CT313H: WEB TECHNOLOGIES AND SERVICES

Building ContactBook App - Backend - Part 2

You will build a contact management app as a SPA app. The tech stack includes *Nodejs/Express, Knex.js, MySQL/MariaDB* for the backend (API server), and *Vue.js* for the frontend (GUI). In the first two lab sessions, you will build the API server for the app.

The API server must support the following requests:

- *POST /api/contacts*: creates a new contact
- *GET /api/contacts*: returns all contacts from the database. This endpoint supports the following optional parameters:
  - *favorite* and *name* are for querying favorite contacts and contacts filtered by name. For example, *GET /api/contacts?favorite&name=duy* returns favorite contacts named "duy"
  - *page* and *limit* are for pagination
- *DELETE /api/contacts*: deletes all contacts in the database
- *GET /api/contacts/<contact-id>*: gets a contact with a specific ID
- *PUT /api/contacts/<contact-id>*: updates a contact with a specific ID
- *DELETE /api/contacts/<contact-id>*: deletes a contact with a specific ID
- All requests for undefined URLs will result in a 404 error with the message "Resource not found"

A contact has the following information: *name (string), email (string), address (string), phone (string), favorite (boolean),* and *avatar (string)*. **The data format used for client-server communication is either JSON or multipart/form-data**. The OpenAPI spec (Swagger) is used to document our API. The source code is managed by git and uploaded to GitHub.

This step-by-step guide will help implement all the above requirements. However, students are free to make their own implementation as long as the requirements are met.

Requirements for the lab report:

- The submitted report is a PDF file containing images showing the results of your works (e.g., images showing the implemented features, successful and failed scenarios, results of the operations, ...). **You should NOT screenshot the source code**.
- **You only need to create ONE report for the whole four lab sessions**. At the end of each lab session, students need to (1) submit the work-in-progress report and (2) push the code to the GitHub repository given by the instructor.
- The report should also filled with student information (student ID, student name, class ID) and the links to the GitHub repositories.
- Plagiarism will result in 0.

(Continue from the result of Part 1)

Step 1: Prepare database

- Install MySQL or MariaDB on your machine if needed.
- Use a MySQL client (phpMyAdmin, HeidiSQL, ...) to create a database named *ct313h_labs*. Next, create a *contacts* table as follows (you can also leverage knex migration for this task):

```sql
CREATE TABLE `contacts` (
    `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    `email` VARCHAR(255) DEFAULT NULL,
    `address` VARCHAR(255) DEFAULT NULL,
    `phone` VARCHAR(15) DEFAULT NULL,
    `favorite` TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
    `avatar` VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

- Install `knex`, `mysql` and `faker-js` (checkout fake-js API here):

```
npm install knex mysql
npm install @faker-js/faker --save-dev
```

- Edit *.env* and add the database connection parameters:

```
PORT=3000

DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASS=root
DB_NAME=ct313h_labs
```

  Make sure to update the above parameters (user/password) according to your database setup.

- In the project directory, create directory *seeds* and run `npx knex init` to create *knexfile.js* file. Edit *knexfile.js* as follows:

```js
require('dotenv').config();
const { DB_HOST, DB_PORT, DB_USER, DB_PASS, DB_NAME } = process.env;

/**
 * @type { import("knex").Knex.Config }
 */
module.exports = {
    client: 'mysql',
    connection: {
        host: DB_HOST,
        port: DB_PORT,
        user: DB_USER,
        password: DB_PASS,
        database: DB_NAME,
    },
    pool: { min: 0, max: 10 },
    seeds: {
        directory: './seeds',
    },
};
```

- Run `npx knex seed:make contacts_seed` to create a seeding script for the contacts table (*./seeds/contacts_seed.js*). Edit the seeding script as follows:

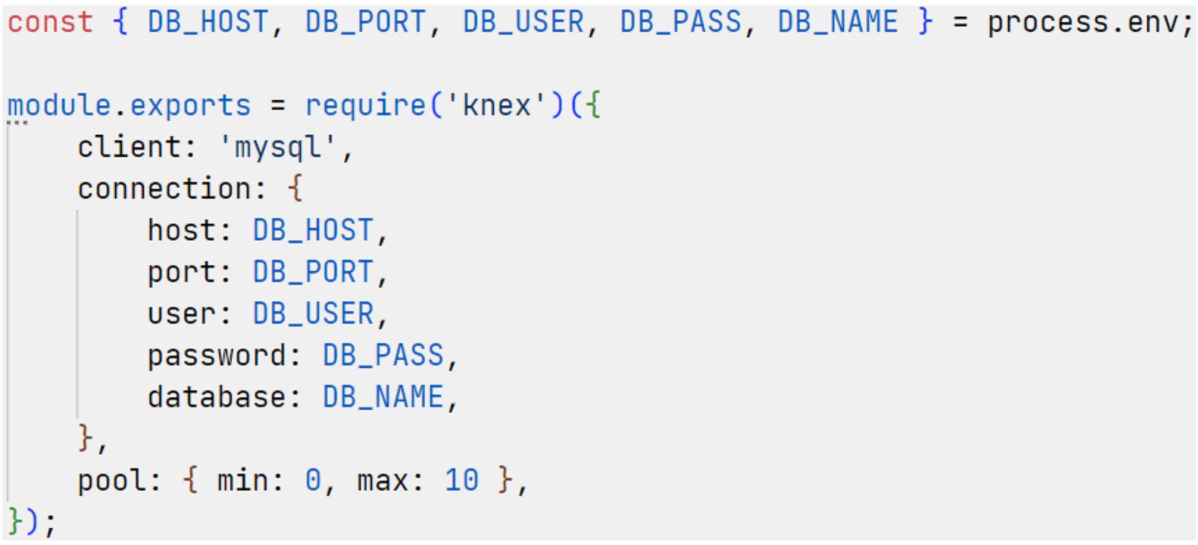
```
const { faker } = require('@faker-js/faker');

function createContact() {
    return {
        name: faker.person.fullName(),
        email: faker.internet.email(),
        address: faker.location.streetAddress(),
        phone: faker.string.numeric('09########'),
        favorite: faker.number.int({
            min: 0,
            max: 1,
        }),
        avatar: '/public/images/blank-profile-picture.png',
    };
}

/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.seed = async function (knex) {
    await knex('contacts').del();
    await knex('contacts').insert(Array(100).fill().map(createContact));
};
```

- Run the seeding scripts in the *seeds* directory by the command: `npx knex seed:run`.
- Verify that fake data are inserted into the database.
- After verification, commit changes to git:

```
git add -u
git add seeds knexfile.js
git commit -m "Setup knex.js and insert fake data"
```

Step 2: Implement route handlers

- Define a module that creates a knex object representing the connection to the database in *src/database/knex.js*:



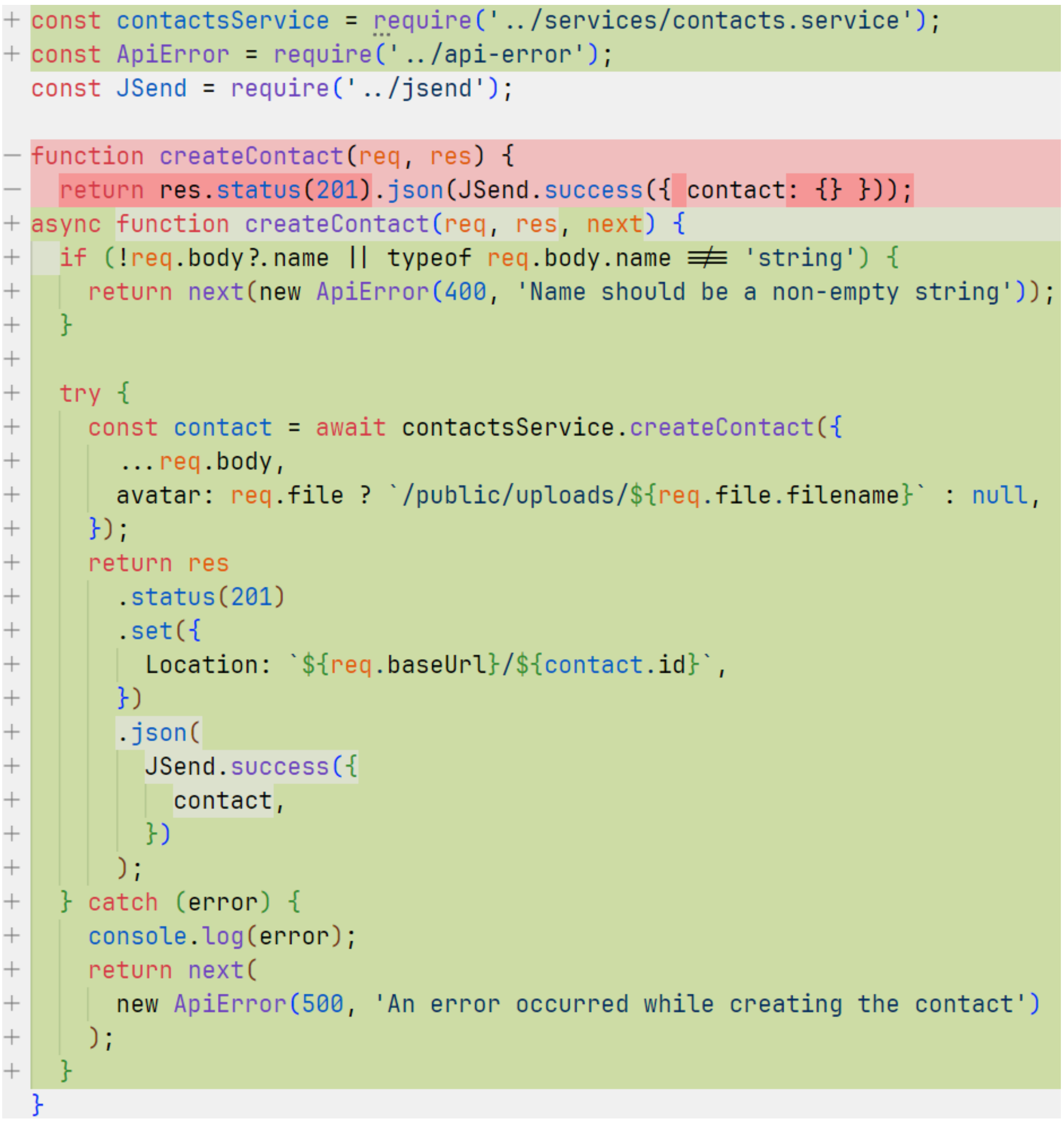
```
const { DB_HOST, DB_PORT, DB_USER, DB_PASS, DB_NAME } = process.env;

module.exports = require('knex')({
    client: 'mysql',
    connection: {
        host: DB_HOST,
        port: DB_PORT,
        user: DB_USER,
        password: DB_PASS,
        database: DB_NAME,
    },
    pool: { min: 0, max: 10 },
});
```

- Create *src/services/contacts.service.js* file to define a set of functions for accessing the database:

```
const knex = require('../database/knex');

function contactRepository() {
    return knex('contacts');
}

function readContact(payload) {
    return {
        name: payload.name,
        email: payload.email,
        address: payload.address,
        phone: payload.phone,
        favorite: payload.favorite,
        avatar: payload.avatar,
    };
}

// Define functions for accessing the database

module.exports = {
    // Export defined functions
}
```

Step 2.1: Implement createContact handler

- Edit *src/controllers/contacts.controller.js*:



```diff
+ const contactsService = require('../services/contacts.service');
+ const ApiError = require('../api-error');
  const JSend = require('../jsend');

- function createContact(req, res) {
-   return res.status(201).json(JSend.success({ contact: {} }));
+ async function createContact(req, res, next) {
+   if (!req.body?.name || typeof req.body.name !== 'string') {
+     return next(new ApiError(400, 'Name should be a non-empty string'));
+   }
+
+   try {
+     const contact = await contactsService.createContact({
+       ...req.body,
+       avatar: req.file ? `/public/uploads/${req.file.filename}` : null,
+     });
+     return res
+       .status(201)
+       .set({
+         Location: `${req.baseUrl}/${contact.id}`,
+       })
+       .json(
+         JSend.success({
+           contact,
+         })
+       );
+   } catch (error) {
+     console.log(error);
+     return next(
+       new ApiError(500, 'An error occurred while creating the contact')
+     );
+   }
  }
```

In case of an error, the call *next(error)* will transfer the execution to the error handling middleware defined in *src/app.js*.

- *contactsService.createContact()* stores the submitted contact to the database. The function *createContact()* is defined (in *src/services/contacts.service.js*) as follows:

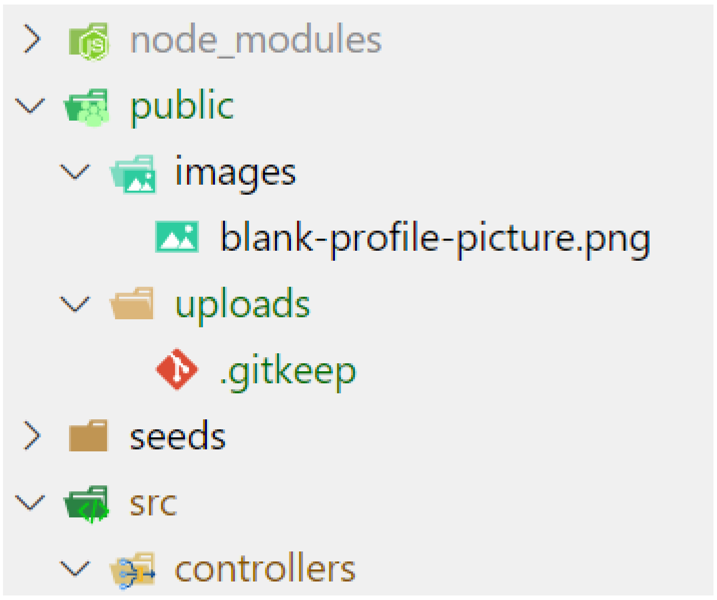
```
...

