Web Development Tutorial - Advanced Backend

Introduction

In backend development, there are a few key parts:

1) Database
2) API

The database stores all the data for the application. The API is a layer of abstraction on top of the database which does the following things:

1. Ensures the database is accessed securely/safely
2. Presents DB data in a cleaned/processed manner
3. Presents calculated information based on DB state

Anyone using the API does not need to worry about the actual DB implementation. They only need to know the inputs and outputs of the API.

In addition to these DB abstractions, backends can be responsible for many more things:

1. Interfacing with APIs from other systems
2. Manage authentication
3. Sending emails
4. Placing orders
5. lots more

API Design

API design is vital in software development, acting as a framework for how software applications interact. A well-crafted API facilitates development by hiding complexity, promoting scalability, and ensuring security. It involves creating clear endpoints, employing *standard HTTP methods* and *status codes*, and maintaining consistent naming conventions. Good API design also includes comprehensive documentation, version control, and a focus on the developer experience, enabling easy integration and a productive development environment.

Further, a good API logically groups endpoints into "resources". Resources are a collection of data or functions that are closely related. For example, if your backend manages a database of users, you might have a *users* resource. All API calls involving this resource would start with/users. For example,

- /users/add-to-cart
- /users/profile-pic
- /users/etc

Resources can also be nested. For example, you might have a product resource: /product. Each product (eg. t-shirt) might have several variants (eg. large, medium, small). You could structure this as /product/variant.

Watch this tutorial on effective API design.

Standard HTTP Methods and Status Codes

(These may seem abstract at first. It will make more sense when you create your first backend)

HTTP Method	**Description**	**Use Cases**
GET	Retrieves data from a specified resource.	Fetching documents, images, or specific data.
POST	Submits data to be processed to a specified resource.	Creating new resources (e.g., user registration).
PUT	Replaces all current representations of the target resource with the uploaded content.	Updating existing resources.
DELETE	Removes all current representations of the target resource given by a URI.	Deleting resources.
PATCH	Applies partial modifications to a resource.	Partially updating resources.
HEAD	Similar to GET, but it transfers the status line and header section only.	Retrieving meta-information.

OPTIONS	Describes the communication options for the target resource.	Discovering allowed methods on a resource.
CONNECT	Establishes a tunnel to the server identified by a given URI.	Used by proxies to enable SSL tunnels.
TRACE	Performs a message loop-back test along the path to the target resource.	Debugging at the protocol level.

Status Code	**Description**	**Category**
200	OK - The request has succeeded.	Success
201	Created - The request has been fulfilled and resulted in a new resource being created.	Success
400	Bad Request - The server cannot or will not process the request due to an apparent client error.	Client Error
401	Unauthorized - The request has not been applied because it lacks valid authentication credentials for the target resource.	Client Error
403	Forbidden - The server understood the request but refuses to authorize it.	Client Error
404	Not Found - The server can't find the requested resource.	Client Error
500	Internal Server Error - The server encountered an unexpected condition that prevented it from fulfilling the request.	Server Error

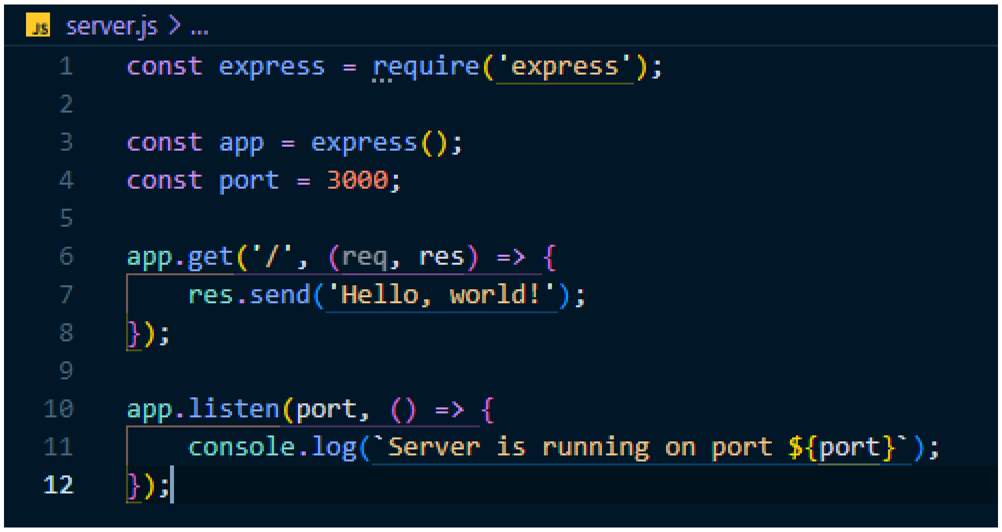
502	Bad Gateway - The server, while acting as a gateway or proxy, received an invalid response from the upstream server.	Server Error
503	Service Unavailable - The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded.	Server Error
504	Gateway Timeout - The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server.	Server Error

