



Vue.js

# VUE.JS FRAMEWORK

## LEVEL 3 SOD

Prepared by **ISHIMWE Gilbert**

*[www.ishprodev.rw](http://www.ishprodev.rw)*

## Learning UNIT 1: Set Up Environment

### Topic one: Description of key concepts

**1.Back end:** The back end refers to parts of a computer application or a program's code that allow it to operate and that cannot be accessed by a user. Most data and operating syntax are stored and accessed in the back end of a computer system. Typically, the code is comprised of one or more programming languages. The back end is also called the data access layer of software or hardware and includes any functionality that needs to be accessed and navigated to by digital means.

**2.Front end:** Front end development is programming which focuses on the visual elements of a website or app that a user will interact with (the client side).

### Purpose of front end and back end

System architectures are broken down into front end and back end components for a variety of purposes. The most common is in software and web development to break down projects in terms of required skills. The front end aspect of a project is usually handled by professionals such as web designers while the back end is handled by engineers and developers.

### Examples of front end and back end

Concepts and components that focus on the front end of a system include:

- Design and markup languages like [HTML](#), [CSS](#) and [JavaScript](#).
- Search engine optimization (SEO).
- [Usability](#) and accessibility testing.
- Graphic design and image editing tools.
- Web performance and browser [compatibility](#).

Inversely, those that focus on the back end of a system include:

- Programming and scripting languages like PHP, [Python](#) and [C#](#).
- Automated testing frameworks.
- Network scalability and [availability](#).
- Database management and [data transformation](#).
- Cybersecurity and data backup practices.

**3. Single Page Application :**An SPA (Single-page application) is a web app implementation that loads only a single web document, and then updates the body content of that single document via JavaScript APIs such as XMLHttpRequest and Fetch when different content is to be shown

**4. NodeJs & NPM** Node.js (Node) is **an open source, cross-platform runtime environment for executing JavaScript code**. Node is used extensively for server-side programming, making it possible for developers to use JavaScript for client-side and server-side code without needing to learn an additional language.

Node.js is a **JavaScript runtime environment that achieves low latency and high throughput by taking a “non-blocking” approach to serving requests**. In other words, Node.js wastes no time or resources on waiting for I/O requests to return

**node is a framework that can run JavaScript code on your machine while npm is a package manager**. Using npm we can install and remove javascript packages also known as node modules.

## Benefits of Node.js

Below are various benefits offered by Node.js:

### 1. Robust Technology Stack

After JavaScript, Node.js has become a robust stand-alone name in the programming field. It has experienced more than 368,985,988 downloads and over 750 contributors and is still growing.

### 2. Perfect for Micro services

Node.js is a lightweight tool, making it suitable for micro services architecture. It allows you to break application logic into smaller modules rather than building a single and large monolithic core.

### 3. Rich Ecosystem

Node.js comes with npm, which provides a marketplace for open-source JavaScript tools and helps advance this technology.

### 4. Strong Corporate Support

After Joyent supported the development of Node.js, it created the Node.js foundation in 2015. It helps in adopting and accelerating Node.js development.

## Drawbacks of Node.js

### 1. Performance Bottlenecks

When using Node.js, you may face some drawbacks, like the inability to process CPU-bound tasks efficiently. To understand the root cause of the problem, you need to have a bit of context about the situation.

### 2. Immature Tools

However, Node.js consists of stable and mature modules, but numerous tools are available in the npm registry that is of poor quality and stored without being tested.

NPM or **N**ode **P**ackage **M**anager is an open-source repository of tools engineers use to develop applications and websites.

NPM is two things:

1. A repository for publishing open-source projects. Simplified version: **a digital storage and retrieval facility**
2. A command-line interface (CLI) for interacting with the repository. Simplified version: **a tool to communicate with the storage facility**

## How to use NPM

Here are some common NPM commands and what they do:

- **npm init:** Creates a package.json file for your project. If you're building an application from scratch, npm init will be one of the first commands you use to include key project information. NPM will automatically update your package.json file whenever you install or remove packages.
- **npm install:** Installs all of the project dependencies in a package.json file.
- **npm install <package-name>:** Installs a specific package from the NPM registry and saves it to your node\_modules folder. For example, *npm install @uxpin/merge-cli* will install the Merge CLI.
- **npm install <package-name> --save:** Installs an NPM package and adds it to the dependencies in your package.json file.
- **npm install <package-name> --save-dev:** installs an NPM package and adds it to the devDependencies
- **npm uninstall <package-name>:** Uninstalls a specific package from your project.
- **npm doctor:** Runs diagnostics on your npm installation to check if it has everything it needs to manage your packages.
- **npm update <package-name>:** Updates a specific package to the latest version.

**5. CLI:** A [command-line interface](#) (CLI) is a text interface developers use to interact with computer programs. This CLI allows you to execute commands to run background operations necessary for software development.

In the case of NPM, the CLI allows you to interact with the package registry. For example, engineers can use commands like *npm install* followed by the package name to install a specific package.

**6. Dependencies:** dependency is **additional code that a programmer wants to call**. Adding a dependency avoids repeating work already done: designing, writing, testing, debugging, and maintaining a specific unit of code.

**7. IDE:** An integrated development environment (IDE) is a software application that helps programmers develop software code efficiently. It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application. Just as writers use text editors and accountants use spreadsheets, software developers use IDEs to make their job easier.

## Introduction Vue.JS Framework

### What is Vue? #

Vue (pronounced /vju:/, like **view**) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex.

**Vue.JS** is an open source progressive JavaScript framework used to develop interactive web interfaces. It is one of the famous frameworks used to simplify web development.

## **THE FEATURES OF VUE JS**

Following are the features available with Vue.JS

### **Virtual DOM**

Vue.JS makes the use of virtual DOM, which is also used by other frameworks such as React, Ember, etc. The changes are not made to the DOM, instead a replica of the DOM is created which is present in the form of JavaScript data structures.

### **Data Binding**

The data binding feature helps manipulate or assign values to HTML attributes, change the style, assign classes with the help of binding directive called **v-bind** available with Vue.JS

### **Components**

Components are one of the important features of Vue.JS that helps create custom elements, which can be reused in HTML.

### **Event Handling**

**v-on** is the attribute added to the DOM elements to listen to the events in Vue.JS

### **Animation/Transition**

Vue.JS provides various ways to apply transition to HTML elements when they are added/updated or removed from the DOM.

### **Computed Properties**

This is one of the important features of Vue.JS. It helps to listen to the changes made to the UI elements and performs the necessary calculations. There is no need of additional coding for this.

### **Templates**

Vue.JS provides HTML-based templates that bind the DOM with the Vue instance data. Vue compiles the templates into virtual DOM Render functions. We can make use of the template of the render functions and to do so we have to replace the template with the render function.

### **Directives**

Vue.JS has built-in directives such as v-if, v-else, v-show, v-on, v-bind, and v-model, which are used to perform various actions on the frontend.

### **Watchers**

Watchers are applied to data that changes. For example, form input elements. Here, we don't have to add any additional events. Watcher takes care of handling any data changes making the code simple and fast.

## **Routing**

Navigation between pages is performed with the help of vue-router.

## **Lightweight**

Vue.JS script is very lightweight and the performance is also very fast.

## **Vue-CLI**

Vue.JS can be installed at the command line using the vue-cli command line interface. It helps to build and compile the project easily using vue-cli.

## **Vue Components**

Components in Vue are composed of three parts; a template (which is like HTML), styles and JavaScript. These can be split into multiple files or the same .vue file. For simplicity here these examples are combined into one file

Here is a minimal example:

```

<template>
  <span class="welcome">
    {{ message }}
  </span>
</template>

<script>
import Vue from 'vue';

export default Vue.extend({
  name: 'Welcome',
  data() {
    return {
      message: 'Hello World'
    }
  }
});
</script>

<style>
.welcome {
  color: blue;
}
</style>

```

Let's look at each of the three parts of a component, starting with the template.

## I.Template component of vue

Vue templates are designed to be similar to vanilla HTML with two main exceptions: directives and custom components. Templates are used in both pages and components and usually sit at the top of a .vue file.

## Custom Components

Custom components are simple to create and use. In fact, this whole section is about creating a custom component. A new .vue file must be made then imported into the page or component it needs to be displayed in. It then needs to be added to the component options object within the script tag (see that section) before it can be added to the template, as shown here.

```
<custom-component :custom-prop="message" />
```

For more information about custom components visit the [Vue Guide](#).

## Directives

Directives are special attributes that can be added to tags in templates. These provide functionality to the component or page, and always start with the v- prefix. While there are many directives available, the most useful 5 categories (in my opinion) are listed here:

- v-bind
- v-on
- v-model
- v-if; v-else-if; v-else
- v-for

### Data Binding: v-bind

v-bind is a directive that pipes a variable into a component and updates that component when the variable changes.

For example, an input element could take a value of counter. When counter is updated by other components, this input will automatically update.

```
<input type="text" v-bind:value="counter" />
```

The directive must prefix a property of the element that will be dynamically updated. Any property can be bound to data using v-bind.

As this is the most common directive a single colon can be used for brevity, as shown here.

```
<input type="text" :value="counter" />
```

### Event Handling: v-on

v-on is a directive that takes a function which is called when the specified event is fired. Like v-bind, the directive must prefix an event name. However, rather than taking a variable this takes an expression or method.

For example, this button will add 1 to the counter when clicked.

```
<button v-on:click="counter += 1">Add 1</button>
```

Like v-bind, this is a commonly used directive and has a shorthand - the @ symbol.

```
<button @click="counter += 1">Add 1</button>
```

v-on can take either an expression or a method name, as shown here. The method can be written with or without the curly brackets.

```
<button @click="increment()">Add 1</button>
```

## Parent-Child Communication



The method of communication between a parent and child component changes depending on the direction. According to best practice, props (and therefore v-bind should be used for downwards communication but events (and therefore v-on) for upwards.

To communicate upwards, emit events from the child component using the \$emit method.

```
export default Vue.extend({
  methods: {
    emitEvent(value) {
      this.$emit('click', value);
    }
  }
});
```

Within the parent component, an event can be consumed in the same way as is done for standard HTML components.

```
<child-component @click="clickHandler" />
```

Although it is possible, the Vue community considers it an anti-pattern to pass callbacks down to the child component via its props, as shown here.

```
<child-component :onclick="clickHandler" />
```

## Two Way Data Binding: v-model

While v-bind enables one-way data binding, Vue also supports two-way binding using the directive v-model.

For example, this input component will react to changes to the variable message but also will push updates to this variable when a user enters text.

```
<div id="app">
  <input v-model="message" />
</div>
```

As an aside, v-model is a shorthand directive that adds two directives under the hood. The example above could be rewritten using v-on and v-bind as shown here.

```
<div id="app">
  <input :value="message" @input="message = $event.target.value" />
</div>
```

## Conditional Display: v-if

v-if is used to conditionally display elements.

For example, this text will only display if the variable visible is true.

```
<div>
```

```
<span v-if="visible">Hello World</span>
</div>
```

Much like standard programming logic, v-else-if and v-else can be used also. This next example will display one of three strings depending on the value of type.

```
<div>
  <span v-if="type === 'A'">Type A!</span>
  <span v-else-if="type === 'B'">Type B!</span>
  <span v-else>Unknown Type!</span>
</div>
```

## Loops: v-for

v-for is used to display multiple copies of similar elements, as you might expect from the name. This is useful for displaying lists and tables.

Here is an example of how this is used to display a list of names. Note that the v-bind directive is needed for the key attribute (:key) so that each generated element is distinct in the DOM.

```
<template>
  <div>
    <p v-for="name in names" :key="name">{{ name }}</p>
  </div>
</template>
```

```
<script>
export default {
  data() {
    return {
      names: [
        'Alice',
        'Bob',
        'Connor',
        'Doug'
      ]
    }
  }
};
</script>
```

## II.Script component of vue

The second part of a Vue component is the script, which can be either JavaScript or transpiled languages such as TypeScript. This code, contained within the <script> tag, defines the functionality of the component.

As you can see from this example, most of the logic is contained within the component options object; the object passed to Vue.extend().

```

<script>
import Vue from 'vue';

export default Vue.extend({
  name: 'Welcome',
  data() {
    return {
      message: 'Hello World'
    }
  }
});
</script>

```

There are many types of component options to choose from; the key ones are listed here.

## Data

The data option provides variables for use in the template.

```

export default Vue.extend({
  data() {
    return {
      counter: 0
    }
  }
});

```

This can be accessed in other component options by referencing this.counter. In the above sections we have already seen how to reference variables, either through double curly brackets or the directives v-model and v-bind.

## Props

Props are passed into components using XML attributes as is standard in HTML.

For example, this shows the variable counter being passed into the prop value of the component MyComponent.

```

<my-component :value="counter" />

```

Within MyComponent.vue, this can be achieved in the component options.

```

export default Vue.extend({
  props {
    value: { type: Number, default: 0 },
  },
});

```

As shown, each value requires meta data such as its data type, default and whether it is required or not. This prop can now be used in the same way as a variable in the template or accessed in other component options using `this.value`.

## Methods

Methods are a key way of enabling functionality in your component. They are defined within the component options and referenced in a similar way to variables.

This example shows an `increment()` method which increases the value of counter by 1.

```
export default Vue.extend({
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    increment() {
      this.counter++;
    }
  }
});
```

To use this in the template, reference `increment()` within a `v-on` directive, as shown in the above section.

```
<button @click="increment">+</button>
```

To learn more about event handling, see the [Vue docs](#).

## Computed

Computed values are similar to methods but differ in how they update. As you might expect, methods are run each time they are called and will recalculate their return value. However, this is not the case for computed values. These are recalculated whenever any of the data they depend on updates, and the resulting updated return value is pushed to any components consuming the computed values. Therefore, they are a powerful tool for manipulating input values but maintaining reactivity.

This example shows how to set up a computed value.

```
export default Vue.extend({
  props {
    value: { type: String, default: " " },
  },
  computed: {
    message() {
      return `[Message: ${this.value}]`;
    }
  }
});
```

This is consumed in the same way as variables are consumed. In this example, when the value prop updates the message computed value will also update causing the text on the screen to do likewise.

```
<span>{{ message }}</span>
```

## Components

Child components can be added as follows.

```
import ChildComponent from '../ChildComponent.vue';
```

```
export default Vue.extend({  
  components: {  
    ChildComponent,  
  },  
});
```

ChildComponent can be now added to the template in one of two ways depending on convention.

```
<child-component />  
<ChildComponent />
```

## III.Style component of vue

Styling within Vue is versatile and modern. Less, Sass and SCSS come built-in, along with support for scoped and modular CSS. As you might expect, multiple style tags can be written per component and styles can be imported from other files too. This allows you choice of a multi-file or single file approach.

