# Опис серверної частини.

Серверна частина нашого додатку побудована на Node.js. [Node.js®](https://nodejs.org/uk/) — це JavaScript–оточення побудоване на JavaScript–рушієві Chrome V8.

У документації до Node.js зазначено, що як асинхронне подієве JavaScript–оточення, Node.js спроектований для побудови масштабованих мережевих додатків. Для кожного з’єднання викликається функція зворотнього виклику, проте коли з’єднань немає Node.js засинає.

Це контрастує з більш загальною моделлю в якій використовуються паралельні OS потоки. Такий підхід є відносно неефективним та дуже важким у використанні. Більше того, користувачі Node.js можуть не турбуватись про блокування процесів, оскільки немає жодних блокувань. Майже жодна з функцій у Node.js не працює напряму з I/O, тому процес не блокується ніколи. Оскільки нічого не блокується на Node.js легко розробляти масштабовані системи.

Для забезпечення створення http-сервера і роботи додатка за протоколом http був використаний фреймворк [Express - Fast, unopinionated, minimalist web framework for Node.js](https://expressjs.com). Як стверджує сайт Express.js Express — це мінімальний і гнучкий фреймворк веб-додатків Node.js, який надає надійний набір функцій для веб- та мобільних додатків.

Для взаємодії з клієнтською частиною нашого додатку за протоколом Web Socket була використана бібліотека [Socket.IO](https://socket.io/docs/v4/). Socket.IO — це бібліотека, яка забезпечує двонаправлений зв'язок між клієнтом і сервером із низькими затримками та на основі подій. Вона побудована на основі протоколу WebSocket і надає додаткові гарантії, такі як повернення до тривалого опитування HTTP, якщо браузер не підтримує роботу за протоколом web socket або автоматичне повторне підключення.

Для зберігання даних додатку використовується система керування базою даних [PostgreSQL](https://www.postgresql.org). PostgreSQL – це потужна система об’єктно-реляційних баз даних з відкритим вихідним кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, які безпечно зберігають і масштабують найскладніші робочі навантаження даних.

Проаналізувавши предметну область нашого завдання прийшли до висновку, у нас існує три окремих сутності, а саме користувач, «кімната для переговорів» або чат, повідомлення.

Користувач має наступні властивості: логін, пароль, ім’я, що відображається, аватар – графічне зображення, яке пов’язане з цим користувачем. Для зберігання аватару будемо використовувати файлову систему, а в базі даних будемо зберігати тільки ім’я файлу.

«Кімната для переговорів» повинна мати такі властивості

Ідентифікатор, тип кімнати (коли програма буде розвиватися, з’явиться можливість створювати не тільки персональні чати, але і групи та інформаційні канали. Крім того в подальшому функціонал програми буде передбачати можливість блокування «кімнат» адміністратором або співрозмовником. Тому в базі даних такі можливості повинні бути передбачені. Назва кімнати є індивідуальною для кожного користувача і тому не повинна бути пов’язана з цією таблицею.

Повідомлення має такі властивості: автор повідомлення, кімната, в яку це повідомлення відправлено, та власне текст повідомлення.

Сутності мають наступні зв’язки;

Користувачі пов’язані з повідомленнями зв’язком «один до багатьох» через поле в таблиці повідомлень – «автор».

«Кімнати» пов’язані з повідомленнями зв’язком «один до багатьох» через поле в таблиці повідомлень «призначення».

«Кімнати» пов’язані з користувачами зв’язком «багато до багатьох». Для реалізації такого зв’язку необхідно ввести додаткову таблицю, що пов’язує кімнати та користувачів. Додатково в базі даних заведена таблиця для зберігання сесій фреймворка Express. ERD діаграма наведена нижче.



Для спрощення побудови SQL запитів, необхідних для роботи програми створено 2 представлення:

room\_members – встановлює зв'язок між членами однієї «кімнати»

CREATE OR REPLACE VIEW public.room\_members

    AS

     SELECT ro.member AS owner,

    o.user\_name AS owner\_name,

    ro.room\_id,

    ro.room\_name,

    rm.member,

    mb.user\_name AS member\_name,

    ro.id as owner\_ru\_id,

    rm.id as member\_ru\_id

   FROM room\_users ro

     JOIN room\_users rm ON ro.room\_id = rm.room\_id AND ro.member <> rm.member

     JOIN users o ON ro.member = o.id

     JOIN users mb ON rm.member = mb.id;

Contacts відображає для кожного користувача його список контактів, тобто перелік користувачів, з яким у даного користувача відсутні «переговорні кімнати»

