Клієнтська частина побудована як односторінковий додаток (SPA).

Всі web-додатки поділяються на односторінкові (SPA) та багатосторінкові (MPA). SPA, Single Page Application, або «додаток однієї сторінки» – це тип веб-застосунків, в яких завантаження необхідного коду відбувається на одну сторінку. Це дозволяє заощадити час на повторне завантаження тих самих елементів.

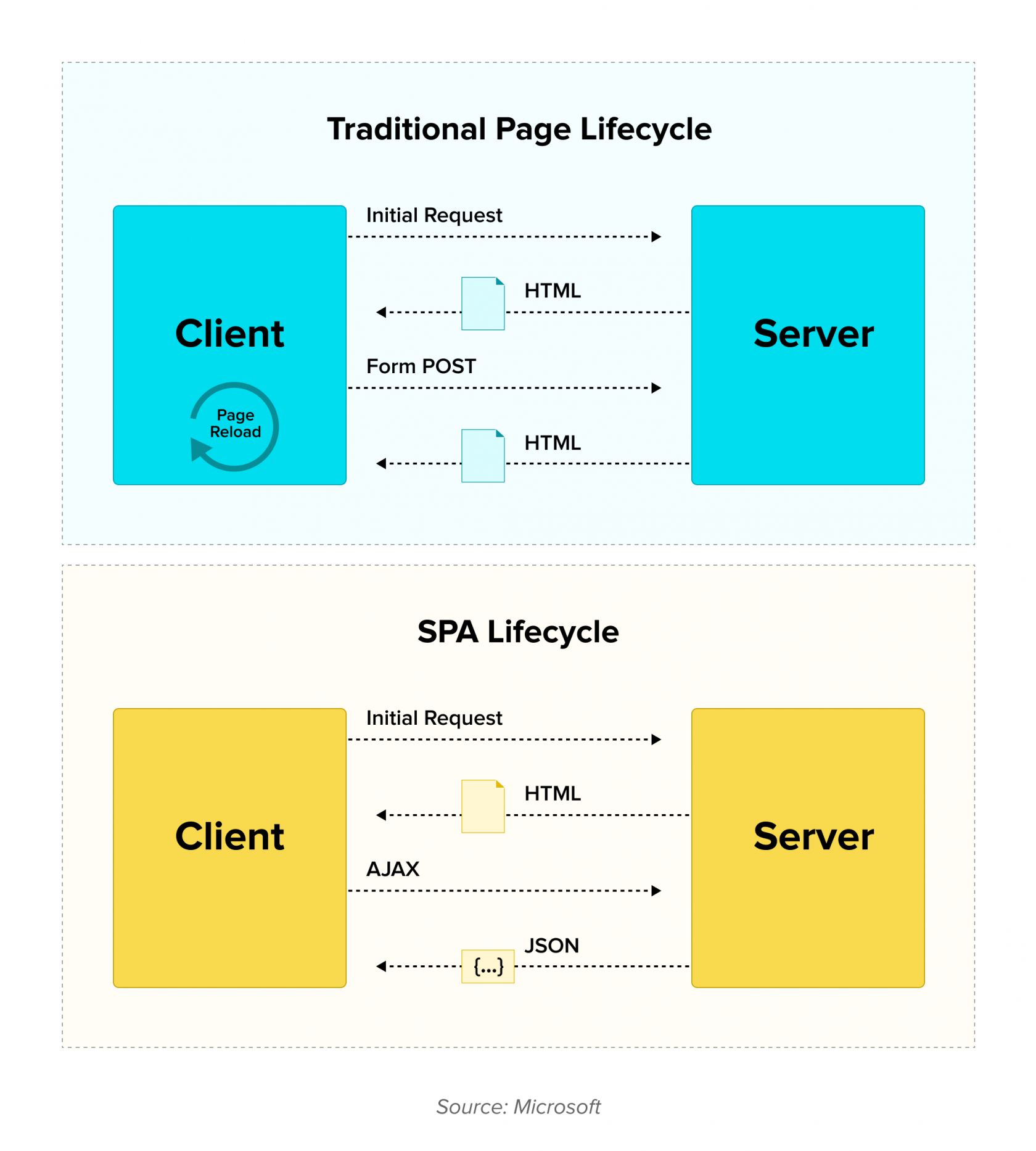
Багатосторінкова програма - це традиційна веб-програма, в якій із сервера запитується нова сторінка для відображення кожного разу, коли відбувається обмін даними туди і назад. Обсяг контенту, який вони несуть, величезний, тому вони, як правило, багаторівневі, зі значною кількістю посилань і складними інтерфейсами користувача.

