

## SECTION -3

# Getting to Know Node.js and MongoDB

## Setting up the development environment

Here, we are going to use **Visual Studio Code (VS Code)** as our code editor. Feel free to use whichever editor you prefer, but keep in mind that the extensions used and settings configured may be slightly different in the editor of your choice.

Let's now install VS Code and useful extensions, and then continue setting up all the tools needed for our development environment.

## Installing VS Code and extensions

Before we can get started developing and setting up the other tools, we need to set up our code editor by following these steps:

1. Download VS Code for your operating system from the official website (at the time of writing, the URL is <https://code.visualstudio.com/>).
2. After downloading and installing the application, open it, and you should see the following window:

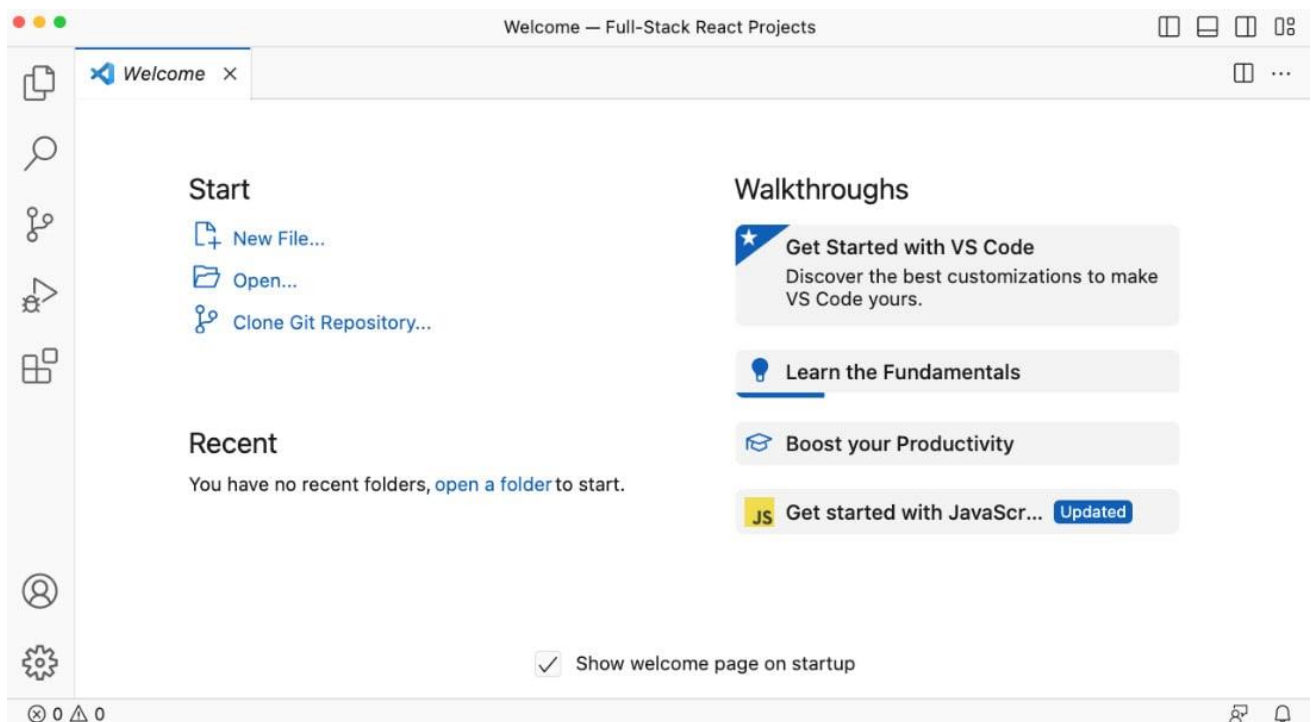


Figure 1.1 – A fresh installation of VS Code (on macOS)

3. To make things easier later, we are going to install some extensions, so click on the **Extensions** icon, which is the fifth icon from the top on the left in the screenshot. A sidebar should open, where you will see **Search Extensions in Marketplace** at the top. Enter an extension name here and click on **Install** to install it. Let's start by installing the **Docker** extension:

