В межах курсової роботи розроблена програма DBSHOP.

Мета проекту – знайомство та вивчення взаємодії програм написаних на мові програмування JavaScript з базами даних, зокрема СУБД MySQL.

Завдання проекту – розробити програму, що дозволяла вести блог, тобто надавала користувачам створювати власні статті, переглядати та коментувати статті інших користувачі.

Програма побудована на багаторівневій архітектурі взаємодії клієнт-сервер.



Рис. 1 схема багаторівневої архітектури взаємодії клієнт-сервер

Вікіпедія дає наступне визначення:

[Архітектура клієнт-сервер](https://uk.wikipedia.org/wiki/Клієнт-серверна_архітектура) є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних застосунків і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти:

* набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
* набір клієнтів, які використовують сервіси, що надаються серверами;
* мережа, яка забезпечує взаємодію між клієнтами та серверами.

Сервери є незалежними один від одного. Клієнти також функціонують паралельно і незалежно один від одного. Немає жорсткої прив'язки клієнтів до серверів. Більш ніж типовою є ситуація, коли один сервер одночасно обробляє запити від різних клієнтів; з іншого боку, клієнт може звертатися то до одного сервера, то до іншого. Клієнти мають знати про доступні сервери, але можуть не мати жодного уявлення про існування інших клієнтів.

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером. Логічно можна відокремити три рівні операцій:

* рівень представлення даних, який по суті являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
* прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;
* рівень управління даними, який забезпечує зберігання даних та доступ до них.

В нашому випадку використовується трирівнева архітектура. Трирівнева клієнт-серверна архітектура, яка почала розвиватися з середини 90-х років, передбачає відділення прикладного рівня від управління даними. Відокремлюється окремий програмний рівень, на якому зосереджується прикладна логіка застосунку. Програми проміжного рівня можуть функціонувати під управлінням спеціальних серверів застосунків, але запуск таких програм може здійснюватися і під управлінням звичайного веб-сервера. Нарешті, управління даними здійснюється сервером даних.

Для роботи з системою користувач використовує стандартне програмне забезпечення — звичайний браузер. Це позбавляє його необхідності завантажувати та інсталювати спеціальні програми (хоча інколи така необхідність все-таки виникає). Але користувачеві слід надати в розпорядженні інтерфейс, який дозволяв би йому взаємодіяти з системою і формувати запити до неї. Форми, що визначають цей інтерфейс, розміщуються на веб-сторінках та завантажуються разом з ними.

Веб-оглядач формує запит та пересилає його до сервера, який здійснює обробку. При необхідності сервер викликає серверні програмні модулі, які забезпечують обробку запиту і в разі потреби звертаються до сервера даних. Сервер даних здійснює операції з даними, що зберігаються в системі та складають її інформаційну основу. Зокрема, він може здійснити вибірку з інформаційної бази відповідно до запиту та передати її модулю проміжного рівня для подальшої обробки. Дані, з якими працює сервер даних, найчастіше організовані як реляційна база даних.

Для реалізації даного проекту були обрані наступні компоненти:

1. **Клієнт** являє собою односторінковий застосунок. [Односторінковий застосунок](https://uk.wikipedia.org/wiki/Односторінковий_застосунок) (англ. single-page application, SPA), також відомий як односторінковий інтерфейс (англ. single-page interface, SPI) - це веб-застосунок чи веб-сайт, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою. В односторінковому застосунку весь необхідний код - HTML, JavaScript, та CSS - завантажується разом зі сторінкою, або динамічно довантажується за потребою, зазвичай у відповідь на дії користувача. Сторінка не оновлюється і не перенаправляє користувача до іншої сторінки у процесі роботи з нею. Взаємодія з односторінковим застосунком часто включає в себе динамічний зв'язок з веб-сервером.