CSS modules are easy to use. Simply add the “module” attribute to the style tag and reference the \$style variable within the template.

```
<template>  
  <span :class="$style.welcome">  
    Hello World!  
  </span>  
</template>
```

```
<style lang="scss" module>  
.welcome {  
  color: blue;  
}  
</style>
```

## Vue project installation

### Prerequisites

- [Install Node.js on Windows](#): This includes a version manager, package manager, and Visual Studio Code. The Node Package Manager (npm) is used to install Vue.js.

## Install Vue CLI

Vue CLI is a toolkit for working with Vue in your terminal / command line. It enables you to quickly scaffold a new project (vue create), prototype new ideas (vue serve), or manage projects using a graphical user interface (vue ui). Vue CLI is a globally installed npm package that handles some of the build complexities (like using Babel or Webpack) for you. *If you are not building a new single-page app, you may not need or want Vue CLI.*

To install Vue CLI, use npm. You must use the -g flag to globally install in order to upgrade (vue upgrade -next):

PowerShell

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

## Install NodeJs

**Node.js** is a run-time environment which includes everything you need to execute a program [written in JavaScript](#). It's used for running scripts on the server to render content before it is delivered to a web browser.

NPM stands for Node Package Manager, which is an application and repository for developing and sharing JavaScript code.

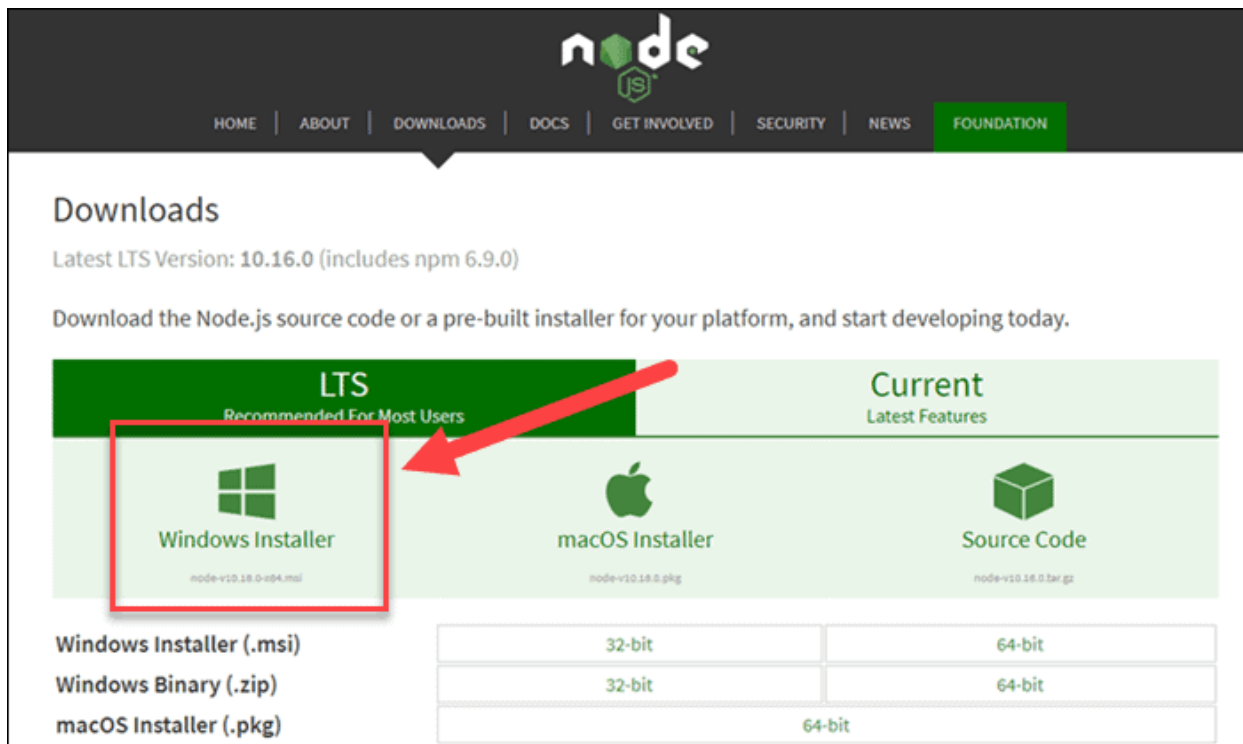
### Prerequisites

- A user account with administrator privileges (or the ability to download and install software)
- Access to the Windows command line (search > cmd > right-click > run as administrator) OR Windows PowerShell (Search > Powershell > right-click > run as administrator)

## How to Install Node.js and NPM on Windows

### Step 1: Download Node.js Installer

In a web browser, navigate to <https://nodejs.org/en/download/>. Click the **Windows Installer** button to download the latest default version. At the time this article was written, version 10.16.0-x64 was the latest version. The Node.js installer includes the NPM package manager.



## Step 2: Install Node.js and NPM from Browser

1. Once the installer finishes downloading, launch it. Open the **downloads** link in your browser and click the file. Or, browse to the location where you have saved the file and double-click it to launch.
2. The system will ask if you want to run the software – click **Run**.
3. You will be welcomed to the Node.js Setup Wizard – click **Next**.
4. On the next screen, review the license agreement. Click **Next** if you agree to the terms and install the software.
5. The installer will prompt you for the installation location. Leave the default location, unless you have a specific need to install it somewhere else – then click **Next**.
6. The wizard will let you select components to include or remove from the installation. Again, unless you have a specific need, accept the defaults by clicking **Next**.
7. Finally, click the **Install** button to run the installer. When it finishes, click **Finish**.

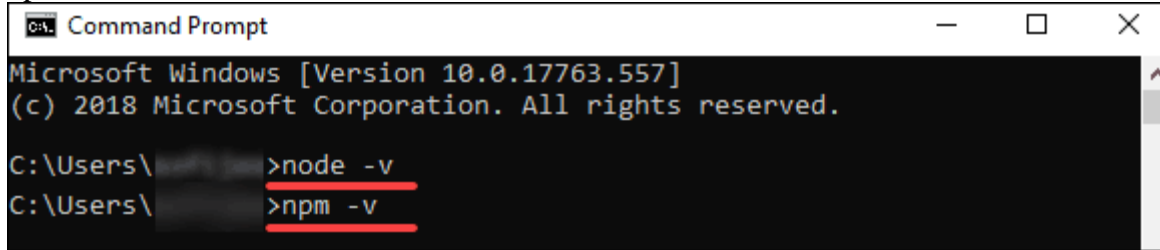
## Step 3: Verify Installation

Open a command prompt (or PowerShell), and enter the following:

```
node -v
```

The system should display the Node.js version installed on your system. You can do the same for NPM:

npm -v

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The window content shows the following text: "Microsoft Windows [Version 10.0.17763.557] (c) 2018 Microsoft Corporation. All rights reserved." followed by a prompt "C:\Users\[redacted]>". The user has entered "node -v" and "npm -v", both of which are underlined in red. The output of these commands is not visible in the screenshot.

```
Microsoft Windows [Version 10.0.17763.557]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\[redacted]>node -v
C:\Users\[redacted]>npm -v
```

### How to Update Node.js and NPM on Windows

The easiest way to [update Node.js](#) and NPM is to download the latest version of the software. On the Node.js download page, right below the **Windows Installer** link, it will display the latest version. You can compare this to the version you have installed.

To upgrade, download the installer and run it. The setup wizard will overwrite the old version, and replace it with the new version.

### How to Uninstall Node.js and NPM on Windows

You can uninstall Node.js from the Control Panel in Windows.

To do so:

1. Click the **Start** button > **Settings** (gear icon) > **Apps**.
2. Scroll down to find **Node.js** and click to highlight.
3. Select **Uninstall**. This launches a wizard to uninstall the software.

### Basic Node.js Usage

Node.js is a framework, which means that it doesn't work as a normal application. Instead, it interprets commands that you write. To test your new Node.js installation, create a **Hello World** script.

1. Start by launching a text editor of your choice.
2. Next, copy and paste the following into the text editor you've just opened:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```



3. Save the file, then exit. Open the PowerShell, and enter the following:

```
node \users\<your_username>\myprogram.js
```

It will look like nothing has happened. In reality, your script is running in the background. You may see a Windows Defender notice about allowing traffic – for now, click **Allow**.

4. Next, open a web browser, and enter the following into the address bar:

```
http://localhost:8080
```

In the very upper-left corner, you should see the text ***Hello World!***

Right now, your computer is acting like a server. Any other computer that tries to access your system on port 8080 will see the *Hello World* notice.

To turn off the program, switch back to PowerShell and press **Ctrl+C**. The system will switch back to a command prompt. You can close this window whenever you are ready.

### Install Vue.js

To install Vue.js:

1. Open a command line (ie. Windows Command Prompt or PowerShell).
2. Create a new project folder: `mkdir VueProjects` and enter that directory: `cd VueProjects`.
3. Install Vue.js using Node Package Manager (npm):

PowerShell

```
npm install vue
```

Check the version number you have installed by using the command: `vue --version`.

### How to run node js in terminal?

You can run your JavaScript file from your terminal only if you have installed Node. Js in your system.

...

#### Steps :

1. Open Terminal or Command Prompt.
2. Set Path to where New. js is located (using `cd`).
3. Type “`node New. js`” and press ENTER.

### Configure NPM

The npm config command can be **used to update and edit the contents of the user and global npmrc files**.

### How to Install Node.js and NPM on Windows?

1. Step 1: Download the Installer. Download the Windows Installer from NodeJs official website. ...
2. Step 2: Install Node.js and NPM. After choosing the path, double-click to install .msi binary files to initiate the installation process. ...
3. Step 3: Check Node.js and NPM Version.

### Test javascript file using Nodejs

In order to use this framework in your application:

1. Open the root folder of your project and create a new folder called **test** in it.
2. Inside the test folder, create a new file called test.js which will contain all the code related to testing.
3. open package.json and add the following line in the scripts block.
4. "scripts": {
5. "test": "mocha --recursive --exit"
- }

### Example:

```
// Requiring module
const assert = require('assert');

// We can group similar tests inside a describe block
describe("Simple Calculations", () => {
  before(() => {
    console.log( "This part executes once before all tests" );
  });

  after(() => {
    console.log( "This part executes once after all tests" );
  });

  // We can add nested blocks for different tests
  describe( "Test1", () => {
    beforeEach(() => {
      console.log( "executes before every test" );
    });

    it("Is returning 5 when adding 2 + 3", () => {
      assert.equal(2 + 3, 5);
    });

    it("Is returning 6 when multiplying 2 * 3", () => {
      assert.equal(2*3, 6);
    });
  });

  describe("Test2", () => {
    beforeEach(() => {
      console.log( "executes before every test" );
    });
  });
});
```

```
});

it("Is returning 4 when adding 2 + 3", () => {
  assert.equal(2 + 3, 4);
});

it("Is returning 8 when multiplying 2 * 4", () => {
  assert.equal(2*4, 8);
});
});
```

Copy the above code and paste it in the test.js file that we have created before. To run these tests, open the command prompt in the root directory of the project and type the following command:

```
npm run test
```

## Output

```
C:\Windows\System32\cmd.exe

Simple Calculations
This part executes once before all tests
  Test1
executes before every test
  ✓ Is returning 5 when adding 2 + 3
executes before every test
  ✓ Is returning 6 when multiplying 2 * 3
  Test2
executes before every test
  1) Is returning 4 when adding 2 + 3
executes before every test
  ✓ Is returning 8 when multiplying 2 * 4
This part executes once after all tests

3 passing (42ms)
1 failing

1) Simple Calculations
   Test2
     Is returning 4 when adding 2 + 3:
     AssertionError [ERR_ASSERTION]: 5 == 4
       + expected - actual

       -5
       +4

       at Context.<anonymous> (test\api\test.js:33:14)
       at processImmediate (internal/timers.js:439:21)
```

## Install Vue CLI with npm

### Installation #

#### Node Version Requirement

Vue CLI 4.x requires [Node.js](#) version 8.9 or above (v10+ recommended). You can manage multiple versions of Node on the same machine with [n](#), [nvm](#) or [nvm-windows](#).

To install the new package, use one of the following commands. You need administrator privileges to execute these unless npm was installed on your system through a Node.js version manager (e.g. n or nvm).

```
npm install -g @vue/cli  
# OR  
yarn global add @vue/cli
```

After installation, you will have access to the vue binary in your command line. You can verify that it is properly installed by simply running `vue`, which should present you with a help message listing all available commands.

You can check you have the right version with this command:

```
vue --version
```

### *Initiate Vue Project using terminal*

Vue.JS is a progressive JavaScript framework used to create user interfaces and *Single-Page Applications* (SPAs), and the best way to get started quickly is to create a Vue.JS project using the Vue CLI (Command-Line Interface).

#### *Prerequisites*

[Node.js](#) version 8.9 or higher is required to use Vue CLI on our terminal (v10+ is recommended). With [nvm](#), we can manage multiple versions of Node on the same machine!

### **What is Vue CLI?**

Vue CLI is an NPM package that is installed on a specific device to allow developers/users to access the vue command through their terminal. This CLI, which can be installed globally or in a specific directory on our PC, allows us to quickly scaffold a new project and build an app with a single command.

it can be configured and extended with plugins for more advanced use cases. It is made up of several parts, including the:

- [CLI service](#) which provides multiple scripts for working with Vue projects, such as the serve, build and inspect scripts.
- [CLI plugins](#) which are NPM packages that provide additional features to our Vue project, some of these plugins includes typescript, PWA, VueX, etc.

### *Installing Vue CLI*

It is always a good idea to check if a package has already been installed on our PC before installing it, and we can do this for Vue CLI by looking at its version:

```
$ vue --version  
$ vue -V
```

If we see a version, it means that the Vue CLI has already been installed on our computer; otherwise, an error indicates that it has not been installed. We can install the Vue CLI by running the following command:

```
$ npm install -g @vue/cli
// Or
$ yarn global add @vue/cli
```

Typically, the CLI is installed globally, rather than locally, so it's accessible throughout the system.

**Note:** Even if the CLI is already installed, it's worth updating it in case it's not already updated to the latest version.

```
$ npm update -g @vue/cli
// Or
$ yarn global upgrade --latest @vue/cli
```

After successfully installing Vue CLI on our PC, we should now be able to access the Vue executable in our terminal to display a list of possible commands and their functions. This can be accomplished by running the following command:

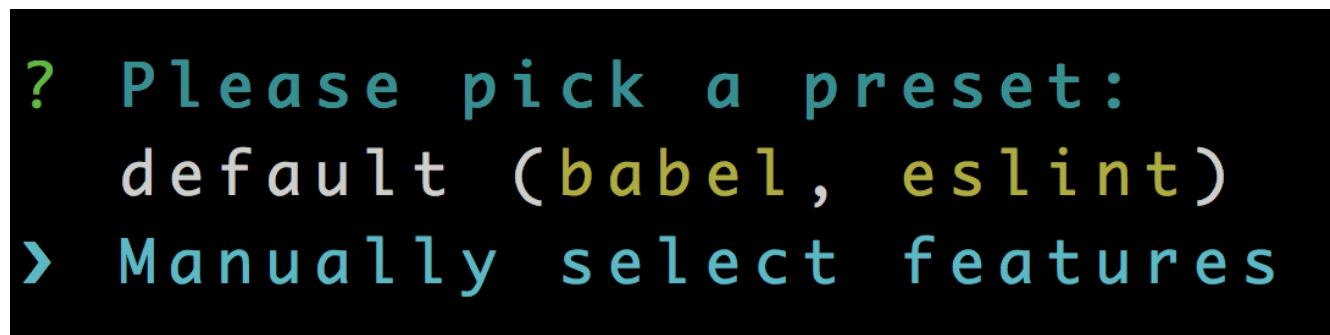
```
$ vue
Creating a Vue project
```

There are two ways we can create our project. With the newer Vue UI, or directly from the command line, which we'll do now with:

```
vue create real-world-vue
```

This command will start the creation of a Vue project, with the name of “real-world-vue”.