Express

Express is a JS framework commonly utilized for creating web applications and APIs.
Express has several organizational features that you should take advantage of:

1. server.js

This file is the entry point of your backend. All API requests will go through here. More specifically, through the app object:

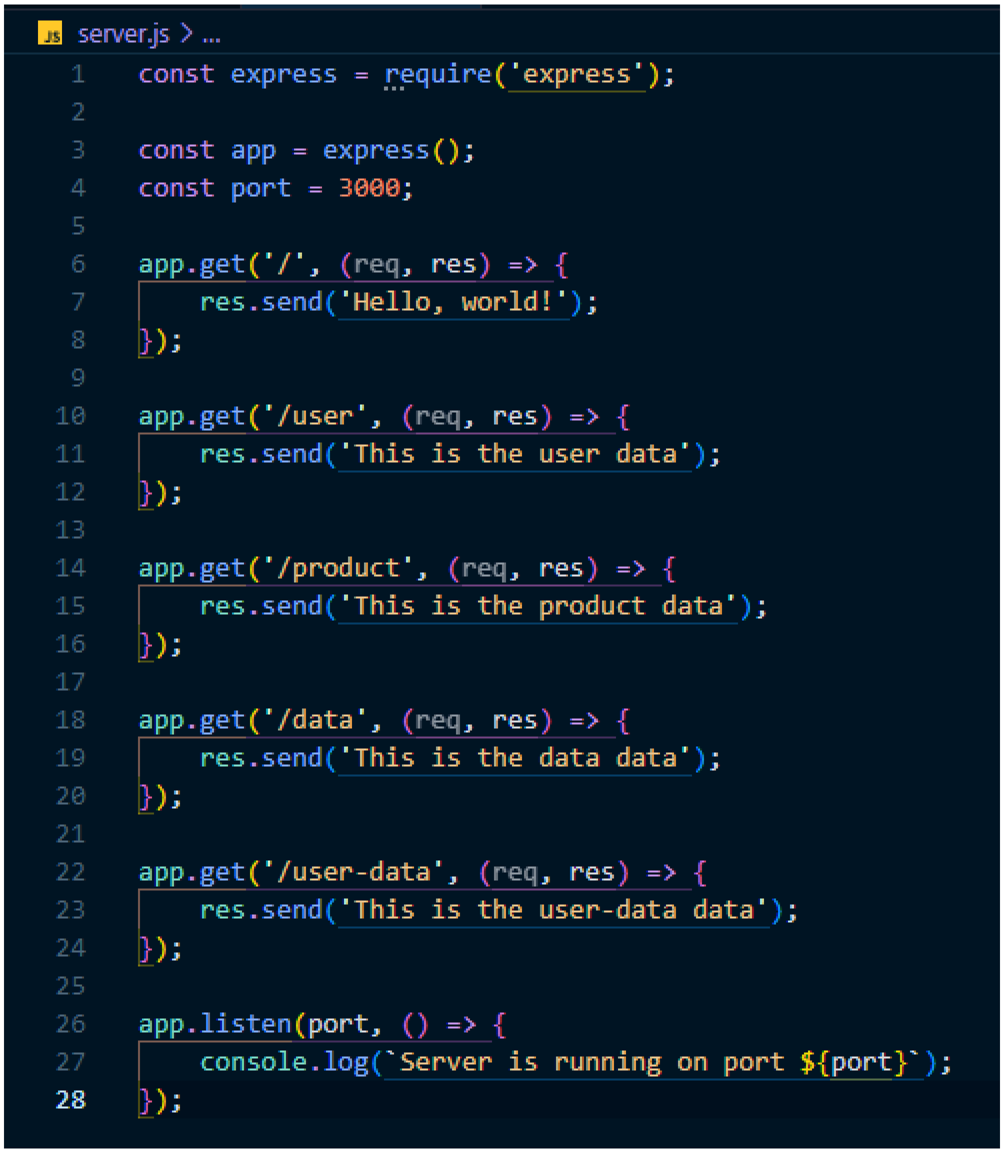


server.js > ...

```javascript
const express = require('express');

const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

In this example, all traffic (API calls) is routed into the app object. The app object then sends the API call to the appropriate API endpoint.

In this case, there is only 1 endpoint (the default "/" endpoint). More can be added:



```
const express = require('express');

const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.get('/user', (req, res) => {
    res.send('This is the user data');
});

app.get('/product', (req, res) => {
    res.send('This is the product data');
});

app.get('/data', (req, res) => {
    res.send('This is the data data');
});

app.get('/user-data', (req, res) => {
    res.send('This is the user-data data');
});

app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

2. Routes

As a project grows in complexity, this server file will become large and complex. Functionality will need to be abstracted in order to ensure the code is maintainable. One useful abstraction provided by Express, are *routes*. Routes can funnel related requests to a specific *route* object.