Для реалізації клієнта обраний прогресивний JavaScript-фреймворк Vue.js. [Vue визначає себе](https://ru.vuejs.org/v2/guide/%20) як прогресивний фреймворк для створення користувацьких інтерфейсів. На відміну від фреймворків-монолітів, Vue створений придатним для поступового впровадження. Його ядро в першу чергу вирішує завдання рівня уявлення (view), що спрощує інтеграцію з іншими бібліотеками та існуючими проектами. З іншого боку, Vue повністю підходить і для створення складних односторінкових додатків (SPA, Single-Page Applications), якщо використовувати його спільно з сучасними інструментами та додатковими бібліотеками. З метою розширення функціоналу до нашого застосунку додано наступні модулі:

* "vue-router" (версія 3.2.0),
* "vuelidate" (версія 0.7.6),
* "vuex": (версія 3.4.0)

та модулі, які спрощують розробку зовнішнього вигляду застосунку:

* "@fortawesome/fontawesome-free" (версія 5.15.1),
* "bootstrap" (версія 4.5.3),

Для організації зв'язку між клієнтом та сервером застосунку використовується технологія [Fetch API](https://developer.mozilla.org/ru/docs/Web/API/Fetch_API), яка підтримується всіма сучасними браузерами. В програмі за це відповідає метод request() модуля client\src\store\http.js.

    async request(

      { commit },

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

    ) {

      commit('setLoading', false)

      try {

        if (

          body &&

          headers['Content-Type'] &&

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

        ) {

          body = JSON.stringify(body)

        }

        if (this.state.auth.currentUser) {

          headers.Authorization = 'Bearer ' + this.state.auth.currentUser.uuid

        }

        const response = await fetch(url, { method, body, headers })

        const data = await response.json()

        if (!response.ok) {

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

        }

        commit('setLoading', false)

        return data

      } catch (error) {

        commit('setLoading', false)

        throw error

      }

    },

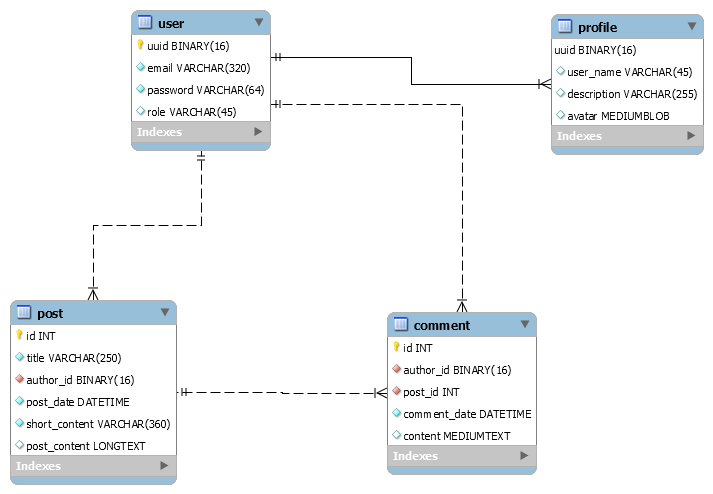
  }

1. **Сервер застосунку –** побудований на Express, який є швидким, гнучким, мінімалістичним веб-фреймворк для додатків Node.js. Додатково використані наступні модулі:

* "bcrypt" (версія 5.0.0),
* "config" (версія 3.3.3),
* "multer" (версія 1.4.2),
* "mysql2" (версія 2.2.5) – [драйвер по роботі з базою даних MySQL](https://www.npmjs.com/package/mysql2),
* "sharp" (версія 0.27.0),
* "uuid" (версія 8.3.2),
* "validator" (версія 13.5.2)

1. **Сервер даних** – СУБД MySQL, яка є найбільш популярною загальнодоступною системою управління базами даних SQL.