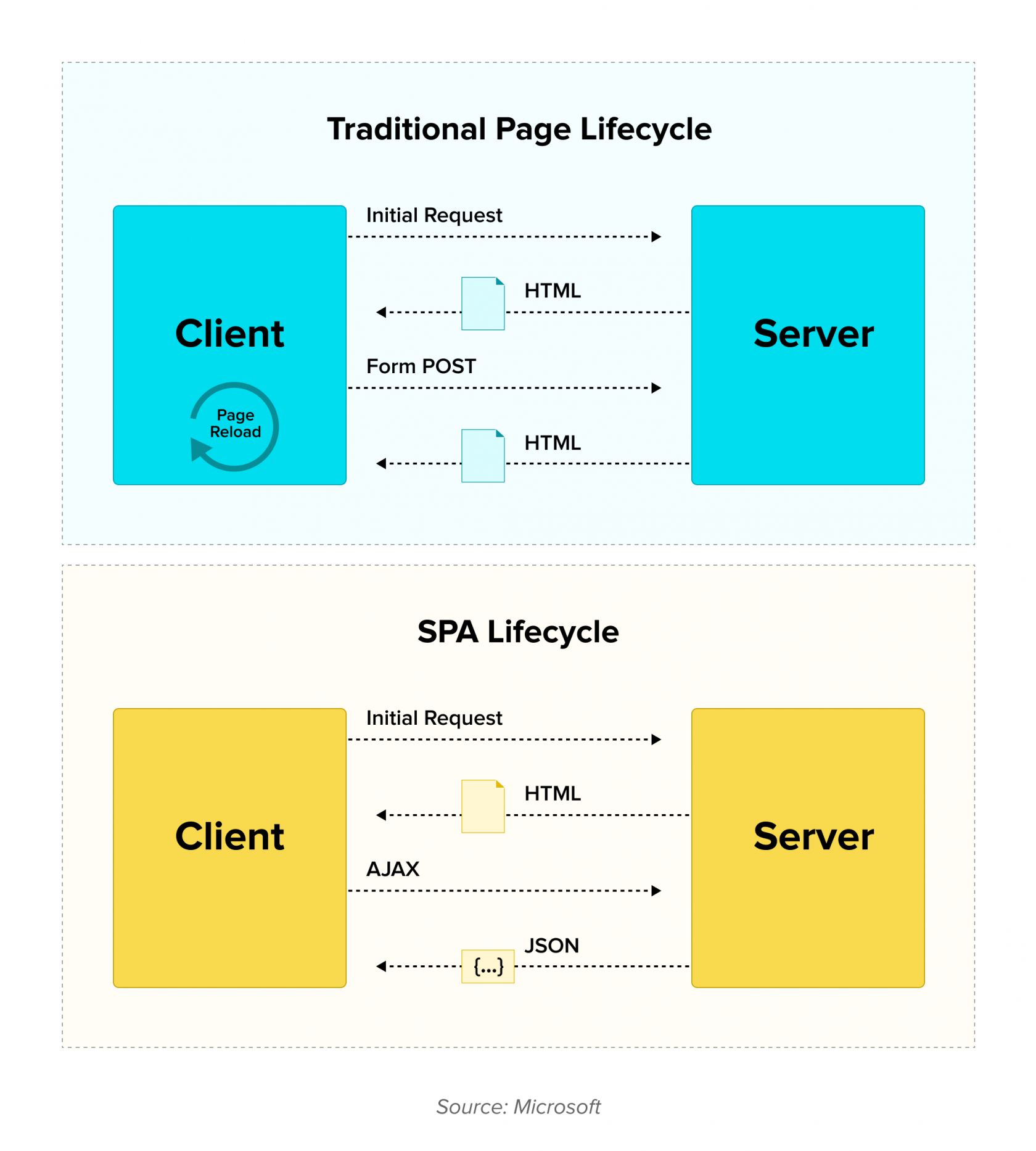
Програма SPA - це буквально одна сторінка, яка постійно взаємодіє з користувачем, динамічно переписуючи поточну сторінку, а не завантажуючи нові сторінки з сервера. Ми бачимо приклади односторінкових програм щодня: Trello, Facebook, Gmail, Twitter - ось кілька прикладів SPA.

Особливість архітектури SPA полягає в тому, що всі елементи, потрібні для роботи софту знаходяться на одній сторінці. Вони завантажуються під час ініціалізації. Також цей додаток завантажує додаткові модулі після запиту від користувача. Будь-яка активність користувача фіксується для зручності навігації. Це дозволяє скопіювати посилання та відкрити софт на тому самому етапі взаємодії на іншій вкладці, браузері або пристрої.

При завантаженні нових модулів в SPA контент на них оновлюється лише частково, тому що не потрібно повторно завантажувати постійні елементи. Це збільшує швидкість відповіді і скорочує обсяг даних, що передається між браузером і сервером.

Даний вид софту за способом взаємодії з користувачем найбільше нагадує роботу десктопних додатків, але дані розміщені на сервері.





