10.2 HD

Tutorial elaboration on Week 9 lecture material, Single Page Application (SPA)

Dang Khoa Le

ID: 103844421

Introduction

This is the demonstration of Single Page Applications, or SPAs, with a particular focus on using Vue.js to develop these modern web applications. Here, we will discuss the SPA concept in depth, discuss why it's beneficial, analyse how SPA is applied using Vue.js with a detailed look on code example material from Week 9, and finally, I will elaborate on how we can use this model to create a basic web application such as social media feature (Messenger app).

Part 1: Understanding Single Page Applications (SPAs)

A. What is a Single Page Application?

A Single Page Application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach gives the feel of a desktop application where the user experiences smooth and fast navigation between different sections without full page reloads.
In an SPA, all necessary HTML, CSS, and JavaScript files are loaded once upon the initial page load. Subsequent interactions with the application only require data exchanges with the server via AJAX requests, typically receiving data in JSON format, which the application then uses to update parts of the page dynamically.

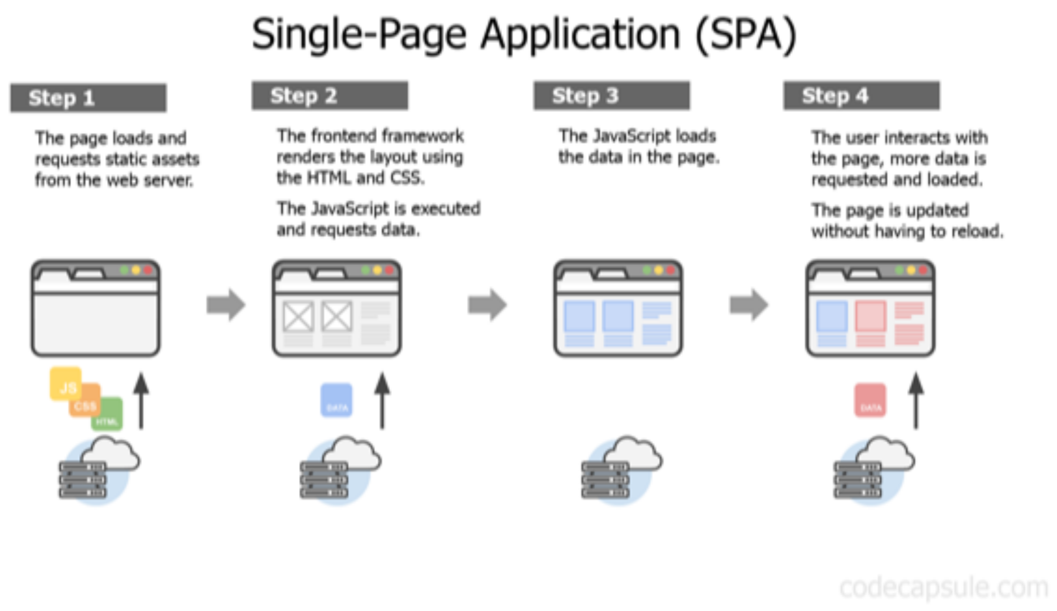

Single-Page Application (SPA)
Step 1
The page loads and requests static assets from the web server.
Step 2
The frontend framework renders the layout using the HTML and CSS.
The JavaScript is executed and requests data.
Step 3
The JavaScript loads the data in the page.
Step 4
The user interacts with the page, more data is requested and loaded.
The page is updated without having to reload.
JS
CSS
HTML
DATA
DATA
codecapsule.com

Figure. Introduce the concept of SPA design website.

B. How Does an SPA Work?

The SPA lifecycle begins when the user makes an initial request to the server:

1. **Initial Request:** The client sends a request to the server.
2. **Single Page Load:** The server responds with the single page and all necessary resources (HTML, CSS, JavaScript).
3. **Dynamic Updates:** Further interactions are handled by AJAX calls to fetch data from the server, which is then used to update parts of the page dynamically.

This process eliminates the need for full page reloads and significantly enhances the user experience by providing a faster and more responsive interface.

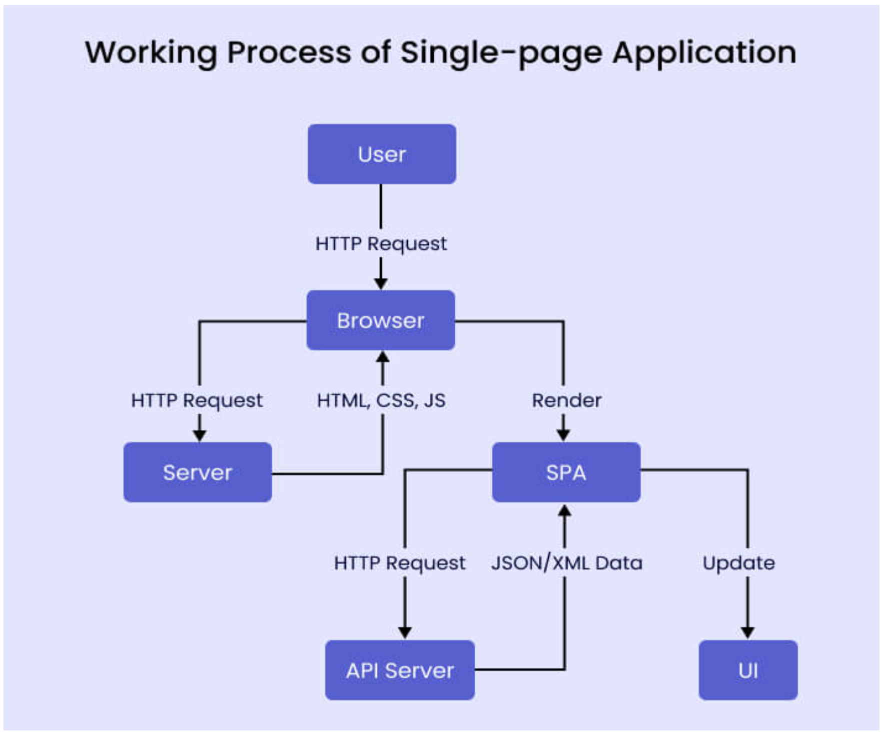


Figure. SPA concept working progress.

C. Benefits of SPAs

SPAs offer several key advantages over traditional multi-page applications (MPAs):

1. **Fast Loading Times:** Since the entire application is loaded once, subsequent interactions are faster as only data is exchanged, not complete HTML pages.
2. **Enhanced User Experience:** SPAs deliver a seamless experience akin to native mobile or desktop applications, making the user interface more interactive and engaging.
3. **Offline Capabilities:** SPAs can cache data and functionality, allowing some level of usage even without an internet connection.
4. **Reduced Server Load:** Because SPAs make fewer full-page requests, they reduce the load on the server, leading to better performance.

D. SPA Disadvantages

• Initial Load Time: SPAs often require a larger initial load, including JavaScript frameworks, libraries, and assets, which can result in a longer initial load time.

• SEO (Search Engine Optimization) Challenges: Since much of the content is loaded dynamically through JavaScript, search engines may have difficulty crawling and indexing the content effectively.

• Security Considerations: SPAs heavily rely on client-side scripts, so lots of application logic and data handling occur on the client-side. This can expose sensitive business logic and data, hackers may inject malicious scripts, attackers may gain unauthorized access.

E. SPAs vs. MPAs

Comparing SPAs to MPAs, we see distinct differences in how they handle client-server interactions:

- **MPA:** Each user action results in a new page load. The server returns a complete HTML document for every request, which can be slow and resource-intensive.
- **SPA:** The application initially loads a single HTML page. Subsequent interactions involve AJAX calls to the server, which returns data in JSON or XML format. The SPA then dynamically updates the page using JavaScript, which is much faster and provides a smoother user experience.