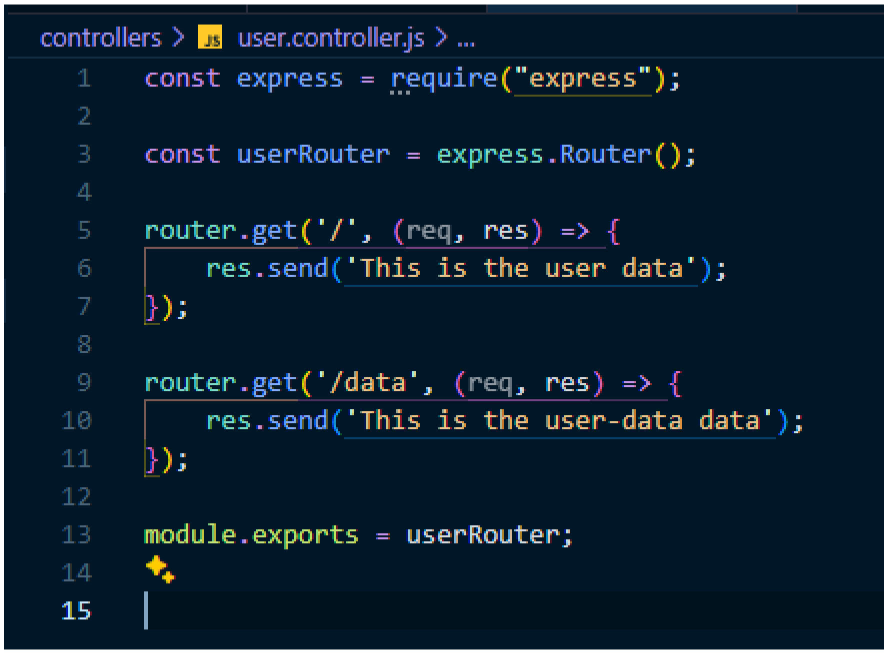
For example, in our code, we have a /user endpoint, and a /user-data endpoint. We can reorganize this to be:

- /user
- /user/data

Both of these endpoints start with the */user* path. We can create a route to handle all APIs under the /user path.

This reorganization makes our API easier to understand to its users and will make the code cleaner. The code will look as follows:

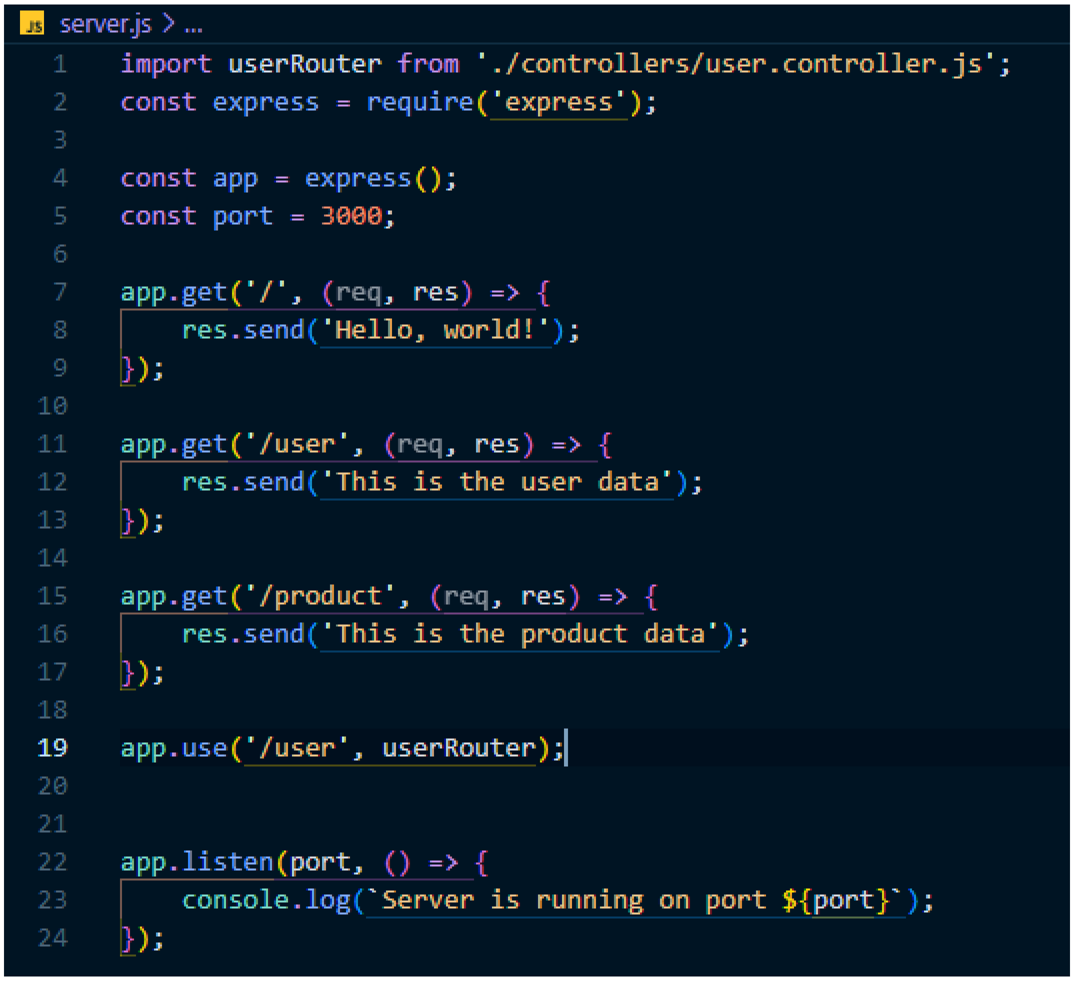
User Router



controllers > user.controller.js > ...

```
const express = require("express");

const userRouter = express.Router();

router.get('/', (req, res) => {
    res.send('This is the user data');
});

router.get('/data', (req, res) => {
    res.send('This is the user-data data');
});

module.exports = userRouter;


```

server.js


server.js > ...

```javascript
import userRouter from './controllers/user.controller.js';
const express = require('express');

const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.get('/user', (req, res) => {
    res.send('This is the user data');
});

app.get('/product', (req, res) => {
    res.send('This is the product data');
});

app.use('/user', userRouter);


app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

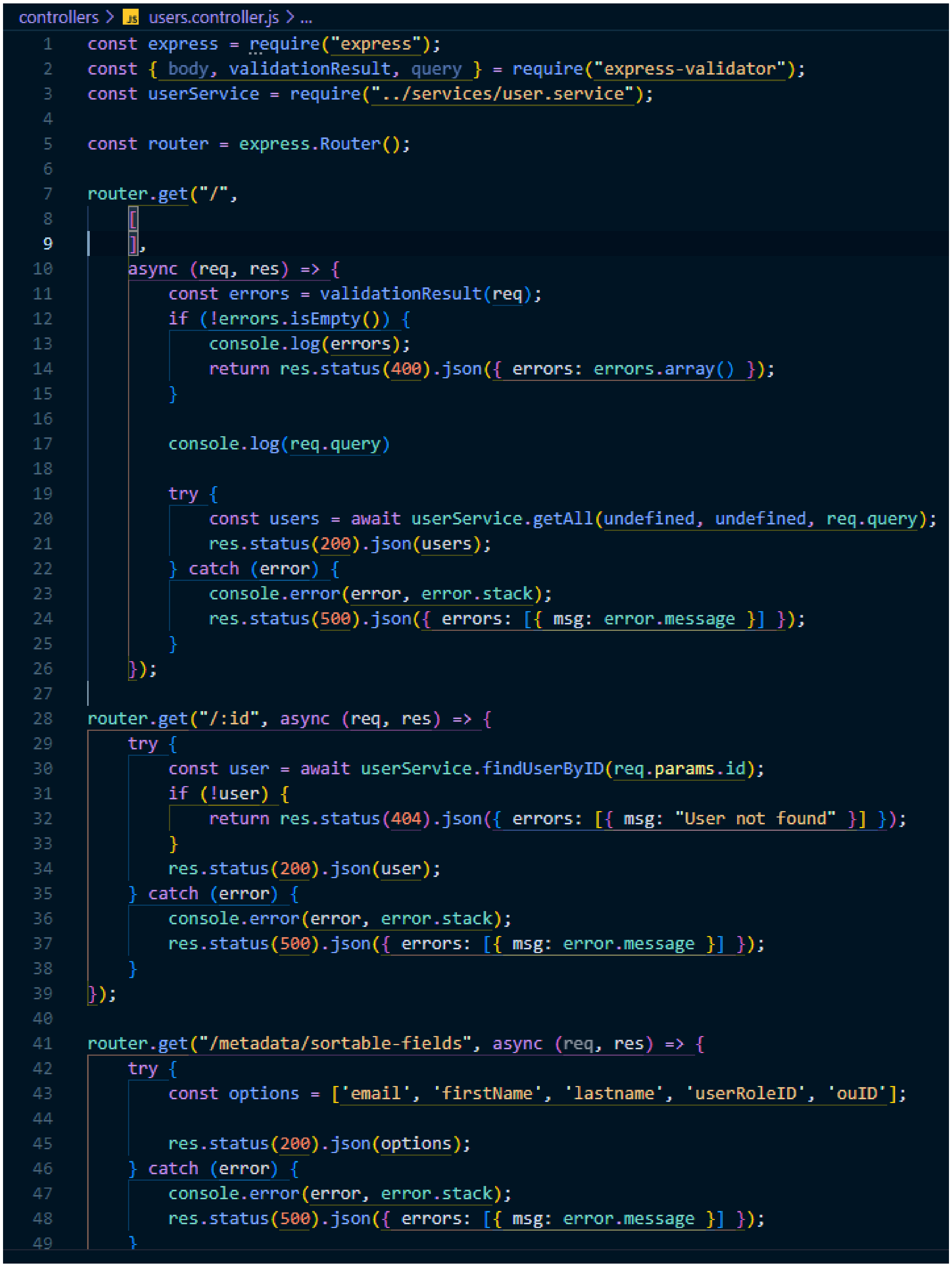
Now, if an API call is made to /user/data, the app will first receive the API call, and then pass it to the user router, which will then run the data endpoint.

Project Design

As a project grows in size, it is important to have a clear organizational structure. This structure will vary depending on the needs of the project, and there is no single structure that works best. The following structure is recommended as a guide/example.