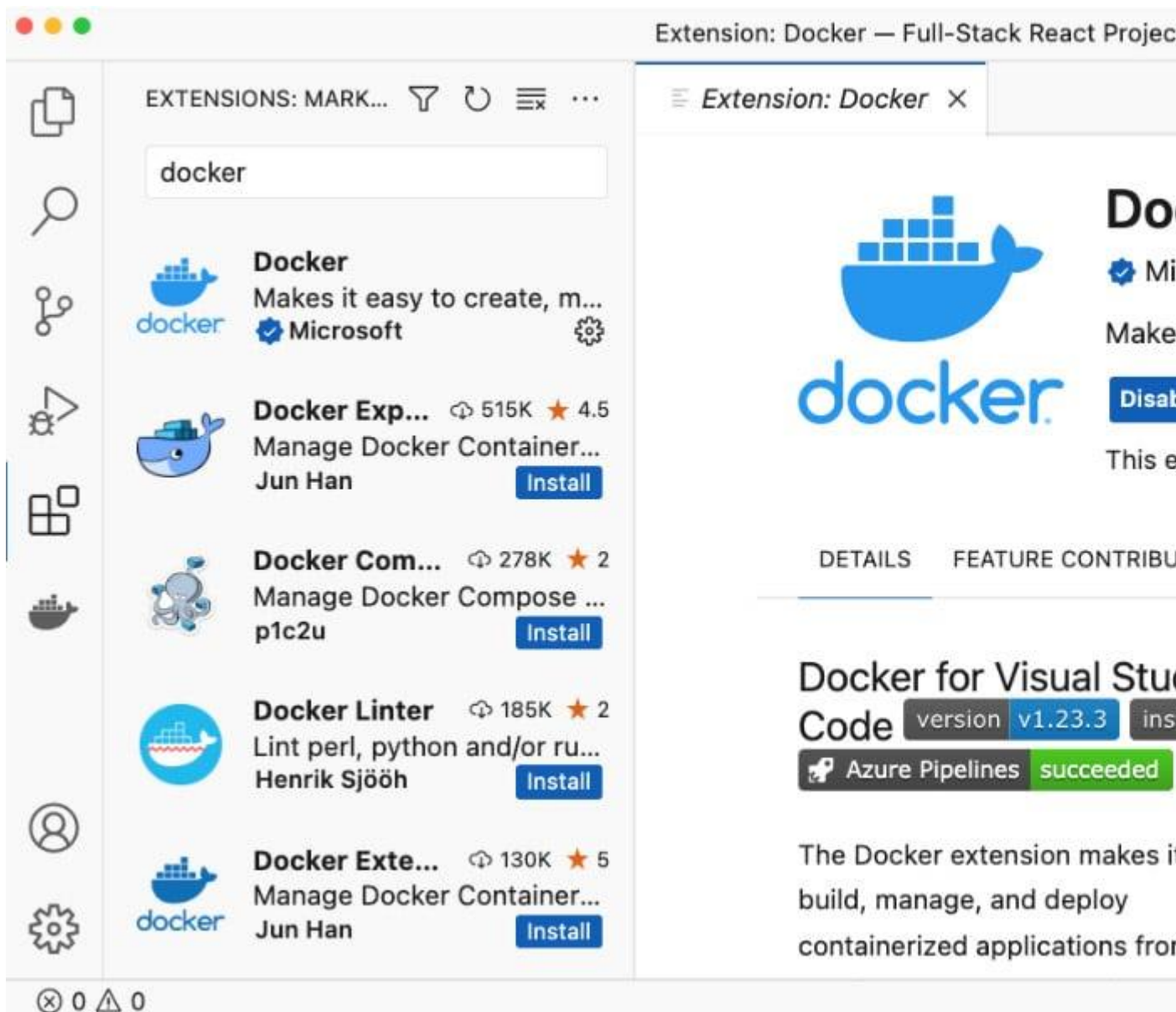


Figure 1.2 – Installing the Docker extension in VS Code

4. Install the following extensions:

- Docker (by Microsoft)
- ESLint (by Microsoft)
- Prettier – Code formatter (by Prettier)
- MongoDB for VS Code (by MongoDB)

Support for JavaScript and Node.js already comes built-in with VS Code.

5. Create a folder for the projects (for example, you can call it **Full-Stack-React-Projects**). Inside this folder, create a new folder called **NodeJsMongoDb**.
6. Go to the **Files** tab (first icon from top) and click the **Open Folder** button to open the empty **NodeJsMongoDb** folder.
7. If you get a dialog asking **Do you trust the authors of the files in this folder?**, check **Trust the authors of all files in the parent folder 'Full-Stack-React-Projects'** and then click on the **Yes, I trust the authors** button.