We'll then be prompted with the option to pick a default preset or to manually select features. Using the down arrow key, we'll highlight **Manually select features**, then hit enter.

A terminal window with a black background and light blue/green text. The prompt is '? Please pick a preset:'. Below it, 'default (babel, eslint)' is shown. The cursor is on the next line, which is '> Manually select features'.

We'll then be presented with a list of feature options. Using the down arrow key, we'll move down and use the spacebar to select **Router**, **Vuex** and **Linter / Formatter**. Then hit enter.

```
? Check the features needed for your project:
  ○ TypeScript
  ○ Progressive Web App (PWA) Support
  ● Router
  ● Vuex
  ○ CSS Pre-processors
> ● Linter / Formatter
  ○ Unit Testing
  ○ E2E Testing
```

We'll then choose a Linter / Formatter. For this project, we'll be using **ESLint + Prettier**.

```
? Pick a linter / formatter config:
  ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
> ESLint + Prettier
```

We'll add the additional feature of **Lint on save**.

```
? Pick additional lint features:
> ● Lint on save
  ○ Lint and fix on commit
```

And for the sake of this course, we'll choose to have dedicated config files.

```
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow
keys)
> In dedicated config files
  In package.json
```

We have the option to save all of these settings as a preset. We'll choose not to with N.

A terminal window with a dark background. The prompt is green text: "? Save this as a preset for future projects? (y/N)". The user has entered 'N', which is shown in white text.

If you'd like to save this as a preset, however, it will be stored in a JSON file named `.vuerc` in your user home directory.

If you have yarn installed, you'll be prompted to choose a package manager. I'll choose [npm](#).

When we hit enter, our project will be created automatically.

## Run The Project

As you can see, my terminal window gives me two commands to run the project.

The first command is to go into the project folder:

**CD my-project-name**

To run the app, run the following command.

**npm run dev**

To be organized, I'm going to run the project from my [Visual Studio Code](#) editor instead of using the Terminal window.

Go to the **Applications** folder → **Visual Studio Code**

Then, **File** → **Open** → Then navigate to the project folder that we created on the desktop and hit **open**.

As you know, the Terminal window is integrated with Visual Studio, so I can simply open it by choosing the **Terminal** option from the menu bar at the top and then **New Terminal** which will open up a new window at the bottom of the editor.

As you can see, the terminal is already into the project so I do not have to use the first command which is

**CD my-project-name**

All I have to do is run the **npm run serve** command.

Which will start running the server on my computer and give me the localhost URL.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

DONE Compiled successfully in 3254ms

App running at:
- Local:   http://localhost:8080/
- Network: http://192.168.0.5:8080/

Note that the development build is not optimized.
To create a production build, run npm run build.
```

As you can see, I get two: one is localhost and the other one is the Network URL which is great when you want to see the project on multiple devices such as checking the project UI on your mobile phone.

Open up the browser and copy the Network URL and paste it there. At this state, we are successfully up and running with our Vue 3 app.

## Description of Vue project folder & files

The [Node.js Package Manager \(npm\)](#) is the default and most popular package manager in the Node.js ecosystem, and is primarily used to install and manage external modules in a Node.js project.

What exactly is the node\_modules folder and what is it for? It just **a directory created by npm and a way of tracking each packages you install locally via package.json**.

The node\_modules folder **contains every installed dependency for your project**. In most cases, you should not commit this folder into your version controlled repository. As you install more dependencies, the size of this folder will quickly grow. Furthermore, the package-lock.

## How To Use Node.js Modules with npm and package.json

### Step 1 — Creating a package.json File

First, you will create a package.json file to store useful metadata about the project and help you manage the project's dependent Node.js modules.

### Using the init Command

First, set up a project so you can practice managing modules. In your shell, create a new folder called locator:

```
$mkdir locator
```



Then move into the new folder:

**\$cd locator**

Now, initialize the interactive prompt by entering:

**\$npm init**

Press ENTER so the default version of 1.0.0 is accepted.

## Step 2 — Installing Modules

It is common in software development to use external libraries to perform ancillary tasks in projects. This allows the developer to focus on the business logic and create the application more quickly and efficiently by utilizing tools and code that others have written that accomplish tasks one needs.

Let's run through this example. In your locator application, you will use the [axios](#) library, which will help you make HTTP requests. Install it by entering the following in your shell:

**\$npm install axios --save**

Now, open the package.json file, using a text editor of your choice. This tutorial will use nano:

**\$ nano package.json**

## Step 3 — Managing Modules

A complete package manager can do a lot more than install modules. npm has over 20 commands relating to dependency management available. In this step, you will:

- List modules you have installed.

**\$ npm ls --all**

- Update modules to a more recent version.

**\$ npm outdated**

- Uninstall modules you no longer need.
- Perform a security audit on your modules to find and fix security flaws.

### Public folder

The public folder is **the folder where we have to store static assets, like CSS files and images, media files that don't change over time**. In the Vue-cli project, the index file is located within the public folder.

**package-lock.json**

- **package-lock.json** is automatically generated for any operations where npm modifies either the **node\_modules** tree or **package.json**.
- It describes the exact tree that was generated, such that subsequent installs are able to generate identical trees, regardless of intermediate dependency updates.

The whole folder structure is actually very typical for any NodeJS project. So, you can find similar structures for other front-end frameworks such react, angular etc.

```
"name": "first-app",  
  
"version": "0.1.0",  
  
"private": true,  
  
"scripts": {  
  
  "serve": "vue-cli-service serve",  
  
  "build": "vue-cli-service build",  
  
  "lint": "vue-cli-service lint"  
},  
  
"dependencies": {  
  
  "core-js": "^3.6.5",  
  
  "vue": "^2.6.11"  
},
```

```
"devDependencies": {  
  
  "@vue/cli-plugin-babel": "~4.5.0",  
  
  "@vue/cli-plugin-eslint": "~4.5.0",  
  
  "@vue/cli-service": "~4.5.0",  
  
  "babel-eslint": "^10.1.0",  
  
  "eslint": "^6.7.2",  
  
  "eslint-plugin-vue": "^6.2.2",  
  
  "vue-template-compiler": "^2.6.11"  
},
```

```

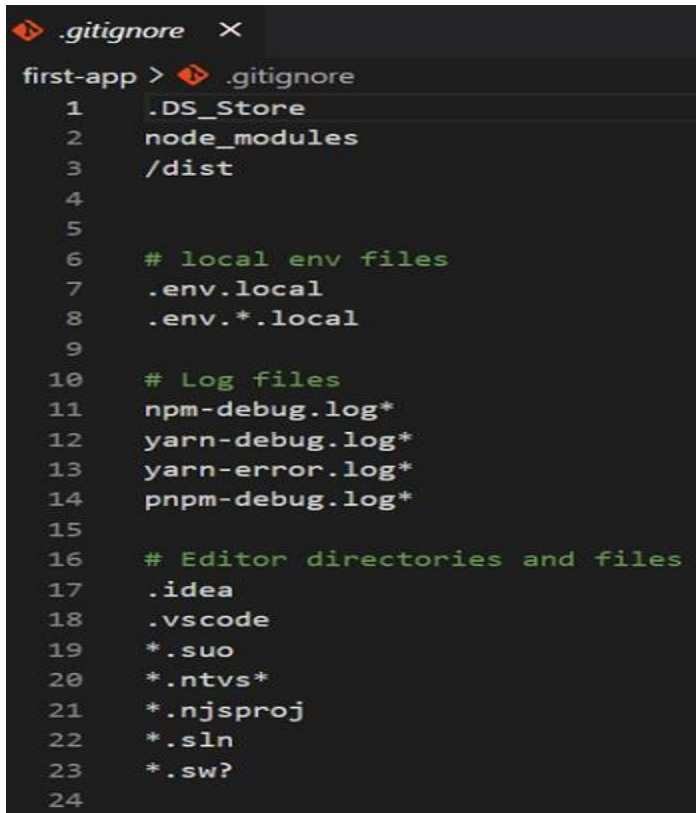
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/essential",
    "eslint:recommended"
  ],
  "parserOptions": {
    "parser": "babel-eslint"
  },
  "rules": {}
},
"browserslist": [
  "> 1%",
  "last 2 versions",
  "not dead"
]
}

```

- This file contains all the dependencies and information that are related to the project.
- This is not something specific to Vue, this is related to NodeJs. Since every Vue project created by Vue-cli and Vite is a node project, the package.json file exists here.
- This file is very crucial since this contains what are the packages the project needs, version details, meta details etc.
- **dependencies** and **devDependencies** contains an object which reflects the package name and version number. devDependencies contains the package names and versions that are only needed during the development stage. Whereas dependencies are responsible for production.
- **scripts** contain the keys and values corresponding to aliases and commands to run, test, build, etc.

## gitignore

- The gitignore file contains information such as what files and folders need to be ignored by the git versioning and which should not be pushed to the github.
- Here, you can see the **node\_modules** folder is mentioned, so it will be ignored.
- Ignored files are usually built artifacts and machine-generated files that can be derived from your repository source or should otherwise not be committed.

A screenshot of a code editor showing the contents of a .gitignore file. The file is titled ".gitignore" with a close button (X) in the top right corner. The editor shows a list of files and folders to be ignored, with line numbers 1 through 24 on the left. The content includes: 1 .DS\_Store, 2 node\_modules, 3 /dist, 4, 5, 6 # local env files, 7 .env.local, 8 .env.\*.local, 9, 10 # Log files, 11 npm-debug.log\*, 12 yarn-debug.log\*, 13 yarn-error.log\*, 14 pnpm-debug.log\*, 15, 16 # Editor directories and files, 17 .idea, 18 .vscode, 19 \*.suo, 20 \*.ntvs\*, 21 \*.njsproj, 22 \*.sln, 23 \*.sw?, 24.

```
.gitignore X
first-app > .gitignore
1  .DS_Store
2  node_modules
3  /dist
4
5
6  # local env files
7  .env.local
8  .env.*.local
9
10 # Log files
11 npm-debug.log*
12 yarn-debug.log*
13 yarn-error.log*
14 pnpm-debug.log*
15
16 # Editor directories and files
17 .idea
18 .vscode
19 *.suo
20 *.ntvs*
21 *.njsproj
22 *.sln
23 *.sw?
24
```

## src

- *src* is the source folder where we code, it contains the source code.
- It contains subfolders, assets, and components.
- The assets folder contains the assets such as files and images that are required by the source code or dynamic files.
- The components folder contains all the components that we code to create the application.
- Here it contains a component called **HelloWorld.vue**.
- Apart from folders, it also contains files such as **App.vue** and **main.js**.

## public

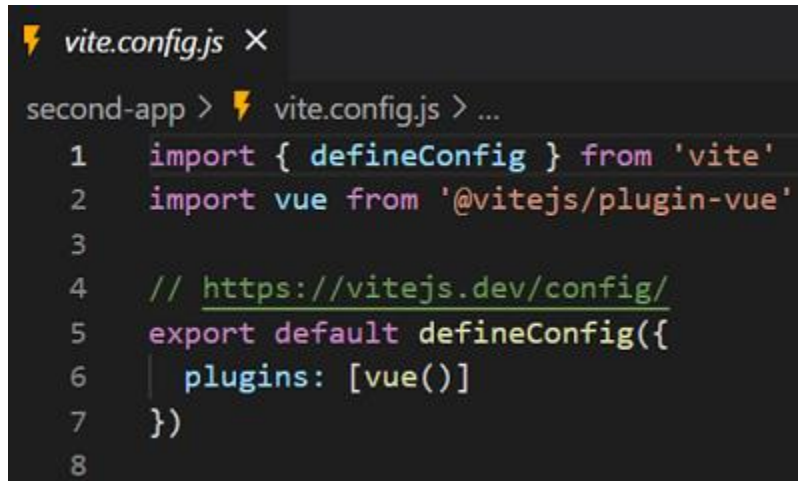
- The public folder is the folder where we have to store static assets, like CSS files and images, media files that don't change over time.
- In the Vue-cli project, the index file is located within the public folder.
- Favicon.ico file is also present here, (it is used to serve the logo that gets displayed in the browser tab as a logo).

## Presence of babel.config.js in Vue-cli



```
B babel.config.js X
first-app > B babel.config.js > ...
1  module.exports = {
2    presets: [
3      '@vue/cli-plugin-babel/preset'
4    ]
5  }
6
```

## Vue.config.js



```
⚡ vite.config.js X
second-app > ⚡ vite.config.js > ...
1  import { defineConfig } from 'vite'
2  import vue from '@vitejs/plugin-vue'
3
4  // https://vitejs.dev/config/
5  export default defineConfig({
6    plugins: [vue()]
7  })
8
```

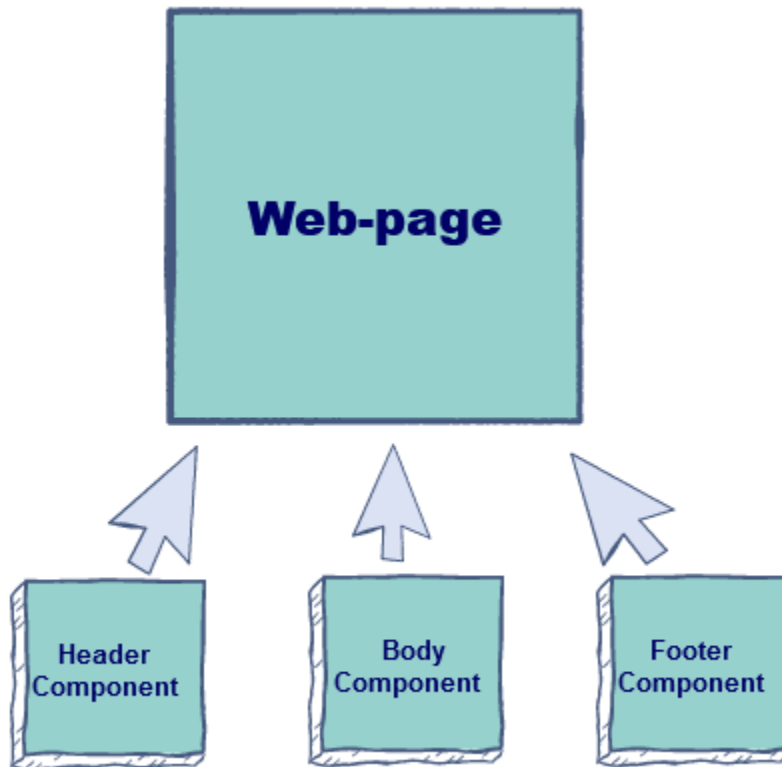
## Learning outcome 2: Apply Vue framework

- Definition of key concepts

✓ **Components:** Vue components are written as a combination of JavaScript objects that manage the app's data and an HTML-based template syntax that maps to the underlying DOM structure

**Vue JS** is a JavaScript framework used to develop single-page applications (SPAs). Based on stack-overflow surveys, Vue JS has shown to be one of the most used JavaScript frameworks.

One important feature of Vue is the ability to use components. **Components** are reusable Vue instances with custom HTML elements. Components can be reused as many times as you want or used in another component, making it a child component. Data, computed, watch, and methods can be used in a Vue component.



### How to create Vue components.

Vue components can be defined in two ways, either locally or globally.

#### Global components

The components are registered globally in your app and can be used anywhere without having to export/import the file it's contained in. Create a component globally with the following syntax:

```
vue.component('name of component, {properties}')
```

It takes two parameters, the name of the component and the object that describes the properties of the component.