```
Project
├── controllers
│   └── users.controller.js
├── services
│   └── users.service.js
├── utilities
│   └── jsonParser.util.js
├── models
│   └── user.model.js
└── server.js
```

Controllers:
Controllers (AKA Routers) are the endpoints of your API. For example, all API calls to /user/… would be routed to the users.controller.js.

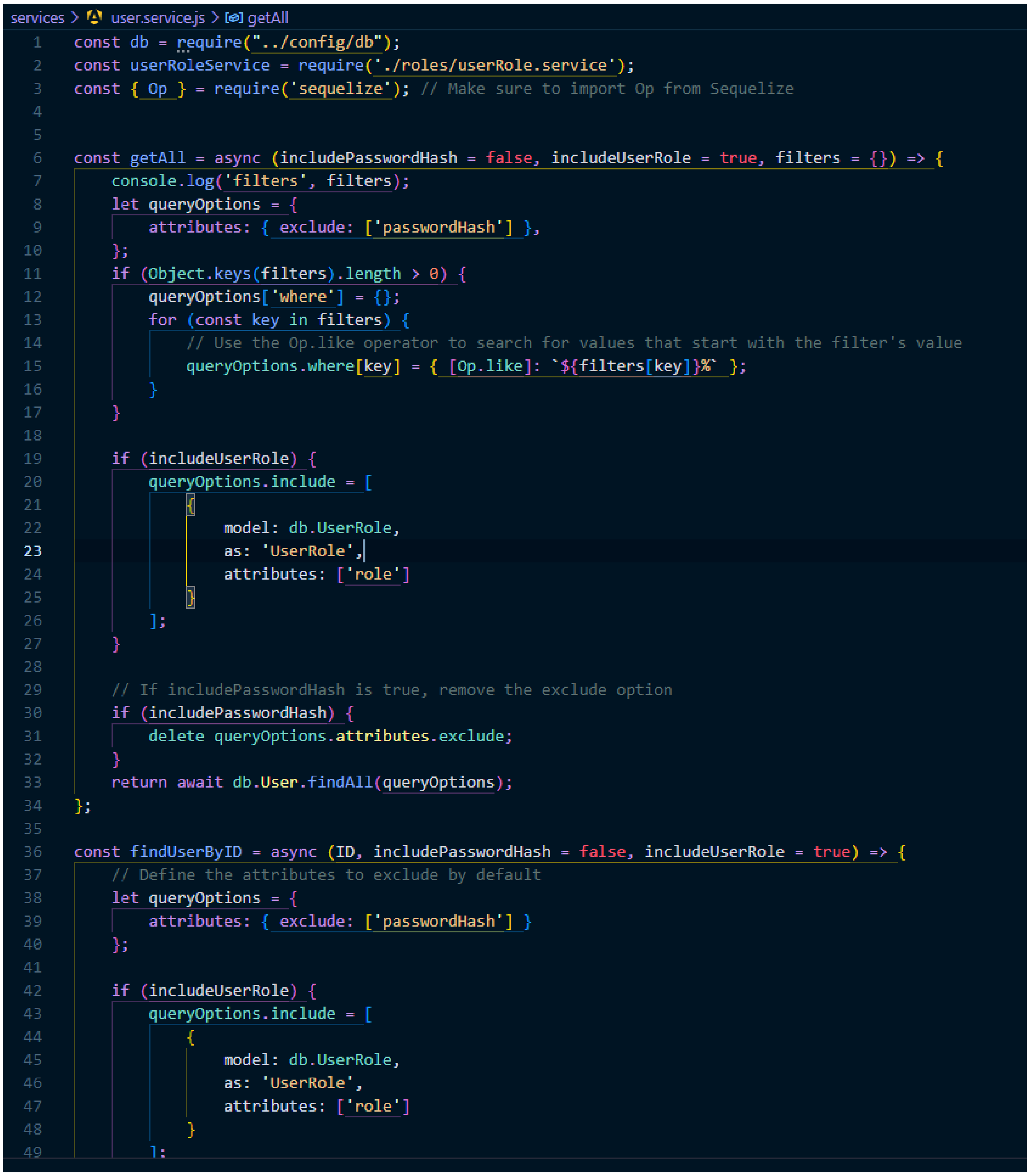


controllers > users.controller.js > ...