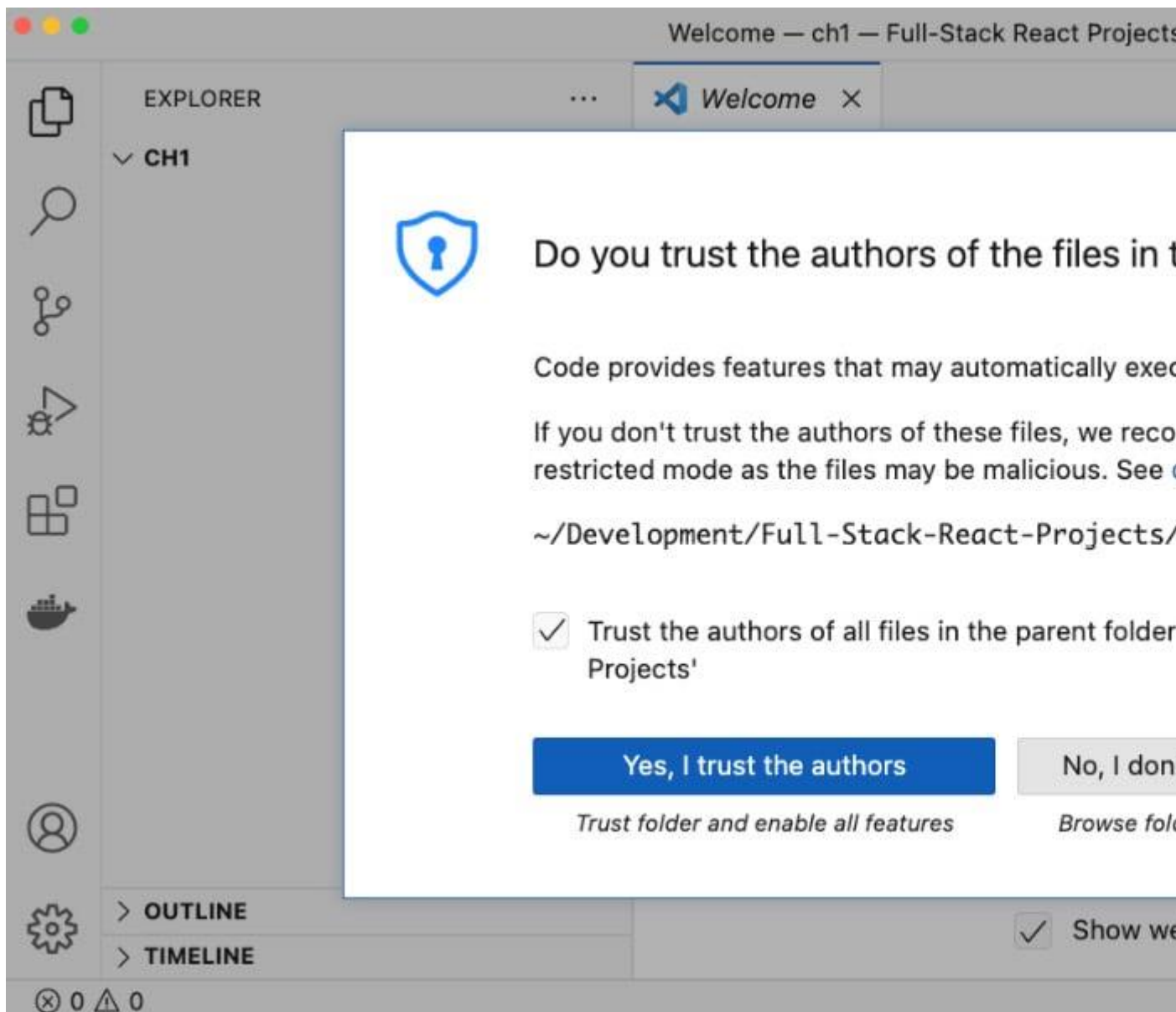


Figure 1.3 – Allowing VS Code to execute files in our project folder

Tip

You can safely ignore this warning in your own projects, as you can be sure that those do not contain malicious code. When opening folders from untrusted sources, you can press **No, I don't trust the authors**, and still browse the code. However, when doing so, some features of VS Code will be disabled.

We have now successfully set up VS Code and are ready to start setting up our project! If you have cloned the folder from the GitHub code examples provided, a notification telling you that a Git repository was found will also pop up. You can simply close this one, as we only want to open the **NodeJsMongoDb** folder.

Now that VS Code is ready, let's continue by setting up a new project with Vite.

## Setting up a project with Vite

For this book, we are going to use **Vite** to set up our project, as it is the most popular and liked according to The State of JS 2022 survey (<https://2022.stateofjs.com/>). Vite also makes it easy to set up a modern frontend project, while still making it possible to extend the configuration later if needed. Follow these steps to set up your project with Vite:

1. In the VS Code menu bar, go to **Terminal | New Terminal** to open a new Terminal.
2. Inside the Terminal, run the following command:

```
$ npm create vite@5.0.0 .
```

Make sure there is a period at the end of the command to create the project in the current folder instead of creating a new folder.

3. When asked if **create-vite** should be installed, simply type **y** and press the Return/Enter key to proceed.
4. When asked about the framework, use the arrow keys to select **React** and press Return. If you are being asked for a project name, press Ctrl + C to cancel, then run the command again, making sure there is a period at the end to select the current folder.
5. When asked about the variant, select **JavaScript**.

6. Now, our project is set up and we can run `npm install` to install the dependencies.
7. Afterward, run `npm run dev` to start the dev server, as shown in the following screenshot:

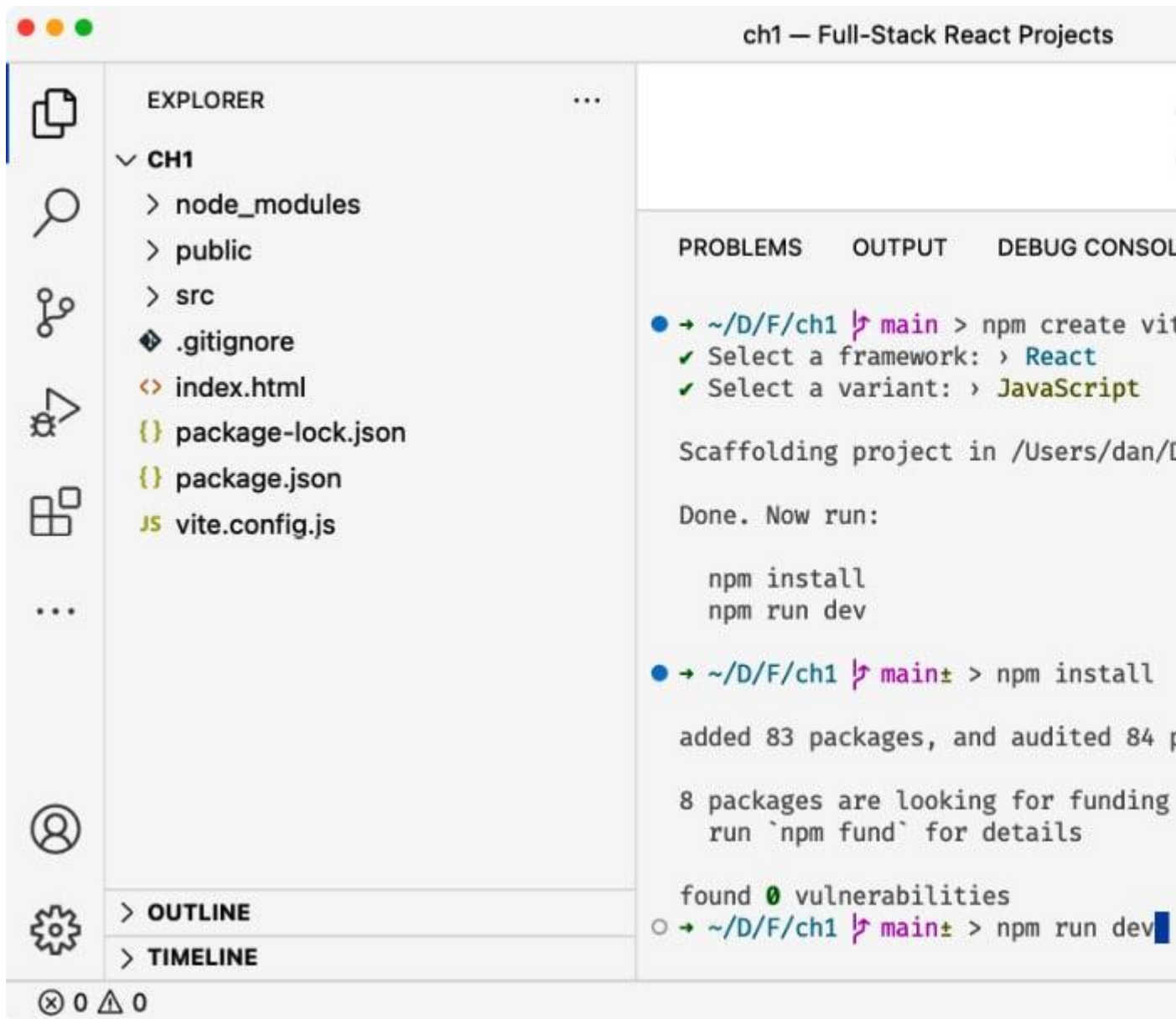


Figure 1.4 – The Terminal after setting up a project with Vite and before starting the dev server