**Переваги Single Page Applications**

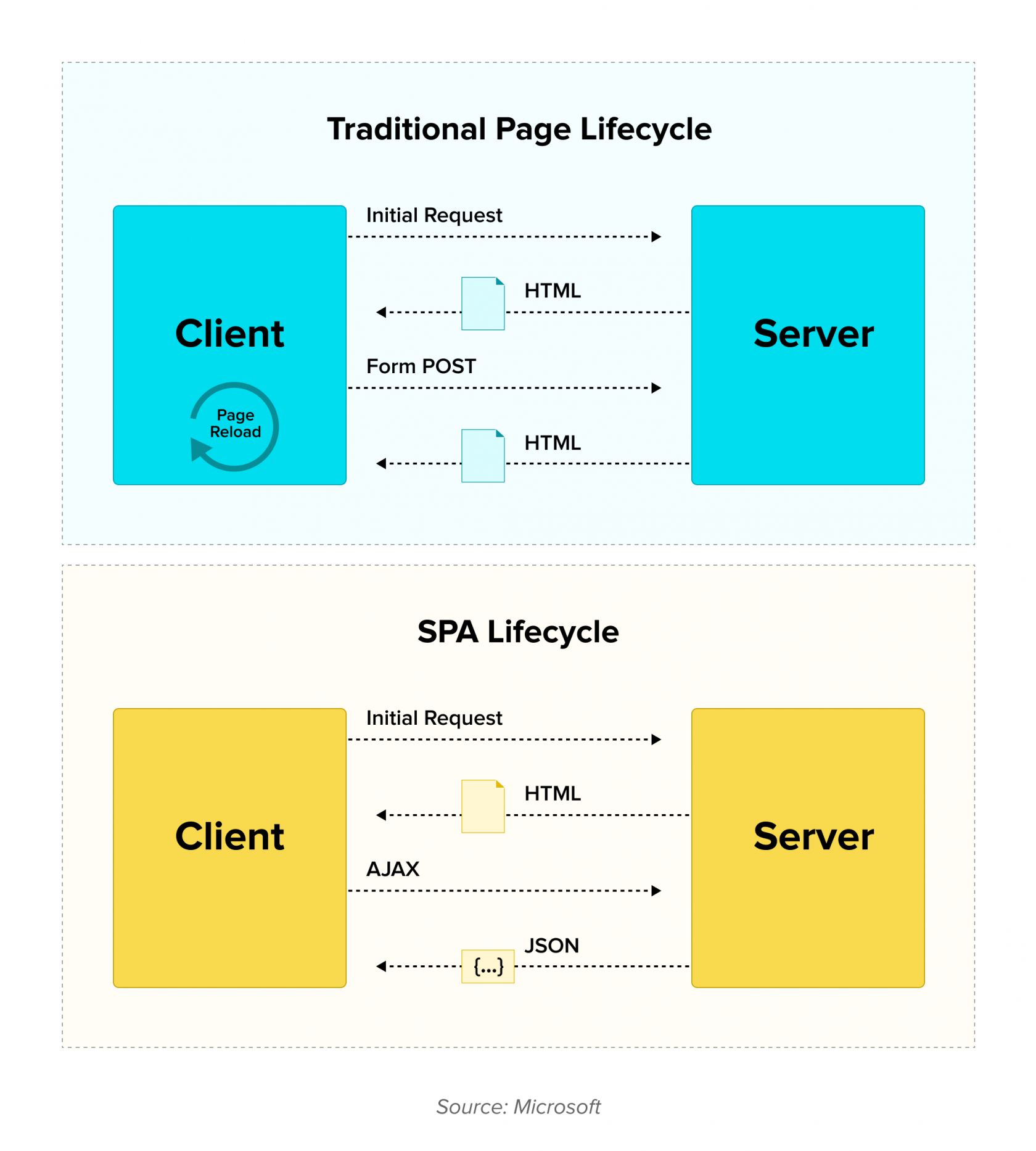
* Доступність . Можна отримати моментальний доступ до функціоналу з будь-якого типу пристрою без проблем сумісності, об'єму пам'яті, потужностей або часу на установку.
* Універсальність . Використовувати програму можна практично з будь-якого пристрою, якщо на ньому є доступ до інтернету. Якщо при розробці інтерфейсу враховувалися різні роздільні здатності екрану, то використовувати SPA однаково зручно і з ПК, і зі смартфона.
* Можливість задіяти великі обсяги даних . Розмір програми та даних, що використовуються, не обмежений пам'яттю пристрою.
* Швидкість . Одна сторінка з усім необхідним не тільки заощаджує час на повторне завантаження даних, а й підвищує продуктивність роботи.
* Можливості розробки . Розробникам доступні фреймворки, які спрощують створення архітектури проекту та надають чимало готових елементів для роботи.

**Недоліки SPA**

* **Необхідність інтернет-з'єднання** . Без доступу до мережі використовувати цей софт неможливо. Але якщо навіть десктопне програмне забезпечення використовує в роботі зовнішні бази даних, то доступ до інтернету необхідний у будь-якому випадку.
* **Проблеми з SEO** . Особливості SPA ускладнюють або унеможливлюють процес індексації пошуковими системами всіх модулів програми. Це може спричинити проблеми з оптимізацією.
* **Не працює у користувачів з відключеною підтримкою JS** . Багато хто відключає відображення JS-елементів у себе в браузерах, а Single Page Application використовує їх у роботі, тому може не працювати.

В нашому додатку односторінковий додаток доповнений можливістю обміну даними за протоколом Web Socket.

**SPA з підтримкою Web Socket**



Для реалізації web-додатку використано фреймворк Vue версії 2.6.11 [The Progressive JavaScript Framework.](https://vuejs.org)

Для забезпечення єдиного місця зберігання та обробки даних використовується [Vuex](https://v3.vuex.vuejs.org) версії 3.4.0 — це шаблон керування станом + бібліотека для додатків Vue.js. Він служить централізованим сховищем для всіх компонентів програми, з правилами, які гарантують, що стан можна змінювати лише передбачуваним чином.

Визначаючи та відокремлюючи концепції, пов’язані з управлінням станом, і дотримуючись правил, які підтримують незалежність між уявленнями та станами, ми надаємо нашому коду більшу структурність і можливість підтримки.

Це основна ідея Vuex, натхненна Flux, Redux і The Elm Architecture. На відміну від інших шаблонів, Vuex також є реалізацією бібліотеки, розробленою спеціально для Vue.js, щоб скористатися перевагами його детальної системи реактивності для ефективних оновлень.

Схематично це може бути відображено наступним чином.



Реалізоване таке єдине місце в файлах store\index.js та додаткових модулях store\auth.js, store\http.js та store\eventHandlers.js

Наш додаток представлено у вигляді 4 сторінок динамічних сторінок. Для переключення між сторінками використовується Vue Router версії 3.2.0 - [The official router for Vue.js.](https://router.vuejs.org) та реалізовано в файлі router/index.js наступним чином:

const routes = [

  {

    path: '/',

    name: 'Chat',

    component: () => import('../views/Chat.vue')

  },

  {

    path: '/login',

    name: 'Login',

    component: () => import('../views/Login.vue')

  },

  {

    path: '/register',

    name: 'Register',

    component: () => import('../views/Register.vue')

