ДЗ на 13.04 и 14.04

Введение

На первой лекции 06.04 мы установили все необходимое ПО для разработки серверной части на языке JavaScript. **VsCode** - текстовый редактор(Продвинутый); **NodeJS** + **npm** - сама оболочка для запуска JS и пакетный менеджер(Для установки сторонних зависимостей); **insomnia**(Кто-то **Postman**) - REST клиент, для запросов к серверу. Написали свой первый сервер, всего с одним методом http://localhost:3000/

Задание

1. Так как у нас курс по языку JavaScript - то нужно подтянуть знания этого языка. Для этого делаем следующее:
    a. В течение всего курса - смотрим этот видеоурок https://www.youtube.com/watch?v=Bluxbh9CaQ0. Вообще, у этого челика есть видеоуроки на очень много тем, кто заинтересован - рекомендуем
    b. Скачиваем себе на телефон приложение **SoloLearn**(Есть как на iOS так и на Android)
    c. Находим там курс по JavaScript
    d. Проходим: Введение, Основные понятия, Условные циклы, функции и Объекты
2. Читаем общую теорию про разработку Клиент-Серверных приложений. https://habr.com/ru/post/495698/
3. На сервере, который мы написали на лекции нужно добавить методы всех типов, https://habr.com/ru/post/447322/
    a. POST /testPost, ответ: 200, { message: ‘It is POST’ }
    b. GET /testGet, ответ: 200, { message: ‘It is GET’ }
    c. DELETE /testDelete, ответ: 400, { message: ‘Sorry, this method not work’ }
    d. PUT /testPut, ответ: 500, { message: ‘Server error’ }
    e. PATCH /testPatch, ответ: 403, { message: ‘Your token incorrect’ }
4. Читаем про коды ответов сервера, что каждый из них обозначает https://habr.com/ru/post/533606/
    a. 200
    b. 400
    c. 401
    d. 402
    e. 403
    f. 404
    g. 500
5. Читаем про то, что такое express - https://expressjs.com/ru/
6. Добавляем метод: POST /addToDo, ответ: 200, { message: ‘Add ToDo success’ }
7. Добавляем метод: DELETE /deleteToDo, ответ: 200, { message: ‘Delete ToDo success’ }
8. Добавляем метод: POST /updateToDo, ответ: 200, { message: ‘Update ToDo success’ }

9. Добавляем метод: GET /todoList, ответ: 200, { todoList: [...] }. Модель одной ToDo: { _id: ‘...’, title: ‘...’, description: ‘...’ }. Список должен генериться методом **for**, где toDo._id = index.

1 этап

1. БД Mongo, развёрнутая локально на на отдельном порте или подключенная MongoCloud
2. В БД одна модель - ToDo
3. Поля ToDo: _id, title, description, isComplete
4. Добавляем метод получить все ToDo. GET /api/todos
5. Добавляем метод получить ToDo по _id. GET /api/todos/id
6. Методы 4 и 5 - можно объединить. Если нет _id в запросе - отдаем список со всеми, если есть - ищем по этому _id
7. Добавляем метод создать ToDo. Параметры: title: ‘...’, description: ‘...’. POST /api/todo
8. Добавляем метод удалить ToDo. DELETE /api/todos/id
9. Добавляем метод обновить ToDo. UPDATE /api/todos/id
10. Обработка ошибок и отправка их на фронт - через asyncHandler !

2 этап

1. Пишем Client часть для работы с сервером
2. Продумываем архитектуру обработки NET запросов
3. Работаем только с сетью - БЕЗ локального кэша
4. Добавляем модель User
5. Поля User: _id, email, password, phone, toDoList: [...]
6. Добавляем CRUD методы к User
7. Оставляем добавление ToDo-s в общий список, не к пользователю!