```
// defining component globally
Vue.component('greeting-card', {
  template: '<p> Hello there </p>'
})
```

```
// The first vue instance
new Vue({
  el: '#app1'
})
```

```
// The second vue instance
new Vue({
  el: '#app2'
})
```

```
<html>
<head>
  <style>
    #app1{
      width:100%;
      height:100px;
      background-color:purple;
    }
    #app2{
      width:100%
      height:100px;
      background-color:green;
    }
  </style>
</head>
<body>
  <script src="https://unpkg.com/vue/dist/vue.js"></script>
  <div id="app1">
    <!-- rendering the component in the first vue instance -->
    <greeting-card></greeting-card>
  </div>
  <div id="app2">
    <!-- rendering the component in the second vue instance -->
    <greeting-card></greeting-card>
  </div>
</body>
</html>
```

## Local components

When a component is created locally, it can only be used where it is created.

In the example above, the welcome-card component was created locally by defining the component object to a variable (component name). It is then registered locally in the Vue instance; that is, it can only be used in that Vue instance.

## Props

Props are used to pass data from the parent components down to the child components – it follows a unidirectional flow.

## Single file components

When creating large web applications with different functionalities, it's best to break down the application into single file components. This enables you to have your templates, scripts, and styles all in one file with a .vue file extension:

## Dynamic components

It's efficient to dynamically switch between components without using a router; this can be done with the dynamic components. The syntax works as follows:

```
<component v-bind:is="currentComponent"></component>
```

- The component tag must be used
- The is attribute takes the current component and switches it.

Here is an example of how it can be used:

```
<div id="app">
  <!--creating button events to dynamically switch components -->
  <button @click="selectedComponent = 'GreenCard'">Show Green Card</button>
  <button @click="selectedComponent = 'PurpleCard'">Show Purple Card</button>
  <hr>
  <!-- binding the current component to the property -->
  <component :is="selectedComponent"></component>
</div>
</template>
<script>
  // Importing the components into the app
  import PurpleCard from "./PurpleCard.vue";
  import GreenCard from "./GreenCard.vue";
  export default {
    name: "app",
```



```

components: {
  GreenCard,
  PurpleCard
},
data: function() {
  return {
    // assigning the current component
    selectedComponent: "GreenCard"
  }
}

```

When naming components, it's important to note that it can be done in two ways:

- Kebab case

my-component-name

- Pascal case

MyComponentName

## Routes

✓ **Routes** : Vue router: **Vue Router helps link between the browser's URL/History and Vue's components allowing for certain paths to render whatever view is associated with it.** A vue router is used in building single-page applications (SPA). The vue-router can be set up by default while creating your new project.

**Vue Router** is the official library for page navigation in Vue applications.

In Vue, **routing** allows users to navigate between web pages without having to refresh the page, which makes navigation in a web application simple, fast, and pleasant.

Routing is one of the most powerful features of modern single-page web applications (SPA). On the client-side, modern single-page apps, such as a Vue application, can transition from page to page without requesting the server.

Vue.js does not have a built-in router feature. To use vue-router, you can install it via the Vue CLI.

You can install it through npm:

```
npm install vue-router
```

After installing the router, you can use router-link to create a route in your Vue component.

```

<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>
>

```

Or you can use the CDN alongside Vue CDN:

```
<router-link to = "/route1">Router Link 1</router-link>
<router-link to = "/route2">Router Link 2</router-link>
```

While you can easily include vue-router with vue-cli, I think it's worthwhile to know how to install it yourself.

Let's set up a simple example of vue-router.

Run the command below to add vue-router to our project.

```
import Vue from 'vue'
import App from './App.vue'
import router from './router' // loads from src/router/index.js
new Vue({
  router,
  render: h => h(App),
}).$mount('#app')
```

Create an *src/router* folder that contains an *index.js* file with the contents below.

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from './views/Home.vue'
import About from './views/About.vue'

Vue.use(VueRouter);

const routes = [
  {
    path: "/",
    name: "home",
    component: Home
  },
  {
    path: '/about',
    name: 'about',
    component: About
  }
]

const router = new VueRouter({
  mode: 'history',
  routes
})
```

export default router

Now we've set up our vue-router, but there's no way to see it yet.

We can use the router-view element to accomplish this. The router-view element essentially provides a location for vue-router to render whatever component the current URL resolves to.

We'll put router-view in the App.vue root component for our example. Let's also create some links so we can swap back and forth between our two paths. vue-router employs special router-link link elements with a to attribute that map to a component

```
<template>
  <div id="app">
    <router-link to="/">Home</router-link>
    <router-link to="/about">About Us</router-link>
    <router-view />
  </div>
</template>
```

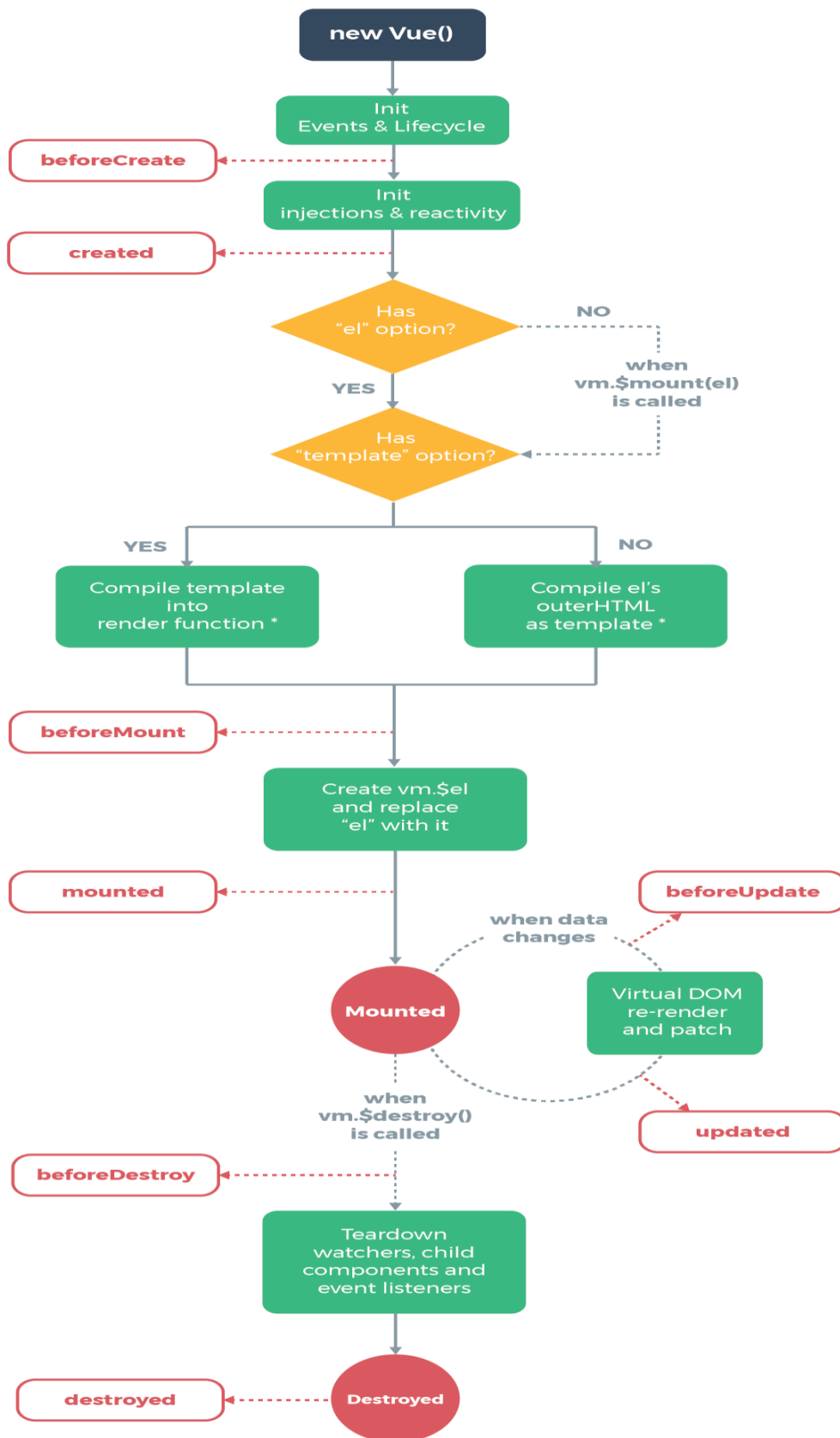
When we run our app, we should see our home component rendering. If we click our router-link elements, the content and the URL will change.

## lifecycle hooks in Vue JS:

✓ **Vue lifecycle:** Lifecycle hooks are **a window into how the library you are using works behind the scenes**. Lifecycle hooks allow you to know when your component is created, added to the DOM, updated, or destroyed. This article will introduce you to the creation, mounting, updating, and destruction hooks in Vue. Js

There are eight lifecycle hooks in Vue JS:

1. beforeCreate
2. created
3. beforeMount
4. mounted
5. beforeUpdate
6. updated
7. beforeDestroy
8. destroyed



\* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

## beforeCreate

**beforeCreate** is the first lifecycle hook that gets called in Vue JS. beforeCreate is called right after a new Vue instance is initialized. Here, the computed properties, watchers, events, data properties, etc., are not set up.

```
<script>
  export default {
    beforeCreate() {
      console.log('beforeCreate hook called');
    }
  }
</script>
```

## created

**created** is the next lifecycle hook that gets called after the beforeCreate hook. Here, the computed properties, watchers, events, data properties, etc., are also activated.

```
<script>
  export default {
    data() {
      return {
        msg: "Hello World",
      }
    }
    created() {
      console.log('created hook called', this.msg);
    }
  }
</script>
```

We will be able to access the data properties that were not accessible in the previous hook.

## beforeMount

**beforeMount** is the next lifecycle hook that gets called after the created hook and right before the Vue instance is mounted on the DOM. The template and the styles are all compiled here, but the DOM cannot be manipulated yet.

```
<script>
  export default {
    beforeMount() {
      console.log('beforeMount hook called');
    }
  }
</script>
```

## mounted

**mounted** is the next lifecycle hook that gets called after the beforeMount hook and right after the Vue instance has been mounted. The app component or any other component becomes functional and is ready to use.

```
<script>
  export default {
    mounted() {
      alert('mounted has been called');
    }
  }
</script>
```

## beforeUpdate

**beforeUpdate** is the next lifecycle hook called after the mounted hook. beforeUpdate is called any time a change is made to the data that requires the DOM to be updated.

```

<template>
  <p>
    {{ msg }}
  </p>
</template>

<script>
  export default {
    data() {
      return {
        msg: "Hello World",
      }
    },
    beforeUpdate(){
      console.log('beforeUpdate hook called');
    },
    mounted(){
      this.$data.hello= 'This is Shubham Kshatriya!';
    }
  }
</script>

```

## updated

**updated** is the next lifecycle hook. updated is called after the beforeUpdate hook and just after a DOM update has occurred.

```

<template>
  <p>
    {{ msg }}
  </p>
</template>

<script>
  export default {
    data() {
      return {
        msg: "Hello World",
      }
    },
    beforeUpdate(){
      console.log('beforeUpdate hook called');
    },
    updated(){
      console.log('updated hook called');
    }
  }
</script>

```

```
},
updated(){
  console.log('updated hook called');
},
mounted(){
  this.$data.hello= 'This is Shubham Kshatriya!';
}
}
</script>
```

## beforeDestroy

The **beforeDestroy** hook is called just before a Vue instance is destroyed. The instance and all the methods are still functional. We can do resource management here.

```
<script>
  export default {
    data() {
      return {
        msg: "Hello World!",
      }
    },
    beforeDestroy() {
      console.log('beforeDestroy hook called');
      this.msg = null
      delete this.msg;
    }
  }
</script>
```

## destroyed

**destroyed** is the last stage lifecycle hook, where the entire Vue instance gets destroyed. Event listeners, mixins, and all directives get unbounded here.



```

<script>
  export default {
    destroyed() {
      this.$destroy()
      console.log('destroyed hook called')
    }
  }
</script>

```

## ✓ State management

A state is basically an object where the data a component needs is stored. For example, we can store a message a component needs in the state and display the message whenever the component is rendered. In Vue components, the state is stored in the data function so that each instance can maintain a copy of its state. The example below shows how to create a state for a single file component

```

<template>
  <button>{{message}}</button>
</template>

<script>
export default(){
  data(){
    message: 'Hello world'
  }
}
</script>

```

### What is State Management?

State management is a way of managing the state across multiple UI components efficiently. As we discussed earlier when an application becomes complex so does managing its state due to shared data across many components that need to interact with each other.

State management libraries provide a solution to this problem by providing a centralized store for storing state that can be used across components even if the components are unrelated.

Some popular state management libraries for Vue are: [Vuex](#), [Redux](#), [Mobx](#), and [vue-stash](#).

## What is Vuex?

[Vuex](#) is a state management library and state management pattern geared towards Vue applications. It makes managing state in Vue applications easier by providing a central storage for the state (and other computations) called a store.

What is Redux in simple words?

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

What is the purpose of MobX?

“MobX is a battle tested library that **makes state management simple and scalable by transparently applying functional reactive programming (TFRP).**”

## ✓ API Endpoint

What is an API? An application programming interface (API) is code that enables two software programs to communicate

### What is an API endpoint?

An API endpoint is a point at which an API -- the code that allows two software programs to communicate with each other -- connects with the software program. APIs work by sending *requests* for information from a web application or web server and receiving a *response*

In other words, API endpoints are the specific digital location where requests for information are sent by one program to retrieve the digital resource that exists there. Endpoints specify where APIs can access resources and help guarantee the proper functioning of the incorporated software. An API's performance depends on its capacity to successfully communicate with API endpoints.