```javascript
const express = require("express");
const { body, validationResult, query } = require("express-validator");
const userService = require("../services/user.service");

const router = express.Router();

router.get("/",
    [
    ],
    async (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            console.log(errors);
            return res.status(400).json({ errors: errors.array() });
        }

        console.log(req.query)

        try {
            const users = await userService.getAll(undefined, undefined, req.query);
            res.status(200).json(users);
        } catch (error) {
            console.error(error, error.stack);
            res.status(500).json({ errors: [{ msg: error.message }] });
        }
    });

router.get("/:id", async (req, res) => {
    try {
        const user = await userService.findUserByID(req.params.id);
        if (!user) {
            return res.status(404).json({ errors: [{ msg: "User not found" }] });
        }
        res.status(200).json(user);
    } catch (error) {
        console.error(error, error.stack);
        res.status(500).json({ errors: [{ msg: error.message }] });
    }
});

router.get("/metadata/sortable-fields", async (req, res) => {
    try {
        const options = ['email', 'firstName', 'lastname', 'userRoleID', 'ouID'];

        res.status(200).json(options);
    } catch (error) {
        console.error(error, error.stack);
        res.status(500).json({ errors: [{ msg: error.message }] });
    }
```

Services:

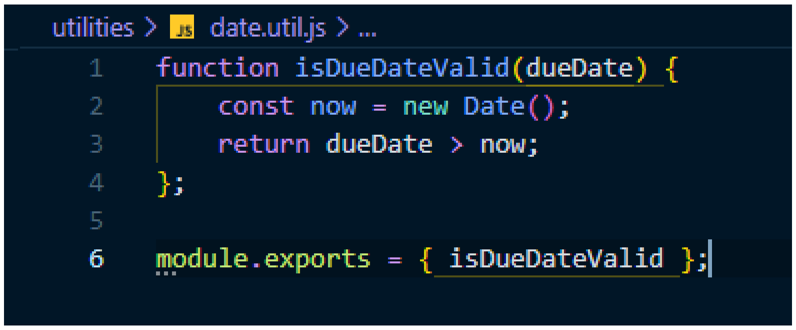
As time goes on, the code inside your controllers will become very long and complex, requiring you to refactor pieces into functions. Some of these functions can be turned into *services*. Services are a bundle of functions tied to a particular resource. For example, users.service.js has many functions that are related to users.


services > user.service.js > getAll

```
const db = require("../config/db");
const userRoleService = require('./roles/userRole.service');
const { Op } = require('sequelize'); // Make sure to import Op from Sequelize


const getAll = async (includePasswordHash = false, includeUserRole = true, filters = {}) => {
    console.log('filters', filters);
    let queryOptions = {
        attributes: { exclude: ['passwordHash'] },
    };
    if (Object.keys(filters).length > 0) {
        queryOptions['where'] = {};
        for (const key in filters) {
            // Use the Op.like operator to search for values that start with the filter's value
            queryOptions.where[key] = { [Op.like]: `${filters[key]}%` };
        }
    }

    if (includeUserRole) {
        queryOptions.include = [
            {
                model: db.UserRole,
                as: 'UserRole',
                attributes: ['role']
            }
        ];
    }

    // If includePasswordHash is true, remove the exclude option
    if (includePasswordHash) {
        delete queryOptions.attributes.exclude;
    }
    return await db.User.findAll(queryOptions);
};

const findUserByID = async (ID, includePasswordHash = false, includeUserRole = true) => {
    // Define the attributes to exclude by default
    let queryOptions = {
        attributes: { exclude: ['passwordHash'] }
    };

    if (includeUserRole) {
        queryOptions.include = [
            {
                model: db.UserRole,
                as: 'UserRole',
                attributes: ['role']
            }
        ];
```

Utilities:

You might have some functions that are more universal, and not tied to a specific resource. These can be categorized as utilities. For example, a date.util.js file can have some functions for date manipulation.



```javascript
function isDueDateValid(dueDate) {
    const now = new Date();
    return dueDate > now;
};