  },

  {

    path: '/contacts',

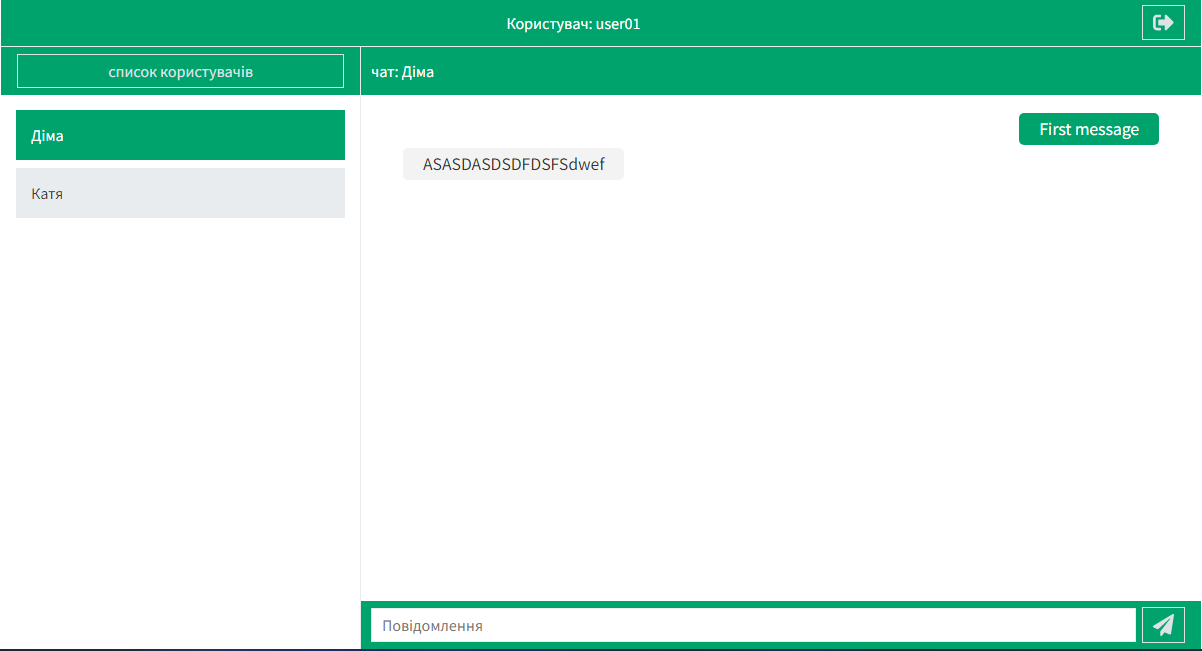
    name: 'Contacts',

    component: () => import('../views/Contacts.vue')

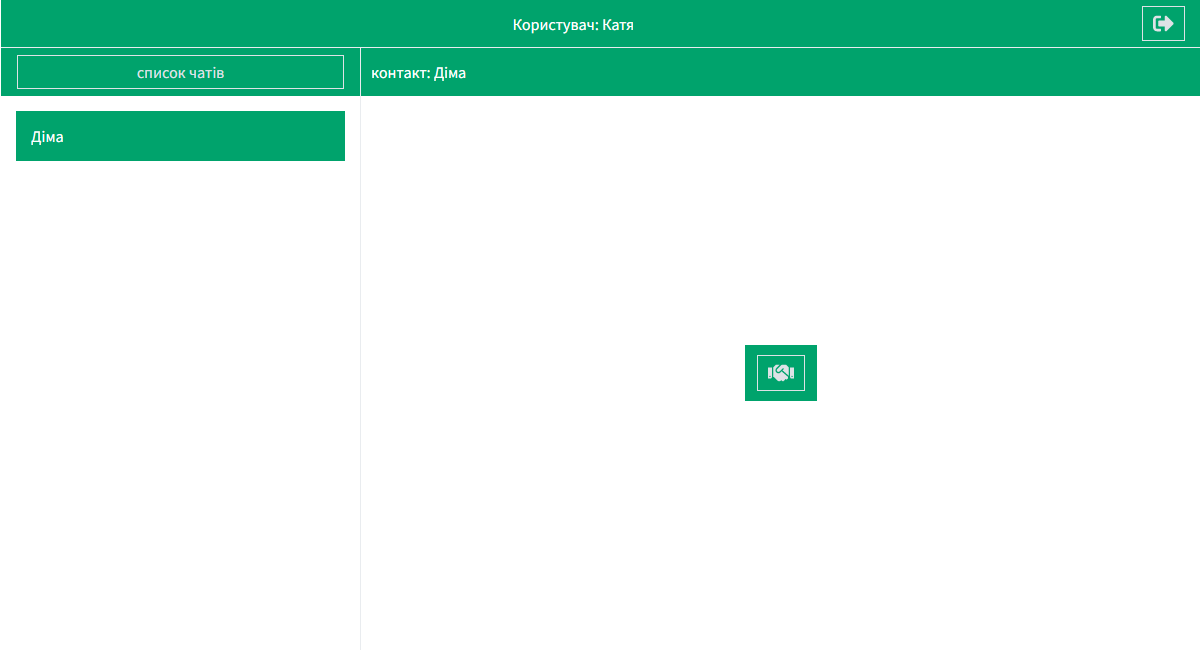
  }

];

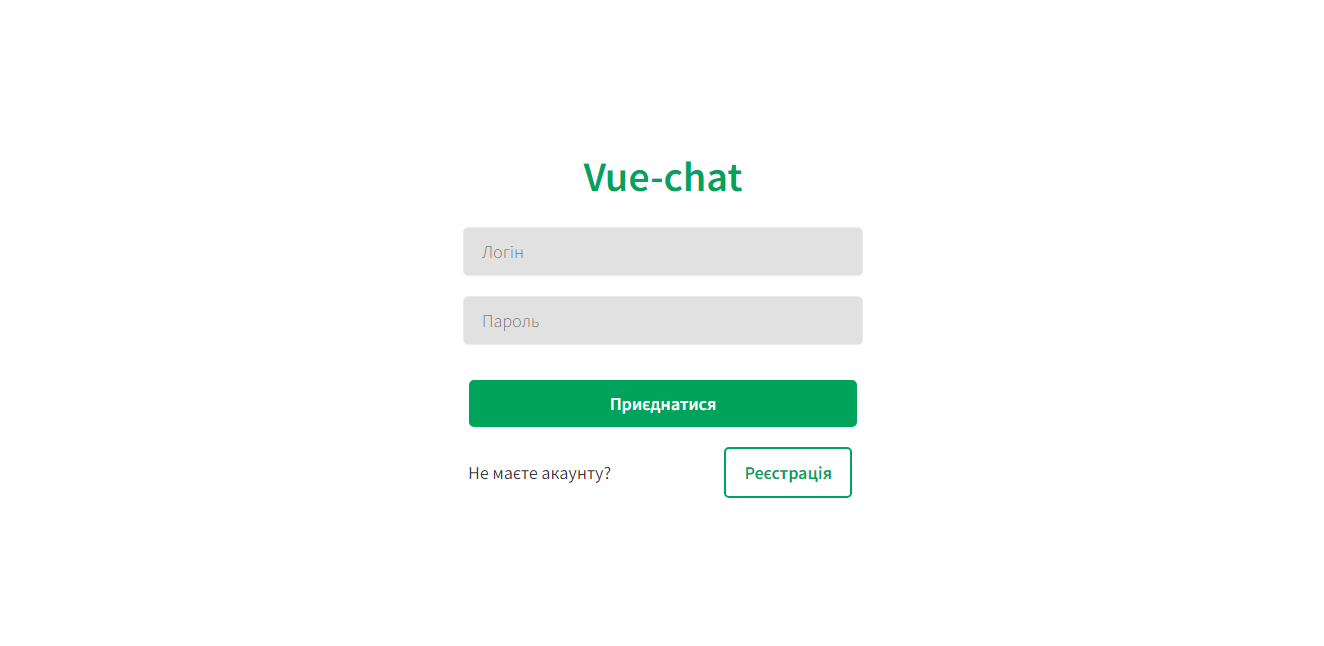
1. Головна сторінка додатку «Chat»