// Define functions for accessing the database

async function createContact(payload) {
    const contact = readContact(payload);
    const [id] = await contactRepository().insert(contact);
    return { id, ...contact };
}

module.exports = {
    // Export defined functions
    createContact
}
```

- Note that the create contact endpoint can accept an upload file field (the *avatarFile* field). `express.js` doesn't automatically handle upload files for you. It means that the expression `` req.file ? `/public/uploads/${req.file.filename}` : null `` will only be evaluated to null. To handle the *avatarFile* field, we can use a middleware called *multer*.
  - Install multer package: `npm i multer`
  - Create *src/middlewares/avatar-upload.middleware.js* file:

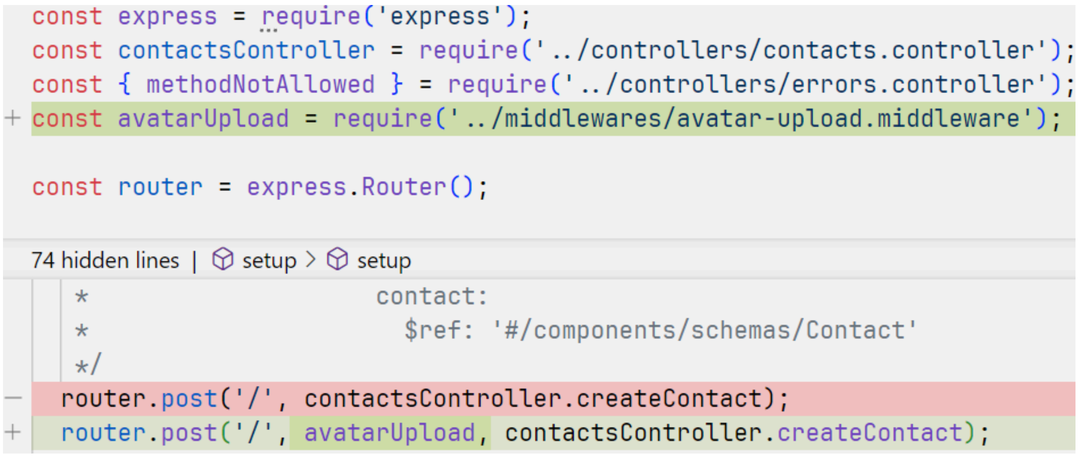
```
 1  const multer = require('multer');
 2  const path = require('path');
 3  const ApiError = require('../api-error');
 4
 5  const storage = multer.diskStorage({
 6    destination: function (req, file, cb) {
 7      cb(null, './public/uploads/');
 8    },
 9    filename: function (req, file, cb) {
10      const uniquePrefix = Date.now() + '-' + Math.round(Math.random() * 1e9);
11      cb(null, uniquePrefix + path.extname(file.originalname));
12    },
13  });
14
15  function avatarUpload(req, res, next) {
16    const upload = multer({ storage: storage }).single('avatarFile');
17
18    upload(req, res, function (err) {
19      if (err instanceof multer.MulterError) {
20        return next(
21          new ApiError(400, 'An error occurred while uploading the avatar')
22        );
23      } else if (err) {
24        return next(
25          new ApiError(
26            500,
27            'An unknown error occurred while uploading the avatar'
28          )
29        );
30      }
31
32      next();
33    });
34  }
35
36  module.exports = avatarUpload;
```

    Lines 5-13 configure where the upload files will go (the *destination* key) and how the upload files are named (the *filename* key). Lines 15-34 define a middleware function that handles a single-file upload field named *avatarFile* in the request. In other words, the *avatarUpload* middleware will parse a *multipart/form-data* request, read the upload file from the *avatarFile* field, and put it in the configured destination (the *public/uploads* directory).

  - Create an empty directory *public/uploads* for storing the upload files. As git cannot keep track of empty directories, create an empty file called *.gitkeep* inside *uploads*:



- Edit *src/routes/contacts.router.js* to add the *avatarUpload* middleware to the create contact route:

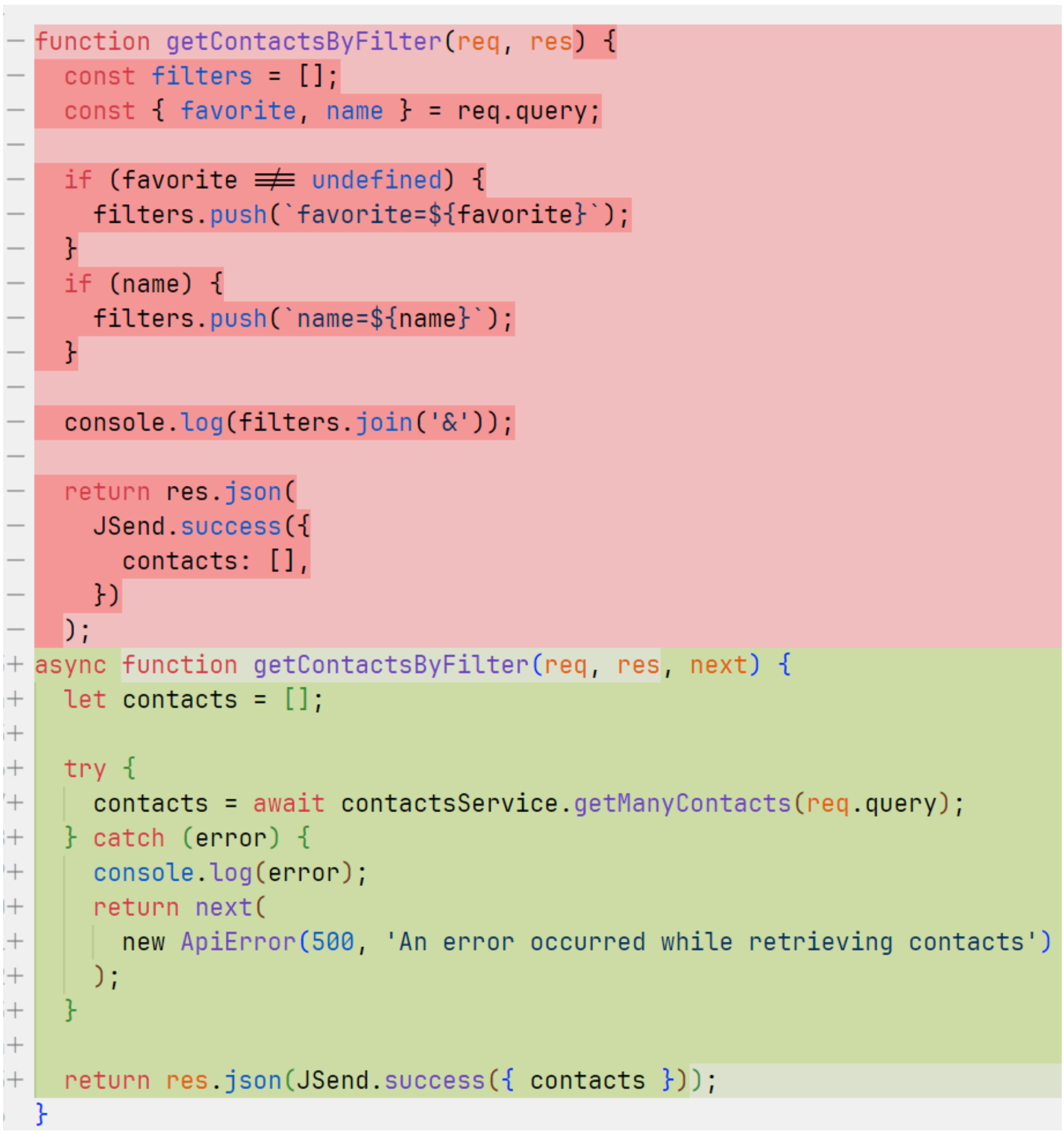


```
const express = require('express');
const contactsController = require('../controllers/contacts.controller');
const { methodNotAllowed } = require('../controllers/errors.controller');
+ const avatarUpload = require('../middlewares/avatar-upload.middleware');

const router = express.Router();

74 hidden lines | setup > setup
 *                     contact:
 *                       $ref: '#/components/schemas/Contact'
 */
- router.post('/', contactsController.createContact);
+ router.post('/', avatarUpload, contactsController.createContact);
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works as expected with and without the avatarFile field. Use your fullname as the contact name to be created.