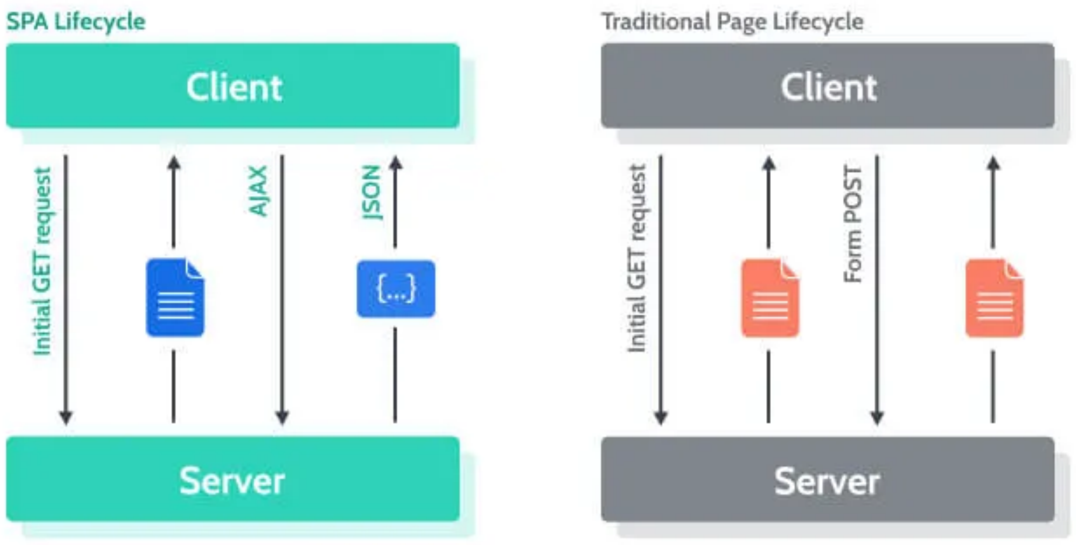
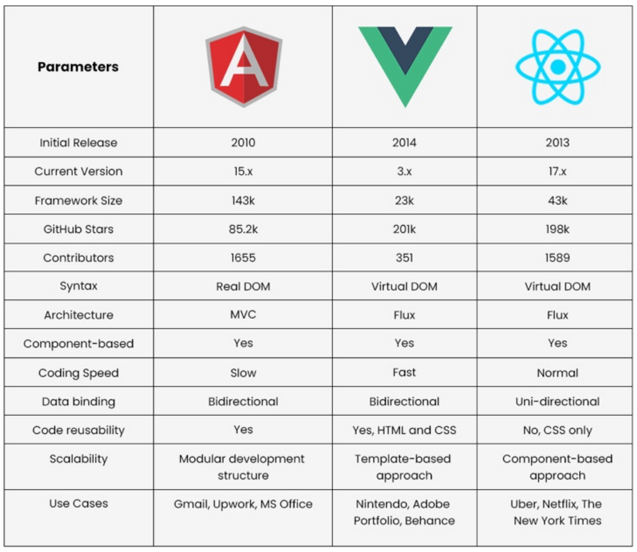


Figure. Differences between a SPA and MPA (Multiple Page Application) concept.

F. Frameworks for SPAs

When it comes to building Single Page Applications (SPAs), several frameworks stand out for their popularity and robust capabilities. The three most prominent ones are Angular, React, and Vue.js. Each of these frameworks has its own philosophy, strengths, and use cases:

- **Angular:** A feature-rich framework offering an all-in-one platform with everything built-in, suitable for large-scale applications.
- **React:** A minimalistic library focused on building user interfaces, where additional features are added through community-developed libraries.
- **Vue.js:** Strikes a balance between Angular and React, offering more features out-of-the-box than React but maintaining simplicity and ease of use.



Parameters			
Initial Release	2010	2014	2013
Current Version	15.x	3.x	17.x
Framework Size	143k	23k	43k
GitHub Stars	85.2k	201k	198k
Contributors	1655	351	1589
Syntax	Real DOM	Virtual DOM	Virtual DOM
Architecture	MVC	Flux	Flux
Component-based	Yes	Yes	Yes
Coding Speed	Slow	Fast	Normal
Data binding	Bidirectional	Bidirectional	Uni-directional
Code reusability	Yes	Yes, HTML and CSS	No, CSS only
Scalability	Modular development structure	Template-based approach	Component-based approach
Use Cases	Gmail, Upwork, MS Office	Nintendo, Adobe Portfolio, Behance	Uber, Netflix, The New York Times

Figure. Comparing key statistics between Angular, React and Vue.

G. Why Vue.js?

Vue.js is particularly compelling for SPA development due to several reasons:

- **Ease of Learning:** Vue.js is designed to be easy to learn and use. Its syntax is intuitive and straightforward, especially for developers who are already familiar with JavaScript, HTML, and CSS. This lowers the entry barrier and accelerates the learning curve.
- **Small and Fast:** Vue.js is lightweight and delivers high performance.
- **Component-Based:** It uses reusable components, which simplifies development and maintenance.
- **Comprehensive Documentation:** Vue.js comes with extensive and well-organized documentation, which includes numerous examples to help developers get started quickly.
- **Vue CLI:** The Vue CLI (Command Line Interface) helps in quickly setting up projects and streamlines the development process.

Now, let's see how Vue.js can be used to implement an SPA by analysing some example code.

Part 2: Analysing the Code

A. Vue.js and SPA

To understand how Vue.js is used to implement an SPA, let's analyse the provided code examples. The main features include routing, login functionality, pagination, and CRUD (Create-Read-Update-Delete) operations.

B. CSS Styling

Font and webpage styling with bootstrap definitions are defined in css files within the 'css' folder. These files are responsible to handle all website stylings with interactive interface. It is also a good programming practice for SPA concept web designs to store all css styling resources externally (instead of implementing directly to a HTML file), which can enhance maintainability and code interpreting for future usage.

C. Image Resource

The ‘img’ folder store 2 image content of ‘smiley1.png’ and ‘smiley2.png’ files.

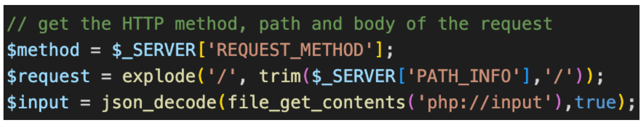
D. Data Storage

1. api_user.php

This PHP file handles user authentication by checking user credentials against the users table in the MariaDB database.

Functionality:

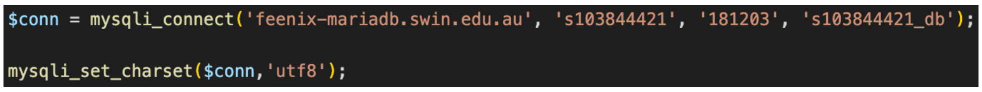
- HTTP Method and Request Parsing:



```php
// get the HTTP method, path and body of the request
$method = $_SERVER['REQUEST_METHOD'];
$request = explode('/', trim($_SERVER['PATH_INFO'],'/'));
$input = json_decode(file_get_contents('php://input'),true);
```

Retrieves the HTTP method (e.g. POST) and parses the request path and input data.

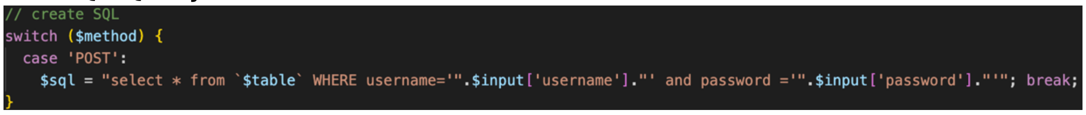
- Database connection:



```php
$conn = mysqli_connect('feenix-mariadb.swin.edu.au', 's103844421', '181203', 's103844421_db');

mysqli_set_charset($conn,'utf8');
```

Connects to the MariaDB database with specified credentials.

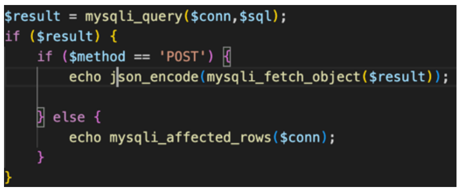
- SQL Query Construction:



```php
// create SQL
switch ($method) {
  case 'POST':
    $sql = "select * from `$table` WHERE username='".$input['username']."' and password ='".$input['password']."'"; break;
}
```

Constructs an SQL query based on the HTTP method. For POST requests, it checks if the provided username and password match a record in the users table.

- SQL Execution and Response:



```php
$result = mysqli_query($conn,$sql);
if ($result) {
    if ($method == 'POST') {
        echo json_encode(mysqli_fetch_object($result));

    } else {
        echo mysqli_affected_rows($conn);
    }
}
```

Executes the SQL query and returns the result as a JSON object for successful POST requests.

2. apis.php

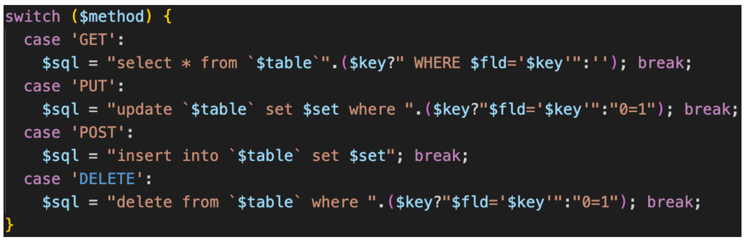
This PHP file manages CRUD (Create, Read, Update, Delete) operations on the idd_person table, handling user activity data.

Functionality:

- HTTP Method and Request Parsing:
- Database connection:

These 2 features utilise same method as api_user.php to retrieves HTTP methods and connect to the MariaDB database.

- SQL Query Construction:



```php
switch ($method) {
  case 'GET':
    $sql = "select * from `$table`".($key?" WHERE $fld='$key'":''); break;
  case 'PUT':
    $sql = "update `$table` set $set where ".($key?"$fld='$key'":"0=1"); break;
  case 'POST':
    $sql = "insert into `$table` set $set"; break;
  case 'DELETE':
    $sql = "delete from `$table` where ".($key?"$fld='$key'":"0=1"); break;
}
```

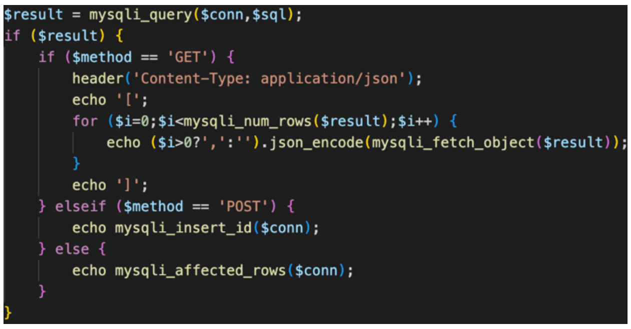
Constructs an SQL query based on the HTTP method.
GET request: Retrieve data from the **idd_person** table.
SELECT * FROM $table` `: Selects all columns from the specified table.
($key ? " WHERE $fld='$key'" : ''): If a key is provided in the request, it adds a WHERE clause to filter the results by the specified field and key value.