module.exports = { isDueDateValid };
```

Models:
Described Later

Databases

Generally, if you have a backend API, you also have a database.

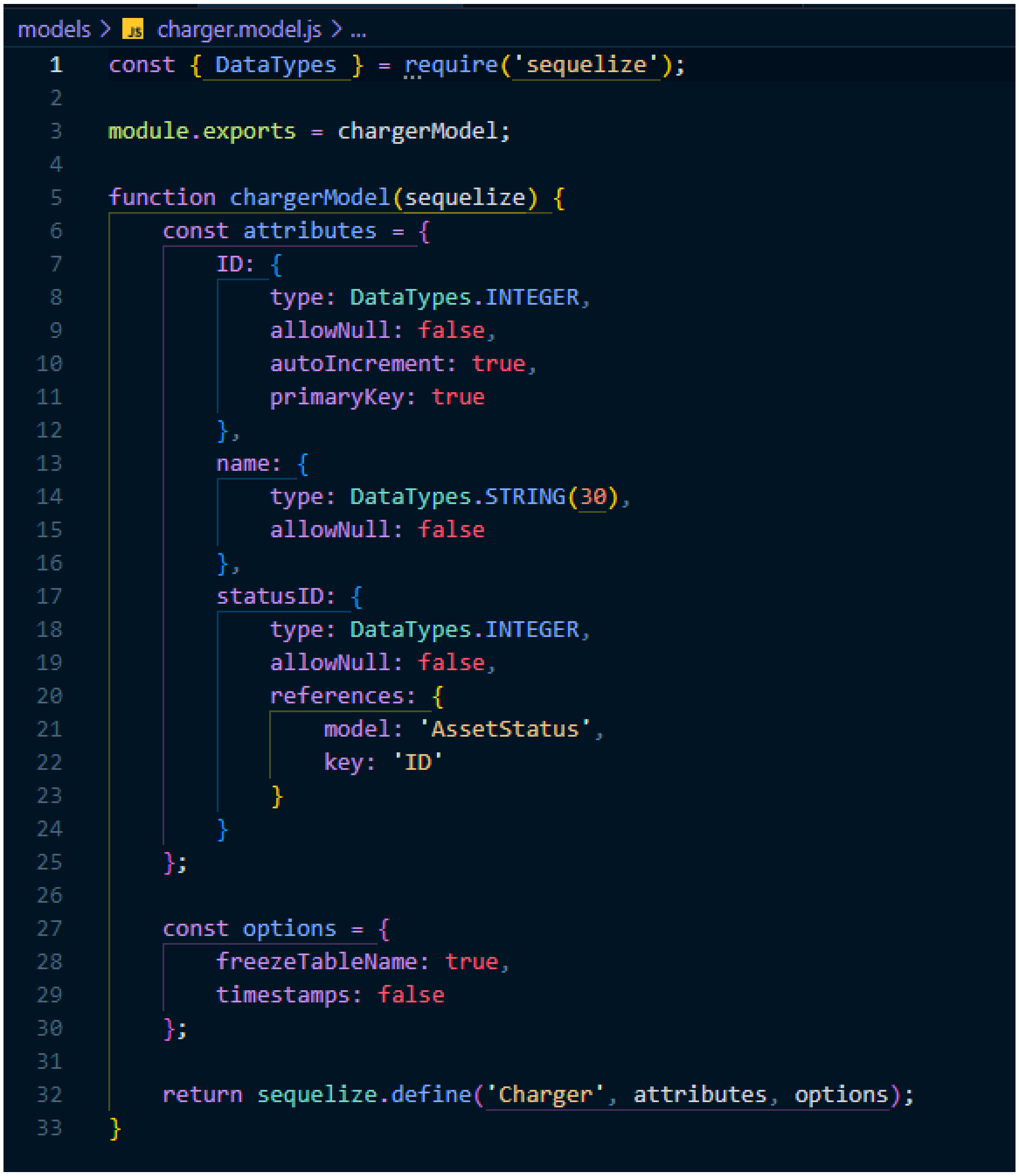
1. Follow this tutorial to set up a MySQL database.
2. Follow this tutorial to connect MySQL to your backend, using Sequelize

ORM

ORM - Object-Relational Mapping: A layer of abstraction on top of databases. This allows you to make database queries using syntax that is more familiar to most programmers, as well as offers various protections for your database. When working with a database, it is highly recommended to use an ORM.

For express applications, I recommend using Sequelize. It is compatible with most DB providers and relatively popular so there is lots of documentation.

In the previous section, I mentioned the Models directory of a project. Here is where you will define the different models (tables) of your database with your ORM.


models > charger.model.js > ...

```
const { DataTypes } = require('sequelize');

module.exports = chargerModel;

function chargerModel(sequelize) {
    const attributes = {
        ID: {
            type: DataTypes.INTEGER,
            allowNull: false,
            autoIncrement: true,
            primaryKey: true
        },
        name: {
            type: DataTypes.STRING(30),
            allowNull: false
        },
        statusID: {
            type: DataTypes.INTEGER,
            allowNull: false,
            references: {
                model: 'AssetStatus',
                key: 'ID'
            }
        }
    };

    const options = {
        freezeTableName: true,
        timestamps: false
    };