Note

For simplicity in setting up, we just used `npm` directly. If you prefer `yarn` or `pnpm`, you can instead run `yarn create vite` or `pnpm create vite`, respectively.

8. In the Terminal, you will see a URL telling you where your app is running. You can either hold Ctrl (Cmd on macOS) and click on the link to open it in your browser, or manually enter the URL in a browser.
9. To test whether your app is interactive, click the button with the text **count is 0**, and it should increase the count every time it is pressed.

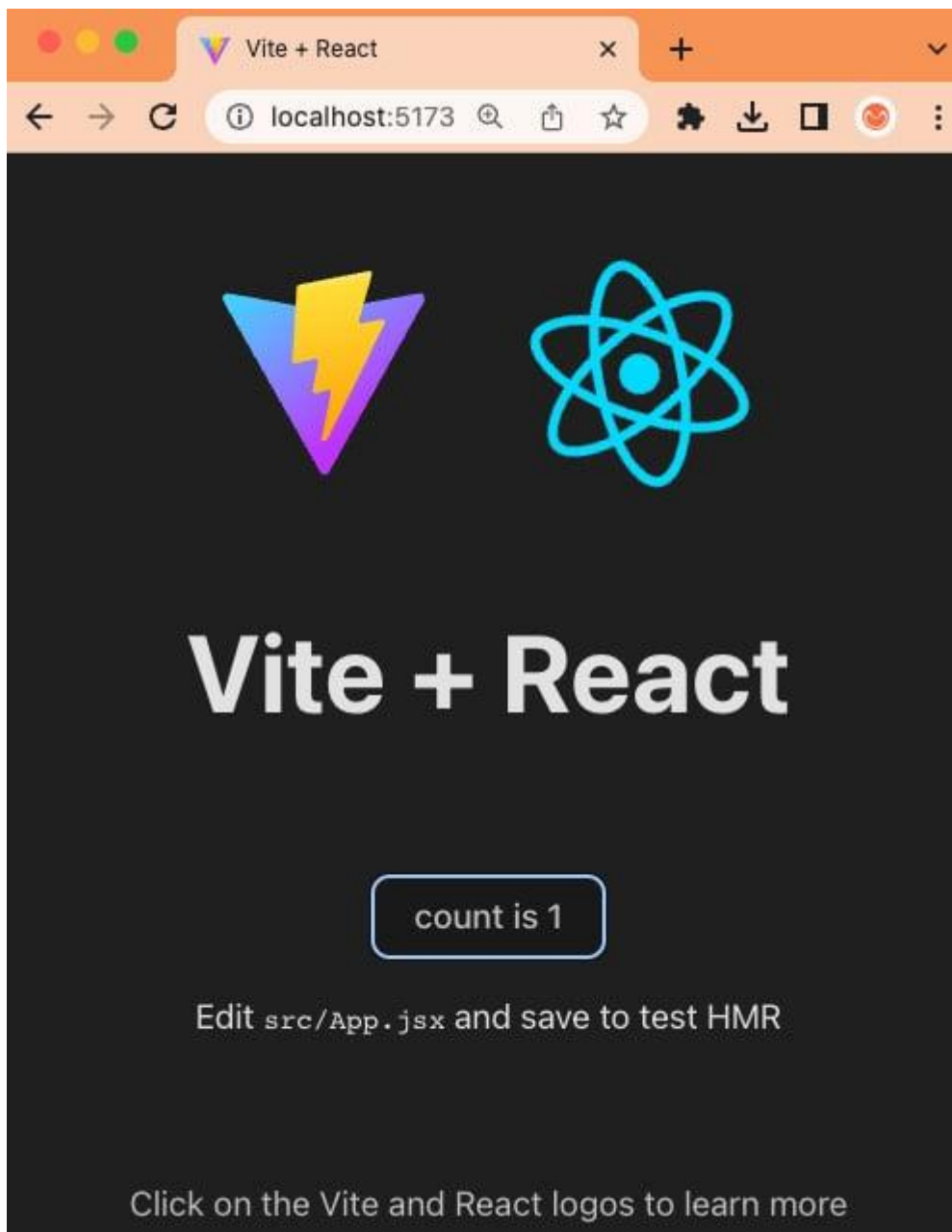


Figure 1.5 – Our first React app running with Vite

## Alternatives to Vite

Alternatives to Vite are bundlers, such as webpack, Rollup, and Parcel. These are highly configurable but often do not offer a great experience for dev servers. They first must bundle all our code together before serving it to the browser. Instead, Vite natively supports the **ECMAScript module (ESM)** standard. Furthermore, Vite requires very little configuration to get started. A downside of Vite is that it can be hard to configure certain more complex scenarios with it. An upcoming bundler that is promising is Turbopack; however, it is still very new at the time of writing. For full-stack development with server-side rendering, we will later get to know Next.js, which is a React framework that also provides a dev server out of the box.

Now that our boilerplate project is up and running, let's spend some time setting up tools that will enforce best practices and a consistent code style.