### How API endpoints work

Systems that communicate through APIs are integrated systems. One side sends the information to the API and is called the [server](#). The other side, the client, makes the requests and manipulates the API. The server side that provides the requested information, or resources, is the API endpoint.

## API REQUEST

An API request refers to the process of making a request to an application programming interface (API) to access its data or functionality. An API is a set of protocols, routines, and tools for building software applications and defining how different software components should interact with each other.

API requests can be made using HTTP methods such as GET, POST, PUT, DELETE, and others. These requests are typically made using a URL endpoint that specifies the API resource to be accessed and any parameters or data needed for the request.

API requests are commonly used in web and mobile applications to retrieve data from servers or databases, perform CRUD (Create, Read, Update, Delete) operations on data, or integrate with third-party services or

applications. API requests can also be used to trigger specific actions or functions within an API, such as sending a message or making a payment.

API requests are an important part of modern software development, as they enable developers to access and utilize a wide range of functionality and data across different applications and services.

In Vue.js, you can make API requests using various methods provided by the framework, such as Axios or the built-in `fetch()` method. Here are the basic steps for making API requests in Vue.js:

**1. Install and import Axios (if you choose to use it) in your project:**

```
Copy code
npm install axios

Copy code
import axios from 'axios'
```

**2. Define a method in your Vue.js component to make the API request. For example, using Axios:**

```
Copy code
javascript

methods: {
  fetchData() {
    axios.get('https://api.example.com/data')
      .then(response => {
        console.log(response.data)
        // update your component data with the API response
      })
      .catch(error => {
        console.log(error)
      })
  }
}
```

3. Call the API request method in your component, for example, in the `created()` lifecycle hook:

```
javascript Copy code  
  
created() {  
  this.fetchData()  
}
```

This is a basic example of how to make API requests in Vue.js using Axios. There are many other options and configurations you can use, such as adding headers, setting timeouts, and handling errors. It's important to also consider best practices for handling API requests in Vue.js, such as using computed properties and data reactivity to update your component as data is fetched.

Axios is a popular Promise-based HTTP client for making HTTP requests from a browser or Node.js. It is a JavaScript library that allows you to make HTTP requests to a server to fetch or save data, which can be used to communicate with APIs, and perform CRUD operations on a web server.

Axios provides a simple and easy-to-use API for sending asynchronous HTTP requests using methods like `get()`, `post()`, `put()`, `delete()`, and `patch()`. It also provides features such as automatic JSON data parsing and support for browser-based XSRF/CSRF protection.

Axios can be used in both browser-based and server-side applications, and it supports all modern browsers (including IE 11) and Node.js version 8.6 and above. Axios can be installed and used in a project using Node Package Manager (npm) by running the following command:

```
Copy code  
  
npm install axios
```

## WHAT IS API HELPER FILE

An API helper file is a module or class in a software application that contains methods or functions for making API requests and handling API responses. The purpose of an API helper file is to simplify the process of making API requests by encapsulating the details of the API request and response logic in a reusable and modular code.

API helper files typically contain methods or functions for common API request types such as GET, POST, PUT, DELETE, and others. These methods may also include logic for handling authentication, headers, query parameters, and error handling.

By using an API helper file, developers can reduce the amount of repetitive code needed for making API requests and handling API responses, which can improve the maintainability, scalability, and reliability of their code. API helper files can also help to enforce consistent API request and response behavior across different parts of an application, which can improve code quality and reduce bugs.

### how to Configure axios in API helper file

To configure Axios in an API helper file, you will first need to install Axios in your project using a package manager like npm. You can install Axios by running the following command:

Once Axios is installed, you can create an API helper file and import Axios into it using the following code:

```
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'http://example.com/api',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
});

export default apiClient;
```

In this example, `apiClient` is a new instance of Axios with some default options set. The `baseURL` option sets the base URL for all API requests, and the `headers` option sets some default headers that will be included in all requests.

You can customize the `apiClient` instance by adding or modifying default options. For example, you could add an `Authorization` header for API requests that require authentication:

```
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'http://example.com/api',
```

```

headers: {
  Accept: 'application/json',
  'Content-Type': 'application/json',
},
});

apiClient.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

export default apiClient;

```

## WHAT IS ENVIRONMENT VARIABLE IN VUE.JS

An environment variable in Vue.js is a key-value pair that can be used to store configuration values for your application. Environment variables are typically used to store sensitive information, such as API keys or database credentials, that you don't want to hardcode in your code.

In Vue.js, environment variables are accessed through the `process.env` object. By default, Vue.js only loads environment variables that start with `VUE_APP_`. For example, if you have an environment variable named `VUE_APP_API_URL`, you can access it in your Vue.js code like this:

```
const apiUrl = process.env.VUE_APP_API_URL
```

You can set environment variables in different ways, depending on your deployment environment. For example, you can set them in a `.env` file, in the command line when starting the development server, or through a CI/CD pipeline.

Using environment variables in Vue.js can make your application more secure, flexible, and easier to configure for different environments.

## how to use Use environment variable in vue.js

To use environment variables in a Vue.js application, you can use the `dotenv` package. Here are the steps to use environment variables in Vue.js:

1. Install the `dotenv` package using the following command:

```
npm install dotenv --save-dev
```

2. Create a `.env` file in the root directory of your Vue.js project. This file will contain all the environment variables you want to use in your application.

```

VUE_APP_API_URL=https://api.example.com
VUE_APP_DEBUG=true

```

3. In your Vue.js application, import dotenv and call config() method to load the environment variables from the .env file.

```
import dotenv from 'dotenv'  
dotenv.config()
```

4. Now, you can access the environment variables in your Vue.js components using the process.env object. For example, to use the VUE\_APP\_API\_URL variable in a component:





```
<template>  
  <div>{{ apiUrl }}</div>  
</template>  
  
<script>  
export default {  
  data() {  
    return {  
      apiUrl: process.env.VUE_APP_API_URL  
    }  
  }  
}  
</script>
```

Note that environment variables starting with VUE\_APP\_ are automatically included in your Vue.js application. Also, keep in mind that environment variables are not secret and can be accessed by anyone with access to your codebase, so avoid storing sensitive information in them.

### Fetch all CRUD APIs and display data to component

CRUD is an acronym that stands for Create, Read, Update, and Delete. CRUD operations refer to the basic operations that can be performed on a data storage system or database.

In the context of APIs, CRUD APIs refer to a set of HTTP endpoints or methods that allow you to perform the four basic operations on a resource or entity in a system:

-  **Create:** This operation creates a new resource or entity in the system. In HTTP, this is typically done using the POST method.
-  **Read:** This operation retrieves one or more resources or entities from the system. In HTTP, this is typically done using the GET method.
-  **Update:** This operation modifies an existing resource or entity in the system. In HTTP, this is typically done using the PUT or PATCH method.
-  **Delete:** This operation deletes an existing resource or entity from the system. In HTTP, this is typically done using the DELETE method.

CRUD APIs are a fundamental building block of many web applications and APIs. By providing endpoints for these basic operations, developers can easily create, retrieve, update, and delete data in a system without having to write custom logic for each operation.

## HOW TO FETCH CREATE API OPERATION TO THE COMPONENT

To fetch the Create API operation and send data from a component in Vue.js, you can follow these steps:

1. Create a form in your Vue.js component that collects the data you want to send to the API. For example, if you have a users API endpoint and you want to create a new user, you can create a form with input fields for the user's name, email, and password.

### HTML:

```
<template>
  <form @submit.prevent="createUser">
    <label>Name:</label>
    <input v-model="name" type="text">
    <label>Email:</label>
    <input v-model="email" type="email">
    <label>Password:</label>
    <input v-model="password" type="password">
    <button type="submit">Create User</button>
  </form>
</template>
```

Here, v-model is a Vue.js directive that binds the input fields to data properties in the component.

2. In the createUser method of your component, send a POST request to the users API endpoint with the data collected from the form. You can use the fetch function or a third-party library such as Axios or Vue-resource to send the request. For example:

### JAVA SCRIPT

```
methods: {
  createUser() {
    const user = {
      name: this.name,
      email: this.email,
      password: this.password
    }
    fetch('https://example.com/users', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(user)
    })
    .then(response => response.json())
    .then(data => {
      console.log('User created:', data)
    })
  }
}
data() {
```



```

return {
  name: "",
  email: "",
  password: ""
}
}

```

Here, user is an object that contains the data collected from the form, and `JSON.stringify(user)` converts it to a JSON string that can be sent in the request body. The `headers` option sets the content type of the request body to `application/json`.

3. Bind the form input fields to data properties in your component so that you can access the form data in the `createUser` method. For example:

```

data() {
  return {
    name: "",
    email: "",
    password: ""
  }
}

```

Here, `name`, `email`, and `password` are data properties in the component that store the values entered in the form input fields.

By following these steps, you can fetch the Create API operation and send data from a Vue.js component to an API endpoint.

## HOW TO FETCH READ API OPERATION TO THE COMPONENT

To fetch the Read API operation and display data in a component in Vue.js, you can follow these steps:

1. In the created lifecycle hook of your component, send a GET request to the API endpoint for the resource you want to retrieve. You can use the `fetch` function or a third-party library such as `Axios` or `Vue-resource` to send the request. For example, if you have a `users` API endpoint and you want to retrieve a list of users, you can send a GET request to <https://example.com/users>.

### JAVA SCRIPT:

```

created() {
  fetch('https://example.com/users')
    .then(response => response.json())
    .then(data => {
      this.users = data
    })
},
return {
  users: []
}
}

```

Here, `this.users` is a data property in the component that stores the data returned from the API. The `fetch` function sends a GET request to the `users` API endpoint, and the response is parsed as JSON and stored in the `users` data property.

2. Use the v-for directive in your component template to loop through the data and display it in the component. For example, you can display the list of users returned from the API as follows:

#### HTML:

```
<template>
  <div>
    <h2>Users:</h2>
    <ul>
      <li v-for="user in users" :key="user.id">{{ user.name }} - {{ user.email }}</li>
    </ul>
  </div>
</template>
```

Here, the v-for directive loops through each user in the users array and displays their name and email.

By following these steps, you can fetch the Read API operation and display data in a Vue.js component

### HOW TO FETCH delete API OPERATION TO THE COMPONENT

To fetch the Delete API operation and send a request to delete data from a component in Vue.js, you can follow these steps:

1. Add a delete button to your component that triggers a method when clicked. For example, if you have a list of users and you want to delete a user when a delete button next to their name is clicked, you can add the following button to your component

```
<template>
  <div>
    <h2>Users:</h2>
    <ul>
      <li v-for="user in users" :key="user.id">
        {{ user.name }} - {{ user.email }}
        <button @click="deleteUser(user.id)">Delete</button>
      </li>
    </ul>
  </div>
</template>
```

Here, the deleteUser method is called when the delete button is clicked and passes the user ID as an argument.

2. In the deleteUser method of your component, send a DELETE request to the API endpoint for the resource you want to delete. You can use the fetch function or a third-party library such as Axios or Vue-resource to send the request. For example, if you have a users API endpoint and you want to delete a user with a specific ID, you can send a DELETE request to `https://example.com/users/{user_id}` where {user\_id} is the ID of the user to delete.

```
methods: {
  deleteUser(userId) {
    fetch(`https://example.com/users/${userId}`, {
      method: 'DELETE'
    })
    .then(response => response.json())
    .then(data => {
      console.log('User deleted:', data)
      // Remove the deleted user from the users array in the component
    })
  }
}
```

```

const index = this.users.findIndex(user => user.id === userId)
if (index > -1) {
  this.users.splice(index, 1)
}
})
}
}

```

Here, the fetch function sends a DELETE request to the users API endpoint with the user ID appended to the URL. After the API returns a successful response, the deleted user is removed from the users array in the component.

By following these steps, you can fetch the Delete API operation and send a request to delete data from a Vue.js component.

## HOW TO FETCH UPDATE API OPERATION TO THE COMPONENT

To fetch the Update API operation and send a request to update data from a component in Vue.js, you can follow these steps:

**1. Add a form to your component that allows the user to edit the data. For example, if you have a list of users and you want to update a user's name and email, you can add the following form to your component:**

**HTML:**

```

<template>
  <div>
    <h2>Users:</h2>
    <ul>
      <li v-for="user in users" :key="user.id">
        <form @submit.prevent="updateUser(user)">
          <label>
            Name:
            <input type="text" v-model="user.name">
          </label>
          <label>
            Email:
            <input type="email" v-model="user.email">
          </label>
          <button type="submit">Update</button>
        </form>
      </li>
    </ul>
  </div>
</template>

```

Here, the updateUser method is called when the form is submitted and passes the user object as an argument. The user object is then updated with the new name and email values.

**2. In the updateUser method of your component, send a PUT or PATCH request to the API endpoint for the resource you want to update. You can use the fetch function or a third-party library such as Axios or Vue-resource to send the request. For example, if you have a users API endpoint and you want to update a user with a specific ID, you can send a PUT or PATCH request to `https://example.com/users/{user_id}` where {user\_id} is the ID of the user to update.**

```

methods: {
  updateUser(user) {
    fetch( `https://example.com/users/${user.id}`, {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(user)
    })
    .then(response => response.json())
    .then(data => {
      console.log('User updated:', data)
    })
  }
}
}

```

Here, the fetch function sends a PUT request to the users API endpoint with the updated user object in the request body. After the API returns a successful response, a message is logged to the console.

By following these steps, you can fetch the Update API operation and send a request to update data from a Vue.js component.

### Learning outcome 3: Plan game

A game is an activity that is structured, usually for entertainment or competition, and involves one or more players following a set of rules or objectives.

Games can take many forms, such as board games, video games, card games, sports, and puzzles. They can be played solo or with others, and they can be competitive or cooperative in nature.

Games often provide a challenge or goal for players to achieve, and they can be a source of enjoyment, learning, and social interaction.

Here are some common characteristics of games:

- ✚ Rules: Games have a set of rules that define what players can and cannot do within the game's framework. These rules establish the game's objectives, mechanics, and boundaries.
- ✚ Goals and objectives: Games have a specific goal or objective that players strive to achieve. This could be anything from winning a race to solving a puzzle.
- ✚ Competition or challenge: Games often involve a competitive element, whether it's against other players or against the game itself. The challenge is what makes the game engaging and rewarding.
- ✚ Interactivity: Games are interactive, meaning that players must actively participate and make choices to progress through the game.
- ✚ Feedback: Games provide feedback to players, such as points, scores, or visual cues, to let them know how well they're doing and how close they are to achieving their goal.
- ✚ Voluntary participation: Playing a game is usually a voluntary activity. Players choose to participate and can choose to stop playing at any time.

- ✚ Fun and enjoyment: Games are meant to be enjoyable and fun for the players. While they may be challenging, they should ultimately be rewarding and entertaining.