Step 2.2: Implement getContactsByFilter handler

- Edit *src/controllers/contacts.controller.js*:



```diff
- function getContactsByFilter(req, res) {
-   const filters = [];
-   const { favorite, name } = req.query;
-
-   if (favorite !== undefined) {
-     filters.push(`favorite=${favorite}`);
-   }
-   if (name) {
-     filters.push(`name=${name}`);
-   }
-
-   console.log(filters.join('&'));
-
-   return res.json(
-     JSend.success({
-       contacts: [],
-     })
-   );
+ async function getContactsByFilter(req, res, next) {
+   let contacts = [];
+
+   try {
+     contacts = await contactsService.getManyContacts(req.query);
+   } catch (error) {
+     console.log(error);
+     return next(
+       new ApiError(500, 'An error occurred while retrieving contacts')
+     );
+   }
+
+   return res.json(JSend.success({ contacts }));
  }
```

- *contactsService.getManyContacts(query)* returns contacts filtered by the name and favorite query parameters. This function can be defined as follows:

```
...

function getManyContacts(query) {
    const { name, favorite } = query;

    return contactRepository()
        .where((builder) => {
            if (name) {
                builder.where('name', 'like', `%${name}%`);
            }
            if (favorite !== undefined &&
                favorite !== '0' &&
                favorite !== 'false') {
                builder.where('favorite', 1);
            }
        })
        .select('*');
}
```

```
module.exports = {
    createContact,
    getManyContacts
}
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works as expected.
- We don't want to get all of the records at once. We would like to paginate the records according to the page and limit query parameters in *getManyContacts(query)*:
  - Define a class named *Paginator* (in *src/services/paginator.js*):

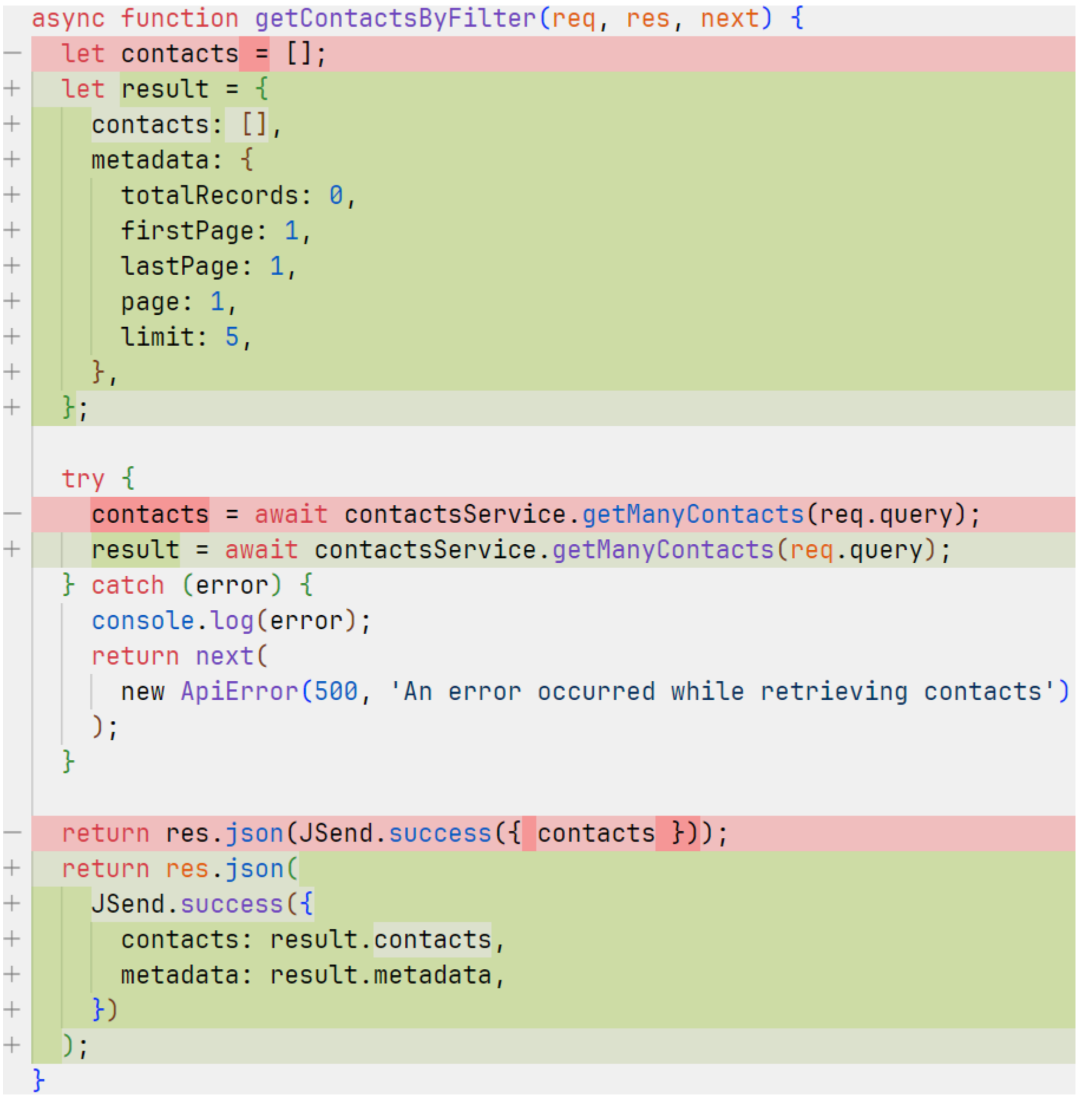
    ```
    class Paginator {
        constructor(page = 1, limit = 5) {
            this.limit = parseInt(limit, 10);
            if (isNaN(this.limit) || this.limit < 1) {
                this.limit = 5;
            }

            this.page = parseInt(page, 10);
            if (isNaN(this.limit) || this.page < 1) {
                this.page = 1;
            }

            this.offset = (this.page - 1) * this.limit;
        }

        getMetadata(totalRecords) {
            if (totalRecords === 0) {
                return {};
            }

            let totalPages = Math.ceil(totalRecords / this.limit);
            return {
                totalRecords,
                firstPage: 1,
                lastPage: totalPages,
                page: this.page,
                limit: this.limit,
            };
        }
    }

    module.exports = Paginator;
    ```

  - Edit *getManyContacts(query)* (in *src/services/contacts.service.js*) as follows:

    ```
    ...
    const Paginator = require('./paginator');

    ...
    async function getManyContacts(query) {
        const { name, favorite, page = 1, limit = 5 } = query;
        const paginator = new Paginator(page, limit);
    ```

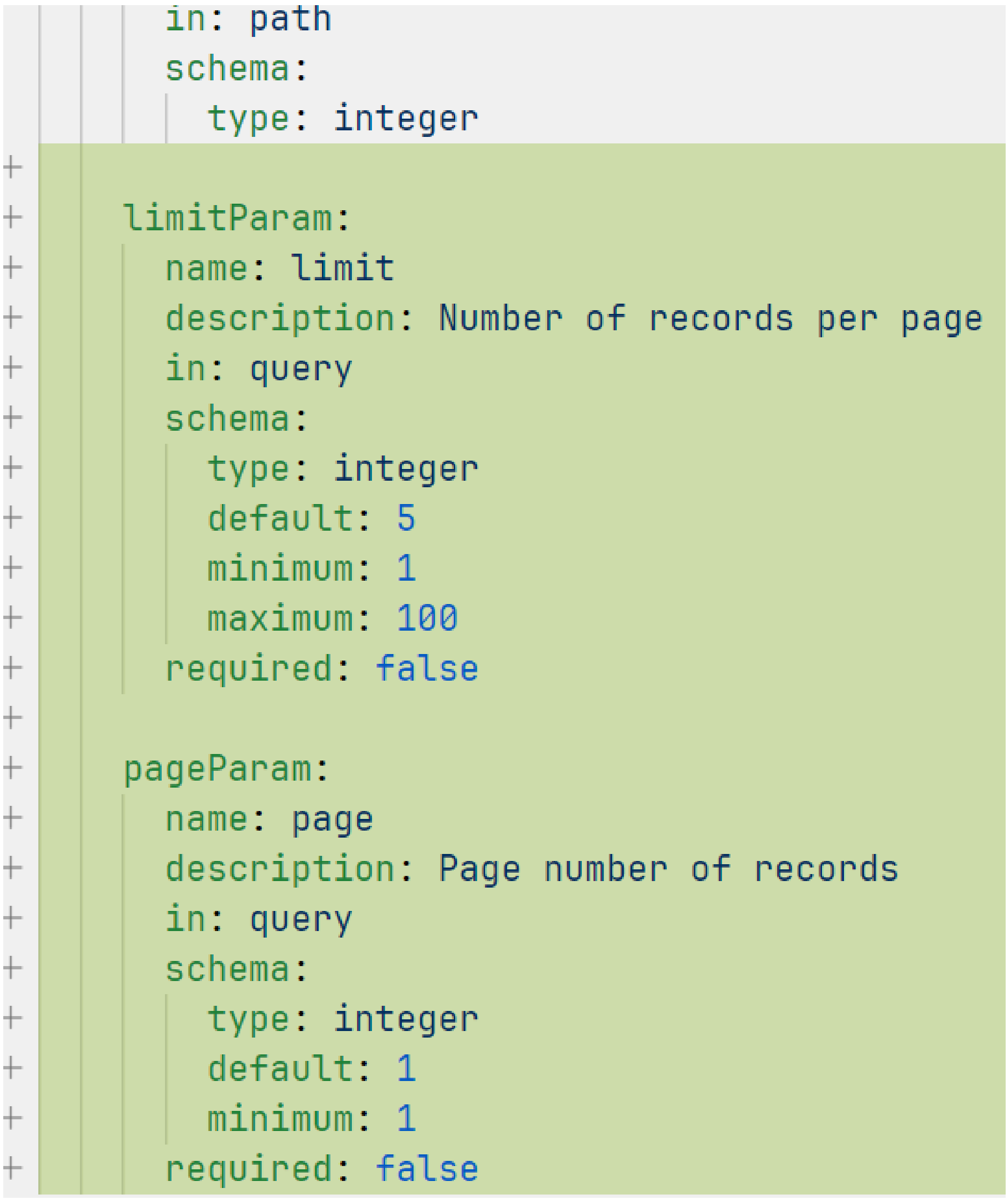
```
    let results = await contactRepository()
        .where((builder) => {
            if (name) {
                builder.where('name', 'like', `%${name}%`);
            }
            if (favorite !== undefined &&
                favorite !== '0' &&
                favorite !== 'false') {
                builder.where('favorite', 1);
            }
        })
        .select(
            knex.raw('count(id) OVER() AS recordCount'),
            'id',
            'name',
            'email',
            'address',
            'phone',
            'favorite',
            'avatar'
        )
        .limit(paginator.limit)
        .offset(paginator.offset);

