders:

- Use commands like `mkdir` to create a new directory and `del` to delete files. For example, `mkdir NewFolder` creates a new folder named "NewFolder."

7. Copy and Move Files:

- Use commands like `copy` to copy files and `move` to move files. For example, `copy file.txt C:\DestinationFolder` copies "file.txt" to "DestinationFolder."

8. Terminate a Command:

- Press `Ctrl + C` to terminate a command that is currently running.

9. **Access Help:**

- Use the `help` command to get a list of available commands or `command /?` to get help for a specific command. For example, `dir /?` provides information about the `dir` command.

10. Close Command Prompt:

- Type `exit` and press Enter to close the Command Prompt window.

Create and access folder using command prompt in windows using mkdir and cd command

1.Press the Windows key + R to open the "Run" dialog box.

2.Type "cmd" or "cmd.exe" and press Enter.

3.The Command Prompt window will open, allowing you to type and execute commands.

4.Determine the location where you want to create the directory. And then perform the following activities:

•Create a directory mkdir directory name

•cd (Change Directory)

•dir [directory_path]: This will list the files and directories in the current directory or the specified directory.

•rmdir (Remove Directory)

•del [file_name]: Delete a file.

•copy [source_file_path] [destination_file_path] : Copy a file from the source path to the destination path.

Note: There are other tasks that can be performed by using command prompt such as delete,remove,copy depending on what you want to perform inside your directory.

Points to Remember

- In vue js framework you can create a project by using command line interface or integrated development environment.
- Front end of application deals with the client side or graphical user interface while the backend deals with server-side part considering the logic part and database.

- Once you install node js it come up with it default package manager (NPM) that you can use to install all dependencies and packages that will be used in development of single page application.
- Using the Command Prompt in Windows allows you to execute various commands to interact with the operating system. Here are the steps to use the Command Prompt:
    1. Open Command Prompt
    2. Navigate Folders
    3. List Files and Folders
    4. Run different commands depending on the task to be performed such as rename, delete and change directory.

Application of learning 1.1.

The KT-company is an IT company located at KICUKIRO District, that company develop softwares by using Vue js framework, as developer who is working there you are assigned task of creating and managing folders by using command prompt that will be used in saving the information of clients on desktop.

Indicative content 1.2: Vue Project Installation

Duration: 8 hrs

Theoretical Activity 1.2.1: Description of key terms used in Vue js framework

Tasks:

1. You are requested to answer the following questions related to the key terms used in Vue Js Framework:

    i. What do you understand about the term Framework?

    ii. List JavaScript Frameworks

    iii. Discuss the importance of Javascript frameworks

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 1.2.1. In addition, ask questions where necessary.

Key readings 1.2.1.: Description of key terms used in Vue js framework

Framework

In the context of software development, a framework is a collection of pre-written code, tools, and libraries that provide a structured and reusable foundation for building applications.

It offers a set of rules, conventions, and guidelines that developers can follow to streamline the development process.

Frameworks aim to simplify and accelerate the development of applications by providing a structured approach to common tasks and challenges. They often include pre-built components, modules, and libraries that handle common functionalities such as database interaction, user authentication, routing, and more. This allows developers to focus on implementing the specific business logic or unique features of their applications rather than dealing with low-level details.

Here are a few key characteristics of frameworks:

1. Abstraction: Frameworks abstract away complex implementation details and provide high-level APIs or interfaces that developers can use to interact with the underlying functionalities.

2. Reusability: Frameworks promote code reuse by providing pre-built components and libraries. Developers can leverage these components to avoid reinventing the wheel and speed up development.

3. Structure and conventions: Frameworks often enforce a specific structure and coding conventions, making it easier to organize and maintain code. They establish guidelines on how components should be structured, how data should be handled, and how interactions between components should be managed.

4. Extensibility: Frameworks are designed to be extended and customized according to the specific needs of an application. Developers can add or modify functionalities by extending or hooking into the framework's existing components.

5. Community and ecosystem: Frameworks typically have a large and active community of developers. This community contributes to the development of the framework, provides support, shares knowledge, and creates a vibrant ecosystem of plugins, extensions, and tools around the framework.

JavaScript is a versatile programming language that is widely used for web development. There are several popular JavaScript frameworks available that provide powerful tools and libraries to simplify and enhance the development process.

Here are some of the most Commonly used JavaScript frameworks:

1. React.js: React is a component-based JavaScript library developed by Facebook. It is used for building user interfaces, particularly for single-page applications. React follows a declarative approach, allowing developers to create reusable UI components that efficiently update and render based on changes in data.

2. Angular: Angular is a full-featured web application framework developed by Google. It provides a complete solution for building large-scale applications. Angular utilizes TypeScript, a superset of JavaScript, and follows a component-based architecture. It offers features like two-way data binding, dependency injection, and powerful templating capabilities.

3. Vue.js: Vue.js is a progressive JavaScript framework that is gaining popularity. It is designed to be approachable and easy to learn, making it suitable for both small and large-scale projects. Vue.js emphasizes simplicity and flexibility, allowing developers to incrementally adopt its features. It provides a virtual DOM, component-based architecture, and a reactive data-binding system.

4. Ember.js: Ember.js is a comprehensive framework for building ambitious web applications. It follows the convention-over-configuration principle and provides a robust set of tools and features. Ember.js offers a strong opinionated structure, an integrated build system, and powerful templating capabilities.

5. Backbone.js: Backbone.js is a lightweight framework that focuses on providing structure to web applications. It provides models, views, collections, and routers as building blocks for creating organized client-side JavaScript applications. Backbone.js is often used with other libraries and frameworks to build scalable applications.

6. Express.js: Express.js is a minimal and flexible web application framework for Node.js. It is primarily used for creating server-side applications and APIs. Express.js provides a simple and intuitive API for handling HTTP requests, routing, and middleware integration.

Note: These are just a few examples of JavaScript frameworks available, and each has its own strengths and use cases. The choice of framework depends on the specific requirements, project complexity, and personal preferences of the developers.

Benefit of Vue js framework

Vue.js offers several benefits that make it a popular choice among developers.

Here are some of the key advantages of using the Vue.js framework:

1. Ease of Learning: Vue.js has a gentle learning curve, making it accessible for beginners. Its syntax is intuitive and easy to understand, allowing developers to quickly grasp the core concepts and start building applications.

2. Flexibility: Vue.js provides a flexible and adaptable approach to development. It can be used for small, single-page applications or scaled up to build large, complex applications. Vue.js allows developers to incrementally adopt its features and integrate it into existing projects without any major disruptions.

3. Component-Based Architecture: Vue.js follows a component-based architecture, which promotes reusability and modular development. Components encapsulate HTML, CSS, and JavaScript logic, making it easier to maintain and debug code. Reusable components save development time and effort.

4. Reactive Data Binding: Vue.js uses a reactive data binding system, which allows developers to establish dynamic relationships between the data and the DOM.

When the data changes, the DOM updates automatically, eliminating the need for manual DOM manipulation. This makes it easier to keep the UI in sync with the underlying data.

5. Virtual DOM: Vue.js utilizes a virtual DOM (VDOM) to optimize rendering performance. It creates an in-memory representation of the real DOM and performs efficient updates only to the parts that have changed. This results in faster rendering and a smoother user experience.

6. Comprehensive Ecosystem: Vue.js has a thriving ecosystem with a wide range of libraries, tools, and plugins. It integrates well with other JavaScript libraries and frameworks, allowing developers to leverage additional functionality when needed. The ecosystem provides solutions for state management, routing, form validation, and more.

7. Developer-Friendly Features: Vue.js offers several features that enhance developer productivity. These include built-in tools like Vue Devtools for debugging and inspecting Vue.js applications, a CLI (Command Line Interface) for project scaffolding, and excellent documentation that covers all aspects of the framework.

8. Active Community and Support: Vue.js has a strong and active community of developers who contribute to its development and provide support through forums, online communities, and resources. The community-driven nature of Vue.js ensures continuous improvement, updates, and a wealth of learning materials.

These benefits make Vue.js a compelling choice for developers looking for a versatile and efficient framework to build web applications.

Practical Activity 1.2.2: Develop JavaScript program

Task:

1. Read key reading 1.2.2 and ask clarification where necessary
2. Referring to the previous theoretical activities (1.2.1.) you are requested to go to the computer lab to develop a Javascript program. This task should be done individually.
3. Apply safety precautions.
4. Present out the steps of developing a Javascript program.
5. Referring to the steps provided on task 3, Develop Javascript program
6. Present your work to the trainer and whole class

Key readings 1.2.2.: Develop JavaScript program

Developing a JavaScript program involves writing code that can be executed in a web browser or server environment. Here are the steps to develop a JavaScript program:

1. Set Up Your Development Environment:

Choose a code editor or Integrated Development Environment (IDE) to write your JavaScript code. Popular options include Visual Studio Code, Sublime Text, Atom, or WebStorm.

2. Create a New JavaScript File:

Open your code editor and create a new file with a `.js` extension. This file will contain your JavaScript code.

3. Write Your JavaScript Code:

Start writing your JavaScript code in the `.js` file. You can include functions, variables, loops, conditionals, and other JavaScript syntax to achieve the desired functionality.

5. Debug Your Code:

Use the browser's developer tools or a JavaScript debugger to identify and fix any errors in your code. Debugging tools can help you step through your code, set breakpoints, and inspect variables.

6. Test Your JavaScript Program:

Run your JavaScript program. Check for any errors in the browser console and ensure that your program functions as intended.

7. Refine and Optimize Your Code:

Refactor your code to improve readability, performance, and maintainability. Consider using best practices, design patterns, and libraries to enhance your JavaScript program.

Installation of node js

To install Node.js, you can follow these steps:

Step 1: Visit the official Node.js website: Go to the official Node.js website at https://nodejs.org/.

Step 2: Choose the appropriate version: Node.js offers two versions: LTS (Long-Term Support) and Current. The LTS version is recommended for most users as it provides stability and support for a longer duration. Select the LTS version unless you have a specific reason to choose the Current version.

Step 3: Download the installer: On the Node.js website, you'll find download buttons for different operating systems (e.g., Windows, macOS, Linux). Click on the button corresponding to your operating system to start the download.

Step 4: Run the installer: Once the installer is downloaded, locate the installer file and run it. The installation process will vary depending on your operating system.

for Windows: Double-click the downloaded .msi file and follow the installation wizard. Choose the default options unless you have specific preferences.

To Verify NodeJs is installed follow these steps:

Step 1: After the installation is complete, you can verify that Node.js is installed correctly by opening a command prompt or terminal window and typing the following command: **node -v**

This command will display the version of Node.js installed on your system. Similarly, you can run the following command to check the version of npm (Node Package Manager) that is installed:

npm -v

If both commands display the version numbers without any errors, Node.js is successfully installed on your system.

Congratulations! You have installed Node.js on your machine, and you're now ready to start developing applications using Node.js and its ecosystem.

Configure NPM

After installing Node.js, npm (Node Package Manager) is automatically installed along with it. However, there are a few initial configurations you can set up for npm.

Here are the steps to configure npm:

Step 1: Open a command prompt or terminal: Launch the command prompt

Step 2: Check npm version: To ensure npm is installed correctly, check its version by running the following command:

npm -v

This command will display the version of npm installed on your system.

Step 3: Update npm (optional): If you have an older version of npm installed and want to update it to the latest version, you can run the following command:

npm install -g npm

This command will update npm to the latest version available.

Step 4: Set up npm configuration (optional): You can configure npm by setting up your name and email address. This information is used when you publish packages to the npm registry.

Run the following commands, replacing "Your Name" and "your.email@example.com" with your actual name and email:

npm config set init.author.name "Your Name"

npm config set init.author.email "your.email@example.com"

Step 5: Change default package installation directory (optional): By default, npm instals packages in a global directory.

If you prefer to change the default directory to a custom location, you can set the prefix configuration. Run the following command, replacing "path/to/custom/directory" with the desired directory path:

npm config set prefix "path/to/custom/directory"

Step 6: Verify the configurations: You can verify the npm configurations by running the following command:

npm config list

This command will display the current configurations for npm, including the values you have set.

That's it! You have successfully configured npm with the desired settings. Now you can use npm to manage packages, install dependencies for your projects, and publish your own packages to the npm registry.

The steps to install Visual Studio Code (VSCode) on your computer

1. Visit the Official Website:

Go to the Visual Studio Code official website at `code.visualstudio.com`.

2. Download the Installation File:

Click on the "Download" button. You'll see options for different operating systems (Windows, macOS, Linux). Choose the one that matches your OS. If you're on Windows, you can choose between a User Installer, System Installer, and a .zip archive. For most users, the User Installer is recommended.

3. Run the Installer:

Once the file has finished downloading, locate it in your downloads folder and run it.

4. Installation Wizard:

Follow the prompts in the installation wizard. You'll be asked to accept the license agreement, choose the installation location, select additional tasks (such as adding VSCode to the PATH and creating a desktop icon), and finally, install the software.

5. Complete the Installation:

Click 'Install' to begin the installation. Once the installation is complete, you can click 'Finish', and you may choose to have VSCode launched immediately after.

6. Open Visual Studio Code:

If it doesn't open automatically after installation, you can find it in your list of installed programs or search for it and open it from there.

7. Optional - Install Additional Components:

VSCode has a rich library of extensions that you can use to customize your experience and add support for additional languages, debuggers, and tools. To access these, click on the Extensions icon in the Activity Bar on the side of the window, then search for the extensions you want and install them.

You can follow these steps to create and run a simple JavaScript program:

1. Create a JavaScript File:

Open your favorite text editor or IDE (such as Visual Studio Code) and create a new file. Save it with a `.js` extension, for example, `app.js`.

2.Write Some JavaScript Code:

Write a simple JavaScript code snippet in the file. For instance, let's create a program that prints "Hello, World!" to the console.

```
// app.js

console.log('Hello, World!');
```

3.Open a Terminal or Command Prompt:

Open your command prompt (Windows). Navigate to the directory where you saved your JavaScript file using the `cd` command.

For example:

4.Run the JavaScript File with Node.js:

Execute the file using Node.js by typing the following command in your terminal or command prompt:

```
node app.js
```

Replace `app.js` with the name of your JavaScript file if it's different.

5.View the Output:

After running the command, you should see the output of your JavaScript code in the terminal. In our example, it will print:

```
Hello, World!
```

And that's it! You've just created and run a simple JavaScript program using Node.js. You can now expand your program to include more complex logic, use modules, and explore the vast features available in the Node.js runtime environment.

Test javascript file using Nodejs

To test a JavaScript file using Node.js, follow these steps:

1) Create or locate the JavaScript file you want to run.
2) Open a terminal or command prompt.
3) Navigate to the directory where the JavaScript file is located
4) Replace test.js with the actual filename if it's different.
5) Run the JavaScript file using Node.js by typing the following command: "node filename.js"
6) Press Enter to execute the command. Node.js will run the JavaScript file, and any output or errors will be displayed in the terminal.

Practical Activity 1.2.3: Create Vue project

Task:

1. Read key reading 1.2.3 and ask clarification where necessary
2. Referring to the previous theoretical activities (1.2.1.) you are requested to go to the computer lab to create Vue project. This task should be done individually.
3. Apply safety precautions.
4. Outline the steps to create Vue project.
5. Referring to the outlined steps provided on task 3, Create Vue project
6. Present your work to the trainer and whole class

Key readings 1.2.3.: Create Vue project

To create a new Vue project, you can use Vue CLI (Command Line Interface) to set up a Vue.js project quickly and efficiently. Here are the steps to create a Vue project using Vue CLI:

1. Install Vue CLI:

Open your command line interface (CLI) and install Vue CLI globally by running the following command:

```
npm install -g @vue/cli
```

2. Create a New Vue Project:

Once Vue CLI is installed, you can create a new Vue project by running:

```
vue create my-vue-project
```

Replace `my-vue-project` with the name you want to give your Vue project.

3. Project Configuration:

After running the command, you will be prompted to choose a preset for your project configuration. You can select the default preset or manually choose features like Vuex, Router, CSS preprocessors, etc.

4. Navigate to Project Directory:

Change your directory to the newly created Vue project by running:

```
cd my-vue-project
```

5. Run the Development Server:

Start the development server to preview your Vue project by running:

```
npm run serve
```

6. Access Your Vue Project:

Open a web browser and navigate to the URL provided by the Vue CLI after starting the development server (usually `http://localhost:8080`).

7. Start Coding:

You can now start coding your Vue project. Edit the files in the project directory, create Vue components, and customize the project as needed.

Install Vue CLI

To install Vue CLI (Command Line Interface) using npm (Node Package Manager), follow these steps:

1. Open a terminal or command prompt on your computer.

2. Ensure that Node.js is installed on your system.

If Node.js is not installed, please visit the Node.js website (https://nodejs.org) and follow the installation instructions specific to your operating system.

Once you have Node.js installed, you automatically have npm available as well. Verify that npm is installed

3. Install Vue CLI globally by using the related command.

```
npm install -g @vue/cli
```

The -g flag is used to install the package globally on your system, allowing you to use the vue command from any directory.

a) Wait for the installation process to complete. It may take a few moments depending on your internet connection speed.

b) Once the installation is finished, you can verify that Vue CLI is installed Vue -v

Initiate Vue Project using terminal

To initiate a new Vue project using the terminal and Vue CLI, you can follow these steps:

Step 1: Open a command prompt or terminal: Launch the command prompt (Windows)on your system.

Step 2: Create a new Vue project: Run the following command to create a new Vue project using Vue CLI:

vue create project-name

Replace "project-name" with the desired name for your project. This command will create a new directory with the specified name and scaffold a basic Vue project inside it.

Step 3: Once prompted vue2 or vue3, select vue2

Difference between vue2 and vue3

Vue 2 and Vue 3 are two major versions of the Vue.js framework. While both versions are designed to help developers build user interfaces, they have some significant differences. Here are some key differences between Vue 2 and Vue 3:

1. Composition API: Vue 3 introduces the Composition API, which provides a new way of organizing and reusing logic in Vue components. It allows developers to define component logic using functions and provides better code organization and reusability compared to the options API in Vue 2.

2. Reactivity System: Vue 3 comes with a revamped reactivity system that is more efficient and flexible than the reactivity system in Vue 2. It uses a proxy-based approach instead of the Object.defineProperty method used in Vue 2, resulting in better performance and improved reactivity.

3. Tree-Shaking and Bundle Size: Vue 3 is designed with better tree-shaking capabilities, which means that unused code can be eliminated more effectively during the build process. This helps reduce the final bundle size of Vue applications.

4. Performance Improvements: Vue 3 introduces various performance optimizations compared to Vue 2. The reactivity system is more performant, and the overall rendering and update mechanisms have been enhanced, resulting in faster rendering and improved runtime performance.

5. Fragments and Teleport: Vue 3 introduces new features like fragments and teleport. Fragments allow developers to group multiple elements without introducing an additional wrapping element, while teleport enables developers to render elements at a different location in the DOM tree, useful for implementing overlays or modals.

6. Composition API Migration: Vue 2 applications can be migrated to Vue 3 while still using the Options API. However, migrating to the Composition API can bring benefits like improved code organization and reusability. Vue 3 provides migration build-time tooling to help with the transition.

7. Size and Ecosystem: Vue 3 has a slightly larger bundle size compared to Vue 2 due to the additional features and optimizations. However, Vue 3 maintains compatibility with most Vue 2 ecosystem libraries and plugins, ensuring a smooth transition for developers.

Step 4: Configure the project (optional): After running the above command, Vue CLI will prompt you to choose a preset for your project. You can select the default preset, manually select features, or choose a preset based on a saved configuration. Use the arrow keys to navigate the options and press Enter to select.

Step 5: Project setup and dependency installation: Once you've chosen the project preset, Vue CLI will proceed with the project setup and install the necessary dependencies. This process may take a few minutes, depending on your internet connection and the selected options.

Navigate through created project by using command

To navigate through a created Vue project using commands in the terminal, you can use the following commands:

1.Change Directory (cd): Use the cd command followed by the directory name to navigate into a specific directory within your Vue project. For example, to navigate into the src directory, you can run:

```
cd src
```

2.List Files and Directories (ls or dir): Use the ls command (on macOS/Linux) or dir command (on Windows) to list the files and directories within the current directory. This can help you see what files and directories are available in the current location. For example:

```
Ls or dir
```

3.Go Up One Level (cd ..): Use cd .. command to go up one level in the directory structure. This allows you to navigate to the parent directory of the current one. For example, if you are in the src directory and want to go back to the project root directory, you can run:

```
cd ..
```

4.Print Working Directory (pwd): Use the pwd command to print the current working directory. This will display the full path of the directory you are currently in. For example:

pwd

These basic commands should help you navigate through the directories of your Vue project using the terminal. By using cd, ls or dir, cd .., and pwd, you can explore the project structure, move into specific directories, and see your current location within the project.

Run the created project in vue

To run a created Vue project in the development server, you can use the following command:

npm run serve

change the logo and welcome message in app.vue

To change the logo and welcome message in the App.vue file of a Vue project, follow these steps:

Step 1: Locate the App.vue file: Navigate to the src folder of your Vue project using a file explorer or terminal. Look for the App.vue file within the src folder.

Step 2: Open the App.vue file: Open the App.vue file in a text editor or an integrated development environment (IDE) of your choice.

Step 3: Update the logo: In the <template> section of the App.vue file, you can find the HTML code that defines the structure of the app. Locate the part where the logo is displayed and modify it according to your needs. This might involve changing the image source (src) or modifying the HTML structure.

For example, if the logo is displayed using an <img> tag like this:

You can replace ./assets/logo.png with the path to your new logo image file.

Step 4: Update the welcome message: The welcome message is usually defined in the data section of the App.vue file. Locate the data property in the Vue component and modify the message property to change the welcome message. For example:

You can replace 'Welcome to My Vue App!' with your desired welcome message.

Step 5: Save the changes: Once you have made the necessary changes to the logo and welcome message, save the App.vue file.

Step 6: Verify the changes: To see the updated logo and welcome message, ensure that your Vue project's development server is running. In the terminal, navigate to the project directory and run npm run serve. Then, open your Vue application in a web browser, and you should see the modified logo and welcome message.

By following these steps, you can customise the logo and welcome message in the App.vue file of your Vue project. Remember to make sure the logo file is present in the appropriate location and the changes are saved before testing the application.

Points to Remember

- JavaScript frameworks are essential tools for web developers to build dynamic and interactive web applications efficiently. There are some commonly used JavaScript frameworks: React, Angular, Express.js, Vue.js.
- Developing a JavaScript program involves writing code that can be executed in a web browser or server environment. Here are the steps to develop a JavaScript program:

    1. Set Up Your Development Environment
    2. Create a New JavaScript File
    3. Write Your JavaScript Code
    5. Debug Your Code
    6. Test Your JavaScript Program
    7. Refine and Optimize Your Code

- To create a new Vue project, you can use Vue CLI (Command Line Interface) to set up a Vue.js project quickly and efficiently. Here are the steps to create a Vue project using Vue CLI:

    1. Install Vue CLI
    2. Create a New Vue Project
    3. Project Configuration
    4. Navigate to Project Directory
    5. Run the Development Server
    6. Access Your Vue Project
    7. Start Coding

Application of learning 1.2.

You have been hired by your school to develop a Vuejs game for students to always play at break time in the computer Lab, as game developer in Vuejs framework create a Vuejs project on your computer desktop using Vue CLI and name it my project. Open it in VS CODE and display the output by loading the server URL on a web browser of your choice.

Indicative content 1.3: Description of Vue Project Folder & Files

Duration: 6 hrs

Theoretical Activity 1.3.1: Description of Folder and file structure in vue js project

Tasks:

1.You are requested to answer the following questions related to the folder and file structure in Vue Js:

I. Provide explanations for the following terms commonly used in Vue projects folder:

a) node_modules

b) public

c) src

d) asset

e) components

f) app.vue

g) main.js

h) app.vue

i) package. json

j) vue.config.j

k) . gitignore

l) babel.config.js

m) jsconfig.json

n) README.md and package-lock.json.

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 1.3.1. In addition, ask questions where necessary.

Key readings 1.3.1.: Description of Folder and file structure in vue js project

1. node_modules: This folder is automatically created by npm or Yarn when installing project dependencies. It contains all the dependencies specified in the project's package.json file.

"node_modules" is a directory that is commonly found in projects that use Node.js for server-side JavaScript development. When working with Node.js, developers often rely on various external libraries or modules to enhance the functionality of their applications. These modules can be installed using a package manager like npm (Node Package Manager).

When a developer installs a module using npm, the module and its dependencies are typically downloaded and stored in the "node_modules" directory within the project's root directory. This directory acts as a local repository for all the installed modules specific to that project.

The "node_modules" directory contains subdirectories for each installed module, along with their respective dependencies if any. These modules are typically stored as separate folders, each containing code files, configuration files, and other assets specific to that module.

The purpose of the "node_modules" directory is to organize and manage the dependencies of a Node.js project. It allows developers to easily include and use external modules in their code by referencing them from the "node_modules" directory.

2. public: The public folder contains static assets that are served as-is by the web server. This includes HTML files, images, fonts, and other resources. The contents of the public folder are typically accessible directly from the web without any processing or compilation.

In a Vue.js project, the term "public" refers to a specific directory within the project structure. The "public" directory is the default location for static assets that need to be publicly accessible by the application. It is where you can place files like HTML, images, fonts, or other static resources that you want to be directly served to the client without going through the build process.

Here's a brief overview of the "public" directory and its purpose in a Vue project:

1. index.html: The main HTML file of your Vue application resides in the "public" directory. This file serves as the entry point for your application and is responsible for loading the Vue app's JavaScript and CSS files.

2. Static Assets: The "public" directory is commonly used to store static assets like images, fonts, or other files that are required by your application. These assets can be referenced directly in your HTML or Vue components using relative paths.

3. External Libraries or Scripts: If your Vue project requires any external libraries or scripts that are not managed by a package manager like npm, you can place them in the "public" directory and include them in your HTML file using script tags.

3. src: The src (source) folder is where the main source code of the Vue.js project resides. It contains the application's JavaScript files, Vue components, stylesheets, and other related files.

In a Vue.js project, the "src" directory is a crucial part of the project structure. It contains the source code and files that make up the Vue application. Here's a breakdown of the "src" directory and its contents in a Vue project:

1. main.js: This file serves as the entry point of the Vue application. It initializes the Vue instance and mounts it to the DOM, typically targeting the "index.html" file. It also imports and registers any global components, plugins, or libraries that are used throughout the application.

2. App.vue: This is the root component of the Vue application. It acts as a container for other components and defines the overall structure and layout of the application. It typically includes the main navigation, header, and footer components.

3. components: The "components" directory contains individual Vue component files. Each component is typically defined in its own file with a ".vue" extension. These components are reusable and can be imported and used within other components or templates.

4. assets: The "assets" directory is used to store static assets like images, stylesheets, or other media files that are specific to the Vue application. These assets can be imported and used within components or templates.

5. router: If the Vue project uses Vue Router for client-side routing, the "router" directory contains the configuration and setup files for the router. It typically includes a "index.js" file that defines the routes and their corresponding components.

6. store: If the Vue project uses Vuex for state management, the "store" directory contains the Vuex store configuration and modules. It typically includes a "index.js" file that sets up the store and imports individual modules.

7. views: The "views" directory contains the Vue components that represent the different pages or views of the application. Each view component typically corresponds to a specific route and may include child components.

4. assets: The assets folder is typically used to store static assets such as images, icons, and stylesheets that are imported and used within the Vue components.

5. components: The components folder is where reusable Vue components are stored. Components are modular building blocks that encapsulate HTML, CSS, and JavaScript logic to create reusable UI elements.

6. App.vue: App.vue is the root component of a Vue.js application. It serves as the main entry point for the application and typically contains the overall layout, navigation, and the router view where other components are rendered.

7. main.js: main.js is the entry point of the Vue.js application. It initializes the Vue instance, configures plugins, sets up global configurations, and mounts the root component (App.vue) to the DOM.

8. package.json: package.json is a metadata file that contains information about the project and its dependencies. It lists the project's dependencies, scripts, and other project-specific configurations.

9. vue.config.js: vue.config.js is a configuration file that allows you to customize various aspects of the Vue.js project's build process. It can be used to configure webpack, set up proxy, define environment variables, and more.

10. .git ignore: .gitignore is a file that specifies which files and folders should be ignored by the version control system (Git) when committing changes. It typically includes files or folders that are generated or contain sensitive information.

11. babel.config.js: babel.config.js is a configuration file for Babel, a JavaScript compiler. It allows you to specify presets, plugins, and other Babel configurations to transform JavaScript code.

12. jsconfig.json: jsconfig.json is a configuration file used for JavaScript projects in Visual Studio Code. It provides IntelliSense and helps with code navigation and suggestions within the editor.

13. README.md: README.md is a Markdown file that typically serves as the project's documentation. It provides an overview of the project, instructions for installation, usage, and other relevant information.

14. package-lock.json: package-lock.json is a file generated by npm when installing or updating dependencies. It locks the specific versions of the installed packages to ensure consistent installations across different environments.

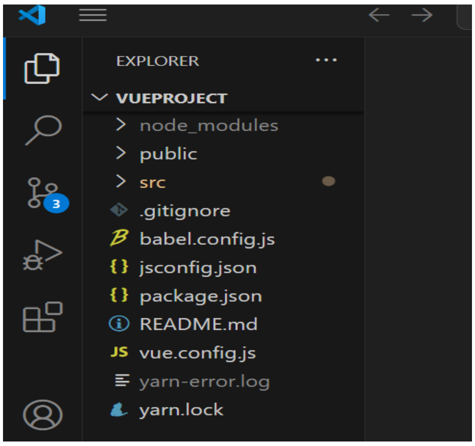
Note: apart from those files and folders you can create others depending on the task that you want to perform.

Practical Activity 1.3.2: Navigate through Vue project folder & files

Task:

1. Read key reading 1.3.2 and ask clarification where necessary
2. Referring to the previous theoretical activities (1.3.1) you are requested to go to the computer lab to navigate through the created Project folder. This task should be done individually.
3. Apply safety precautions
4. Outline the steps to Navigate through the created project folder.
5. Referring to the outlined steps provided on task 3, navigate through the created Project folder.
6. Present your work to the trainer and whole class

Key readings 1.3.2.: Navigate through Vue project folder & files



EXPLORER

VUEPROJECT
- node_modules
- public
- src
- .gitignore
- babel.config.js
- jsconfig.json
- package.json
- README.md
- vue.config.js
- yarn-error.log
- yarn.lock

Navigating through a Vue project folder allows you to explore the project structure, view files, make changes, and manage your Vue.js application effectively.

Here are the steps to navigate through a created Vue project folder:

1. Open Command Line Interface (CLI):

- Open your preferred command line interface (CLI) application such as Command Prompt, Terminal, or PowerShell on your computer.

2. Change Directory to Your Vue Project:

- Use the `cd` command to change the directory to your Vue project folder. For example, if your project is named "my-vue-project":

```
cd path/to/your/vue/project/my-vue-project
```

3. List Files and Folders:

- To view the contents of the current directory, you can use the `ls` command on Unix-based systems or `dir` command on Windows:

```
ls    // Unix-based systems

dir   // Windows
```

4. Navigate to Specific Folders:

- Use the `cd` command to enter specific folders within the project directory. For example, to enter the `src` folder:

```
cd src
```

5. View Files and Content:

- Use text editors or CLI tools to view the content of files within the project. For example, to view the contents of the `App.vue` file:

```
cat App.vue    // Displays the content of the file (replace `cat` with appropriate command based on your CLI)
```

6. Navigate Back to Previous Directory:

- To navigate back to the previous directory, you can use the `cd ..` command:

```
cd ..
```

7. Run Development Server:

- If you want to run the development server to preview your Vue project, you can use the following command:

```
npm run serve
```

8.Access the Vue Application:

- Open a web browser and navigate to the URL provided by the Vue CLI after starting the development server (usually `http://localhost:8080`) to access and interact with your Vue application.

To create a project in Vue.js, you'll need to follow a few steps. Here's a guide to help you get started:

Step 1: Install Node.js Before setting up a Vue.js project, make sure you have Node.js installed on your computer. Node.js is required to run the build tools and development server used by Vue.js.

Step 2: Install Vue CLI Vue CLI is a command-line tool specifically designed for scaffolding Vue.js projects. It provides a streamlined way to set up a project with a pre-configured build system and various development tools.

To install Vue CLI, open your command-line interface (CLI) and run the following command:

```
npm install -g @vue/cli
```

Step 3: Create a New Vue Project Once Vue CLI is installed, you can create a new Vue project by running the following command:

```
vue create my-project
```

Replace "my-project" with the desired name of your project. This command will prompt you to choose a preset for your project configuration. You can either select a default preset or manually configure the project. For beginners, the default preset is a good starting point.

Step 4: Navigate to the Project Directory After creating the Vue project, navigate to its directory using the following command:

```
cd my-project
```

Step 5: Start the Development Server Once inside the project directory, you can start the development server using the following command:

```
npm run serve
```

This will compile the project and launch a local development server. You'll see a message indicating the server address (usually http://localhost:8080). Open that address in your web browser, and you should see your Vue.js project running.

Step 6: Explore and Modify the Project At this point, you have a basic Vue.js project set up. You can start exploring the project structure and making modifications. The main project files are located in the "src" directory, including the entry point file "main.js" and the root component file "App.vue."

You can edit these files and create additional components in the "src" directory to build your application. Vue CLI also provides a hot-reload feature, which means the browser automatically refreshes whenever you make changes to your code.

These are the essential steps to create a Vue.js project using Vue CLI. From here, you can continue building your application by adding components, managing state with Vuex, and integrating with APIs or backend services as needed.

To open a Vue.js project in Visual Studio Code (VS Code)

you can follow these steps:

Step 1: Install Visual Studio Code If you don't have Visual Studio Code installed on your computer, you can download it from the official website (https://code.visualstudio.com/) and follow the installation instructions for your operating system.

Step 2: Open Visual Studio Code Launch Visual Studio Code by clicking on its icon or searching for it in your application launcher.

Step 3: Open the Project Folder In VS Code, go to the "File" menu and select "Open Folder" (or use the shortcut Ctrl+K Ctrl+O on Windows/Linux or Command+K Command+O on macOS). A file browser dialog will appear.

Navigate to the directory where your Vue.js project is located, select the project folder, and click "Open" or press Enter. This will open the project in VS Code.

Step 4: Explore the Project Structure Once the project is open, you'll see the file and folder structure in the Explorer sidebar on the left-hand side of the VS Code window. The key files and folders in a Vue.js project are typically:

●**src:** Contains the main source code of your Vue.js application.

●**public:** Contains static files that are directly copied to the build output.

●**node_modules:** Contains the dependencies installed via npm or yarn.

●**package.json:** The project configuration file that includes metadata and script commands.

Step 5: Install Dependencies If you've just opened a new Vue.js project or a project that you haven't worked on before, you'll need to install the project dependencies. Open a terminal in VS Code by going to the "Terminal" menu and selecting "New Terminal" (or using the shortcut Ctrl+`).

In the terminal, navigate to your project's root directory and run the following command to install the dependencies:

```
npm install
```

This will read the dependencies listed in the package.json file and download them into the node_modules folder.

Step 6: Start Development Once the dependencies are installed, you can start working on your Vue.js project in VS Code. You can edit the files, create or modify components, and manage the project structure.

To start the development server and see your changes in the browser, open a terminal in VS Code and run the following command:

```
npm run serve
```

The development server will compile your Vue.js project and provide a local preview accessible at the address displayed in the terminal (usually http://localhost:8080). Open this address in your web browser to see your Vue.js application in action.

To explore the App.vue file in a Vue.js project,

you can follow these steps:

1.**Open the App.vue file:**

- In Visual Studio Code, locate the src folder in the Explorer sidebar on the left-hand side.
- Expand the src folder to reveal its contents.
- Look for the App.vue file and click on it to open it in the editor.

2.**Understand the structure:** The App.vue file is the root component of a Vue.js application. It typically consists of three sections: <template>, <script>, and <style>. These sections define the HTML template, JavaScript logic, and CSS styles for the component, respectively.

3.**Explore the <template> section:** The <template> section contains the markup structure and HTML elements that make up the component's UI. This is where you define the layout, structure, and content of your application.

- **Identify the root element:** Look for the outermost HTML element, which serves as the root element of the component. By default, it is usually a <div> element with the class name app.
- **Examine the content:** Explore the child elements within the root element and inspect their structure, attributes, and classes. This is where you define the various components, elements, and data bindings that make up your application's UI.

4.**Analyse the <script> section:** The <script> section contains the JavaScript code that provides the logic and functionality for the component. It includes the Vue.js component options and lifecycle hooks.

5.Review the import statements: Look for any imported modules or dependencies that the component requires.

- **Understand the data property:** Identify the data property, which defines the component's initial data and state. This is where you can declare and initialize variables used within the component.
- **Explore the methods:** Look for any defined methods within the component. These methods handle events, perform computations, and interact with the component's data.
- **Investigate lifecycle hooks:** Take note of any lifecycle hooks such as created, mounted, or updated. These hooks provide a way to execute code at specific stages of the component's lifecycle.

5.Inspect the <style> section: The <style> section contains the CSS styles specific to the App.vue component.

6.**Examine the scoped attribute**: Notice that the <style> tag may have the scoped attribute. This ensures that the defined styles only apply to the App.vue component and its child elements.

6.**Make modifications:** You can experiment and modify the App.vue file to suit your application's needs. You can add or remove HTML elements, adjust data properties, define new methods, or apply custom styles.

To insert an image into the asset folder of your project

you can follow these steps:

Locate the asset folder:

- In most Vue.js projects, the asset folder is typically named "assets" and is located in the root directory of your project. Open your project's file explorer or navigate to the project directory using your operating system's file explorer.
- Prepare the image file:
- Ensure you have the image file you want to insert into the asset folder. If you don't have the image file yet, make sure to have it ready on your computer.
- Copy the image file to the asset folder:
- Drag and drop the image file from its current location into the asset folder in your project's file explorer. Alternatively, you can use the copy-paste command (Ctrl+C and Ctrl+V on Windows/Linux, or Command+C and Command+V on macOS) to copy the image file from its location and paste it into the asset folder.
- Verify the image is in the asset folder:
- Check that the image file now appears in the asset folder. You should see the image file listed among the other files and folders in the asset folder.
- Reference the image in your code:
- To use the image in your Vue.js components or templates, you'll need to reference it using its relative path from the asset folder.
- For example, if your image file is named "my-image.png" and located directly inside the asset folder, you can reference it in your code like this:

```
<img src="@/assets/my-image.png" alt="My Image">
```

Note that @/ is a special alias in Vue CLI that refers to the root directory of your project.

Adjust the relative path as needed depending on the location of the image file within the asset folder and the structure of your project.

Use the image in your application:

With the image file in the asset folder and properly referenced, you can now use the image in your Vue components or templates by placing the appropriate code. For example, you might insert an <img> tag with the specified src attribute.

```
<template>
```

```
  <div>
    <img src="@/assets/my-image.png" alt="My Image">
  </div>
</template>
```

run vue project

To run a Vue.js project, you need to start the development server. Here are the steps to run a Vue project:

1. Open your command-line interface (CLI): Launch your preferred CLI, such as Command Prompt (Windows), Terminal (macOS), or any other terminal emulator.

2. Navigate to your project directory: Use the cd command to navigate to the directory where your Vue.js project is located. For example:

```
cd path/to/your/project
```

Replace path/to/your/project with the actual path to your Vue project folder.

Start the development server: Once you are inside the project directory, run the following command to start the development server:

```
npm run serve
```

1. This command runs the predefined script in your project's package.json file to start the server. It will compile your Vue project and provide a local development server that you can access in your web browser.

2. Access your Vue project in the browser: After running the command, you'll see output in the CLI indicating that the server is running. Look for a message that includes the local address where your Vue project is being served, typically http://localhost:8080/.

Open your web browser and enter the provided address (http://localhost:8080/ or a different port if specified) to access your Vue.js project. You should see your Vue application running in the browser.

3. Make changes and see live updates: The development server has a hot-reload feature, which means that any changes you make to your project's source code will automatically trigger a recompile and update in the browser. You can modify your Vue components, styles, and templates, and see the changes instantly without manually refreshing the page.

> Keep the CLI running with the development server to keep your Vue project running and see live updates as you make changes. If you want to stop the development server, you can use the Ctrl+C command in the CLI.

Points to Remember

●In a Vue project created using Vue CLI, the project folder and files are structured in a way to facilitate the development, configuration, and deployment of Vue.js applications. There are typical structure of a Vue project folder and the of key files:Node _modules, Public folder,src,Asset, Components, helloWorld.vue,app.vue, main.js, App.vue,Package.json Vue.config.js, .git ignore,babel.config.js.

●Navigating through a Vue project folder allows you to explore the project structure, view files, make changes, and manage your Vue.js application effectively. Here are the steps to navigate through a created Vue project folder:

1. Open Command Line Interface (CLI)
2. Change Directory to Your Vue Project
3. List Files and Folders
4. Navigate to Specific Folders
5. View Files and Content
6. Navigate Back to Previous Directory
7. Run Development Server
8. Access the Vue Application

Application of learning 1.3.

As a developer joining a Vue.js project for the first time, you're tasked with familiarizing yourself with the project's folder structure and files, through navigating the Vue project's created of folders and files. By removing, all unwanted links on main interface and display on welcome message that attract visitors of the school website.

Learning outcome 1 end assessment

Written assessment

1. Which term refers to the client-side of a web application, which is what the users interact with directly in their web browsers, while ________ are responsible for creating an engaging and user-friendly experience?

A. Frontend

B. Backend

C. Middleware

D. Fullstack

2. Select the best term that define single page application (SPA):

A. A web application that loads a single HTML page and dynamically updates it as the user interacts with the app

B. An application that requires multiple HTML pages to function

C. An application that only works offline

D. An application that runs only on mobile devices

3. Which command can be used to check the version of Node.js and npm installed on your computer?

A. node -v

B. npm -version

C. npm -v

D. node -version

4. How can you install Vue.js in your project?

A. npm install vue

B. npm add vue

C. npm create vue

D. npm vue install

5. Which of the following are necessary steps to run a simple JavaScript program that can add up two numbers using Node.js?

A. Write the JavaScript code

B. Save the file with a .js extension

C. Open the terminal and navigate to the directory containing the file

D. Run the command node filename.js

E. All of the above

6. Suppose that you are a developer and you want to create a game project with the name "snake game." Which command can be used to create the desired project?

A. npm create snake-game

B. npm init snake-game

C. vue create snake-game

D. create-react-app snake-game

7. CLI stands for:

a) Command Line Interface

b) Computer Learning Interface

c) Client-Side Interface

d) Client-Side Logic

8. Which of the following is NOT a common feature of an IDE?

a) Code editor

b) Build tools

c) Debugging capabilities

d) Web browser

9. The frontend of a web application primarily deals with:

a) Server-side logic

b) Database interactions

c) User interface and presentation

d) Network communication

10. In a Single Page Application (SPA)

a) The entire application runs on a single HTML page.

b) Multiple HTML pages are loaded for each user action.

c) The server handles all the rendering.

d) The user experience is static.

11. Vue.js is primarily a:

a) Backend framework

b) Frontend framework

c) Database management system

d) Programming language

12. Match the following files and folders typically included in a Vue.js project with their descriptions:

a. src

b. components

c. node_modules

d. App.vue

e. package.json

f. .gitignore

i. Contains metadata about the project and dependencies

ii. Stores reusable Vue components

iii. Main folder for source code

iv. Entry point component of a Vue.js application

v. Ignored files and directories for version control

vi. Installed packages and dependencies

13.Respond by True or False to the followings:

a) A framework provides a structured and reusable foundation for building applications.

b) The cd command is used to create a new directory in the Command Prompt.

c) IDEs typically include a code editor, build tools, and debugging capabilities.

d) The Vue.js framework is primarily used for backend development.

e) Single Page Applications (SPAs) require page reloads for every user action.

f) The virtual DOM is a real-time representation of the browser's DOM.

g) Vue.js follows a component-based architecture.

h) The del command is used to list files and directories in the Command Prompt.

i) A dependency in Vue.js refers to an external library or module required by your project.

j) The mkdir command is used to change the current directory in the Command Prompt.

Practical assessment

XYZ Rwandan museum is a museum located in Musanze district, Muhoza sector, they have a campaign directed toward educating children about historical figures and their contribution to our history.

In the beginning this campaign was conducted via historians in the museum explaining the children about those historical figures, but this method was ineffective since children would get bored and stop paying attention.

As developer, you have been assigned a task to create a **game project** folder by using vue framework that will be used while developing the system that will help them display “**welcome to Rwanda Museum** ''on the main interface that can attract users.

The needed tools, materials and equipment have been provided by company.

References

Shavin, M. (2024). Learning Vue. O'Reilly Media.

Lynch, A., & Murray, S. (2018). Fullstack Vue: The Complete Guide to Vue.js. Fullstack.io.

Hufkens, S. (2021). Getting to Know Vue.js: Learn to Build Single Page Applications in Vue from Scratch. Apress.

Bemmerl, H. (2021). Vue.js 3 Cookbook: Discover actionable solutions for building modern web apps with the latest Vue features. Packt Publishing.