Модель бази даних



**Опис програми та взаємодія компонентів**

1. **При завантаженні клієнтської частини** програми в браузер виконується функція init() модуля main.js

async function init() {

  if (!localStorage.getItem('currentUser')) {

    store.commit('logout')

  } else {

    const uuid = JSON.parse(localStorage.getItem('currentUser'))

    try {

      const user = await store.dispatch('getUser', uuid)

      if (Object.keys(user).length !== 0) {

        store.commit('setCurrentUser', user)

      } else {

        console.log(`Користувач з uuid = ${uuid} не зареєстрований в програмі.`)

        store.commit('logout')

      }

    } catch (error) {

      console.log(error)

      store.commit('logout')

    }

  }

  new Vue({

    router,

    store,

    render: h => h(App),

  }).$mount('#app')

}

При цьому з локального сховища браузера (localstorage) зчитується ідентифікатор користувача програми, якщо він був там збережений. Після цього з сервера завантажується інформація про цього користувача з використанням методу getUser модуля store\auth.js

async getUser({ dispatch }, uuid) {

      try {

        const user = await dispatch('request', {

          url: `/api/auth/user/${uuid}`,

          method: 'GET',

        }) // повертає 200 user = { uuid: user.uuid, email: user.email, user\_name: user.user\_name, role: user.role, avatar: avatar } || {}

        return user

      } catch (error) {

        console.log('getUser error:', error)

        throw error

      }

    }

Метод getUser звертається до точки входу сервера застосунку за url `/api/auth/user/${uuid} з використанням http-методу ‘GET’. У відповідь отримує інформацію про користувача у вигляді об’єкта { uuid: user.uuid, email: user.email, user\_name: user.user\_name, role: user.role, avatar: avatar }, дані з якого використовуються в подальшій роботі програми. Цей об'єкт заноситься до поля currentUser об'єкту state модуля store\auth.js.

Точка входу /api/auth/user/${uuid} обробляється функцією у відповідному маршруті сервера застосунку (модуль routers\auth.routers.js)

/\*\*Повертає дані користувача за його uuid

 \* запит: метод - GET; шлях - /api/auth/user/:uuid

 \* uuid = req.params.uuid

 \* повертає статус - 200 та

 \* { uuid: user.uuid, email: user.email, user\_name: user.user\_name, role: user.role, avatar: user.avatar }

 \* або помилку 404 'Користувача не знайдено'

 \* або помилку 500

 \*/

router.get('/user/:uuid', async function (req, res) {

  const uuid = req.params.uuid

  try {

    const user = await User.getUser(uuid)

    if (Object.keys(user).length === 0) {

      res.status(404).send(

        JSON.stringify({

          source: `GET /api/auth/user/${req.params.uuid} error: `,

          message: 'Користувача не знайдено',

        })

      )

      return

    }

    res.send(JSON.stringify(user))

  } catch (error) {

    console.log(`GET /api/auth/user/${req.params.uuid} error: `, error)

    res.status(500).send(

      JSON.stringify({

        source: `GET /api/auth/user/${req.params.uuid}`,

        message: error.message,

      })

    )

  }

})

в цьому маршруті відбувається звернення до метода getUser(uuid) моделі User (модуль models\User.js)

  async getUser(uuid) {

    try {

      const connection = await db()

      const queryUser =

        'SELECT BIN\_TO\_UUID(usr.uuid) as uuid, usr.email, prf.user\_name, usr.role, prf.avatar FROM `' +

        dbName +

        '`.`user` as usr ' +

        'inner join `' +

        dbName +

        '`.`profile` as prf on prf.uuid = usr.uuid ' +

        'WHERE BIN\_TO\_UUID(usr.uuid) = ?'

      const [rows] = await connection.query(queryUser, [uuid])

      if (rows.length === 0) {

        return {}

      }

      return rows[0]

    } catch (error) {

      console.log('getUser error: ', error)