In this chapter, we will first learn how to write and run scripts with Node.js. Then, we will move on to introducing Docker as a way to set up a database service. Once we have set up Docker and a container for our database, we are going to access it to learn more about MongoDB, the document database that we will use going forward. Finally, we will connect everything we have learned in this chapter by accessing MongoDB via Node.js scripts.

By the end of this chapter, you will have an understanding of the most important tools and concepts in backend development with JavaScript. This chapter gives us a good foundation to create a backend service for our first full-stack application in the upcoming chapters.

In this chapter, we are going to cover the following main topics:

- Writing and running scripts with Node.js
- Introducing Docker, a platform for containers
- Introducing MongoDB, a document database
- Accessing the MongoDB database via Node.js

# Writing and running scripts with Node.js

For us to become full-stack developers, it is important to get familiar with backend technologies. As we are already familiar with JavaScript from writing frontend applications, we can use Node.js to develop backend services using JavaScript. In this section, we are going to create our first simple Node.js script to get familiar with the differences between backend scripts and frontend code.

## Similarities and differences between JavaScript in the browser and in Node.js

Node.js is built on V8, the JavaScript engine used by Chromium-based browsers (Google Chrome, Brave, Opera, Vivaldi, and Microsoft Edge). As such, JavaScript code will run the same way in the browser and Node.js. However, there are some differences, specifically in the environment. The environment is built on top of the engine and allows us to render something on a website in the browser (using the `document` and `window` objects). In Node.js, there are certain modules provided to interface with the operating system, for tasks such as creating files and handling network requests. These modules allow us to create a backend service using Node.js.

Let's have a look at the Node.js architecture versus JavaScript in the browser:

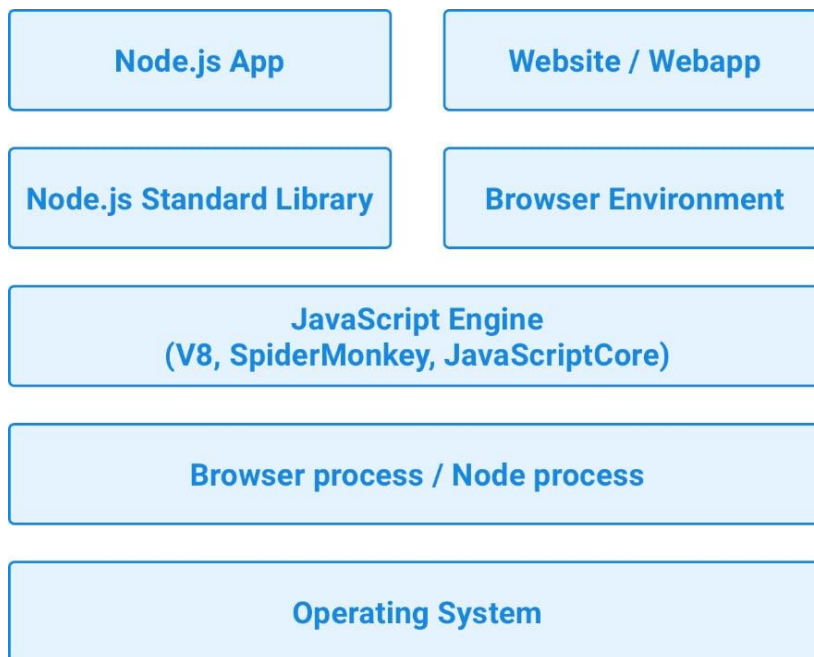


Figure 2.1 – The Node.js architecture versus JavaScript in the browser

As we can see from the visualization, both Node.js and browser JavaScript run on a JavaScript engine, which is always V8 in Node.js, and can be V8 for Chromium-based browsers, SpiderMonkey for Firefox, or JavaScriptCore for Safari.

Now that we know that we can run JavaScript code in Node.js, let's try it out!

## Creating our first Node.js script

Before we can start writing backend services, we need to get familiar with the Node.js environment. So, let's start by writing a simple "hello world" example:

1. Open the new **NodejsMongodb** folder in VS Code
2. Create a new **backend** folder in the **NodejsMongodb** folder. This will contain our backend code
3. In the **backend** folder, create a **helloworld.js** file and enter the following code:

```
console.log('hello node.js world!')
```

Open a Terminal in the `NodejsMongodb` . Then type `cd backend` and run the following command to execute the Node.js script:

### `node helloworld.js`

You will see that the console output shows `hello node.js world!` . When writing Node.js code, we can make use of familiar functions from the frontend JavaScript world and run the same JavaScript code on the backend!

### Note

While most frontend JavaScript code will run just fine in Node.js, not all code from the frontend will automatically work in a Node.js environment. There are certain objects, such as `document` and `window`, that are specific to a browser environment. This is important to keep in mind, especially when we introduce server-side rendering later.

Now that we have a basic understanding of how Node.js works, let's get started handling files with Node.js.

## Handling files in Node.js

Unlike in the browser environment, Node.js provides functions to handle files on our computer via the `node:fs` (filesystem) module. For example, we could make use of this functionality to read and write various files or even use files as a simple database.

Follow these steps to create your first Node.js script that handles files:

1. Create a new `backend/files.js` file.
2. Import the `writeFileSync` and `readFileSync` functions from the `node:fs` internal Node.js module. This module does not need to be installed via npm, as it is provided by the Node.js runtime.

```
import { writeFileSync, readFileSync } from 'node:fs'
```

3. Create a simple array containing users, with a name and email address:

```
const users = [{ name: 'Adam Ondra', email: 'adam.ondra@climb.ing' }]
```

- Before we can save this array to a file, we first need to convert it to a string by using `JSON.stringify`:

```
const usersJson = JSON.stringify(users)
```

- Now we can save our JSON string to a file by using the `writeFileSync` function. This function takes two arguments – first the filename, then the string to be written to the file:

```
writeFileSync('backend/users.json', usersJson)
```

- After writing to the file, we can attempt reading it again using `readFileSync` and parsing the JSON string using `JSON.parse`:

```
7. const readUsersJson = readFileSync('backend/users.json')  
   const readUsers = JSON.parse(readUsersJson)
```

- Finally, we log the parsed array:

```
console.log(readUsers)
```

- Now we can run our script. You will see that the array gets logged and a `users.json` file was created in our `backend/` folder:

```
$ node backend/files.js
```

You may have noticed that we have been using `writeFileSync`, and not `writeFile`. The default behavior in Node.js is to run everything asynchronously, which means that if we used `writeFile`, the file may not have been created yet at the time when we called `readFile`, as asynchronous code is not executed in order.