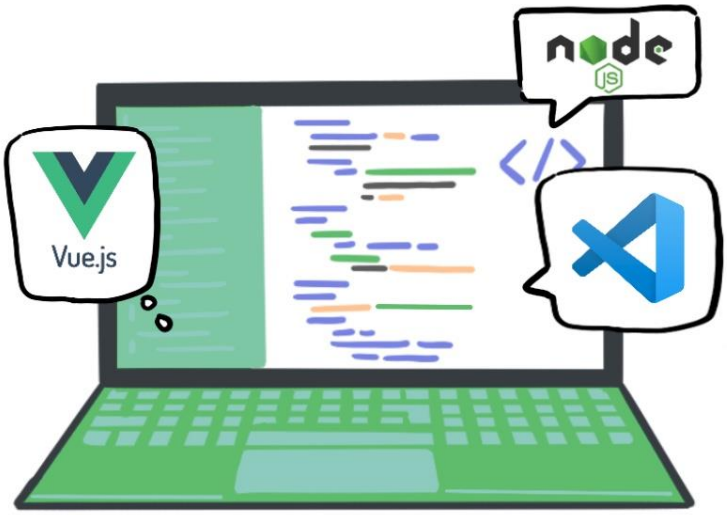
Vue.js. (n.d.). Vue CLI [Computer software]. GitHub. Retrieved March 14, 2022, from https://github.com/vuejs/vue-cli

Vue.js. (n.d.). *Vue CLI*. Retrieved March 14, 2022, from https://cli.vuejs.org/

Vue.js Forum. (n.d.). Vue.js Community Forum. Retrieved March 14, 2022, from https://forum.vuejs.org/

Vue.js. (n.d.). Installation guide. Retrieved March 14, 2022, from https://vuejs.org/v2/guide/installation.html

Learning Outcome 2: Apply Vue Framework



Indicative contents

2.1 Create folder structure

2.2 Apply Vue component structure

2.3 Apply navigation in Vue project using router

2.4 Data manipulation in Vue

2.5 API requests

2.6 Manage data using state management

Key Competencies for Learning Outcome 2: Apply Vue Framework

Knowledge	Skills	Attitudes
• Description of the term API request • Description of elements that make up a form • Description of the term state management	• Creating folder structure • Creating Vue components • Using Vue components • Using form in Vue • Applying navigation in Vue project using router • Managing data using state management	• Having Team work spirit • Being critical thinker • Being Innovative • Being attentive. • Being creative • Being Problem solver • Being Practical oriented

Duration: 30 hrs

Learning outcome 2 objectives:

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

1. Describe clearly the term API request e in accordance with user stories
2. Describe properly elements that make up a form based on user story
3. Describe properly the term state management in accordance with user stories
4. Create correctly routes in line with project pages
5. Develop correctly reusable components in accordance with most reusable HTML elements
6. Handle properly form data based on user requirements
7. Validate correctly form data based on user requirements
8. Develop correctly features in accordance with user requirements
9. Make correctly API requests in accordance with system requirements

Resources

Equipment	**Tools**	**Materials**
• Computer	• Text Editor (vscode) • Nodejs • Vue framework	• Internet

Indicative content 2.1: Create Folder Structure

Duration: 3 hrs

Theoretical Activity 2.1.1: Description of key concepts

Tasks:

1. You are requested to answer the following questions related to the folder structure.

I. What do you understand about the following terms used in Vue framework:

a) Components

b) Vue lifecycle

c) Routes

d) State management

e) API Endpoint

f) . env file

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.1.1. In addition, ask questions where necessary.

Key readings 2.1.1.: Description of key concepts

●Components

Components in Vue.js are reusable and self-contained building blocks that encapsulate the HTML, CSS, and JavaScript logic required for a specific piece of functionality or UI element. Vue components allow developers to create modular, maintainable, and reusable code.

●Routes

Routes in Vue.js are used to define the navigation paths and mapping between URLs and corresponding components in a single-page application (SPA). With the help of a router library like Vue Router, routes can be defined to load different components based on the URL, enabling navigation and rendering different views based on user interactions.

●Vue lifecycle

The Vue lifecycle refers to a series of predefined methods or hooks that are invoked at different stages of a Vue component's lifespan.

These hooks allow developers to perform actions or execute code at specific moments during the component's creation, update, and destruction. Examples of lifecycle hooks include created, mounted, updated, and destroyed.

●State management

State management in Vue.js involves managing and synchronizing the shared data and state across different components in an application.

The most common approach to state management in Vue.js is using a library called Vuex, which provides a centralized store for managing application state and enables components to access and modify the state in a predictable and organized manner.

●API Endpoint

In the context of Vue.js, an API endpoint refers to the URL or route on a server that exposes a specific functionality or resource via an API (Application Programming Interface). It is the entry point for making HTTP requests to interact with the backend server, such as retrieving data, submitting forms, or performing CRUD (Create, Read, Update, Delete) operations.

●. env file

The. env file is a configuration file used to store environment variables in a Vue.js project. Environment variables are values that can be accessed within the application and can vary based on the environment (development, production, etc.). The .env file allows developers to define and manage environment-specific settings, such as API endpoints, database credentials, or application-specific constants, without hardcoding them directly in the code.

Practical Activity 2.1.2: Create folder structure in VueJS project

Task:

1. Read key reading 2.1.2 and ask clarification where necessary
2. Referring to the previous theoretical activities (2.1.1), you are requested to go to the computer lab to create folder structure. This task should be done individually.
3. Apply safety precautions.
4. Present out the steps to create folder structure.
5. Referring to the steps provided on task 3, create folder structure.
6. Present your work to the trainer and whole class

Key readings 2.1.2.: Create folder structure in VueJS project

Creating a well-organized folder structure is essential for maintaining and scaling a Vue.js project effectively.

Here are the steps to create a typical folder structure in a Vue.js project:

1. Initialize a Vue Project:

If you haven't already set up a Vue project, you can use Vue CLI to create one. Run the following command in your command line interface:

```
vue create my-vue-project
```

Replace `my-vue-project` with the name of your project.

2. Navigate to Your Project Directory:

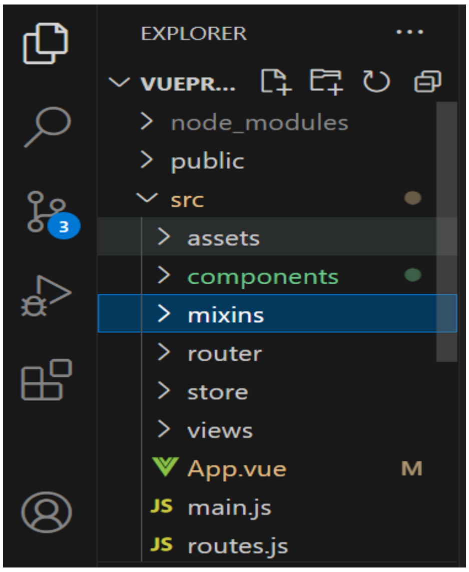
Change your directory to the newly created Vue project:

```
cd my-vue-project
```

3. Create a Structured Folder Layout:

Within your Vue project directory, create a structured folder layout that suits your project needs.

Here is a common folder structure used in Vue.js projects:



```
node_modules
public
src
  assets
  components
  mixins
  router
  store
  views
  App.vue     M
  main.js
  routes.js
```

4. Assets:

The `/assets` directory is typically used to store static assets like images, fonts, and CSS files that are used in the project.

5. Components:

The `/components` directory is where you store Vue components that can be reused across different parts of your application. Each component should have its own folder with a `.vue` file for the component.

6. Views:

The `/views` directory contains Vue components representing different views or pages of your application. Each view component typically corresponds to a specific route in your application.

7. App.vue:

The `App.vue` file is the root Vue component that serves as the entry point of your application. It typically contains the main template and structure of your app.

8. main.js:

The `main.js` file is the entry point of your Vue application where the Vue instance is created and the app is mounted to the DOM. It is where you import Vue and other dependencies.

9. Customize the Structure:

Feel free to customize the folder structure based on the needs of your project. You can create additional folders for services, utilities, plugins, or any other specific functionality.

Create components folder works in Vue

1. Open your Vue.js project in your preferred code editor.

2. In the root directory of your project, create a new folder called "components". This is where you will store your Vue components.

3. Inside the "components" folder, create a new file called "ExampleComponent.vue". This will be a sample component.

4. Open "ExampleComponent.vue" in your code editor and add code depending on the activity to be performed.

Create a router folder in a Vue.js project

1. Assuming you have already set up a Vue.js project using the Vue CLI, open your project's file structure in your preferred code editor.

2. Inside the src directory, create a new folder called router. This is where we'll store our router-related files.

Here's an example of how you can create a router folder in a Vue.js project:

1. Open your Vue.js project in your preferred code editor.

2. In the root directory of your project, create a new folder called "router". This is where you will store your router-related files.

3. Inside the "router" folder, create a new file called "index.js". This will be the main file for configuring the Vue Router.

4. Open "index.js" in your code editor and add code depending on the activity to be performed.

Create a store folder in a Vue.js project

1. Assuming you have already set up a Vue.js project using the Vue CLI, open your project's file structure in your preferred code editor.

2. Inside the src directory, create a new folder called store. This is where we'll store our Vuex store-related files.

Here's an example of how you can create a store folder in a Vue.js project:

1. Open your Vue.js project in your preferred code editor.

2. In the root directory of your project, create a new folder called "store". This is where you will store your Vuex store-related files.

3. Inside the "store" folder, create a new file called "index.js". This will be the main file for configuring the Vuex store.

4. Open "index.js" in your code editor and add code depending on the GUI to be designed.

To use the store configuration in your main Vue component, follow these steps:

Create a views folder in a Vue.js project

1. Assuming you have already set up a Vue.js project using the Vue CLI, open your project's file structure in your preferred code editor.

2. Inside the src directory, create a new folder called views. This is where we'll store our view components.

3. Within the views folder, create individual Vue component files for each view you want to create.

Here's an example of how you can create a views folder in a Vue.js project:

1. Open your Vue.js project in your preferred code editor.

2. In the root directory of your project, create a new folder called "views". This is where you will store your Vue views or pages.

3. Inside the "views" folder, create a new file called "Home.vue". This will be an example view component.

4. Open "Home.vue" in your code editor and add code:

Create a mixins folder in a Vue.js project

1. Assuming you have already set up a Vue.js project using the Vue CLI, open your project's file structure in your preferred code editor.

2. Inside the src directory, create a new folder called mixins. This is where we'll store our mixins files.

3. Within the mixins folder, create individual JavaScript files for each mixins you want to create.

Here's an example of how you can create a mixins folder in a Vue.js project:

1. Open your Vue.js project in your preferred code editor.

2. In the root directory of your project, create a new folder called "mixins". This is where you will store your Vue mixins.

3. Inside the "mixins" folder, create a new file called "exampleMixin.js". This will be an example mixins.

4. Open "exampleMixin.js" in your code editor and add code.

Points to Remember

- There are key concepts used to apply the Vue framework that developers should understand to form the foundation of Vue development. Here are key concepts used to apply the Vue framework: Components, Routes, Vue lifecycle, State management, API Endpoint, . env file
- Creating a well-organized folder structure is essential for maintaining and scaling a Vue.js project effectively. Here are the steps to create a typical folder structure in a Vue.js project:
    1. Initialize a Vue Project
    2. Navigate to Your Project Directory
    3. Create a Structured Folder Layout

Application of learning 2.1.

You have been assigned task to create a project called **Vue-project** and within that project there is a folder called src, as a developer in the company you are requested to create the following folders (Router, Store, Views and Mixins) and those folders will be used to save other project files.

Indicative content 2.2: Apply Vue Component Structure

Duration: 3 hrs

Theoretical Activity 2.2.1: Description of components in VueJS framework

Tasks:

1. You are requested to answer the following questions related to the components in Vue Js framework:

   I. What are the key differences between CSS, HTML, and JavaScript?"

   II. What do you understand about component?

   III. What are the main parts that make up a component in Vue.js

   IV. What are the benefits of using components in Vue.js?

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.2.1. In addition, ask questions where necessary.

Key readings 2.2.1.: Description of components in VueJS framework

HTML, CSS, and JavaScript are three fundamental technologies used for web development, and they serve different purposes.

In Vue.js, components are the building blocks of an application. They allow you to encapsulate reusable code and create a modular structure for your UI.

CSS (Cascading Style Sheets), HTML (HyperText Markup Language), and JavaScript are the core technologies of the World Wide Web. Each serves a distinct purpose in the development of web pages and web applications:

HTML (HyperText Markup Language):

Purpose: HTML is used to create the structure and content of a web page. It defines the skeleton of the page and is responsible for the organization of text, images, and other media.

Functionality: HTML consists of a series of elements represented by tags (like `<p>`, `<div>`, `<img>`, etc.) that browsers interpret to display content.

Semantics: HTML elements convey the meaning and structure of the content. For instance, `<header>`, `<footer>`, `<article>`, and `<section>` are semantic elements that indicate the role of different parts of the web page.

CSS (Cascading Style Sheets):

Purpose: CSS is used for describing the presentation and design of a web page written in HTML or XML (including XML dialects such as SVG or XHTML). CSS describes how elements should be rendered on screen, on paper, in speech, or on other media.

Functionality: CSS allows you to apply styles to HTML elements, such as colors, fonts, spacing, positioning, and animations. It separates content (HTML) from presentation.

Cascading: CSS rules cascade, which means that the styling can be defined in multiple stylesheets or within the same sheet, and the specificity and order of these rules determine how the styles are applied to the elements.

JavaScript:

Purpose: JavaScript is a scripting language used to create dynamic content, control multimedia, animate images, and pretty much everything else that involves interactivity on the web.

Functionality: JavaScript can manipulate the Document Object Model (DOM), allowing for the modification of HTML and CSS to update the user interface. It can also handle events, perform calculations, and make logical decisions.

Interactivity: JavaScript is what makes web pages interactive. It can respond to user actions like clicks, form submissions, and mouse movements, and can update the page content without the need to reload the page (AJAX).

Each component consists of three main parts:

➢Template: That include all HTML elements to be displayed on user interface

➢Script: Contains Javascript elements that help in interaction with graphical user interface

➢Style: Where we include the appearance or CSS codes.

Note

●The <template> section contains the HTML template for your component.

●The <script> section defines the component's options, methods, and logic.

●The <style scoped> section allows you to define component-specific styles that won't affect other components.

Reusable components

Reusable components in Vue.js are modular building blocks that encapsulate a specific piece of functionality or UI element. They are designed to be self-contained and can be easily reused across different parts of an application or even in multiple projects.

Reusable components play a vital role in Vue.js development by promoting code reusability, modularization, and maintainability.

Here are some key benefits of components in Vue.js:

1. Code Reusability: Components in Vue.js encapsulate UI and logic, making them self-contained and reusable across different parts of your application. Reusing components reduces code duplication, saves development time, and promotes consistency throughout the application.

2. Modularity: Vue.js follows a component-based architecture, where the application is built by composing reusable components together. Each component represents a specific functionality or UI element, making it easier to understand and manage the codebase. This modularity facilitates separation of concerns, improves code organisation, and enables better collaboration among developers.

3. Scalability: By breaking down complex UIs into smaller, reusable components, Vue.js allows you to scale your application more efficiently. You can add, modify, or remove components as needed, without affecting other parts of the application. This modular approach makes it easier to maintain and extend your codebase as your application grows.

4. Maintainability: Reusable components in Vue.js promote maintainability by isolating specific functionality or UI elements. Each component focuses on a specific task, making it easier to test, debug, and update independently. When a change is required, you can modify a single component without affecting other parts of the application.

5. Composition: Vue.js encourages the composition of components, where larger components are built by combining smaller, reusable components

Theoretical Activity 2.2.2: Description of CSS framework

Tasks:

1. You are requested to answer the following questions related to the CSS framework:

   I. What do you understand about the term CSS and CSS framework?

   II. What are some examples of CSS frameworks?

   III. What are the Benefits of using CSS frameworks?

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 2.2.2. In addition, ask questions where necessary.

Key readings 2.2.2.: Description of CSS framework

CSS stands for Cascading Style Sheets. It is a style sheet language used to describe the presentation and appearance of a document written in HTML (Hypertext Markup Language) or XML (extensible Markup Language).

Inline Styles: You can apply CSS directly to individual HTML elements using the style attribute.

Internal Stylesheet: You can define CSS rules within the <style> tags in the <head> section of an HTML document. This method allows you to apply styles to multiple elements within the document.

External Stylesheet: You can link an external CSS file to an HTML document using the <link> tag. The CSS code resides in a separate file with a .css extension. This method allows you to keep your CSS code separate from the HTML file, making it easier to manage and reuse across multiple HTML pages.

How CSS works with HTML:

1. Selectors: CSS uses selectors to target specific HTML elements. Selectors can be based on element names, classes, IDs, attributes, or their relationships in the document structure.

2. Declarations: Once you've selected the desired elements, you define declarations to specify how they should be styled. Declarations consist of a property and a value. For example, you might set the color property to red and the font-size property to 16px.

3. Rule Sets: CSS declarations are grouped together into rule sets, which consist of a selector followed by a set of declarations enclosed in curly braces {}. Multiple rule sets can be combined to apply styles to different elements.

4. Linking CSS to HTML: CSS can be applied to an HTML document in several ways. You can use inline styles by adding the style attribute directly to an HTML tag. Alternatively, you can include CSS code within the <style> tags in the head section of an HTML document. Another common method is to link an external CSS file using the <link> tag, where the CSS code resides in a separate file with a .css extension.

A CSS framework is a collection of pre-written CSS styles, components, and utilities that help developers build websites and web applications more efficiently. These frameworks provide a consistent and standardized approach to styling and layout, saving time and effort by providing ready-to-use CSS classes and components.

Here are some popular CSS frameworks along with examples:

1. Bootstrap: Bootstrap is one of the most widely used CSS frameworks. It offers a comprehensive set of responsive CSS classes, pre-built UI components, and a responsive grid system. It includes styles for typography, buttons, forms, navigation bars, modals, and much more.

Example: Official Bootstrap website (https://getbootstrap.com/)

2. Foundation: Foundation is a flexible and customizable CSS framework. It provides a responsive grid system, styles for typography, buttons, forms, and a wide range of components such as navigation bars, tooltips, modals, and tabs. Foundation also includes utility classes for rapid prototyping.

Example: ZURB Foundation website (https://foundation.zurb.com/)

3. Bulma: Bulma is a lightweight and modern CSS framework. It focuses on simplicity and offers a responsive grid system, typography styles, form styles, buttons, cards, and various UI components. Bulma also provides a modular architecture and allows customization through SASS variables.

Example: Bulma CSS website (https://bulma.io/)

4. Tailwind CSS: Tailwind CSS is a utility-first CSS framework that provides a large set of utility classes that can be combined to build custom designs. It offers a highly customizable and low-level approach to styling, allowing developers to quickly prototype and create unique designs.

Example: Tailwind CSS website (https://tailwindcss.com/)

5. Material-UI: Material-UI is a CSS framework based on Google's Material Design guidelines. It provides a set of React components that implement the Material Design principles. Material-UI offers a rich collection of ready-to-use UI components, including buttons, cards, forms, navigation bars, and more.

Example: Material-UI website (https://mui.com/)

These CSS frameworks can significantly speed up development by providing pre-built styles and components, responsive layouts, and a standardized approach to design. You can choose a framework based on your project requirements and preferences, and leverage their features to create visually appealing and responsive web applications.

Benefits of using CSS Framework

1. Rapid Development
2. Consistency
3. Responsiveness
4. Reusability
5. Cross-browser Compatibility

Theoretical Activity 2.2.3: Description of reusable components and bootstrap in Vue JS framework

Tasks:

1. You are requested to answer the following questions related to the reusable components and bootstrap in Vue Js framework:

   I. What do you understand about Bootstrap?

   II. What are reusable components and how do they benefit the development process?

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 2.2.3. In addition, ask questions where necessary.

Key readings 2.2.3.: Description of reusable components and bootstrap in Vue JS framework

Reusable components in Vue.js are components that are designed to be easily reused in different parts of your application. They allow you to encapsulate a specific functionality or UI pattern into a self-contained unit that can be easily imported and used wherever needed.

Reusable components promote code reusability, maintainability, and consistency across your application.

Reusable components in Vue.js offer several benefits:

- Code Reusability
- Modularity and Maintainability
- Consistency and Standardization
- Scalability
- Collaboration
- Testing and Debugging
- Ecosystem and Community

Bootstrap is a popular CSS framework that provides a set of pre-designed UI components and styles to build responsive and visually appealing web applications. Vue.js can be easily integrated with Bootstrap to leverage its powerful CSS styling and ready-to-use components.

Practical Activity 2.2.4: Create reusable component in components folder

Task:

1. Read key reading 2.2.4 and ask clarification where necessary

2. Referring to the previous practical activities (2.1.2), you are requested to go to the computer lab to create reusable component in components folder. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create reusable component in components folder.

5. Referring to the steps provided on task 3, Create reusable component in components folder.

6. Present your work to the trainer and whole class

Key readings 2.2.4.: Create reusable component in components folder

To create a reusable component in the Vue.js framework and organize it within the components folder, you can follow these steps:

1.Create the Component File:

Inside your `src/components` directory, create a new file for your component. For example, if you want to create a button component, you can name it `MyButton.vue`.

2.Define the Component

In `MyButton.vue`, you can start by defining the template, script, and style sections. Here’s a simple example:

```
<template>
  <button :class="buttonClass" @click="handleClick">
    <slot></slot>
  </button>
</template>
<script>
export default {
  name: 'MyButton',
  props: {
    buttonClass: {
      type: String,
      default: 'default-button'
    }
  },
 methods: {
  handleClick() {
   this.$emit('click');
  }
 }
}
</script>
<style scoped>
.default-button {
 background-color: blue;
 color: white;
 padding: 10px;
 border: none;
 border-radius: 5px;
}
</style>
```

3.Use the Component

- To use your new component, you need to import it into the parent component where you want to use it. For example, in `App.vue`:

```
<template>
 <div>
  <MyButton buttonClass="custom-button" @click="handleButtonClick">
   Click Me!
  </MyButton>
 </div>
</template>
<script>
```

```
import MyButton from './components/MyButton.vue';

export default {

  components: {

    MyButton

  },

  methods: {

    handleButtonClick() {

      alert('Button was clicked!');

    }

  }

}

</script>

<style>

.custom-button {

  background-color: green;

}

</style>
```

4. Organize Components:

If you have several reusable components, consider organizing them into subfolders within the `components` directory based on their functionality or category. For example, you could have a `buttons` folder for all button-related components.

5. Documentation

It’s also a good practice to document your components, either in comments or in a separate markdown file, explaining their purpose, props, and usage examples.

Rules for naming a component in a Vue.js project:

1. Component name should consist of alphanumeric characters and hyphens
2. They must begin with an alphabet, not a number

3. Component name are case sensitive
4. Name should not contain uppercase letters
5. Name should be descriptive and indicative of the component’s purpose

To use the component in another Vue component, follow these steps:

1. Open the Vue component where you want to use "My_Component" such as app.vue.

2. Import the component at the top of the file using the relative path to the component file. **For example:**

```
<script>
import MyComponent from '../components/MyComponent.vue';
export default {
  components: {
    MyComponent,
  },
};
</script>
```

3. You can now use the "MyComponent" in the template section of your component. For example:

```
<template>
  <div>
    <h1>My App</h1>
    <my-component></my-component>
  </div>
</template>
```

Note: By creating a new component and using it in your Vue.js application, you can build modular and reusable UI elements. Remember to import and register the component in the necessary Vue components where you want to use it.

Practical Activity 2.2.5: Apply Bootstrap to Vue components

Task:

1. Read key reading 2.2.5 and ask clarification where necessary

2. Referring to the previous practical activities (2.2.4), you are requested to go to the computer lab to apply bootstrap to Vue components. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to apply bootstrap to Vue components.

5. Referring to the steps provided on task 3, apply bootstrap to Vue components.

6. Present your work to the trainer and whole class

Key readings 2.2.5.: Apply Bootstrap to Vue components

To apply Bootstrap to Vue components, you can follow these steps:

1.**Install Bootstrap:**

Make sure you have Bootstrap installed in your Vue.js project. If you haven't installed it yet, you can do so by running the following command in your project's root directory:

npm install bootstrap or yarn add bootstrap

This command installs the Bootstrap package and its dependencies.

Here's an example of how you can install Bootstrap in a Vue.js project:

1. Open your Vue.js project in your preferred command-line interface or terminal.

2. Navigate to the root directory of your Vue.js project.

3. Install Bootstrap using npm (Node Package Manager) by running the following command:

npm install bootstrap

This command will install the latest version of Bootstrap and its dependencies into your project.

4. Once the installation is complete, you can import and use Bootstrap in your Vue components.

For example, let's assume you have a component called "App.vue". Open the "App.vue" file in your code editor.

5. At the top of the "App.vue" file, import the Bootstrap CSS by adding the following line:

import 'bootstrap/dist/css/bootstrap.css';

This line imports the Bootstrap CSS file into your component.

6. Save the changes to the "App.vue" file.

Now, you have successfully installed Bootstrap in your Vue.js project. You can start using Bootstrap classes, components, and styles in your Vue components.

1. To use Bootstrap classes and components in your Vue templates, you can refer to the Bootstrap documentation and use the appropriate class names and component syntax.

For example, you can add Bootstrap classes to your HTML elements like this:

```
<template>
  <div>
    <h1 class="text-primary">Hello, Vue.js with Bootstrap!</h1>
    <button class="btn btn-primary">Click me</button>
  </div>
</template>
```

By installing and utilizing Bootstrap in your Vue.js project, you can enhance the styling and responsiveness of your application using Bootstrap's pre-built CSS and components

2. Import Bootstrap in the Main Project File:

a. Open the main project file, typically named main.js or main.ts.

b. Import Bootstrap by adding the following lines at the top of the file:

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap';

c. The first line imports the Bootstrap CSS styles, and the second line imports the Bootstrap JavaScript functionalities.

Here's an example of how you can import Bootstrap in the main project file of a Vue.js project:

1. Open your Vue.js project in your preferred code editor.

2. Locate the main project file, which is typically named "main.js" or "main.ts". This is the entry point of your Vue.js application.

3. Open the main project file in your code editor.

4. At the top of the main project file, import Bootstrap by adding the following line:

import 'bootstrap/dist/css/bootstrap.css';

This line imports the Bootstrap CSS file into your project.

5. Save the changes to the main project file.

Now, Bootstrap will be imported and applied globally to your Vue.js application.

By importing Bootstrap in the main project file, you ensure that the Bootstrap styles are available throughout your application. This allows you to use Bootstrap classes and components in any of your Vue components without the need to import Bootstrap in each individual component file.

3. Import Bootstrap in Vue Components:

a. Open the Vue component file where you want to apply Bootstrap styles.

b. Import Bootstrap by adding the following line at the top of the file:

import 'bootstrap/dist/css/bootstrap.css';

c. This line imports the Bootstrap CSS styles specifically for this component.

Here's an example of how you can import Bootstrap in Vue components:

1. Open the Vue component file where you want to import Bootstrap.

2. At the top of the component file, import Bootstrap by adding the following line:

import 'bootstrap/dist/css/bootstrap.css';

This line imports the Bootstrap CSS file into the specific Vue component.

3. Save the changes to the component file.

Now, Bootstrap will be imported and applied specifically to that Vue component.

By importing Bootstrap in a specific Vue component, you can use Bootstrap classes and components within that component's template. This allows you to have more control over where Bootstrap is applied in your application, as you can choose to import it only in the components that require it.

4. Use Bootstrap Classes in HTML Templates:

a. In your Vue component's HTML template, you can now use Bootstrap classes to style your elements.

b. Refer to the Bootstrap documentation (https://getbootstrap.com/docs) to find the appropriate classes for the desired styles.

c. Apply Bootstrap classes to the HTML elements as needed.

Here's an example of how you can use Bootstrap classes in HTML templates of Vue components:

1. Open the Vue component file where you want to use Bootstrap classes.

2. In the template section of the component, you can use Bootstrap classes to style your HTML elements.

For example, let's say you have a component called "MyComponent.vue". Open the "MyComponent.vue" file in your code editor.

3. In the template section of "MyComponent.vue", you can use Bootstrap classes to style your HTML elements.

Here's an example:

```
<template>
  <div>
    <h1 class="text-primary">Hello, Vue.js with Bootstrap! </h1>
    <button class="btn btn-primary">Click me</button>
    <div class="alert alert-success" role="alert">
      This is a success message using Bootstrap classes.
    </div>
  </div>
</template>
```

In this example, we have used the Bootstrap classes `text-primary` to style the heading, `btn btn-primary` to style the button, and `alert alert-success` to style the success message.

4. Save the changes to the component file.

Now, you can see that the HTML elements in the template section of your Vue component are styled using Bootstrap classes.

By using Bootstrap classes in your Vue component templates, you can leverage the pre-built styles and components provided by Bootstrap to enhance the appearance and functionality of your application. Remember to import Bootstrap CSS in the component or main project file to ensure that the Bootstrap classes are available for use.

5.Utilize Bootstrap Components:

a. Bootstrap provides various pre-built components that you can use in your Vue components.

b. Refer to the Bootstrap documentation for the available components and their usage examples.

c. Import the required Bootstrap components into your Vue component file by adding the appropriate import statements.

d. Use the imported Bootstrap components in your component's template section.

Here's an example of how you can utilize Bootstrap components in Vue templates:

1. Make sure you have imported Bootstrap CSS in your Vue project, as mentioned in the previous examples.

2. Open the Vue component file where you want to utilize Bootstrap components.

3. In the template section of the component, you can use Bootstrap components by including their corresponding HTML markup and classes.

For example, let's say you have a component called "MyComponent.vue". Open the "MyComponent.vue" file in your code editor.

4. In the template section of "MyComponent.vue", you can utilize Bootstrap components. **Here's an example:**

```
<template>
```

```
 <div>
  <h1>My Component</h1>
  <button class="btn btn-primary">Click me</button>
  <div class="alert alert-success" role="alert">
   This is a success message using Bootstrap classes.
  </div>
  <form>
   <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" id="name" v-model="name">
   </div>
   <div class="form-group">
    <label for="email">Email:</label>
    <input type="email" class="form-control" id="email" v-model="email">
   </div>
   <button type="submit" class="btn btn-primary">Submit</button>
  </form>
 </div>
</template>
<script>
export default {
 data() {
  return {
   name: '',
   email: '',
  };
 },
};

</script>
```

In this example, we have utilized Bootstrap components such as `btn`, `alert`, and `form-control`. We have used the `btn` class to style the button, the `alert` class to style the success message, and the `form-control` class to style the input fields within the form.

5. Save the changes to the component file.

Now, you can see that the Bootstrap components are utilized in the template section of your Vue component.

Practical Activity 2.2.6: Reuse components in multiple places

Task:

1. Read key reading 2.2.6 and ask clarification where necessary

2. Referring to the previous practical activities (2.2.5), you are requested to go to the computer lab to create components in Vue JS Framework in components folder and reuse the created component in other components such as app.vue. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create components in VueJS Framework in components folder.

5. Referring to the steps provided on task 3, create components in VueJS Framework in components folder.

6. Present your work to the trainer and whole class

Key readings 2.2.6.: Reuse components in multiple places

Reusing components in multiple places is a common practice in software development and design. It promotes code efficiency, maintainability, and consistency across different parts of a system.

Here's an example of reusing components in multiple places using a web development scenario:

Let's say you are building a web application that requires a consistent header component across multiple pages. Instead of duplicating the header code in each page, you can create a reusable header component and include it wherever needed.

Here's how it can be done:

1. Identify the header component: Analyze your design and determine the common elements that make up the header, such as the logo, navigation menu, and user profile information.

2. Extract the header component: Create a separate header component file or module that contains the HTML, CSS, and JavaScript code for the header. You can use a framework like React, Angular, or Vue.js to define the component structure and behaviour.

3. Generalize the component: Make the header component as generic as possible by removing any specific content or styling that is unique to a particular page. Focus on defining the structure and functionality that remains consistent across all pages.

4. Create a component library: Establish a component library within your project or as a separate repository where you store reusable components. This library should contain the header component along with any other reusable components you develop.

5. Document the component usage: Provide clear documentation on how to use the header component. Specify the required props or configurations, such as the logo image path, navigation links, or user profile data. Document any customization options or styling guidelines to ensure consistent usage.

6. Test and validate the component: Test the header component in different scenarios to ensure it functions correctly and adapts to various screen sizes or user interactions. Validate its responsiveness, accessibility, and compatibility across different browsers or devices.

7. Integrate the component into pages: In each web page that requires the header, import or include the header component from your component library. Pass the necessary props or configurations to customize the header based on the specific page requirements.

8. **Maintain and update the component:** As you continue to develop the web application, regularly review and update the header component as needed. Add new features, fix any issues, or enhance its design based on evolving requirements.

Example of reusing components in multiple places using the Vue.js framework with code snippets:

Let's say you have a reusable "Card" component that displays a title and content. You want to use this component in two different views: "Home" and "About". Here's how you can achieve this:

1. **Create the Card component:**

```
<template>
  <div class="card">
   <h2>{{ title }}</h2>
   <p>{{ content }}</p>
  </div>
</template>
<script>
export default {
  props: ['title', 'content']
}
</script>
<style scoped>
.card {
  /* Card styling */
}
</style>
```

2. **Register the Card component:**

```
// In your main.js or entry point file
import Vue from 'vue';
import Card from './components/Card.vue';
Vue.component('Card', Card);
```

3. **Use the Card component in the "Home" view:**

```
<template>
  <div>
   <h1>Home</h1>
   <Card title="Welcome" content="This is the homepage content"></Card>
  </div>
</template>
<script>
import Card from '../components/Card.vue';
export default {
  components: {
   Card
  }
}
</script>
```

4. **Use the Card component in the "About" view:**

```
<template>
  <div>
   <h1>About</h1>
   <Card title="About Us" content="This is the about page content"></Card>
  </div>
</template>
<script>
import Card from '../components/Card.vue';
export default {
  components: {
   Card
  }
}
</script>
```

In both the "Home" and "About" views, you can use the <Card> component by including it within the template. The component accepts title and content props that you can pass to customize the card's title and content.

The above example demonstrates how you can create a reusable component and use it in different views within your Vue.js application, promoting code reuse, maintainability, and consistency.

Points to Remember

- In Vue.js, components are the building blocks of a Vue application, allowing developers to encapsulate and reuse UI elements and functionality. There are three main parts of a Vue component include: Template, Script, Style.
- There are various CSS frameworks available to assist developers in creating responsive and visually appealing web designs efficiently. Some popular examples of CSS frameworks include Bootstrap, Foundation, Bulma, Tailwind CSS, Materialize CSS, Semantic UI, and UIKit.
- In Vue.js, reusable components are a fundamental concept that allows developers to create modular, self-contained elements that can be easily reused across different parts of a Vue application. By leveraging reusable components in Vue.js, developers can achieve several benefits, including: Code Reusability, Modularity and Maintainability, Consistency and Standardization, Scalability.
- To create a reusable component in the Vue.js framework and organize it within the components folder, you can follow these steps:
    1. Define the Component:
    2. Customize the Component
    3. Export and Use the Component
    4. Save the component file
    5. Use the Component in Other Files
    6. Use the component within the template section of the file
- To apply Bootstrap to Vue components, you can follow these steps:

1. Install Bootstrap
2. Import Bootstrap in Vue Components
3. Use Bootstrap Classes in HTML Templates
4. Utilize Bootstrap Components

- Let's say you are building a web application that requires a consistent header component across multiple pages. Instead of duplicating the header code in each page, you can create a reusable header component and include it wherever needed.

  Here's how it can be done:

  - Identify the header component
  - Extract the header component
  - Generalize the component
  - Create a component library
  - Document the component usage
  - Test and validate the component
  - Integrate the component into pages
  - Maintain and update the component

Application of learning 2.2.

You have given task of building a web application that requires a consistent header component across multiple pages. As developer who have skills in Vue js framework create a header component in components folder and include it in other pages like Home.vue, About us.vue and Contacts.vue.

Indicative content 2.3: Apply Navigation in Vue Project Using Router

Duration: 5 hrs

Theoretical Activity 2.3.1: Description of Router and route in Vue framework

Tasks:

1. You are requested to answer the following questions related to the Router and route in Vue framework:

   I. What are the differences between Vue's route and router?

   II. What are the key concepts and features associated with Vue Router?

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 2.3.1. In addition, ask questions where necessary.

Key readings 2.3.1.: Description of Router and route in Vue framework

In the Vue framework, the **Vue Router** is a powerful routing library that allows you to implement client-side navigation in your single-page applications (SPAs).

It provides a way to define and manage different routes within your application, allowing users to navigate between different views or components without requiring a full page reload.

The Vue Router works by mapping URLs to specific components, enabling dynamic content rendering based on the current URL. It provides a declarative syntax to define routes and seamlessly integrates with the Vue.js ecosystem.

Here are some key features and concepts related to the Vue Router:

1. Router instance: To use the Vue Router, you need to create a router instance using new VueRouter(). This instance serves as the central configuration point for your application's routes.

2. Routes: Routes define the mapping between URLs and components. Each route consists of a URL pattern and a corresponding component that should be rendered when the URL matches the pattern. Routes can also have additional configuration options, such as route parameters, query parameters, and route guards.

3. Route Components: Components are associated with each route and define the content to be rendered when the route is active. These components can be created separately or inline using the component property of the route definition.

4. Router View: The <router-view> component is used to render the matched route component based on the current URL. It acts as a placeholder where the appropriate component is dynamically inserted based on the active route.

5. Navigation: The Vue Router provides programmatic navigation methods that allow you to navigate to different routes programmatically. You can use methods like router.push(), router.replace(), and router.go() to navigate to a specific URL or manipulate the browser's history.

6. Route Parameters: Route parameters allow you to define dynamic segments in the URL. For example, a route pattern like /users/:id can match URLs like /users/1 or /users/42, with the :id segment being passed as a parameter to the corresponding component.

7. Route Guards: Route guards are functions that can be applied to routes to control access or perform actions before or after navigation. They allow you to guard routes based on conditions, such as user authentication or data loading. Route guards include beforeEach, beforeResolve, afterEach, and per-route guards.

8. Nested Routes: The Vue Router supports nested routes, where components can have their own sub-routes. This allows you to create complex nested navigation structures within your application.

Difference between Vue Router and a route.

Vue Router: The Vue Router is a library in Vue.js that provides navigation and routing capabilities for single-page applications (SPAs).

It is responsible for managing the application's routes, handling navigation between different views or components, and updating the content based on the current URL. The Vue Router is an instance of the VueRouter class that you create and configure in your application. It acts as the central point for defining routes, navigating programmatically, and controlling the overall routing behavior of your application.

Route:

- A route, on the other hand, is an individual entry in the routing configuration defined by the Vue Router.
- It represents a specific URL pattern and is associated with a component to be rendered when the URL matches that pattern.
- **A route defines the relationship between** a URL and a component, specifying which component should be displayed when the URL is accessed.
- Each route can have additional properties such as route parameters, query parameters, and route guards.

Theoretical Activity 2.3.2: Description of declarative navigation

Tasks:

1. You are requested to answer the following questions related to the declarative navigation:

   I. What do you understand about declarative navigation in Vue framework?

   II. What are the advantages of using declarative navigation in Vue.js applications?

   III. Provide examples of implementing navigation in the Vue framework

2. Provide the answer of asked questions by writing them on paper.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 2.3.2. In addition, ask questions where necessary.

Key readings 2.3.2.: Description of declarative navigation

Declarative navigation in the Vue.js framework refers to the approach of defining and managing navigation in a declarative manner using the Vue Router.

It allows you to specify the desired navigation behavior and destination using template directives or router components, without explicitly writing imperative code for navigation.

In declarative navigation, you define links or buttons that trigger navigation by specifying the target route or URL.

Advantages of declarative navigation

Clarity: Declarative navigation makes it easier to understand the high-level structure of an application or user interface, as it clearly defines the available routes or states.

Modularity: It promotes a modular and component-based approach to UI development, as you can define how components relate to each other through declarative navigation.

Maintainability: When navigation is declared separately from the actual user interface components, it's easier to make changes and updates to the navigation logic without affecting the core components.

Testing: Declarative navigation can simplify testing, as you can easily verify that the navigation flows are correctly defined and functioning as expected.

The Vue Router takes care of handling the navigation behind the scenes based on the provided configuration.

Here are a few examples of declarative navigation in Vue:

1. <router-link> component: The <router-link> component is a built-in component provided by the Vue Router. It renders an anchor tag (<a>) that automatically updates the URL and triggers navigation to the specified route. You can use it like this:

<router-link to="/home">Home</router-link>

In this example, clicking the "Home" link will navigate to the /home route, as defined in the router configuration.

2. Dynamic route parameters: Declarative navigation allows you to pass dynamic route parameters as part of the URL. For example:

<router-link :to="`/users/${userId}`">User Profile</router-link>

In this case, the userId is a variable that represents a specific user's ID. Clicking the link will navigate to a route that includes the dynamic userId in the URL.

3. Programmatic navigation with methods: In addition to declarative navigation using components, Vue Router also provides programmatic navigation methods. These methods allow you to navigate imperatively from within your Vue components or methods. For example:

```
// In a Vue component

methods: {

  navigateToHome() {

    this.$router.push('/home');

  }

}
```

In this example, the navigateToHome method triggers navigation to the /home route when called.

Theoretical Activity 2.3.3: Description of parameters used inside the router

Tasks:

1. You are requested to answer the following questions related to the parameters used inside the router:

I. Explain the concept of 'Router' in Vue.js.

2. Provide the answer of asked questions by writing them on paper.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 2.3.3. In addition, ask questions where necessary.

Key readings 2.3.4.: Description of parameters used inside the router

In the Vue Router, there are various parameters that can be used to configure and customize the behavior of routes. These parameters allow you to pass dynamic values, query parameters, and apply route-specific guards.

Examples of Parameters inside the router:

Dynamic Route Parameters: Dynamic route parameters allow you to define segments in the URL that can be dynamically captured and passed to the associated component. They are denoted by a colon (:) followed by the parameter name in the route path.

Example:

```
{
  path: '/users/:id',
  component: User,
  props: true
}
```

In this example, the :id parameter represents a dynamic segment in the URL. When the user navigates to a URL like /users/123, the value 123 will be captured as the id parameter and passed as a prop to the User component.

Query Parameters: Query parameters allow you to pass additional data as key-value pairs in the URL. The first query parameter is denoted by a question mark(?) whereas the rest of other parameters are denoted by ampersand (&) sign.

Example:

```
{
  path: '/search',
  component: Search,
  props: route => ({ query: route.query.q })
}
```

In this example, when the user navigates to a URL like /search?q= keyword&, the query parameter q with the value keyword will be available as route.query.q. You can access and use this query parameter within the Search component.

Route Guards: Route guards are functions that allow you to guard routes and perform actions before or after navigation.

They can be used to control access to routes, load data, or perform authentication checks. The Vue Router provides different types of route guards, including

- beforeEach
- beforeResolve
- afterEach, and
- per-route guards.

For example, the beforeEach guard is commonly used for authentication checks before navigating to a specific route:

```
router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !isAuthenticated()) {
    next('/login');
  } else {
    next();
  }
});
```

In this example, the beforeEach guard checks if the destination route requires authentication (to.meta.requiresAuth) and if the user is authenticated. If the user is not authenticated, they are redirected to the login page (next('/login')).

Practical Activity 2.3.4: Install router and create router instance

Task:

1. Read key reading 2.3.4 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.3.1) you are requested to go to the computer lab to install Router in Vue JS and create a Vue router instance in the created Vue router package. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to install Router in Vue JS and create a Vue router instance in the created Vue router package.

5. Referring to the steps provided on task 3, install Router in Vue JS and create a Vue router instance in the created Vue router package.

6. Present your work to the trainer and whole class

Key readings 2.3.4.: Install router and create router instance

To install the Vue Router and create a router instance in the Vue.js framework, you'll need to follow these steps:

1. Install the Vue Router package: Open your project's terminal or command prompt and navigate to the root directory of your Vue.js project. Then, run the following command to install the Vue Router package using npm:

```
npm install vue-router
```

Alternatively, if you prefer using Yarn, you can run the following command:

```
yarn add vue-router
```

This will install the Vue Router package and its dependencies in your project.

Here are the steps to install the Vue Router package in a Vue.js project:

1. Open your Vue.js project in your preferred command-line interface or terminal.

2. Navigate to the root directory of your Vue.js project.

3. Install the Vue Router package using npm (Node Package Manager) by running the following command:

```
npm install vue-router
```

This command will install the latest version of the Vue Router package and its dependencies into your project.

4. Once the installation is complete, you can import and use the Vue Router in your project.

To use the Vue Router in your Vue.js application, you need to configure it and set up your routes. Typically, this is done in the main project file, such as "main.js" or "main.ts".

5. Open the main project file in your code editor.

6. Import Vue and VueRouter at the top of the file:

```
import Vue from 'vue';

import VueRouter from 'vue-router';
```

7. Use the VueRouter plugin by adding the following line:

```
Vue.use(VueRouter);
```

8. Save the changes to the main project file.

Now, you have successfully installed the Vue Router package in your Vue.js project. You can proceed with creating a router instance, defining routes, and using the router in your Vue components.

Note: Remember to import the Vue Router package and configure it in the main project file to ensure that the router is properly integrated into your Vue application.