      throw error

    }

  }

В цьому методі здійснюється sql-запит до бази даних

SELECT

BIN\_TO\_UUID(usr.uuid) as uuid,

usr.email,

prf.user\_name,

usr.role, prf.avatar

FROM

`user` as usr inner join

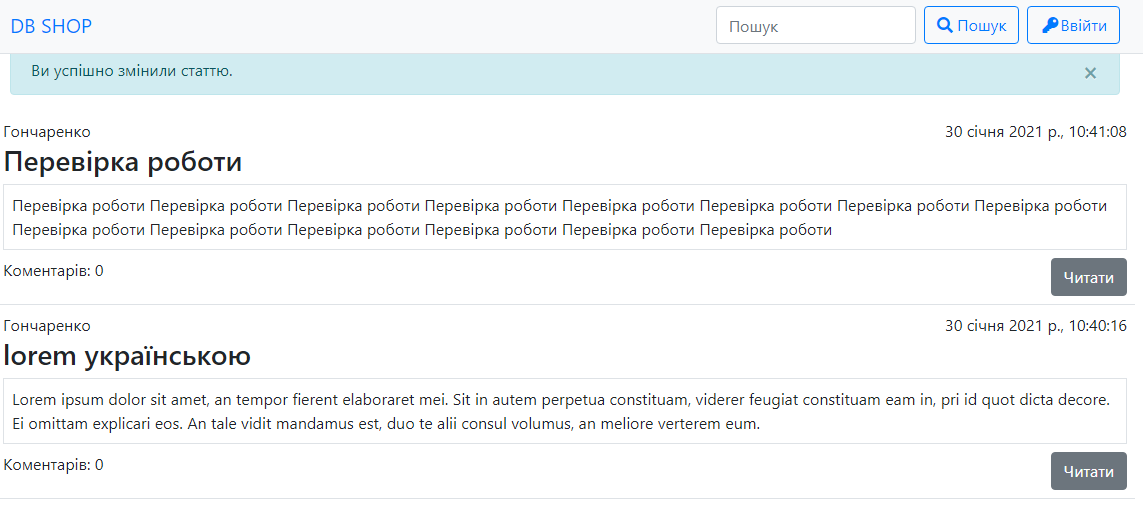
`profile` as prf on prf.uuid = usr.uuid

WHERE

BIN\_TO\_UUID(usr.uuid) = ?,

як параметр передається ідентифікатор користувача

2. В подальшому відкривається **головна сторінка** клієнтської програми на якому відображається перелік статей, які заведені в програмі. Крім того на головній сторінці відображається панель навігації програми. Ця панель доступна майже на всіх сторінках програми крім сторінки входу та реєстрації користувача.



При завантажені головної сторінка спрацьовує метод mounted() Vue - компонента client\src\views\Home.vue , який відповідає за головну сторінку програми.

  async mounted() {

    try {

      await this.$store.dispatch('getPosts')

      return

    } catch (error) {

      this.$store.commit('addMessage', {

        type: 'danger',

        message: error.message,

      })

      console.log('Home mounted error:', error)

      throw error

    }

  }

В цьому методі відбувається звернення до методу getPosts() модуля client\src\store\post.js.

    async getPosts({ dispatch, commit, state }) {

      try {

        let posts

        if (state.filter === '') {

          posts = await dispatch('request', {

            url: '/api/post/',

            method: 'GET',

          })

        } else {

          posts = await dispatch('request', {

            url: `/api/post/search/${state.filter}`,

            method: 'GET',

          })

        }

        commit('setPosts', posts)

        return

      } catch (error) {

        console.log('getPosts error:', error)

        throw error

      }

    }

Цей метод в залежності від наявної інформації в полі вводу навігаційної панелі або до точки входу сервера застосунку з url /api/post/ (метод GET), або з url `/api/post/search/${state.filter}` (метод GET). Обидва методи повертають перелік інформації про статті у вигляді масиву об'єктів типу [{id, title, author, date, short,comments}], де

* id – ідентифікатор статті,
* title – заголовок статті,
* author - ім'я автора,
* date – дата створення статті,
* short – короткий зміст статті,
* comments - кількість коментарів

цей масив записується в змінну state.posts модуля client\src\store\post.js та відображається на головній сторінці програми.