CREATE OR REPLACE VIEW public.contacts

 AS

 SELECT u.id AS owner,

    u1.id,

    u1.login,

    u1.user\_name,

    u1.state,

    u1.avatar,

    u1.created\_at,

    u1.modified\_at

   FROM users u

     JOIN users u1 ON u.id <> u1.id

     LEFT JOIN room\_members rm ON u.id = rm.owner AND u1.id = rm.member

  WHERE rm.room\_id IS NULL

  ORDER BY u1.id;

Для забезпечення роботи нашої програми з базою даних використовується драйвер [node-postgres](https://github.com/brianc/node-postgres). Node-postgres це неблокуючий клієнт PostgreSQL для Node.js. Node-postgres має наступні властивості:

* чистий клієнт JavaScript і власні прив’язки бібліотеки libpq мають однаковий API;
* пул підключень;
* розширене приведення типу даних між PostgreSQL та JavaScript;
* підтримка властивостей PostgreSQL:
  + параметризовані запити;
  + іменовані оператори з кешуванням плану запитів;
  + асинхронні сповіщення за допомогою LISTEN/NOTIFY;
  + масовий імпорт та експорт за допомогою COPY TO/COPY FROM.

Для взаємодії з базою даних створений модуль db\index.js. При завантаженні цього модуля створюються пул з’єднань з базою даних, використовуючи рядок приєднання до бази даних зазначений в змінній оточення.

const connectionString = process.env.DATABASE\_URL;

pool = new Pool({

      connectionString,

      ssl: false,

    })

Чому використовується пул з’єднань?

Підключення нового клієнта до сервера PostgreSQL вимагає рукостискання, яке може тривати 20-30 мілісекунд. Протягом цього часу узгоджуються паролі, може бути встановлений SSL, а інформація про конфігурацію передається клієнту та серверу. Виникнення цих витрат кожного разу , коли ми хочемо виконати запит, істотно уповільнює роботу нашої програми.

Сервер PostgreSQL може обробляти лише обмежену кількість клієнтів одночасно . Залежно від доступної пам’яті вашого сервера PostgreSQL, ви можете навіть вийти з ладу сервера, якщо підключите необмежене число клієнтів.

PostgreSQL може обробляти лише один запит на одному підключеному клієнті за принципом «першим прийшов – першим вийшов». Якщо веб-додаток із багатьма клієнтами використовує лише один підключений клієнт, усі запити між усіма одночасними запитами будуть конвеєрними та виконуватися послідовно, один за одним. Не добре!

Виходячи з цих міркувань в програмі обрано механізм роботи через пул. Максимальна кількість одночасних клієнтів, які можуть бути підключеними через пул є 10. Якщо буде більше одночасних запитів, то вони будуть ставитись в чергу. Метод getClient() повертає нового клієнта з пула.

В цьому ж модулі реалізований метод який використовується для виконання всіх параметризованих запитів до бази даних:

  /\*\* виконує SQL запит з вказаним клієнтом

   \* @param { PoolClient } client

   \* @param { String } sql

   \* @param {Array} params

   \* @returns  { Promise<pg.Result> } повертає результат виконання sql

   \*/

  async clientQuery(client, sql, params) {

    try {

      console.dir({ sql, params });

      const result = await client.query(sql, params);

      return result;

    } catch (error) {

      if (!error.type) {

        error.type = 'db error';

      }

      if (!error.source) {

        error.source = 'db index clientQuery';

        console.log(error);

      }

      throw error;

    }

  },

};

В програмі використовуються виключно параметризовані запити до бази даних. Це дає можливість запобігти атакам на додаток, які називаються SQL Injection. SQL Injection – це техніка введення коду, яка може знищити базу даних. SQL Injection є однією з найпоширеніших методик веб-зламу. SQL Injection – це розміщення шкідливого коду в операторах SQL через введення веб-сторінки.