Here's an example of how you can set up the Vue Router in the main project file:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
// Import your Vue components for routing
import Home from './components/Home.vue';
import About from './components/About.vue';
import Contact from './components/Contact.vue';
Vue.use(VueRouter);
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/contact', component: Contact },
];
const router = new VueRouter({
  routes,
  mode: 'history', // Use 'history' mode for cleaner URLs (optional)
});
new Vue({
  router,
  render: (h) => h(App),
}).$mount('#app');
```

In this example, we import Vue and VueRouter, as well as the components that will be used for routing (Home, About, and Contact). We then use `Vue.use(VueRouter)` to install the Vue Router plugin. Next, we define an array of route objects, where each object represents a route with a corresponding path and component. Finally, we create a new Vue instance with the router configuration and mount it to the root element of our application.

After installing the Vue Router package and setting up the router, you can define and navigate to routes within your Vue components using router-link and router-view components.

Remember to import the Vue Router package in your project files where you need to use it.

1. Create a router instance:

In your application's entry point file (usually main.js), import the necessary modules and create a router instance.

1. Open your Vue.js project in your preferred code editor.

2. Open the main project file, such as "main.js" or "main.ts".

2. Import Vue and VueRouter at the top of the file:

import Vue from 'vue';

import VueRouter from 'vue-router';

3. Import your Vue components that will be used as routes.

For example:

import Home from './components/Home.vue';

import About from './components/About.vue';

import Contact from './components/Contact.vue';

Make sure to update the paths according to your project structure.

4. Use the VueRouter plugin by adding the following line:

Vue.use(VueRouter);

5. Create an array of route objects, where each object represents a route with a corresponding path and component. For example:

```
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/contact', component: Contact },
];
```

6. Create a new instance of VueRouter by passing the routes array to the constructor, and assign it to a variable. For example:

```
const router = new VueRouter({
  routes,
});
```

You can also specify additional configuration options for the router, such as the mode (e.g., 'history' for cleaner URLs).

7. Create a new Vue instance and pass the router instance as an option:

```
new Vue({
  router,
}).$mount('#app');
```

Make sure to replace `#app` with the ID or selector of the root element in your HTML file where the Vue app will be mounted.

8. Save the changes to the main project file.

Here's an example of how you can create a router instance in a Vue.js project using the Vue Router package:

1. Open your Vue.js project in your preferred code editor.

2. Open the main project file, such as "main.js" or "main.ts".

3. Import Vue and VueRouter at the top of the file:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
```

4. Import your Vue components that will be used as routes. For example:

```
import Home from './components/Home.vue';
import About from './components/About.vue';

import Contact from './components/Contact.vue';
```

Make sure to update the paths according to your project structure.

5. Use the VueRouter plugin by adding the following line:

```
Vue.use(VueRouter);
```

6. Create an array of route objects, where each object represents a route with a corresponding path and component. For example:

```
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/contact', component: Contact },
];
```

7. Create a new instance of VueRouter by passing the routes array to the constructor, and assign it to a variable. For example:

```
const router = new VueRouter({
  routes,
});
```

You can also specify additional configuration options for the router, such as the mode (e.g., 'history' for cleaner URLs).

8. Create a new Vue instance and pass the router instance as an option:

```
new Vue({
  router,
  render: (h) => h(App),
}).$mount('#app');
```

Make sure to replace `App` with the root component of your Vue application.

9. Save the changes to the main project file.

Now, you have created a router instance using the Vue Router package in your Vue.js project. The router instance will handle the routing functionality and allow you to navigate between different components based on the defined routes.

Remember to import the Vue Router package and create the router instance in the main project file to ensure that the router is properly configured and integrated into your Vue application.

Practical Activity 2.3.5: Importing component into router instance

Task:

1. Read key reading 2.3.5 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.3.1), you are requested to go to the computer lab to import the components into router instance. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to Import the components into router instance.

5. Referring to the steps provided on task 3, Import the components into router instance.

6. Present your work to the trainer and whole class

Key readings 2.3.5.: Importing component into router instance

To import and use components in a Vue Router instance, you need to follow these steps:

A. **Import the components:** In your router configuration file (usually router.js), import the components that you want to use in your routes.

Here are the steps to import components in a Vue.js project:

1. Open the file where you want to import the components. This could be a Vue component file, such as "App.vue", or any other file where you want to use the components.

2. At the top of the file, import the components you want to use. For example:

```
import MyComponent from './components/MyComponent.vue';

import AnotherComponent from './components/AnotherComponent.vue';
```

Make sure to update the paths according to your project structure.

3. Save the changes to the file.

Now, you have successfully imported the components into your Vue.js project. You can use these components in your Vue templates or other parts of your application.

Here's an example:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from './components/Home.vue';
import About from './components/About.vue';
import Contact from './components/Contact.vue';
Vue.use(VueRouter);
// Create the router instance
const router = new VueRouter({
  routes: [
    { path: '/', component: Home },
    { path: '/about', component: About },
    { path: '/contact', component: Contact }
  ]
});
export default router;
```

In this example, the Home, About, and Contact components are imported into the router configuration file. Make sure to provide the correct paths to your component files.

B. **Use the components in the router configuration:** Inside the router configuration, you can use the imported components by referencing them in the component property of each route.

To use components in the router configuration of a Vue.js project, follow these steps:

1. Open the main project file

2. Import the Vue and VueRouter at the top of the file:

```
import Vue from 'vue';

import VueRouter from 'vue-router';
```

3. Import the Vue components that you want to use as routes. For example:

```
import Home from './components/Home.vue';

import About from './components/About.vue';

import Contact from './components/Contact.vue';
```

Make sure to update the paths according to your project structure.

4. Use the VueRouter plugin by adding the following line:

```
Vue.use(VueRouter);
```

5. Create an array of route objects, where each object represents a route with a corresponding path and component. For example:

```
const routes = [

  { path: '/', component: Home },

  { path: '/about', component: About },

  { path: '/contact', component: Contact },

];
```

In this example, the Home component is associated with the root path ("/"), the About component is associated with the "/about" path, and the Contact component is associated with the "/contact" path.

6. Create a new instance of VueRouter by passing the routes array to the constructor, and assign it to a variable. For example:

```
const router = new VueRouter({

  routes,

});
```

7. Create a new Vue instance and pass the router instance as an option:

```
new Vue({

  router,
}).$mount('#app');
```

Make sure to replace `#app` with the ID or selector of the root element in your HTML file where the Vue app will be mounted.

8. Save the changes to the main project file.

Here's how you can do it:

```
const router = new VueRouter({
  routes: [
    { path: '/', component: Home },
    { path: '/about', component: About },
    { path: '/contact', component: Contact }
  ]
});
```

In this example, the imported components (Home, About, and Contact) are used as the values for the component property of each route. This ensures that the respective component will be rendered when the corresponding route is accessed.

C. **Use the router instance in your Vue app:** In your application's entry point file (usually main.js), import the router instance and use it when creating the Vue app.

To use the router instance in your Vue.js app, follow these steps:

1. Open the main project file, such as "main.js" or "main.ts", in your code editor.

2. Import the Vue and VueRouter at the top of the file:

```
import Vue from 'vue';

import VueRouter from 'vue-router';
```

3. Import the Vue components that you want to use as routes. For example:

```
import Home from './components/Home.vue';

import About from './components/About.vue';

import Contact from './components/Contact.vue';
```

Make sure to update the paths according to your project structure.

4. Use the VueRouter plugin by adding the following line:

```
Vue.use(VueRouter);
```

5. Create an array of route objects, where each object represents a route with a corresponding path and component. For example:

```
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/contact', component: Contact },
];
```

6. Create a new instance of VueRouter by passing the routes array to the constructor, and assign it to a variable. For example:

```
const router = new VueRouter({
  routes,
});
```

7. Create a new Vue instance and pass the router instance as an option:

```
new Vue({
  router,
}).$mount('#app');
```

Make sure to replace `#app` with the ID or selector of the root element in your HTML file where the Vue app will be mounted.

8. Now, in your Vue components, you can use the router instance to navigate between routes and access route information.

For example, you can use the `<router-link>` component to create navigation links:

```
<router-link to="/">Home</router-link>
<router-link to="/about">About</router-link>
<router-link to="/contact">Contact</router-link>
```

And you can use the `<router-view>` component to render the component associated with the current route:

<router-view></router-view>

9. Save the changes to the main project file.

Now, you have successfully used the router instance in your Vue.js app. You can navigate between routes using `<router-link>` components and display the corresponding components using `<router-view>`.

Remember to import the components, configure the router, and use the router instance in the main project file to ensure that the router is properly integrated into your Vue application and can be used for navigation.

Here's an example:

```
import Vue from 'vue';

import App from './App.vue';

import router from './router';

new Vue({

  router,

  render: h => h(App)

}).$mount('#app');
```

In this example, the imported router instance is passed to the router option of the Vue app during initialization. This makes the router available throughout the Vue app, allowing you to use its navigation capabilities and render the appropriate components based on the defined routes.

Practical Activity 2.3.6: Passing router into Vue instance

Task:

1. Read key reading 2.3.6 and ask clarification where necessary

2. Referring to the previous practical activities (2.3.5) you are requested to go to the computer lab to pass router into Vue instance. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to pass router into Vue instance.

5. Referring to the steps provided on task 3, pass router into Vue instance

6. Present your work to the trainer and whole class

Key readings 2.3.6.: Passing router into Vue instance

To pass the Vue Router instance into the Vue app, you need to follow these steps:

Import the required modules: In your application's entry point file (usually main.js), import the necessary modules for Vue and the Vue Router. Here's an example:

```
import Vue from 'vue';

import App from './App.vue';

import VueRouter from 'vue-router';

Vue.use(VueRouter);
```

In this example, the Vue and VueRouter modules are imported.

Import and configure the routes:

Next, import the router configuration file (usually router.js) that defines your routes.

Here's an example:

```
import router from './router';
```

Ensure that you provide the correct path to your router configuration file.

Create the Vue app: Finally, create the Vue app instance and pass the router to the router option.

Here's an example:

```
new Vue({

  router,

  render: h => h(App)

}).$mount('#app');
```

In this example, the router instance is passed to the router option of the Vue app. This makes the router available throughout the Vue app and allows you to utilize its navigation capabilities.

Set up the template to use the router views: In your Vue component template, use the <router-view> component to define the area where the router will render the appropriate components based on the current route. Here's an example:

```
<template>
  <div>
    <router-view></router-view>
  </div>
</template>
```

The <router-view> component serves as a placeholder that will be replaced with the component corresponding to the current route.

Practical Activity 2.3.7: Create navbar reusable component

Task:

1. Read key reading 2.3.7 and ask clarification where necessary

2. Referring to the previous practical activities (2.2.6), you are requested to go to the computer lab to create a reusable navigation bar (navbar) component in Vue.js. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create a reusable navigation bar (navbar) component in Vue.js.

5. Referring to the steps provided on task 3, Create a reusable navigation bar (navbar) component in Vue.js.

6. Present your work to the trainer and whole class

Key readings 2.3.7.: Create navbar reusable component

To create a reusable navigation bar (navbar) component in Vue.js, you can follow these steps:

A. **Create a new Vue component file**: In your project's component directory, create a new file called Navbar.vue (or any other appropriate name). This file will contain the code for your navbar component.

B. Define the Navbar component: Open the Navbar.vue file and define your navbar component.

1. Open your Vue.js project in your preferred code editor.

2. Navigate to the directory where you want to create the Navbar component. Typically, components are stored in a "components" directory within the project structure.

3. Create a new file with a ".vue" extension for the Navbar component. For example, you can name it "Navbar.vue".

4. Open the newly created file in your code editor.

5. In the file, start by adding a template section where you define the HTML structure of the Navbar component.

For example:

Create navbar component that contains the following elements: Home, Registration, List, About us and Contact us

```
<template>
  <nav class="navbar">
    <ul class="navbar-list">
      <li class="navbar-item"><a href="#home">Home</a></li>
      <li class="navbar-item"><a href="#registration">Registration</a></li>
      <li class="navbar-item"><a href="#list">List</a></li>
      <li class="navbar-item"><a href="#about-us">About Us</a></li>
      <li class="navbar-item"><a href="#contact-us">Contact Us</a></li>
    </ul>
  </nav>
</template>
```

6. **Next, add a script** section where you define the JavaScript logic of the Navbar component.

For example:

```
<script>
export default {
  name: 'Navbar',
};
</script>
```

In this example, we define the component's name as "Navbar" using the `name` property. You can replace it with the desired name for your Navbar component.

7. Optionally, you can also add a style section where you define the CSS styles specific to the Navbar component.

For example:

```
<style scoped>
.navbar {
  background-color: #333;
  padding: 10px;
}
.navbar-list {
  list-style-type: none;
  display: flex;
  justify-content: space-around;
  margin: 0;
  padding: 0;
}
.navbar-item a {
  color: white;
  text-decoration: none;
  padding: 10px 20px;
  display: block;
  transition: background-color 0.3s ease;
}
.navbar-item a:hover {
  background-color: #555;
}
</style>
```

The `scoped` attribute ensures that the styles defined within this section only apply to the current component and do not affect other components.

8. **Save the changes to the Navbar component file.**

Now, you have successfully defined the Navbar component. You can use this component in other parts of your Vue application by importing it and including it in other Vue templates.

Remember to follow the Vue component structure with the template, script, and style sections to create reusable and modular components in your Vue.js project.

Here's a basic example:

```
<template>
  <nav>
    <ul>
      <li><router-link to="/">Home</router-link></li>
      <li><router-link to="/about">About</router-link></li>
      <li><router-link to="/contact">Contact</router-link></li>
    </ul>
  </nav>
</template>
```

```
<script>

export default {

  name: 'Navbar',

};

</script>

<style scoped>

/* Add styling for the navbar component */

</style>
```

In this example, the <nav> element represents the navbar container. Inside the container, <ul> and <li> elements are used to create a list of navigation links. The <router-link> component is used to create clickable links that navigate to different routes defined in your Vue Router.

Note that the name property is set to 'Navbar' to give the component a name.

C. Import and use the Navbar component: In the parent component or the entry point file (main.js), import the Navbar component and include it in the template where you want to display the navbar.

To import and use the Navbar component in your Vue.js project, follow these steps:

1. Open the Vue component file where you want to use the Navbar component. For example, let's assume you want to use it in the "App.vue" component.

2. At the top of the file, import the Navbar component using the relative path to the component file. For example:

```
import Navbar from './components/Navbar.vue';
```

Make sure to update the path based on the location of your Navbar component file.

1. In the same component file, include the Navbar component within the template section where you want it to be rendered.

For example, you can include it at the top of the App.vue template:

```
<template>

  <div id="app">
```

;

```
    <navbar></navbar>

    <!-- Rest of your component's HTML code goes here -->

  </div>

</template>
```

Here, `<navbar></navbar>` is the custom element representing the Navbar component. You can use any name you like for the custom element, but it should match the component's name defined in the Navbar component file.

4. **Save the changes to the component file.**

Now, you have successfully imported and used the Navbar component in your Vue.js project. The Navbar component will be rendered in the specified location within the parent component's template.

Make sure to import the Navbar component at the top of the component file where you want to use it, and include the component's custom element within the template section. This allows Vue.js to recognize and render the Navbar component correctly.

Here's an example:

```
<template>

  <div>

    <navbar></navbar>

    <!-- Rest of your application content -->

  </div>

</template>

<script>

import Navbar from './components/Navbar.vue';

export default {

  components: {

    Navbar,

  },

};
```

```
</script>
```

In this example, the Navbar component is imported and registered as a child component within the parent component. You can then use the <navbar> custom element in the template to render the navbar component.

D.**Customize the navbar styling and behavior:** You can customize the navbar component by adding CSS styles to the <style> section of the Navbar.vue file. Additionally, you can modify the content of the navbar links or add any desired behavior to the navigation links.

To customize the styling and behavior of the Navbar component in your Vue.js project, follow these steps:

1. Open the Navbar component file (e.g., "Navbar.vue") in your code editor.

2. To customize the styling, locate the style section within the component file. This is where you can modify the CSS styles specific to the Navbar component. For example:

```
<style scoped>
nav {
  background-color: #f0f0f0;
  padding: 10px;
}
ul {
  list-style-type: none;
  display: flex;
  justify-content: space-between;
}
li {
  margin-right: 10px;
}
a {
  text-decoration: none;
  color: #333;
  font-weight: bold;
}
</style>
```

Here, you can modify the CSS properties to change the appearance of the Navbar. You can update the background color, padding, font styles, and more to match your desired design.

3. To customize the behavior, you can add event handlers or modify the component's JavaScript logic. For example, you can add a click event handler to perform an action when a navigation link is clicked. Modify the script section of the component file accordingly. Here's an example:

```
<script>
export default {
  name: 'Navbar',
  methods: {
    handleClick() {
      // Perform action when a navigation link is clicked
      console.log('Navigation link clicked!');
    },
  },
}
</script>
```

In this example, a `handleClick` method is added to the component. You can define your own logic within this method to perform the desired action when a navigation link is clicked. For instance, you can navigate to a different route, update component data, or trigger other functions.

4. Save the changes to the Navbar component file.

Practical Activity 2.3.8: Using nested routes

Task:

1. Read key reading 2.3.8 and ask clarification where necessary

2. Referring to the previous practical activities (2.3.7), you are requested to go to the computer lab to use the nested routes in your application. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to use the nested routes in your application.

5. Referring to the steps provided on task 3, Use the nested routes in your application

6. Present your work to the trainer and whole class

Key readings 2.3.8.: Using nested routes

To use nested routes in Vue.js, you can follow these steps:

Set up the parent route: In your router configuration file (usually router.js), define a parent route that will serve as the container for the nested routes.

Here's an example:

```
import Vue from 'vue';

import VueRouter from 'vue-router';

import Child1Component from './components/Child1Component.vue';

import Child2Component from './components/Child2Component.vue';

Vue.use(VueRouter);

const router = new VueRouter({

  routes: [

    {

      path: '/parent',

      component: ParentComponent,
   children: [
    { path: 'child1', component: Child1Component },
    { path: 'child2', component: Child2Component },
   ],
  },
 ],
});
export default router;
```

In this example, the /parent route is defined as the parent route. The ParentComponent is the component that will serve as the container for the nested routes. The children property is an array of nested routes, each defined with a path and a corresponding component.

Create the parent component: Create a Vue component file for the parent component (ParentComponent in the example above). This component will be responsible for rendering the nested routes. Here's a basic example:

```
<template>
 <div>
  <h1>Parent Component</h1>
  <router-view></router-view>
 </div>
</template>
<script>
export default {
 name: 'ParentComponent',
};
</script>
<style scoped>
/* Add styling for the parent component */
</style>
```

In this example, the parent component template contains a <h1> heading and a <router-view> component. The <router-view> component will render the nested routes based on the current route.

Create the child components: Create Vue component files for the child components (Child1Component and Child2Component in the example above). These components will be rendered within the parent component based on the corresponding nested routes. Customise the child components according to your needs.

Use the nested routes in your application: In your application's templates or components, you can use the nested routes by navigating to the parent route and then appending the child route to it. Here's an example:

```
<template>
 <div>
  <!-- Some content -->
  <router-link to="/parent/child1">Go to Child 1</router-link>
  <router-link to="/parent/child2">Go to Child 2</router-link>
  <!-- Rest of your application content -->
 </div>
</template>
```

In this example, the <router-link> components are used to create links to the child routes (child1 and child2) under the parent route (/parent).

Practical Activity 2.3.9: Using dynamic router

Task:

1. Read key reading 2.3.8 and ask clarification where necessary

2. Referring to the previous practical activities (2.3.8), you are requested to go to the computer lab to use the dynamic router. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to use the dynamic router.

5. Referring to the steps provided on task 3, Use the dynamic router.

6. Present your work to the trainer and whole class

Key readings 2.3.9.: Using dynamic router

To use dynamic routing in Vue.js, you can follow these steps:

A. Set up the dynamic route in the router configuration: In your router configuration file (usually router.js), define a dynamic route by using a route parameter.

1. Open the main project file, such as "main.js" or "main.ts", in your code editor.

2. Import the Vue and VueRouter at the top of the file:

```
import Vue from 'vue';

import VueRouter from 'vue-router';
```

3. Import the Vue components that you want to use as routes. For example:

```
import Home from './components/Home.vue';

import About from './components/About.vue';

import Contact from './components/Contact.vue';

import User from './components/User.vue';
```

Make sure to update the paths according to your project structure.

4. Use the VueRouter plugin by adding the following line:

```
Vue.use(VueRouter);
```

5. Create an array of route objects, where each object represents a route with a corresponding path and component. For example:

```
const routes = [

  { path: '/', component: Home },

  { path: '/about', component: About },

  { path: '/contact', component: Contact },

  { path: '/user/:id', component: User }, // Dynamic route with parameter ':id'

];
```

In this example, the User component is associated with the "/user/:id" path, where ":id" represents a dynamic parameter that can be passed in the URL.

6. Create a new instance of VueRouter by passing the routes array to the constructor, and assign it to a variable. For example:

```
const router = new VueRouter({
  routes,
});
```

7. Create a new Vue instance and pass the router instance as an option:

```
new Vue({
  router,
}).$mount('#app');
```

Make sure to replace `#app` with the ID or selector of the root element in your HTML file where the Vue app will be mounted.

8. Save the changes to the main project file.

Here's an example:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
Vue.use(VueRouter);
const router = new VueRouter({
  routes: [
    {
      path: '/users/:id',
      component: UserComponent,
      props: true
    },
  ],
});
export default router;
```

In this example, the /users/:id route is defined as a dynamic route. The :id portion is a route parameter that can capture any value specified in the URL. The UserComponent is the component that will be rendered for this route. The props: true option enables passing the route parameters as props to the component.

B. Create the dynamic component: Create a Vue component file for the dynamic component (UserComponent in the example above). This component will be responsible for rendering the content based on the dynamic route parameter.

1. Open the Vue component file where you want to create the dynamic component.

2. In the template section, define a placeholder element where the dynamic component will be rendered. For example:

```
<template>
 <div>
  <component :is="dynamicComponent"></component>
 </div>
</template>
```

In this example, we use the `<component>` element with the `:is` attribute to bind the dynamic component. The `dynamicComponent` property will hold the name of the component to be rendered dynamically.

3. In the script section, define the dynamicComponent data property with an initial value representing the component name you want to render. For example:

```
<script>
export default {
 data() {
  return {
   dynamicComponent: 'ComponentA',
  };
 },
}
</script>
```

In this example, we set the initial value of `dynamicComponent` to `'ComponentA'`. You can replace it with the desired component name.

4. Create the component files for the dynamic components you want to render. For example, create "ComponentA.vue" and "ComponentB.vue" files.

5. Import the dynamic component files at the top of the component file where you created the dynamic component. For example:

```
import ComponentA from './ComponentA.vue';
import ComponentB from './ComponentB.vue';
```

Make sure to update the paths based on the location of your dynamic component files.

6. Register the imported components within the components object of the Vue component. For example:

```
components: {
  ComponentA,
  ComponentB,
},
```

7. Optionally, you can add a method or event handler to dynamically switch the component being rendered. For example, you can add a button that triggers a method to switch between ComponentA and ComponentB. Modify the script section of the component file accordingly.

Here's an example:

```
methods: {
  switchComponent() {
    this.dynamicComponent = this.dynamicComponent === 'ComponentA' ? 'ComponentB' : 'ComponentA';
  },
},
```

In this example, the `switchComponent` method toggles between ComponentA and ComponentB by updating the value of `dynamicComponent`.

8. Save the changes to the component file.

Here's a basic example:

```
<template>
  <div>
    <h1>User Details</h1>
    <p>User ID: {{ id }}</p>
  </div>
</template>
<script>
export default {
  name: 'UserComponent',
  props: ['id'],
};
</script>
<style scoped>
/* Add styling for the dynamic component */
</style>
```

In this example, the component template displays the user details with the captured id prop. The props option is used to specify the prop name.

3**.Use the dynamic route in your application:** In your application's templates or components, you can use the dynamic route by navigating to a URL that includes the route parameter value.

To use the dynamic route in your Vue.js application, follow these steps:

1. Open the Vue component file where you want to use the dynamic route.

2. Import the VueRouter at the top of the file:

import VueRouter from 'vue-router';

3. Create a new instance of VueRouter and assign it to a variable. For example:

```
const router = new VueRouter({
  routes: [
    // Define your routes here
  ],
});
```

Make sure to replace the `routes` array with your actual routes configuration.

4. In the component's script section, define a method that handles the navigation to the dynamic route.

For example:

```
methods: {
  goToUser(userId) {
    this.$router.push(`/user/${userId}`);
  },
},
```

In this example, the `goToUser` method uses the `$router.push()` method to navigate to the dynamic route. It appends the `userId` parameter to the route path.

5. In the component's template, add an element or event handler that triggers the method defined in the previous step. For example:

```
<template>
  <div>
    <button @click="goToUser(123)">Go to User</button>
  </div>
</template>
```

In this example, a button element is used to trigger the `goToUser` method with a specific `userId` parameter (in this case, 123). You can modify the parameter value or use a different element or event to trigger the navigation.

6. Save the changes to the component file.

Now, when the button or element that triggers the `goToUser` method is clicked, it will navigate to the dynamic route with the specified parameter. For example, it will navigate to "/user/123" in this case.

Ensure that you have properly configured the dynamic route in your VueRouter instance with the corresponding component and route path. This allows Vue.js to match the URL and render the appropriate component when the dynamic route is accessed.

Remember to import VueRouter, create a new instance, define the method to handle navigation, and trigger the method in the template to use the dynamic route in your Vue.js application

Here's an example:

```
<template>
  <div>
    <!-- Some content -->
    <router-link :to="'/users/' + userId">Go to User</router-link>
    <!-- Rest of your application content -->
  </div>
</template>
<script>
export default {
  data() {
    return {
      userId: 123, // Example dynamic value
    };
  },
};
</script>
```

In this example, the userId data property is used to specify the dynamic value for the route parameter. The :to binding on the <router-link> component creates a link with the corresponding URL including the dynamic value.

Theoretical Activity 2.3.10: Description of 404 page

Tasks:

1. You are requested to answer the following questions related to 404 page in website development:
   - I. What do you understand about 404 page?
   - II. List and explain the advantages of the 404 page.
2. Provide the answer of asked questions by writing them on paper.
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 2.3.10. In addition, ask questions where necessary.

Key readings 2.3.10.: Description of 404 page

A 404 page, also known as an error page or "page not found" page, is displayed to users when they try to access a webpage that doesn't exist or can't be found.

While it may seem like an inconvenience, a well-designed 404 page can actually provide several advantages.

Here are some of them:

1. Improved user experience
2. Branding opportunity
3. Reducing bounce rate
4. Search Engine Optimisation(SEO) benefits
5. Error tracking and analysis

Directives

In Vue.js, directives are special attributes prefixed with v- that are used to add dynamic behavior to HTML elements or components. They are essentially markers on the DOM that tell Vue.js to do something to the element or component.

Here are some commonly used directives with examples:

1. **v-bind:** Binds an attribute or a component prop to an expression.

```
<div v-bind:class="{ active: isActive }"></div>
```

In this example, the class attribute of the div element will dynamically update based on the value of isActive.

2. **v-if / v-else-if / v-else:** Conditionally renders elements based on the truthiness of the expression.

```
<p v-if="seen">Now you see me</p>

<p v-else>Now you don't</p>
```

3. **v-for:** Iterates over items in an array or object to render a list of elements.

```
<ul>

  <li v-for="item in items" :key="item.id">{{ item.text }}</li>

</ul>
```

4. **v-on:** Listens to DOM events and triggers methods or inline expressions.

```
<button v-on:click="incrementCounter">Increment</button>
```

5. **v-model:** Creates two-way data bindings on form input and textarea elements.

```
<input v-model="message" placeholder="Enter message">
```

6. **v-show:** Toggles the visibility of elements based on a boolean condition.

```
<p v-show="isVisible">This will show or hide</p>
```

7. **v-cloak:** This directive remains on the element until the associated Vue instance finishes compilation. It is used to hide un-compiled mustache bindings.

```
<div v-cloak>

  {{ message }}

</div>
```

8. **v-pre:** Skips compilation for this element and all its children. Useful for displaying raw mustache tags.

```
<span v-pre>{{ rawText }}</span>
```

9. **v-once**: Render the element and component once only. Subsequent re-renders will not affect it.

```
<h1 v-once>{{ title }}</h1>
```

These directives provide powerful ways to interact with the DOM and manage the state of your Vue.js applications efficiently.

Practical Activity 2.3.11: Create 404 Page

Task:

1. Read key reading 2.3.11 and ask clarification where necessary.

2. Referring to the previous theoretical activities (2.3.10), you are requested to go to the computer lab to create a 404-page component. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create a 404-page component.

5. Referring to the steps provided on task 3, create a 404-page component.

6. Present your work to the trainer and whole class.

Key readings 2.3.11.: Create 404 Page

Create a custom 404 page in a Vue.js application:

1. Create a 404 component: In your project's component directory, create a new file called NotFound.vue (or any other appropriate name) to represent the 404-page component.

```
<template>
  <div>
    <h1>404 - Page Not Found</h1>
    <!-- Add custom content for the 404 page -->
  </div>
</template>
<script>
export default {
  name: 'NotFound',
};
</script>
<style scoped>
/* Add styling for the 404 component */
</style>
```

In this example, the template includes a simple heading and can be customized to add any desired content specific to your 404 page.

2. Set up the catch-all route: In your router configuration file (usually router.js), define a catch-all route at the end of your routes array. This catch-all route will match any route that doesn't match the defined routes.

```
import Vue from 'vue';
import VueRouter from 'vue-router';
Vue.use(VueRouter);
const router = new VueRouter({
  routes: [
    // Other routes...
    // Catch-all route for 404 page
    { path: '*', component: NotFound },
  ],
});
export default router;
```

In this example, the * path is used as a wildcard to match any route that doesn't match the other defined routes. The NotFound component is the component you created for the 404 page.

3. Customize the behavior and appearance

You can customize the behavior and appearance of the 404 page by modifying the content and styles within the NotFound.vue component. You can add additional elements, apply CSS styles, or even fetch data specific to the 404 page.

In Vue.js, you can customize the behavior and appearance of your components in several ways. Here are some common techniques to achieve this:

1. Props:

You can pass data from parent components to child components using props. Props allow you to customize the behavior of child components based on the data passed from the parent component.

Parent Component:

```
<template>
  <child-component :propName="parentData"></child-component>
</template>
<script>
import ChildComponent from './ChildComponent.vue';
export default {
  components: {
    ChildComponent
  },
  data() {
    return {
      parentData: 'Hello from parent!'
    };
  }
};
</script>
```

Child Component:

```
<template>
  <div>{{ propName }}</div>
</template>
<script>export default {
  props: {
    propName: String
  }
};
</script>
```

2. Computed Properties:

Computed properties allow you to perform operations on the data and return a modified result. They are cached based on their dependencies, and they can be used to customize the appearance of your components.

```
<template>
  <div>{{ reversedMessage }}</div>
</template>
<script>
export default {
  data() {
    return {
      message: 'Hello Vue!'
    };
  },
  computed: {
    reversedMessage() {
      return this.message.split('').reverse().join('');
    }
  }
};
</script>
```

3. Methods:

You can define custom methods in your component to handle specific behavior. Methods can be used to customize both behavior and appearance.

```
<template>
  <button @click="changeColor">Change Color</button>
</template>
<script>
export default {
  methods: {
    changeColor() {
      // Logic to change the appearance of the component
      // For example, changing the background color
      this.$el.style.backgroundColor = 'red';
    }
  }
};
</script>
```

4. Custom Events:

You can emit custom events from child components to notify parent components about specific actions. Parent components can listen for these events and perform custom actions based on them.

Child Component:

```
<template>
  <button @click="emitCustomEvent">Click me</button>
</template>
<script>
export default {
  methods: {
    emitCustomEvent() {
      this.$emit('customEvent', 'Data sent from child component');
    }
  }
};
</script>
```

Parent Component:

```
<template>
  <child-component @customEvent="handleCustomEvent"></child-component>
</template>
<script>
import ChildComponent from './ChildComponent.vue';
export default {
  components: {
    ChildComponent
  },
  methods: {
    handleCustomEvent(data) {
      console.log('Received data:', data);
      // Handle the data and customize the behavior accordingly
    }
  }
};
</script>
```

5. Directives:

Vue.js provides directives like v-bind and v-if that allow you to manipulate the DOM and customize the appearance of elements based on data.

```
<template>
```

```
 <div v-if="isVisible" v-bind:style="{ backgroundColor: bgColor }">
Customizable Content
 </div>
</template>
<script>
export default {
 data() {
   return {
     isVisible: true,
     bgColor: 'lightblue'
   };
 }
};
</script>
```

These are some of the ways you can customize the behavior and appearance of components in Vue.js. Depending on your specific use case, you can combine these techniques to achieve more complex customizations

Points to Remember

- There are differences between Vue's route and router where Vue Router is the routing library used in Vue.js applications to manage navigation and routing logic, while a route is a specific configuration that maps a URL path to a component within the application.
- Declarative navigation in Vue.js allows developers to define routing and navigation logic in a straightforward and intuitive manner using template directives and components. Here are some examples of declarative navigation in Vue: Router Links, Dynamic route parameters, Programmatic navigation with methods.
- Declarative navigation in Vue.js allows developers to define routing and navigation logic in a straightforward and intuitive manner using template directives and components. Here are some examples of declarative navigation in Vue: Router Links, Dynamic route parameters, Programmatic navigation with methods.

- In the Vue Router, there are various parameters that can be used to configure and customize the behavior of routes. These parameters allow you to pass dynamic values, query parameters, and apply route-specific guards.
- **To install the Vue Router and create a router instance in the Vue.js framework, you'll need to follow these steps:**

  1. Install the Vue Router package
  2. Create a router instance:
  3. Import your Vue components that will be used as routes.
  4. Use the Vue Router plugin
  5. Create an array of route objects, where each object represents a route with a corresponding path and component.
  6. Create a new instance of Vue Router by passing the routes array to the constructor, and assign it to a variable.
  7. Create a new Vue instance and pass the router instance as an option:
  8. Save the changes to the main project file.
- **To import component in Vue Router instance, you need to follow these steps:**

  1. Import the components
  2. Use the components in the router configuration
  3. Use the router instance in your Vue app
- To pass the Vue Router instance into the Vue app, you need to follow these steps:

  1. Import the required modules
  2. Import and configure the routes
  3. Create the Vue app
  4. Set up the template to use the router views
- To create a reusable navigation bar (navbar) component in Vue.js, you can follow these steps:

  1. Create a new Vue component file
  2. Define the Navbar component:
  3. Import and use the Navbar component
  4. Customize the navbar styling and behavior
- To use nested routes in Vue.js, you can follow these steps:

  1. Set up the parent route
  2. Create the parent component

3. Create the child components
4. Use the nested routes in your application

- To use dynamic routing in Vue.js, you can follow these steps:

    1. Set up the dynamic route in the router configuration
    2. Create the dynamic component
    3. Use the dynamic route in your application

- A 404 page in a web application offers several advantages that contribute to a positive user experience and overall website performance. There are the advantages of implementing a 404 page like: Improved user experience, Branding opportunity, Reducing bounce rate.
- **To create a custom 404 page in a Vue.js application:**

    1. Create a 404 component
    2. set up the catch-all route
    3. Customize the behavior and appearance

Application of learning 2.3.

BG Tech is a company that develop websites, as software developer you have been given task for creating a Vue.js application with routing, a navigation bar, nested routes, dynamic routes, and a custom 404 page.

The website has to contains the following components:

✓ Home.vue

✓ About.vue

✓ Contact.vue

✓ Registration.vue

✓ Login.vue

✓ Profile.vue

✓ Dashboard.vue

✓ Overview.vue

✓ Settings.vue

- ✓ Reports.vue
- ✓ NotFound.vue

Indicative content 2.4: Data Manipulation in Vue

Duration: 8 hrs

Theoretical Activity 2.4.1: Description of Vue lifecycle

Tasks:

1. You are requested to answer the following questions related to the Vue cycle:

   I. What do you understand about vue life cycle?

   II. List the main lifecycle hooks available in Vue.js 2.x

   III. Discuss the main lifecycle hooks available in Vue.js 2.x

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.4.1. In addition, ask questions where necessary.

Key readings 2.4.1.: Description of Vue lifecycle

Vue life cycle: In Vue.js, the lifecycle of a component refers to the different stages or phases that a component goes through from its creation to its destruction. These stages are represented by a series of lifecycle hooks, which are predefined methods that allow you to perform actions at specific points during the component's lifecycle.

Components have a life cycle consisting of various stages or phases that occur from the moment the component is created until it is destroyed. These phases are known as the Vue lifecycle hooks, and they allow you to execute custom code at specific points during a component's lifespan.

Here is an overview of the main lifecycle hooks available in Vue.js 2.x:

1.**beforeCreate:** This hook is called before the component instance is initialized, and therefore, data and events have not been set up yet.

2.**created:** At this stage, the component has been initialized, data and events are set up, but the template has not been compiled or mounted to the DOM.

3.**beforeMount:** This hook is called right before the component's template is compiled and inserted into the DOM. The component has not been mounted yet.

4.**mounted:** The component's template has been compiled and mounted into the DOM. At this point, the component is visible and can be interacted with.

5.**beforeUpdate:** This hook is triggered when a reactive property used by the component changes, right before the re-rendering process begins.

6.**updated:** The component has been re-rendered due to a change in reactive properties. Any changes made to the DOM during this hook will not trigger further updates.

7.**beforeDestroy:** This hook is called right before a component is destroyed. It allows you to clean up any event listeners, timers, or other resources before the component is removed.

8.**destroyed**: The component has been destroyed, and all its data, watchers, and child components have been cleaned up.

Additionally, Vue 3 introduced some changes in the lifecycle hooks. The most notable change is the introduction of the beforeUnmount and unmounted hooks, which replace the beforeDestroy and destroyed hooks, respectively. The rest of the lifecycle hooks remain the same.

Theoretical Activity 2.4.2: Introduction to JSON data

Tasks:

1. You are requested to answer the following questions related to the Json data:
   I. Describe JSON
   II. Describe API data
   III. What do you understand about component props, state management, and application configuration?
2. Provide the answer for the asked questions and write them on papers.
3. Present the findings/answers to the whole class
4. For more clarification, read the key readings 2.4.2. In addition, ask questions where necessary.

Key readings 2.4.2.: Introduction to JSON data

JSON (JavaScript Object Notation) data is commonly used to represent structured data. 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 provides a convenient way to store and transmit data between a server and a client, or within a Vue.js application.

JSON data is represented as key-value pairs, similar to JavaScript object literals. The keys are always strings, and the values can be strings, numbers, booleans, arrays, objects, or null. JSON supports nested structures, allowing for complex data hierarchies.

Here's an example of a simple JSON object:

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "hobbies": ["reading", "traveling"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "country": "USA"
  }
}
```

In this example, the JSON object represents a person's information. It has properties like "name", "age", and "isStudent", with corresponding values. The "hobbies" property is an array containing multiple values, and the "address" property is an object with nested properties.

JSON is widely supported across programming languages and platforms, making it a popular choice for data exchange in web services and APIs. It is easy to parse and generate using built-in functions or libraries available in most programming languages.

To parse a JSON string into a JavaScript object, you can use the `JSON.parse()` method. To convert a JavaScript object into a JSON string, you can use the `JSON.stringify()` method. These methods facilitate working with JSON data in various programming environments.

JSON data in Vue.js can be used in various scenarios, such as:

1. API Data: When fetching data from an API, the response is often in JSON format. Vue.js can consume this JSON data and use it to populate the application's components with dynamic content.

API data refers to the information that is retrieved from an Application Programming Interface (API). An API is a set of rules and protocols that allows different software applications to communicate and exchange data with each other.

When you make a request to an API, you typically receive a response that contains data in a structured format, such as JSON or XML. This data can include various types of information, such as text, numbers, dates, images, or even more complex data structures.

API data can come from a wide range of sources, including weather APIs, social media APIs, financial APIs, and many others. For example, a weather API might provide data about current weather conditions, temperature, humidity, and forecast information. A social media API might provide data about users, posts, comments, and likes.

Once you retrieve the data from an API, you can use it in your application to display information, perform calculations, make decisions, or integrate it with other systems or services.

To work with API data, you typically send HTTP requests to the API's endpoints, specifying any required parameters or authentication credentials. The API then processes your request and returns the requested data in the response.

API data is an essential component of many modern applications, enabling developers to access and utilize external data sources, services, and functionalities to enhance their own applications and provide valuable features to users.

2. Component Props: JSON data can be passed as props to child components, allowing them to render the data dynamically. This enables reusability and flexibility in component composition.

By passing props, you can provide dynamic data to child components, making them reusable and customizable. The parent component can pass different values to the child component's props, allowing for flexibility and reusability.

To define props in a Vue component, you can use the `props` option. Here's an example:

```
Vue.component('ChildComponent', {
  props: ['message'],
  template: '<div>{{ message }}</div>'
});
```

In this example, the `ChildComponent` has a single prop called `message`. The value of this prop can be passed from the parent component when using the `ChildComponent` in its template.

To pass a value to a prop in the parent component's template, you can use the `v-bind` directive or the shorthand `:`. For example:

```
<ChildComponent :message="Hello, World!"></ChildComponent>
```

In this case, the value "Hello, World!" is passed to the `message` prop of the `ChildComponent`.

Inside the child component, you can access the prop's value using the `this` keyword. For example, in the `ChildComponent` template, we can use `{{ message }}` to display the value of the `message` prop.

Props are read-only by default, meaning that the child component should not modify the prop's value directly. If the child component needs to modify the prop's value, it should emit an event to notify the parent component, and the parent component can update the prop accordingly.

Using props in Vue.js allows for the passing of data between components, enabling reusability and flexibility in building Vue applications.

3. State Management: JSON data can be stored in the application's state management solution, such as Vuex or Vue's built-in data property. This allows the data to be shared across multiple components and kept in sync.

In complex applications, as the number of components and their interactions grow, managing the shared data and ensuring consistency can become challenging. State management provides a solution by decoupling the data from individual components and centralizing it in a dedicated store or state container.

State management libraries or patterns, such as Vuex (for Vue.js) or Redux (for React), are commonly used to implement state management in JavaScript frameworks. These libraries provide a predictable and structured way to manage and update the application's state.

The core concepts of state management typically include:

1. State: The data that represents the current state of the application. It can include user information, settings, fetched data, or any other relevant data.

2. Actions: Functions or methods that define how to modify the state. Actions are triggered by events, user interactions, or asynchronous operations. They encapsulate the logic for updating the state.

3. Mutations: Functions or methods that actually modify the state. Mutations are responsible for changing the state based on the actions. They ensure that the state is updated in a controlled and predictable manner.

4. Getters: Functions or methods that provide computed properties based on the state. Getters allow components to access and derive values from the state without directly modifying it.

5. Store: The centralized container that holds the state, actions, mutations, and getters. It acts as a single source of truth for the application's data and provides an interface for components to interact with the state.

By implementing state management, you can achieve several benefits, including:

Simplified data flow: Components can access and modify the state without passing data through multiple levels of component hierarchy.

- Improved code organization: State management separates the concerns of data management from the components, resulting in cleaner and more maintainable code.

- **Enhanced reusability:** Components become more reusable as they rely on the shared state rather than local component-level data.

- **Easy debugging and testing:** With a centralized state, it becomes easier to debug and test the application's data flow and interactions.

State management is particularly useful in large-scale applications or those with complex data dependencies, providing a structured and scalable approach to handle and update the application's state.

Configuration: JSON data can be used for application configuration, where settings and options are defined in a structured manner. This makes it easy to modify the configuration without modifying the code itself.

The configuration JSON file typically contains a collection of key-value pairs, where the keys represent the configuration property names, and the values represent the corresponding values or settings. The values can be strings, numbers, booleans, arrays, objects, or null.

Here's an example of a simple configuration JSON file:

```
{
  "appName": "My Application",
  "theme": "dark",
  "apiUrl": "https://api.example.com",
  "maxItems": 10,
  "features": ["feature1", "feature2", "feature3"],
  "credentials": {
    "username": "user123",
    "password": "secretpassword"
  }
}
```

In this example, the configuration JSON file defines various properties such as the application name (`appName`), the theme (`theme`), the API URL (`apiUrl`), the maximum number of items (`maxItems`), an array of features (`features`), and a nested object containing credentials (`credentials`).

The configuration JSON file is typically used by an application or system to read and apply these settings during runtime. The application can parse and access the values from the configuration file to configure its behavior, appearance, or other aspects based on the defined settings.

Configuration JSON files are often used to make an application more flexible and customizable without modifying the source code. By modifying the values in the configuration file, users or administrators can change the behavior or settings of an application without the need for recompiling or redeploying the application.

It's worth noting that the specific structure and content of a configuration JSON file may vary depending on the application or system that uses it. The properties and values in the configuration file should be defined according to the requirements and conventions of the particular application or system.

Practical Activity 2.4.3: Import necessary packages & components

Task:

1. Read key reading 2.4.3 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.2.1), you are requested to go to the computer lab to import necessary packages & components. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to import necessary packages & components.

5. Referring to the steps provided on task 3, Import necessary packages & components.

6. Present your work to the trainer and whole class

Key readings 2.4.3.: Import necessary packages & components

Install the required packages using a package manager like npm or yarn

To install required packages using a package manager like npm or axios, you can follow these general steps:

1. Set up Node.js and npm:

Install Node.js from the official website (https://nodejs.org) if you haven't already.

npm (Node Package Manager) comes bundled with Node.js, so you should have it installed automatically.

2. Create a new project or navigate to an existing project:

Open a terminal or command prompt and navigate to the directory where you want to create or work on your project.

3. Initialize the project (if starting a new project):

Run the following command to initialize a new project and create a `package.json` file:

```
npm init
```

Follow the prompts to provide information about your project or press Enter to accept the default values.

4. Install required packages:

- Use the package manager (npm) to install the required packages.

- For example, to install a package like axios, run the following command:

```
npm install axios
```

- This command will download and install the latest version of the axios package and its dependencies into the `node_modules` directory.

5. Use the installed package in your code:

- In your JavaScript code, import or require the installed package as needed.

- For example, if you're using axios, add the following line at the beginning of your JavaScript file:

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

6. Start using the package:

- You can now use the installed package in your code. Refer to the package's documentation or examples to learn how to use it effectively.

Remember to include the `node_modules` directory in your project's `.gitignore` file to avoid committing the installed packages to your version control system.

These steps should help you install the required packages using npm or axios for your project. Make sure to replace "axios" with the actual package name you need to install.

Example

To make an HTTP request using the imported axios package, you can follow these steps:

1. Import the axios package:

- Make sure you have already installed the axios package by running `npm install axios`.

- In your JavaScript file, import the axios package at the top of your file:

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

2. Use axios to make the HTTP request:

- You can use axios to send different types of HTTP requests, such as GET, POST, PUT, DELETE, etc.

- Here's an example of making a GET request to retrieve data from a remote API:

```
axios.get('https://api.example.com/data')
  .then(response => {
    // Handle the response data
    console.log(response.data);
  })
  .catch(error => {
    // Handle the error
    console.error(error);
  });
```

3. Handle the response and error:

- In the above example, the `.then()` method is called when the request is successful, and the response is received.

- The response object contains various properties such as `data`, `status`, `headers`, etc. You can access the response data using `response.data`.

- The `.catch()` method is called when an error occurs during the request, such as a network error or server error.

- Inside the `.catch()` block, you can handle the error appropriately, such as displaying an error message or taking corrective actions.

4. Customize the request:

- You can customize the HTTP request by providing additional options to the axios method.

- For example, you can pass query parameters, headers, request body, or authentication credentials as options.

- Here's an example of making a POST request with a request body:

```
axios.post('https://api.example.com/data', { name: 'John', age: 30 })
    .then(response => {
      // Handle the response
      console.log(response.data);
    })
    .catch(error => {
      // Handle the error
      console.error(error);
    });
```

These steps should help you make an HTTP request using the imported axios package in your JavaScript code. Remember to handle the response and error appropriately based on your application's requirements.

To import necessary packages and components in a Vue.js application, you can follow these steps:

1.Install the required packages using a package manager like npm or Yarn. For example, to install a package named "axios" for making HTTP requests, you can run:

```
npm install axios
```

2.In your Vue component file, import the required packages and components using the import statement.

Here's an example:

```
<template>
  <!-- Your component template -->
</template>
<script>
// Import packages
```

```
import axios from 'axios';
// Import components
import MyComponent from './MyComponent.vue';
export default {
  components: {
    MyComponent // Register the imported component
  },
  // Your component code
};
</script>
```

In the example above, the axios package is imported using the import statement. The package name is specified, and the package is assigned to the axios variable.

Additionally, a custom component named MyComponent is imported from a file named MyComponent.vue. The component is assigned to the MyComponent property in the components object.

Note that the relative path to the component file may vary depending on your project structure.

3.Once you've imported the packages and components, you can use them in your Vue component. For example, you can make an HTTP request using the imported axios package:

```
<script>
import axios from 'axios';
export default {
  methods: {
    fetchData() {
      axios.get('/api/data')
        .then(response => {
          // Handle the response data
          console.log(response.data);
        })
        .catch(error => {
          // Handle the error
          console.error(error);
        });
    }
  }
};
</script>
```

In the above example, the fetchData method makes an HTTP GET request to the /api/data endpoint using the axios package. The response data is logged to the console, and any errors are logged as well.

Practical Activity 2.4.4: Apply Vue lifecycle methods

Task:

1. Read key reading 2.4.4 and ask clarification where necessary

2. Referring to the previous theoretical activities, you are requested to go to the computer lab to apply vue lifecycle methods. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to apply vue lifecycle methods.

5. Referring to the steps provided on task 3, apply vue lifecycle methods.

6. Present your work to the trainer and whole class

Key readings 2.4.4.: Apply Vue lifecycle methods

To apply Vue lifecycle methods in a Vue.js application, follow these steps:

1. Understanding Vue Lifecycle Methods:

- Familiarize yourself with the different lifecycle methods available in Vue.js, such as `beforeCreate`, `created`, `beforeMount`, `mounted`, `beforeUpdate`, `updated`, `beforeDestroy`, and `destroyed`. Each method corresponds to a different stage in the lifecycle of a Vue instance

2. **Implement Lifecycle Methods in Vue Components:**

- Inside your Vue components, you can define and implement the necessary lifecycle methods. For example, to log a message when a component is created, you can add the `created` method to your component options:

```
export default {
 created() {
  console.log('Component created');
 }
}
```

3. **Use Lifecycle Methods for Data Initialization:**

- Utilize lifecycle methods to initialize data, set up watchers, perform API calls, or any other necessary tasks at specific stages of the component lifecycle. For instance, you can fetch data from an API in the `created` hook:

```
export default {
 data() {
  return {
   items: []
  };
 },
 created() {
  axios.get('https://api.example.com/items')
```

```
        .then(response => {
          this.items = response.data;
        })
        .catch(error => {
          console.error(error);
        });
    }
  }
```

4. Handle DOM Manipulation in Lifecycle Hooks:

- Use lifecycle hooks like `mounted` to perform DOM manipulations or interact with the DOM after the component has been mounted. For example, you can access a DOM element and apply a plugin after the component is mounted:

```
  export default {
    mounted() {
      // Access a DOM element and apply a plugin
      const element = document.getElementById('myElement');
      // Apply a plugin or manipulate the DOM element
    }
  }
```

5. Clean Up Resources in Lifecycle Hooks:

- Utilize `beforeDestroy` to clean up resources, such as event listeners or subscriptions, before a component is destroyed. This ensures that resources are properly released to prevent memory leaks:

```
  export default {
    beforeDestroy() {
      // Clean up resources before component is destroyed
      // Unsubscribe from event listeners, clear intervals, etc.
    }
    }
```

6. Test and Debug:

- Test your Vue components to ensure that the lifecycle methods are being triggered at the expected stages and that the desired functionality is achieved. Use browser developer tools and Vue Devtools for debugging.

Commonly used Vue.js lifecycle methods and how to apply them:

1. **beforeCreate:** This method is called before the component is created, and it is useful for setting up initial data or performing other tasks that need to happen before the component is fully initialized.