3 этап

1. Добавляем метод /auth/login. Авторизация пользователя по логину и паролю.
2. Ответ от сервера: userId = ‘...’. Параметры запроса: login: ‘...’, password: ‘...’, deviceId: ‘...’
3. Если пользователя с такими данными не нашлось - 404
4. В запросах, связанный с ToDo, добавляем _id пользователя, кто это делает, чтобы прикрепить ToDo к этому пользователю. Параметр: userId: ‘...’
5. Этот параметр добавляем в ЗАГОЛОВКИ всех запросов. Headers: userId=’...’
6. Обновляем методы для работы с ToDo.

4 этап

1. Добавляем в бд еще одну модель: Token
2. Поля Token: _id = ‘...’, UserId(чей токен) = ’...’, accessToken(сам токен), deviceId = ’...’
3. Смотрим на варианты, как работает JWT и ТД

5 этап

1. Переписываем метод авторизации(/auth/login). В ответе теперь возвращается не _id пользователя, а его accessToken: ‘...’.
2. Токен создается и записывается в БД. Поля токена: _id: ‘...’, userId: ‘...’, accessToken: ‘...’, deviceId: ‘...’.
3. Если логин и пароль ОК -> ищем в БД Token по такому deviceId && userId. Если он есть - мы его удаляем(весь объект целиком) и создаем новый, если нет -> создаем новый.
4. Переписываем NetClient. Меняем заголовок (Headers) userId -> accessToken
5. Добавляем на сервере метод - получение пользователя по accessToken
6. Устанавливаем middleware обработчик на запросы, которые идут в **авторизованную зону**. Проверка пользователя по accessToken. Если он некорректный - 403, message: tokenIncorrect

6 этап

1. Добавляем на сервере метод /auth/registration. Параметры: email: ‘...’, password: ‘...’.
2. После валидации всех полей на сервере идет проверка по email.
3. Если пользователь с таким email уже существует -> 400, данный email уже используется.
4. Если email свободен -> Создаем пользователя в БД.
5. Модели User добавляем поле isConfirmed = Boolean(default: false)

7 этап

1. Добавляем в бд модель: ConfirmRegistration
2. Поля: _id, email, userId, secretKey
3. Обновляем метод /auth/registration. После валидации полей несколько сценариев:
    a. Пользователь с таким email есть и он isConfirmed=true -> такой пользователь есть и он уже активен, ответ: 400, данный email уже используется.
    b. Пользователь с таким email есть и он isConfirmed=false -> кто то уже производил попытку регистрации с этого email, но не подтвердил ее -> заменяем поля объекта User, на те, которые были отправлены в новой регистрации, ищем в бд объект типа ConfirmRegistration для этого email. Нашли -> удаляем его и создаём новый; Не нашли -> создаём новый
    c. Пользователя с таким email ещё нет -> Создаем пользователя с такими данными, ищем в бд объект типа ConfirmRegistration для этого email. Нашли -> удаляем его и создаём новый; Не нашли -> создаём новый
4. В случае b и c - Отправляем на почту клиента ссылку для активации аккаунта. Ответ от сервера: 200, ссылка для активации аккаунта, отправлена на почту. Ссылка собирается из объекта ConfirmRegistration
5. Формат ссылки: IP/api/auth/confirmRegistration?id=USER_ID&secretKey=SECRET_KEY

6. USER_ID - Id пользователя, которого мы создали в бд; SECRET_KEY - секретный ключ, просто рандомный набор символов. Генерим как токен или типо того)
7. Добавляем на сервере метод GET /auth/confirmRegistration. query=id=USER_ID&secretKey=SECRET_KEY
8. При выполнении запроса /auth/confirmRegistration ищем в БД объект ConfirmRegistration по USER_ID && SECRET_KEY.
9. Нашли -> ставим пользователю(id нам известен - из ссылки или из объекта ConfirmRegistration) isConfirmed=true, удаляем целиком объект ConfirmRegistration и ответ на запрос = 200, ваша учётная запись успешно подтверждена.
10. Не нашли -> ответ 403, неверная ссылка
11. Переделываем метод /auth/login. Если пользователь ещё не активировал аккаунт(проверка по полю isConfirmed) -> ответ сервера: 400, ваш аккаунт ещё не активирован, проверьте почту $EMAIL$.