What are the types of games?

There are many different types of games, and they can be classified in a variety of ways. Here are some common ways of categorizing games:

By gameplay mechanics:

- ✚ Action games
- ✚ Adventure games
- ✚ Role-playing games
- ✚ Strategy games
- ✚ Simulation games
- ✚ Sports games
- ✚ Puzzle games
- ✚ Casual games

By platform or technology:

- ✚ Board games
- ✚ Card games
- ✚ Video games
- ✚ Online games
- ✚ Mobile games

By number of players:

- ✚ Single-player games
- ✚ Multiplayer games

By objective:

- ✚ Competitive games
- ✚ Cooperative games
- ✚ Educational games
- ✚ Serious games
- ✚ Social games

By genre:

- ✚ Fantasy games
- ✚ Science fiction games
- ✚ Horror games
- ✚ Historical games
- ✚ Realistic games

These categories are not mutually exclusive, and many games can fit into multiple categories.

**A game controller is an input device used to interact with and control video games. It is typically held in the hands and allows the player to control the movements and actions of characters in the game.**

**There are many types of game controllers, including:**

**Gamepad: A gamepad is a controller that typically has two analog sticks, a directional pad, and multiple buttons for controlling the game.**

**Joystick: A joystick is a stick-like controller that can be moved in different directions to control the game.**

**Keyboard and mouse: Many PC games are played using a keyboard and mouse, with the keyboard used for movement and the mouse used for aiming and other actions.**

**Motion controller: A motion controller uses sensors to track the movement of the controller in space, allowing the player to control the game using gestures and movements.**

**Arcade controller: An arcade controller is a specialized controller used for arcade-style games, typically featuring a joystick and buttons arranged in a specific layout.**

**Different game consoles and platforms may have their own unique controllers, such as the Nintendo Switch's Joy-Con controllers or the PlayStation's DualShock controller.**

=====

**In a game, the storyline refers to the plot or narrative that the player experiences as they progress through the game. The storyline typically includes a series of events or quests that the player must complete in order to advance the plot and ultimately reach the game's conclusion.**

**The storyline in a game may involve a main character or cast of characters, a setting or world in which the game takes place, and a conflict or goal that the player must overcome or achieve. The storyline can be conveyed through dialogue, cutscenes, and gameplay mechanics, such as the objectives or tasks that the player is given.**

**A strong storyline in a game can provide motivation for the player to continue playing, as well as a sense of immersion in the game world. It can also add depth and complexity to the game's themes and characters, making the game more engaging and memorable.**

---

**A narrative game is a type of video game that emphasizes storytelling and character development as a central gameplay mechanic. Narrative games typically place a greater emphasis on the narrative elements than on gameplay mechanics or action sequences. They may also be called story-driven games or interactive fiction.**

**Narrative games often feature a strong cast of characters and a well-developed world or setting that players can explore. Players typically progress through the game by making choices that affect the outcome of the story or by solving puzzles that advance the narrative. These games may use a variety of storytelling techniques, including dialogue, cutscenes, voiceovers, and interactive elements such as branching dialogue trees or multiple endings.**

**Narrative games can be found in a variety of genres, including adventure games, role-playing games, visual novels, and interactive movies. Some popular examples of narrative games include The Walking Dead, Life is Strange, and Detroit: Become Human.**

**The goal of a narrative game is to create an immersive and engaging experience for the player that focuses on the narrative and characters, rather than just the mechanics of the game. The player is meant to feel like they are an active participant in the story, making decisions that have a meaningful impact on the outcome.**

---

**Game settings refer to the environment, world, and atmosphere in which a game takes place. The settings can greatly affect the player's experience and immersion in the game world. Here are some examples of different types of game settings:**

- ❖ **Historical settings:** Games set in a particular historical period or event, such as Assassin's Creed set in the Crusades or Red Dead Redemption 2 set in the American Old West.
- ❖ **Fantasy settings:** Games set in a fictional world with magic, mythical creatures, and other fantasy elements, such as The Elder Scrolls series or World of Warcraft.
- ❖ **Science fiction settings:** Games set in the future or in outer space with advanced technology and futuristic elements, such as Mass Effect or Halo.
- ❖ **Real-world settings:** Games set in real-world locations, such as Grand Theft Auto set in a fictionalized version of Miami or Call of Duty set in real-world battlefields.
- ❖ **Post-apocalyptic settings:** Games set in a world after a catastrophic event, such as The Last of Us or Fallout.
- ❖ **Horror settings:** Games set in a dark, frightening world with horror elements, such as Resident Evil or Silent Hill.

**The setting of a game can greatly influence the gameplay, mechanics, and overall feel of the game. The world and atmosphere created by the setting can also be a major factor in how immersive and engaging the game is for the player.**

---

**The heads-up display (HUD) in a game is the graphical user interface (GUI) that displays important information to the player during gameplay. It typically appears on the screen as an overlay, providing the player with real-time data and feedback about the game world and the player's actions.**

**The HUD can vary depending on the game, but it usually includes some or all of the following elements:**

**Health and stamina bars:** These bars show the player's current health and how much stamina they have left.

**Ammo count:** Displays how many bullets, arrows, or other projectiles the player has left.

**Mini-map:** A small map that shows the player's location and nearby points of interest.

**Objective markers:** Arrows or symbols that point the player in the direction of their next objective or goal.

**Inventory:** Displays the items that the player is currently carrying and can use.

**Score or experience points:** Shows how many points the player has earned or how close they are to leveling up.

**Dialogue options:** Displays dialogue choices during conversations with non-playable characters (NPCs).

The HUD is an important aspect of the game that provides essential information to the player without obstructing their view of the game world. A well-designed HUD can enhance the player's experience by providing easy-to-read information and feedback, while a poorly-designed HUD can be distracting and frustrating for the player.

---

Game characteristics refer to the defining qualities and features that make up a game. Here are some of the key characteristics of a game:

- **Rules:** Games have a set of established rules that dictate how the game is played and how players can interact with the game world.
- **Goals and objectives:** Games have specific goals or objectives that players must achieve in order to win or progress through the game.
- **Interactivity:** Games allow players to interact with the game world, usually through a controller or other input device.
- **Challenge:** Games are designed to be challenging, and may require players to use skill, strategy, or critical thinking to overcome obstacles and achieve their objectives.
- **Feedback:** Games provide feedback to the player through visual and auditory cues, such as sound effects, animations, and on-screen notifications.
- **Competition:** Many games involve some form of competition, either against other players or against the game itself.
- **Progression:** Games often include a sense of progression, as players advance through levels, unlock new abilities or equipment, and face more difficult challenges.
- **Theme and narrative:** Many games have a theme or narrative that provides context for the game world and the actions of the player.

These characteristics are not always present in every game, but they are common features that help define what makes a game a game.

---



Game environment refers to the virtual world or setting in which a game takes place. It can include everything from the game's scenery, weather, and lighting to the objects, characters, and obstacles that populate the game world.

A well-designed game environment can enhance the player's sense of immersion and make the game world feel more realistic and engaging. Here are some common elements that can be found in a game environment:

1. **Terrain:** The landscape and geography of the game world, including features such as mountains, forests, and rivers.
2. **Buildings and structures:** The architecture and layout of buildings and other structures, such as houses, castles, and bridges.
3. **Weather and climate:** The weather conditions in the game world, such as rain, snow, or fog, and the impact they have on gameplay.
4. **Lighting:** The way that light is used in the game world, including natural lighting, such as sunlight or moonlight, and artificial lighting, such as streetlights or torches.
5. **Objects and items:** The various objects and items that populate the game world, such as furniture, tools, and weapons.
6. **Non-playable characters (NPCs):** The characters that the player encounters in the game world but cannot control, such as vendors, enemies, and quest-givers.
7. **Sound effects and music:** The sounds and music that accompany the game environment, such as ambient noise, background music, and sound effects for actions and events.

All of these elements work together to create a cohesive game environment that is both immersive and believable, and that helps to support the game's narrative and gameplay mechanics.

---

Game interface, also known as the user interface (UI), refers to the means by which players interact with the game world and the game's mechanics. It includes all of the visual and audio elements that help players understand how to play the game and how to achieve their objectives.

A well-designed game interface should be intuitive, easy to use, and visually appealing. Here are some common elements that can be found in a game interface:

1. **Menu screens:** These screens allow players to access different parts of the game, such as the settings, inventory, or options menu.
2. **Heads-up display (HUD):** This is the display of information that appears on the screen during gameplay, such as the player's health, ammo, and score.
3. **Control scheme:** The way that players interact with the game, such as using a gamepad, keyboard and mouse, or touch controls.
4. **On-screen prompts:** These are visual cues that appear on the screen to guide the player, such as button prompts or tutorial messages.
5. **Audio cues:** These are sounds that are used to convey information to the player, such as the sound of a weapon firing or the sound of a character speaking.
6. **Visual cues:** These are visual elements that are used to convey information to the player, such as the color of an object indicating its importance or the appearance of a character indicating their emotional state.

A well-designed game interface should provide players with the information and feedback they need to play the game effectively, without being overwhelming or distracting. It should also be visually appealing and consistent with the overall style of the game.

---

Game consoles are specialized computers designed for playing video games. They typically include a dedicated hardware system, a controller, and a library of games. There are several major game console manufacturers, including:

- ✚ Sony: Sony produces the PlayStation line of consoles, which includes the PlayStation 5, PlayStation 4, and PlayStation VR.
- ✚ Microsoft: Microsoft produces the Xbox line of consoles, which includes the Xbox Series X, Xbox Series S, and Xbox One.
- ✚ Nintendo: Nintendo produces the Switch line of consoles, which includes the Switch and Switch Lite, as well as older consoles such as the Nintendo 3DS and Wii U.
- ✚ Other companies: Other companies have produced game consoles in the past, such as Sega with the Dreamcast and Atari with the Atari 2600.

Game consoles offer several advantages over other gaming platforms, such as PCs and mobile devices. They typically offer more powerful hardware, which allows for more advanced graphics and gameplay. They also offer a more consistent user experience, as all games are designed to work with the specific console hardware and controller. Additionally, game consoles often have exclusive titles that are not available on other platforms. However, they can be more expensive than other gaming platforms, and they require a TV or monitor to play on.

Game mechanics are the **aspects of the game design, or rules, in which the player follows within the game world**. These dictate how the player acts within the game. For example, a game mechanic of having missions and objectives for that mission forces the player to adhere to those game mechanics.

## TOPIC 2:Description of the Game

### 2.1Game type

There are many different types of games, each with its own unique characteristics and gameplay mechanics. Here are some of the most common types of games:

1. Action games: These games typically involve fast-paced, physical challenges such as fighting, shooting, or platforming. Examples include the "Call of Duty" series and "Super Mario Bros."
2. Adventure games: These games usually have a strong emphasis on storytelling and exploration, often involving puzzles or riddles that the player must solve. Examples include "The Legend of Zelda" series and "Tomb Raider."
3. Role-playing games (RPGs): These games typically involve creating and developing a character, and exploring a vast game world filled with quests and challenges. Examples include "The Elder Scrolls" series and "Final Fantasy."

4. Strategy games: These games often require careful planning and resource management, as the player builds and manages a civilization, army, or other type of organization. Examples include "Civilization" and "Starcraft."
5. Sports games: These games simulate real-life sports such as football, basketball, or soccer, often allowing players to compete against each other online. Examples include "FIFA" and "NBA 2K."
6. Simulation games: These games simulate real-world activities such as driving, flying, or running a business, often with a focus on realism and accuracy. Examples include "SimCity" and "Flight Simulator."
7. Puzzle games: These games challenge the player's problem-solving skills, often with abstract or creative challenges. Examples include "Tetris" and "Minecraft."

There are also many sub-genres within each of these categories, and many games combine elements from multiple genres to create a unique gameplay experience.

## **2.2:Game objective**

The game objective is the primary goal that a player must achieve in order to win the game. It is the central purpose of the game and often provides a sense of direction and motivation for the player. The objective can vary depending on the type of game and its genre. Here are some examples:

1. Action games: The objective is often to defeat enemies or bosses, or to reach a certain point in the game world.
2. Adventure games: The objective is often to explore the game world, uncover secrets, and complete quests or puzzles.
3. Role-playing games (RPGs): The objective is often to level up the player character, acquire new equipment, and complete story missions.
4. Strategy games: The objective is often to build and manage a civilization or army, and to defeat other players or AI opponents.
5. Sports games: The objective is often to score more points or goals than the opposing team.
6. Simulation games: The objective is often to successfully run a business, city, or other organization.
7. Puzzle games: The objective is often to solve puzzles or challenges by manipulating objects or patterns.

In addition to the main objective, many games also have secondary objectives or optional goals that can provide additional rewards or challenges for the player.

## **2.3:Game target devices**

Game target devices refer to the devices or platforms that a game is designed to run on. The target device can have a significant impact on the game's performance, graphics quality, and overall player experience. Here are some of the most common game target devices:





1. Personal computers (PCs): Games designed for PCs are typically optimized for use with a keyboard and mouse, and often have high-end graphics and processing requirements.
2. Consoles: Consoles are dedicated gaming devices such as the PlayStation, Xbox, and Nintendo Switch. Games designed for consoles are optimized for use with a gamepad controller, and often have more consistent performance than PC games.
3. Mobile devices: Games designed for mobile devices such as smartphones and tablets are often optimized for touchscreens, and may have simplified gameplay mechanics to accommodate smaller screens and lower processing power.

4. Virtual reality (VR) devices: VR games are designed to run on specialized VR devices such as the Oculus Rift or HTC Vive, and provide an immersive, 360-degree gaming experience.
5. Web browsers: Some games are designed to run directly in a web browser, using technologies such as HTML5 and JavaScript.
6. Smart TVs: Some games are designed to run on smart TVs, either using a game controller or the TV's remote control.

The choice of target device can depend on a variety of factors, such as the game's genre, target audience, and development budget.

## 2.4: Game dimension



Game dimension refers to the number of dimensions that are used to represent the game world or game elements. In other words, it is the number of axes along which a game world can be explored or navigated. Here are the most common types of game dimensions:

-  2D (Two-dimensional): Games with a 2D dimension have two axes of movement, typically represented on a flat plane. Examples of 2D games include classic platformers like Super Mario Bros. and puzzle games like Tetris.
-  2.5D (Two-and-a-half-dimensional): Games with a 2.5D dimension use 2D graphics and gameplay mechanics, but can give the illusion of depth through the use of parallax scrolling or 3D effects. Examples of 2.5D games include the original Doom and the later installments of the Paper Mario series.
-  3D (Three-dimensional): Games with a 3D dimension have three axes of movement, allowing players to explore the game world in three dimensions. Examples of 3D games include open-world games like Grand Theft Auto and first-person shooters like Call of Duty.
-  4D (Four-dimensional): Games with a 4D dimension are relatively rare, but typically use time as the fourth dimension, allowing players to manipulate the flow of time within the game world. Examples of 4D games include Braid and Superhot.