```
new Vue({
  beforeCreate() {
   // Initialization code before the component is created
  },
});
```

Here's an example of using the `beforeCreate` Vue.js lifecycle method:

```
new Vue({
  beforeCreate() {
    console.log('beforeCreate hook');
    // Perform any necessary setup or initialization tasks here
  },
  created() {
    console.log('created hook');
// Data observation and event initialization have been set up at this point
  },
  // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `beforeCreate` method as one of the component options. The `beforeCreate` hook is called before the component is created, which means that data observation and event initialization have not been set up yet.

Inside the `beforeCreate` method, you can perform any necessary setup or initialization tasks that need to be done before the component is fully created. This could include tasks like fetching initial data, setting up event listeners, or configuring external dependencies.

When the Vue instance is created, the `beforeCreate` hook will be triggered, and the console will log 'beforeCreate hook'. After the `beforeCreate` hook is executed, the `created` hook will be called, and the console will log 'created hook'.

It's important to note that the `beforeCreate` hook is typically used for tasks that need to be done before the component is fully initialized. If you need to access or modify data or perform tasks that rely on the component's reactive properties, you may want to use the `created` hook or other appropriate lifecycle hooks.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

2. Created: This method is called immediately after the component is created. It's a good place to perform data fetching or other asynchronous operations.

```
new Vue({
  created() {
   // Data fetching and other setup
  },
});
```

Here's an example of using the `created` Vue.js lifecycle method:

```
new Vue({
  data() {
    return {
      message: 'Hello, Vue!'
    };
```

```
 },
 created() {
  console.log('created hook');
  console.log('Message:', this.message);
  // Perform any necessary operations or logic after the component is created
 },
 // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `created` method as one of the component options. The `created` hook is called after the component has been created, and data observation and event initialization have been set up.

Inside the `created` method, you can perform any necessary operations or logic that need to be done after the component is created. This could include tasks like making API calls, initializing external libraries, or setting up additional data properties.

In the example, we have a `message` data property defined using the `data` function. Inside the `created` hook, we log 'created hook' to the console and then access and log the value of the `message` property using `this.message`.

When the Vue instance is created, the `created` hook will be triggered, and the console will log 'created hook' followed by the value of the `message` property.

It's important to note that the `created` hook is a commonly used lifecycle hook for performing initialization tasks that require access to the component's reactive properties. It provides a good opportunity to set up data, perform API calls, or perform any other necessary operations after the component has been created.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

3. beforeMount: This method is called before the component is inserted into the DOM. It's a suitable place to make any necessary DOM manipulations before the component is rendered.

```
new Vue({
 beforeMount() {
  // DOM manipulations
 },
});
```

Here's an example of using the `beforeMount` Vue.js lifecycle method:

```
new Vue({
 data() {
   return {
     message: 'Hello, Vue!'
   };
 },
 beforeMount() {
   console.log('beforeMount hook');
   console.log('Message:', this.message);
   // Perform any necessary operations or logic before the component is mounted to the DOM
 },
 // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `beforeMount` method as one of the component options. The `beforeMount` hook is called right before the component is mounted to the DOM.

Inside the `beforeMount` method, you can perform any necessary operations or logic that need to be done before the component is mounted. This could include tasks like manipulating the component's data or making final adjustments to the component's structure.

In the example, we have a `message` data property defined using the `data` function. Inside the `beforeMount` hook, we log 'beforeMount hook' to the console and then access and log the value of the `message` property using `this.message`.

When the Vue instance is created, the `beforeMount` hook will be triggered, and the console will log 'beforeMount hook' followed by the value of the `message` property.

It's important to note that the `beforeMount` hook is typically used for tasks that need to be done before the component is rendered and attached to the DOM. If you need to perform operations that rely on the component's rendered output or interact with the DOM, you may want to use the `mounted` hook or other appropriate lifecycle hooks.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

5. Mounted: This method is called after the component has been inserted into the DOM. It's commonly used for interacting with the DOM or initializing third-party libraries.

```
new Vue({
    mounted() {
  // DOM interactions or third-party library initialization
  },
});
```

Here's an example of using the `mounted` Vue.js lifecycle method:

```
new Vue({
  data() {
    return {
      message: 'Hello, Vue!'
    };
  },
  mounted() {
    console.log('mounted hook');
    console.log('Message:', this.message);
    // Perform any necessary operations or logic after the component is mounted
to the DOM
  },
  // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `mounted` method as one of the component options. The `mounted` hook is called after the component has been mounted to the DOM.

Inside the `mounted` method, you can perform any necessary operations or logic that need to be done after the component is mounted. This could include tasks like interacting with the DOM, setting up event listeners, or fetching additional data.

In the example, we have a `message` data property defined using the `data` function. Inside the `mounted` hook, we log 'mounted hook' to the console and then access and log the value of the `message` property using `this.message`.

When the Vue instance is created and the component is mounted, the `mounted` hook will be triggered, and the console will log 'mounted hook' followed by the value of the `message` property.

It's important to note that the `mounted` hook is commonly used for performing tasks that require access to the component's rendered output or interaction with the DOM. It provides a good opportunity to set up event listeners, initialize external libraries, or perform any other necessary operations after the component has been mounted.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

5. beforeUpdate: This method is called when data changes and just before the DOM is re-rendered. You can use it to perform tasks before the component's state is updated.

```
new Vue({
  beforeUpdate() {
   // Tasks before the component is updated
  },
});
```

Here's an example of using the `beforeUpdate` Vue.js lifecycle method:

```
new Vue({
  data() {
    return {
      message: 'Hello, Vue!',
      count: 0
    };
  },
  beforeUpdate() {
    console.log('beforeUpdate hook');
    console.log('Message:', this.message);
    console.log('Count:', this.count);
    // Perform any necessary operations or logic before the component updates
  },
  // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `beforeUpdate` method as one of the component options. The `beforeUpdate` hook is called before the component updates, but after the data has changed.

Inside the `beforeUpdate` method, you can perform any necessary operations or logic that need to be done before the component updates. This could include tasks like accessing the previous and current values of data properties

6. updated: This method is called after the component's data has changed and the DOM has been re-rendered. You can perform tasks that require access to the updated DOM in this method.

```
new Vue({
  updated() {
   // Tasks after the component is updated and the DOM is re-rendered
  },
});
```

Here's an example of using the `updated` Vue.js lifecycle method:

```
new Vue({
  data() {
    return {
      message: 'Hello, Vue!',
      count: 0
    };
  },
  updated() {
    console.log('updated hook');
    console.log('Message:', this.message);
    console.log('Count:', this.count);
    // Perform any necessary operations or logic after the component updates
  },
  // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `updated` method as one of the component options. The `updated` hook is called after the component has been updated due to changes in data.

Inside the `updated` method, you can perform any necessary operations or logic that need to be done after the component updates. This could include tasks like interacting with the DOM, making additional API calls, or updating external libraries.

In the example, we have a `message` and `count` data properties defined using the `data` function. Inside the `updated` hook, we log 'updated hook' to the console and then access and log the values of the `message` and `count` properties using `this.message` and `this.count`.

When the Vue instance is created and the component is rendered, any changes to the data properties will trigger the `updated` hook after the component updates to reflect those changes. The console will log 'updated hook' followed by the values of the `message` and `count` properties.

It's important to note that the `updated` hook is commonly used for performing tasks that need to be done after the component updates, such as interacting with the DOM or updating external dependencies. It provides a good opportunity to perform any necessary operations based on the updated state of the component.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

7. beforeDestroy: This method is called just before a component is destroyed. You can use it to clean up resources or remove event listeners.

```
new Vue({
  beforeDestroy() {
   // Cleanup operations
  },
});
```

Here's an example of using the `beforeUpdate` Vue.js lifecycle method:

```
new Vue({
  data() {
    return {
      message: 'Hello, Vue!',
      count: 0
    };
  },
  beforeUpdate() {
    console.log('beforeUpdate hook');
    console.log('Message:', this.message);
    console.log('Count:', this.count);
    // Perform any necessary operations or logic before the component updates
  },
  methods: {
    incrementCount() {
      this.count++;
    }
  },
  // Other lifecycle hooks and component options...
});
```

In this example, we define a Vue instance and include the `beforeUpdate` method as one of the component options. The `beforeUpdate` hook is called before the component updates, but after the data has changed.

Inside the `beforeUpdate` method, you can perform any necessary operations or logic that need to be done before the component updates. This could include tasks like accessing the previous and current values of data properties, performing calculations, or making additional API calls.

In the example, we have a `message` and `count` data properties defined using the `data` function. Inside the `beforeUpdate` hook, we log 'beforeUpdate hook' to the console and then access and log the values of the `message` and `count` properties using `this.message` and `this.count`.

We also have a `incrementCount` method that increments the `count` property when called.

When the Vue instance is created and the component is rendered, any changes to the data properties will trigger the `beforeUpdate` hook before the component updates to reflect those changes. The console will log 'beforeUpdate hook' followed by the values of the `message` and `count` properties.

To trigger the `beforeUpdate` hook, you can call the `incrementCount` method or make changes to the `message` property.

It's important to note that the `beforeUpdate` hook is typically used for tasks that need to be done before the component updates but after the data has changed. If you need to perform operations after the component updates, you may want to use the `updated` hook or other appropriate lifecycle hooks.

Remember, Vue.js provides a range of lifecycle hooks that allow you to perform actions at different stages of a component's lifecycle, giving you control and flexibility in managing your component's behavior and state.

8. destroyed: This method is called after the component is destroyed. It's the last opportunity to perform cleanup before the component is removed completely.

```
new Vue({
  destroyed() {
   // Final cleanup operations
  },
});
```

To apply these lifecycle methods, define them as functions within your Vue component's options object. For example, if you have a Vue component called `MyComponent`, you can add these methods as shown below:

```
const MyComponent = {
  beforeCreate() {
   // Initialization code
  },
  created() {
   // Data fetching and setup
  },
  beforeMount() {
   // DOM manipulations
  },
  mounted() {
   // DOM interactions or library initialization
  },
  beforeUpdate() {
   // Tasks before component update
  },
  updated() {
   // Tasks after component update
  },
beforeDestroy() {
   // Cleanup operations
  },
  destroyed() {
   // Final cleanup operations
  },
};
new Vue({
  el: '#app',
  components: {
   MyComponent,
  },
});
```

Practical Activity 2.4.5: Using Vue layout components

Task:

1. Read key reading 2.4.5 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.2.1), you are requested to go to the computer lab to use Vue layout components. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to use Vue layout components.

5. Referring to the steps provided on task 3, use Vue layout components.

6. Present your work to the trainer and whole class

Key readings 2.4.5.: Using Vue layout components

To use Vue layout components in your Vue.js application, you can follow these steps:

1. Install Vue CLI (if not already installed):

- If you haven't set up a Vue.js project, you can install Vue CLI by running:

```
npm install -g @vue/cli
```

2. Create a Vue.js Project:

- Create a new Vue.js project using Vue CLI by running:

```
vue create my-vue-app
```

3. Navigate to Project Directory:

- Change into the project directory:

```
cd my-vue-app
```

4.Install Vue Router (if needed):

- If you plan to use Vue Router for routing in your application, you can install it by running:

```
npm install vue-router
```

6. Import Layout Components:

- Import the necessary layout components from the library in your Vue components where you want to use them. For example, if you are using Vuetify, you can import Vuetify components like this:

```
import { VApp, VContainer, VRow, VCol } from 'vuetify';
```

7. Use Layout Components in Vue Templates:

- Incorporate the imported layout components in your Vue component templates. For instance, you can use Vuetify layout components to structure your layout as follows:

```
<template>
 <v-app>
  <v-container>
   <v-row>
    <v-col>
     <!-- Your content goes here -->
    </v-col>
   </v-row>
  </v-container>
 </v-app>
```

```
</template>
```

8. Customize Layout Components:

- Customize the layout components according to your design requirements by adjusting props, styles, or adding additional classes as needed. Refer to the documentation of the layout components library for customization options.

9. Run the Vue.js Application:

- Finally, run your Vue.js application to see the layout components in action and ensure they are rendering as expected:

```
npm run serve
```

Layout components in Vue.js are typically used to structure the overall layout of a web application. You can create custom layout components or use existing ones to organize your application's structure.

Example of how to use Vue layout components to create a simple layout:

1. First, make sure you have Vue.js installed and set up in your project.

2. Create a new Vue component for your layout. You can do this in a single-file component (*.vue) or define it in your main Vue instance. For this example, let's create a single-file component called `AppLayout.vue` for the layout.

```
<template>
 <div>
  <header>
  <nav>
        <!-- Navigation links or components go here -->
  </nav>
  </header>
   <main>
  <slot></slot>
  </main>
  <footer>
  <!-- Footer content goes here -->
```

```
   </footer>
  </div>
</template>
<style scoped>
/* Add your CSS styles for the layout component here */
</style>
```

In this layout component, we have a header, main content area, and a footer. We also use a `<slot></slot>` to allow other components to inject content into the main section.

3. Use your layout component in a parent component. For example, let's create a component called `Home.vue` that will use the `AppLayout` component.

```
<template>
  <app-layout>
   <div>
   <!-- Content specific to the home page goes here -->
   </div>
  </app-layout>
</template>
<script>
import AppLayout from './AppLayout.vue';
export default {
  components: {
   AppLayout,
  },
};
</script>
```

In this example, we import the `AppLayout` component and use it to wrap the content specific to the home page.

4. Make sure you have your main Vue instance configured to render the `Home` component or any other components as needed:

```
import Vue from 'vue';

import Home from './Home.vue';

new Vue({

  render: (h) => h(Home),

}).$mount('#app');
```

Note: In your project, you would typically have more components and routes, but this demonstrates the basic usage of a layout component to structure your Vue.js application's layout.

Practical Activity 2.4.6: Create form in Vue component

Task:

1. Read key reading 2.4.6 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.4.1), you are requested to go to the computer lab to create form in Vue components. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create form in Vue components.

5. Referring to the steps provided on task 3, create form in Vue components.

6. Present your work to the trainer and whole class

Key readings 2.4.6.: Create form in Vue component

To create a form in Vue.js, you can follow these steps:

1.**Set up a Vue.js project:** Set up a new Vue.js project or navigate to your existing Vue.js project folder.

2.**Create a Vue component:** Create a new Vue component file (e.g., Form.vue) where you'll define your form.

3.**Define the form template:** In the component file, define the form template using HTML markup and Vue.js directives.

In this example let us create a form that will display name, sex, address, phone, age, trade, level, fatherName, motherName. For a given student

```
<template>
 <form @submit.prevent="submitForm">
  <div>
   <label for="name">Name:</label>
   <input type="text" id="name" v-model="formData.name" required />
  </div>
  <div>
   <label for="sex">Sex:</label>
   <select id="sex" v-model="formData.sex" required>
    <option value="male">Male</option>
    <option value="female">Female</option>
   </select>
  </div>
  <div>
   <label for="address">Address:</label>
   <input type="text" id="address" v-model="formData.address" required />
  </div>
  <div>
   <label for="phone">Phone:</label>
   <input type="text" id="phone" v-model="formData.phone" required placeholder="+250..." pattern="\+250\d{9}" />
  </div>
  <div>
   <label for="age">Age:</label>
```

```
      <input type="number" id="age" v-model="formData.age" required />
    </div>
    <div>
      <label for="trade">Trade:</label>
      <input type="text" id="trade" v-model="formData.trade" required />
    </div>
    <div>
      <label for="level">Level:</label>
      <input type="text" id="level" v-model="formData.level" required />
    </div>
    <div>
      <label for="fatherName">Father's Name:</label>
      <input type="text" id="fatherName" v-model="formData.fatherName" required />
    </div>
    <div>
      <label for="motherName">Mother's Name:</label>
      <input type="text" id="motherName" v-model="formData.motherName" required />
    </div>
    <button type="submit">Submit</button>
  </form>
</template>
```

4. **Define the form data and methods**: In the component's <script> section, define the form data and methods to handle form submission and data binding.

Example

```
<script>
export default {
  data() {
    return {
      formData: {
        name: '',
        sex: '',
        address: '',
        phone: '',
        age: '',
        trade: '',
        level: '',
        fatherName: '',
        motherName: '',
      },
    };
  },
  methods: {
    submitForm() {
      // Handle form submission
      console.log('Form Data:', this.formData);
    },
  },
};
</script>
```

5. **Register and use the component:** In your main Vue app file (e.g., main.js or App.vue), import and register the Form component, and use it within your template.

Remember that we have created a component called **DataCollectionForm.vue.**

Example

(main.js or App.vue):

```
<template>
  <div id="app">
    <DataCollectionForm />
  </div>
</template>
<script>
import DataCollectionForm from './components/DataCollectionForm.vue';
export default {
  name: 'App',
  components: {
    DataCollectionForm,
  },
};
</script>
```

6. **Style the form (optional):** You can add CSS styles to the form component or apply CSS frameworks like Bootstrap or Tailwind CSS to enhance the form's appearance.

```
<style scoped>
/* Add your styles here */
form {
  display: flex;
  flex-direction: column;
  max-width: 400px;
  margin: auto;
}
div {
  margin-bottom: 10px;
}
label {
  margin-bottom: 5px;
}


input, select {
  padding: 5px;
  font-size: 1em;
}
</style>
```

7. **Run the Vue.js application:** Start your Vue.js development server or compile your project, and you should see the form rendered in the browser.

Input binding

This is also known as two-way data binding, is a powerful feature in Vue.js that allows you to bind the data in your Vue component to the value of an input element. This enables automatic synchronization between the input field and the underlying data, ensuring that any changes to the input are reflected in the data, and vice versa. To use input binding in Vue, you can follow these steps:

1.**Set up a Vue component:** Create a new Vue component or open an existing one where you want to use input binding.

2.**Define the data property:** In the component's <script> section, define a data property that represents the value you want to bind to the input element.

Example:

```
<script>
export default {
  data() {
    return {
      message: ''
    };
  }
}
</script>
```

2.**Bind the data property to the input element**: In the component's template, use the v-model directive to bind the input element's value to the data property.

Example:

```
<template>
  <div>
    <input type="text" v-model="message">
    <p>Input value: {{ message }}</p>
  </div>
</template>
```

3.**React to changes in the data property:** You can use the bound data property (message in this example) to access the input value and react to any changes. You can display the value in the template or perform any necessary operations.

Example:

```
<template>
  <div>
    <input type="text" v-model="message">
    <p>Input value: {{ message }}</p>
    <p>Character count: {{ message.length }}</p>
  </div>
</template>
```

In the example above, the message data property is bound to the input element, and any changes to the input field will update the message value. The template then displays the value of the message and the character count.

4. **Handle input events (optional):** If you want to perform additional actions when the input value changes, you can add event listeners or use computed properties or watchers to respond to those changes.

Example (using a computed property):

```
<script>

export default {

  data() {

    return {

      message: ''

    };

  },

  computed: {

    messageLength() {

      return this.message.length;

    }

  }

}

</script>
```

In this example, a computed property messageLength is used to calculate the length of the message value whenever it changes. The template then displays the computed property value.

Validate form inputs

To validate form inputs in Vue.js, you can follow these steps:

1.**Set up a Vue component:** Create a new Vue component or open an existing one where you want to implement form validation.

2.**Define the data properties:** In the component's <script> section, define the data properties that represent the form inputs and their validation states.

Example:

```
<script>
```

```
export default {
  data() {
    return {
      name: '',
      email: '',
      errors: {
        name: '',
        email: ''
      }
    };
  }
}
</script>
```

In this example, name and email represent the form inputs, and errors hold the validation error messages for each input.

2.**Implement form validation logic:** Create a method to handle form validation. This method should check the validity of each input and update the errors data property accordingly.

Example:

```
<script>
export default {
  data() {
    return {
      name: '',
      email: '',
      errors: {
        name: '',
        email: ''
      }
    };
  },
  methods: {
    validateForm() {
      this.errors.name = this.name.trim() === '' ? 'Name is required.' : '';
      this.errors.email = this.email.trim() === '' ? 'Email is required.' : '';
    }
  }
}
</script>
```

3.In the validateForm method, we check if the name and email inputs are empty. If they are, we set the corresponding error message; otherwise, we clear the error message.

4.Add validation feedback to the template: In the component's template, display the validation feedback based on the errors data property.

Example:

```
<template>
  <div>
    <label for="name">Name:</label>
    <input type="text" id="name" v-model="name">
    <div class="error-message">{{ errors.name }}</div>
    <label for="email">Email:</label>
    <input type="email" id="email" v-model="email">
    <div class="error-message">{{ errors.email }}</div>
    <button @click="validateForm">Submit</button>
  </div>
</template>
```

5.Here, we've added <div> elements to display the error messages for the name and email inputs. The error messages will be shown only if there is a corresponding error in the errors data property.

6.Trigger form validation: Add a button or any other event to trigger the form validation logic implemented in the validateForm method.

In the example above, we added a Submit button that calls the validateForm method when clicked. You can also trigger the validation on other events like form submission or input field blur.

7.Style the validation feedback (optional): You can apply CSS styles to the error message elements or add custom CSS classes to style the inputs based on their validation state. This step is optional but can improve the visual presentation of the validation feedback.

Submit form data

To submit form data in Vue.js, you can follow these steps:

1.Set up a Vue component: Create a new Vue component or open an existing one where you want to handle form submission.

2.Define the data properties: In the component's <script> section, define the data properties that represent the form inputs and the data you want to submit.

Example:

```
<script>
export default {
  data() {
    return {
      name: '',
      email: ''
    };
  }
}
</script>
```

In this example, name and email represent the form inputs.

2.**Implement form submission logic:** Create a method to handle the form submission. This method will be triggered when the form is submitted, and you can perform any necessary actions such as sending the form data to a server or performing client-side processing.

Example:

```
<script>
export default {
  data() {
    return {
      name: '',
      email: ''
    };
  },
  methods: {
    submitForm() {
      const formData = {
        name: this.name,
        email: this.email
      };
      // Perform actions with formData (e.g., API call, client-side processing)
      console.log(formData);
    }
  }
}
</script>
```

3.In the submitForm method, we create an object formData and assign the values of name and email from the data properties. You can perform any necessary actions using this form data object.

4.**Set up the form in the template:** In the component's template, define the form structure and bind the form inputs to the corresponding data properties using the v-model directive.

Example:

```
<template>
  <form @submit.prevent="submitForm">
    <label for="name">Name:</label>
    <input type="text" id="name" v-model="name">
    <label for="email">Email:</label>
    <input type="email" id="email" v-model="email">
    <button type="submit">Submit</button>
  </form>
</template>
```

5.Here, we bind the name and email inputs to the name and email data properties using v-model. The @submit.prevent event listener ensures that the default form submission behavior is prevented, allowing you to handle the submission with your custom method.

6.**Trigger form submission:** Use a submit button or any other event to trigger the form submission logic implemented in the submitForm method.

In the example above, the Submit button triggers the form submission when clicked. Alternatively, you can use other events like form submission on pressing Enter key or any custom event.

7.**Handle the form data in the submitForm method**: Inside the submitForm method, you can handle the form data as required. This might involve making an API call to send the data to a server, performing client-side validation, or any other necessary processing.

Practical Activity 2.4.7: Display JSON data in a table

Task:

1. Read key reading 2.4.7 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.4.2), you are requested to go to the computer lab to display JSON data in a table. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to display JSON data in a table.

5. Referring to the steps provided on task 3, display JSON data in a table.

6. Present your work to the trainer and whole class

Key readings 2.4.7.: Display JSON data in a table

To display JSON data in a table, you can follow these steps:

1. Parse the JSON data: Start by parsing the JSON data into a JavaScript object. If you're working in a browser environment, you can use the JSON.parse() function. If you're using a server-side environment like Node.js, you can use the built-in JSON.parse() function as well.

Example (in JavaScript):

```
const jsonData = '{"name": "John", "age": 30, "city": "New York"}';

const data = JSON.parse(jsonData);
```

2. Identify the table structure: Determine the structure of the table you want to display. This includes the column headers and the data cells. You can use the keys from the parsed JSON object to populate the column headers.

3. Create the HTML table structure: Use HTML to create the table structure. You'll need to define the <table>, <thead>, <tbody>, and appropriate <tr> and <td> elements.

Example:

```
  <div v-if="submitted">

    <h3>Submitted Data:</h3>
```

```
<table>
  <tr>
    <th>Name</th>
    <th>Sex</th>
    <th>Address</th>
    <th>Phone</th>
    <th>Age</th>
    <th>Trade</th>
    <th>Level</th>
    <th>Father's Name</th>
    <th>Mother's Name</th>
  </tr>
  <tr>
    <td>{{ formData.name }}</td>
    <td>{{ formData.sex }}</td>
    <td>{{ formData.address }}</td>
    <td>{{ formData.phone }}</td>
    <td>{{ formData.age }}</td>
    <td>{{ formData.trade }}</td>
    <td>{{ formData.level }}</td>
    <td>{{ formData.fatherName }}</td>
    <td>{{ formData.motherName }}</td>
  </tr>
```

4. Generate table rows dynamically: Iterate over the parsed JSON data and generate the table rows dynamically. For each object in the JSON data, create a new <tr> element and populate the <td> cells with the corresponding values.

Example:

```
const tbody = document.querySelector('tbody');
```

```
for (const key in data) {
  const row = document.createElement('tr');
  const value = data[key];
  const cell = document.createElement('td');
  cell.textContent = value;
  row.appendChild(cell);
  tbody.appendChild(row);
}
```

5. Append the table to the document: Once you've created the table structure with the populated rows, append it to the appropriate element in your document.

For example, you can append it to a <div> element with an id attribute.

Example:

```
const tableContainer = document.querySelector('#tableContainer');
tableContainer.appendChild(table);
```

6. Style the table (optional): You can apply CSS styles to the table to enhance its appearance or match the overall design of your application. This step is optional but can improve the table's visual presentation.

```
<style scoped>
/* Add your styles here */
form {
  display: flex;
  flex-direction: column;
  max-width: 300px; /* Adjust the width as needed */
  margin: auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
  background-color: #f9f9f9;
}
.form-group {
  margin-bottom: 15px;
}
label {
  margin-bottom: 5px;
  font-size: 0.9em; /* Reduce the font size */
}
input, select {
  padding: 5px;
  font-size: 0.9em; /* Reduce the font size */
  width: 100%;
  box-sizing: border-box;
}
.submit-button {
  padding: 8px 15px;
  font-size: 0.9em; /* Reduce the font size */
  color: #fff;
  background-color: #007bff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}
.submit-button:hover {
  background-color: #0056b3;
```

```
}


.submit-button:active {
  background-color: #004494;
}
.submit-button:focus {
  outline: none;
}
table {
  width: 100%;
  border-collapse: collapse;
  margin: 20px auto;
  max-width: 600px; /* Adjust the width as needed */
}
th, td {
  border: 1px solid #ddd;
  padding: 8px;
  text-align: left;
}
th {
  background-color: #f2f2f2;
}
</style>
```

Points to Remember

- In Vue.js 2.x, there are several main lifecycle hooks that allow developers to manage the lifecycle of Vue components effectively. There are the main lifecycle hooks available in Vue.js 2.x like:beforeCreate,created,beforeMount,mounted etc…
- JSON data in Vue.js can be utilized in various scenarios to enhance data handling and application functionality. There are various scenarios where JSON data can be used in Vue.js, such as: API Data, Component Props, State Management, Configuration.
- To apply Vue lifecycle methods in a Vue.js application, follow these steps:

  1. Understanding Vue Lifecycle Methods

  2. Implement Lifecycle Methods in Vue Components

  3. Use Lifecycle Methods for Data Initialization

  4. Handle DOM Manipulation in Lifecycle Hooks

  5. Clean Up Resources in Lifecycle Hooks

  6. Test and Debug

- To display JSON data in a table, you can follow these steps:

  1. Parse the JSON data

  2. Identify the table structure

  3. Create the HTML table structure

  4. Style the table (optional)

- HTML forms consist of various elements that facilitate the collection and submission of user input. There are key elements commonly used in HTML forms: Form, input, type, textarea etc…
- In Vue.js, input binding refers to the process of binding form input elements to data properties in the Vue instance. There are different ways to achieve input binding in Vue.js, allowing for dynamic updates and synchronization between the input fields and the underlying data. There are the types of input binding in Vue.js: v-modeDirective,value` and `@input` Shorthand, Checkbox Binding. Radio Button Binding.
- To create a form in Vue.js, you can follow these steps:

  1. Set up a Vue.js project

  2. Create a Vue component

3. Define the form template
4. Define the form data and methods
5. Register and use the component
6. Style the form (optional)
7. Run the Vue.js application

Application of learning 2.4.

As software developer you have been tasked to create a Vue.js application that fetches JSON data, displays the data in a table, and includes a form to add new data to the table. This project will help your school to manage student’s data including: Names, Address, Email, Phone number and DOB.

Your project has to include Vue.js lifecycle hooks, handling JSON data, and input binding.

Indicative content 2.5: API Request

Duration: 6 hrs

Theoretical Activity 2.5.1: Introduction to API Request

Tasks:

1. You are requested to answer the following questions related to the description of API:

   I. What do you understand about the term API?

   II. What are the different types of API and how do they differ from each other?

   III. Describe key aspects or components of APIs.

   IV. What are main benefits of using APIs in your application?

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.5.1. In addition, ask questions where necessary.

Key readings 2.5.1.: Introduction to API Request

API: API stands for Application Programming Interface. It defines a set of rules and protocols that allow one software application to interact with another.

APIs specify the methods and data formats that applications can use to request and exchange information, enabling seamless integration and communication between different software systems. APIs are commonly used in web development to connect web applications with external services and resources.

An API, or Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate and interact with each other. It defines the methods and data formats that applications can use to request and exchange information. APIs are used to enable the integration of different systems and services, allowing them to work together seamlessly.

APIs are commonly used in web development to enable interaction between web applications and external services or databases. They can also be used in desktop and mobile applications, operating systems, and other types of software.

There are several types of APIs, including:

1.**Web APIs:** These APIs are accessible over the internet using HTTP protocols and are designed to be used by web applications. They allow web services to communicate with each other and enable the integration of third-party services into web applications.

2.**Library APIs:** These APIs are provided as libraries of code that developers can use to perform specific functions in their applications. Library APIs are typically written in programming languages like Java, Python, or C++.

3.**Operating System APIs:** These APIs provide access to the underlying functions of an operating system. They allow applications to interact with hardware devices, file systems, and other system-level features.

4.**Database APIs:** These APIs allow applications to interact with databases, including reading, writing, updating, and deleting data. They provide a way for applications to query databases and retrieve specific information.

Here are key aspects and components of an API:

1. Endpoint:

●An endpoint is a specific URL or URI (Uniform Resource Identifier) where an API can be accessed. Each endpoint typically corresponds to a specific function or resource.

2. Request:

●A request is made by a client application to the API endpoint, specifying the desired action, parameters, and data.

3. HTTP Methods:

●APIs often use standard HTTP methods like GET, POST, PUT, and DELETE to define the actions they can perform. For example, a GET request is used to retrieve data, while a POST request is used to submit data.

4. Parameters:

●Parameters are additional pieces of information included in a request to provide more context or specify criteria. They are often included in the URL or request body.

5. Headers:

●Headers contain additional information about the request or response. They can include authentication tokens, content type, and other metadata.

6. Response:

●The API responds to a client's request with data or status information. The response is typically in a standardized format such as JSON or XML.

7. Status Codes:

●HTTP status codes indicate the success or failure of a request. Common status codes include 200 (OK), 201 (Created), 400 (Bad Request), and 404 (Not Found).

8. Authentication:

●Many APIs require authentication to ensure that only authorized users or applications can access the data or services. This can be done using API keys, OAuth tokens, or other authentication mechanisms.

Benefits of APIs

Modularity: Different parts of an application can be developed and maintained independently.

Interoperability: Enables different systems to work together.

Scalability: APIs can handle increased loads by scaling services independently.

Security: Controlled access through authentication mechanisms.

Theoretical Activity 2.5.2: Description of Axios

Tasks:

1. You are requested to answer the following questions related to the Axios:

   I. What do you understand about the term Axios?

   II. What are the advantages of using Axios?

   III. What are the key features of Axios?

2. Provide the answer to the asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.5.2. In addition, ask questions where necessary.

Key readings 2.5.2.: Description of Axios

Definition of Axios

Axios is a popular JavaScript library that is commonly used for making HTTP requests from web browsers or Node.js.

Advantages of Axios

It provides a simple and powerful way to send asynchronous HTTP requests to a server and handle the responses.

Axios supports various features such as request cancellation, interceptors, automatic request transformation, and more.

In Vue.js, Axios is often used to make API requests from the client-side to fetch data from a server or send data to be processed. Here's a description of Axios and its usage in Vue.js.

By integrating Axios with Vue.js, you can create dynamic and interactive web applications that fetch and display data from APIs, making your applications more powerful and versatile.

Features of Axios:

1.**Promise-based:** Axios is built on top of promises, making it easy to work with asynchronous code and handle responses in a clean and concise manner.

2.**Interceptors:** Axios allows you to intercept requests or responses before they are handled by then or catch. This can be useful for tasks like adding authentication tokens to headers or logging requests and responses.

3.**Automatic JSON data transformation:** Axios automatically parses JSON responses, eliminating the need to manually parse JSON data.

4.**Browser and Node.js support:** Axios can be used both in web browsers and Node.js environments, making it versatile for different types of applications.

Theoretical Activity 2.5.3: Description of CRUD operation

Tasks:

1. You are requested to answer the following questions related to the CRUD Operation:

   I. What do you understand about CRUD operation?

   II. Explain the usage of Create, Retrieve, Update, and Delete operations (CRUD)

2. Provide the answer to the asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. For more clarification, read the key readings 2.5.3. In addition, ask questions where necessary.

Key readings 2.5.3.: Description of CRUD operation

Definition

CRUD stands for Create, Read, Update, and Delete. It refers to the basic operations that can be performed on data in a persistent storage system, such as a database. In the context of Vue.js, CRUD operations are commonly used when interacting with APIs or managing data within the application. Here's a description of each CRUD operation in Vue.js:

Uses of each component of CRUD

1.Create (C): The Create operation involves adding new data to the system. In Vue.js, this typically involves sending a request to an API endpoint to create a new resource. For example, you can have a form in a Vue component that collects user input, and upon submission, the data is sent to the server to create a new record. After a successful response, you can update the component's data or trigger a redirect to reflect the newly created resource.

2.Read (R): The Read operation involves retrieving data from the system. In Vue.js, this is often done by making a request to an API endpoint to fetch data. The retrieved data can then be used to populate components and display information to the user. Vue's data properties or state management solutions like Vuex can be used to store and manage the fetched data.

3.Update (U): The Update operation involves modifying existing data in the system. In Vue.js, this is typically done by sending a request to an API endpoint with updated data. For example, you can have a form that allows users to edit information, and upon submission, the updated data is sent to the server. After a successful response, you can update the corresponding data in the Vue component or trigger a refresh to reflect the changes.

4.Delete (D): The Delete operation involves removing data from the system. In Vue.js, this is usually accomplished by sending a request to an API endpoint with the identifier of the data to be deleted. After a successful response, you can remove the corresponding data from the Vue component or trigger a refresh to reflect the deletion.

Practical Activity 2.5.4: Install Axios file.

Task:

1. Read key reading 2.5.4 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.5.2), you are requested to go to the computer lab to Install Axios file and configure in API helper file. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to Install Axios file and configure in API helper file.

5. Referring to the steps provided on task 3, Install Axios file and configure in API helper file.

6. Present your work to the trainer and whole class

Key readings 2.5.4.: Install Axios file.

To install and configure Axios in an API helper file, you can follow these steps:

1.Install Axios:

Make sure you have Node.js and npm (Node Package Manager) installed on your machine. Open your terminal or command prompt and run the following command to install Axios:

npm install axios or **yarn add axios**

2.Create an API Helper file:

Create a new JavaScript file (e.g., apiHelper.js) where you'll define your Axios configuration and helper functions.

3.Import Axios:

In your apiHelper.js file, import Axios at the top:

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

4.Configure Axios:

You can set up global configurations for Axios, such as a default base URL or headers. Here's an example of setting a base URL for your API requests:

```
axios.defaults.baseURL = 'https://api.example.com';
```

You can also configure headers, authentication tokens, interceptors, and other settings as per your project requirements. Refer to the Axios documentation for more details on available configuration options.

5.**Create helper functions:**

In your apiHelper.js file, you can define various helper functions to make API requests using Axios. Here's an example of a simple GET request:

```
async function getData(endpoint) {

  try {

    const response = await axios.get(endpoint);

    return response.data;

  } catch (error) {

    // Handle error

    console.error(error);

  }

}

module.exports = {

  getData

};
```

This example uses the axios.get() method to perform a GET request to the specified endpoint. The async/await syntax allows you to write asynchronous code in a synchronous style. If the request is successful, the response data is returned. Otherwise, the error is logged to the console.

6.Usage:

In other parts of your application, you can import the API helper functions and use them to make API requests. Here's an example of how you can use the getData function defined in apiHelper.js:

```
const apiHelper = require('./apiHelper');

// Call the getData function with the desired endpoint

apiHelper.getData('/users')

  .then(data => {

    // Use the retrieved data

    console.log(data);

  });
```

In this example, the getData function is called with the /users endpoint. The returned data is then logged to the console. You can modify the usage according to your API endpoints and data handling requirements.

Practical Activity 2.5.5: Creating JSON file

Task:

1. Read key reading 2.5.5 and ask clarification where necessary

2. Referring to the previous practical activities (2.4.7), you are requested to go to the computer lab to create JSON files in an existing Vue Project folder. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create JSON files in an existing Vue Project folder.

5. Referring to the steps provided on task 3, create JSON files in an existing Vue Project folder.

6. Present your work to the trainer and whole class

Key readings 2.5.5.: Creating JSON file

1.Create a JSON file

You can create a JSON file manually using a text editor, or you can create it programmatically using your preferred programming language. Here's an example of creating a JSON file named "data.json" with some sample data:

```
[
 {
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com"
 },
 {
  "name": "Jane Smith",
  "age": 25,
  "email": "jane@example.com"
 },
 {
  "name": "Bob Johnson",
  "age": 35,
  "email": "bob@example.com"
 }
]
```

2.Read the JSON file:

Depending on the programming language you're using, there are different ways to read the contents of a JSON file. You need to read the file and parse its contents into an object or array in memory. Here's an example using Python:

```
import json
```

```
# Read the JSON file
with open('data.json') as file:
    data = json.load(file)
```

3.Display data in a table:

Once you have the JSON data in memory, you can display it in a table format. The approach to displaying data in a table varies depending on the programming language and the UI framework/library you're using. Here's an example using HTML and JavaScript:

```
 <!DOCTYPE html>
<html>
<head>
    <title>JSON to Table</title>
    <style>
        table {
            border-collapse: collapse;
        }
        th, td {
            border: 1px solid black;
            padding: 8px;
        }
    </style>
</head>
<body>
    <table id="myTable">
        <thead>
            <tr>
                <th>Name</th>
                <th>Age</th>
            <th>Email</th>
          </tr>
        </thead>
        <tbody>
        </tbody>
    </table>
    <script>
        // Read the JSON file (assuming the data variable contains the JSON data)
        var jsonData = JSON.parse(`[{"name": "John Doe","age": 30,"email":
"john@example.com"},{"name": "Jane Smith","age": 25,"email":
"jane@example.com"},{"name": "Bob Johnson","age": 35,"email":
"bob@example.com"}]`);
        var tableBody = document.querySelector('#myTable tbody');
        // Iterate over the JSON data and add rows to the table
        for (var i = 0; i < jsonData.length; i++) {
          var row = tableBody.insertRow();
          var nameCell = row.insertCell();
          nameCell.textContent = jsonData[i].name;
          var ageCell = row.insertCell();
          ageCell.textContent = jsonData[i].age;
          var emailCell = row.insertCell();
          emailCell.textContent = jsonData[i].email;       }
    </script>
</body>
</html>
```

Note: In this example, the JSON data is directly embedded in the HTML file for simplicity. However, in a real-world scenario, you would load the JSON data dynamically using an AJAX request or by fetching it from an API.

Practical Activity 2.5.6: Integrating API calls from backend

Notes to the trainer

1. Read key reading 2.5.6 and ask clarification where necessary

2. Referring to the previous practical activities (2.5.5), you are requested to go to the computer lab to integrate the provided API with your project in order to access the backend data. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to integrate API in an existing Vue Project folder.

5. Referring to the steps provided on task 3, integrate API in an existing Vue Project folder.

6. Present your work to the trainer and whole class

Key readings 2.5.6.: Integrating API calls from backend

To integrate API calls into your Vue components you have to follow those steps:

- **Set Up Axios in Your Vue Project**
  To install axios, open your project in terminal or in IDE and type that command **npm install axios** if you are using node package manager or **yarn add axios** if you are using yarn
- **Create a service file to manage API calls (ex. Student.js)**

The service files have to be a javascript file used to manage API call from backend means depending on the base URL that have been provided by backend developer you have to include that base URL in your service file.

For example:

Suppose you have to integrate the API that will be used to connect the frontend and backend of student registration where the information of student has to be recorded by using form containing the following information:

```
The base URL that have been developed by  backend is API_URL = 'http://localhost:3000/api/students'
```

Create a studentservices file in services folder of your project and assign the following codes:

```
import axios from 'axios';   // for declaring the installed axios

const API_URL = 'http://localhost:3000/api/students';  // the base URL or API developed by backend

export default {

  createStudent(student) {

    return axios.post(API_URL, student);

  },

  getAllStudents() {

    return axios.get(API_URL);

  },

  getStudentById(id) {

    return axios.get(`${API_URL}/${id}`);

  },

  updateStudent(id, student) {

    return axios.put(`${API_URL}/${id}`, student);

  },

  deleteStudent(id) {

    return axios.delete(`${API_URL}/${id}`);

  }

};
```

- **Create Vue Components**
- **A component for creating and editing records.**

Create a component in components folder that will be used to record the information of student. That form will contain the following field as given to you by backend developer:

name, sex, address, dob, phone, trade, level.

Example code:

```
<template>
  <div>
    <h2>{{ isEdit ? 'Edit' : 'Add' }} Student</h2>
    <form @submit.prevent="submitForm">
      <!-- Form Fields -->
      <input v-model="student.name" placeholder="Name" required /> <br><br>
      <input v-model="student.sex" placeholder="Sex" required /><br><br>
      <input v-model="student.address" placeholder="Address" required /> <br><br>
      <input v-model="student.dob" type="date" placeholder="DOB" required /> <br><br>
      <input v-model="student.phone" placeholder="Phone" required /> <br><br>
      <input v-model="student.trade" placeholder="Trade" required /> <br><br>
      <input v-model="student.level" placeholder="Level" required /> <br><br>
      <button type="submit">{{ isEdit ? 'Update' : 'Submit' }}</button>
    </form>
    <!-- Success Message -->
    <p v-if="successMessage" class="success-message">{{ successMessage }}</p>
  </div>
</template>


<script>
import studentservices from '@/services/studentservices';


export default {
  props: {
    editStudent: Object,
  },
  data() {
    return {
      student: this.editStudent || {
        name: '',
        sex: '',
        address: '',
        dob: '',
        phone: '',
        trade: '',
        level: '',
      },
      isEdit: !!this.editStudent,
      successMessage: '', // Success message
    };
  },
  watch: {
    editStudent(newVal) {
      if (newVal) {
        this.student = { ...newVal };
        this.isEdit = true;
      } else {
        this.resetForm();
      }
    },
  },
  methods: {
    submitForm() {
      const action = this.isEdit ? studentservices.updateStudent(this.student.id,
this.student) : studentservices.createStudent(this.student);
      action.then(() => {
        this.successMessage = this.isEdit ? 'Student updated correctly!' : 'Student
inserted correctly!';
        this.$emit('refresh'); // Emit event to refresh the student list
        this.resetForm();
      }).catch((error) => {
        console.error(error);
      });
    },
    resetForm() {
      this.student = {
        name: '',
        sex: '',
        address: '',
        dob: '',
        phone: '',
        trade: '',
        level: '',
      };
      this.isEdit = false;
      this.successMessage = '';
    },
  },
};
```

```
</script>
<style scoped>
.form-container {
  background-color: #f9f9f9;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  max-width: 400px;
  margin: auto;
  align-content: center;
}
.student-form {
  display: flex;
  flex-direction: column;
  gap: 15px;
}
.form-group {
  display: flex;
  flex-direction: column;
}
.form-group label {
  font-weight: bold;
  margin-bottom: 5px;
}
.form-group input {
  padding: 8px;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 14px;
}
.submit-button {
  background-color: #007bff;
  color: white;
  padding: 10px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}
.submit-button:hover {
  background-color: #0056b3;
}
.success-message {
  color: green;
  margin-top: 10px;
  text-align: center;
}
</style>
```

- **A component for displaying the list**. In component folder and assign the following code.