8 этап

1. Читаем статью - https://habr.com/ru/company/voximplant/blog/323160/
2. Добавляем систему с двумя токенами accessToken + refreshToken
3. Обновляем модель Token. Добавляем поле refreshTooken: '...'
4. Поле refreshToken - создаётся точно так же как и accessToke
5. В методе /auth/login - изменяем ответ: возвращаем два токена (accessToken + refreshToken)
6. На сервере добавляем метод /auth/refreshToken. В параметрах: refreshToken: '...'. Ищем модель Token с таким значение refreshToken.
7. Нашли -> обновляем оба токена(accessToken + refreshToken, внутри объекта) -> ответ на фронт: accessToken: '...', refreshToken: '...'
8. Не нашли -> 403, tokenIncorrect
9. Переписываем NetClient приложения для обработки ошибки 403 + /refreshToken. Чтобы сделать *бесшовное обновление токенов*. Отправляем запрос с accessToken -> 403 -> отправляем запрос /refreshToken -> все окей -> обновляем два токена на телефоне -> отправляем первоначальный запрос с НОВЫМ accessToken

9 этап

1. Добавляем модели Token время жизни. Два поля: atExpired: Date, rtExpired: Date. В них записываем время, ДО КОТОРОГО данный токен АКТИВЕН.
2. Добавляем их обработку в middleware на сервере
3. Отправка ошибки на фронт 403 + tokenExpired
4. Так как все запросы идут с заголовком token - легко проверяем. Метод /refreshToken - выносим выше, так как он пойдет без заголовка

10 этап

1. В ответ с ошибкой добавляем поле reason - причина. Это будет ENUM

2. Переписываем на фронте NetClient для обработки этого поля. На основе него будут происходить определенные действия
3. Переписываем на сервере систему работы с токенами
4. Если что то не так с Access token -> error: 403 + message(любой) + reason(ACCESS_TOKEN_INCORRECT)
5. Если ошибка в методе /refreshToken -> error: 403 + message(любой) + reason(REFRESH_TOKEN_INCORRECT)

!ДАЛЬШЕ НЕ ДЕЛАТЬ!

11 этап

1. Добавляем модели пользователя поле phone: ‘...’
2. Добавляем модель SmsRegistration. Поля: _id, phone, smsCode
3. Добавляем на сервере метод для авторизации/регистрации по номеру телефона + СМС
4. /auth/phoneAuth. Поля: phone: ‘...’
5. При вызове данного метода ищем в БД объект SmsRegistration по полю phone. Нашли -> удаляем и создаем новый. Не нашли -> создаем новый
6. Отправляем пользователю на телефон СМС с кодом подтверждение
7. Добавляем на сервере метод подтверждение номера телефона
8. /auth/confirmPhoneAuth. Поля: phone: ‘...’, smsCode: ‘...’, deviceId; ‘...’
9. Производим проверку правильного кода подтверждения - Ищем в БД объект SmsRegistration по phone && smsCode.
10. Не нашли -> выброс из метода, ответ 400, неверный СМС-код
11. Нашли -> удаляем целиком объект
12. Ищем в БД пользователя по номеру телефона
13. Не нашли -> создаем нового. Нашли - ничего не делаем
14. Ищем в БД объект Token для этого пользователя + deviceId
15. Не нашли -> создаем новый объект Token
16. Нашли -> удаляем его целиком -> создаем новый. Процесс создания токен - описан выше.
17. Ответ на фронт: 200, accessToken: ‘...’, refreshToken: ‘...’