    let totalRecords = 0;
    results = results.map((result) => {
        totalRecords = result.recordCount;
        delete result.recordCount;
        return result;
    });

    return {
        metadata: paginator.getMetadata(totalRecords),
        contacts: results,
    };
}
...
```

- Edit *getContactsByFilter()* in *src/controllers/contacts.controller.js* to include the pagination metadata in the response:



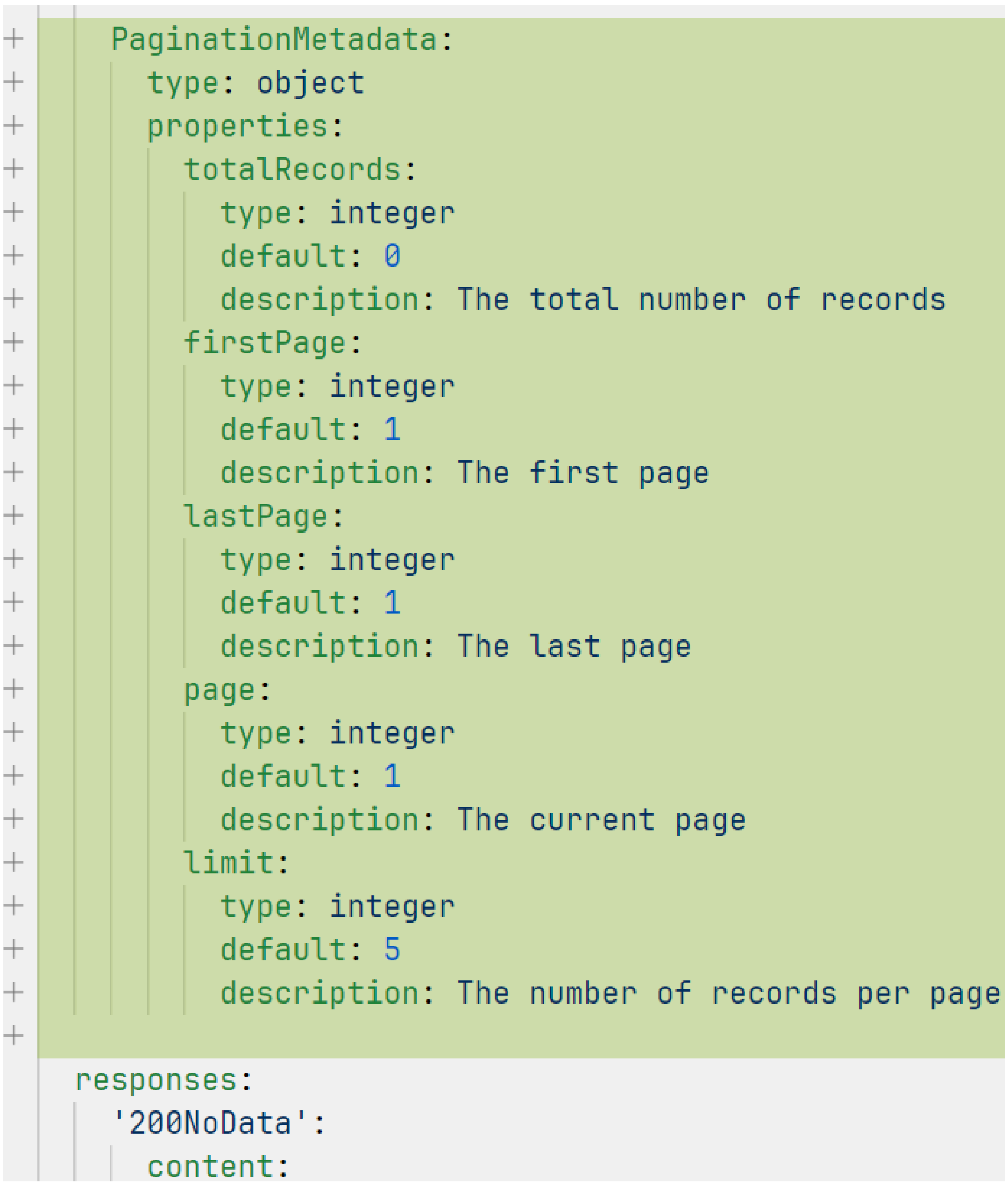
```
  async function getContactsByFilter(req, res, next) {
-   let contacts = [];
+   let result = {
+     contacts: [],
+     metadata: {
+       totalRecords: 0,
+       firstPage: 1,
+       lastPage: 1,
+       page: 1,
+       limit: 5,
+     },
+   };

    try {
-     contacts = await contactsService.getManyContacts(req.query);
+     result = await contactsService.getManyContacts(req.query);
    } catch (error) {
      console.log(error);
      return next(
        new ApiError(500, 'An error occurred while retrieving contacts')
      );
    }