```
<template>
  <div>
    <h2>Student List</h2>
    <table>
      <thead>
  <tr>
    <th>ID</th>
    <th>Name</th>
    <th>Sex</th>
    <th>Address</th>
    <th>DOB</th>
    <th>Phone</th>
    <th>Trade</th>
    <th>Level</th>
    <th>Actions</th>
  </tr>
</thead>
<tbody>
  <tr v-for="student in students" :key="student.id">
    <td>{{ student.id }}</td>
    <td>{{ student.name }}</td>
    <td>{{ student.sex }}</td>
    <td>{{ student.address }}</td>
    <td>{{ student.dob }}</td>
    <td>{{ student.phone }}</td>
    <td>{{ student.trade }}</td>
    <td>{{ student.level }}</td>
    <td>
      <button @click="editStudent(student)">Edit</button>
      <button @click="deleteStudent(student.id)">Delete</button>
    </td>
  </tr>
```

```
      </tbody>
    </table>

    <!-- Success Message -->
    <p v-if="successMessage" class="success-message">{{ successMessage }}</p>
  </div>
</template>

<script>
import studentservices from '@/services/studentservices';
export default {
  data() {
    return {
      students: [], // Array to store all students
      successMessage: '', // Success message
    };
  },
  methods: {
    fetchStudents() {
      studentservices.getAllStudents()
        .then((response) => {
          this.students = response.data;
        })
        .catch((error) => {
          console.error(error);
        });
    },
    editStudent(student) {
      this.$emit('edit', student); // Emit event to edit the selected student
    },
    deleteStudent(id) {
      studentservices.deleteStudent(id)
        .then(() => {
          this.successMessage = 'Student deleted successfully!';
          this.fetchStudents(); // Refresh the student list after deletion
        })
        .catch((error) => {
          console.error(error);
        });
    }
  },
  created() {
    this.fetchStudents(); // Fetch all students when the component is created
  },
  watch: {
    successMessage() {
      if (this.successMessage) {
        setTimeout(() => {
          this.successMessage = ''; // Clear the message after a few seconds
        }, 3000);
      }
    }
  }
```

```
};
</script>
<style scoped>
.table-container {
  margin: 20px auto;
  padding: 20px;
  background-color: #f9f9f9;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  max-width: 1000px;
}
.student-table {
  width: 100%;
  border-collapse: collapse;
  margin-top: 15px;
}
.student-table th, .student-table td {
  padding: 12px;
  text-align: left;
  border-bottom: 1px solid #ddd;
}
.student-table th {
  background-color: #007bff;
  color: white;
}
.student-table tr:hover {
  background-color: #f1f1f1;
```

```
}

</style>
```

- Set Up Routing (Configure routes in your router.js (or equivalent) file to map to these components.) Remember that before you have to install vue-router package.

For the routing setting, it will contain all routes that have been used.

For example, for our case student registration

```
import Vue from 'vue';

import Router from 'vue-router';

import studentform_component from '@/components/studentform_component.vue';

import studentlist_component from '@/components/studentlist_component.vue';

import studentdetail_component from '@/components/studentdetail_component.vue';

Vue.use(Router);

export default new Router({

 routes: [

  {

   path: '/',

   name: 'StudentList',

   component: studentlist_component

  },

  {

   path: '/add',

   name: 'StudentForm',

   component: studentform_component

  },

  {
   path: '/student/:id',
   name: 'StudentDetails',
   component: studentdetail_component
  }
 ]
});
```

- **Handle API Responses**
  - Display success or error messages based on the API call results.
  - Manage loading states to enhance user experience.

  Example: Handling API Responses in StudentForm.vue:

```
methods: {
 submitForm() {
  this.isLoading = true; // Show loading spinner (if implemented)
  studentService.createStudent(this.student)
   .then(response => {
    this.isLoading = false;
    alert('Student created successfully!');
    this.$emit('studentAdded', response.data);
   })
   .catch(error => {
    this.isLoading = false;
    alert('Error creating student: ' + error.message);
   });
 }
}
```

- **Testing and Debugging**

  Test your application to ensure API calls are working as expected.

  Use browser developer tools to monitor network requests and debug issues.

Points to Remember

- There are different types of APIs commonly used in software development such as Web APIs, Library APIs, Operating System APIs, Database APIs.
- Axios is a popular JavaScript library used for making HTTP requests from the browser or Node.js. There are the key features of Axios like: Promise-based, Interceptors, Browser and Node.js support, Automatic JSON data transformation.
- The CRUD acronym stands for Create, Read, Update, and Delete, which are the four basic functions that can be performed on a database or data storage system.
- **To install and configure Axios in an API helper file, you can follow these steps:**
  1. Install Axios
  2. Create an API Helper file
  3. Import Axios
  4. Configure Axios
- **To create a JSON file and display its data in a table, you can follow these steps:**
  1. Create a JSON file
  2. Read the JSON file
  3. Display data in a table
- **To integrate API calls into your Vue components you have to follow the following steps:**
  - ✓ Set Up Axios in Your Vue Project
  - ✓ Create a service file to manage API calls (ex. Student.js)
  - ✓ Create Vue Components
    - A component for creating and editing records.
    - A component for displaying the list.
  - ✓ Set Up Routing (Configure routes in your router.js (or equivalent) file to map to these components.)
  - ✓ Handle API Responses
  - ✓ Testing and Debugging

Application of learning 2.5.

FGB TSS is one of technical secondary schools located in Muhanga district. The school is facing with problem of data collection about student's records that is done manually, as a developer who has skills in Vue Js framework, you have been hired to develop a web app that will be used to collect those needed data, the data have to be stored in one file as JSON file.

Indicative content 2.6: Manage Data Using State Management

Duration: 5 hrs

Theoretical Activity 2.6.1: Description of key concepts used in state management

Tasks:

1. You are requested to answer the following questions related to the Key concepts used in state management:

   I. What do you understand about the term state management?

   II. Describe the key concepts involved in state management, such as getter, action, mutation, dispatch, and state.

   III. What are the benefits of state management?

   IV. Provide a list of state management libraries and explain each of them?

2. Provide the answer of asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. Listen carefully to the trainer's clarification and ask questions where necessary.

5. Read Theoretical activity 2.6.1 in the trainee manual for more clarification.

Key readings 2.6.1.: Description of key concepts used in state management

State management

State management is an essential concept in software development that allows you to efficiently manage and manipulate data within an application. There are several state management approaches available, and the choice depends on the framework or library you are using.

Here, I will provide a general overview of state management and a few popular techniques.

Getter

These refer to a concept within Vuex, which is the state management library for Vue.js applications. Getters allow you to access and compute values from the state in a Vuex store.

Action

These refer to functions that are used to trigger changes to the state of a Vue application.

Mutation

These refer to changes made to the state of a Vuex store. Vuex is a state management library for Vue.js that helps you manage the shared state of your application in a predictable and centralized way. Mutations are one of the core concepts in Vuex, and they are used to modify the state in a controlled and predictable manner.

Dispatch

The dispatch method is used to trigger actions in Vuex. Actions are functions that can perform asynchronous operations and then commit mutations to modify the state.

Benefits of state management

1.**Centralized Data:** State management allows you to maintain a centralized data store (state) that can be accessed and modified by multiple components. This ensures that the data remains consistent and up to date across the entire application.

2.**Improved Code Organization:** By separating the state from the components, you can keep your code more organized and maintainable. Components can focus on rendering and user interactions, while state management handles data-related concerns.

3.**Reactivity:** Vue's reactivity system allows you to bind components to the state. When the state changes, components that depend on it will automatically update, reducing the need for manual DOM manipulation and event handling.

4.**Cross-Component Communication**: State management makes it easy to communicate and share data between components that may not have a parent-child relationship. This is particularly useful for sibling or distant components.

5.**Global Access:** State management allows you to create a single source of truth that can be accessed from any component in your application. This simplifies data sharing and reduces the need for prop drilling or event buses.

6.**Time-Travel Debugging:** Many state management libraries, like Vuex, integrate with Vue Devtools, enabling powerful debugging features such as time-travel debugging. This allows you to inspect and rewind the application's state changes, making it easier to identify and fix issues.

7.**Optimized Performance:** State management libraries often come with optimizations for better performance. They can batch state changes and minimize unnecessary re-renders of components, leading to more efficient applications.

8.**Persistence:** State management libraries often provide options for data persistence, making it easier to save and load application state, which is important for maintaining a consistent user experience across sessions.

9.**Scalability:** As your application grows, state management makes it easier to handle the increased complexity of data management and component communication. It provides a scalable solution for maintaining a maintainable codebase.

10.**Easier Testing:** Separating state from components makes it easier to write unit tests. You can test your state management logic independently, and your components can be more easily mocked for testing.

State management Libraries

1.**Pinia:** Pinia is a state management library for Vue.js, a popular JavaScript framework for building user interfaces. It is designed to help developers manage the state of their Vue.js applications in a more organized and efficient way

2.**Redux:** Redux is a popular state management library for JavaScript applications. It is often used with React but can be used with other frameworks or libraries as well. Redux stores the entire application state in a single store object, and state mutations are performed through dispatched actions. It provides a predictable state management approach and enables easy debugging and time-travel debugging.

3.Vuex: Vuex is the state management library specifically designed for Vue.js applications. It provides a centralized store where all the application-level state is stored. Vuex uses mutations and actions to modify the state and provides reactivity, allowing components to update automatically when the state changes.

Theoretical Activity 2.6.2: Description of Vue DevTools in browser

Tasks:

1. You are requested to answer the following questions related to description of vue Devtools in browser.

   I. What do you understand about the term vue Devtools?

   II. What are the key features and functionalities offered by Vue Devtools, and could you provide an explanation for each?

2. Provide the answer to asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. Listen carefully to the trainer's clarification and ask questions where necessary.

5. Read Theoretical activity 2.6.2 in the trainee manual for more clarification.

Key readings 2.6.2.: Description of Vue DevTools in browser

Vue Devtools is a browser extension or standalone application designed to help developers debug and inspect Vue.js applications. It provides a rich interface to interact with Vue components, Vuex store, events, and more, making it easier to develop and maintain Vue applications.

Key Features and Functionalities of Vue Devtools

Component Inspector

Component Tree: Displays a hierarchical view of all Vue components in the application. You can inspect each component's data, props, computed properties, methods, and events.

State and Props: Allows you to view and modify the state and props of a component in real-time, which is useful for debugging and testing different scenarios.

Vuex Integration

State Management: Provides tools to inspect and modify the Vuex store state. You can view the current state, commit mutations, and dispatch actions directly from the Devtools.

Time Travel Debugging: Lets you travel back and forth through state changes to see how different actions affect the state over time. This is useful for understanding the flow of your application's state.

Event Tracking

Event Log: Logs all events emitted by Vue components, allowing you to see what events are fired and in what order. This helps in understanding the event flow in your application.

Event Payloads: You can inspect the payloads of events, making it easier to debug issues related to event data.

Performance Monitoring

Performance Metrics: Offers tools to measure and analyze the performance of your Vue application. You can see the time taken for component rendering and identify performance bottlenecks.

Routing Inspection

Vue Router Integration: If you're using Vue Router, the Devtools provide a way to inspect the router state, including the current route, route parameters, and navigation history.

Custom Inspector

Custom Panels and Inspectors: Allows developers to create custom panels and inspectors for their own plugins or libraries, extending the functionality of Vue Devtools.

State Snapshot and Export

State Snapshots: Enables taking snapshots of the current application state, which can be useful for debugging or sharing with other developers.

State Export/Import: You can export the state of your application and import it later, making it easier to reproduce bugs or continue debugging sessions.

Installation and Usage

Browser Extension: Available for Chrome and Firefox. You can install it from the respective browser's extension store.

- ✓ Chrome Web Store
- ✓ Firefox Add-ons

Standalone Application: Useful for inspecting applications that run in environments where browser extensions are not available, such as Electron or NativeScript.

Install via npm: npm install -g @vue/devtools

Run the standalone app: vue-devtools

Example of Usage

Inspecting Components: Open your application in the browser and open Vue Devtools from the browser's developer tools. Navigate to the "Components" tab to see the tree of Vue components. Select a component to view its data, props, and computed properties.

Vuex State Management: Go to the "Vuex" tab in Vue Devtools to see the current state of your Vuex store. You can dispatch actions or commit mutations from this tab and see how they affect the state.

Event Tracking: Use the "Events" tab to see all events emitted by your Vue components. This helps you understand the flow of events and debug issues related to event handling.

Theoretical Activity 2.6.3: Introduction to state Modules

Tasks:

1. You are requested to answer the following questions related to state modules

   I. What do you understand about state modules?
   II. What are the purposes of state modules in Vue.js?

2. Provide the answer to asked questions by writing them on paper.

3. Present your findings to your classmates and trainer

4. Listen carefully to the trainer's clarification and ask questions where necessary.

5. Read Theoretical activity 2.6.3 in the trainee manual for more clarification.

Key readings 2.6.3.: Introduction to state Modules

In Vue.js, especially when using Vuex for state management,

State modules are a way to organize the Vuex store into smaller, manageable pieces. This modular approach allows developers to divide the store into distinct modules, each with its own state, mutations, actions, and getters.

Purposes of State Modules in Vue.js

Scalability and Maintainability

Organized Codebase: State modules help keep the codebase organized, especially in large applications with complex state logic. Each module can manage a specific part of the application's state, making it easier to locate and manage code.

Separation of Concerns: By separating different concerns into different modules, developers can focus on specific parts of the state without being overwhelmed by the entire state management logic.

Namespace Management

Namespacing: Modules can be namespaced, meaning that the state, mutations, actions, and getters within a module can be scoped to that module. This prevents naming conflicts and makes it clear which part of the state each piece of logic pertains to.

Reusability

Reusable Modules: Modules can be designed to be reusable across different parts of the application. For example, an authentication module can be used in multiple applications with minimal changes.

Testing and Debugging

Isolated Testing: Since each module is independent, it can be tested in isolation, leading to simpler and more effective unit tests.

Easier Debugging: Debugging is easier as issues can be traced back to specific modules, allowing developers to quickly identify and fix bugs.

Improved Collaboration

Team Collaboration: In a team setting, different team members can work on different modules simultaneously without interfering with each other, improving productivity and reducing merge conflicts.

Practical Activity 2.6.4: Install Vue DevTool in a browser

Task:

1. Read key reading 2.6.4 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.6.2), you are requested to go to the computer lab to install vue DevTools in the browser. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to install vue DevTools in the browser.

5. Referring to the steps provided on task 3, install vue DevTools in the browser.

6. Present your work to the trainer and whole class

Key readings 2.6.4.: Install Vue DevTool in a browser

Install vue devtools

Considering google chrome browser you can follow the following steps:

1.**Open Google Chrome.**

2.Visit the Chrome Web Store (https://chrome.google.com/webstore/category/extensions) and search for "Vue.js devtools."

3.Find the Vue.js devtools extension in the search results.

4.Click the "Add to Chrome" button to install the extension.

5.A confirmation dialog will appear. Click "Add Extension" to install it.

6.Once installed, you'll see the Vue.js icon in the Chrome toolbar

To install Vue.js Devtools in Mozilla Firefox, you can follow these steps

1.Visit the Mozilla Firefox Add-ons Website: Go to the Mozilla Firefox Add-ons website (https://addons.mozilla.org/) and search for "Vue.js devtools" in the search bar.

2.Find the Vue.js Devtools Extension: Look for the official Vue.js Devtools extension in the search results. Make sure it's developed by the Vue.js core team or another reputable source.

3.Click on "Add to Firefox": Once you find the Vue.js Devtools extension, click on the "Add to Firefox" button next to it.

4.Install the Extension: Firefox will prompt you to confirm the installation. Click "Add" to install the Vue.js Devtools extension.

5.Access Vue Devtools: After the installation is complete, open your Vue.js application in Firefox. Right-click on the page, select "Inspect Element" from the context menu, and navigate to the "Vue" tab within the Developer Tools panel. There, you will find the Vue Devtools interface, allowing you to inspect components, monitor state, debug events, and analyze the performance of your Vue.js application.

Practical Activity 2.6.5: Installing and configuring State Management

Task:

1. Read key reading 2.6.5 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.6.1), you are requested to go to the computer lab to install state management. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to install state management.

5. Referring to the steps provided on task 3, install state management.

6. Present your work to the trainer and whole class

Key readings 2.6.5.: Installing and configuring of State Management

To install and configure state management the following steps are followed:

1. Create a Vue.js Project:

If you don't already have a Vue.js project, you can create one using Vue CLI. If you haven't installed Vue CLI, you can do so with the following command:

```
npm install -g @vue/cli
```

2. Then, create a new Vue.js project:

```
vue create my-vuex-app
```

3. **Follow the prompts to set up your project.**

4. **Install Vuex:**

Once your Vue.js project is set up, you can install Vuex by running the following command in your project directory:

```
npm install vuex
```

5. **Set up the Vuex Store:**

The Vuex store is where you define and manage the state of your application. Create a store by creating a JavaScript file, typically named `store.js`, in your project's source directory (e.g., `src/store/store.js`). In this file, you need to define your state, mutations, actions, and getters.

Example of how to set up a Vuex store:

```
// src/store/store.js

import Vue from "vue";

import Vuex from "vuex";

Vue.use(Vuex);

const store = new Vuex.Store({

 state: {

  // Your application's state goes here

  counter: 0,

 },

 mutations: {

  // Mutations for modifying the state

  increment(state) {

    state.counter++;

  },

 },

 actions: {
  // Actions for asynchronous operations
  incrementAsync({ commit }) {
   setTimeout(() => {
    commit("increment");
   }, 1000);
  },
 },
 getters: {
  // Getters for computed properties based on state
  doubleCounter(state) {
   return state.counter * 2;
  },
 },
});
```

4. Connect the Store to Your Vue App:

In your Vue app's main entry point (usually `src/main.js`), import the Vuex store and use it with your Vue instance.

```
// src/main.js
  import Vue from 'vue';
  import App from './App.vue';
  import store from './store/store';
  new Vue({
  render: (h) => h(App),
  store, // Register the Vuex store
  }).$mount('#app');
```

5. Use State in Your Components:

In your Vue components, you can access and modify the state by using computed properties, methods, and mutations. Here's an example component that uses the state defined in the Vuex store:

```
<!-- src/components/Counter.vue -->
<template>
 <div>
   <p>Counter: {{ counter }}</p>
   <p>Double Counter: {{ doubleCounter }}</p>
   <button @click="increment">Increment</button>
   <button @click="incrementAsync">Increment Async</button>
 </div>
</template>
<script>
export default {
 computed: {
   counter() {
     return this.$store.state.counter;
   },
   doubleCounter() {
     return this.$store.getters.doubleCounter;
   },
 },
 methods: {
   increment() {
     this.$store.commit("increment");
   },
   incrementAsync() {
     this.$store.dispatch("incrementAsync");
   },
  },
};
</script>
```

6. Now, you can use the `Counter` component in your application to interact with the state managed by Vuex.

Practical Activity 2.6.6: Define state modules

Task:

1. Read key reading 2.6.6 and ask clarification where necessary

2. Referring to the previous theoretical activities (2.6.3), you are requested to go to the computer lab to define state modules. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to define state modules.

5. Referring to the steps provided on task 3, define state modules.

6. Present your work to the trainer and whole class

Key readings 2.6.6.: Define state modules

To define a state module in Vuex, the state management library for Vue.js, follow these steps:

1. Install Vuex:

Ensure Vuex is installed in your Vue.js project. If not, install Vuex using npm or yarn:

```
npm install vuex
```

2. **Create a Vuex Store:**

In your project directory, create a new directory named `store`. Within the `store` directory, create a file named `index.js` (or any other name you prefer) where you will define your Vuex store.

3.**Define State:**

Inside the `index.js` file, define the initial state of your application. This is where you store the data that you want to manage centrally. Here's an example of defining a simple counter state:

```
const state = {
  count: 0
};
```

4.**Create Mutations:**

Mutations are functions that directly modify the state. Define mutations to update the state in a predictable way. Here's an example of a mutation to increment the counter:

```
const mutations = {
  increment(state) {
    state.count++;
  }
};
```

5. Define Actions:

Actions are functions that commit mutations. They can contain asynchronous operations. Define actions to handle more complex logic or API calls. Here's an example of an action to increment the counter:

```
const actions = {
  increment(context) {
    context.commit('increment');
  }
};
```

6. Set up Getters:

Getters are used to compute derived state based on the store's state. Define getters to access and compute values from the state. Here's an example of a getter to get the doubled count:

```
const getters = {
  doubleCount: state => {
    return state.count * 2;
  }
};
```

7. Create the Vuex Store:

Combine the state, mutations, actions, and getters into a Vuex store instance. Here's how you create the Vuex store with the defined state, mutations, actions, and getters:

```
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);
export default new Vuex.Store({
  state,
  mutations,
  actions,
  getters
});
```

8. Integrate Vuex Store in Vue Application:

Import the Vuex store in your main Vue component (e.g., `App.vue`) and use it in the Vue instance:

```
import Vue from 'vue';
import App from './App.vue';
import store from './store';
new Vue({
  store,
  render: h => h(App)
}).$mount('#app');
```

9.**Access State in Components:**

You can access the state, commit mutations, dispatch actions, and retrieve values from getters in your Vue components using Vuex helpers like `mapState`, `mapMutations`, `mapActions`, and `mapGetters`.

State modules help you separate and organize the state management for different parts of your application, making it more maintainable and scalable as your app grows. They also make it easier to collaborate with other developers on larger projects by providing a clear structure and organization of your application's data.

How to define a state module in Vuex:

Consider a simple shopping cart. We have a state module for managing a shopping cart, including the state itself, mutations to modify the state, actions to perform operations, and getters to compute derived state.

```
// shoppingCartModule.js
const state = {
  cartItems: []
};
const mutations = {
  addToCart(state, product) {
   state.cartItems.push(product);
  },
  removeFromCart(state, productIndex) {
   state.cartItems.splice(productIndex, 1);
  }
};
const actions = {
  addToCart({ commit }, product) {
   commit('addToCart', product);
  },
  removeFromCart({ commit }, productIndex) {
   commit('removeFromCart', productIndex);
  }
};
const getters = {
  cartTotal(state) {
   return state.cartItems.reduce((total, product) => total + product.price, 0);
  }
};
export default {
  state,
  mutations,
  actions,
  getters
};
```

Practical Activity 2.6.7: Getting data from state getters

Task:

1. Read key reading 2.6.7 and ask clarification where necessary

2. Referring to the previous practical activities you are requested to go to the computer lab to get data from state getters. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to get data from state getters.

5. Referring to the steps provided on task, get data from state getters.

6. Present your work to the trainer and whole class

Key readings 2.6.7.: Getting data from state getters

To get data from state getters in a Vuex store within a Vue.js application, you can follow these steps:

1. Define Getters in Vuex Store:

- First, make sure you have defined getters in your Vuex store. Getters are functions that compute derived state based on the store's state. Here's an example of defining a getter in your Vuex store:

```
const getters = {
  getUserById: (state) => (id) => {
    return state.users.find(user => user.id === id);
  }
};
```

2. Access Getters in Vue Components

- To access the data computed by getters in your Vue components, you can use the `mapGetters` helper provided by Vuex. Import `mapGetters` from Vuex in your component and map the getters you want to access.

3. Example of Getting Data from Getters in a Vue Component

- Here's an example of how you can access data computed by a getter in a Vue component:

```
<template>
  <div>
    <h2>User Details</h2>
    <p>User ID: {{ userId }}</p>
    <p>User Name: {{ userName }}</p>
  </div>
</template>
<script>
import { mapGetters } from 'vuex';
```

```
export default {
 computed: {
  ...mapGetters(['getUserById']),
  userId() {
   return this.getUserById(1).id; // Get user ID for user with ID 1
  },
  userName() {
   return this.getUserById(1).name; // Get user name for user with ID 1
  }
 }
};
</script>
```

4.Explanation:

In the above example, we use the `mapGetters` helper to map the `getUserById` getter from the Vuex store to the component's computed properties. We then access the computed values of `userId` and `userName` by calling the `getUserById` getter with the desired ID (in this case, ID 1) to retrieve the corresponding user's ID and name.

5.Display Data in the Component:

Once you have retrieved the data from the getter in the component, you can display it in the template using Vue.js data binding syntax (e.g., `{{ userId }}`, `{{ userName }}`).

It is possible to access data from state getters using computed properties. State getters are used to retrieve and compute derived data from your Vue.js state. Computed properties allow you to define these derived values and use them in your template or methods.

Examples on how to get data from state getters in Vue.js

Consider a Vuex store set up, and you want to access a getter from your store in your Vue component.

we define a Vuex getter called someGetter in the store, which computes a derived value based on the state. In your Vue component, you use a computed property called computedGetter to access this getter by using this.$store.getters.someGetter.

Now, when you use {{ computedGetter }} in your template, it will display the computed value from the getter.

Note: Computed properties automatically update when their dependencies change, which makes them an excellent choice for accessing derived data from your state getters

1. Define your Vuex store with a getter:

```
// store.js

import Vuex from 'vuex'

const store = new Vuex.Store({

 state: {

  // Your state data here

 },

 getters: {

  // Getter to compute derived data

  someGetter: state => {

  return state.someValue * 2; // Example computation

  }

 }

})
```

2. In your Vue component, use the computed property to access the getter:

```
<template>

 <div>

  <!-- Access the getter using a computed property -->

  <p>Computed Getter Value: {{ computedGetter }}</p>

 </div>

</template>
```

```
<script>

export default {

 computed: {

  // Define a computed property to access the getter

  computedGetter() {

  return this.$store.getters.someGetter;

  }

 }

}

</script>
```

Practical Activity 2.6.8: Commit mutations

Task:

1. Read key reading 2.6.8 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to commit Mutations. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to commit Mutations.

5. Referring to the steps provided on task, commit Mutations.

6. Present your work to the trainer and whole class

Key readings 2.6.8.: Commit mutations

Steps to define and commit mutations in a Vue.js application with Vuex:

Following the steps below, we are going to define mutations, to update the state in a Vuex store and commit those mutations from your Vue components when you want to change the application's state. This will help us to maintain a clear and predictable state management process in your Vue.js application

1. Setup Vuex Store:

Create a Vuex store in your Vue.js application. You typically do this in a separate JavaScript file as shown below:

```
  // store.js
  import Vue from 'vue';
  import Vuex from 'vuex';
  Vue.use(Vuex);
  const store = new Vuex.Store({
  state: {
          // Your application's state properties
  },
  mutations: {
          // Your mutation functions will be defined here
  },
  });
  export default store;
```

2. Define Mutations:

Inside the mutations section of your Vuex store, you can define mutation functions. Mutations take two arguments:

state: The current state of your application.

payload: The data you want to use to update the state.

Example of mutation:

```
  mutations: {
  incrementCounter(state) {
          state.counter++;
```

```
},
updateUsername(state, newUsername) {
        state.username = newUsername;
},
},
```

3. Commit Mutations:

To commit a mutation from a Vue component, you can use the commit method provided by the Vuex store. You should call commit with the mutation's name as the first argument and, optionally, a payload as the second argument if the mutation expects one.

Example

```
// Inside a Vue component
methods: {
increaseCounter() {
        this.$store.commit('incrementCounter');
},
changeUsername(newUsername) {
        this.$store.commit('updateUsername', newUsername);
},
},
```

4**. Access State:**

You can access the updated state in your Vue components using this.$store.state.

Example

```
// Inside a Vue component
computed: {
counter() {
        return this.$store.state.counter;
},
username() {
        return this.$store.state.username;
},
},
```

Practical Activity 2.6.9: Dispatch actions

Task:

1. Read key reading 2.6.9 and ask clarification where necessary

2. Referring to the previous practical activities you are requested to go to the computer lab to dispatch the Action from a Vue Component. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to dispatch the Action from a Vue Component.

5. Referring to the steps provided on task, dispatch the Action from a Vue Component.

6. Present your work to the trainer and whole class

Key readings 2.6.9.: Dispatch actions

Dispatching actions in Vuex involves several steps. Actions are functions that can contain arbitrary asynchronous operations and can commit mutations to change the state of your application.

Here's a step-by-step guide along with code examples:

Step 1: Define Your Vuex Store

First, create a Vuex store by importing Vue and Vuex, defining your state, mutations, actions, and getters.

```
// store.js

import Vue from 'vue';

import Vuex from 'vuex';
```

```
Vue.use(Vuex);
const state = {
// Your state properties go here
};
const mutations = {
  // Your mutation functions go here
};
const actions = {
  // Your action functions go here
};
const getters = {
  // Your getter functions go here
};
export default new Vuex.Store({
  state,
  mutations,
  actions,
  getters,
});
```

Step 2: Define Actions in the Vuex Store

Inside the actions object, define your action functions. Actions receive a context object which has the same methods and properties as the store instance. You can commit mutations from within actions.

```
<script>
// store.js
// ... (previous code)
const actions = {
 // Example action
```

```
  fetchDataFromAPI: async (context) => {
    try {
      const response = await fetch("https://api.example.com/data");
      const data = await response.json();
      context.commit("SET_DATA", data); // Commit a mutation to update state
    } catch (error) {
      console.error("Error fetching data:", error);
    }
  },
};
// ... (export store)
</script>
```

In the above code, fetchDataFromAPI is an action that fetches data from an API asynchronously and commits a mutation called SET_DATA to update the state.

Step 3: Dispatch the Action from a Vue Component

In your Vue component, you can dispatch the action using this.$store.dispatch('actionName'). Make sure the component has access to the Vuex store, either through a mixin or directly, depending on your project setup.