PUT request: Update existing data in the **idd_person** table
UPDATE $table SET $set: Updates the specified columns and values.
($key ? "$fld='$key'" : "0=1"): Adds a WHERE clause to update the record with the specified field and key value. If no key is provided, it defaults to 0=1, which will result in no rows being updated.

POST request: Insert new data into the **idd_person** table.
INSERT INTO $table SET $set: Inserts a new record with the specified columns and values.

DELETE request: Delete data from the **idd_person** table.
DELETE FROM $table` `: Deletes records from the specified table.
($key ? "$fld='$key'" : "0=1"): Adds a WHERE clause to delete the record with the specified field and key value. If no key is provided, it defaults to 0=1, which will result in no rows being deleted.

- SQL Execution and Response:

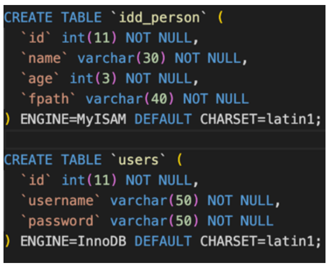


```
$result = mysqli_query($conn,$sql);
if ($result) {
    if ($method == 'GET') {
        header('Content-Type: application/json');
        echo '[';
        for ($i=0;$i<mysqli_num_rows($result);$i++) {
            echo ($i>0?',':'').json_encode(mysqli_fetch_object($result));
        }
        echo ']';
    } elseif ($method == 'POST') {
        echo mysqli_insert_id($conn);
    } else {
        echo mysqli_affected_rows($conn);
    }
}
```

Executes the SQL query and returns appropriate responses based on the HTTP method. For GET requests, it returns JSON data; for POST requests, it returns the insert ID.

3. db_person.sql
This SQL script initializes the database schema and inserts initial data for the **idd_person** and **users** tables.
Functionality:

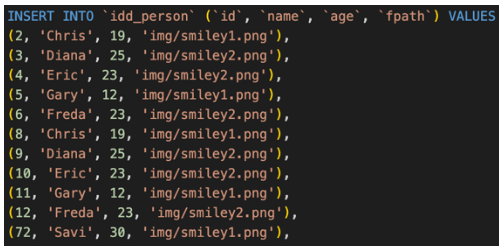
- Table Creation:



```
CREATE TABLE `idd_person` (
  `id` int(11) NOT NULL,
  `name` varchar(30) NOT NULL,
  `age` int(3) NOT NULL,
  `fpath` varchar(40) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(50) NOT NULL,
  `password` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Creates the idd_person and users tables with specified columns and data field types (e.g. integer, string, etc).

- Data Insertion:



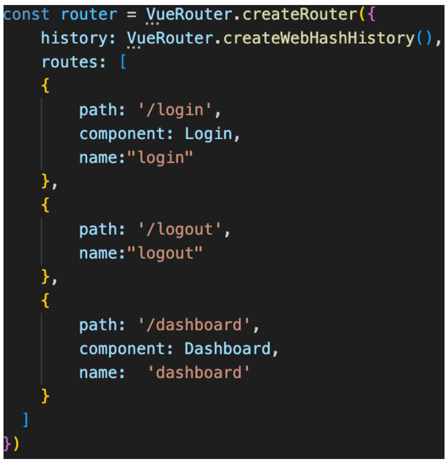
```
INSERT INTO `idd_person` (`id`, `name`, `age`, `fpath`) VALUES
(2, 'Chris', 19, 'img/smiley1.png'),
(3, 'Diana', 25, 'img/smiley2.png'),
(4, 'Eric', 23, 'img/smiley2.png'),
(5, 'Gary', 12, 'img/smiley1.png'),
(6, 'Freda', 23, 'img/smiley2.png'),
(8, 'Chris', 19, 'img/smiley1.png'),
(9, 'Diana', 25, 'img/smiley2.png'),
(10, 'Eric', 23, 'img/smiley2.png'),
(11, 'Gary', 12, 'img/smiley1.png'),
(12, 'Freda', 23, 'img/smiley2.png'),
(72, 'Savi', 30, 'img/smiley1.png'),
```

Inserts sample data into the idd_person table with entry fields of ‘id’, ‘name’, ‘age’ and ‘fpath’ (the path to the corresponding image file source).

E. Create VueRouter for Menu in SPA

Routing is essential in SPAs to manage different views or pages within the single-page context. Vue Router is the official router for Vue.js, which deeply integrates with Vue to make building SPAs a breeze. The router feature is implemented from **main.js**.

- **VueRouter**: This is used to create a router instance, which is essential for navigating between different components in a Vue SPA.
- **WebHashHistory**: This is a mode of history for the router which uses the URL hash to simulate a full URL so that the page won't be reloaded when the URL changes.
- **Routes**: Defined array of route objects, each containing ‘path’, ‘component’, and ‘name’. This sets up the paths that the application will respond to and the components that will be rendered.



```
const router = VueRouter.createRouter({
    history: VueRouter.createWebHashHistory(),
    routes: [
    {
        path: '/login',
        component: Login,
        name:"login"
    },
    {
        path: '/logout',
        name:"logout"
    },
    {
        path: '/dashboard',
        component: Dashboard,
        name:  'dashboard'
    }
  ]
})
```

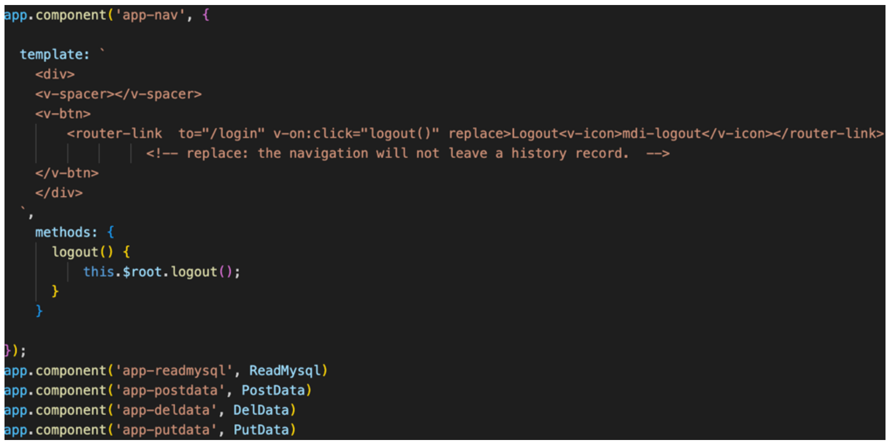
Here, we define routes for the ‘login’, ‘logout’, and ‘dashboard’ views. When the user navigates to a specific path, the corresponding component is rendered without reloading the page.

F. SPA Menu Component

The menu component uses Vue's capabilities to link routes and manage user interactions:

- <router-link>: This component is used to link to different routes within the application. It replaces traditional anchor tags (<a>) and provides navigation without page reloads.
- v-spacer: This is a Vuetify component used to create flexible spaces.
- v-btn: Vuetify button component allowing the login request to be sent.

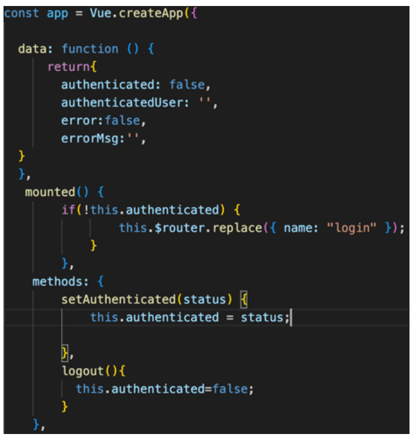
- logout(): Calls a logout function defined on the root instance ($root).