    return sequelize.define('Charger', attributes, options);
}
```

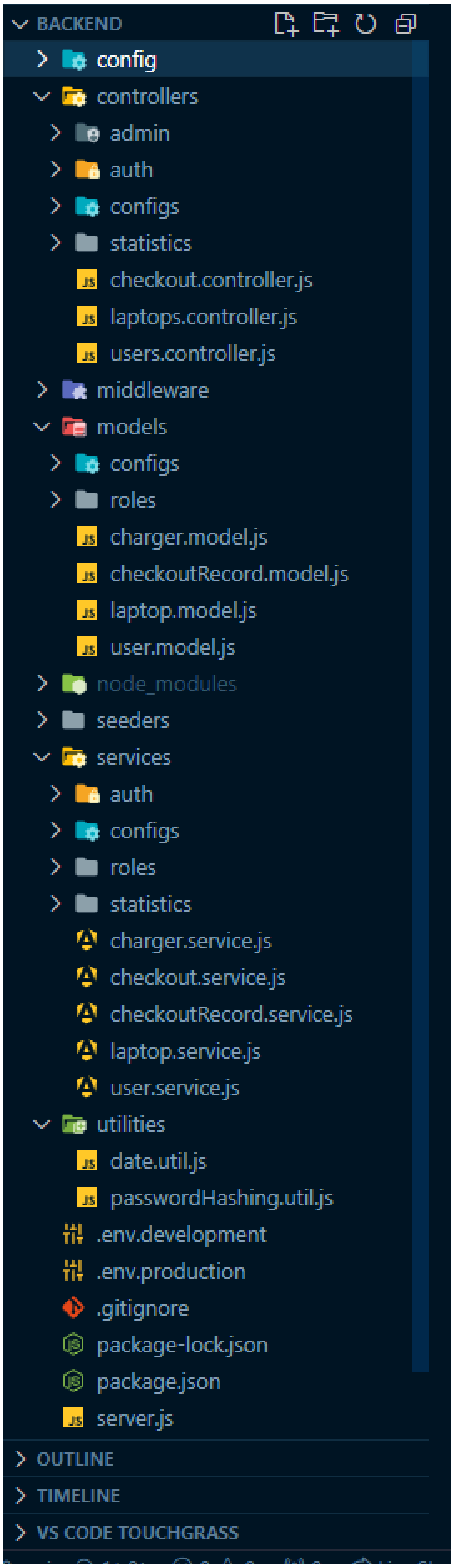
High-Level View

When an API call is made to /users/data, what happens?

1. API comes into app
2. API call is routed to users
3. API call is handled by the appropriate function in users

4. This function possibly uses user.service.js and user.model.js to get some data
5. Data is returned to the computer that called the API

Example File Structure:



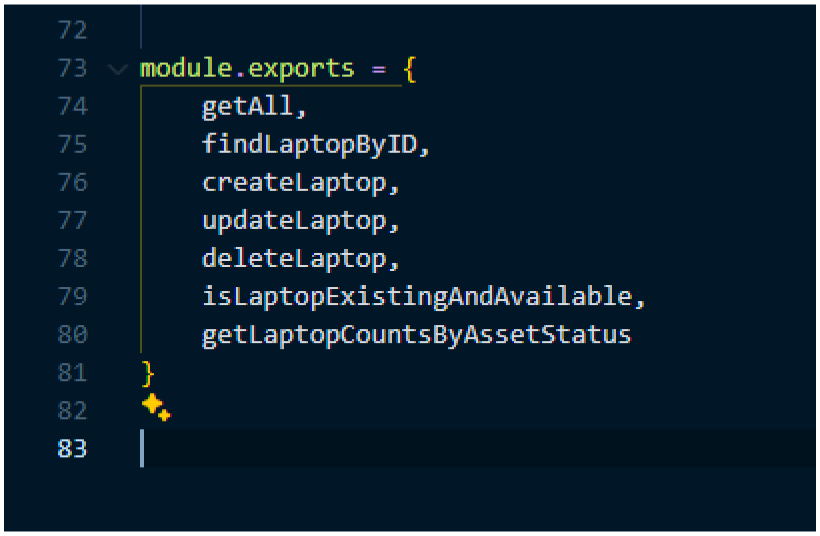
```
BACKEND
  > config
  v controllers
    > admin
    > auth
    > configs
    > statistics
      checkout.controller.js
      laptops.controller.js
      users.controller.js
  > middleware
  v models
    > configs
    > roles
      charger.model.js
      checkoutRecord.model.js
      laptop.model.js
      user.model.js
  > node_modules
  > seeders
  v services
    > auth
    > configs
    > roles
    > statistics
      charger.service.js
      checkout.service.js
      checkoutRecord.service.js
      laptop.service.js
      user.service.js
  v utilities
      date.util.js
      passwordHashing.util.js
    .env.development
    .env.production
    .gitignore
    package-lock.json
    package.json
    server.js
> OUTLINE
> TIMELINE
> VS CODE TOUCHGRASS
```

Appendix

Importing/Exporting

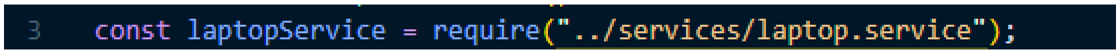
- when importing/exporting, use JSON decomposition:

Export:



```
module.exports = {
    getAll,
    findLaptopByID,
    createLaptop,
    updateLaptop,
    deleteLaptop,
    isLaptopExistingAndAvailable,
    getLaptopCountsByAssetStatus
}

```

Import:



```
const laptopService = require("../services/laptop.service");
```