```
// YourComponent.vue
<template>
  <!-- Your component template goes here -->
</template>
<script>
export default {
  // ... (component properties and lifecycle hooks)
  methods: {
    fetchData() {
      this.$store.dispatch('fetchDataFromAPI'); // Dispatch the action
    },
  },
};
</script>
```

In the above code, the fetchData method in the component dispatches the fetchDataFromAPI action from the Vuex store.

Now, when you call this.$store.dispatch('fetchDataFromAPI'), it triggers the fetchDataFromAPI action in the store, which, in turn, can perform asynchronous operations and commit mutations to update the state of your application.

Points to Remember

- State management libraries play a crucial role in managing the state of an application, especially in complex front-end applications with multiple components and data interactions. There are some popular state management libraries like: Pinia, Redux, Vuex.
- Vue Devtools is a browser extension that provides a set of powerful tools for debugging and inspecting Vue.js applications. These are the key features and functionalities of Vue Devtools like: Component Tree, Component Inspection, State Management.
- State management modules in Vue.js, like Vuex, serve several important purposes in Vue.js applications. There are the key purposes of state modules in Vue.js like: Organization, Encapsulation, Reusability, Scalability.
- **Install vue Devtools** Considering google chrome browser you can follow the following steps:

  1. Open Google Chrome.

  2. Visit the Chrome Web Store (https://chrome.google.com/webstore/category/extensions) and search for "Vue.js Devtools."

  3. Find the Vue.js Devtools extension in the search results.

  4. Click the "Add to Chrome" button to install the extension.

  5. A confirmation dialog will appear. Click "Add Extension" to install it.

  6. Once installed, you'll see the Vue.js icon in the Chrome toolbar

- **To install and configure state management the following steps are followed:**

1. Create a Vue.js Project:

2. Then, create a new Vue.js project:

3. Follow the prompts to set up your project.

4. Install Vuex

5. Set up the Vuex Store

6. Now, you can use the `Counter` component in your application to interact with the state managed by Vuex.

- To define a state module in Vuex, the state management library for Vue.js, follow these steps:

  1. Install Vuex

  2. Create a Vuex Store

  3. Define State

  4. Create Mutations

  5. Define Actions

  6. Set up Getters

  7. Create the Vuex Store

  8. Integrate Vuex Store in Vue Application

- **To get data from state getters in a Vuex store within a Vue.js application, you can follow these steps:**

  1. Define Getters in Vuex Store

  2. Access Getters in Vue Components

  5. Display Data in the Component

- **Steps to define and commit mutations in a Vue.js application with Vuex:**

  1. Setup Vuex Store

  2. Define Mutations

  3. Commit Mutations

  4. Access State

- **Dispatching actions in Vuex involves several steps.**

  1. Define Vuex Store

2. Define Actions in the Vuex Store

3. Dispatch the Action from a Vue Component

Application of learning 2.6.

JAMTECH is a private company located in KAYONZA district that develop website for different institutions, due to different clients that they have, they are hiring a software developer who will develop for them the front end of DEREVA Hotel that will be used while connecting to back end. As Front-end developer, you are hired to perform that task. The front-end part will contain the followings:

- ✓ Home, About, Contacts, Registration, footer that contains different stakeholders address.
- ✓ All pages will have same header
- ✓ The registration form will be used while registering a room and display all records in table as JSON data and will be used once connecting to back end part
- ✓ Application will include routing between different views, fetching data from an API, displaying it in components, and managing state with Vuex.
- ✓ The application will have a 404 page that will help in case of connection to server.
- ✓ A structured folder with clear separation of concerns (components, views, etc.), functioning navigation using Vue Router, API integration using Axios, and state management using Vuex.

Learning outcome 2 end assessment

Written assessment

1. What is Vue.js? Select the best option

A. A programming language

B. A JavaScript framework for building user interfaces

C. A database management system

D. A server side scripting language

2. Which directive is used for two-way data binding in Vue.js?

A. v-model

B. v-bind

C. v-show

D. v-if

3. How do you create a new Vue instance?

A. new Vue()

B. createVueInstance()

C. Vue.create()

D. initVue()

4. What is the purpose of the `v-for` directive in Vue.js?

A. Conditional rendering

B. Two-way data binding

C. List rendering

D. Event handling

5. Which lifecycle hook is called after the Vue instance has been mounted to the DOM?

A. created

B. mounted

C. updated

D. destroyed

6. How can you communicate between parent and child components in Vue.js?

A. Props and events

B. Data and methods

C. Slots

D. Both A and C

7. What does the `v-bind` directive do in Vue.js?

A. Binds a class to an element

B. Binds an attribute to an expression

C. Binds a style to an element

D. Binds an event to a method

8. Which Vue.js feature allows you to transition between elements when they are inserted or removed from the DOM?

A. v-model

B. v-transition

C. v-if

D. v-show

9. What is the purpose of the `v-if` directive in Vue.js?

A. Conditional rendering

B. Two-way data binding

C. List rendering

D. Event handling

10. How can you include external libraries or plugins in a Vue.js project?

A. Using the `<script>` tag in HTML

B. Using the `import` statement in JavaScript

C. Both A and B

D. Vue.js does not support external libraries

11. What is the role of Vuex in Vue.js?

A. Routing

B. State management

C. Form validation

D. Animation

12. Which of the following is the correct way to bind a class dynamically based on a condition in Vue.js?

A. v-class

B. v-bind:class

C. v-style

D. v-if:class

13. How can you handle user input in Vue.js?

A. Using the `v-model` directive

B. Using the `v-bind` directive

C. Using the `v-show` directive

D. Using the `v-if` directive

14. What does the `vshow` directive do in Vue.js?

A. Renders an element only if a condition is true

B. Toggles the visibility of an element based on a condition

C. Binds a class to an element

D. Binds an attribute to an expression

15. How can you optimize performance in a Vue.js application?

A. Use computed properties

B. Use `v-if` instead of `v-show`

C. Use key attributes in `v-for` loops

D. All of the above

16. What is the purpose of the `v-on` directive in Vue.js?

A. Two-way data binding

B. Event handling

C. List rendering

D. Conditional rendering

17. Which Vue.js feature allows you to reuse content across different components?

A. Mixins

B. Filters

C. Directives

D. Components

18. What is the purpose of the `v-model` directive in Vue.js?

A. Bind a class to an element

B. Bind an attribute to an expression

C. Implement two-way data binding on form elements

D. Render a list of items

19. In Vue.js, what does the `nextTick` method do?

A. Delays the execution of a function until the next animation frame

B. Forces a rerender of the component

C. Executes a function after the DOM has been updated

D. Navigates to the next route in the application

20. Which Vue.js lifecycle hook is called when a component is about to be destroyed?

A. beforeCreate

B. beforeDestroy

C. destroyed

D. beforeMount

Practical assessment

BTICTHUB is a private company located in Huye district that develop website for different institutions, due to different clients that they have, they are hiring a software developer who will develop for them the front end of UBUMWE Hotel that will be used while connecting to back end. As Front-end developer, you are hired to perform that task. The front end part will contain the followings:

- ✓ Home, About, Contacts, Registration, footer that contains different stakeholders address.
- ✓ All pages will have same header
- ✓ The registration form will be used while registering a room and display all records in table as JSON data and will be used once connecting to back end part
- ✓ Application will include routing between different views, fetching data from an API, displaying it in components, and managing state with Vuex.
- ✓ The application will have a 404 page that will help in case of connection to server.
- ✓ A structured folder with clear separation of concerns (components, views, etc.), functioning navigation using Vue Router, API integration using Axios, and state management using Vuex.

References

Cuomo, S. (2023). Vue.js 3 for Beginners: Learn the Essentials of Vue.js 3 and Its Ecosystem to Build Modern Web Applications. Packt Publishing.

Passaglia, A. (2018). Vue.js 2 cookbook: Build modern, interactive web applications with Vue.js. Packt Publishing.

Hanchett, E., & Listwon, B. (2018). Vue.js in Action. Manning Publications.

Peacemaker19881. (n.d.). API Vue [Computer software]. GitHub. https://github.com/peacemaker19881/apivue.git

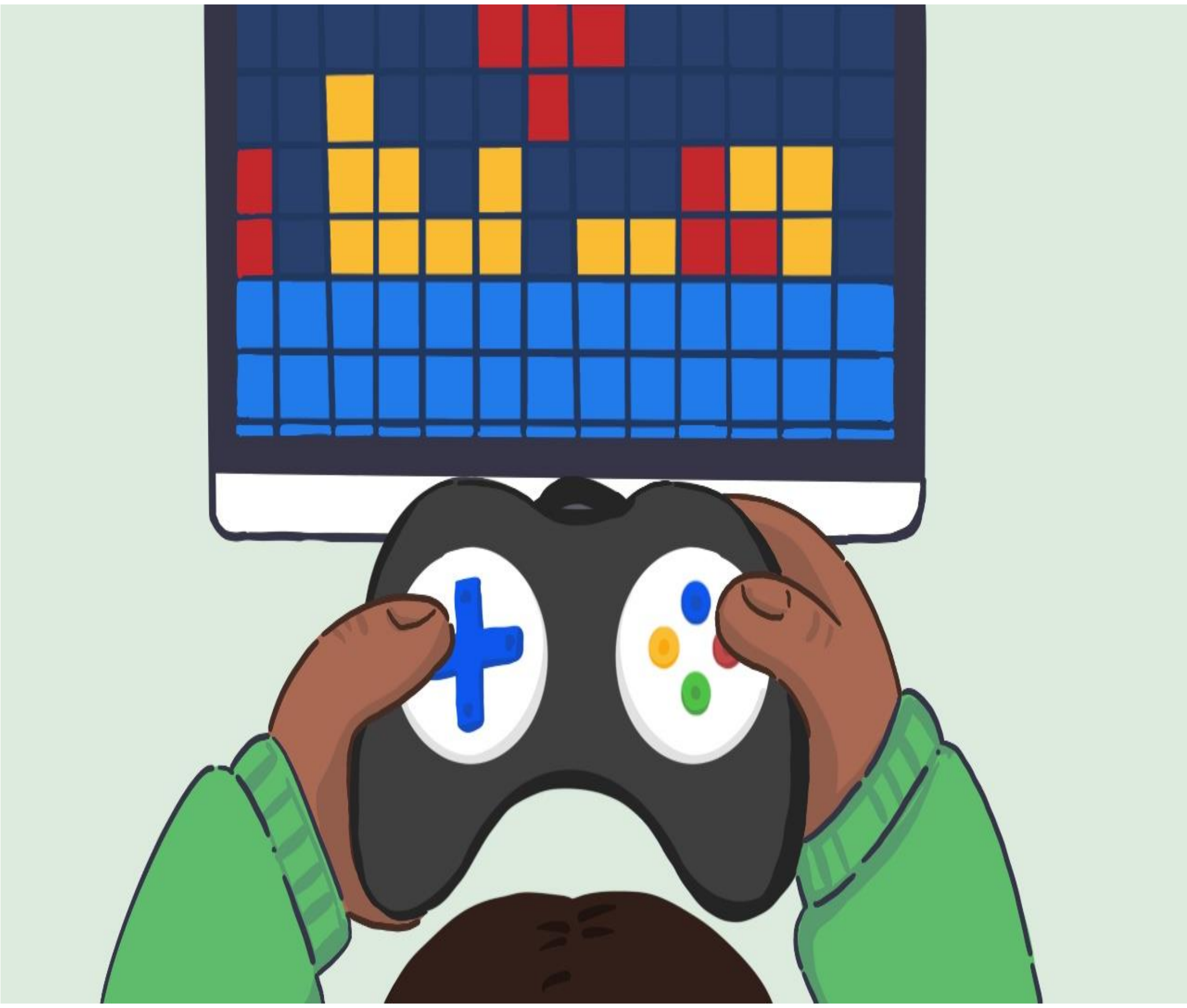
peacemaker19881. (n.d.). apibackend. GitHub. https://github.com/peacemaker19881/apibackend.git

Peacemaker19881. (n.d.). Student Registration System [GitHub repository]. GitHub. Retrieved March 14, 2022, from https://github.com/peacemaker19881/studentreg.git

Vue.js. (n.d.). *Vue CLI* [Computer software]. GitHub. Retrieved March 14, 2022, from https://github.com/vuejs/vue-cli

Vue.js. (n.d.). Vue.js forum*.* Retrieved March 14, 2022, from https://forum.vuejs.org/

Learning Outcome 3: Plan game



Indicative contents
3.1 Description of the Game **3.2 Creation of Narrative** **3.3 Game mechanics** **3.4 Identification of game controls** **3.5 Identification of Game Interface**

Key Competencies for Learning Outcome 3 : Plan game

Knowledge	**Skills**	**Attitudes**
• Definition of computer game • Description of game type • Differentiate narrative from storyline • Description of game mechanics • Description of game target devices • Description of game environment (scenery) • Identification of game controls • Identification of Game Interface	• Creating narrative	• Having Team work spirit • Being critical thinker • Being Innovative • Being attentive. • Being creative • Being Problem solver • Being Practical oriented

Duration: 25 hrs

Learning outcome 3 objectives:

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

1. Describe correctly game based on purpose
2. Describe clearly game mechanics based on game type
3. Identify correctly game controls based on game to be developed
4. Identify appropriately game interface based on purpose
5. Create correctly narrative based on game type

Resources

Equipment	**Tools**	**Materials**
• Computer	• Text Editor(VS Code) • Node js	• Internet

Indicative content 3.1: Description of the Game

Duration: 5 hrs

Theoretical Activity 3.1.1: Description of key concepts used in planning games

Tasks:

1. You are requested to answer the following questions related to the key concepts used in planning games:
   I. What do you understand about the term game?
   II. What are the different types of games?
   III. What is the distinction between narrative and storyline?
2. Provide the answer of asked questions by writing them on paper.
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 3.1.1. In addition, ask questions where necessary.

Key readings 3.1.1.: Description of key concepts used in planning games

Game

A computer game, also known as a video game, is a form of interactive entertainment that is played on a computer or a gaming console.

Game concepts

Narrative

A narrative refers to the story or storyline of a game. It encompasses the plot, characters, settings, and events that unfold throughout the game experience.

Storyline

A storyline refers to the sequence of events that occur within a game's narrative. It is the plotline or the structured progression of the game's story, including the key events, conflicts, and resolutions that the player experiences.

Game controller

A game controller refers to a custom component or code that handles user input and controls the behavior of a game built using the Vue.js framework. It is responsible for capturing user actions, updating the game state, and coordinating the game's mechanics and interactions.

Game Settings

Game settings typically refer to the configurable options or parameters that allow players to adjust various aspects of the game according to their preferences or requirements. These settings can include options related to graphics, audio, controls, difficulty levels, language, and other game-specific features

Game control

Game control refers to the management of game-related functionality, including handling user input, managing game state, and coordinating game mechanics. It involves creating components and implementing logic to control the behavior of a game built using the Vue.js framework.

Game HUD (heads-up display)

Game "HUD" refers to the Heads-Up Display, which is a user interface element that provides important information and feedback to the player during gameplay. The HUD is typically overlaid on the game screen and displays real-time data or indicators relevant to the game's progress, such as health, score, ammunition, timers, or other important statistics.

Game characters

Game characters refer to the interactive entities or avatars that players control or interact with within a game. These characters can include player-controlled characters, non-player characters (NPCs), enemies, allies, or any other entities that play a role in the game's mechanics and narrative.

Game environment

Game environment refers to the virtual space or world in which a game takes place. It encompasses the visual and interactive elements that make up the game's setting, including the terrain, objects, backgrounds, and overall atmosphere.

Game interface

Game environment refers to the virtual space or world in which a game takes place. It encompasses the visual and interactive elements that make up the game's setting, including the terrain, objects, backgrounds, and overall atmosphere.

Game consoles

Game consoles" typically refers to the developer tools or browser consoles that allow you to monitor and debug Vue.js applications while they are running. These consoles provide a range of features and functionalities to help you inspect the component hierarchy, track component state, analyse performance, and debug issues within your Vue.js application.

Game types

Puzzle Games: Vue.js can be used to create puzzle games like Sudoku, crossword puzzles, matching games, or sliding puzzles.

Quiz Games: Vue.js can power interactive quiz games where users can answer questions and receive instant feedback.

Word Games: Word-based games such as word search, hangman, or anagrams can be developed using Vue.js.

Memory Games: Vue.js is well-suited for creating memory games like card matching or pattern recognition.

Arcade Games: Vue.js can be used to create simple browser-based arcade games like platformers, endless runners, or retro-style games.

Theoretical Activity 3.1.2: Description of characteristics of game

Tasks:

1. You are requested to answer the following questions related to the characteristics of game:
    - I. What is the main objective of the game?
    - II. What devices are typically targeted for games and how are they utilized?
    - III. Explain the concepts of game dimension and game perspective
2. Provide the answer for the asked questions and write them on papers.
3. Present the findings/answers to the whole class
4. For more clarification, read the key readings 3.1.2. In addition, ask questions where necessary.

Key readings 3.1.2.: Description of characteristics of game

Game objective

In Vue.js, the term "game objective" refers to the specific goal or target that players need to achieve in a game.

Game target devices

In Vue.js, the term "game target devices" refers to the specific devices or platforms that you are aiming to support and run your Vue.js game on. These target devices can include various hardware and software configurations that players might use to access and play your game.

Examples of Target Devices

- Desktop Browsers
- Mobile Browsers
- Progressive Web Apps (PWAs)
- Web-Based Game Platforms

Game dimension

Game dimension can refer to two different concepts: screen dimensions and game world dimensions.

Screen Dimensions: This refers to the size of the game window or viewport on the player's screen. It determines the visible area where the game is displayed. The screen dimensions are typically measured in pixels and are important for ensuring that the game content fits properly within the available space.

Game World Dimensions: This refers to the virtual space or dimensions within the game where the gameplay and interactions take place. It represents the size, boundaries, and scale of the game world or level.

Game perspective

Game perspective refers to the visual representation or viewpoint from which the game world and its elements are displayed to the player. It determines how the game world is perceived in terms of depth, scale, and the relationship between objects within the game.

Application of learning 3.1.

N/A

Indicative content 3.2: Creation of Narrative

Duration: 5 hrs

Theoretical Activity 3.2.1: Description of storyline

Tasks:

1. You are requested to answer the following questions related to the Narrative:
   I. What do you understand about a storyline?
   II. What are the key components that make up a storyline?
2. Provide the answer for the asked questions and write them on papers.
3. Present the findings/answers to the whole class
4. For more clarification, read the key readings 3.2.1. In addition, ask questions where necessary.

Key readings 3.2.1.: Description of storyline

Storyline

The storyline of a game refers to the narrative or plot that unfolds as players progress through the game. It is essentially the story that the game developers have created to engage and immerse players in the gaming experience.

Components of a storyline

1. **Characters:** Well-developed characters with distinct personalities, motivations, and backstories. Protagonists, antagonists, and supporting characters play essential roles in driving the narrative.

2. **Setting:** The world or environment where the game takes place. This includes the physical locations, time period, and any unique or fantastical elements that define the game world.

3. **Plot:** The sequence of events that make up the central story. This encompasses the main quest, missions, and challenges that players undertake as they progress through the game.

4. **Conflict:** The central tension or problem that the characters must resolve. This conflict can take various forms, such as a villain's plan, an impending disaster, or internal struggles within the protagonist.

5. **Themes**: Underlying messages, ideas, or moral lessons that the game aims to convey. Themes add depth to the storyline and can resonate with players on a deeper level.

6. **Mood and Tone:** The emotional atmosphere or tone of the game's narrative. Whether it's dark and suspenseful, lighthearted and humorous, or somewhere in between, the mood contributes to the overall player experience.

7. **World-building:** The process of creating a detailed and immersive game world. This includes the lore, history, cultures, and rules that govern the game universe.

8. **Backstory:** Additional information about the characters or the game world that may not be directly presented in the main storyline. Backstory provides context and depth to the narrative.

9. **Dialogue:** The written or spoken interactions between characters. Dialogue is crucial for character development, conveying information, and establishing the tone of the narrative.

Practical Activity 3.2.2: Create a storyline

Task:

1. Read the key reading 3.2.2
2. As a Software developer, you are asked to go into the computer lab to:
   create a sample storyline of your choice.
3. Present your work to the trainer and your classmates

Key readings 3.2.2.: Create a storyline

Creating a compelling storyline involves a series of steps that help structure the narrative, develop characters, establish conflicts, and engage the audience.

Here are the essential steps to create a storyline:

1.Identify the Central Theme:

Determine the central theme or message you want to convey through your storyline. This theme will guide the development of your plot and characters.

2.Develop Characters:

Create well-rounded and engaging characters with unique personalities, motivations, strengths, and weaknesses. Consider their backgrounds, relationships, and character arcs.

3. Outline the Plot:

Outline the main events of your storyline, including the introduction, rising action, climax, falling action, and resolution. Structure the plot to build tension and keep the audience engaged.

4. Establish Setting:

Define the time and place in which your storyline takes place. Develop the setting to create a vivid and immersive world for your characters to inhabit.

5. Introduce Conflict:

Introduce conflicts and obstacles that challenge your characters and drive the narrative forward. Create tension and suspense to keep the audience invested in the story.

6. Create Plot Twists:

Incorporate unexpected plot twists and turns to surprise and intrigue the audience. These elements add depth and complexity to your storyline.

7. Write Dialogue:

Craft engaging dialogue that reveals character personalities, advances the plot, and conveys emotions. Develop a narrative voice that suits the tone and style of your storyline.

8. Build Pacing and Flow:

Manage the pacing of your storyline to maintain a balance between action, exposition, character development, and resolution. Ensure that the story flows smoothly from one scene to the next.

9. Revise and Refine:

Revise your storyline multiple times to refine and improve its elements. Seek feedback from peers, mentors, or beta readers to identify areas for enhancement.

10. Ensure Consistency:

Maintain consistency in the storyline, characters, world-building, and themes throughout your narrative. Ensure that all elements work together cohesively.

11. Add Emotional Depth:

Infuse your storyline with emotional depth by exploring characters' feelings, motivations, and relationships. Emotions can resonate with the audience and enhance the impact of your narrative.

12. Create a Memorable Ending:

Craft a satisfying and memorable ending that resolves conflicts, provides closure, and leaves a lasting impression on the audience.

Example of a game storyline

We can consider a game called The quest for the Lost Amulet as shown below:

Title: "The Quest for the Lost Amulet"

Introduction:

You are an adventurer in a medieval fantasy world known for its magical artifacts. Your latest quest is to find the Lost Amulet, a powerful artifact said to grant its bearer immense power. The amulet is rumored to be hidden deep within the Enchanted Forest. Your journey begins at the entrance to the forest.

Gameplay:

1. **Scene 1: Entrance to the Enchanted Forest**

❖You start at the entrance to the Enchanted Forest.

❖You have a choice to "Enter the forest" or "Turn back."

❖Depending on your choice, you'll encounter different challenges or opportunities.

2. **Scene 2: The Talking Tree**

As you venture deeper into the forest, you come across a talking tree.

The tree tells you a riddle. You must solve the riddle to proceed.

If you answer correctly, the tree offers you a magical compass that helps you navigate the forest.

If you answer incorrectly, you are lost in the forest, and the game ends.

3. **Scene 3: The Hidden Meadow**

Using the magical compass, you find a hidden meadow with a beautiful waterfall.

You can choose to "Rest by the waterfall" or "Continue exploring."

Resting restores your energy but may waste time. Continuing may lead to more discoveries.

4. **Scene 4: The Goblin Bridge**

You come across a rickety bridge guarded by mischievous goblins.

You have to make a decision, such as "Bribe the goblins," "Fight the goblins," or "Sneak past them."

Your choice affects your character's health and the goblins' reactions.

5. **Scene 5: The Cave of Trials**

You find an ancient cave filled with traps and puzzles.

Solve the challenges in the cave to access the amulet's chamber.

Puzzles may include logic puzzles, mazes, and riddles.

6. **Scene 6: The Lost Amulet**

You finally reach the amulet's chamber.

You must face a guardian or a final challenge to obtain the amulet.

The outcome depends on your previous choices and your character's attributes.

Conclusion:

If you successfully obtain the Lost Amulet, you win the game.

If you fail in any part of the journey or make the wrong choices, you lose the game.

You can choose to play again and make different choices, exploring alternative paths in the forest.

Practical Activity 3.2.3: Prepare different game sounds and background music

Task:

1. Read key reading 3.2.3 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to Download and prepare different Game sounds and background Music. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to Download and prepare different Game sounds and background Music.

5. Referring to the steps provided on task, Download and prepare different Game sounds and background Music.

6. Present your work to the trainer and whole class

Key readings 3.2.3.: Prepare different Game sounds and background Music

To download and prepare different game sounds effectively, you can follow a series of steps that involve sourcing, organizing, editing, and integrating audio assets for game development. Here are the steps to download and prepare different game sounds:

1. **Identify Sound Requirements:**

Determine the specific sound requirements for your game, including background music, sound effects, ambient sounds, character voices, and other audio elements needed to enhance the gameplay experience.

2. **Search for Sound Resources:**

Explore online platforms and libraries that offer a wide range of game sounds, such as Freesound.org, SoundBible, ZapSplat, and GameDev Market. Look for royalty-free or licensed sound packs that align with your game's theme and style.

3. **Download Sound Assets:**

Select and download the desired game sounds that match your game's atmosphere, setting, and gameplay events. Ensure that you have the appropriate licenses or permissions to use the sound assets in your game project.

4. **Organize Sound Files:**

Create a structured folder system to organize and categorize the downloaded sound files based on their type, purpose, and relevance to different game elements. This organization will make it easier to access and manage the audio assets during game development.

5. **Edit and Customize Sounds:**

Use audio editing software such as Audacity, Adobe Audition, or GarageBand to edit and customize the downloaded sound files. Trim, loop, fade, adjust volume levels, apply effects, or combine multiple sounds to create unique audio assets for your game.

6. **Optimize Sound Quality:**

Ensure that the sound files are optimized for game development by converting them to the appropriate file format (e.g., WAV, MP3, OGG) and adjusting the bitrate and sample rate for optimal performance and quality in the game engine

7. **Create Sound Variations:**

Generate variations of sound effects by modifying pitch, speed, intensity, or adding filters to create diverse audio outcomes for different in-game actions, environments, or events. This adds depth and realism to the game audio experience.

8. **Test and Iterate:**

Integrate the prepared game sounds into your game project and test them in different gameplay scenarios to evaluate their effectiveness and coherence with the game visuals and mechanics. Iterate on the sound design based on feedback and testing results.

A trainee shall be introduced to different online game sound and effects websites. He will be tasked to download and play those sounds and music from platforms like: GameSounds.xyz, Kenny.nl, 99 Sounds, Sound Image, Open Game Art, The Motion Monkey, Zapsplat, SoundBible, Freesound.org.

Theoretical Activity 3.2.4: Identification of game environment components

Tasks:

1. You are requested to answer the following questions related to the components of game environment:
   I. What do you understand about Game environments?
   II. What are the key aspects that make up game environments?
   III. Explain the different game components, including game levels, reward levels, and missions.
2. Provide the answer of asked questions by writing them on paper.
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 3.2.4. In addition, ask questions where necessary.

Key readings 3.2.4.: Identification of components of game environment

Game environments, also known as scenery or game worlds, refer to the virtual spaces in which games take place. They encompass the landscapes, structures, objects, and overall visual and interactive elements that create the setting for gameplay.

Key aspects of game environments

1. Landscapes and Terrain

2. Architecture and Structures

3. Natural Elements

4. Props and Objects

5. Lighting and Atmosphere

6. Level Design and Layout

7. Interactive Elements

Game Level: A game level refers to a distinct segment or stage within a game that presents a specific set of challenges, objectives, and gameplay mechanics.

Reward Level: A reward level, also known as a bonus level or a special level, is a specific level within a game that offers additional rewards or unique gameplay experiences to players.

Mission (Main and Side): In the context of games, missions refer to specific tasks or objectives that players are assigned to complete within the game's narrative or gameplay structure. Missions are an integral part of games, particularly in story-driven or mission-based genres like role-playing games (RPGs) or action-adventure games.

Main Mission: The main mission, also known as the main quest or main storyline, represents the core narrative and primary objectives of the game.

Side Mission: Side missions, also referred to as side quests or optional quests, are additional tasks or objectives that players can undertake alongside the main mission.

Points to Remember

- There are the differences between narrative and storyline where the narrative provides the overarching framework and thematic depth of a story while the storyline focuses on the specific events and plot progression that drive the narrative forward.
- There are various types of games that cater to different interests and preferences. Here are some common types of games: Puzzle Games, Quiz Games, Word Games, Memory Games, Arcade Games.
- Game objectives are the specific goals or targets that players aim to achieve within a game. These objectives provide structure, motivation, and a sense of progression for players as they navigate through the gameplay experience and Game target devices refer to the platforms or devices on which games are designed to be played.
- A storyline is composed of various components that work together to create a cohesive and engaging narrative structure. These components help shape the plot, characters, conflicts, and themes of the story. There are the key components that make up a storyline like: Characters, Setting, Plot, Conflict.
- Creating a compelling storyline involves a series of steps that help structure the narrative, develop characters, establish conflicts, and engage the audience.

  Here are the essential steps to create a storyline:

  1. Identify the Central Theme
  2. Develop Characters
  3. Outline the Plot
  4. Establish Setting
  7. Write Dialogue
- To download and prepare different game sounds effectively, you can follow a series of steps that involve sourcing, organizing, editing, and integrating audio assets for

game development. Here are the steps to download and prepare different game sounds:

1. Identify Sound Requirements
2. Search for Sound Resources
3. Download Sound Assets
4. Organize Sound Files
6. Optimize Sound Quality
7. Create Sound Variations

- Game environments are crucial elements that contribute to the overall gaming experience, setting the tone, atmosphere, and immersion for players. There are key aspects that make up game environments: Landscapes and Terrain, Architecture and Structures, Natural Elements, Props and Objects.

Application of learning 3.2.

The game development team has a plan to develop a game that will enable the person moving on a plain road to Jump an obstacle every time he meets it on the way. The development team will need game effects like background music and other sound effects. As a VueJs expert, you have been tasked to download and arrange the sound effects and background music.

Create a storyline that will be used for development of the cited game.

Indicative content 3.3: Game Mechanics

Duration: 5 hrs

Theoretical Activity 3.3.1: Description of key elements for defining game mechanics

Tasks:

1. You are requested to answer the following questions related to the Key element for define game mechanics:

   I. What is the game HUD (Heads-Up Display)?

   II. What are the essential elements that define a game?

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 3.3.1. In addition, ask questions where necessary.

Key readings 3.3.1.: Description of Key elements for defining game mechanics

Game HUD (heads-up display)

Game "HUD" refers to the Heads-Up Display, which is a user interface element that provides important information and feedback to the player during gameplay.

Game steps

Assume we consider a Jump game which will involve a character or object jumping from one platform to another while facing various challenges and obstacles.

Throughout this Jump Game, we shall maintain a sense of progression, reward players for their skill and exploration, and keep the gameplay challenging but fair. Additionally, we shall use music and visuals to enhance the overall experience and make the Jump Game engaging and enjoyable for players.

1. **Introduction:**

Introduce the protagonist or player character who will be doing the jumping.

Establish the setting, whether it's a fantastical world, a cityscape, a jungle, or any other environment.

Present the motivation or goal for the character to engage in the Jump Game, such as rescuing someone, collecting valuable items, or reaching a specific destination.

2. **Tutorial or Warm-up:**

Provide a brief tutorial level or warm-up phase to teach players the basic controls and mechanics of the Jump Game. This may include how to jump, double jump, or perform special moves.

3. **Obstacle and Challenge Introduction:**

Introduce the various obstacles and challenges the character will face, such as gaps, moving platforms, spikes, enemies, and other hazards.

Present the character with the first set of relatively easy obstacles to build their confidence and understanding of the game's mechanics.

4. **Increasing Difficulty:**

As the character progresses, gradually increase the difficulty of the challenges and obstacles. This could include larger gaps, faster-moving platforms, and more complex enemy encounters.

Introduce power-ups or special abilities that the character can use to overcome obstacles.

5. **Story Progression:**

Advance the storyline as the character moves through the game, revealing more about the character's motivation and the world they're in.

Incorporate cut scenes or dialogue sequences to provide context and character development.

6. **Boss Battles or Mini-Bosses:**

Include occasional boss battles or mini-boss encounters that require unique strategies and platforming skills to defeat.

7. **New Environments:**

Change up the environments periodically to keep the gameplay fresh and visually interesting. This might involve transitioning from a forest to a snowy mountain or from a cave to a futuristic city.

8. **Puzzles and Secrets:**

Include hidden areas, collectibles, and puzzles that add an extra layer of gameplay and encourage exploration.

9. **Climax:**

Build up to a climactic moment in the story, which may involve a major plot twist or a final showdown with the main antagonist.

10. **Resolution:**

Resolve the character's primary goal and conclude the storyline. This could involve rescuing the person they set out to save, obtaining the valuable item, or reaching the destination.

11. **Endgame or Post-Game Content:**

Optionally, include additional challenges or content for players who want to continue playing after the main storyline is complete.

12. **Conclusion:**

Provide a satisfying conclusion to the story, allowing players to reflect on the character's journey and accomplishments.

The key elements that make up a game include:

1. **Scores:** Scores represent the player's performance and progress in the game. They can be based on various factors, such as completing objectives, defeating enemies, or solving puzzles. Scores provide a sense of achievement and can drive competition among players.

2. **Level:** A game typically consists of multiple levels or stages that increase in difficulty as the player progresses.

3. **Speed:** Speed is often a critical element in many games, especially action or racing games. It can refer to the speed of the player character, enemies, or other game elements.

4. **Time:** Time is a common gameplay element that adds urgency and challenges the player's ability to make decisions quickly.

5. **Target Device:** The target device refers to the platform or device on which the game will be played, such as a computer, console, or mobile device.

Determining game mechanics involves defining the rules, interactions, and systems that govern gameplay in a game.

Here are some steps to help you determine game mechanics:

1. **Identify the Core Gameplay Objective:** Start by identifying the primary objective or goal of your game. This could be completing levels, solving puzzles,

defeating enemies, achieving high scores, or any other desired outcome. The core objective will guide the design of your game mechanics.

2. **Break Down Gameplay into Actions:** Analyse the actions that players will take within the game to achieve the objective. For example, if it's a platformer game, actions may include jumping, running, and collecting items. If it's a strategy game, actions may involve resource management, building structures, and deploying units.

3. **Define Player Abilities and Constraints:** Determine the abilities and constraints of the player character or characters. This includes their movement capabilities, special powers or skills, limitations, and any resource management aspects. Consider how these abilities and constraints affect gameplay and create meaningful choices for players.

4. **Establish Rules and Interactions:** Define the rules that govern how different game elements interact with each other and with the player. This includes rules for collision detection, physics, combat, puzzle-solving, and any other relevant mechanics. Determine how different actions or inputs trigger specific outcomes or responses.

5. **Introduce Challenges and Obstacles**: Identify the challenges and obstacles that players will encounter throughout the game. These can be environmental hazards, enemies, puzzles, time constraints, or any other element that adds difficulty and requires the player to strategize or problem-solve.

6. **Determine Progression and Rewards:** Decide how players will progress and be rewarded in the game. This can involve unlocking new levels, gaining access to new abilities or items, earning points or achievements, or experiencing story developments. Consider how these progression and reward systems motivate players and provide a sense of accomplishment.

7. **Balance and Iteration:** Continuously balance and iterate on your game mechanics based on playtesting and player feedback. Adjust the difficulty level, fine-tune the rules, and ensure that the mechanics provide a challenging yet enjoyable experience.

Points to Remember

- Several key elements make up a game, contributing to its structure, mechanics, and overall player experience. There are essential elements that define a game like: Scores, Level, Speed, Time, Target Device.
- The Game HUD (Heads-Up Display) is a crucial element in video games that provides players with essential information, feedback, and interactive elements during gameplay.

Indicative content 3.4: Identification of Game Controls

Duration: 5 hrs

Theoretical Activity 3.4.1: Identification of game controls

Tasks:

1. You are requested to answer the following questions related to the drawing materials:
   I. What are the input keys used in the game?
   II. Provide an explanation of the term 'primary control' in gaming.
   III. Explain secondary control in the context of gaming.
   IV. Provide an explanation of support controls in the context of gaming.
2. Provide the answer to asked questions by writing them on paper.
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 3.4.1. In addition, ask questions where necessary.

Key readings 3.4.1.: Identification of game controls

Inputs/keys

These refer to user input through keyboard and mouse events, which are essential for controlling the game

Types of Keyboard inputs in VueJS Game

Keyboard Input: You can listen for keyboard events to control game characters or trigger actions. Common keyboard events include keydown, keyup, and keypress.

Mouse Input: You can capture mouse events like clicks, mouse movement, and mouse wheel scrolling to interact with the game environment or UI elements.

Touch Input (for mobile devices): For mobile games, you can capture touch events like touchstart, touchmove, and touchend to control game elements and respond to user actions on touchscreen devices.

Game Controller Input: If your game is designed for game consoles or compatible with game controllers, you can listen for gamepad events to control characters and gameplay.

Hand accessibility

This refers to features in a Vue.js game or web application that are designed to improve accessibility for users with limited hand mobility or dexterity.

Primary Controls:

Primary controls refer to the core input mechanisms that are essential for playing the game. These controls are fundamental to the gameplay and often include actions like movement, jumping, shooting, or any primary actions that the player needs to perform. In a Vue.js game, you might implement primary controls using event listeners, key bindings, or other input handling methods to capture and respond to the player's actions.

Secondary Controls:

Secondary controls are additional input mechanisms that are not as essential as the primary controls but still serve a purpose in the game. These controls might include things like interacting with in-game menus, toggling various game settings, or performing secondary actions. Secondary controls are often used for features that enhance the player's experience or provide access to non-core gameplay elements.

Support Controls:

Support controls are usually related to user interface elements and navigation within the game. They help players navigate menus, manage their inventory, access options, or interact with non-gameplay-related features. In Vue.js, support controls can involve user interface components, buttons, or other UI elements that allow players to interact with the game's menus and settings.

Types of Game Controllers

1. **Keyboard Controls:**

Keyboard controls are the simplest way to interact with a Vue.js game. You can listen for key events and use them to move characters or trigger actions within your game.

2. **Gamepad Controller:**

You can integrate gamepad support using the Gamepad API to allow players to use game controllers like Xbox or PlayStation controllers to play your game. The API provides information about button presses and analog stick positions.

3. **Touch Controls:**

For mobile games, you can implement touch controls using Vue.js. This involves detecting touch events such as tapping, swiping, or dragging to control game elements.

4. **Accelerometer and Gyroscope Controls:**

Some mobile games make use of device sensors like accelerometers and gyroscopes to control game elements. You can access these sensors through the Device Orientation and Motion API.

5. **Custom Game Controllers:**

Depending on your game's complexity, you may create custom on-screen controls or virtual controllers using HTML, CSS, and Vue.js components for specific gameplay requirements.

Points to Remember

- In a VueJS game, keyboard inputs play a crucial role in enabling player interaction and controlling gameplay elements. There are the types of keyboard inputs commonly used in VueJS games like: Keyboard Input, Mouse Input, Touch Input (for mobile devices) Game Controller Input.

Indicative content 3.5: Identification of Game Interface

Duration: 5 hrs

Theoretical Activity 3.5.1: Description of game interface components

Tasks:

1. You are requested to answer the following questions related to the Game interface components:

   I. Explain splash screen
   II. Describe characters in the game.
   III. What do you understand about the following terms used in Vue.js game?
      a) Game playable characters and non-playable characters
      b) Characters' relationship
      c) Characters Interactivity
      d) Game Dimensions
      e) Game perspective
      f) Playing Zone / Game Boundaries
      g) Scenes of different levels
      h) Design tools for environment
   iv. What are the key elements of good characters and game environment components?
   v. Describe the different examples of alert messages, including success, failure, information, and warning.
   vi. Explain game play

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 3.5.1 In addition, ask questions where necessary.

Key readings 3.5.1.: Description of game interface components

Identification of Game Interface

Splash screen

A splash screen in a Vue.js game is an introductory screen that appears when the game is launched, typically displaying the game's logo, title, or any other relevant information. It serves as a loading screen or an initial visual representation before the game's content is fully loaded and ready to be played

Game characters

In a Vue.js game, game characters are interactive entities or objects that players can control or interact with. They are typically represented visually and have their own attributes, behaviors, and roles within the game world.

Examples

Player Character: The main character controlled by the player. This can be a protagonist, a hero, or any other character that the player embodies throughout the game. The player character can have attributes such as health, stamina, and special abilities. You can create a Vue.js component for the player character that handles movement, animations, and interactions.

Non-Player Characters (NPCs): NPCs are characters controlled by the game's artificial intelligence (AI). They can be allies, enemies, or neutral characters that the player interacts with. NPCs can have their own behaviors, dialogues, and quests. Each NPC can be represented by a Vue.js component that handles their AI, animations, and interactions with the player.

Enemies: Enemies are characters that the player must defeat or overcome to progress in the game. They can have different attack patterns, health levels, and abilities. Create Vue.js components for each enemy type, allowing them to move, attack, and react to the player's actions.

Boss Characters: Boss characters are powerful adversaries that provide significant challenges for the player. They often have multiple stages, unique abilities, and require specific strategies to defeat. Implement boss characters as specialized Vue.js components that handle their complex behaviors, transformations, and unique attacks.

NPCs for Quests and Dialogue: In addition to combat-focused characters, you can create NPCs that provide quests, offer information, or engage in dialogue with the player. These characters can be represented by Vue.js components that display dialogues, choices, and manage quest-related interactions.

Supporting Characters: Supporting characters can include allies, mentors, or companions that aid the player throughout the game. They can have their own storylines, abilities, and interactions. Develop Vue.js components for these characters to handle their unique features and interactions with the player.

Characters’ relationship

A game character relationship refers to the connections, interactions, and dynamics between different characters within a computer game. These relationships can be an essential part of the game's story, gameplay mechanics, and overall experience. Game character relationships can take various forms, and they play a crucial role in shaping the narrative and player experience.

Characters Interactivity

Character interactivity in a Vue.js game refers to the ability for players to interact with and control the actions of game characters. It involves creating mechanisms and implementing features that allow players to engage with characters in meaningful ways.

Elements of good characters

Visual Appeal: Create visually appealing character designs that are unique, well-detailed, and visually consistent with the overall art style of the game. Use Vue.js components to implement character visuals and animations, ensuring they are smooth and responsive.

Personality and Backstory: Develop a compelling personality and backstory for each character. Give them distinct traits, motivations, and goals that drive their actions and interactions in the game. Use Vue.js data properties and methods to manage and showcase character traits and behaviors.

Character Arc and Growth: Provide characters with opportunities for growth and development throughout the game. Allow them to evolve, learn, and overcome challenges, giving players a sense of progression and investment. Use Vue.js data properties and methods to track character progress and trigger relevant events.

Unique Abilities or Skills: Assign unique abilities, skills, or powers to characters, making them stand out and contribute to gameplay mechanics. Use Vue.js methods to handle character-specific abilities and implement their effects within the game.

Voice Acting and Dialogue: If applicable, consider adding voice acting or well-written dialogue for characters. This can enhance their personality and make them more relatable to players. Use Vue.js components and data properties to handle dialogue interactions and implement dynamic conversations.

Character Relationships: Establish meaningful relationships between characters, whether it's friendships, rivalries, romances, or alliances. Use Vue.js data properties and methods to manage character relationships and reflect their dynamics throughout the game.

Memorable Interactions: Create memorable interactions between characters and the player. Implement engaging dialogue choices, quests, or events that allow players to form connections with the characters and impact their stories. Use Vue.js methods and event handling to manage these interactions and their consequences.

Emotional Engagement: Design characters that evoke emotions and resonate with players. This can be achieved through compelling storytelling, relatable struggles, or emotional moments within the game. Use Vue.js components and animations to enhance emotional impact during key character moments.

Define Game Dimensions

Game dimensions typically refer to the size and layout of a game that you are developing using Vue.js or integrating into a Vue.js application.

How to Define a Game Dimension?

Determine the Canvas Size: Decide on the desired width and height of the game canvas. This will be the area where the game graphics, UI, and other elements will be rendered. You can set these dimensions as variables or constants in your Vue.js code.

Create the Game Component: Set up a Vue.js component that represents the game. Within the component, define the canvas element using the determined width and height. Use HTML and Vue.js syntax to create the canvas element with the specified dimensions.

Responsive Design (Optional): If you want your game to be responsive and adapt to different screen sizes, consider implementing responsive design techniques. You can use CSS media queries or Vue.js directives like v-bind to dynamically adjust the canvas dimensions based on the device or screen size.

Coordinate System: Determine the coordinate system you will use for positioning game elements within the canvas. For example, you might use a Cartesian coordinate system with (0,0) at the top-left corner of the canvas. Define the coordinate system logic and any necessary conversions in your Vue.js code.

Scaling and Viewport: If your game involves scaling or zooming, implement the necessary logic to handle scaling the game elements while maintaining the aspect ratio. Use Vue.js methods and data properties to manage the scaling and viewport calculations.

Element Positioning: When placing game elements within the canvas, use the determined dimensions and coordinate system to position them correctly. You can use CSS positioning properties or Vue.js directives like v-bind to dynamically position and style the game elements based on the defined dimensions.

Collision Detection (Optional): If your game requires collision detection, consider the game dimensions when implementing collision detection algorithms. Make sure to account for the dimensions of game elements and their positions within the canvas to accurately detect collisions.

Define Game perspective

Game Perspective relates more to the position of the player in regard to the playable character, but considering who the playable character is, and who the player is playing as, is also worth defining and considering.

How to define a game perspective

Choose the Perspective Type: Decide on the type of perspective you want for your game. Common perspectives include 2D top-down, side-scrolling, isometric, or 3D. Each perspective has its own visual style and considerations.

Create the Game Component: Set up a Vue.js component that represents the game. This component will contain the necessary logic and rendering for the chosen perspective. Define the structure of the component and any additional child components that will handle the rendering of the game world.

Define the Camera: Implement a camera system that determines the viewpoint and position from which the game world is rendered. For 2D perspectives, this could involve setting the camera's x and y coordinates. For 3D perspectives, it may involve specifying the camera's position, target, and field of view.

Render the Game World: Use Vue.js directives and templates to render the game world based on the defined perspective. This can include rendering 2D sprites, tiles, or 3D models depending on the chosen perspective. Apply appropriate CSS styles, transformations, or WebGL rendering techniques for optimal visual presentation.

Handle Player Input: Implement player input handling based on the chosen perspective. This can involve capturing keyboard, mouse, or touch events and translating them into game actions. Use Vue.js event handling methods and directives to capture and respond to player input.

Coordinate System: Determine the coordinate system that aligns with the chosen perspective. For example, in a top-down 2D perspective, you might use a Cartesian coordinate system where (0,0) is the top-left corner. In a 3D perspective, you may use a 3D coordinate system with x, y, and z axes. Define the coordinate system logic and any necessary conversions in your Vue.js code.

Camera Movement and Controls: Enable camera movement and controls if applicable to the chosen perspective. This can involve panning, zooming, or rotating the camera based on player input. Implement Vue.js methods and data properties to update the camera position and orientation accordingly.

Visual Effects and Rendering Optimization: Consider adding visual effects and optimizations specific to the chosen perspective. This can include depth sorting, parallax scrolling, lighting effects, or shader effects. Use Vue.js components, directives, and methods to handle these visual effects and rendering optimizations.

Define Playing Zone / Game Boundaries

The playing zone in a Vue.js game refers to the area or boundaries within which the gameplay takes place. It defines the spatial limits where game elements, characters, and interactions occur. Defining the playing zone is essential for creating a controlled and immersive gameplay experience.

Define Scenes of different levels

In a Vue.js game, a scene refers to a distinct level or area within the game where specific gameplay and challenges occur. Each scene typically has its own set of game elements, objectives, and visual presentation. Defining scenes for different levels in a Vue.js game involves organizing and managing the various components and data related to each level

Define design tools for environment

Graphic Editors: Graphic editing software such as Adobe Photoshop, GIMP, or Pixlr can be used to create or modify game sprites, textures, backgrounds, and other visual elements. These tools provide powerful features for creating pixel art, image editing, and applying effects to enhance the visual appeal of your game.

3D Modeling Software: If you're developing a 3D game environment, 3D modeling software like Blender, Autodesk Maya, or Unity's built-in editor can be used to create and animate 3D models, characters, props, and environments. These tools offer a wide range of tools and features to create detailed 3D assets.

Level Editors: Level editors provide a visual interface for designing and constructing game levels. Tools like Tiled, Ogmo Editor, or Unity's Tilemap Editor can be used to create 2D tile-based maps, place game objects, define collision areas, and manage game assets within the levels. These editors often support exporting level data in formats compatible with Vue.js game engines or frameworks.

Vue.js UI Libraries: Vue.js UI libraries like Vuetify, Element UI, or Bootstrap-Vue offer pre-designed UI components and themes that can be used to create user interfaces for your game. These libraries provide a wide range of customizable UI elements such as buttons, menus, dialogs, and forms, allowing you to design visually appealing and responsive interfaces for your game.

Particle Editors: Particle editors like the Phaser Particle Editor or Unity's Particle System Editor enable the creation of particle effects, such as explosions, fire, smoke, and magic effects. These tools allow you to visually design particle systems and customize their properties, such as size, color, movement, and behavior, enhancing the visual effects in your Vue.js game.

Sound Editing Software: Sound editing software like Audacity or Adobe Audition can be used to create or edit audio assets for your game. These tools allow you to record, edit, and manipulate sound effects, music, and voiceovers. You can use them to enhance the audio experience and create immersive soundscapes for your Vue.js game.

Integrated Development Environments (IDEs): IDEs like Visual Studio Code, WebStorm, or Atom provide powerful code editing features and extensions for Vue.js development. These tools offer syntax highlighting, code completion, debugging, and other productivity features, making it easier to develop and maintain your Vue.js game project.

Alert Messages

Alert messages are types of feedback or notifications provided by computer programs, systems, or applications to communicate the status or outcomes of various actions or processes to users. These messages serve to inform users about the state of the system, any issues that may have occurred, or general information about the operation.

Examples of Alert Messages

1. **Success Messages:**

Purpose: Success messages indicate that a task or operation has been completed successfully without any issues or errors. They inform the user that the action they took has achieved the desired result.

Example: "Your file has been successfully saved," "Payment processed successfully."

2. **Failure Messages:**

Purpose: Failure messages communicate that an action or operation has encountered an issue or error and could not be completed as expected. They inform the user about the nature of the problem and sometimes offer suggestions on how to resolve it.

Example: "Invalid username or password, please try again," "The server is currently unreachable."

3. **Information Messages:**

Purpose: Information messages provide general information or updates about the system's status or operations. They are used to convey non-critical, informative details to the user. Information messages are often used for status updates or explanations.

Example: "You have 5 new emails," "An update is available, please restart your computer to install it."

4. **Warning Messages:**

Purpose: Warning messages indicate potential issues or situations that may not be errors but still require the user's attention. They are used to caution users about actions they are about to take or potential problems they may encounter.

Example: "Are you sure you want to delete this file? This action cannot be undone," "Low battery warning, please connect your device to a charger."

Game Play Guide

A gameplay guide in Vue.js refers to a set of instructions, tips, or tutorials that help players understand and navigate the gameplay mechanics and features of a Vue.js game. It provides guidance and information to players to enhance their understanding of the game and improve their gameplay experience.

Points to Remember

- Creating compelling and engaging characters is essential in game development to immerse players in the game world and drive the narrative forward. There are key elements that contribute to making good characters in games: Visual Appeal, Personality and Backstory, Character Arc and Growth, Voice Acting and Dialogue
- Alert messages are used in various applications to communicate important information, warnings, notifications, or feedback to users.
- There are different examples of alert messages commonly found in digital interfaces: Success Messages, Failure Messages, Information Messages, Warning Messages.

Application of learning 3.5.

N/A

Learning outcome 3 end assessment

Written assessment

1. What contributes to making a good character in games? Select one answer

    A. Visual Appeal, Personality and Backstory, Character Arc and Growth, Voice Acting and Dialogue

    B. Visual Appeal, Sound Effects, Level Design

    C. Gameplay Mechanics, Marketing Strategy, Music Composition

    D. Difficulty Levels, Achievements, User Interface

2. Personality and Backstory are important elements in creating compelling characters for game narratives. (True/False)

3. Voice acting is not necessary for engaging characters in video games. (True/False)

4. What are examples of alert messages commonly found in digital interfaces? Select one

    A. Victory Messages, Defeat Messages, Neutral Messages

    B. Success Messages, Failure Messages, Information Messages, Warning Messages

    C. Start Messages, Stop Messages, Continue Messages

    D. Loading Messages, Waiting Messages, Ready Messages

5. Alert messages are used in applications solely for decorative purposes. (True/False)

6. Warning messages are used to notify users of potential issues. (True/False)

7. Which types of keyboard inputs are commonly used in Vue.js games? Select one option

    A. Keyboard Input, Gesture Input, Voice Input

    B. Keyboard Input, Mouse Input, Touch Input (for mobile devices), Game Controller Input

    C. Keyboard Input, Joystick Input, Camera Input

    D. Keyboard Input, Scroll Input, Screen Tap Input

8. Mouse Input is typically used in Vue.js games to control gameplay elements. (True/False)

9. Game Controller Input is only used for console games and not for web-based games developed with Vue.js. (True/False)

10. What are the essential elements that define a game?

    A. Scores, Level, Speed, Time, Target Device

    B. Characters, Dialogue, Music, Graphics

    C. Networking, Servers, Code Structure

    D. None of the above

11. Speed is not an essential element in defining a game. (True/False)

12. The target device can impact the design and mechanics of a game. (True/False)

13. What is the purpose of the Game HUD in video games?

    A. To display essential information, feedback, and interactive elements during gameplay

    B. To enhance the background music

    C. To provide real-time weather updates

    D. To manage network connections

14. The Game HUD is only used in first-person shooter games. (True/False)

15. HUD elements include health bars, score displays, and mini-maps. (True/False)

16. What are key aspects that make up game environments?

    A. Landscapes and Terrain, Architecture and Structures, Natural Elements, Props and Objects

    B. Character Dialogues, Soundtracks, Game Mechanics

    C. Server Performance, Code Efficiency, AI Behavior

    D. None of the above

17. Game environments do not affect the player's immersion. (True/False)

18. Natural elements like trees and rivers contribute to the game's atmosphere. (True/False)

19. What are the steps to download and prepare different game sounds?

    A. Identify Sound Requirements, Search for Sound Resources, Download Sound Assets, Organize Sound Files, Optimize Sound Quality, Create Sound Variations

    B. Search for Sound Resources, Download Sound Assets, Identify Sound Requirements, Create Sound Variations

    C. Download Sound Assets, Optimize Sound Quality, Identify Sound Requirements

    D. None of the above

20. Organizing sound files is an unnecessary step in preparing game sounds. (True/False)

21. Creating sound variations can enhance the audio experience in a game. (True/False)

22. What are the essential steps to create a storyline?
    A. Identify the Central Theme, Develop Characters, Outline the Plot, Establish Setting, Write Dialogue
    B. Develop Characters, Establish Setting, Write Dialogue
    C. Outline the Plot, Write Dialogue, Develop Characters
    D. None of the above
23. Establishing a setting is not important when creating a storyline. (True/False)
24. Writing dialogue helps in character development and advancing the plot. (True/False)
25. What role does the central theme play in a storyline?
    A. It serves as the foundation and guiding concept for the narrative
    B. It determines the game's graphics
    C. It only affects the background music
    D. None of the above
26. A well-developed character arc is crucial for player engagement. (True/False)
27. Game controller input is less responsive than keyboard input in Vue.js games. (True/False)
28. Information messages in applications provide users with general updates or instructions. (True/False)
29. Props and objects in game environments have no impact on gameplay mechanics. (True/False)
30. Speed adjustments in games can influence the difficulty level and player experience. (True/False)

Practical assessment

The game development team. has a plan to develop a game that will enable the person moving on a plain road to Jump an obstacle every time he meets it on the way. The development team will need game effects like background music and other sound effects. As a VueJs expert you have been tasked to download and arrange the sound effects and background music.

Create a storyline that will be used for development of the cited game.

References

Schell, J. *(2008).* The art of game design: A book of lenses. CRC Press.

Salen, K., & Zimmerman, E. (2003). Rules of Play: Game Design Fundamentals. MIT Press.

Adams, E. W., & Dormans, J. (2012). Game Mechanics: Advanced Game Design. New Riders.

Serpa, A. (2023). The Cores of Game Design: Mechanics, Economics, Narrative, and Aesthetics. Routledge.

MasterClass. (n.d.). How to design a video game character. MasterClass. Retrieved March 14, 2022, from https://www.masterclass.com/articles/how-to-design-a-video-game-character#what-makes-a-good-video-game-character

Webdeasy. (n.d.). Multiplayer game programming with Vue.js. Retrieved March 14, 2022, from https://webdeasy.de/en/multiplayer-game-programming-with-vue-js/

Vue.js Examples. (n.d.). A simple snake game written with Vue.js. Retrieved March 14, 2022, from https://vuejsexamples.com/a-simple-snake-game-written-with-vue-js/

Learning Outcome 4: Develop Game

Indicative contents

4.1 Design game interface

4.2 Develop game functionalities

4.3 Deploy game project on Netlify

Key Competencies for Learning Outcome 4 : Develop Game

Knowledge	Skills	Attitudes
• Description key concepts about Game development	• Creating Narrative • Designing game interface • Developing game functionalities • Deploying game project on Netlify	• Having Team work spirit • Being critical thinker • Being Innovative • Being attentive • Being creative • Being Problem solver • Being Practical oriented

Duration: 45 hrs

Learning outcome 4 objectives:

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

1. Describe clearly key concepts about game development based Vue JS framework
2. Design Properly game characters based on the storyline
3. Develop correctly game functionalities based on game purpose
4. Create correctly narrative as used in game development.
5. Deploy correctly game on static hosting platforms based on system requirements

Resources

Equipment	Tools	Materials
• Computer	• Vue js • Node js • Browser • VS Code • Illustrator	• Internet

Indicative content 4.1: Design Game Interface

Duration: 10 hrs

Theoretical Activity 4.1.1: Description of the key concepts used to develop game in VueJS Framework

Tasks:

1. You are requested to answer the following questions related to the Key concepts used to develop game in Vue Js framework:
   I. What do you understand about the term deployment?
   II. Name different deployment platforms that are commonly used.
   III. Explain domain name
   IV. How do SASS and SVG differ from each other?
   V. What do you understand about the term canvas?
2. Provide the answer of asked questions by writing them on paper.
3. Present your findings to your classmates and trainer
4. For more clarification, read the key readings 4.1.1. In addition, ask questions where necessary.

Key readings 4.1.1.: Description of the key concepts used to Develop Game in VueJS Framework

✓ **Deployment**

Game deployment refers to the process of releasing a video game for distribution and making it available to players. It involves several steps to ensure that the game is ready for release and can be accessed by the intended audience

✓ **Deployment/Hosting platforms**

Amazon GameLift: Amazon GameLift is a fully managed service by Amazon Web Services (AWS) that provides scalable hosting for multiplayer games. It offers features such as session management, matchmaking, and real-time game analytics.

Microsoft Azure PlayFab: PlayFab is a game development platform powered by Microsoft Azure. It provides services for multiplayer game hosting, player authentication, matchmaking, and player data storage, among other features.

Google Cloud Game Servers: Google Cloud Game Servers is a managed hosting solution that offers high-performance infrastructure for multiplayer games. It supports various game engines and provides features like autoscaling, game server management, and matchmaking.

Game Server Providers: There are numerous third-party game server providers that offer hosting services for specific games or game genres. Examples include Nitrado, Multiplay, and Survival Servers. These providers often specialize in hosting specific game titles or offer a wide range of game server options.

Unity Multiplayer: Unity Multiplayer is a networking solution provided by Unity Technologies. It allows developers to create and host multiplayer games using Unity game engine. Unity Multiplayer includes features like matchmaking, relay servers, and networking infrastructure.

Steam works: Steam works is a suite of tools and services provided by the Steam platform. It includes features for multiplayer matchmaking, lobby management, and dedicated server hosting. Steam works primarily focused on PC gaming and is widely used by developers on the Steam platform.

Minecraft Realms: Minecraft Realms is an official hosting service provided by Mojang Studios for the popular game Minecraft. It offers a simple and straightforward way to host Minecraft servers, allowing players to easily invite friends and manage their gameplay experience

Domain name: A domain name is a unique address that identifies a website on the internet. It serves as a user-friendly way to access websites, as it is easier to remember and type than the numerical IP address associated with a website's server

SASS

Sass (Syntactically Awesome Style Sheets) is a preprocessor scripting language that is used to extend the capabilities of CSS (Cascading Style Sheets). It provides a more efficient and powerful way to write CSS code, allowing developers to write cleaner, more maintainable stylesheets.

CANVAS: A game canvas typically refers to the area or surface where the game graphics and gameplay elements are rendered or displayed. It serves as the visual representation of the game world and allows players to interact with the game.

SVG: SVG (Scalable Vector Graphics) can be used in games for various purposes, such as creating game assets, UI elements, and animations. SVG is a vector-based image format that allows for smooth scaling and resizing without losing quality, which makes it suitable for different screen resolutions and devices.

Practical Activity 4.1.2: Design game environment using html canvas

Task:

1. Read key reading 4.2.2 and ask clarification where necessary

2. Referring to the previous theoretical activities (4.1.1), you are requested to go to the computer lab to design game environment using html canvas. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to design game environment using html canvas.

5. Referring to the steps provided on task 3, design game environment using html canvas.

6. Present your work to the trainer and whole class

Key readings 4.1.2.: Design game environment using html canvas

Design game environment

✓ Setup Html Canvas

Step 1: Create an HTML file

Open a text editor and create a new file.

Save it with a .html extension (e.g., index.html). This file will contain your HTML code.

Step 2: Set up the HTML structure

Inside the HTML file, start by setting up the basic structure. Add the following code:

```
<!DOCTYPE html>

<html>

<head>

  <title>Canvas Example</title>

</head>

<body>
```

```
  <canvas id="myCanvas"></canvas>
</body>
</html>
```

Step 3: Style the canvas

You can add some CSS styles to the canvas element if you want to customize its appearance. For example:

```
<style>
  #myCanvas {
    border: 1px solid black;
    background-color: #f1f1f1;
  }
</style>
```

Step 4: Add JavaScript code

To interact with the canvas, you'll need to use JavaScript. Add the following code just before the closing `</body>` tag:

```
<script>
  const canvas = document.getElementById('myCanvas');
  const context = canvas.getContext('2d');
  // Your drawing code goes here
</script>
```

Step 5: Start drawing on the canvas

Now that the basic setup is complete, you can start adding your drawing code within the JavaScript block. For example, let's draw a simple rectangle:

```
<script>
  const canvas = document.getElementById('myCanvas');
  const context = canvas.getContext('2d');
  // Draw a rectangle
  context.fillStyle = 'red';
  context.fillRect(50, 50, 100, 100);
</script>
```

Step 6: Save and open the HTML file

Save the HTML file and open it in a web browser. You should see your canvas displayed with the drawing you specified.

✓Style Environment using SASS

To style an HTML canvas using SASS (Syntactically Awesome Style Sheets), you can follow these steps:

1. Create a new SASS file with a `.scss` extension. For example, you can name it `styles.scss`.

2. Link the SASS file to your HTML file using the `<link>` tag in the `<head>` section of your HTML file. Make sure to include the compiled CSS file generated from the SASS file. For example:

```
<link rel="stylesheet" href="styles.css">
```

3. Open the `styles.scss` file in a text editor and write your SASS code to style the canvas element. For example:

```
canvas {
 width: 500px;
 height: 300px;
background-color: #eee;
border: 1px solid #ccc;
// Add more styles as needed
}
```

4. Save the `styles.scss` file.

5. Compile the SASS file to generate the CSS file. You can use a SASS compiler like Dart Sass, LibSass, or a build tool like Gulp or Webpack. If you're using Dart Sass, open a command prompt or terminal in the directory where the `styles.scss` file is located and run the following command:

```
sass styles.scss styles.css
```

This command will compile the `styles.scss` file into `styles.css`.

6. The compiled CSS file (`styles.css`) will now contain the CSS generated from your SASS code. Make sure the CSS file is correctly linked in your HTML file, as mentioned in step 2.

7. Open your HTML file in a web browser, and the canvas element should now be styled according to the rules defined in your SASS code.

Practical Activity 4.1.3: Design environment components with SVG or Illustrator

Task:

1. Read key reading 4.1.3 and ask clarification where necessary

2. Referring to the previous theoretical activities (4.1.1), you are requested to go to the computer lab to design environment components with SVG or Illustrator.

This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to design environment components with SVG or Illustrator.

5. Referring to the steps provided on task 3, design environment components with SVG or Illustrator.

6. Present your work to the trainer and whole class

Key readings 4.1.3.: Design environment components with SVG or Illustrator

✓ **Design environment components with SVG**

Designing environment components using SVG (Scalable Vector Graphics) involves creating scalable and visually appealing graphics for web or print projects. Here are the steps to design environment components using SVG:

Step 1: Set up a Vue.js project

Start by setting up a new Vue.js project using the Vue CLI or your preferred method. Make sure you have Vue.js and its dependencies installed.

Step 2: Create a component for the game environment

Create a new Vue component that will represent your game environment. You can use the Vue CLI to scaffold a new component or manually create a new `.vue` file. For example, you can create a `GameEnvironment.vue` file.

Step 3: Add an SVG element to the component

Inside your `GameEnvironment.vue` component, add an SVG element to serve as the canvas for your game environment. You can do this by using the `<svg>` tag.

For example:

```
<template>
  <div>
    <svg>
      <!-- SVG content goes here -->
    </svg>
  </div>
</template>
```

Step 4: Design your game environment using SVG elements

Within the `<svg>` tag, you can design your game environment using various SVG elements such as `<rect>`, `<circle>`, `<line>`, and so on. You can position and style these elements using SVG attributes and CSS classes.

For example:

```
<template>
  <div>
    <svg>
      <rect class="player" x="50" y="50" width="50" height="50"></rect>
      <circle class="obstacle" cx="150" cy="150" r="25"></circle>
      <!-- Other SVG elements -->
    </svg>
  </div>
</template>
<style scoped>
```

```
.player {
  fill: blue;
}
.obstacle {
  fill: red;
}
</style>
```

Step 5: Bind SVG attributes to data properties

To make your game environment dynamic, you can bind SVG attributes to data properties in your component. This allows you to update the position, size, and other attributes of SVG elements based on changes in your component's data. For example

```
<template>
  <div>
    <svg>
      <rect class="player" :x="playerX" :y="playerY" width="50" height="50"></rect>
      <!-- Other SVG elements -->
    </svg>
  </div>
</template>
<script>
export default {
  data() {
    return {
      playerX: 50,
      playerY: 50,
      // Other data properties
    };
  },
};
</script>
```

Step 6: Add interactivity and animations

SVG elements can be made interactive and animated using Vue.js event handling and transitions. You can bind event listeners to SVG elements and update data properties accordingly. You can also use Vue.js transition components to animate the SVG elements. **For example:**

```
<template>
  <div>
    <svg @click="movePlayer">
      <transition name="slide">
        <rect v-if="showPlayer" class="player" :x="playerX" :y="playerY" width="50" height="50"></rect>
      </transition>
      <!-- Other SVG elements -->
    </svg>
  </div>
</template>
<script>
export default {
  data() {
    return {
      playerX: 50,
      playerY: 50,
      showPlayer: true,
      // Other data properties
    };
   },
   methods: {
    movePlayer() {
```

- ✓ **Design environment components with Adobe Illustrator**

Here are the steps to design environment components using Adobe Illustrator:

Step 1: Set Up Your Workspace

Open Adobe Illustrator and create a new document. Choose dimensions suitable for your game (e.g., 1920x1080 pixels for HD or the specific resolution of your game).

Set the Color Mode to RGB for digital outputs.

Use layers to organize your components (e.g., Background, UI Elements, Characters).

Step 2: Research and Conceptualize

Analyze the Game Theme: Determine the style (cartoonish, realistic, futuristic, etc.) and mood (bright, dark, mysterious).

Sketch on Paper: Draw rough ideas of your environment components (e.g., trees, buildings, paths, obstacles).

Gather References: Look for inspiration from similar games or concept art.

Step 3: Create a Background

Use the Rectangle Tool (M) to draw a base background (e.g., sky or ground).

Add gradients for a dynamic look (e.g., sky transitioning from light blue to dark blue).

Incorporate texture using Patterns or Textures (Window → Swatches → Patterns).

Step 4: Design Environmental Components

Trees, Rocks, and Grass:

Use the Pen Tool (P) for custom shapes.

Apply Gradient Fills and Shadows for depth.

Group elements together (Ctrl+G or Cmd+G) for easier manipulation.

Buildings and Structures:

Use the Shape Tools (Rectangle, Ellipse, Polygon) for geometric elements.

Add details like windows or roofs using Boolean Operations in the Pathfinder panel.

Water or Clouds:

For water, create wavy lines using the Pen Tool or Brush Tool.

For clouds, combine multiple ellipses and smooth their edges.

Step 5: Add UI Elements

Design buttons, scoreboards, and health bars using the Rounded Rectangle Tool.

Add icons (e.g., stars, hearts) using pre-made vector shapes or the Pen Tool.

Use contrasting colors to ensure visibility against the background.

Step 6: Use Layers and Groups

Organize components into layers (e.g., foreground, midground, and background).

Group similar elements (e.g., all tree layers) for easy adjustments.

Step 7: Add Texture and Effects

Use Texture Brushes to create a more organic look.

Apply Blur Effects (Effect → Blur → Gaussian Blur) to simulate depth or focus.

Use Opacity Masks for fading or blending elements.

Step 8: Test Scalability and Resolution

Zoom in and out to ensure details look good at various sizes.

Check for pixelation issues when exporting for the game's resolution.

Step 9: Export Assets

Export components individually (File → Export → Export As), using formats like PNG or SVG.

Ensure transparency for elements that need to overlay (e.g., trees or UI components).

Practical Activity 4.1.4: Designing game HUD (heads-up display)

Task:

1. Read key reading 4.1.4 and ask clarification where necessary

2. Referring to the previous theoretical activities (3.1.1), you are requested to go to the computer lab to design game HUD (heads-up display). This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to design game HUD (heads-up display).

5. Referring to the steps provided on task 3, design game HUD (heads-up display)

6. Present your work to the trainer and whole class

Key readings 4.1.4.: Designing game HUD (heads-up display)

Designing a game HUD (heads-up display) is crucial for providing players with essential information and interactive elements during gameplay. Here are the steps to design a game HUD effectively:

1.Define HUD Elements:

Identify the key information and interactive elements that need to be displayed on the HUD, such as health bars, score counters, minimaps, inventory icons, and game controls.

2.Sketch a Layout:

Create a rough sketch or wireframe of the HUD layout, indicating the placement of each element on the screen. Consider the visual hierarchy and organization of information for easy readability.

3.Create a New Document:

Open a design tool like Adobe Illustrator or a graphic editing software and create a new document with the dimensions matching the game screen resolution.

4.Design Visual Elements:

Design graphics for each HUD element, including icons, buttons, progress bars, text styles, and background shapes. Ensure consistency in style and color scheme throughout the HUD.

5.Organize Elements:

Arrange the HUD elements on the canvas according to your layout sketch. Use layers to keep different elements separate for easy editing and management.

6.Add Interactivity:

Design interactive elements like buttons or sliders that players can interact with during gameplay. Ensure these elements are visually distinguishable and responsive to user input.

7.Incorporate Game Branding:

Include elements of the game's branding, such as logos, fonts, and color schemes, to maintain consistency with the overall game design.

8.Test for Readability and Usability:

Check the readability and usability of the HUD design by simulating gameplay scenarios. Ensure that information is displayed clearly and that interactive elements are easily accessible.

9.Optimize for Different Screen Sizes:

Consider responsive design principles to ensure that the HUD elements scale appropriately for different screen sizes and resolutions.

10.Export Assets:

Export the HUD design elements as separate assets (PNG, SVG, etc.) for integration into the game development environment.

11.Integrate into Game:

Work with game developers to integrate the HUD design assets into the game engine. Test the HUD in the actual game environment to ensure proper functionality and alignment.

In Vue.js, you can design containers for game stats by using components. Components in Vue.js allow you to encapsulate and reuse code, making it easier to manage and organize your application.

Example:

1. Create a new Vue component file, let's call it `GameStatsContainer.vue`. This file will contain the container component for the game stats.

2. In the `GameStatsContainer.vue` file, define the template section where you'll write the HTML structure for your container.

For example:

```
<template>
  <div class="game-stats-container">
    <!-- Game stats content here -->
  </div>
</template>
```

3. Add any necessary CSS classes or styles to the container component to customize its appearance.

For example:

```
<style scoped>
.game-stats-container {
  background-color: #f5f5f5;
  padding: 20px;
  border-radius: 5px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
</style>
```

4. Inside the container component, you can add additional child components or elements to display the game stats. For example, you might have components for displaying player scores, game timer, and other relevant information. You can also use data properties or props to pass dynamic data to these child components.

```
<template>
  <div class="game-stats-container">
    <PlayerScore :score="playerScore" />
    <GameTimer :time="gameTime" />
    <!-- Other game stats components -->
  </div>
</template>
<script>
import PlayerScore from './PlayerScore.vue';
import GameTimer from './GameTimer.vue';
export default {
  components: {
    PlayerScore,
    GameTimer
  },
  data() {
    return {
      playerScore: 0,
      gameTime: 120
    };
  }
};
</script>
```

5. Make sure to import and register any child components used inside the container component. In the example above, `PlayerScore` and `GameTimer` are imported and registered as child components.

6. Customize the container component as per your game's design and requirements. You can add event handling, computed properties, and methods to manipulate the game stats data.

●Design container for character stats

To design a container for character stats in Vue.js, you can follow these steps:

1. Set up a new Vue.js project or navigate to the existing project directory where you want to add the character stats container.

2. Create a new Vue component file for the character stats container. You can name it something like `CharacterStatsContainer.vue`.

3. In the `CharacterStatsContainer.vue` file, define the template for your container. You can use HTML markup to structure the container elements. For example:

```
<template>
  <div class="character-stats-container">
    <h2>Character Stats</h2>
    <!-- Place your character stat elements here -->
  </div>
</template>
```

4. Define the styles for the container in the component's `<style>` section or in a separate CSS file. You can use CSS or a CSS preprocessor like Sass or Less to style your container. Here's an example using CSS:

```
<style>
.character-stats-container {
  border: 1px solid #ccc;
  padding: 10px;
  background-color: #f5f5f5;
}
</style>
```

5. Define the necessary data properties in the Vue component's `<script>` section. These properties will hold the character stats that you want to display in the container. For example:

```
<script>
export default {
  data() {
    return {
      characterName: 'John Doe',
      level: 10,
      healthPoints: 100,
      attackPoints: 50,
      defensePoints: 30,
      // Add more character stats as needed
    };
  },
};
</script>
```

6. Use the data properties within the template to display the character stats. You can use Vue's data binding syntax (double curly braces) or directives like `v-bind` to display the values dynamically. For example:

```
<template>
  <div class="character-stats-container">
    <h2>Character Stats</h2>
    <p>Character Name: {{ characterName }}</p>
    <p>Level: {{ level }}</p>
    <p>Health Points: {{ healthPoints }}</p>
    <p>Attack Points: {{ attackPoints }}</p>
    <p>Defense Points: {{ defensePoints }}</p>
  </div>
</template>
```

7. Optionally, you can enhance the container by adding computed properties or methods to perform calculations or transformations on the character stats.

8. Finally, you can include the `CharacterStatsContainer` component in your application by importing and registering it in a parent component or in your main Vue instance.

●Design container for character resources (armor, weapon, tools)

To design a container for character resources in Vue.js, you can follow these steps:

1. Create a new Vue component: Start by creating a new Vue component file, such as `CharacterContainer.vue`, where you'll define the container for character resources.

2. Define the template: In the template section of the component file, define the structure and layout of the container. This could be a `<div>` or any other suitable HTML element.

```
<template>
 <div class="character-container">
  <!-- Your character resource content here -->
 </div>
</template>
```

3. Style the container: Add CSS styles to the component to give it the desired appearance. You can use inline styles, CSS classes, or even CSS frameworks like Bootstrap or Tailwind CSS.

```
<template>
 <div class="character-container">
  <!-- Your character resource content here -->
 </div>
</template>
<style>
.character-container {
 /* Add your styles here */
}
</style>
```

4. Add props or data: Decide what data you want to display in the container. You can pass this data to the component as props or define it within the component using data properties.

```
<template>
```

```
  <div class="character-container">
    <h2>{{ characterName }}</h2>
    <!-- Additional character resource content here -->
  </div>
</template>
<script>
export default {
  props: {
    characterName: {
      type: String,
      required: true
    }
  }
};
</script>
```

In the example above, the component expects a `characterName` prop of type `String`, which is required to be passed when using the component.

5. Use the container component: Now, you can use the `CharacterContainer` component within other components by importing it and including it in their templates. Pass the necessary props as needed.

Practical Activity 4.1.5: Designing game characters

Task:

1. Read key reading 4.2.5 and ask clarification where necessary

2. Referring to the previous theoretical activities (3.1.1), you are requested to go to the computer lab to design game characters. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to design game characters.

5. Referring to the steps provided on task 3, design game characters

6. Present your work to the trainer and whole class

Key readings 4.1.5.: Designing game characters

Design characters using Illustrator

1. **Plan and Sketch:** Begin by brainstorming and sketching ideas for your game characters. Consider their roles, personalities, and any specific traits or features they should have. Sketch out different poses, outfits, and facial expressions to explore various design options.

2. **Set Up the Workspace:** Launch Adobe Illustrator and create a new document with the desired dimensions for your character. Make sure to set the resolution suitable for your game's requirements.

3. **Basic Shapes:** Start by using basic shapes like rectangles, circles, and ellipses to create the outline of your character. Use the shape tools in Illustrator's toolbar to draw the body, head, limbs, and other components of your character.

4. **Refine the Outlines:** Adjust and refine the shapes to match your character design. Utilize the Direct Selection Tool (A) and the Pen Tool (P) to modify and fine-tune the curves and angles of your character's body parts.

5. **Fill and Stroke:** Choose appropriate colors for your character's body parts. Use the Fill and Stroke options in the toolbar to apply colors and outline styles to the shapes. Experiment with gradients, patterns, and strokes to enhance the visual appeal of your character.

6. **Details and Features:** Add details like facial features, hair, clothing, accessories, and other distinguishing elements. Use a combination of shapes and the Pen Tool to create these details. Incorporate layers to organize different parts of your character for easier editing and manipulation.

7. **Shadows and Highlights:** Create depth and dimension by adding shadows and highlights to your character. Experiment with different shades and opacities to achieve the desired effect. Consider the light source and apply shadows accordingly to create a more realistic appearance.

8. **Textures and Patterns:** Apply textures or patterns to specific parts of your character, such as clothing or accessories. Illustrator provides various tools for

creating and applying textures, including the Pattern Fill and Clipping Mask options.

9. Finalize and Export: Review your character design, make any necessary adjustments, and ensure all elements are cohesive. Once satisfied, save your Illustrator file. You can then export your character as a PNG, SVG, or any other appropriate format for use in your Vue.js game.

Remember, this is just a general guide, and the specific steps may vary depending on your character design and style. Practice, explore different techniques, and don't be afraid to experiment to create unique and engaging game characters using Illustrator and Vue.js.

✓ Design characters with SVG

Designing game characters using SVG (Scalable Vector Graphics) in Vue.js can be a creative and engaging process. Here's a step-by-step guide to help you design Vue.js game characters using SVG:

1. **Set up your Vue.js project:** Begin by setting up a new Vue.js project or using an existing one. Install any required dependencies, such as the `vue-svg-loader` or `vue-svgicon` packages, which enable you to work with SVG files in Vue.js.

2. **Plan your character:** Before diving into the design, spend some time planning your character. Consider its appearance, characteristics, and any animations or interactions you want to incorporate.

3. **Create an SVG component:** In your Vue.js project, create a new component specifically for your game character. You can do this by generating a new `.vue` file, for example, `Character.vue`.

4. **Define the character's SVG markup:** Open the `Character.vue` file and define the SVG markup within the template section. You can start with a basic structure, including the necessary SVG elements such as `<svg>`, `<rect>`, `<circle>`, `<path>`, etc. Customize the size and position of the SVG elements to create the desired character shape.

5. **Add data properties and methods:** Within the `Character.vue` component, define any necessary data properties and methods. For example, you might have properties like `positionX` and `positionY` to track the character's position, or `health` to represent its health status. Additionally, methods can be used to handle character movements or interactions.

6**. Style the character:** Apply CSS styles to your SVG elements using Vue.js's scoped styles or CSS modules. Define the desired colors, strokes, gradients, or animations to give your character its unique appearance.

7. **Bind SVG attributes to data properties:** To make your character dynamic and responsive, use Vue.js directives to bind SVG attributes to your component's data properties. For example, you can bind the `cx` and `cy` attributes of a `<circle>` element to the `positionX` and `positionY` data properties, respectively.

8. Implement interactions and animations: Use Vue.js event handling and animation features to implement character interactions and animations. For instance, you can use `@click` event listeners to trigger specific actions when the character is clicked or apply CSS transitions to animate movements.

9. **Test and refine:** Test your character component within your Vue.js project, making adjustments as needed. Iterate on the design and functionality until you achieve the desired results.

10. **Reuse and extend:** Once you've designed one character, you can reuse the SVG component as a template for creating additional characters. Customize the SVG markup, styles, and data properties to differentiate each character.

Points to Remember

- Various platforms offer different features, advantages and capabilities.
  Here are different deployment platforms/Hosting platforms used: Amazon GameLift, Google Cloud Game Servers, Unity Multiplayer, Minecraft Realms.
- There are differences between SASS and SVG where SASS (Syntactically Awesome Style Sheets) is a preprocessor scripting language that is used to extend the capabilities of CSS (Cascading Style Sheets). while SVG (Scalable Vector Graphics) can be used in games for various purposes, such as creating game assets, UI elements, and animations.
- Designing a game environment using HTML canvas involves several steps to create a visually engaging and interactive space for gameplay. Here are the steps to design a game environment using HTML canvas:

  Step 1: Create an HTML file
  Step 2: Set up the HTML structure
  Step 3: Style the canvas
  Step 4: Add JavaScript code
  Step 5: Start drawing on the canvas
  Step 6: Save and open the HTML file
- Designing environment components using SVG (Scalable Vector Graphics) or Adobe Illustrator involves creating scalable and visually appealing graphics for web or print projects. Here are the steps to design environment components using SVG or Illustrator:

  Step 1: Set up a Vue.js project
  Step 2: Create a component for the game environment
  Step 3: Add an SVG element to the component
  Step 4: Design your game environment using SVG elements
  Step 5: Bind SVG attributes to data properties
  Step 6: Add interactivity and animations
- **Designing a game HUD (heads-up display) is crucial for providing players with essential information and interactive elements during gameplay. Here are the steps to design a game HUD effectively:**

  1. Define HUD Elements
  2. Sketch a Layout
  3. Create a New Document
  4. Design Visual Elements
  5. Organize Elements
  6. Optimize for Different Screen Sizes
  7. Integrate into Game
- **Design characters using Illustrator**

1. Plan and Sketch
2. Set Up the Workspace
3. Basic Shapes
4. Refine the Outlines
5. Fill and Stroke

- **Design characters with SVG**

Designing game characters using SVG (Scalable Vector Graphics) in Vue.js can be a creative and engaging process. Here's a step-by-step guide to help you design Vue.js game characters using SVG:

1. Set up your Vue.js project
2. Plan your character
3. Create an SVG component
4. Define the character's SVG mark-up
5. Add data properties and methods
6. Style the character
7. Bind SVG attributes to data properties
8. Reuse and extend

Application of learning 4.1.

X Rwandan museum is a museum located in Musanze district, Muhoza sector, they have a campaign directed toward educating children about historical figures and their contribution to our history. In the beginning, this campaign was conducted via historians in the museum explaining the children about those historical figures, but this method was ineffective since children would get bored and stop paying attention.

X Rwandan Museum would like to hire a game developer, to build a game where children would learn while having fun. The game to develop will be a picture slider puzzle, where the user will get a picture with pieces arranged randomly and will have to rearrange them by clicking on the piece to move. You are requested to do the following:

- ✓ Designing game environment using html canvas
- ✓ Designing environment components with SVG or Illustrator
- ✓ Designing game HUD (heads-up display)
- ✓ Designing game characters

All tools, materials and equipment will be provided by the company.

Indicative content 4.2: Develop Game Functionalities

Duration: 20 hrs

Practical Activity 4.2.1: Develop game settings page/section

Task:

1. Read key reading 4.2.1 and ask clarification where necessary

2. Referring to the previous theoretical activities (3.1.1), you are requested to go to the computer lab to develop a game setting page/section. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to develop a game setting page/section.

5. Referring to the steps provided on task 3, develop a game setting page/section.

6. Present your work to the trainer and whole class

Key readings 4.2.1.: Develop game settings page/section

To develop a game settings page/section in Vue.js, you can follow these general steps:

1. Set up a new Vue.js project

If you don't have a Vue.js project set up already, create a new one using Vue CLI or any other method of your choice.

2. Define setting section

In the component file, define the template section. This will contain the HTML structure for your setting section. You can use standard HTML tags or Vue.js directives for dynamic rendering.

```
<template>
 <div class="config-row">
   <div>Change level</div>
   <div class="game-levels">
```

```
    <button type="secondary" class="button">1</button>
    <button type="secondary" class="button">2</button>
    <button type="secondary" class="button">3</button>
  </div>
 </div>
</template>
```

3. Declare and bind variable: Within your component, define data properties to store the settings data. For instance, you may have properties like `volume`, `level`, `language`, etc. Initialize these properties with default values.

Example:

```
<script>
export default {
 name: "GameWrapper",
 data() {
  return {
    GAME_LEVEL: 1,
  };
 },
};
</script>
```

"The preceding code snippet demonstrates how the variable GAME_LEVEL is declared. Now let's proceed to incorporate it into the HTML code.

```
<template>
 <div class="config-row">
  <div>Active level: {{ GAME_LEVEL }}</div>
  <div>Change level</div>
  <div class="game-levels">
    <button type="secondary" class="button">1</button>
```

```
      <button type="secondary" class="button">2</button>

      <button type="secondary" class="button">3</button>

    </div>

  </div>

</template>
```

Practical Activity 4.2.2: Manipulating events

Task:

1. Read key reading 4.2.2 and ask clarification where necessary

2. Referring to the previous practical activities (4.1.1), you are requested to go to the computer lab to manipulate events. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to manipulate events.

5. Referring to the steps provided on task 3, manipulate events.

6. Present your work to the trainer and whole class

Key readings 4.2.2.: Manipulating events

Manipulating events in a game involves controlling and influencing various actions, triggers, and outcomes within the game world to create engaging gameplay experiences. Here are the steps to manipulate events effectively in a game:

1.**Define Event Types:**

Identify the types of events you want to manipulate in the game, such as player actions, enemy behavior, environmental changes, power-ups, obstacles, and story progression events.

2.**Event System Setup:**

Set up an event system in your game code that can handle different types of events and trigger appropriate responses based on specific conditions or inputs.

3.Event Triggers:

Define triggers that initiate events, such as player interactions, time-based triggers, location-based triggers, or specific game states. These triggers will activate the events you want to manipulate.

4.Event Actions:

Specify the actions or behaviors associated with each event. This could include character movements, object interactions, spawning enemies, changing game variables, triggering animations, or altering the game environment.

5.Event Conditions:

Set conditions that determine when events should be triggered or manipulated. Conditions could be based on player progress, in-game events, environmental factors, or other variables that affect gameplay.

6.Event Sequencing:

Arrange events in a sequence to create a coherent gameplay flow. Consider the order in which events should occur and how they relate to each other to build engaging gameplay scenarios.

7.Event Feedback:

Provide feedback to players to indicate the occurrence and effects of manipulated events. Visual and audio cues, UI notifications, animations, and changes in the game world can help communicate event outcomes to players.

8.Testing and Iteration:

Test the manipulated events in different gameplay scenarios to ensure they work as intended and enhance the player experience. Iterate on event sequences based on player feedback and playtesting results.

9.Balancing Events:

Balance the difficulty, frequency, and impact of manipulated events to maintain a challenging yet enjoyable gameplay experience. Adjust event parameters to ensure a fair and rewarding gameplay balance.

Defining methods can be achieved by either directly storing the value in a declared variable or by sending it via an API request to a backend server, utilizing local storage, saving it in the store, or employing any other method that suits your project.

Example:

```
<template>
 <div class="config-row">
   <div>Active level: {{ GAME_LEVEL }}</div>
   <div>Change level</div>
   <div class="game-levels">
     <button type="secondary" class="button" @click="updateGameLevel(1)
>1</button>
     <button type="secondary" class="button"@click="updateGameLevel(2)
>2</button>
     <button type="secondary" class="button" @click="updateGameLevel(3)
>3</button>
   </div>
 </div>
</template>
<script>
export default {
 name: "GameWrapper",
 data() {
   return {
     GAME_LEVEL: 1,
   };
 },
 methods: {
   updateGameLevel(level) {
     this.GAME_LEVEL = level;
   },
 },
};
</script>
```

Setup animation speed

To set up the animation speed of an element, you can use CSS transitions. Use CSS class to define the animation with the desired style properties. For instance, you can create a class called `animation-fast` with a shorter animation duration.

```
<template>
 <div>
   <div class="animation-object">...</div>
 </div>
</template>
<style>
.animation-object {
 width: 50px;
 height: 50px;
 background: black;
 animation: animation-object 2s infinite linear;
 @keyframes animation-object {
   0% { left: 0; }
   100% { left: 100%; }
 }
}
</style>
```

In Vue game development, you can leverage the synergy between CSS features and Vue's functionalities. Here's a code snippets showcasing how you can animate objects like game characters, non-playable characters, or obstacles.

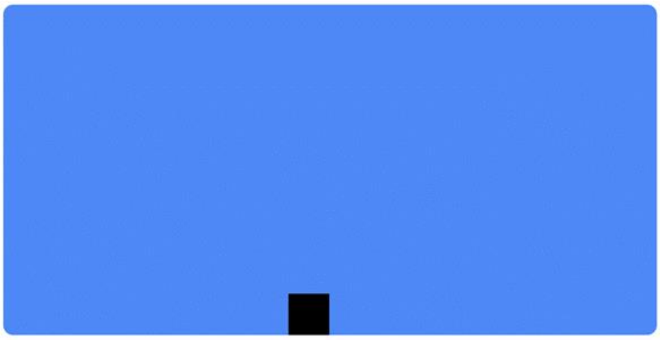
```
<template>
 <div class="game">
   <div class="obstacle" :style="obstacleStyle"></div>
 </div>
</template>
<script>
export default {
 name: "App",
 data: () => ({
   SPEED: 10,
   GAME_INTERVAL: null,
   obstaclePosition: {
     x: window.innerWidth,
   },
 }),
 mounted() {
   this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
 },
 methods: {
   gameLoop() {
     this.updateObstaclePosition();
     this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
   },
   updateObstaclePosition() {
     this.obstaclePosition.x -= this.SPEED;
     if (this.obstaclePosition.x < -50) {
       this.obstaclePosition.x = window.innerWidth;
     }
   },
 },
 computed: {
   obstacleStyle() {
     return {
       left: `${this.obstaclePosition.x}px`,
     };
   },
 },
};
</script>

<style>
.game {
 display: flex;
 width: 100%;
 height: 100vh;
 background: #3f86ff;
 position: relative;
 width: 800px;
 height: 400px;
 overflow: hidden;
 border-radius: 12px;
}
.obstacle {
 position: absolute;
 bottom: 0;
 left: 0;
 width: 50px;
 height: 50px;
 background: black;
}
</style>
```

Here, we set up the Vue component and define its behavior. We have properties like SPEED that controls how fast the obstacle moves, GAME_INTERVAL that manages the game loop, and obstaclePosition that tracks the obstacle's position.

The mounted lifecycle hook initiates the game loop when the component is mounted to the DOM. gameLoop() is a recursive function that updates the obstacle's position.

The CSS styles define the appearance of our game environment. The "game" class sets up a blue background with rounded corners, while the "obstacle" class represents a black square at the bottom.

Interface:



Practical Activity 4.2.3: Set up game conditions

Task:

1. Read key reading 4.2.3 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to set up game conditions. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to set up game conditions.

5. Referring to the steps provided on task 3, set up game conditions.

6. Present your work to the trainer and whole class

Key readings 4.2.3.: Set up game conditions

Setting up game conditions is essential for defining the rules, constraints, and triggers that govern gameplay mechanics and outcomes.

Here are the steps to set up game conditions effectively:

To define game conditions and repeatable actions, we can use the reactive nature of the framework to create data properties that represent the current state of the game. We are going to develop basic functionality for jumping when the spacebar is pressed, and reset the position of the character after 1 second.

Set the conditions

In the mounted lifecycle hook, when any key is pressed, we check if the spacebar is pressed. If it is, the jump function is triggered.

```
mounted() {
  window.addEventListener("keypress", (e) => {
    if (parseInt(e.charCode) === 32 && !this.GAME_OVER) {
      this.jump();
    }
  });
},
```

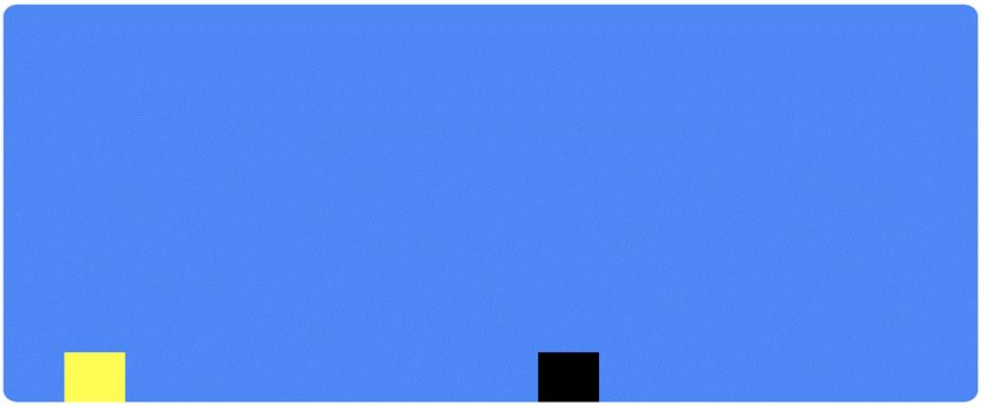
Loops and intervals

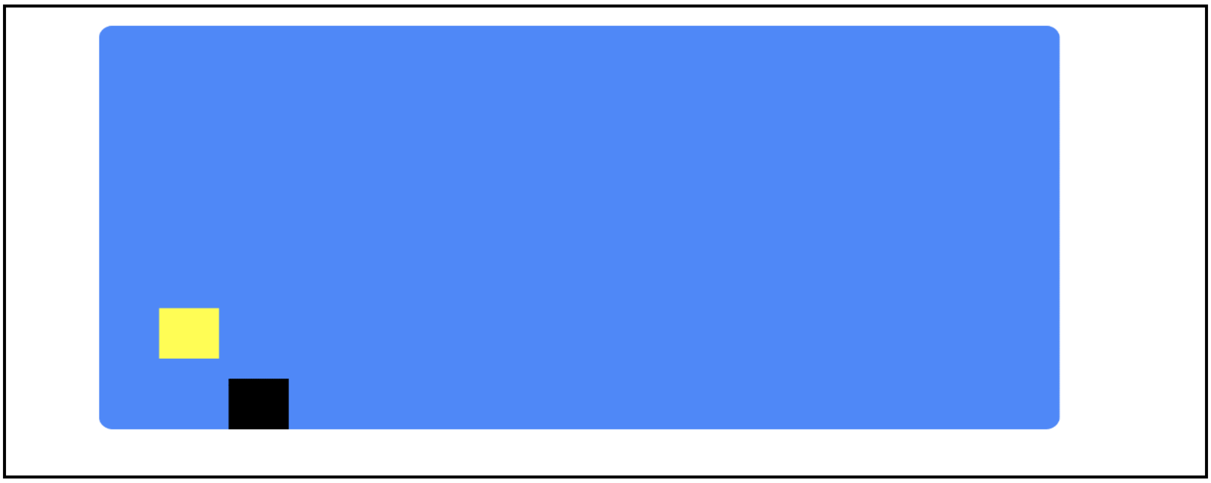
The requestAnimationFrame() function is called recursively in the gameLoop() method, creating a loop that keeps running as long as the game is active. This ensures that actions, like jumping, are timed to last 1 second.

The loop gives us the ability to move obstacle style by leveraging the updateObstaclePosition() method when it is called, synchronizing with the timed jumping action.

```
mounted() {
  window.addEventListener("keypress", (e) => {
    if (parseInt(e.charCode) === 32 && !this.GAME_OVER) {
      this.jump();
    }
  });
  this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
},
methods: {
  gameLoop() {
    this.updateObstaclePosition();
    this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
  },
  updateObstaclePosition() {
    this.obstaclePosition.x -= this.SPEED;
    if (this.obstaclePosition.x < -50) {
      this.obstaclePosition.x = window.innerWidth;
    }
  },
  jump() {
    this.characterPosition.y = 70;
    this.IS_JUMPING = true;
    setTimeout(() => {
      this.IS_JUMPING = false;
      this.characterPosition.y = 0;
    }, 1000);
  },
},
computed: {
  obstacleStyle() {
    return {
      left: `${this.obstaclePosition.x}px`,
    };
  },
  characterStyle() {
    return {
      bottom: `${this.characterPosition.y}px`,
    };
  },
},
```

Graphical User Interface





Practical Activity 4.2.4: Setting up random mechanisms to create diversity in the game

Task:

1. Read key reading 4.2.4 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to set up random mechanisms to create diversity in the game. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to set up random mechanisms to create diversity in the game.

5. Referring to the steps provided on task 3, set up random mechanisms to create diversity in the game.

6. Present your work to the trainer and whole class

Key readings 4.2.4.: Setting up random mechanisms to create diversity in the game

To set up random mechanisms for creating diversity in a game using Vue.js, you can follow these steps:

1. Generate Random Numbers

Use JavaScript's built-in `Math.random()` function to generate random numbers. You can multiply it by a range and add an offset to generate random numbers within a specific range. **For example**, to generate a random number between 1 and 10:

```
const randomNumber = Math.floor(Math.random() * 10) + 1;
```

2. Use Computed Properties

In Vue.js, you can use computed properties to generate random values that change dynamically. Computed properties are cached based on their reactive dependencies, so the value will only change when the dependencies change. This can be useful for generating random values within a specific scope or when certain conditions are met.

```
Data () {
  return {
     randomValue: 0,
  };
},
computed: {
  generateRandomValue() {
     return Math.floor(Math.random() * 10) + 1;
  },
},
```

3. Trigger Random Events

You can use Vue's event handling mechanisms to trigger random events. For example, you might have a button that, when clicked, triggers a random event. You can define an event handler method and call it when the button is clicked. Inside the event handler, you can generate random numbers and perform specific actions based on the outcome.

```
  <template>
    <div>
  <button @click="triggerRandomEvent">Trigger Random Event</button>
    </div>
  </template>
```

```
<script>
export default {
 methods: {
  triggerRandomEvent() {
   const randomNumber = Math.floor(Math.random() * 10) + 1;
   // Perform actions based on the random number
   if (randomNumber <= 5) {
    // Action 1
   } else {
    // Action 2
   }
  },
 },
};
</script>
```

Practical Activity 4.2.5: Setup incrementals for game scores and increase game difficulties

Task:

1. Read key reading 4.2.5 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to set up incrementals and increase game difficulties. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to set up incrementals and increase game difficulties.

5. Referring to the steps provided on task 3, Set up incrementals and increase game difficulties

6. Present your work to the trainer and whole class

Key readings 4.2.5.: Setup incrementals for game scores and increase game difficulties

Setting up incrementals and increasing game difficulties are important aspects of game design to keep players engaged and challenged as they progress. Here are the steps to set up incrementals and increase game difficulties effectively:

1.**Establish Base Difficulty**

Define the initial difficulty level of the game to provide a starting point for players. This base difficulty should be challenging enough to engage players without being overly frustrating.

2.**Identify Incremental Elements**

Identify aspects of the game that can be incrementally improved or upgraded as players progress, such as player abilities, enemy strength, level complexity, resource generation, or game mechanics.

3.**Define Incremental Progression**

Create a progression system that allows players to incrementally improve their skills, unlock new content, or gain advantages over time. This could involve leveling up, earning experience points, unlocking upgrades, or acquiring new abilities.

4.**Introduce Scaling Challenges**

Introduce challenges that scale in difficulty as players advance through the game. This could include tougher enemies, more complex puzzles, time constraints, limited resources, or increased obstacles to overcome.

5.**Adjust Enemy AI**

Modify enemy AI behavior to become more challenging as players progress. Increase enemy aggression, tactics, speed, and strength to provide a greater challenge for experienced players.

6.**Scale Rewards and Penalties**

Scale the rewards and penalties in the game based on player performance. Offer greater rewards for overcoming difficult challenges and impose harsher penalties for mistakes or failures to maintain a balance of risk and reward.

7.**Introduce New Mechanics**

Introduce new gameplay mechanics, features, or obstacles as players progress through the game. This keeps the gameplay experience fresh and engaging, requiring players to adapt to new challenges and strategies.

8.**Iterative Difficulty Adjustment**

Continuously monitor player feedback and playtesting results to adjust the game difficulty iteratively. Fine-tune the difficulty curve to ensure a gradual and satisfying increase in challenge without overwhelming players.

9.**Dynamic Difficulty Scaling**

Implement dynamic difficulty scaling mechanisms that adjust the game difficulty based on player performance or skill level. This ensures that the game remains challenging and engaging for players of different skill levels.

10.**Provide Difficulty Options**

Offer difficulty settings or options that allow players to customize their gameplay experience based on their preferences. Provide options for both casual and hardcore players to enjoy the game at their desired challenge level.

Below is an example of how to set up incrementals and increase game difficulties:

1. In the Vue component, declare variables and create methods that increase the score and change the level of game difficulty.