**The choice of game dimension can have a significant impact on the gameplay mechanics, graphics, and overall player experience.**

## 2.5: Game perspective

Game perspective refers to the point of view from which the player views the game world. There are several types of game perspectives that developers can use to create different gameplay experiences. Here are some of the most common types of game perspectives:

-  First-person: In a first-person perspective, the player sees the game world through the eyes of the character they are controlling. This perspective is often used in first-person shooters and immersive games like virtual reality experiences.
-  Third-person: In a third-person perspective, the player sees the game world from behind the character they are controlling. This perspective allows for a wider view of the game world and is often used in action-adventure games and platformers.

- ✚ Top-down: In a top-down perspective, the player views the game world from above, looking down on the characters and environment. This perspective is often used in strategy and simulation games.
- ✚ Side-scrolling: In a side-scrolling perspective, the player views the game world from the side as it scrolls horizontally. This perspective is often used in platformers and action games.
- ✚ Isometric: In an isometric perspective, the game world is viewed from a three-quarter perspective, giving the illusion of three-dimensional depth. This perspective is often used in strategy and simulation games.
- ✚ The choice of game perspective can have a significant impact on the gameplay mechanics, graphics, and overall player experience. Different perspectives can create different levels of immersion, challenge, and engagement for players.

### **TOPIC 3.Creation of Narrative**

#### 3.1 Storyline

#### 3.2 Sounds

#### 3.3 Background music

#### 3.4 Environment (scenery)

#### 3.5 Game level / reward level

In video games, a game level is a specific area or stage of the game that the player progresses through. Each level is designed to challenge the player in some way, either by introducing new obstacles or enemies, increasing the difficulty of existing challenges, or testing the player's skills in some other way.

Levels can take many different forms depending on the type of game, and they can be structured in a variety of ways. For example, in a platformer game, levels might consist of a series of platforms and obstacles that the player must navigate through, while in a racing game, levels might take the form of different race tracks or courses.

In more open-world games, levels might refer to different areas of the game world that the player can explore, each with its own challenges and objectives. Levels can also be used to mark progress within the game, with each level representing a milestone that the player has reached.

Overall, game levels are an essential part of video game design, providing structure, challenge, and a sense of progression for players as they journey through the game.

---

Designing a game level involves several steps, and the specific process can vary depending on the type of game and the tools you are using. Here are some general steps you can follow to create a game level:

- ✚ Define your objectives: Before you start designing your level, you need to define what the player's objectives are. What challenges do you want them to face, what skills do you want them to use, and what do you want them to achieve by the end of the level?
- ✚ Create a concept: Once you have your objectives, create a concept for your level. Consider the environment, obstacles, enemies, and other elements that will challenge the player and create an engaging experience.

- ✚ Create a layout: Sketch out a rough layout of your level, including the placement of obstacles, enemies, and other elements. Consider the pacing of the level and the flow of gameplay, making sure that there are no dead ends or areas that are too easy or too difficult.
- ✚ Build the environment: Using your layout as a guide, start building the environment for your level. This can involve creating 3D models, adding textures and lighting, and placing props and other objects.
- ✚ Add gameplay elements: Once you have the environment in place, start adding gameplay elements such as enemies, traps, puzzles, and other challenges. Make sure that these elements are balanced and challenging but not frustrating.
- ✚ Playtest and iterate: Test your level frequently to ensure that it is fun and engaging for the player. Make adjustments as needed, tweaking the layout, adjusting the difficulty, or adding or removing elements as necessary.
- ✚ Polish and finalize: Once you have a level that is fun and challenging, add finishing touches such as sound effects, music, and special effects to give the level a polished, professional feel.

### **3.6 Mission: main and side**

The main mission of a game is the central objective that the player is trying to achieve, while a side mission is an optional mission or quest that the player can choose to undertake. Here are some key differences between the two:

- ✚ Importance: The main mission is usually the most important objective in the game, and completing it is necessary to progress through the game's story or to achieve the ultimate goal. Side missions, on the other hand, are typically optional and do not affect the overall outcome of the game.
- ✚ Focus: The main mission is the primary focus of the game, and the gameplay is often structured around achieving it. Side missions, on the other hand, are usually more focused on exploration, discovery, and providing additional challenges or rewards.
- ✚ Difficulty: The main mission is usually the most challenging and demanding objective in the game, requiring the player to use all of their skills and abilities to achieve it. Side missions, on the other hand, can vary widely in difficulty, and are often designed to be more accessible and approachable than the main mission.
- ✚ Rewards: Completing the main mission typically provides the player with the biggest reward or sense of accomplishment, and often leads to a satisfying conclusion to the game's story. Side missions, on the other hand, usually provide smaller rewards such as experience points, new items or equipment, or additional story information.
- ✚ Overall, the main mission is the central focus of the game, while side missions are optional activities that can provide additional challenges and rewards. Both types of missions are important for providing a well-rounded and engaging gameplay experience.

## **TOPIC4 GAME MECHANICS**

Game mechanics are the rules, systems, and interactions that define how a game is played. They are the building blocks of the gameplay experience, and determine how the player interacts with the game world, the challenges they face, and the actions they can take.

#### 4.1 Key elements for defines game mechanics

- game hud (heads-up display)
- Steps of the game
- Scores
- Level
- Speed
- Time
- Target Device

how to determine game mechanism

Determining game mechanics involves a combination of creative and technical processes. Here are some general steps you can take to determine the game mechanics for your game:

- ✚ Define the core gameplay: Start by defining the core gameplay experience you want to create. What kind of game do you want to make? What is the central challenge or objective? What kind of player actions do you want to enable?
- ✚ Research and analyze existing games: Research and analyze games that are similar to the game you want to create. Analyze their mechanics, and identify what works and what doesn't. This can give you inspiration and ideas for your own game mechanics.
- ✚ Brainstorm mechanics: Once you have a clear idea of the core gameplay, start brainstorming mechanics that will enable the player to achieve the objectives of the game. Consider the controls, movement, combat, progression, economy, and puzzles you want to include in the game.
- ✚ Create a prototype: Create a prototype of the game that includes your proposed game mechanics, and test it out to see how it feels and plays. Iterate on your design based on player feedback, and refine your mechanics to make the gameplay more engaging and intuitive.
- ✚ Refine and balance: Once you have a rough prototype, work on refining and balancing the game mechanics. Ensure that the mechanics are well-designed, intuitive, and balanced, and that they work together to create a cohesive gameplay experience.
- ✚ Test and iterate: Continuously test and iterate on your game mechanics throughout the development process. This will help you identify and address any issues or imbalances in the mechanics, and ensure that the gameplay is engaging and enjoyable.
- ✚ Determining game mechanics is an iterative process that requires a combination of creative thinking, technical skills, and playtesting. By following these steps, you can develop effective and engaging game mechanics that will help make your game a success.

#### **TOPIC5:: Identification of game controls/.**

Identifying game controls is an important part of game design. Here are some general steps you can take to identify the game controls for your game:



- ✚ Determine the platform: The platform you are designing the game for will impact the types of controls you can use. For example, a PC game may have keyboard and mouse controls, while a mobile game may use touch controls. Consider the platform and devices your game will be played on when designing the controls.
- ✚ Identify the core gameplay mechanics: Identify the core gameplay mechanics you want to include in your game. What kind of actions will the player need to take to achieve their objectives? What are the essential actions that the player must be able to perform?
- ✚ Choose the input devices: Choose the input devices that will allow the player to perform the necessary actions. This may include buttons, joysticks, touchscreens, motion controls, or other input devices.
- ✚ Map the controls: Map the necessary actions to the chosen input devices. Determine which buttons or inputs will trigger which actions, and make sure the mapping is intuitive and easy to understand.
- ✚ Test and iterate: Test the controls in a prototype or early version of the game, and iterate on the design based on player feedback. Refine the controls to make them more intuitive and responsive, and ensure that they provide a smooth and satisfying gameplay experience.

Identifying game controls requires careful consideration of the platform, gameplay mechanics, input devices, and player experience. By following these steps and testing the controls throughout the development process, you can create a game with responsive and intuitive controls that enhance the player's experience.

## **WHAT ARE THE INPUTS OF GAME CONTROLS**

The inputs of game controls refer to the physical or virtual devices that a player uses to interact with the game. These inputs can vary depending on the platform and type of game, but here are some common examples:

- ✚ Keyboard: A keyboard is a common input device for PC games. Players can use the keyboard to input text and perform actions using various keys.
- ✚ Mouse: A mouse is often used in combination with a keyboard for PC games. Players can use the mouse to control the movement and actions of the player character, as well as to interact with game menus and interfaces.
- ✚ Gamepad/controller: A gamepad or controller is a handheld input device that can be used for console and PC games. It typically features buttons, joysticks, triggers, and other inputs that can be used to control the player character and perform actions in the game.
- ✚ Touchscreen: Touchscreens are commonly used for mobile games. Players can use their fingers to interact with the game world, move the player character, and perform actions.
- ✚ Motion controls: Motion controls use sensors to detect the player's movements and translate them into actions in the game. Examples include the Wii Remote, PlayStation Move, and Kinect.



- ✚ Voice commands: Some games allow players to use voice commands to control the player character and perform actions. This can be done using a microphone or other input device.

The inputs of game controls can vary depending on the type of game and the platform it is designed for. Game designers need to carefully consider the inputs available to players and design the game controls accordingly to create an intuitive and enjoyable gameplay experience.

## **WHAT ARE THE KEYS OF GAME CONTROLS**

The keys of game controls refer to the specific buttons and controls used to interact with a game. The keys can vary depending on the platform, type of game, and input device being used, but here are some common examples:

- ✚ Arrow keys: The arrow keys on a keyboard are commonly used to move the player character in a 2D game or to control the camera in a 3D game.
- ✚ WASD keys: The WASD keys are commonly used in PC games to control the player character's movement. W moves the character forward, A moves left, S moves backwards, and D moves right.
- ✚ Spacebar: The spacebar is often used to make the player character jump or perform some other action.
- ✚ Action buttons: Action buttons are used to perform actions in the game. These can include buttons like X, Y, A, and B on a console controller, or mouse buttons on a PC.
- ✚ Trigger buttons: Trigger buttons are often used in console games to perform actions like shooting or accelerating. They are typically located on the back of the controller and can be pressed with the index fingers.
- ✚ Touchscreen controls: Touchscreen controls on mobile devices can include tapping, swiping, and other gestures to move the player character or perform actions.
- ✚ Joystick: A joystick is a physical control stick that can be used to control the player character's movement or other actions in the game.

### **Hand accessibility**

- Primary control: thumb and index
- Secondary control: Middle fingers

- Support: Ring & pinkie fingers

## WHAT ARE THE TYPES OF GAME CONTROLLES

**There are several types of game controllers, each designed to provide a unique gaming experience. Here are some common types of game controllers:**

- ✚ Gamepad: A gamepad is a handheld controller with buttons, triggers, and joysticks that is typically used for console gaming. It is designed to be held in both hands and used to control the player character's movement, actions, and other game features.
- ✚ Keyboard and mouse: A keyboard and mouse are commonly used for PC gaming. The keyboard is used to input text and commands, while the mouse is used to control the player character's movement and actions.
- ✚ Joystick: A joystick is a physical control stick that can be used to control the player character's movement or other actions in the game. It is often used for flight simulators and other games that require precise control.
- ✚ Racing wheel: A racing wheel is a specialized controller designed for racing games. It includes a steering wheel, pedals, and other controls that mimic the experience of driving a car.
- ✚ Light gun: A light gun is a controller that looks like a gun and is used to shoot at targets on the screen. It is commonly used for shooting games and arcade games.
- ✚ Motion controllers: Motion controllers use sensors to detect the player's movements and translate them into actions in the game. Examples include the Wii Remote, PlayStation Move, and Kinect.
- ✚ Virtual reality controllers: Virtual reality controllers are designed to provide a more immersive gaming experience in virtual reality games. They often include motion sensors, buttons, and other controls that allow players to interact with the game world in a more natural way.

### **Identification of Game Interface**

The game interface refers to the visual and interactive elements that allow players to navigate and interact with the game. It includes menus, buttons, icons, and other on-screen elements. Here are some common components of game interfaces:

- ✚ Main menu: The main menu is the first screen that players see when starting the game. It typically includes options to start a new game, continue a saved game, adjust settings, and exit the game.
- ✚ Heads-up display (HUD): The HUD displays important information about the game, such as the player's health, inventory, and score. It is typically displayed on the screen while playing the game.

- ✚ Inventory: The inventory displays the items that the player has collected throughout the game. It allows the player to manage their items and use them when needed.
- ✚ Map: The map displays the layout of the game world and allows players to navigate to different locations.
- ✚ Dialogue boxes: Dialogue boxes display conversations between characters in the game. They allow players to make choices and advance the story.
- ✚ Tooltips: Tooltips provide additional information about items or actions in the game. They can be displayed when the player hovers over an item or button.
- ✚ Quick time events (QTEs): QTEs are interactive sequences that require the player to press a button or perform an action in response to on-screen prompts.

These are just a few examples of the components of a game interface. The specific interface elements will depend on the type of game and the goals of the game designer. A well-designed game interface should be intuitive, easy to navigate, and enhance the overall gaming experience.

### ✓ **Splashscreen**

A splash screen is the introductory screen that appears when a game is launched. It typically displays the game's title, logo, and any relevant graphics or animations. The purpose of a splash screen is to create a sense of anticipation and excitement for the player and to provide a visual identity for the game. It also serves a practical purpose by allowing the game to load any necessary assets or resources while the splash screen is being displayed. Once the game has finished loading, the splash screen typically disappears and the main menu or game interface is displayed. A well-designed splash screen can help to establish the tone and style of the game, and create a memorable first impression for the player.

### **Here are some general steps for designing a splash screen for a game:**

Determine the theme and style of your game: The splash screen should reflect the overall theme and style of your game. This includes the color scheme, graphics, and typography.

- ✚ Create a visual concept: Develop a rough idea of what you want your splash screen to look like. Sketch out some basic layouts and ideas.
- ✚ Choose a color scheme: Choose a color scheme that matches the theme and style of your game. Consider using colors that are visually appealing and stand out.
- ✚ Design your logo: If your game has a logo, include it on the splash screen. Make sure the logo is easily recognizable and reflects the tone of the game.

- ✚ Choose graphics and animation: Consider adding graphics or animations to make the splash screen more visually appealing. However, be careful not to make the splash screen too complex or slow to load.
- ✚ Optimize for different devices: Make sure your splash screen is optimized for different screen sizes and resolutions.
- ✚ Test and refine: Once you have designed your splash screen, test it on different devices and with different users. Refine the design as needed based on feedback.

Remember, the main purpose of a splash screen is to create a memorable first impression for the player. Make sure your splash screen reflects the theme and style of your game, and is visually appealing and easy to read.

## **GAME PLAY GUIDE**

A game play guide is a document or set of instructions that provides players with information on how to play a game. It typically includes a detailed explanation of the game's rules, mechanics, controls, objectives, and other important information that players need to know in order to play the game effectively.