This behavior might be annoying when writing simple scripts like we did, but is very useful when dealing with, for example, network requests, where we do not want to block other users from accessing our service while dealing with another request.

After learning about handling files with Node.js, let's learn more about how asynchronous code is executed in the browser and Node.js.

## Concurrency with JavaScript in the browser and Node.js

An essential and special trait of JavaScript is that most API functions are asynchronous by default. This means that code does not simply run in the sequence in which it is defined. Specifically, JavaScript is event-driven. In the browser, this means that JavaScript code will run because of user interactions. For example, when a button is clicked, we define an `onClick` handler to execute some code.

On the server side, input/output operations, such as reading and writing files, and network requests, are handled asynchronously. This means that we can handle multiple network requests at once, without having to deal with threads or multiprocessing ourselves. Specifically, in Node.js, `libuv` is responsible for assigning threads for I/O operations while giving us, as a programmer, access to a single runtime thread to write our code in. However, this does not mean that each connection to our backend will create a new thread. Threads are created on the fly when advantageous. As a developer, we do not have to deal with multi-threading and can focus on developing with asynchronous code and callbacks.

If code is synchronous, it is executed directly by putting it on the **call stack**. If code is asynchronous, the operation is started, and the instance of that operation is stored in a queue, together with a callback function. The Node.js runtime will first execute all code left in the stack. Then, the **event loop** will come in and check whether there are any completed tasks in the queue. If that is the case, the callback function is executed by putting it on the stack. A callback function can then again either execute synchronous or asynchronous code. When we add an event listener – for example, an `onClick` listener in the browser – when the user clicks the related element, the callback will also be put in the task queue, which means it will be executed when nothing else is left on the stack. Similarly, in Node.js, we can add listeners for network events, and execute a callback when a request comes in.

In contrast to multi-threaded servers, a Node.js server accepts all requests in a single thread, which contains the event loop. Multi-threaded servers have the disadvantage that threads can block I/O completely and slow down the server. Node.js, however, delegates operations in a fine-grained way on the fly to threads. This results in less blocking of I/O operations by default. The downside with Node.js is that we have less control over how the multi-threading happens and thus need to be careful to avoid using synchronous functions whenever possible. Otherwise, we will block the main Node.js thread and slow down our server. For simplicity, we still use synchronous functions in this chapter. Going forward, in the next chapters, we

will avoid using those and rely solely on asynchronous functions (when possible) to get the best performance.

The following diagram visualizes the difference between multi-threaded servers and a Node.js server:

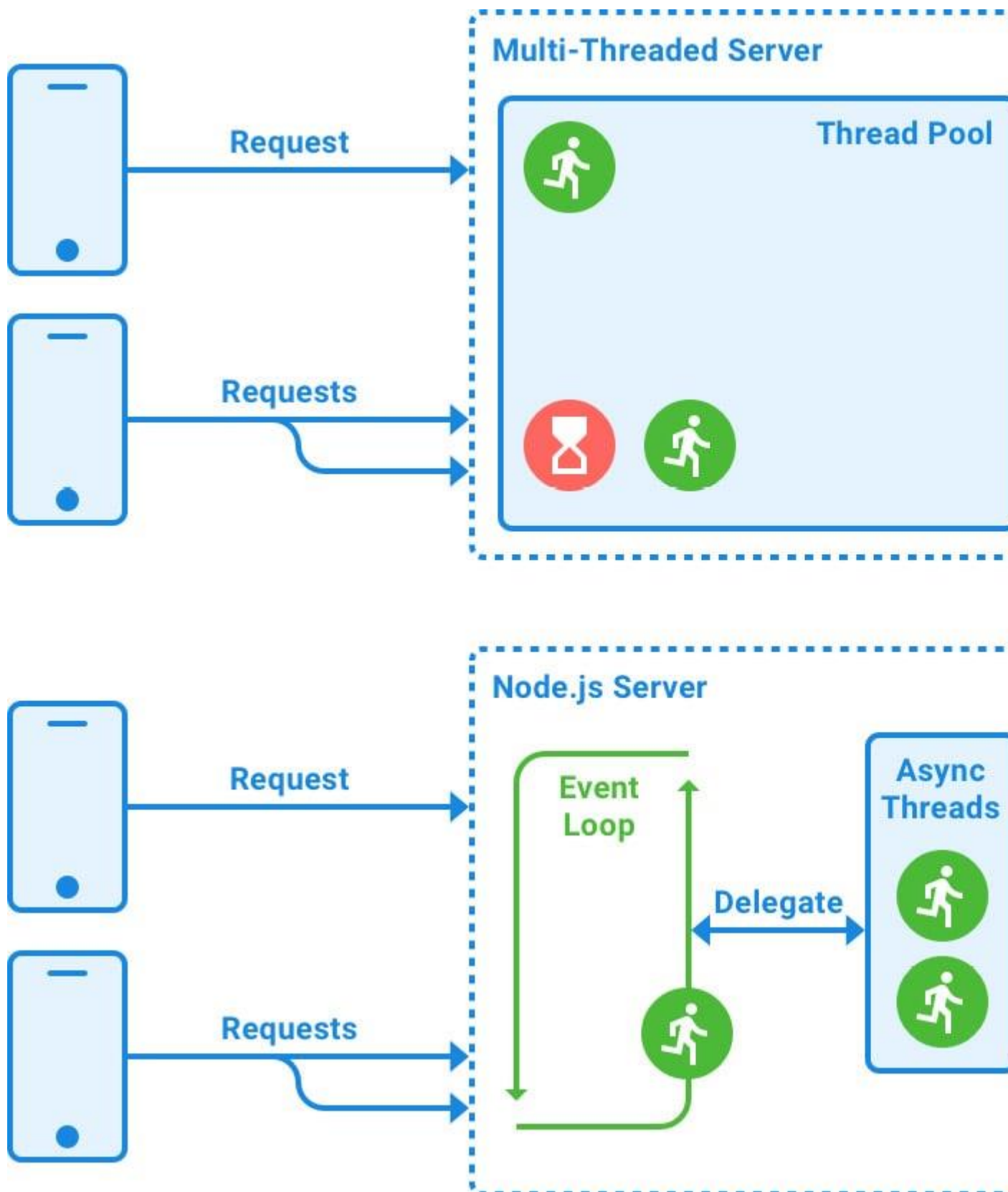


Figure 2.2 – The difference between multi-threaded servers and a Node.js server

We can see this asynchrony in action by using `setTimeout`, a function that you may be familiar with from frontend code. It waits a specified number of milliseconds and then executes the code specified in the callback function. For example, if we run the following code (with a Node.js script or in the browser, the result is the same for both):

```
console.log('first')
setTimeout(() => {
  console.log('second')
}, 1000)
console.log('third')
```

We can see that they get printed in the following order:

```
first
third
second
```

This makes sense, because we are delaying the “second” `console.log` by a second. However, the same output will happen if we execute the following code:

```
console.log('first')
setTimeout(() => {
  console.log('second')
}, 0)
console.log('third')
```

Now that we are waiting zero milliseconds before executing the code, you would think that “second” gets printed after “first.” However, that is not the case. Instead, we get the same output as before:

```
first
third
second
```

The reason is that when we use `setTimeout`, the JavaScript engine calls either a web API (on the browser) or a native API (on Node.js). This API runs in native code in the engine, tracks the timeout internally, and puts the callback into the task queue,

because the timer completes right away. While this is happening, the JavaScript engine continues processing the other code by pushing it onto the stack and executing it. When the stack is empty (there is no more code to execute), the event loop advances. It sees that there is something in the task queue, so it executes that code, resulting in “second” being printed last.

### Tip

You can use the Loupe tool to visualize the inner workings of the Call Stack, web APIs, Event Loop, and Callback/Task Queue: <http://latentflip.com/loupe/>

Now that we have learned how asynchronous code is handled in the browser and Node.js, let's create our first web server with Node.js!

## Creating our first web server

Now that we have learned the basics of how Node.js works, we can use the `node:http` library to create a simple web server. For our first simple server, we are just going to return a **200 OK** response and some plain text on any request. Let's get started with the steps:

1. Create a new `backend/simpleweb.js` file, open it, and import the `createServer` function from the `node:http` module:

```
import { createServer } from 'node:http'
```

2. The `createServer` function is asynchronous, so it requires us to pass a callback function to it. This function will be executed when a request comes in from the server. It has two arguments, a request object (`req`) and a response object (`res`). Use the `createServer` function to define a new server:

```
const server = createServer((req, res) => {
```

3. For now, we will ignore the request object and only return a static response. First, we set the status code to `200`:

```
res.statusCode = 200
```

4. Then, we set the `Content-Type` header to `text/plain`, such that the browser knows what kind of response data it is dealing with:

```
res.setHeader('Content-Type', 'text/plain')
```

5. Lastly, we end the request by returning a **Hello HTTP world!** string in the response:

```
6. res.end('Hello HTTP world!')
   })
```

7. After defining the server, we need to make sure to listen on a certain host and port. These will define where the server will be available. For now, we use localhost on port **3000** to make sure our server is available via **http://localhost:3000/**:

```
8. const host = 'localhost'
   const port = 3000
```

9. The **server.listen** function is also asynchronous and requires us to pass a callback function, which will execute as soon as the server is up and running. We can simply log something here for now:

```
10. server.listen(port, host, () => {
11.   console.log(`Server listening on http://${host}:${port}`)
   })
```

12. Run the Node.js script as follows:

```
$ node backend/simpleweb.js
```

13. You will notice that we get our **Server listening on http://localhost:3000** log message, so we know the server was started successfully. This time, the Terminal does not return control to us; the script keeps running. We can now open **http://localhost:3000** in a browser:

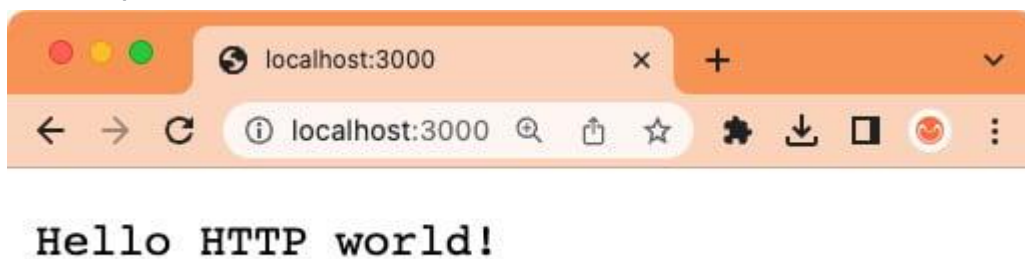


Figure 2.3 – A plaintext response from our first web server!

Now that we have set up a simple web server, we can extend it to serve a JSON file instead of simply returning plaintext.