-   return res.json(JSend.success({ contacts }));
+   return res.json(
+     JSend.success({
+       contacts: result.contacts,
+       metadata: result.metadata,
+     })
+   );
  }
```

- We need to edit the OpenAPI spec to reflect the new page and limit query parameters and the metadata included in the response. First edit the parameters component in *src/docs/components.yaml* to include the limit and page parameters:



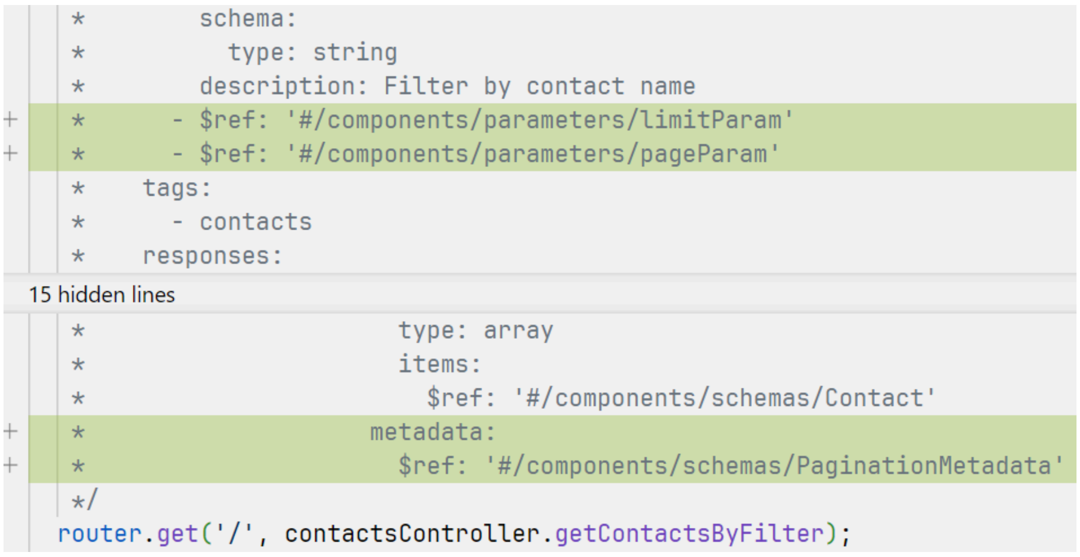
```
        in: path
        schema:
          type: integer
+
+    limitParam:
+      name: limit
+      description: Number of records per page
+      in: query
+      schema:
+        type: integer
+        default: 5
+        minimum: 1
+        maximum: 100
+      required: false
+
+    pageParam:
+      name: page
+      description: Page number of records
+      in: query
+      schema:
+        type: integer
+        default: 1
+        minimum: 1
+      required: false
```

Next, edit the schemas component to add the PaginationMetadata schema:



```
+     PaginationMetadata:
+       type: object
+       properties:
+         totalRecords:
+           type: integer
+           default: 0
+           description: The total number of records
+         firstPage:
+           type: integer
+           default: 1
+           description: The first page
+         lastPage:
+           type: integer
+           default: 1
+           description: The last page
+         page:
+           type: integer
+           default: 1
+           description: The current page
+         limit:
+           type: integer
+           default: 5
+           description: The number of records per page
+
    responses:
      '200NoData':
        content:
```

Finally, edit the JSDoc for the route:

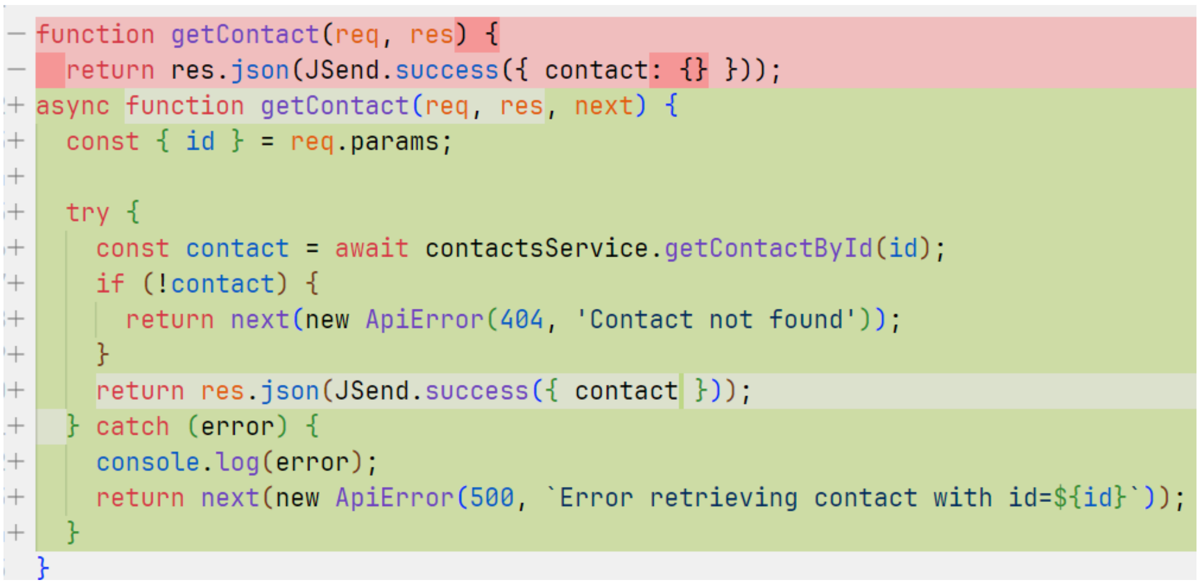


```
  *        schema:
  *          type: string
  *        description: Filter by contact name
+ *      - $ref: '#/components/parameters/limitParam'
+ *      - $ref: '#/components/parameters/pageParam'
  *    tags:
  *      - contacts
  *    responses:
15 hidden lines
  *                      type: array
  *                      items:
  *                        $ref: '#/components/schemas/Contact'
+ *                    metadata:
+ *                      $ref: '#/components/schemas/PaginationMetadata'
  */
 router.get('/', contactsController.getContactsByFilter);
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works correctly with different sets of page and limit parameters.