Game play guides can take many forms, depending on the complexity and genre of the game. For example, a game play guide for a simple mobile puzzle game might include step-by-step instructions on how to solve each level, while a guide for a complex role-playing game might provide a detailed walkthrough of the game's story and quests, as well as advice on character development and strategy.

Game play guides can be provided by the game's developer or publisher, or they can be created by dedicated fans and communities. They can be in the form of text-based documents, videos, or interactive tutorials. Providing a clear and comprehensive game play guide can enhance the player's experience and increase their engagement with the game.

## **ALERT MESSAGE**

An alert message is a pop-up notification that appears on a computer or mobile device to inform the user about a particular event or situation that requires their attention. Alert messages can be used to provide important information, warnings, errors, or confirmation messages to users in real-time.

For example, when a user enters incorrect login credentials on a website, an alert message may appear stating that the username or password is incorrect. Or, when a user tries to delete an important file on their computer, an alert message may appear asking for confirmation before proceeding with the action.

Alert messages are an important part of user interface design and can help users stay informed and avoid errors or mistakes. It is important to ensure that alert messages are clear, concise, and provide enough information for users to take appropriate actions.

## Learning outcome 4: Develop Game

### • Definition of key concepts

✓ **Deployment** : Deployment in game development refers to the process of making a game ready for release or distribution to the target audience. It involves preparing the game for different platforms, such as PC, consoles, mobile devices, or the web. The deployment process may include tasks such as packaging the game, creating installers, testing and debugging, and uploading the game to distribution platforms.

**Here are some of the key steps involved in the deployment process in game development:**

- 1) **Build the game:** This involves compiling the game code, assets, and resources into a single executable file or package.
- 2) **Test the game:** It is important to test the game thoroughly to ensure that it is stable, runs smoothly, and is free of bugs and glitches. This may involve beta testing with a group of testers to get feedback and fix any issues.
- 3) **Prepare the game for distribution:** This involves creating installers, setting up distribution platforms, and creating promotional materials such as trailers, screenshots, and game descriptions.
- 4) **Release the game:** This is the final step in the deployment process, where the game is released to the target audience. This may involve uploading the game to various distribution platforms, such as Steam, Apple App Store, Google Play Store, or the web.

### ✓ **Deployment/Hosting platforms**

**Hosting platforms** are online services that provide the infrastructure and tools needed to store, manage, and run web applications, websites, and other digital content. They allow individuals and organizations to publish their content on the internet without having to set up and maintain their own servers and infrastructure.

Some common types of hosting platforms include:

- 🌐 **Web hosting:** This type of hosting is designed for hosting websites, blogs, and other web-based content. Web hosting services typically provide the server space, bandwidth, and tools needed to create and manage websites, such as control panels, site builders, and content management systems.
- 🌐 **Cloud hosting:** This type of hosting provides scalable and flexible hosting solutions that can adapt to changing traffic and resource demands. Cloud hosting services allow users to pay only for the resources they need, such as storage, processing power, and bandwidth, and scale up or down as needed.

- ✚ Application hosting: This type of hosting is designed for hosting and running web applications and software. Application hosting services provide the platform, tools, and environment needed to develop, test, deploy, and manage web applications.
- ✚ E-commerce hosting: This type of hosting is designed specifically for hosting online stores and e-commerce websites. E-commerce hosting services typically provide features such as shopping cart software, payment gateways, and inventory management tools.

## WHAT IS HOSTING PLATFORMS IN GANE DEVELOPMENT

**In game development, hosting platforms** refer to the services that provide infrastructure and tools needed to store, manage, and run online games or multiplayer games. These platforms are designed to handle the networking requirements and provide the necessary computing resources to run game servers, matchmaking, player management, and other related services.

Game developers can use hosting platforms to deploy their games on the cloud, making them accessible to players from anywhere in the world. The hosting platforms provide the necessary infrastructure and tools to ensure that the game servers run smoothly, with minimal lag or downtime, and can scale up or down depending on player demand.

In summary, hosting platforms in game development are online services that provide the infrastructure and tools needed to run online games, allowing game developers to focus on creating the game content rather than managing the infrastructure.

### ✓ Domain name

A **domain name** is a unique string of characters that identifies a website on the internet. It is the human-readable address that people use to access a website, such as `www.example.com`. A domain name is used to translate the IP address of a website into a more user-friendly format that people can easily remember and type into a web browser.

### A domain name consists of two parts:

- ❖ the top-level domain (TLD) and the second-level domain (SLD). The top-level domain is the part of the domain name that appears to the right of the last dot in the name, such as `.com`, `.org`, `.net`, `.edu`, or `.gov`.
- ❖ The second-level domain is the part of the domain name that appears to the left of the TLD, such as "example" in `www.example.com`.

Domain names are registered with domain name registrars, which are organizations accredited by the Internet Corporation for Assigned Names and Numbers (ICANN). When someone registers a domain name, they are given the exclusive right to use that domain name for a specified period of time, typically one to ten years, after which they can renew the registration.

Domain names are an important part of online branding and marketing, as they provide a unique and memorable way for people to access a website. They are also used for email addresses, such as `info@example.com`, and can be used to protect intellectual property by preventing others from using similar domain names that could cause confusion or infringe on trademark rights.

### ✓ SASS

**SASS (Syntactically Awesome Style Sheets)** is a CSS preprocessor that extends the capabilities of traditional CSS. It is a scripting language that is compiled into CSS, providing web developers with additional features such as variables, mixins, nested selectors, and inheritance.

SASS code is written in a .scss or .sass file, which is then compiled into standard CSS code that can be used on a website. Some of the benefits of using SASS include:

- ✚ Variables: SASS allows developers to declare variables for commonly used values, such as colors or font sizes, and then reference these variables throughout the code. This makes it easier to maintain consistency across a website and update values in one place.
- ✚ Mixins: Mixins are reusable blocks of code that can be called throughout the SASS file. This makes it easier to apply complex styles to multiple elements without having to repeat the code.
- ✚ Nesting: SASS allows developers to nest selectors, making it easier to organize and write CSS code.
- ✚ Inheritance: SASS supports inheritance, allowing developers to create classes that inherit properties from other classes. This makes it easier to write and maintain complex styles.

SASS is widely used by web developers to streamline the process of writing and maintaining CSS code. It is compatible with most modern web browsers and can be used with popular front-end development frameworks such as Bootstrap and Foundation.

## ✓ CANVAS

**In web development, Canvas** is a HTML5 element that allows developers to create and render graphics, animations, and other visual content on a web page. It provides a 2D drawing API that allows developers to draw shapes, lines, text, and images using JavaScript.

Canvas is a rectangular area defined by a width and height in pixels, and can be styled using CSS. The content of a canvas element is not pre-defined and must be drawn programmatically using JavaScript.

Canvas has a number of features that make it a powerful tool for web developers, including:

- Drawing and animation: Canvas provides a range of methods for drawing and animating shapes, images, and other content on a web page.
- Interactivity: Developers can use JavaScript event listeners to create interactive elements on the canvas, such as buttons or menus.
- Performance: Canvas is designed to be fast and efficient, making it a good choice for creating high-performance visual content.

- **Compatibility:** Canvas is supported by all modern web browsers, making it a reliable way to create visual content that works across different platforms.

Canvas is widely used in web development for creating games, data visualizations, and other types of interactive content. It is often used in conjunction with other web technologies such as HTML, CSS, and JavaScript to create rich and engaging web experiences.

## ✓ SVG

**SVG stands for Scalable Vector Graphics.** It is a vector image format used in web development that describes graphics in terms of mathematical equations rather than pixels. This allows SVG images to be scaled up or down without losing quality, making them ideal for use in responsive web design.

SVG images are created using XML markup language and can be edited using a text editor or specialized SVG editors such as Adobe Illustrator, Inkscape, or Sketch. They can also be animated and interacted with using JavaScript.

Some benefits of using SVG images in web development include:

- **Scalability:** SVG images can be scaled up or down without losing quality, making them ideal for use in responsive web design.
- **Smaller file size:** SVG images have a smaller file size than raster images, such as JPEG or PNG, which means faster load times and better performance.
- **Accessibility:** SVG images are accessible to screen readers and other assistive technologies, making them ideal for creating inclusive web experiences.
- **Interactivity:** SVG images can be animated and interacted with using JavaScript, providing opportunities for creating engaging and dynamic web content.
- **SVG images are widely used in web development for creating icons, logos, infographics, and other types of visual content. They are supported by all modern web browsers and can be used inline in HTML or included as external files.**

## How to Design game environment

Designing a game environment involves creating a world that is engaging, immersive, and memorable for players. Here are some tips for designing a game environment:

- ❖ **Establish a strong visual style:** The game environment should have a strong visual style that sets it apart from other games. This could involve creating a unique color palette, using specific textures or materials, or designing distinct landmarks or architecture.



- ❖ **Create a cohesive world:** The game environment should feel like a cohesive world that has its own logic and rules. This means designing a consistent geography, climate, and ecology, and making sure that everything fits together logically.
- ❖ **Add interactive elements:** Interactive elements in the environment can make it more engaging for players. This could involve adding hidden secrets or Easter eggs, designing puzzles or challenges that require the player to interact with the environment, or creating NPCs that offer quests or dialogue.
- ❖ **Use sound and music:** Sound and music can add another layer of immersion to the game environment. Consider designing ambient sound effects that match the environment, or using music to create emotional tones or cues.
- ❖ **Test and iterate:** Like any aspect of game design, designing a game environment is an iterative process. Test the environment with real players, gather feedback, and make improvements based on that feedback.

By following these tips, you can design a game environment that is immersive, engaging, and memorable for players, and that adds another layer of enjoyment to the overall game experience.

### **how to Setup Html Canvas of game interface?**

Setting up an HTML canvas is an important step in creating a game interface. Here are the basic steps to set up an HTML canvas for a game:

- **Create a canvas element:** In your HTML file, create a canvas element using the "canvas" tag. Give the canvas an ID so you can reference it later in your JavaScript code.
- **Set the canvas size:** Use CSS to set the width and height of the canvas element. This should be set to the size you want your game interface to be.
- **Get the canvas context:** In your JavaScript code, get the canvas context by using the "getContext" method. This will return an object that you can use to draw on the canvas.
- **Draw on the canvas:** Once you have the canvas context, you can use it to draw on the canvas. This could involve drawing shapes, images, or text. You can also use the canvas to create animations, respond to user input, or create interactive elements.
- **Update the canvas:** In order to create animations or other interactive elements, you will need to update the canvas on a regular basis. This can be done using the "requestAnimationFrame" method or by setting up a timer to call a function that updates the canvas.

Here is an example code snippet to create a canvas element:

html

Copy code

```
<canvas id="game-canvas" width="800" height="600"></canvas>
```

And here is an example JavaScript code to get the canvas context and draw a rectangle on the canvas:

javascript

Copy code

```
const canvas = document.getElementById('game-canvas');
const ctx = canvas.getContext('2d');

ctx.fillStyle = 'blue';
ctx.fillRect(50, 50, 100, 100);
```

This will draw a blue rectangle with a width of 100 and a height of 100 at the position (50, 50) on the canvas.

## how to Draw in canvas HTML tags using Js

Drawing on a canvas element in HTML using JavaScript involves using the canvas API to create shapes, images, and other graphical elements on the canvas. Here are the basic steps to draw on a canvas using JavaScript:

1. Get the canvas context: To draw on a canvas, you need to first get the canvas context. This can be done using the "getContext" method, which returns an object that allows you to draw on the canvas.

javascript

Copy code

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
```

2. Set drawing properties: Before drawing, you may want to set certain properties such as fill color, stroke color, line width, and font.


javascript

Copy code

```
ctx.fillStyle = 'red';
ctx.strokeStyle = 'blue';
ctx.lineWidth = 2;
ctx.font = '20px Arial';
```

3. Draw shapes: There are several methods for drawing shapes on the canvas, including `fillRect`, `strokeRect`, `fillCircle`, and `strokeCircle`.


javascript

 Copy code

```
ctx.fillRect(x, y, width, height);
ctx.strokeRect(x, y, width, height);
ctx.beginPath();
ctx.arc(x, y, radius, startAngle, endAngle, anticlockwise);
ctx.fill();
ctx.stroke();
```

4. Draw text: You can also draw text on the canvas using the `fillText` or `strokeText` methods.


javascript

 Copy code

```
ctx.fillText('Hello, world!', x, y);
ctx.strokeText('Hello, world!', x, y);
```

5. Use images: You can draw images on the canvas using the `drawImage` method. This method can take an image element or a URL for the image file.

javascript

 Copy code

```
const img = new Image();
img.src = 'path/to/image.png';
img.onload = function() {
  ctx.drawImage(img, x, y, width, height);
}
```

6. Update the canvas: To create animations or interactive elements, you will need to update the canvas on a regular basis using the `requestAnimationFrame` method or a timer.

```
javascript Copy code

function draw() {
  // Clear canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // Draw shapes, images, and text
  // ...

  // Update canvas
  requestAnimationFrame(draw);
}

requestAnimationFrame(draw);
```

By following these steps, you can create dynamic and interactive graphical elements on a canvas element in HTML using JavaScript.

### how to design game interface using Style Environment using SASS

SASS (Syntactically Awesome Style Sheets) is a preprocessor scripting language that extends CSS and makes it more powerful and flexible. Here are some steps to design a game interface using SASS:

- 1) Install SASS: To use SASS, you will need to install it on your computer. You can do this using a package manager like npm or by downloading the SASS compiler directly.
- 2) Create a SASS file: Create a SASS file for your game interface styles and save it with a ".scss" extension. This file will contain all of your SASS code.
- 3) Set up SASS variables: Use SASS variables to store commonly used values like colors, font sizes, and spacing. This will make it easier to update your styles later on.

```
scss                                                                    Copy code

$primary-color: #FF6347;
$secondary-color: #00BFFF;
$font-size: 16px;
$spacing: 10px;
```

- 4) Use SASS nesting: SASS nesting allows you to nest selectors inside one another, which makes your code more readable and easier to manage.

```
scss                                                                    Copy code

.game-container {
  background-color: $primary-color;
  padding: $spacing;

  .game-header {
    font-size: $font-size;
    color: $secondary-color;
  }
}
```

- 5) Use SASS mixins: Mixins are reusable blocks of code that can be included in multiple styles. This can make your code more efficient and easier to maintain.

```
scss                                                                    Copy code

@mixin button-style {
  background-color: $secondary-color;
  color: #fff;
  padding: $spacing;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.game-container {
  .game-button {
    @include button-style;
  }
}
```

- 6) Compile the SASS file: Use the SASS compiler to compile your SASS file into CSS. You can do this from the command line or by using a tool like CodeKit or Prepros.
- 7) Link the CSS file: Link the compiled CSS file to your HTML file using a "link" tag.

```
html                                                                    Copy code

<head>
  <link rel="stylesheet" href="styles.css">
</head>
```