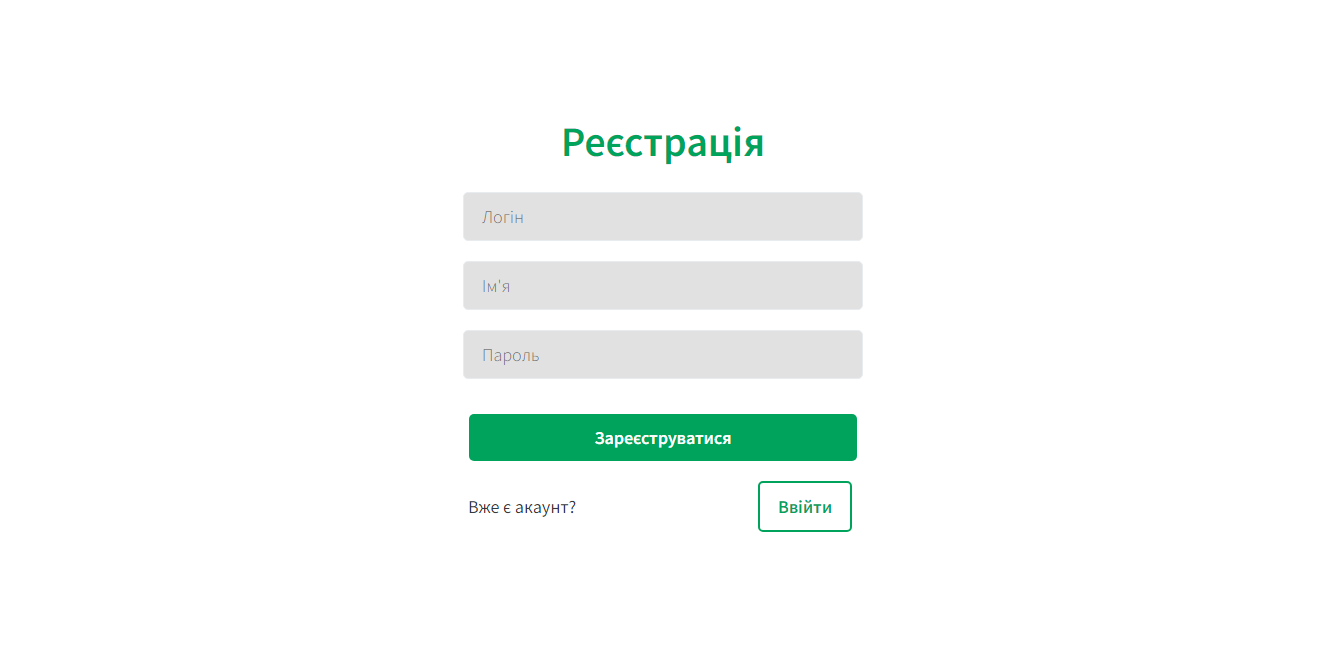
1. Сторінка вибору контакту для створення чату «Contacts»