Step 2.3: Implement getContact handler

- Edit *src/controllers/contacts.controller.js*:



```diff
- function getContact(req, res) {
-   return res.json(JSend.success({ contact: {} }));
+ async function getContact(req, res, next) {
+   const { id } = req.params;
+
+   try {
+     const contact = await contactsService.getContactById(id);
+     if (!contact) {
+       return next(new ApiError(404, 'Contact not found'));
+     }
+     return res.json(JSend.success({ contact }));
+   } catch (error) {
+     console.log(error);
+     return next(new ApiError(500, `Error retrieving contact with id=${id}`));
+   }
  }
```

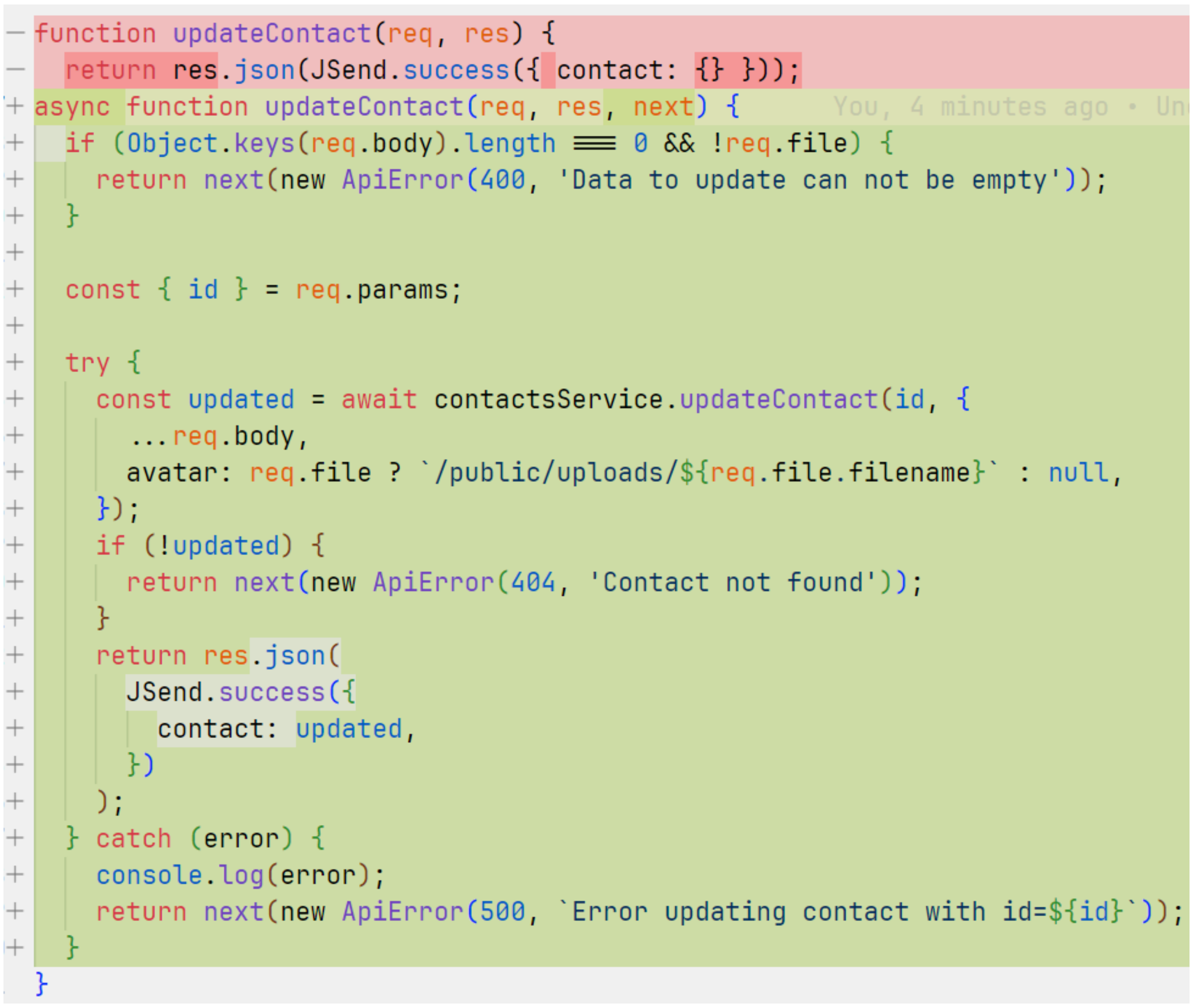
- *contactsService.getContactById(id)* finds a contact by ID. The function *getContactById(id)* can be defined as follows:

```
...

async function getContactById(id) {
    return contactRepository().where('id', id).select('*').first();
}

module.exports = {
    createContact,
    getManyContacts,
    getContactById
}
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works correctly.

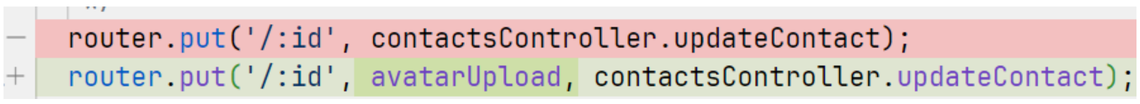
Step 2.4: Implement updateContact handler

- Edit *src/controllers/contacts.controller.js*:



```diff
- function updateContact(req, res) {
-   return res.json(JSend.success({ contact: {} }));
+ async function updateContact(req, res, next) {      You, 4 minutes ago • Un
+   if (Object.keys(req.body).length === 0 && !req.file) {
+     return next(new ApiError(400, 'Data to update can not be empty'));
+   }
+
+   const { id } = req.params;
+
+   try {
+     const updated = await contactsService.updateContact(id, {
+       ...req.body,
+       avatar: req.file ? `/public/uploads/${req.file.filename}` : null,
+     });
+     if (!updated) {
+       return next(new ApiError(404, 'Contact not found'));
+     }
+     return res.json(
+       JSend.success({
+         contact: updated,
+       })
+     );
+   } catch (error) {
+     console.log(error);
+     return next(new ApiError(500, `Error updating contact with id=${id}`));
+   }
  }
```

- Add *avatarUpload* middleware to the update contact route:



```diff
-   router.put('/:id', contactsController.updateContact);
+   router.put('/:id', avatarUpload, contactsController.updateContact);
```

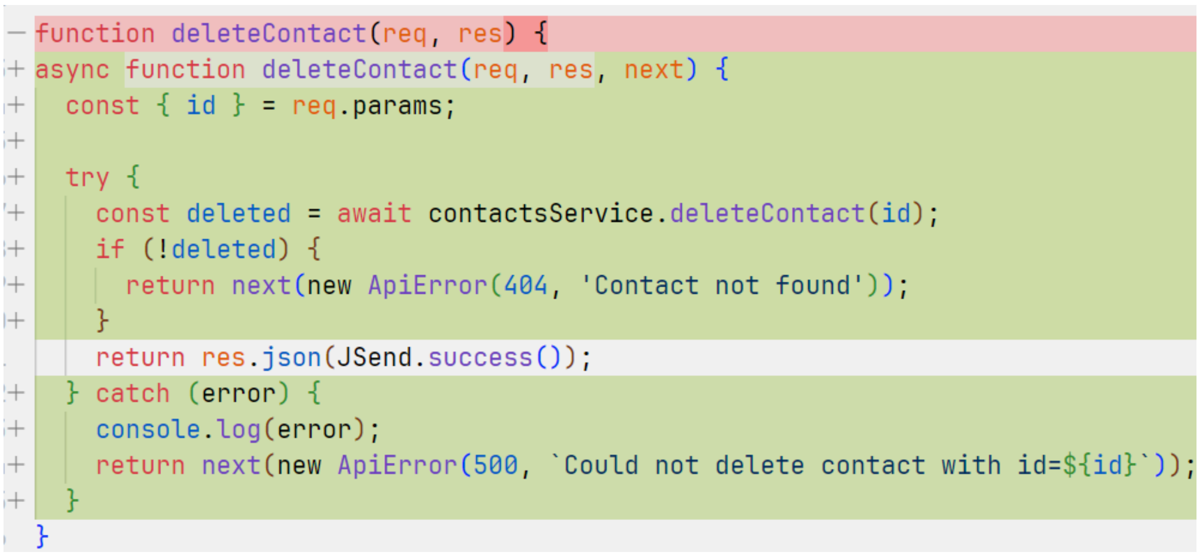
- *contactsService.updateContact(id, payload)* finds a contact by ID and updates this contact with *payload*. The function *updateContact(id, payload)* can be defined as follows:

```
const { unlink } = require('node:fs');

...

async function updateContact(id, payload) {
    const updatedContact = await contactRepository()
        .where('id', id)
        .select('*')
        .first();

    if (!updatedContact) {
        return null;
    }

    const update = readContact(payload);
    if (!update.avatar) {
        delete update.avatar;
    }

    await contactRepository().where('id', id).update(update);
    if (
        update.avatar &&
        updatedContact.avatar &&
        update.avatar !== updatedContact.avatar &&
        updatedContact.avatar.startsWith('/public/uploads')
    ) {
        unlink(`.${updatedContact.avatar}`, (err) => {});
    }

    return { ...updatedContact, ...update };
}

module.exports = {
    createContact,
    getManyContacts,
    getContactById,
    updateContact
}
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works correctly.

Step 2.5: Implement deleteContact handler

- Edit *src/controllers/contacts.controller.js*:



```
- function deleteContact(req, res) {
+ async function deleteContact(req, res, next) {
+   const { id } = req.params;
+
+   try {
+     const deleted = await contactsService.deleteContact(id);
+     if (!deleted) {
+       return next(new ApiError(404, 'Contact not found'));
+     }
      return res.json(JSend.success());
+   } catch (error) {
+     console.log(error);
+     return next(new ApiError(500, `Could not delete contact with id=${id}`));
+   }
  }
```

- *contactsService.deleteContact(id)* finds a contact by ID and deletes this contact. The function *deleteContact(id)* can be defined as follows:

```
...

async function deleteContact(id) {
    const deletedContact = await contactRepository()
        .where('id', id)
        .select('avatar')
        .first();

    if (!deletedContact) {
        return null;
    }
    await contactRepository().where('id', id).del();

    if (
        deletedContact.avatar &&
        deletedContact.avatar.startsWith('/public/uploads')
    ) {
        unlink(`.${deletedContact.avatar}`, (err) => {});
    }

    return deletedContact;
}

module.exports = {
    createContact,
    getManyContacts,
    getContactById,
    updateContact,
    deleteContact
}
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works correctly.

Step 2.6: Implement deleteAllContacts handler

- Edit *src/controllers/contacts.controller.js*:

```diff
- function deleteAllContacts(req, res) {
+ async function deleteAllContacts(req, res, next) {
+   try {
+     await contactsService.deleteAllContacts();
+
      return res.json(JSend.success());
+   } catch (error) {
+     console.log(error);
+     return next(
+       new ApiError(500, 'An error occurred while removing all contacts')
+     );
+   }
  }
```

- *contactsService.deleteAllContacts()* removes all contacts. The function *deleteAllContacts()* can be defined as follows:

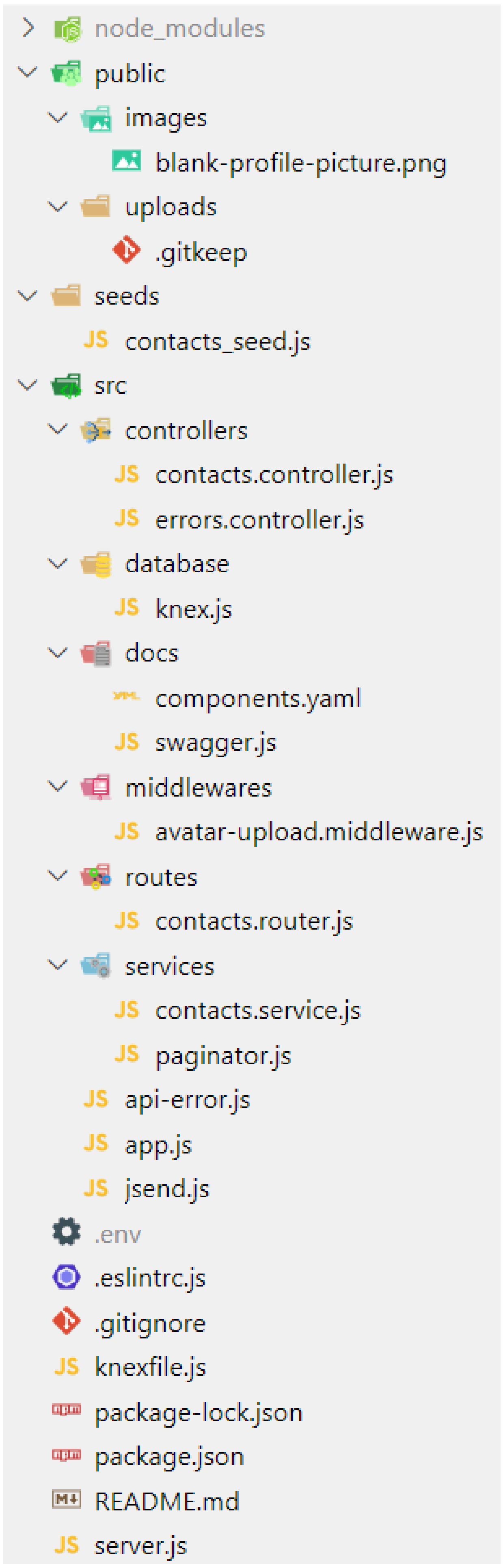
```
...

async function deleteAllContacts() {
    const contacts = await contactRepository().select('avatar');
    await contactRepository().del();

    contacts.forEach((contact) => {
        if (contact.avatar && contact.avatar.startsWith('/public/uploads')) {
            unlink(`.${contact.avatar}`, (err) => {});
        }
    });
}
```

```
module.exports = {
    createContact,
    getManyContacts,
    getContactById,
    updateContact,
    deleteContact,
    deleteAllContacts
}
```

- Use the Swagger UI (http://localhost:3000/api-docs/) to verify the handler works correctly.
- Make sure all handlers work correctly, then commit changes to git and GitHub:

```
git add -u
git add public/uploads src/database src/middlewares src/services
git commit -m "Implement handlers"
git push origin master  # Upload local commits to GitHub
```

The directory structure for the project currently is as follows:



```
> node_modules
v public
  v images
      blank-profile-picture.png
  v uploads
      .gitkeep
v seeds
    contacts_seed.js
v src
  v controllers
      contacts.controller.js
      errors.controller.js
  v database
      knex.js
  v docs
      components.yaml
      swagger.js
  v middlewares
      avatar-upload.middleware.js
  v routes
      contacts.router.js
  v services
      contacts.service.js
      paginator.js
    api-error.js
    app.js
    jsend.js
  .env
  .eslintrc.js
  .gitignore
  knexfile.js
  package-lock.json
  package.json
  README.md
  server.js
```

Step 3: Describing other response types

Currently, we only specify the 200 OK response type for the REST endpoints. Look at the endpoint handlers and describe other possible response types that the server can return to the client using OpenAPI spec.