12 этап

1. Добавляем на сервере метода получения пользователя. GET /api/me
2. Получение пользователя должно произойти по его accessToken, который мы передаем в заголовках всех запросов
3. На стороне приложения добавляем экран - Профиль. Данные на экран грузятся благодаря методу из п. 1
4. Заменяем поле у модели User.isConfirmed -> User.emailConfirmed
5. Переименовываем модель ConfirmRegistration -> EmailConfirmation
6. Добавляем метод POST /api/addEmail. Поля: email: ‘...’
7. Ищем пользователя с таким email в БД
8. Нашли -> выброс из метода, ответ: 400, Данный email уже используется
9. Не нашли -> проверяем что у данного пользователя(по accessToken) нет email
10. У него есть email -> выброс из метода, 400, у данного пользователя уже есть email
11. У пользователя нет email -> Вызываем метод для создание и дальнейшей обработки EmailConfirmation(Принцип его работы описан выше)
12. Ответ при успешном выполнении: На вашу почту было отправлено письмо с подтверждением email
13. Изменяем название метода IP/api/auth/confirmRegistration -> IP/api/auth/confirmEmail

14. Изменяем ответ на метод IP/api/auth/confirmEmail. При успешном выполнении: 200, Email $EMAIL$ успешно подтвержден

13 этап

1. Добавляем метод POST /api/addPhone. Поля: phone: ‘...’
2. На сервере ищем пользователя с таким номером телефона
3. Нашли -> выброс из метода, 400, данный номер телефона уже используется
4. Не нашли -> ищем в БД SmsRegistration по данному номеру телефона
5. Нашли -> удаляем, создаём новую
6. Не нашли -> создаём новую
7. Логика создания описана выше. Поля: phone:.’...’, smsCode: ‘...’
8. Добавляем на сервере метод: POST /api/confirmAddPhone. Поля: phone: ‘...’, smsCode: ‘...’
9. Ищем в БД объект SmsRegistration по phone && smsCode
10. Не нашли -> выброс из метода, 400, введен неверный СМС-код
11. Нашли -> удаляем и номер телефона записываем в объект User
12. 200, Номер телефона успешно добавлен

14 этап

1. Переименовываем SmsRegistration -> SmsConfirmation
2.
13. Переименовываем SmsRegistration -> SmsConfirmation
14. Переименовать POST /api/addEmail -> POST /api/updateEmail
15. Переименовать POST /api/addPhone -> POST /api/updatePhone
16. Метод POST /api/logOut
17. Запаковать авторизацию в отдельную папку(пакет npm, git repo). Настройка, что нам нужно: настраивается через файл json. Пишем в нем все руками или через метод LOGIN_SERVICE.configure(). Сделать его в виде middleWare обработчика ИЛИ дать возможность его использовать через колбэки. Пояснение: делаем ему .init(), и там передаём все необходимые коллбэки, на все чихи червиса.Чтобы мы на любом нашем сервере - могли его использовать для следующих действий:
    a. Регистрация(телефон + смс, email + password + confirm)
    b. Авторизация(телефон + смс, email + password)
    c. Аутентификация (работа с токенами)

15(ПЕРЕДЕЛКА)

1. Добавляем в БД модель **UserCredentials**. Поля: _id, email, password, phone, isAccountConfirmed
2. Добавляем в БД модель **Email**. Поля: _id, userCredId, value, isConfirmed
3. Добавляем в БД модель **Phone**. Поля: _id, userCredId, value, isConfirmed
4. Добавляем в БД модель **Phone**. Поля: _id, userCredId, value, isConfirmed
5. /registration(email, password). Ответ от сервера: Ok/Error.
    5.1. Ищем пользователя по email(Просто ищем в БД модель Email с таким value);
    5.2. Нашли -> Выброс на фронт ошибки: Данный Email уже используется
    5.3. Не нашли -> Создаем UserCredential, создаем Email, с таким userCredId.
    5.4. Ищем EmailConfirmation для этого объекта Email.
    5.5. Нашли -> удаляем. И создаем новый
    5.6. Не нашли -> просто создаем новый
    5.7. Отправка сообщения на почту для подтверждения регистрации с query параметрами
6. /login(email, password, deviceId). Ответ сервера: Token/Error