В модулі db/Model.js визначений **клас Model**, в якому реалізовані основні операції по роботі з таблицями в базі даних. В якості властивостей цей клас приймає назву таблиці та клієнта з пулу, під’єднаного до бази даних. В класі є наступні методи

find: Побудова та виконання звичайного select до таблиці this.table. Методу

в якості параметрів передається масив з переліком назв колонок таблиці, які будуть повертатися, як результат виконання SQL, перелік умов для виборки у вигляді об’єкту JavaScript в форматі {key: value}. Також передається масив колонок для побудови сортування. Результатом є масив об’єктів JavaScript;

findOne: Аналогічний попередньому, але повертає тільки перший рядок результату;

delete: Видалення записів з таблиці. Критерії відбору передаються в якості об’єкту JavaScript в форматі {key: value}. Результатом роботи методу є кількість рядків, які були видалені;

insert: Додавання запису в таблицю. Назви колонок та їх значення передаються в параметрі columns у вигляді об’єкту JavaScript в форматі {key: value}. В масиві returning передається перелік колонок запису в таблиці, що був доданий. Результатом є масив об’єктів JavaScript;

update: Внесенні змін в таблицю. Назви колонок та їх значення передаються в параметрі columns. Критерії відбору передаються в якості об’єкту JavaScript в форматі {key: value} в параметрі params. Повертається кількість змінених рядків.

Для роботи з конкретними таблицями визначені класи-нащадки класу Model, в яких реалізовані специфічні методи для цих сутностей.

**Клас User** визначений в модулі user/User.js. У зв’язку із тим, що дана сутність є важливою з точки зору безпеки (в таблиці зберігається хеш пароля) та з метою уніфікації даних, що повертаються користувачу, в даному класі визначено перелік колонок таблиці, які можуть повертатися з бази даних (пароль ніколи не передається). Визначено наступні методи:

findUser – пошук користувача за параметрами. Наприклад за його логіном.

У зв’язку із тим, що дуже часто використовується пошук користувача за його ідентифікатором, ця процедура винесена в окремий метод getUserById(userId), де ідентифікатор передається як параметр.

verifyPassword(userId, verifiedPassword) – перевіряє відповідність пароля, переданого для перевірки тому, який збережений в базі даних.

newUser(login, password, username) - створює новий запис в таблиці public.users на підставі логіна, пароля та імені користувача. При цьому перевіряються передані параметри, з наступними умовами: довжина логіна і пароля не менше 6 символів, поле «ім’я користувача» повинне бути заповнене.

getContacts(userId) – повертає перелік користувачів, у яких для користувача з userId відсутні чати. Результат повертається у вигляді масиву об’єктів JavaScript.

updateUser(userId, updateData) – повертає кількість змінених рядків в таблиці public.users. Дані, які необхідно змінити передаються в змінній updateData у вигляді об’єкту JavaScript в форматі {key: value}. Зміну таких даних, як ім’я користувача та його аватар необхідно внести не тільки до таблиці «users», але і до таблиці «room\_users». Такі зміни вносяться в межах однієї транзакції.

**Клас Room** визначений в модулі room/Room.js. Для забезпечення уніфікації в модулі визначено перелік колонок, які повертаються користувачеві.

Методи визначені в класі:

findRooms(params). В змінній params передаються параметри пошуку у вигляді об’єкту JavaScript в форматі {key: value}.

findUserRooms(userId) повертає перелік кімнат, в яких учасником є користувач з ідентифікатором userId

newRoom(members, roomState = 0, roomType = 0) процедура створення нової кімнати. Змінна roomState - ознака стану кімнати. Якщо не дорівнює 0 кімната заблокована. Змінна roomType тип кімнати. Можливі значення:

0 – чат на двох осіб;

1 – група для спілкування;

3 – інформаційний канал.

В програмі реалізований тільки варіант чату.

**Клас Message** визначений в модулі message/Message.js. В класі визначені наступні методи:

getMessagesInRoom (roomId) – повертає повідомлення, зареєстровані в кімнаті визначеним ідентифікатором.

addMessage(message) – додає повідомлення до бази даних.

Бізнес логіка по роботі програми з базою даних реалізована в відповідних модулях user\index.js, user\User.Service.js, room\index.js, room\Room.Service.js, message\index.js, message\Message.Servece.js, auth\index.js.

Окремо необхідно показати методі зібрані в модулі auth\index.js. Ці методи пов’язані з автентифікацією користувача. В модулі визначені наступні методи: authentication(req, res, next) – якщо в сесії збережено ідентифікатор користувача, перевіряється наявність цього користувача в базі даних (з урахуванням його стану). Якщо такий користувач знайдений, то його дані записуються в об’єкт запиту req і можуть бути використані в подальшій обробці запиту, якщо ідентифікатор відсутній в сесії, або користувач відсутній в базі, або заблокований в об’єкт запиту req запис про користувача буде відсутній. Цей метод використовується як проміжний обробник як при роботі з протоколом http так і з протоколом WebSocket.