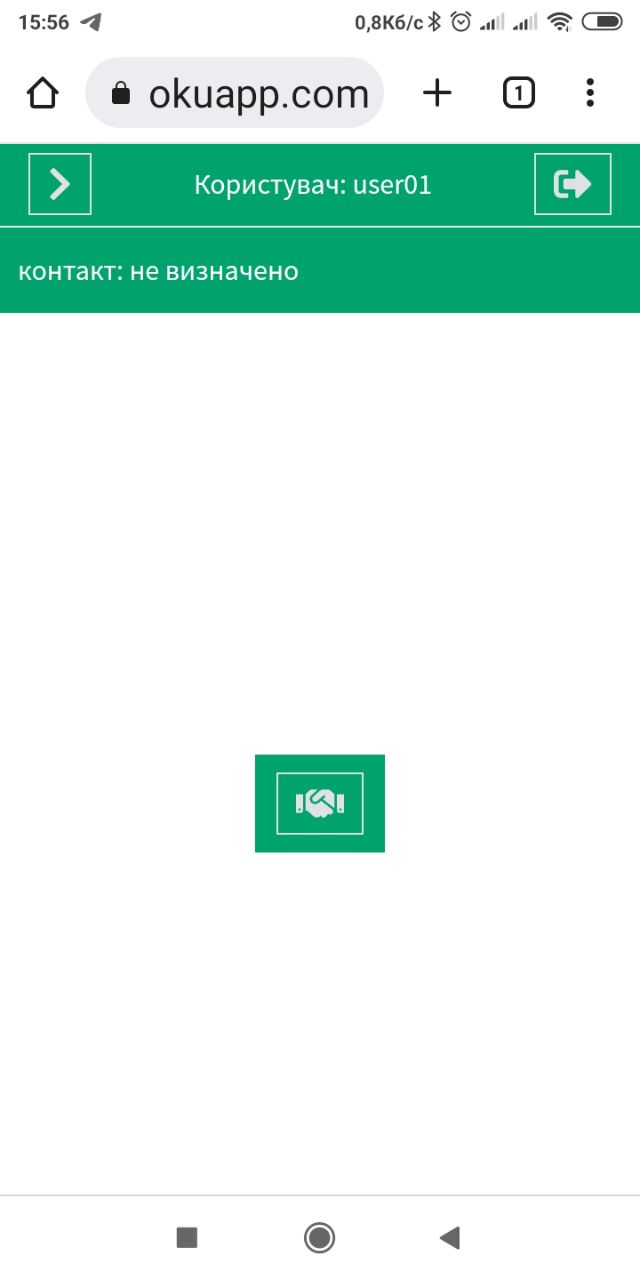
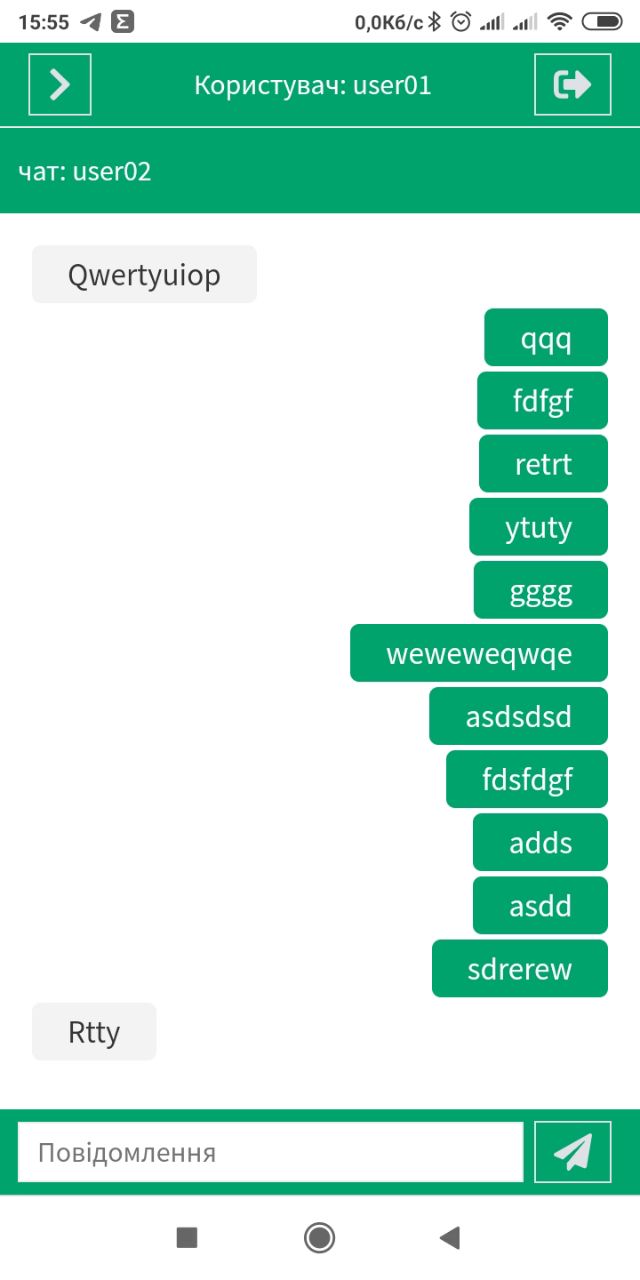