```
app.component('app-nav', {

  template: `
    <div>
    <v-spacer></v-spacer>
    <v-btn>
        <router-link  to="/login" v-on:click="logout()" replace>Logout<v-icon>mdi-logout</v-icon></router-link>
                  <!-- replace: the navigation will not leave a history record.  -->
    </v-btn>
    </div>
  `,
    methods: {
      logout() {
        this.$root.logout();
      }
    }

});
app.component('app-readmysql', ReadMysql)
app.component('app-postdata', PostData)
app.component('app-deldata', DelData)
app.component('app-putdata', PutData)
```

This component includes a logout button that, when clicked, calls the logout method and navigates to the login route using router-link.
Then, it create 4 Vue application components so the Menu can switch upon request on Post-Read-Put-Delete (CRUD) activities.

G. Login Component
The login component handles user authentication:
1. Initialisation:
From main.js, The Vue application is initialised using Vue.createApp and is defined with data properties, a lifecycle hook (mounted), and methods.



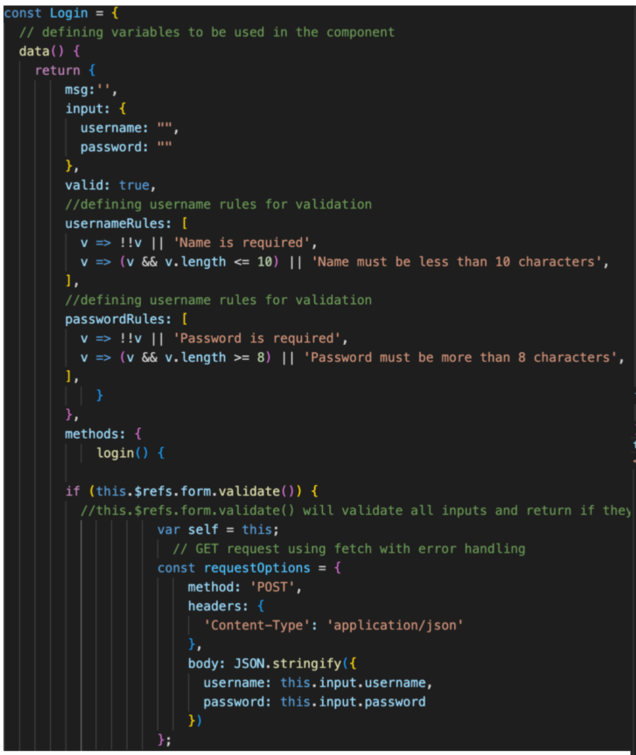
```
const app = Vue.createApp({

  data: function () {
      return{
        authenticated: false,
        authenticatedUser: '',
        error:false,
        errorMsg:'',
  }
  },
   mounted() {
        if(!this.authenticated) {
                this.$router.replace({ name: "login" });
            }
        },
    methods: {
        setAuthenticated(status) {
            this.authenticated = status;

        },
        logout(){
          this.authenticated=false;
        }
    },
```

Data Properties
- authenticated: A boolean to track if the user is authenticated. Default is false.
- authenticatedUser: A string to store the username of the authenticated user. Default is an empty string.
- error: A boolean to indicate if there is an error. Default is false.
- errorMsg: A string to store error messages. Default is an empty string.

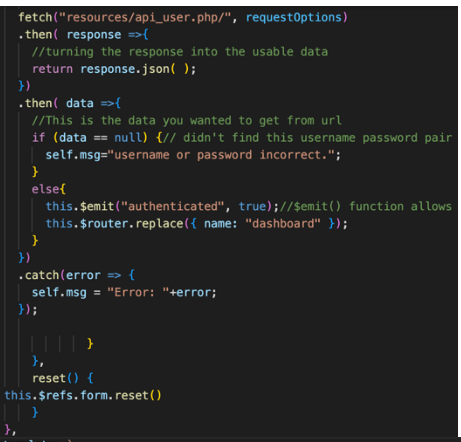
Lifecycle Hook
- mounted(): This is a Vue lifecycle hook that runs after the component is mounted (inserted into the DOM). It checks if the user is authenticated. If not, it redirects the user to the login page using Vue Router.

Methods
setAuthenticated(status): This method updates the authenticated status.
logout(): This method sets authenticated to false, effectively logging out the user.

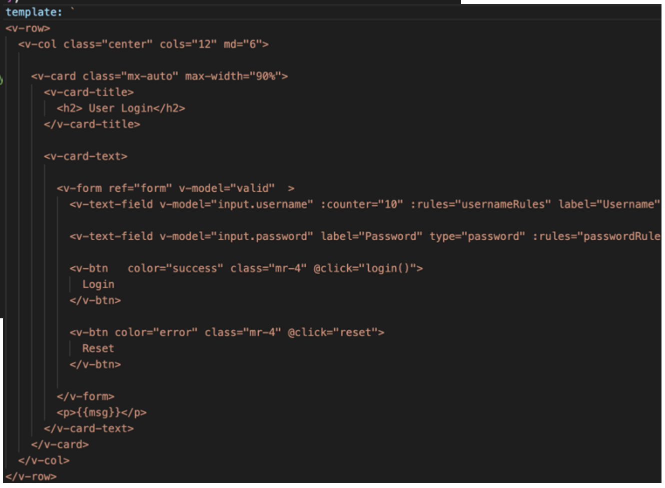
2. Login application:
This component uses a form to capture the username and password, then sends an AJAX POST request to the server for authentication. Upon successful login, it emits an authenticated event and redirects the user to the dashboard.



```
const Login = {
  // defining variables to be used in the component
  data() {
    return {
        msg:'',
        input: {
          username: "",
          password: ""
        },
        valid: true,
        //defining username rules for validation
        usernameRules: [
          v => !!v || 'Name is required',
          v => (v && v.length <= 10) || 'Name must be less than 10 characters',
        ],
        //defining username rules for validation
        passwordRules: [
          v => !!v || 'Password is required',
          v => (v && v.length >= 8) || 'Password must be more than 8 characters',
        ],
            }
        },
        methods: {
            login() {

        if (this.$refs.form.validate()) {
          //this.$refs.form.validate() will validate all inputs and return if they
                      var self = this;
                        // GET request using fetch with error handling
                      const requestOptions = {
                          method: 'POST',
                          headers: {
                            'Content-Type': 'application/json'
                          },
                          body: JSON.stringify({
                            username: this.input.username,
                            password: this.input.password
                          })
                      };
```



```
  fetch("resources/api_user.php/", requestOptions)
  .then( response =>{
    //turning the response into the usable data
    return response.json( );
  })
  .then( data =>{
    //This is the data you wanted to get from url
    if (data == null) {// didn't find this username password pair
      self.msg="username or password incorrect.";
    }
    else{
      this.$emit("authenticated", true);//$emit() function allows
      this.$router.replace({ name: "dashboard" });
    }
  })
  .catch(error => {
    self.msg = "Error: "+error;
  });

                }
    },
    reset() {
this.$refs.form.reset()
    }
},
```



```
template: `
<v-row>
  <v-col class="center" cols="12" md="6">

    <v-card class="mx-auto" max-width="90%">
      <v-card-title>
        <h2> User Login</h2>
      </v-card-title>

      <v-card-text>

        <v-form ref="form" v-model="valid"  >
          <v-text-field v-model="input.username" :counter="10" :rules="usernameRules" label="Username"

          <v-text-field v-model="input.password" label="Password" type="password" :rules="passwordRule

          <v-btn   color="success" class="mr-4" @click="login()">
            Login
          </v-btn>

          <v-btn color="error" class="mr-4" @click="reset">
            Reset
          </v-btn>

        </v-form>
        <p>{{msg}}</p>
      </v-card-text>
    </v-card>
  </v-col>
</v-row>
```

The Login component is defined with its own data properties and methods. This component is responsible for handling the user login functionality, including form validation and authentication requests to the server.

Data Properties

- msg: A string to store messages to be displayed to the user, such as errors or information.
- input: An object containing username and password properties that bind to the form inputs.
- valid: A boolean to track the validity of the form.
- usernameRules: An array of validation rules for the username field. The rules ensure that the username is not empty and does not exceed 10 characters.
- passwordRules: An array of validation rules for the password field. The rules ensure that the password is not empty and is at least 8 characters long.

Methods

login(): This method is called when the user submits the login form. It performs the following steps:

- Form Validation: Checks if the form is valid using this.$refs.form.validate().
- Prepare Request: Sets up a POST request with the username and password as JSON in the request body.
- Send Request: Sends the request to the server at the api_user.php endpoint.
- Handle Response: Processes the server response. If the response data is null, it means the username-password pair is incorrect, and an error message is shown. If the response data is valid, it emits an authenticated event and redirects the user to the dashboard.
- Error Handling: Catches any errors that occur during the request and sets an error message.

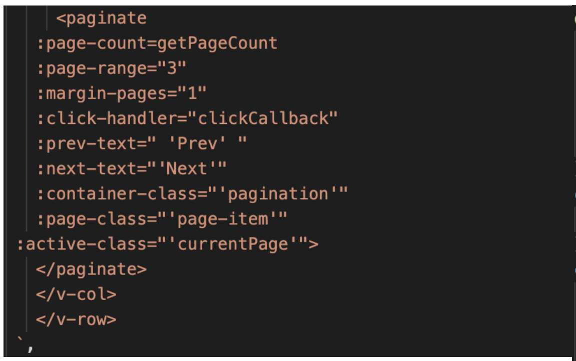
reset(): This method resets the form, clearing all the input fields and validation states.

Also, we have these events additionally to opt in functionalities:

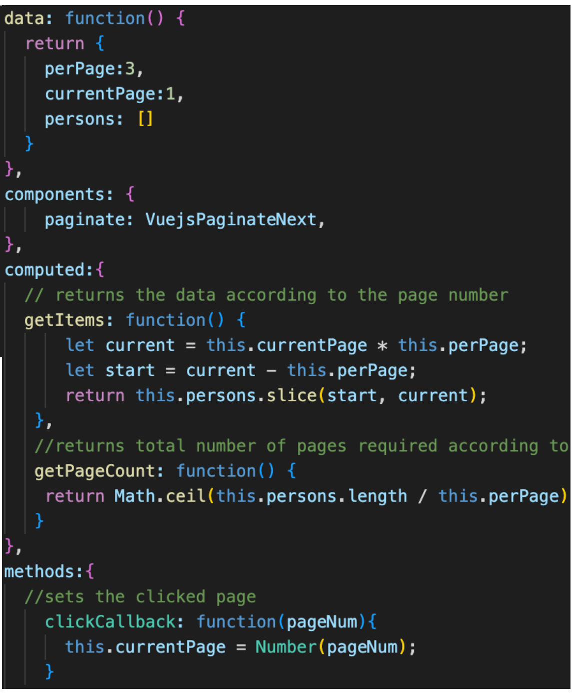
- Form Validation: The form validation is handled using the this.$refs.form.validate() method, which checks all inputs against their respective rules. If the form is valid, it proceeds with the login process.
- Custom Events: The $emit("authenticated", true) function emits a custom event named authenticated with the value true, indicating that the user has been successfully authenticated. This allows parent components to react to the login state change.
- Routing: The this.$router.replace({ name: "dashboard" }) line redirects the user to the dashboard route upon successful login.
- Template: Initialise the page design with form entry for username and password with appropriate error message display if requested.

H. Pagination in SPA

Pagination is crucial for managing large datasets in an SPA. The **vuejs-paginate** library simplifies this process.