Для отримання відповіді на цей запис на сервері застосунку реалізовані відповідні маршрути в модулі routers\post.routers.js:

/\*\* повертає перелік статей в програмі

 \* запит: метод - GET; шлях - //api/post/

 \* повертає 200 [{id, title, author, date, short,comments}], де author - ім'я автора, comments - кількість коментарів

 \* або 500

 \*/

router.get('/', async function (req, res) {

  try {

    const posts = await Post.getPosts()

    res.send(JSON.stringify(posts))

  } catch (error) {

    console.log('GET /api/post error: ', error)

    res.status(500).send(

      JSON.stringify({

        source: 'GET /api/post',

        message: error.message,

      })

    )

  }

})

В цьому маршруті відбувається звернення до метода getPosts() або findPosts(serchStr) моделі models\Post.js

  async getPosts() {

    try {

      const connection = await db()

      const query =

        'SELECT `post`.`id`, `post`.`title`, `profile`.`user\_name` as author,' +

        ' `post`.`post\_date` as date, `post`.`short\_content` as short, `cmt`.`comments` FROM `' +

        dbName +

        '`.`post` ' +

        'inner join `' +

        dbName +

        '`.`profile` on `post`.`author\_id` = `profile`.`uuid` ' +

        'left join (select `comment`.`post\_id`,  count(\*) as comments from `' +

        dbName +

        '`.`comment` group by `comment`.`post\_id`) as cmt on `cmt`.post\_id = `post`.`id`' +

        'order by `post`.`post\_date` DESC'

      const [rows] = await connection.query(query)

      return rows

    } catch (error) {

      console.log('getPosts: ', error)

      throw error

    }

  },

  async findPosts(searchStr) {

    try {

      const connection = await db()

      const query =

        'SELECT `post`.`id`, `post`.`title`, `profile`.`user\_name` as author,' +

        ' `post`.`post\_date` as date, `post`.`short\_content` as short, `cmt`.`comments` FROM `' +

        dbName +

        '`.`post` ' +

        'inner join `' +

        dbName +

        '`.`profile` on `post`.`author\_id` = `profile`.`uuid` ' +

        'left join (select `comment`.`post\_id`,  count(\*) as comments from `' +

        dbName +

        '`.`comment` group by `comment`.`post\_id`) as cmt on `cmt`.post\_id = `post`.`id`' +

        'where (`post`.`title` like ' +

        connection.escape('%' + searchStr + '%') +

        ') OR (`post`.`short\_content` like ' +

        connection.escape('%' + searchStr + '%') +

        ') OR (`post`.`post\_content` like ' +

        connection.escape('%' + searchStr + '%') +

        ')'

      ;('order by `post`.`post\_date` DESC')

      const [rows] = await connection.query(query)

      return rows

    } catch (error) {

      console.log('getPosts: ', error)

      throw error

    }

  }

В методах виконуються такі SQL запити:

SELECT

 `post`.`id`,

 `post`.`title`,

`profile`.`user\_name` as author,

 `post`.`post\_date` as date,

  `post`.`short\_content` as short,

`cmt`.`comments`

FROM

`post` inner join

`profile` on `post`.`author\_id` = `profile`.`uuid` left join

(SELECT

`comment`.`post\_id`,

  count(\*) as comments

FROM

`comment`

GROUP BY

`comment`.`post\_id`) as cmt on `cmt`.post\_id = `post`.`id`

ORDER BY `post`.`post\_date` DESC

та

SELECT

 `post`.`id`,

 `post`.`title`,

`profile`.`user\_name` as author,

 `post`.`post\_date` as date,

  `post`.`short\_content` as short,

`cmt`.`comments`

FROM

`post` inner join

`profile` on `post`.`author\_id` = `profile`.`uuid` left join

(SELECT

`comment`.`post\_id`,

  count(\*) as comments

FROM

`comment`

GROUP BY

`comment`.`post\_id`) as cmt on `cmt`.post\_id = `post`.`id`

WHERE

(`post`.`title` like '%' + searchStr + '%')

OR (`post`.`short\_content` like '%' + searchStr + '%')

OR (`post`.`post\_content` like '%' + searchStr + '%')

ORDER BY `post`.`post\_date` DESC

Результат виконання цих запитів повертається клієнту.

Кожна стаття на головній сторінці представлена наступною інформацією:

автор та дата створення,

заголовок статті,

короткий зміст статті,

кількість коментарів до цієї статті,

кнопка "Читати", при натисканні на яку відкривається сторінка перегляду обраної статті (компонент client\src\views\PostRead.vue).

3. Сторінка перегляду обраної статті.