loginHandler(req, res) – забезпечує виконання процедури входу користувача до програми. Даний обробник використовується за протоколом http. В тілі об’єкту запиту req повинен бути присутні логін та пароль. Як результат успішного входу в сесію зберігається ідентифікатор користувача, а інформація про користувача повертається клієнту. В разі неуспішного входу клієнту повертається http status code = 401 разом з повідомленням про помилку.

registerHandler(req, res) – забезпечує виконання процедури реєстрації користувача в програми. Даний обробник використовується за протоколом http. В тілі об’єкту запиту req повинен бути присутні логін, ім’я користувача та пароль. Як результат успішного реєстрації в сесію зберігається ідентифікатор користувача, а інформація про користувача повертається клієнту. В разі неуспішного входу клієнту повертається http status code = 400 або 500 разом з повідомленням про помилку.

logoutHandler(req, res) – забезпечує виконання процедури виходу користувача з системи. Хоча дана процедура працює за протоколом http, вона завершує також з’єднання за протоколом WebSocket ідентифікатор, якого записаний в сесії.

**Головним файлом серверної частини є файл index.js**. В першу чергу там підключається фреймворк express.js та створюється екземпляр http сервера.

Із змінних оточення зчитуються константи для конфігурації серверної частини. Це PORT – порт, який буде прослуховувати сервер (http та ws), PUBLIC\_PATH – шлях, де зберігаються статичні файли сервера (клієнтська частина застосунку), SESSION\_SECRET, SESSION\_EXPIRES – константи для налаштування сесії.

Визначаються глобальні змінні activeSockets (зберігається всі підключені сокети), activeUsers (зберігається всі підключені користувачі)

Після цього підключається обробник статичних файлів для визначеного каталога. Відповідно звернення з використанням методу GET до http сервера за шляхом ‘/’ завантажує в браузер файл index.html, за шляхом ‘/<path/filename>’ завантажує в браузер файл, якщо він існує.

Підключаються необхідні модулі.

Ініціалізується екземпляр сесії з модуля express-session та підключається до бази даних.

Обробник сесії підключається до http сервера.

Підключається обробник автентифікації.

Встановлюються обробники з’єднань за протоколом http. В програмі обробляються наступні запити від клієнта:

Метод POST, маршрут '/api/login', обробник loginHandler;

Метод POST, маршрут '/api/ register', обробник registerHandler;

Метод POST, маршрут '/api/ logout, обробник logoutHandler;

Крім того всі запити від клієнта методом GET переправляються на домашню сторінку програми.

Далі іде налаштування роботи сервера за протоколом WebSocket з модуля 'socket.io' .

Socket.io, як і express.js, підтримує використання проміжних обробників, хоча і з дещо іншим контрактом виклику. В додатку використовують два проміжних обробники, які повинні працювати однаково для різних протоколів. Це обробник сесії та автентифікації. Тому, для надання можливості використання проміжних обробників розроблених для express.js з Socket.io необхідно написати функціє-обгортку:

const wrap = (middleware) => (socket, next) => {

  console.log('middleware.name - ', middleware.name);

  return middleware(socket.request, {}, next);

};

З використанням цієї обгортки підключаються обробники сесії та автентифікації. Наступний проміжний обробник перевіряє чи проведена автентифікація. В разі відсутності користувача в змінній socket.request.user генерується помилка з’єднання 'unauthorized', яка передається клієнту.

Далі для сервера встановлюється обробник для події 'connect'.

В цій події визначений об’єкт з’єднання socket.

Зчитуються об’єкти сесії та користувача. Знов в разі відсутності користувача з’єднання розривається. Ідентифікатор сокета в якості ключа записується в activeSocets, в якості значення записується сам об’єкт сокета та користувач.

В activeUsers в якості ключа використовується ідентифікатор користувача, в якості значення записується масив сокетів, в яких зареєстрований цей користувач.

Визначається перелік кімнат, в яких зареєстрований цей користувач.

Цей перелік та інформація про користувача відправляється клієнту.

Після цього встановлюються обробники для подій цього сокету.

В модулі handlers\index.js dстановлюються обробники для наступних подій:

'get rooms' – клієнт запитує перелік своїх кімнат;

'new chat' – клієнт вимагає створити новий чат;

'disconnect' – припинення з’єднання;

'join' – клієнт входить до визначеної кімнати;

'who am i' – клієнт запитує інформацію про поточного користувача;

'contacts' – клієнт запитує інформацію про свої контакти;

'message' – клієнт відправив повідомлення;

'update user' – клієнт надіслав оновлену інформацію про себе;

Сервер запускається з прослуховуванням визначеного порта.