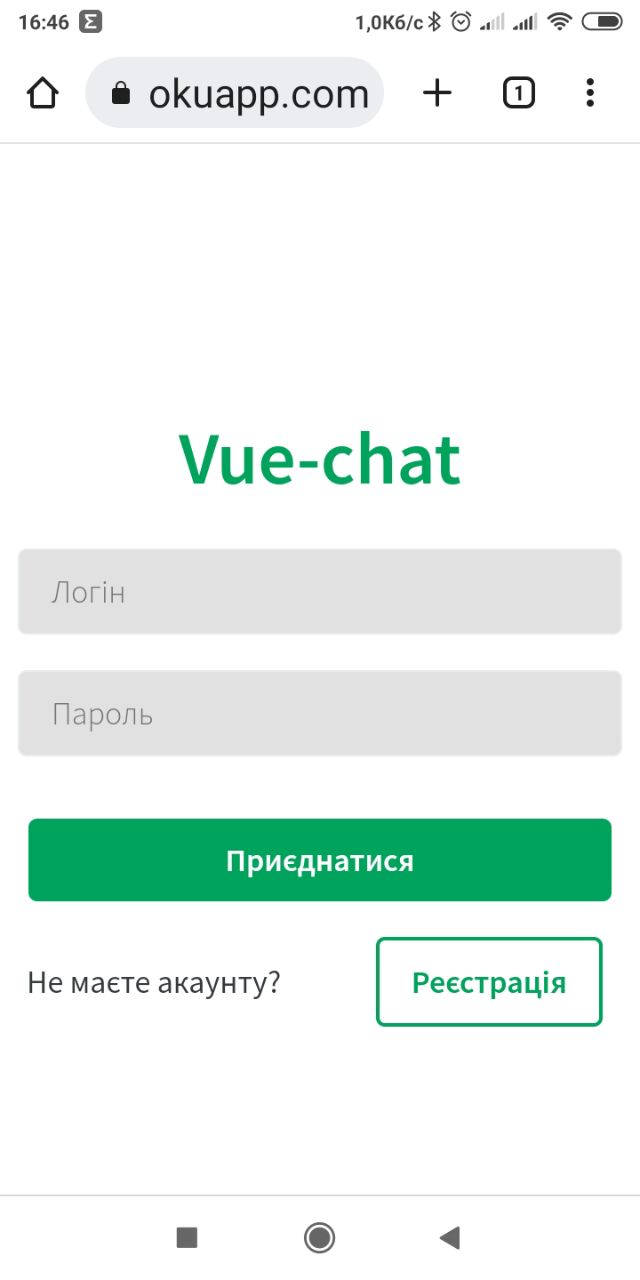
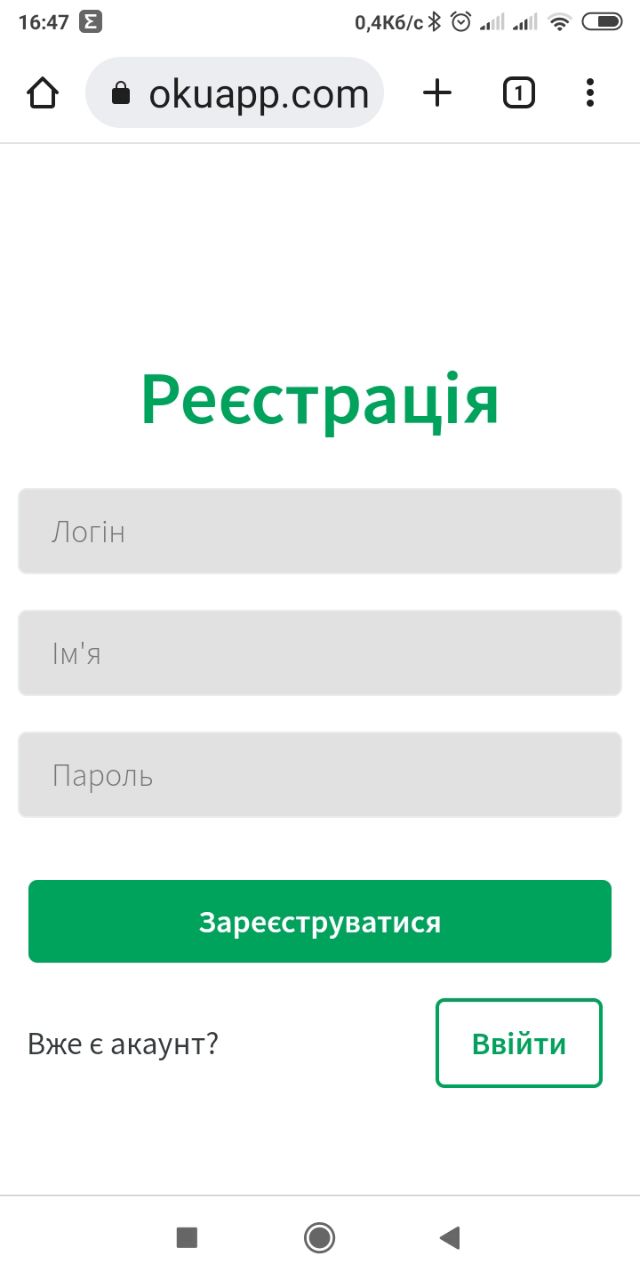
1. Сторінка аутентифікації в програмі «Login»



1. Сторінка реєстрації нового користувача «Register»



Ті самі сторінки, але на мобільному пристрої



Перші дві сторінки обмінюються з сервером повідомленнями за протоколом web socket, а дві другі за протоколом http.

Сервер налаштований таким чином, що всі повідомлення, які надходять до нього за протоколом http (метод GET) перенаправляються на головну сторінку. При цьому з сервера завантажується файл index.html і всі файли, які визначені в ньому.

Також відбувається аутентифікація користувача за ідентифікатором сесії в cookie, звісно, якщо він там є. Після цього відбувається спроба з’єднатися з сервером за протоколом web socket. Дані дії визначені методом mounted сторінки «Chat»

async mounted() {

    console.log('mounted');

    if (!this.socket) {

      await this.$store.dispatch('newConnection');

    }

    console.log(`this.room - ${JSON.stringify(this.room)}`);

    if (Object.keys(this.room).length !== 0) {

      console.log('change room on mounted');

      this.$store.dispatch('changeRoom', this.room.id);

    }

  }

та методом «'newConnection'» модуля store

async newConnection({ state, commit }) {

      console.log('New connection');

      const socket = io(window.location.href, {

        transports: ['websocket'],

      });

      socket.on('connect', () => {

        console.log(`on connect. Ідентифікатор сокету - ${socket.id}`);

        if (socket.connected) {

          console.log('socket connected');

          socket.emit('who am i', async (user) => {

            console.log(`who am i відповідь ${JSON.stringify(user)}`);

            commit('setUser', user);

          });

          commit('setSocket', socket);

        } else {

          console.log('З`єднання НЕ встановлено');

          commit('setSocket', null);

        }

        socket.on('rooms', (rooms) => onRooms(rooms, commit));

        socket.on('user', (user) => onUser(user, commit));

        socket.on('contacts', (contacts) =>

          onContacts(contacts, state, commit)

        );

        socket.on('message', (message) => onMessage(message, commit));

        socket.on('new chat', (sendRoom, isOwner, contact) => {

          onNewChat(sendRoom, isOwner, contact, commit, state);

        });

      });

      socket.on('connect\_error', async (error) =>

        onConnectError(error, socket, commit)

      );

      socket.on('disconnect', async (reason) => onDisconnect(reason, commit));

      socket.on('server error', (error) => onServerError(error));

    }

В методі 'newConnection' відбувається спроба встановити з’єднання з поточним сервером (дані беруться з window.location.href) за протоколом websocket. Для цього використовується клієнтська частина бібліотеки Socket.io. При цьому створюється змінна socket яка є екземпляром класу Socket. Socket є основним класом для взаємодії з сервером. Він успадковує більшість методів Node.js EventEmitter, наприклад emit, on, once або off, тобто генерацію події, прослуховування та обробку даних, які були відправлені під час генерації події. В нашій програмі встановлюються обробники наступних подій яки генеруються на сервері:

* сonnect – подія, яка відбувається у випадку успішного з’єднання з сервером;
* message – подія, яка виникає при отриманні повідомлення з сервера;
* user – подія, яка виникає, якщо сервер надсилає інформацію про поточного користувача в даному з’єднанні;
* rooms подія, яка виникає, якщо сервер надсилає перелік кімнат (чатів), в яких приймає участь поточний користувач;
* contacts – подія, коли сервер надсилає перелік можливих контактів для даного користувача (тобто перелік користувачів системи, з якими у даного користувача відсутні чати);
* new chat – подія, яка виникає в разі отримання інформації від сервера, що створений новий чат, учасником якого є поточний користувач;
* connect\_error – подія, яка виникає внаслідок помилки з’єднання;
* disconnect – подія, яка виникає, якщо з’єднання розірване;
* server error – подія, завдяки якій клієнт може отримати інформацію про помилкі, які виникли в процесі обробки повідомлень на сервері.