```
    <paginate
  :page-count=getPageCount
  :page-range="3"
  :margin-pages="1"
  :click-handler="clickCallback"
  :prev-text=" 'Prev' "
  :next-text="'Next'"
  :container-class="'pagination'"
  :page-class="'page-item'"
:active-class="'currentPage'">
  </paginate>
  </v-col>
  </v-row>
`,
```



```
data: function() {
  return {
    perPage:3,
    currentPage:1,
    persons: []
  }
},
components: {
    paginate: VuejsPaginateNext,
},
computed:{
  // returns the data according to the page number
  getItems: function() {
      let current = this.currentPage * this.perPage;
      let start = current - this.perPage;
      return this.persons.slice(start, current);
   },
   //returns total number of pages required according to
   getPageCount: function() {
    return Math.ceil(this.persons.length / this.perPage)
   }
},
methods:{
  //sets the clicked page
    clickCallback: function(pageNum){
      this.currentPage = Number(pageNum);
    }
```

Here is the breakdown of pagination in details, elaborating on how pagination feature is implemented.

Template: Initialise the pagination page from **app-read.js** with Prev, Next and individual page buttons redirecting user to the requested page.

Data: Data component consist of perPage variable indicating the number of row per page, currentPage presents the initial page when loading (1) and the set of persons data fetched.

Paginate Component: This integrates pagination into the Vue component. It handles navigation between pages of data.

Computed Properties:

getItems: Determines the current set of items to display based on the current page and items per page.

getPageCount: Calculates the total number of pages needed.

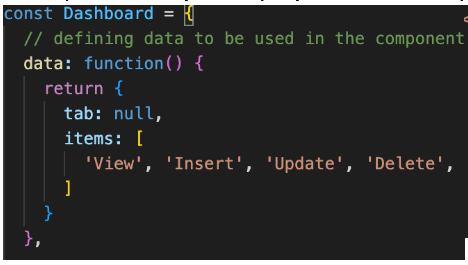
Methods:

clickCallback: Updates the current page when a pagination button is clicked.

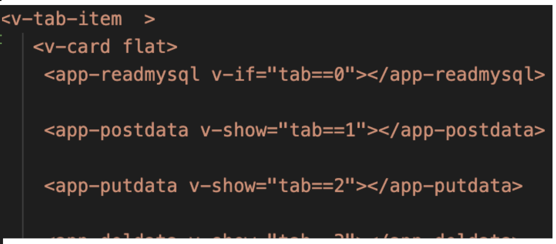
I. CRUD Operations

In this SPA design, we use CRUD operations (Create, Read, Update, Delete) for managing data and control activities.

Firstly, we have the Dashboard component to redirect user to call appropriate AJAX activities (JavaScript file) upon user request.



```
const Dashboard = {
  // defining data to be used in the component
  data: function() {
    return {
      tab: null,
      items: [
        'View', 'Insert', 'Update', 'Delete',
      ]
    }
  },
```

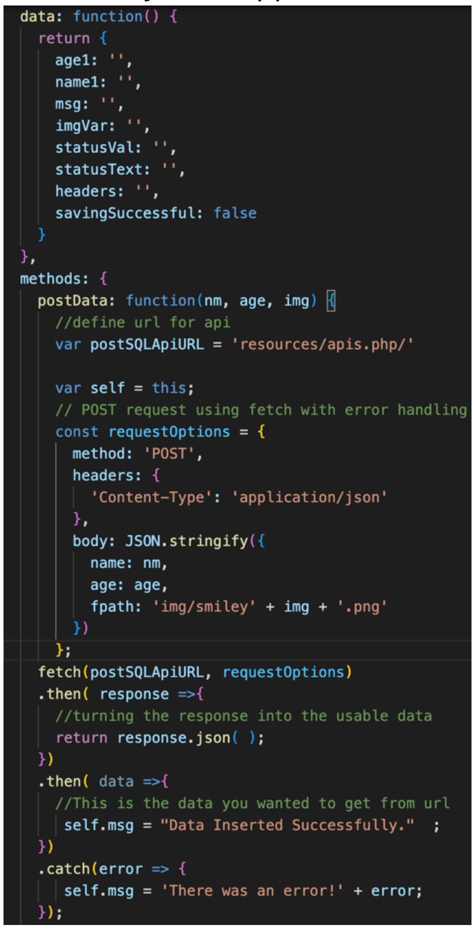


```
<v-tab-item  >
  <v-card flat>
   <app-readmysql v-if="tab==0"></app-readmysql>

   <app-postdata v-show="tab==1"></app-postdata>

   <app-putdata v-show="tab==2"></app-putdata>
```

Initially, the Vue app Dashboard defines 4 data items 'View', 'Insert', 'Update' and 'Delete', that upon clicking to any of these Vue tab item, it will call the corresponding Vue app (for instance, upon clicking to 'tab 1' or 'Insert', it calls the app-postdata.js file, which handle the create post feature). Now, let's examine the CRUD feature by each application.

1. Create (Insert, from app-postdata)

- **Template:** Design the website with form entries for Name, Age and Smiley Color tick-boxes to request user's input with output message and status indicating whether the POST attempt is successful or return with error.
- **Data Properties:** The component's data properties include age1, name1, msg, imgVar, statusVal, statusText, headers, and savingSuccessful which help in storing user input (age, name, imgVar), output messages, and status of the operation.
- **postData Method**: This method sends a POST request to the **apis.php** endpoint.
    - It constructs the request with the necessary headers and body, converting the input data to a JSON string.
    - fetch is used to send the request. If successful, it updates msg to inform the user that the data was inserted successfully and sets savingSuccessful to true. If there's an error, it updates msg with the error message.



```
data: function() {
  return {
    age1: '',
    name1: '',
    msg: '',
    imgVar: '',
    statusVal: '',
    statusText: '',
    headers: '',
    savingSuccessful: false
  }
},
methods: {
  postData: function(nm, age, img) {
    //define url for api
    var postSQLApiURL = 'resources/apis.php/'