```
<script>
 data: {
  score: 0,
  difficulty: 1
 },
 methods: {
  incrementScore() {
   this.score += this.difficulty;
  },
  increaseDifficulty() {
   this.difficulty++;
  }
 }
});
<script>
```

2. In your HTML template, bind the score and difficulty data properties to the respective elements to display them.

```
<div id="app">
 <p>Score: {{ score }}</p>
 <p>Difficulty: {{ difficulty }}</p>
 <button @click="incrementScore">Increment Score</button>
 <button @click="increaseDifficulty">Increase Difficulty</button>
</div>
```

3. The `**incrementScore**` method increases the score based on the current difficulty level. Each time you click the "Increment Score" button, the score will be incremented by the current difficulty value.

4. The `**increaseDifficulty**` method increases the difficulty level by 1 each time you click the "Increase Difficulty" button.

Practical Activity 4.2.6: Create and display Alert messages

Task:

1. Read key reading 4.2.6 and ask clarification where necessary
2. You are requested to go to the computer lab to create and display alert messages. This task should be done individually.
3. Apply safety precautions.
4. Present out the steps to Create and display alert messages.
5. Referring to the steps provided on task 3, Create and display alert messages.
6. Present your work to the trainer and whole class

Key readings 4.2.6.: Create and display Alert messages

Creating and displaying alert messages in a game is essential for communicating important information, warnings, notifications, or feedback to players.

Here are the steps to create and display alert messages effectively:

1.**Define Alert Types**

Identify the different types of alert messages you need in the game, such as notifications, warnings, error messages, achievements, tutorial prompts, mission updates, or player feedback.

2.**Design Alert UI**

Design the user interface elements for displaying alert messages, including text boxes, pop-up windows, banners, icons, buttons, or other visual components that draw attention to the message.

3.**Create Alert Content**

Write clear and concise content for each alert message, ensuring that the message effectively conveys the intended information to the player. Use appropriate language, tone, and formatting for different types of alerts.

4.**Implement Alert System**

Implement an alert system in your game code that can handle the creation, management, and display of alert messages. This system should be able to queue and display alerts in a timely manner.

5.**Trigger Alert Events**

Define events or conditions that trigger the display of alert messages, such as completing a level, receiving a reward, encountering an obstacle, reaching a milestone, or interacting with a specific object.

6.**Display Timing**

Determine the timing and duration for displaying alert messages on the screen. Ensure that alerts appear at appropriate moments and remain visible for an adequate amount of time for players to read and understand the message.

7.**Visual and Audio Feedback**

Use visual and audio cues to draw attention to alert messages, such as flashing text, sound effects, animations, or highlighting the message with a distinct color or icon.

8.**Interactive Alerts**

Make some alert messages interactive, allowing players to dismiss, acknowledge, or respond to the message through buttons, choices, or actions that affect gameplay.

9.**Localization**

If your game supports multiple languages, ensure that alert messages are localized and displayed in the player's preferred language. Translate alert content and UI elements to provide a seamless experience for players worldwide.

10.**Testing and Feedback**

Test the alert system in various gameplay scenarios to ensure that messages are displayed correctly, are easily readable, and provide valuable information to players. Gather feedback from playtesters to improve the effectiveness of alert messages.

Example

1. Create an Alert component:

Create a new Vue component file, let's say `Alert.vue`, and define the following template and script:

```
<template>
  <div v-if="show" class="alert" :class="type">
    {{ message }}
    <button @click="dismissAlert">Close</button>
  </div>
</template>
<script>
export default {
  props: {
    type: {
      type: String,
      default: "info", // default alert type is "info"
    },
    message: {
      type: String,
      required: true,
    },
  },
  data() {
    return {
      show: true,
    };
  },
  methods: {
    dismissAlert() {
      this.show = false;
    },
  },
};
</script>
<style scoped>
.alert {
  padding: 10px;
  margin-bottom: 10px;
}
.info {
  background-color: #cce5ff;
}
```

```
.success {
  background-color: #d4edda;
}
.warning {
  background-color: #fff3cd;
}
.error {
  background-color: #f8d7da;
}
</style>
```

2. Use the Alert component in your main Vue component:

In your main Vue component, you can import and use the `Alert` component whenever you want to display an alert. For example:

```
<template>
  <div>
    <button @click="showAlert">Show Alert</button>
    <Alert v-if="displayAlert" :type="alertType" :message="alertMessage" />
  </div>
</template>
<script>
import Alert from ". /components/Alert.vue";
export default {
  components: {
    Alert,
  },
  data() {
    return {
      displayAlert: false,

      alertType: "",

      alertMessage: "",

    };

  },

  methods: {

    showAlert() {

      this.displayAlert = true;

      this.alertType = "success";

      this.alertMessage = "This is a success message.";

    },

  },

};

</script>
```

In the example above, a button triggers the `showAlert` method when clicked, which sets the necessary data properties (`displayAlert`, `alertType`, and `alertMessage`) to show the alert. You can customize the type and message of the alert as needed.

Practical Activity 4.2.7: Store data in state management

Task:

1. Read key reading 4.2.7 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to perform data store in state management. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to perform data store in state management.

5. Referring to the steps provided on task 3, Perform data store in state management.

6. Present your work to the trainer and whole class

Key readings 4.2.7.: Store data in state management

1. Install Vuex: If you haven't already, install Vuex by running the following command in your project directory:

```
npm install vuex
```

2. Create a store: In your project, create a new file called `store.js` or `store/index.js` to define your Vuex store. The store will contain your application's data and the methods to modify that data.

Here's an example of a basic store setup:

```
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);
const store = new Vuex.Store({
  state: {
    // Your application's data
    counter: 0,
    username: ''
  },
  mutations: {
    // Methods to modify the data
    incrementCounter(state) {
      state.counter++;
    },
    setUsername(state, username) {
      state.username = username;
    }
  },
  actions: {
    // Async actions to modify the data
    incrementCounterAsync({ commit }) {
      setTimeout(() => {
        commit('incrementCounter');
      }, 1000);
    }
  },
  getters: {
    // Computed properties based on the state
    counterSquared(state) {
      return state.counter ** 2;
    }
  }
});   export default store;
```

In this example, the store has a `state` object that holds the application's data, `mutations` to modify the data synchronously, `actions` to modify the data asynchronously, and `getters` to compute values based on the state.

3. **Connect the Store to your Vue application:** In your main Vue component file (usually `main.js`), import the Vuex store and connect it to your Vue application using the `store` option:

```
import Vue from 'vue';
import App from './App.vue';
import store from './store';
  new Vue({
   store,
   render: (h) => h(App)
  }).$mount('#app');
```

4. **Access and modify the data in your components:** Now you can access and

modify the data in your Vue components. Use the `mapState`, `mapMutations`, `mapActions`, and `mapGetters` helpers to simplify the process.

Here's an example component that uses the store:

```
<template>
 <div>
   <p>Counter: {{ counter }}</p>
   <p>Counter squared: {{ counterSquared }}</p>
   <input v-model="username" type="text" placeholder="Enter your name" />
   <button @click="incrementCounter">Increment Counter</button>
   <button @click="incrementCounterAsync">Increment Counter Async</button>
 </div>
</template>
<script>
import { mapState, mapMutations, mapActions, mapGetters } from "vuex";
export default {
 computed: {
   ...mapState(["counter", "username"]),
   ...mapGetters(["counterSquared"]),
 },
 methods: {
   ...mapMutations(["incrementCounter"]),
   ...mapActions(["incrementCounterAsync"]),
 },
};
</script>
```

In this example, the component uses the `mapState` helper to map the `counter` and `username` from the store's state to the component's computed properties. The `mapMutations` helper maps the `incrementCounter` mutation to the component.

With the fundamental building blocks of game development now in our grasp, let's embark on the exciting journey of assembling these individual components into a cohesive and fully functional game.

Sample code

1.**main.js**

```
import Vue from "vue";
import App from "./App.vue";
import store from "./store";
import "./assets/css/global.css";
new Vue({
 el: '#app',
 store,
 render: (h) => h(App),
});
```

2.**App.vue**

```
<template>
 <div id="app">
   <Game />
 </div>
</template>
<script>
import Game from "./components/Game.vue";
export default {
 name: "App",
 components: {
   Game,
 },
};
</script>
<style>
#app {
 font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
 -webkit-font-smoothing: antialiased;
 -moz-osx-font-smoothing: grayscale;
 text-align: center;
 color: #2c3e50;
 height: 100vh;
 width: 100vw;
 display: flex;
 justify-content: center;
 align-items: center;
}
</style>
```

3**.Store**

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
export const store = new Vuex.Store({
 strict: true,
 state: {
   gameConfig: {
     GAME_SPEED: 5,
     GAME_LEVEL: 1,
   },
 },
 getters: {
   getGameConfig: (state) => {
     return state.gameConfig;
   },
 },
 mutations: {
   SET_LEVEL(state, payload) {
     switch (payload) {
       case 1:
         state.gameConfig = {
           GAME_SPEED: 5,
           GAME_LEVEL: 1,
         };
         break;
       case 2:
         state.gameConfig = {
           GAME_SPEED: 10,
           GAME_LEVEL: 2,
         };
         break;
       case 3:
         state.gameConfig = {
           GAME_SPEED: 20,
           GAME_LEVEL: 3,
         };
         break;
        default:
          break;
      }
    },
  },
  actions: {
    updateGameLevel({ commit }, payload) {
      commit("SET_LEVEL", payload);
    },
  },
});
export default store;
```

4. **Button Components**

Template

```
<template>
  <div class="page-container">
    <div class="page-wrapper">
      <div class="game">
        <div class="wrap-alert-message">
          <div class="alert" v-if="GAME_OVER">Game is over</div>
        </div>
        <div class="environment-container" v-if="!GAME_OVER">
          <div class="wrap-clouds">
            <div class="cloud cloud1">
              <img src="./../assets/images/cloud1.svg" alt="character"
                width="104" />
            </div>
```

```
   <div class="cloud cloud2">
    <img src="./../assets/images/cloud2.svg" alt="character"
     width="104"/>
   </div>
   <div class="cloud cloud3">
    <img src="./../assets/images/cloud3.svg" alt="character"
     width="104"  />
   </div>
   <div class="cloud cloud4">
    <img src="./../assets/images/cloud3.svg"
     alt="character" width="104"/>
   </div>
  </div>
 </div>
 <div v-if="!GAME_OVER" class="character" :style="characterStyle"></div>
 <div v-if="!GAME_OVER" class="obstacle" :style="obstacleStyle"></div>
</div>
<div id="config-wrapper">
 <div class="config-row">
  <Button type="primary" v-if="GAME_OVER" @click="restartGame"
   >Play</Button>
  <Button type="primary" v-else :disabled="true" @click="restartGame"
   >Play</Button>
 </div>
 <div class="config-row">Score: {{ SCORE }}</div>
 <div class="config-row">
  <div>Game level: {{ getGameConfig.GAME_LEVEL }}</div>
```

```
    </div>
    <div class="config-row">
      <div>Change level</div>
      <div class="game-levels">
        <Button
          type="secondary"
          :disabled="getGameConfig.GAME_LEVEL === 1 || !GAME_OVER"
          @click="updateLevel(1)" >1</Button>
        <Button type="secondary" :disabled="getGameConfig.GAME_LEVEL === 2 || !GAME_OVER"
          @click="updateLevel(2)"
          >2</Button
        >
        <Button
          type="secondary"
          :disabled="getGameConfig.GAME_LEVEL === 3 || !GAME_OVER"
          @click="updateLevel(3)"
          >3</Button
        >
      </div>
    </div>
    <div class="config-row" v-if="GAME_OVER && GAME_RESTARTED">
      <div class="alert">You need to restart the game</div>
    </div>
  </div>
</div>
<div class="footer">To jump or start, press <b>space</b> button</div>
```

```
 </div>
</template>
```

Scripts

```
<script>
// IMPORT STORE
import { mapGetters, mapActions } from "vuex";
// IMPORT REUSABLE BUTTON COMPONENT
import Button from "./button";
export default {
 name: "GameWrapper",
 components: {
  Button,
 },
 data() {
  return {
    // INITIAL POSITION OF MAIN CHARACTER
    characterPosition: {
     x: 0,
     y: 0,
    },
    // INITIAL POSITION OF OBSTACLE
    obstaclePosition: {
     x: window.innerWidth,
     y: 0,
    },
    GAME_INTERVAL: null,
    GAME_OVER: false,
    IS_JUMPING: false,
    SPEED: 5,
    SCORE: 0,
    GAME_LEVEL: 0,
    GAME_RESTARTED: false,
    TIME_PLAYED: 0,
    START_TIME: 0,
    MINUTES_ELAPSED: 0,
    autoplay: false,
  };
 },
 mounted() {
  // WHEN PAGE IS REFRESHED, PLAY THE GAME
  this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
  // CHECK IF PLAYER PRESSES SPACE
  window.addEventListener("keypress", (e) => {
    if (parseInt(e.charCode) === 32 && !this.GAME_OVER) {
     this.jump();
    } else {
     this.restartGame();
    }
  });
 },
 beforeDestroy() {
  cancelAnimationFrame(this.GAME_INTERVAL);
  window.removeEventListener("keypress", this.jump);
 },
 methods: {
  // RESTART THE GAME
  restartGame() {
    this.GAME_RESTARTED = false;
    this.characterPosition = {
     x: 0,
     y: 0,
    };
    this.obstaclePosition = {
     x: window.innerWidth,
     y: 0,
    };
    this.SPEED = 0;
    this.GAME_OVER = false;
    this.gameLoop();
    this.playSound();
  },
  // GAME LOOP: IF GAME IS ON, ITERATES THROUGH THE GAME
  gameLoop() {
    if (!this.GAME_OVER) {
     this.updateObstaclePosition();
     this.checkCollision();
     this.GAME_INTERVAL = requestAnimationFrame(this.gameLoop);
    }
  },
  // ANIMATE OBSTACLE: FROM RIGHT TO LEFT
  updateObstaclePosition() {
    this.obstaclePosition.x -= this.getGameConfig.GAME_SPEED || this.SPEED;
    // IF OBSTACLE REACHES LEFT SIDE, RESET IT
    if (this.obstaclePosition.x < -50) {
     this.obstaclePosition.x = window.innerWidth;
    }
  },
  // CHECK COLLISION
  checkCollision() {
    const characterRect = {
     x: this.characterPosition.x,
     y: this.characterPosition.y,
     width: 40,
     height: 80,
    };
    const obstacleRect = {
     x: this.obstaclePosition.x,
     y: this.obstaclePosition.y,
     width: 30,
     height: 60,
    };
    // IF CHARACTER TOUCHES OBSTACLE, END THE GAME
    if (this.isCollision(characterRect, obstacleRect)) {
     this.GAME_OVER = true;
     this.GAME_RESTARTED = true;
     this.SCORE++;
    } else {
     this.SCORE++;
    }
  },
  // CHECK IF CHARACTER TOUCHES OBSTACLE
  isCollision(character, obs) {
    return (
     character.x < obs.x &&
     character.x + character.width + 20 > obs.x &&
     !this.IS_JUMPING
    );
  },
  // CHECK IF KEY ON KEYBOARD IS PRESSED
  isKeyDown(key) {
    return (
     document.activeElement === document.body &&
     document.activeElement.addEventListener &&
     document.activeElement.addEventListener(
       "keydown",
       (event) => {
         if (event.key === key) {
           return true;
         }
       },
       false
     )
    );
  },
  // IF SPACE BUTTON IS PRESSED, CHARACTER SHOULD JUMP TO AVOID
```

```
COLLISION
  jump() {
    this.characterPosition.y = 70;
    this.IS_JUMPING = true;
    this.playSound();
    setTimeout(() => {
      this.IS_JUMPING = false;
      this.characterPosition.y = 0;
    }, this.jumpHangTime());
  },
  // SET TIME(in ms) THE CHARACTER HANGS IN THE AIR
  jumpHangTime() {
    switch (this.getGameConfig.GAME_LEVEL) {
      case 1:
        return 600;
      case 2:
        return 400;
      case 3:
        return 200;
      default:
        break;
    }
  },
  // PLAY SOUND
  playSound() {
    const audio = new Audio(require("./../assets/sounds/audio.ogg"));
    audio.volume = 0.5;
```

```
     audio.play();
   },
   updateLevel(level) {
     this.updateGameLevel(level);
   },
   ...mapActions(["updateGameLevel"]),
 },
 computed: {
   ...mapGetters({
     getGameConfig: "getGameConfig",
   }),

   obstacleStyle() {
     return {
       left: `${this.obstaclePosition.x}px`,
       bottom: `${this.obstaclePosition.y}px`,
     };
   },
   // MAKE CHARACTER JUMP WHEN SPACEBAR IS PRESSED
   characterStyle() {
     return {
       bottom: `${this.characterPosition.y}px`,
     };
   },
 },
};
</script>
```

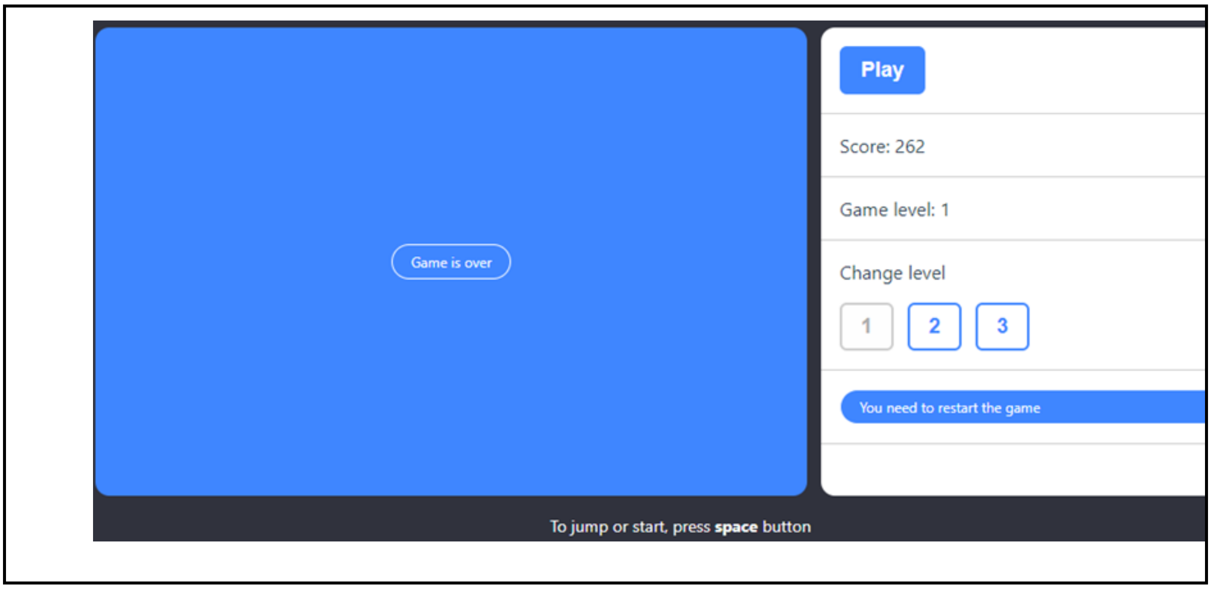
SCSS Styles

```
<style lang="scss">
.page-container {
 width: 100%;
 position: relative;
 .page-wrapper {
   display: flex;
   flex-direction: row;
   margin: 0 auto;
   width: 100%;
   max-width: 1000px;
   gap: 12px;
   .game {
     display: flex;
     width: 100%;
     height: 100vh;
     background: #3f86ff;
     position: relative;
     width: 800px;
     height: 400px;
     overflow: hidden;
     border-radius: 12px;
     .character,
     .obstacle {
       background-size: cover;
       position: absolute;
       bottom: 0;
```

```
  }
  .character {
    width: 40px;
    height: 80px;
    left: 50px;
    background-image: url("./../assets/images/person.gif");
    background-position: center;
  }
  .obstacle {
    width: 30px;
    height: 60px;
    background-image: url("./../assets/images/obstacle1.svg");
  }
}
#config-wrapper {
  width: 500px;
  background: #ffffff;
  border-radius: 12px;
  padding: 12;
  text-align: left;
  .config-row {
    padding: 16px;
    border-bottom: 1px solid #cdcdcd;
  }
  .game-levels {
    padding: 16px 0 0 0;
    display: flex;
```

```
      flex-direction: row;
      gap: 12px;
    }
  }
}
.environment-container {
  position: relative;
  top: 0;
  left: 0;
  width: 100%;
  height: 240px;
  .wrap-clouds {
    animation: clouse-animation 5s infinite linear;
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100px;
    @keyframes clouse-animation {
      0% {
        transform: translateX(100%);
      }
      100% {
        transform: translateX(-100%);
      }
    }
    .cloud {
      position: absolute;
      bottom: 0;
      left: 0;
      img {
        width: 50px;
      }
    }
    .cloud1 {
      top: 20px;
      left: 50px;
    }
    .cloud2 {
      top: 120px;
      left: 250px;
    }

    .cloud3 {
      top: 10px;
      left: 440px;
    }
    .cloud4 {
      top: 80px;
      left: -80px;
    }
  }
}
.wrap-alert-message {
   position: absolute;
   width: 100%;
   height: 100%;
   display: flex;
   align-items: center;
   justify-content: center;
 }
 .alert {
   padding: 6px 16px;
   border-radius: 16px;
   background: #3f86ff;
   border: 1px solid #ffffff;
   color: #ffffff;
   font-size: 12px;
 }
 .footer {
   font-size: 14px;
   color: #ffffff;
   padding: 16px 0;
 }
}
</style>
```

Game interface



Game is over

Score: 262

Game level: 1

You need to restart the game

To jump or start, press **space** button

Points to Remember

- To develop a game settings page/section in Vue.js, you can follow these general steps:

  1. Set up a new Vue.js project
  2. Define setting section
  3. Declare and bind variable

- Setting up game conditions is essential for defining the rules, constraints, and triggers that govern gameplay mechanics and outcomes. Here are the steps to set up game conditions effectively:

  1. Set the conditions
  2. Set loops and intervals

- To set up random mechanisms for creating diversity in a game using Vue.js, you can follow these steps:

  1. Generate Random Numbers
  2. Use Computed Properties
  3. Trigger Random Events

- To store data in state management effectively, you can follow these general steps:

  1. Install Vuex
  2. Create a store

3. Connect the Store to your Vue application
4. Access and modify the data in your components

Application of learning 4.2.

XCT game company is a game development company located in Huye district, that company used to develop different games that are played online for entrainment purpose due to many tasks that they have, they have hired you as game developer responsible for the jump game where you have to do the followings tasks:

- ✓ Develop a game settings page/section in Vue.js
- ✓ Manipulate events effectively in a game
- ✓ Set up game conditions effectively
- ✓ To set up random mechanisms for creating diversity in a game
- ✓ Creating and displaying alert messages
- ✓ To store data in state management

The company for support has provided all tools, materials and equipment.

Indicative content 4.3: Deploy Game Project on Netlify

Duration: 15 hrs

Practical Activity 4.3.1: Create deploy account on Netlify

Task:

1. Read key reading 4.3.1 and ask clarification where necessary

2. You are requested to go to the computer lab to create deployment account. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to create deployment account.

5. Referring to the steps provided on task 3, create deployment account.

6. Present your work to the trainer and whole class

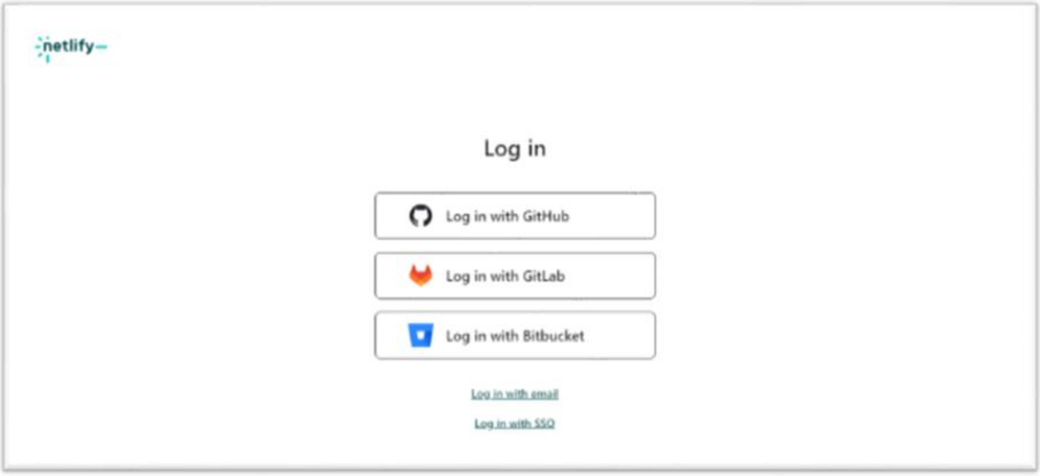
Key readings 4.3.1.: Create deployment account on Netlify

Creating a deployment account involves setting up an environment where you can deploy your application or project for public access.

Here are the general steps to create a deployment account:

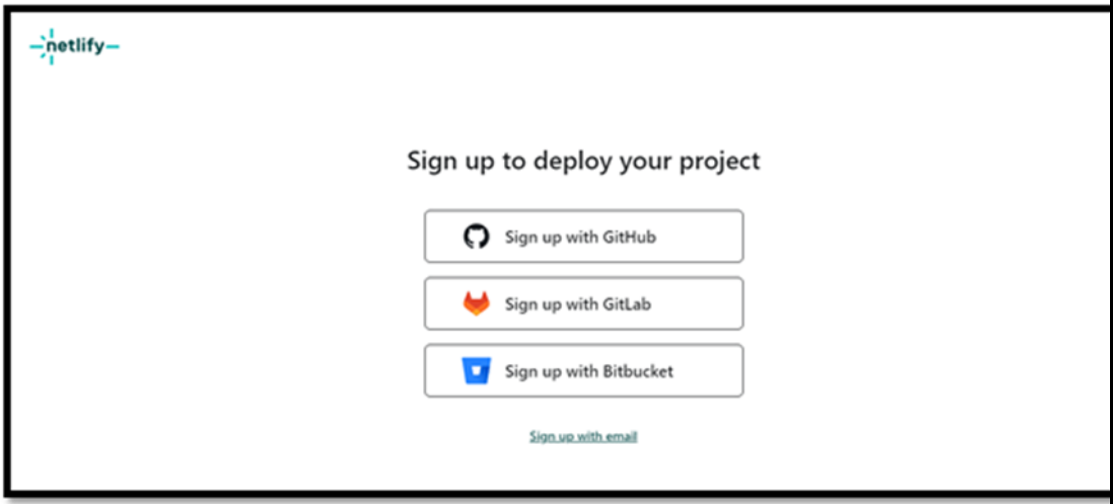
Step 1: Visit the Netlify Website

Open your web browser and go to the Netlify website at https://www.netlify.com/.



Step 2: Sign Up

Locate the "Sign Up" button on the Netlify homepage and click on it. This will take you to the sign-up page.



Step 3: Sign Up with Email:

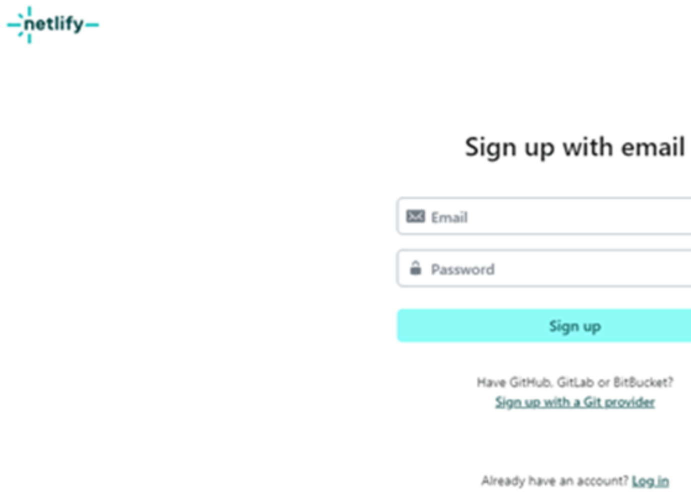
You can sign up for Netlify using your email address. Enter your email in the provided field

Step 4: Create a Password:

Choose a secure password for your Netlify account. Make sure it meets the password requirements, usually a combination of uppercase and lowercase letters, numbers, and special characters.

Step 5: Create an Account:

After entering your email and password, click on the "Create Account" or a similar button.



Sign up with email

Have GitHub, GitLab or BitBucket?
Sign up with a Git provider

Already have an account? Log in

Step 6: Verification

Netlify may send you a verification email. Check your email inbox for a message from Netlify and follow the instructions to verify your email address.

Step 7: Login

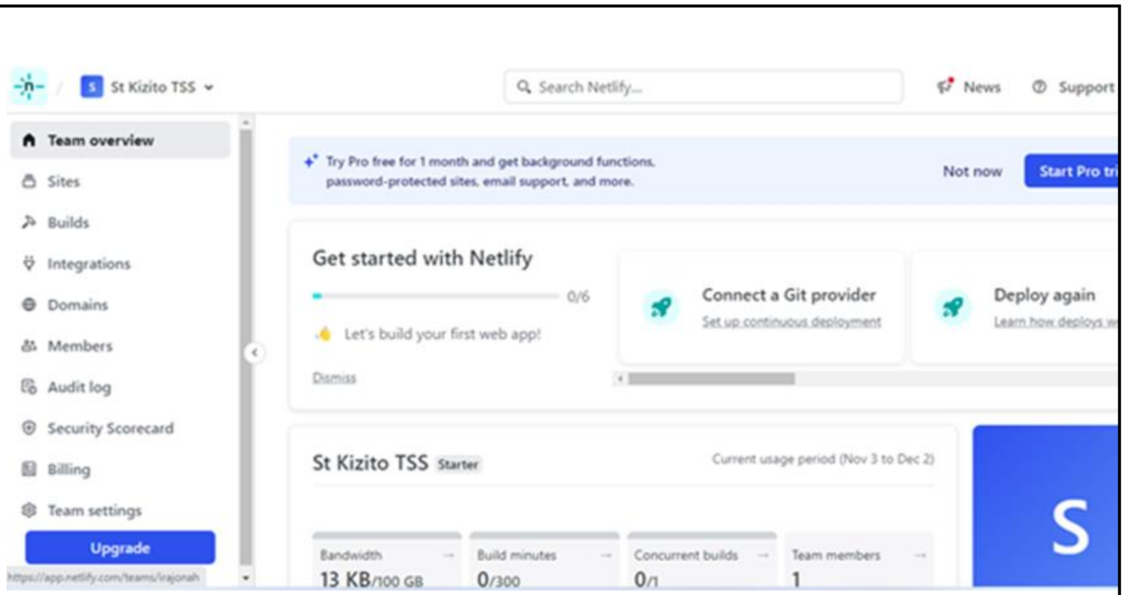
Once your email is verified, return to the Netlify website and log in with your email and password.

Step 8: Set Up Your Account

Netlify may guide you through a setup process where you can choose a plan (free or paid), set up your team, and configure other account settings.

Step 9: Dashboard

After completing the setup, you'll be directed to your Netlify dashboard. Here, you can start creating and managing your web projects. Now you can explore Netlify's features, deploy websites, and take advantage of its hosting and continuous integration capabilities.



Try Pro free for 1 month and get background functions, password-protected sites, email support, and more.

Practical Activity 4.3.2: Connecting the project with Git repository

Task:

1. Read key reading 4.3.2 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to connect the project with the git repository. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to connect the project with git repository.

5. Referring to the steps provided on task 3, connect the project with git repository.

6. Present your work to the trainer and whole class

Key readings 4.3.2.: Connecting the project with Git repository

To connect a project with a Git repository, you'll need to follow a series of steps. assuming that you have Git installed on your computer and that you have a project you want to connect to a repository.

Here's what you need to do:

1. **Open a Terminal or Command Prompt**

A Using the Start Menu: Click on the Start Menu icon (Windows icon) in the bottom-left corner of your screen. In the search bar, type cmd or Command Prompt. Press Enter or click on the Command Prompt or Windows Terminal icon in the search results.

B. Using the Run Dialog: Press Win + R to open the Run dialog. Type cmd and press Enter. This will open the Command Prompt.

C.Using File Explorer: Open File Explorer. Navigate to the directory where you want to open the command prompt. In the address bar, type cmd and press Enter.

2. Navigate to Your Project Directory

Use the `cd` command to change directories to your project's root directory.

```
cd path/to/your/project
```

3. Initialize a New Git Repository

Initialize a new Git repository in your project folder by running:

```
git init
```

This command creates a new `.git` directory in your project, which will hold all the version history.

4. **Add Project Files to the Repository**

Before committing your files, you need to add them to the staging area. To add all files in the project directory, use:

```
git add.
```

If you want to add specific files, replace `.` with the file names.

5. **Commit the Files**

Save the changes in the repository with a commit. You should include a message describing what you are committing:

```
git commit -m "Initial commit"
```

6. Link to a Remote Repository

If you haven't already, create a repository on a Git hosting service like GitHub, GitLab, or Bitbucket. Once created, link your local repository to the remote one using the following command, replacing `your-remote-repository-url` with the URL provided by the hosting service:

```
git remote add origin your-remote-repository-url
```

7. Push Your Commit to the Remote Repository

To push your commit to the remote repository, use:

```
git push -u origin master
```

If you're using a branch other than `master` (e.g., `main`), replace `master` with the name of your branch.

8. Verify Connection

Check that the remote repository is correctly set by listing the configured remote

```
git remote -v
```

After these steps, your project should be connected to the Git repository. Remember, if you make further changes to your project files, you'll need to `git add`, `git commit`, and `git push` those changes to update the remote repository.

Practical Activity 4.3.3: Configure deployment commands

Task:

1. Read key reading 4.3.3 and ask clarification where necessary

2. Referring to the previous practical activities, you are requested to go to the computer lab to configure deployment commands. This task should be done individually.

3. Apply safety precautions.

4. Present out the steps to configure deployment commands.

5. Referring to the steps provided on task 3, configure deployment commands.

6. Present your work to the trainer and whole class

Key readings 4.3.3.: Configure deployment commands

To configure deployment commands in Netlify, you need to set up a Netlify account, connect your Git repository, and specify the build settings for your project.

Here’s how you can do it:

1.Create a Netlify Account

If you don't already have a Netlify account, go to [Netlify's website] (https://www.netlify.com/) and sign up for a new account.

2. Add a New Site

Once logged in to your Netlify dashboard, click on the "New site from Git" button to start the process of connecting your Git repository.

3. Connect Your Git Repository

Choose the Git provider where your repository is hosted (GitHub, GitLab, or Bitbucket) and authenticate if necessary. Then, select the repository you want to deploy.

4. Configure Build Settings

Netlify will ask you to configure your build settings. This is where you specify the deployment commands and the directory that Netlify should publish. Here are the common fields you need to fill out:

Build command

This is the command that Netlify will run to build your project. For example, if you're using a static site generator like Jekyll, it might be `jekyll build`. For a React app created with Create React App, it would be `npm run build` or `yarn build`.

Publish directory

This is the directory that Netlify should deploy. It's the output of your build command. For example, with Create React App, the default is `build/`.

Environment Variables (if necessary)

If your build requires any environment variables, you can set them here.

5. Deploy Site

After configuring your build settings, click the "Deploy site" button. Netlify will start the first deployment of your site by running the build command you specified and deploying the contents of the publish directory.

6.Continuous Deployment

Netlify offers continuous deployment. Once you've connected your Git repository and configured your build settings, Netlify will automatically deploy your latest changes every time you push to your Git repository.

7.Additional Settings

If you need to change your build settings later on, you can go to your site's settings on Netlify, navigate to the "Build & deploy" section, and adjust the "Build settings".

8.Branch Deploys

You can also configure Netlify to deploy previews for feature branches. This can be done in the "Build & deploy" settings where you can specify which branches to deploy.

9.Custom Domains

After deploying your site, you can set up a custom domain in the "Domain management" section of your site settings.

10.Trigger Deploy

If you need to trigger a new deploy manually, you can do so from the "Deploys" tab by clicking "Trigger deploy".

By following these steps, you can configure deployment commands in Netlify and enjoy the benefits of continuous deployment for your project. Remember to check your build configuration and adjust it according to the specific requirements of your project.

Practical Activity 4.3.4: Create and merge (Pull Request) PR on Github

Task:

1. Read key reading 4.3.4 and ask clarification where necessary
2. Referring to the previous practical activities, you are requested to go to the computer lab to create and merge (Pull Request) PR on GitHub. This task should be done individually.
3. Apply safety precautions.
4. Present out the steps to create and merge (Pull Request) PR on GitHub.
5. Referring to the steps provided on task 3, Create and merge (Pull Request) PR on GitHub.

6. Present your work to the trainer and whole class

Key readings 4.3.4.: Create and merge (Pull Request) PR on Github

Creating and merging a Pull Request (PR) on GitHub involves a few steps. Here's a step-by-step guide to help you through the process:

Create a Pull Request

1.Fork the Repository (if it's not your own)

If you want to contribute to someone else's repository, you should fork it first by clicking the "Fork" button on the top right of the repository page.

2. Clone the Repository

Clone the repository to your local machine using the `git clone` command with your repository URL

```
git clone https://github.com/your-username/repository-name.git
```

3. Create a New Branch

Navigate to the repository directory and create a new branch for your changes:

```
cd repository-name
git checkout -b your-branch-name
```

4. Make Your Changes

Make the necessary changes or additions to the codebase. Be sure to follow any contributing guidelines provided by the repository owner

5.Commit Your Changes

Stage your changes and commit them with a meaningful commit message:

```
git add.
git commit -m "Add a descriptive message explaining your changes"
```

6.Push Your Changes

Push your branch and changes to your forked repository on GitHub:

```
git push origin your-branch-name
```

7. Create the Pull Request

Go to the original repository on GitHub. You should see a button to "Compare & pull request" for your branch. Click it.

Fill in the PR title and description, explaining the changes you've made. If the PR is related to an open issue, reference it in the description.

8.Submit the Pull Request

After reviewing your changes and ensuring that they're ready to be merged, click "Create pull request".

Merge a Pull Request

Once a pull request is created, the maintainers of the original repository will review your changes. If they approve the changes and everything is in order, they can merge the pull request.

1. Review the Pull Request

Reviewers will look at the changes, run tests, and possibly request changes or additional information.

2.Make Any Requested Changes

If the reviewers ask for changes, make them in your branch and push the updates. The pull request will automatically update with the new commits.

3.Merge the Pull Request

Once the pull request is approved by the maintainers, they can merge it into the main codebase. They will click the "Merge pull request" button, often choosing to squash the commits or rebase them as needed.

4.Pull Request Closed

After merging, the pull request will be marked as close

5.Delete Your Branch (optional)

After the pull request is merged, you can safely delete your branch, both locally and on GitHub. GitHub provides a button to "Delete branch" right on the merged pull request page.

Remember, it's important to keep your branch up-to-date with the base branch (often `main` or `master`) to minimize merge conflicts. You can do this by pulling the latest changes from the original repository and merging them into your branch before submitting the pull request.

Practical Activity 4.3.5: Testing provided Netlify domain

Task:

1: Read the key reading 4.3.5

2: As a Software developer, you are asked to go into the computer lab to perform a domain testing as provided to you by netlify.

3: Present your work to the trainer and your classmates

Key readings 4.3.5.: Testing provided Netlify domain

To test your site using the provided Netlify domain, follow these steps:

1. **Complete Deployment**

Ensure that your site has been deployed successfully on Netlify. You can check the status of your deployment in the "Deploys" tab of your Netlify dashboard.

2.**Find Your Netlify Subdomain**

Once the deployment is successful, Netlify will provide you with a default subdomain for your site. You can find this under the "Site settings" or directly on your site's overview page in the Netlify dashboard.

3.**Visit the URL**

Open your web browser and go to the provided Netlify subdomain URL (e.g., `https://your-project-name.netlify.app`). This will take you to the live version of your site hosted on Netlify.

4.**Test Site Functionality**

Navigate through the site to ensure all pages and links are working correctly.

Test any forms, search functionality, or interactive elements to ensure they are operating as expected.

Check the site on different browsers and devices to ensure compatibility and responsiveness.

If your site uses secure connections (HTTPS), verify that there are no security warnings and that SSL is working correctly.

5.Check for Errors

Use the developer tools in your browser (usually accessible by pressing `F12` or right-clicking the page and selecting "Inspect") to check for any console errors or network issues that could affect the functionality or performance of your site.

6.Performance Testing

Consider using online tools like Google Page Speed Insights, Lighthouse, or GTmetrix to analyze the performance and loading times of your site.

7.Feedback and Iteration

If you find any issues, make the necessary changes to your project, commit them to your repository, and push the updates. Netlify will automatically create a new deploy with these changes, and you can test again once the deployment is complete.

Points to Remember

- Creating a deployment account involves setting up an environment where you can deploy your application or project for public access. Here are the general steps to create a deployment account:

  Step 1: Visit the Netlify Website

  Step 2: Sign Up

  Step 3: Sign Up with Email

  Step 4: Create a Password

  Step 5: Create an Account

  Step 6: Verification

  Step 7: Login

  Step 8: Set Up Your Account

  Step 9: Dashboard

- Creating and merging a Pull Request (PR) on GitHub involves a few steps. Here's a step-by-step guide to help you through the process:

  1. Create a Pull Request

  2. Merge a Pull Request

- To test your site using the provided Netlify domain, follow these steps:

    1. Complete Deployment
    2. Find Your Netlify Subdomain
    3. Visit the URL
    4. Test Site Functionality
    5. Check for Errors
    6. Performance Testing
    7. Feedback and Iteration

Application of learning 4.3.

The development team have developed a jumping game using Vue.js. The game includes a character that can jump when a specific key is pressed. The character moves up, experience gravity, and fall back down. Additionally, there should be a ground level, and the character should not be able to jump indefinitely. Implement a scoring system that increments as the character successfully jumps over obstacles or reaches higher platforms.

That developed game need to be deployed on Netlify.

As game developer, you have been hired to perform that task.

Learning outcome 4 end assessment

Written assessment

1. Which platform is commonly used for deploying static websites?

A) Netlify

B) AWS

C) Heroku

D) Docker

2. What does a domain name uniquely identify?

A) A physical server location

B) A website's IP address

C) A website's brand and identity

D) A website's security certificate

3. What does SASS stand for?

A) Scalable Application Style Sheets

B) Syntactically Awesome Style Sheets

C) Structured Application Style Sheets

D) Scripted Awesome Style Sheets

4. Which HTML5 element is commonly used for dynamic, scriptable rendering of graphics?

A) <div>

B) <svg>

C) <canvas>

D) <img>

5. What advantage does SVG offer over traditional image formats like JPEG or PNG?

A) It supports animations and interactivity

B) It has smaller file sizes

C) It is easier to embed in HTML

D) All of the above

6. What aspect of game design focuses on the visual representation of the game world?

A) Game mechanics

B) Game interface design

C) Game physics

D) Game narrative

7. Which JavaScript method is used to get the rendering context and drawing functions for a <canvas> element?

A) getContext()

B) renderContext()

C) drawContext()

D) canvasContext()

8. Which JavaScript function is used to draw a filled rectangle on a <canvas> element?

A) fillRect()

B) drawRect()

C) rectFill()

D) drawFilledRectangle()

9. How does SASS extend CSS?

A) By adding variables, nesting, and mixins

B) By converting CSS to JavaScript

C) By optimizing CSS for faster loading

D) By embedding JavaScript directly into stylesheets

10. What does the HUD in a game typically display?

A) Player controls

B) Game settings

C) Character health, score, and resources

D) Background story

11. Fill the empty spaces with the correct keywords in the following statements

a) Containers for game stats typically include ________ and ________.

b) SetInterval() in JavaScript is used to ________.

c) To deploy a game project on Netlify, you need to connect your project to a ________ repository.

Practical assessment

The development team have developed a jumping game using Vue.js. The game includes a character that can jump when a specific key is pressed. The character moves up, experience gravity, and fall back down. Additionally, there should be a ground level, and the character should not be able to jump indefinitely. Implement a scoring system that increments as the character successfully jumps over obstacles or reaches higher platforms.

That developed game need to be deployed on netlify

As game developer you have been hired to perform that task.

References

Salen, K., & Zimmerman, E. (2003). Rules of play: Game design fundamentals. *MIT Press.* https://www.amazon.com/Rules-Play-Game-Design-Fundamentals/dp/0262240459

Game Developer. (n.d.). *Blogs*. Game Developer.
https://www.gamedeveloper.com/blogs#close-modal

Johnson, A. (2020). Integrating the Vue Framework in Game Development: Best Practices and Case Studies. Game Development Quarterly, volume(issue), page numbers.

Smith, J. (2021). Setting Up the Development Environment for Game Development. Journal of Game Development, volume(issue), page numbers. URL

Williams, R. (2021). Game Planning and Design Strategies: A Comprehensive Guide. International Journal of Game Design and Development, volume(issue), page numbers.

RTB | RWANDA TVET BOARD

October 2024