Більшість з цих повідомлень просто змінюють стан відповідних змінних на клієнті. Наприклад подія message просто додає нове повідомлення до переліку, а user встановлює інформацію про поточного користувача.

В той же час деякі з обробників подій потребують додаткових роз’яснень.

Головну подію сonnect ми вже розібрали. Нашої уваги потребують ще наступні події:

* new chat

function onNewChat(sendRoom, isOwner, contact, commit, state) {

  console.log(

    `on new chat.

      Room- ${JSON.stringify(sendRoom)},

      isOwner- ${isOwner},

      contact- ${JSON.stringify(contact)}`

  );

  commit('addRoom', sendRoom);

  commit('removeContact', contact);

  if (isOwner) {

    state.socket.emit('join', sendRoom.room\_id, (messages, roomId) => {

      console.log(`join відповідь повідомлення - ${JSON.stringify(messages)},

      кімната - ${roomId}`);

      const room = state.rooms.find((r) => r.room\_id === roomId);

      if (room) {

        commit('setRoom', room);

        commit('setMessages', messages);

      } else {

        commit('setRoom', {});

        commit('setMessages', []);

      }

    });

  }

  return;

}

Під час обробки цієї події додається новий чат до переліку чатів, доступних поточному користувачу (commit('addRoom', sendRoom);). Користувач, який є кореспондентом для цього чату видаляється з переліку контактів (commit('removeContact', contact);). Якщо поточний користувач є власником цього чату, тобто чат створювався з ініціативи поточного користувача, то користувач входить у цю кімнату і у відповідь з сервера отримує ідентифікатор кімнати та перелік повідомлень з цієї кімнати.

* connect\_error

function onConnectError(error, socket, commit) {

  console.log(`Відбулась помилка з'єднання: ${error.message}!`);

  if (error.message === 'unauthorized') {

    socket.disconnect(true);

  }

  commit('setConnectionError', error.message);

  return;

}

В разі, якщо відбулась помилка з’єднання, пов’язана з аунтифікацією з’єднання розривається. Крім того цей стан відслідковується на сторінках «Chat» та «Contacts» і відбувається переадресація на сторінку «Login».

connectionError() {

if (this.connectionError === 'unauthorized') {

  this.$router.push({ path: '/login' });

  }

}

* disconnect при виникненні цієї події необхідно скинути стан змінних до початкового

function onDisconnect(reason, commit) {

  console.log(`З'єднання розірване, причина - ${reason}.`);

  commit('setUser', {});

  commit('setContact', {});

  commit('setRoom', {});

  commit('setRooms', []);

  commit('setMessages', []);

  commit('setContacts', []);

  commit('setSocket', null);

  return;

}

Крім роботи за протоколом websocket в клієнтській частині реалізована робота за протоколом http в модулі store\http.js. В цьому модулі реалізований тільки 1 метод – request:

async request(

      { commit },

      { url, method = 'GET', body = null, headers = {} }

    ) {

      commit('setLoading', true);

      try {

        if (

          body &&

          headers['Content-Type'] &&

          headers['Content-Type'] == 'application/json'

        ) {

          body = JSON.stringify(body);

        }

        const response = await fetch(url, {

          method,

          credentials: 'include', // include, \*same-origin, omit

          body,

          headers

        });

        //console.dir(response);

        let data = await response.text();

        try {

          data = JSON.parse(data);

        } catch (error) {

          data = { message: data };

        }

        if (!response.ok) {

          throw new Error(data.message || 'Виникла якась помилка!');

        }

        return data;

      } catch (error) {

        console.dir(error);

        throw error;

      } finally {

        commit('setLoading', false);

      }

    }

  }

Відправка даних на сервер відбувається за допомогою методу fetch() інтерфейсу Fetch API браузера. Дані передаються в тілі повідомлення у вигляді JSON рядка.

Таким чином реалізовані 3 метода модуля store\auth.js

async registerUser({ dispatch, commit }, user) {

      /\* user - {login, username, password } \*/

      try {

        const newUser = await dispatch('request', {

          url: '/api/register',

          method: 'POST',

          body: user,

          headers: {

            'Content-Type': 'application/json',

          },

        }); // повертає 200 { id, login, username, state }

        commit('setCurrentUser', newUser);

        return;

      } catch (error) {

        console.log('registerUser error:', error);

        throw error;

      }

    },

    async loginUser({ dispatch, commit }, data) {

      /\* data - {login, password } \*/

      try {

        const user = await dispatch('request', {

          url: '/api/login',

          method: 'POST',

          body: data,

          headers: {

            'Content-Type': 'application/json',

          },

        }); // повертає { id, login, user\_name, state, created\_at, modified\_at }

        //console.log(user);

        commit('setUser', user);

        return;

      } catch (error) {

        console.log('loginUser error:', error);

        throw error;

      }

    },

    async logout({ commit, dispatch }) {

      try {

        await dispatch('request', {

          url: '/api/logout',

          method: 'POST',

          headers: {

            'Content-Type': 'application/json',

          },

        });

        commit('logout');

      } catch (error) {

        console.log('logout error:', error);

        throw error;

      }

    },

  }