    var self = this;
    // POST request using fetch with error handling
    const requestOptions = {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        name: nm,
        age: age,
        fpath: 'img/smiley' + img + '.png'
      })
    };
  fetch(postSQLApiURL, requestOptions)
  .then( response =>{
    //turning the response into the usable data
    return response.json( );
  })
  .then( data =>{
    //This is the data you wanted to get from url
     self.msg = "Data Inserted Successfully."  ;
  })
  .catch(error => {
     self.msg = 'There was an error!' + error;
  });
```

So, in brief, this component defines a form captures the name and age. When the user submits the form. The create method sends a POST request to the server, adding the new record to the database and displaying a output message.

2. Read (View, from app-readmysql)

This feature retrieves data from the database and displays it, with pagination support.

- **Template:** Construct a table with rows, columns and legends appropriately with data fetched from **apis.php** with pagination feature implemented.
- **Data Properties:** This component's data properties include perPage, currentPage, persons, and errorMessage. These properties manage the pagination and store the data fetched from the server.
- **Computed Properties:**

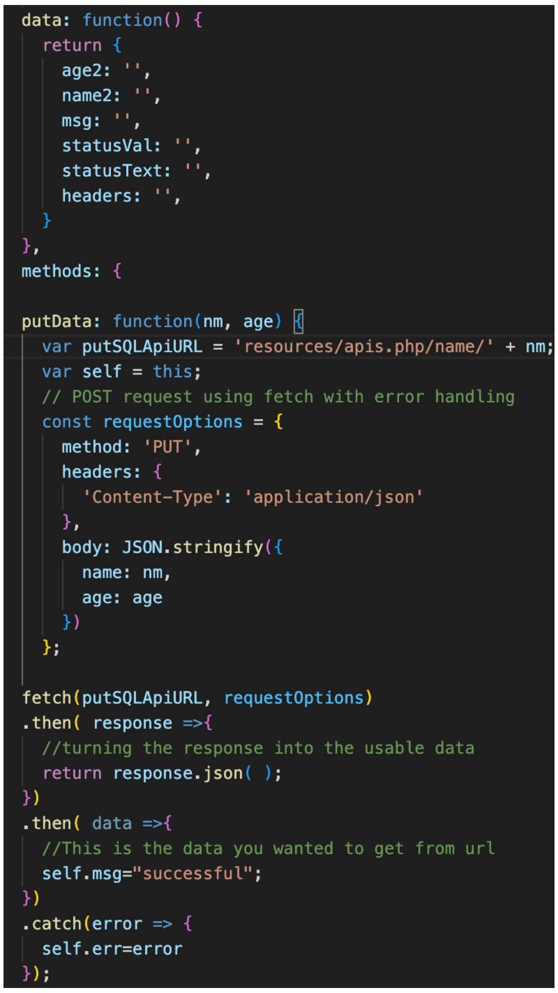
getItems:
getPageCount:

- **Methods:**

clickCallback: Updates the current page number when a pagination button is clicked.
created Hook: This lifecycle hook fetches the data from the server when the component is created.
It sends a GET request to the **apis.php** endpoint and updates the persons array with the fetched data. If an error occurs, it updates errorMessage with the error message.

3. Update (from app-putdata)

- **Template:** Design the website with form entries to allow user make changes to a particular person data based on their name, with the age entry to be updated, or print out the output message and status indicating whether the PUT attempt is successful or return with error.
- **Data Properties:** The component's data properties include age2, name2, msg, statusVal, statusText, and headers, which are used to store user input and status messages.
- **putData Method**: This method sends a PUT request to the **apis.php** endpoint with the specified name.
    - It constructs the request with the necessary headers and body, converting the input data to a JSON string.
    - fetch is used to send the request. If successful, it updates msg to inform the user that the update was successful. If there's an error, it updates err with the error message.



```
data: function() {
  return {
    age2: '',
    name2: '',
    msg: '',
    statusVal: '',
    statusText: '',
    headers: '',
  }
},
methods: {

putData: function(nm, age) {
  var putSQLApiURL = 'resources/apis.php/name/' + nm;
  var self = this;
  // POST request using fetch with error handling
  const requestOptions = {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      name: nm,
      age: age
    })
  };

fetch(putSQLApiURL, requestOptions)
.then( response =>{
  //turning the response into the usable data
  return response.json( );
})
.then( data =>{
  //This is the data you wanted to get from url
  self.msg="successful";
})
.catch(error => {
  self.err=error
});
```

4. Delete (from app-deldata)

- **Template:** Design the website with form entries allowing user to delete a particular data field based on the name input as well as print out the output message and status indicating whether the DELETE attempt is successful or return with error.

- **Data Properties**: The component's data properties include name3, msg, statusVal, statusText, and headers, which are used to store user input and status messages.
- **delData** Method: This method sends a DELETE request to the **apis.php** endpoint with the specified name.
  - It constructs the request with the necessary headers and body, converting the input data to a JSON string.
  - fetch is used to send the request. If successful, it updates msg to inform the user that the data was deleted successfully. If there's an error, it updates msg and statusText with the error message.

Summary
Each of these CRUD operations is implemented as a method in a Vue component. They use fetch to send HTTP requests to the **apis.php** endpoint (fetching the idd_person SQL table) with the appropriate method (POST, GET, PUT, DELETE). The server-side **apis.php** file handles these requests, interacts with the database, and returns the results. The client-side Vue components then update their state based on the server's response, providing feedback to the user on the success or failure of the operation.

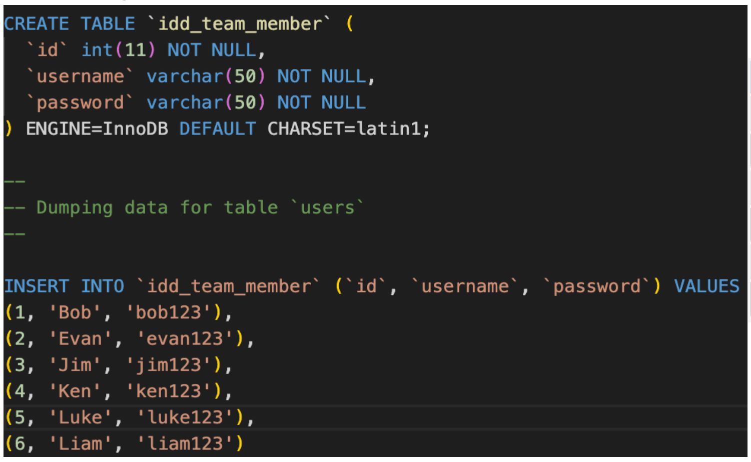
Part 3: Building a Social Media Feature with Vue.js
Now that we have covered the essential components and operations of an SPA, let's briefly discuss how these can be combined to build a basic web application.

As mentioned, SPA design websites can effectively enhance application platform performances as raising the user experience with a better and more optimized UI (User Interface). In this section, we will delve into implementation of how to apply SPA design by Vue.js to create a Messenger app, where user can Post-View-Update-Delete their text message while allowing multiple users participating together.

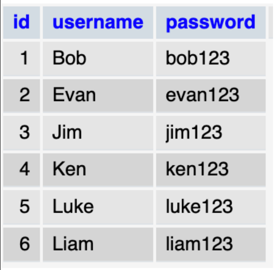
This design has effectively utilise the code sample from Week 9 materials, with multiple major changes to ensure seamless and effective user interaction.

Major Changes:

1. **User Authentication:** Create a SQL table **idd_team_member** to MariaDB containing a set of 'id', 'username', and 'password' for credential log in feature.

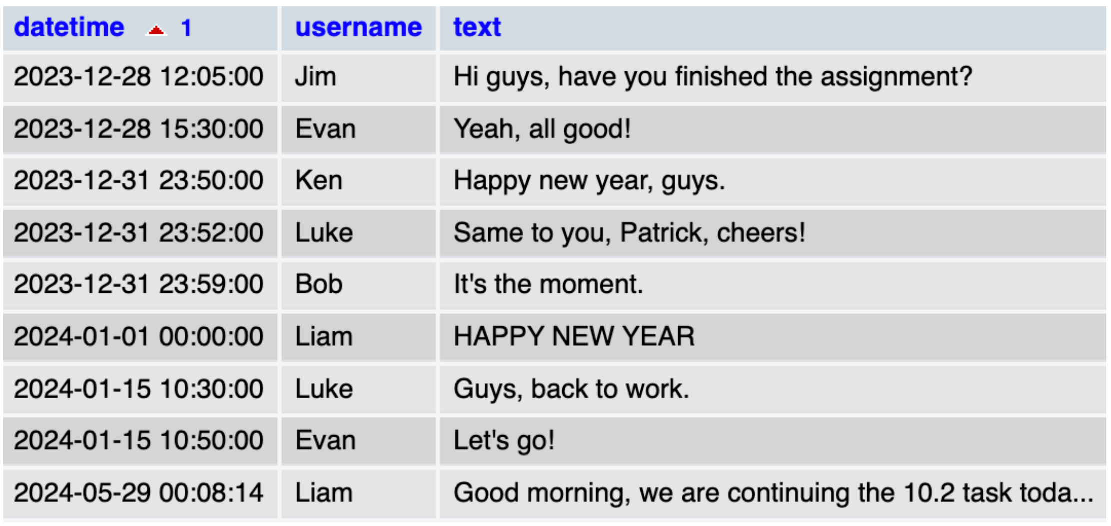


```
CREATE TABLE `idd_team_member` (
  `id` int(11) NOT NULL,
  `username` varchar(50) NOT NULL,
  `password` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `users`
--

INSERT INTO `idd_team_member` (`id`, `username`, `password`) VALUES
(1, 'Bob', 'bob123'),
(2, 'Evan', 'evan123'),
(3, 'Jim', 'jim123'),
(4, 'Ken', 'ken123'),
(5, 'Luke', 'luke123'),
(6, 'Liam', 'liam123')
```



id	username	password
1	Bob	bob123
2	Evan	evan123
3	Jim	jim123
4	Ken	ken123
5	Luke	luke123
6	Liam	liam123

2. **Data Management:** Create a SQL table **idd_messenger** to MariaDB, saving the ‘text’ message variable (string format) that user sent with ‘username’ (string format) and ‘datetime’ (SQL date time format) fields indicating the sender of that message and the time they post it.



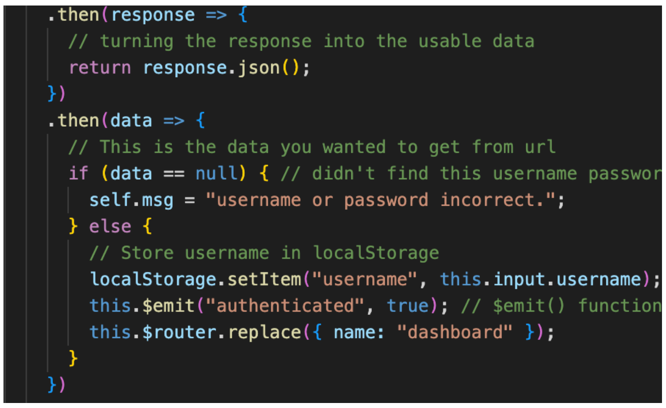
datetime ▲ 1	username	text
2023-12-28 12:05:00	Jim	Hi guys, have you finished the assignment?
2023-12-28 15:30:00	Evan	Yeah, all good!
2023-12-31 23:50:00	Ken	Happy new year, guys.
2023-12-31 23:52:00	Luke	Same to you, Patrick, cheers!
2023-12-31 23:59:00	Bob	It's the moment.
2024-01-01 00:00:00	Liam	HAPPY NEW YEAR
2024-01-15 10:30:00	Luke	Guys, back to work.
2024-01-15 10:50:00	Evan	Let's go!
2024-05-29 00:08:14	Liam	Good morning, we are continuing the 10.2 task toda...

Figure. idd_messenger SQL table.

3. **localStorage integration:**

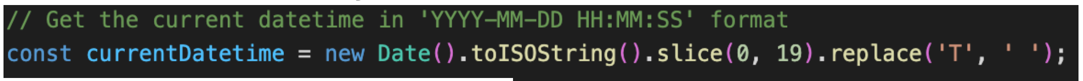
Upon user login attempt, if the username and password input to the field is authenticated from **api_user.php**, that username will be set to localStorage under the item ‘username’ for future usage.

Hence, when validated, I use the localStorage.setItem method to save the ‘username’ variable to localStorage, in order for that user to use this username for any CRUD operation without having to input their name to the form once again. This also opt in accuracy for data management upon CRUD activities.



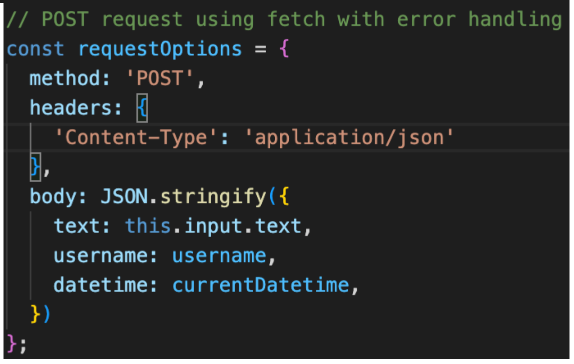
```
.then(response => {
  // turning the response into the usable data
  return response.json();
})
.then(data => {
  // This is the data you wanted to get from url
  if (data == null) { // didn't find this username passwor
    self.msg = "username or password incorrect.";
  } else {
    // Store username in localStorage
    localStorage.setItem("username", this.input.username);
    this.$emit("authenticated", true); // $emit() function
    this.$router.replace({ name: "dashboard" });
  }
})
```

4. **Current Datetime functionality:**



```
// Get the current datetime in 'YYYY-MM-DD HH:MM:SS' format
const currentDatetime = new Date().toISOString().slice(0, 19).replace('T', ' ');
```

Whenever user post a message, the currrentDatetime variable will automatically being created to set the current date-time data in YYYY-MM-DD HH:MM:SS” format along with the text message.



```
// POST request using fetch with error handling
const requestOptions = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    text: this.input.text,
    username: username,
    datetime: currentDatetime,
  })
};
```

Explanation:

- It create a new Date object, which set the current date-time data, for instance “May 29, 2024, 12:34:56.123 UTC”.

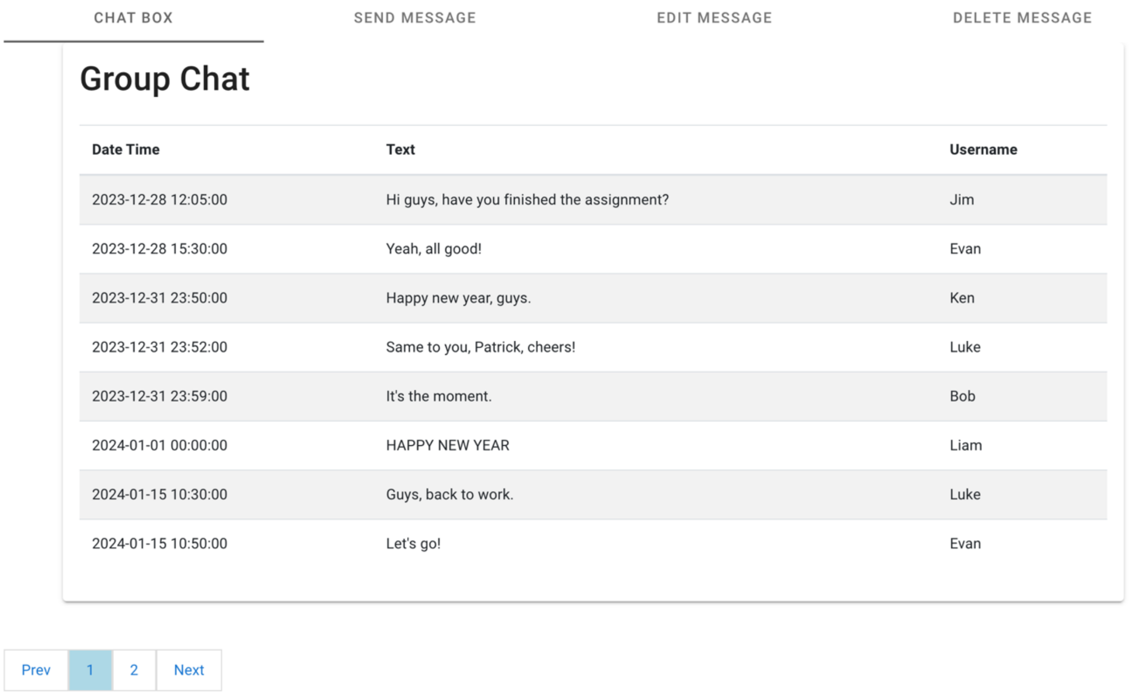
- Then it convert to ISO string, so the current format is "2024-05-29T14:34:56.123Z".
- Next, it slices the string from the 19th index to cut off the millisecond component and only use the first 19 character, so the current format is "2024-05-29T14:34:56".
- Finally, it replace the 'T' character with an empty string to format as "2024-05-29 14:34:56", which we obtain the desired datetime formatting.
- Eventually, while calling the POST request, the message will be saved to idd_messenger with the username and datetime variable formatted.

5. CRUD operations:

Firstly, it is mentionable that I have changed the View – Insert – Update - Delete items from Dashboard app into "Chat Box – Send Message – Edit Message – Delete Message" so that the UI design can be more understandable in this particular context, a Message app.

As this app design utilises a similar infrastructure as the Week 9 materials, hence, there are not much difference from the CRUD Interfacings.

We have the View (Chat Box) feature showing a set of "Text" message data as a table, with "Date Time" and "Username" data placed alongside. The data will be organized in FIFO order, by which mean showing the oldest message first (based on date-time order). We also apply pagination in this app design to allow user accessing the message faster without too much scrolling effort.



Group Chat

Date Time	Text	Username
2023-12-28 12:05:00	Hi guys, have you finished the assignment?	Jim
2023-12-28 15:30:00	Yeah, all good!	Evan
2023-12-31 23:50:00	Happy new year, guys.	Ken
2023-12-31 23:52:00	Same to you, Patrick, cheers!	Luke
2023-12-31 23:59:00	It's the moment.	Bob
2024-01-01 00:00:00	HAPPY NEW YEAR	Liam
2024-01-15 10:30:00	Guys, back to work.	Luke
2024-01-15 10:50:00	Let's go!	Evan

Figure. App View interface.

Then, we have the Create (Send Message) application, user just have to post their text message while the 'datetime' will be set using the **Current Datetime functionality**, and 'username' will be fetched from the localStorage by the 'username' item. We use the localStorage.getItem("username") method to do that.

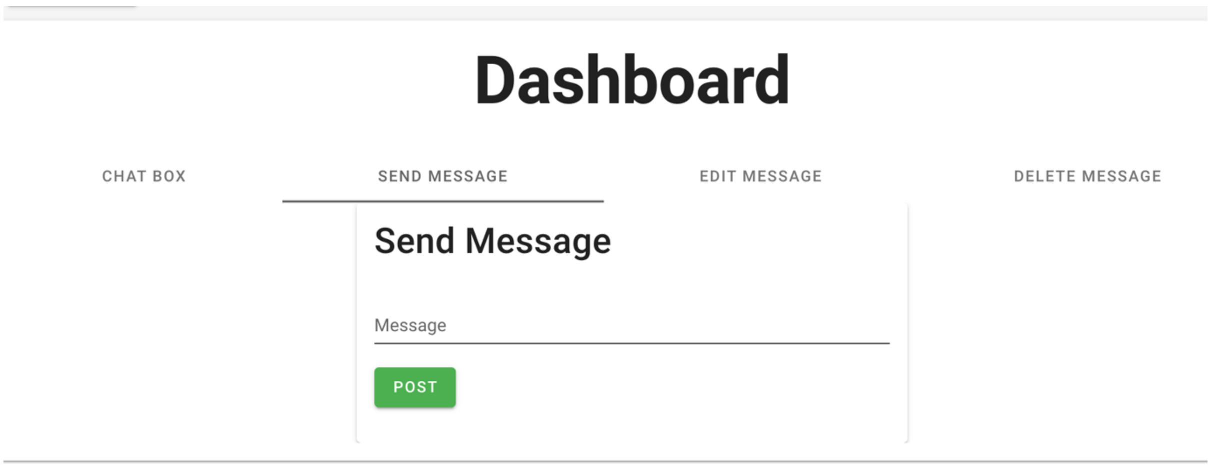
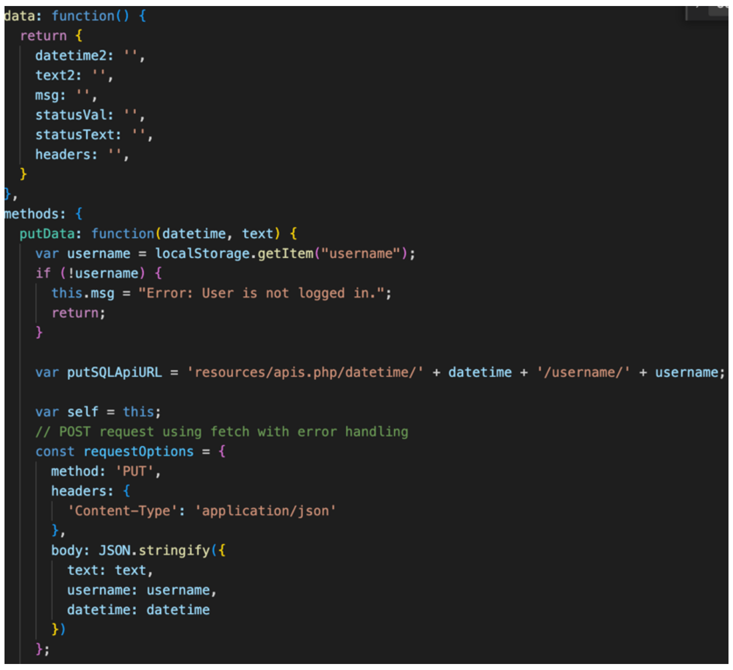


Figure. App Post interface.

Then, from app-putdata, we operate Update (Edit Message) functionality by checking for both 'username' (get from localStorage) and 'datetime' from the form input to target that specific message data from **apis.php** endpoint and perform the PUT request to edit the 'text' component.

While automatically set the 'username' variable from localStorage, we ensure that user can only edit the message that they are the sender, and not allowing them to edit the other user's message, hence, enhance user experience.

We also set the error debugger to check the 'username' variable is not set (although this cannot happens since only valid credential information allow user to log in and use this feature), in order to print out error message and preserve data consistency.



```
data: function() {
  return {
    datetime2: '',
    text2: '',
    msg: '',
    statusVal: '',
    statusText: '',
    headers: '',
  }
},
methods: {
  putData: function(datetime, text) {
    var username = localStorage.getItem("username");
    if (!username) {
      this.msg = "Error: User is not logged in.";
      return;
    }

    var putSQLApiURL = 'resources/apis.php/datetime/' + datetime + '/username/' + username;

    var self = this;
    // POST request using fetch with error handling
    const requestOptions = {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        text: text,
        username: username,
        datetime: datetime
      })
    };
```

Lastly, by similar to the Week 9 material, we also allowing output message and status to be printed out indicating whether the PUT request is successful or return error.

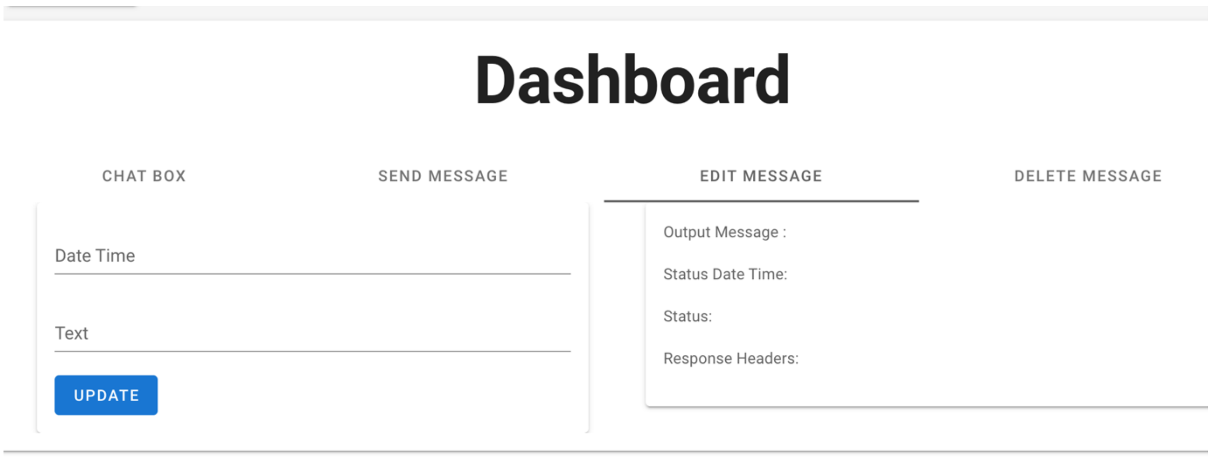
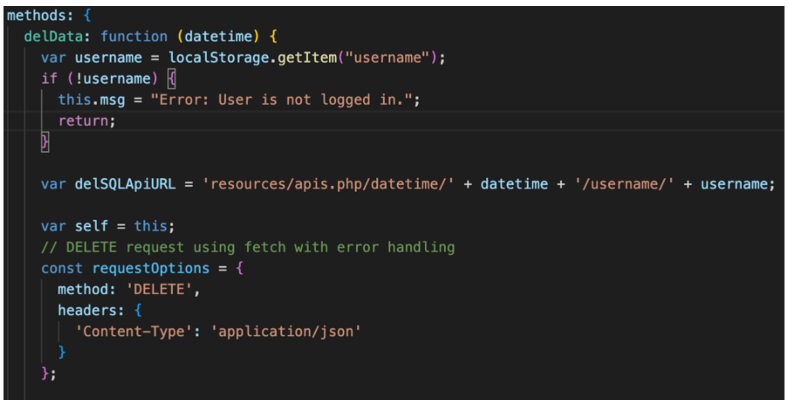


Figure. App Update interface.

And finally, we have the Delete (Delete Message) feature. In similar to app-putdata, the app-deldata AJAX component also check for that message's 'username' (from localStorage) and 'datetime' (from entry) to perform the DELETE request and permanently remove that specific message's data. Error message and status will also be sent upon systematic errors.



```
methods: {
  delData: function (datetime) {
    var username = localStorage.getItem("username");
    if (!username) {
      this.msg = "Error: User is not logged in.";
      return;
    }

    var delSQLApiURL = 'resources/apis.php/datetime/' + datetime + '/username/' + username;

    var self = this;
    // DELETE request using fetch with error handling
    const requestOptions = {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json'
      }
    };
```

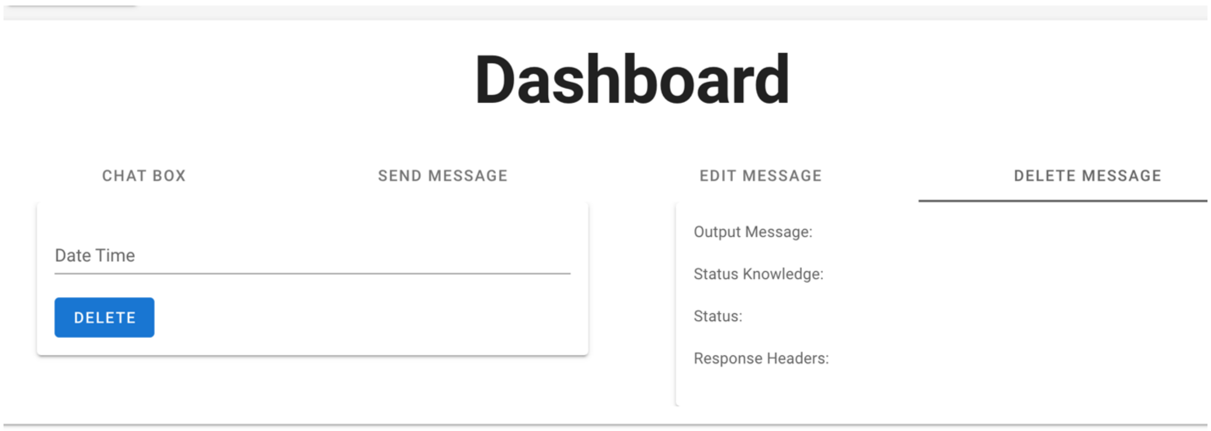


Figure. App Delete interface.

6. Instruction:

The Messenger App is accessible via
https://mercury.swin.edu.au/cos30043/s103844421/COS30043/Messenger%20App%20using%20SPA/

Upon logging to the Mercury server, please use these credential information:

Username: s103844421

Password: Kho@le2003

Presentation video can be accessed via https://www.youtube.com/watch?v=gKVB8mPfI9Q