## Extending the web server to serve our JSON file

We can now try combining our knowledge of the `node:fs` module with the HTTP server to create a server that serves the previously created `users.json` file. Let's get started with the steps:

1. Copy the `backend/simpleweb.js` file to a new `backend/webfiles.js` file.
2. At the beginning of the file, add an import of `readFileSync`:

```
import { readFileSync } from 'node:fs'
```

3. Change the `Content-Type` header to `application/json`:

```
res.setHeader('Content-Type', 'application/json')
```

4. Replace the string in `res.end()` with the JSON string from our file. In this case, we do not need to parse the JSON, as `res.end()` expects a string anyway:

```
res.end(readFileSync('backend/users.json'))
```

5. If it is still running, stop the previous server script via `Ctrl + C`. We need to do this because we cannot listen on the same port twice.
6. Run the server and refresh the page to see the JSON from the file being printed. Try changing the `users.json` file and see how it is read again on the next request (when refreshing the website):

```
$ node backend/webfiles.js
```

While useful as an exercise, files are not a proper database to be used in production. As such, we are later going to introduce MongoDB as a database. We are going to run the MongoDB server in Docker, so let's first briefly have a look at Docker.

# Introducing Docker, a platform for containers

Docker is a platform that allows us to package, manage, and run applications in loosely isolated environments, called **containers**. Containers are lightweight, are isolated from each other, and include all dependencies needed to run an application. As such, we can use containers to easily set up various services and apps without having to deal with managing dependencies or conflicts between them.

## Note

There are also other tools, such as Podman (which even has a compatibility layer for the Docker CLI commands), and Rancher Desktop, which also supports Docker CLI commands.

We can use Docker locally to set up and run services in an isolated environment. Doing so avoids polluting our host environment and ensures that there is a consistent state to build upon. This consistency is especially important when working in larger development teams, as it ensures that everyone is working with the same state.

Additionally, Docker makes it easy to deploy containers to various cloud services and run them in a **continuous integration/continuous delivery (CI/CD)** workflow.

In this section, we will first get an overview of the Docker platform. Then, we will learn how to create a container and how to access Docker from VS Code. At the end, we will understand how Docker works and how it can be used to manage services.

## The Docker platform

The Docker platform essentially consists of three parts:

- **Docker Client:** Can run commands by sending them to the **Docker daemon**, which is either running on the local machine or a remote environment.

- **Docker Host:** Contains the Docker daemon, images, and containers.
- **Docker Registry:** Hosts and stores docker images, extensions, and plugins.  
By default, the public registry **Docker Hub** will be used to search for images.

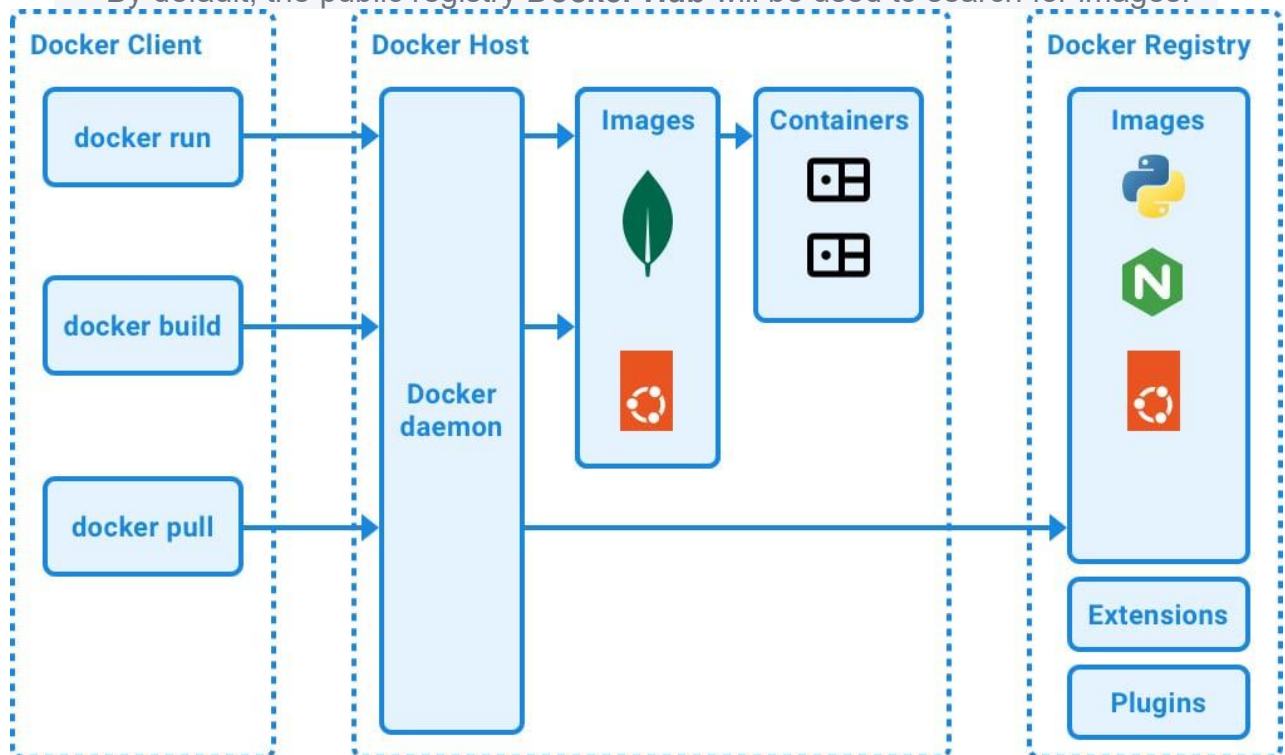


Figure 2.4 – Overview of the Docker platform

**Docker images** can be thought of as read-only templates and are used to create containers. Images can be based on other images. For example, the **mongo** image, which contains a MongoDB server, is based on the **ubuntu** image.

**Docker containers** are instances of images. They run an operating system with a configured service (such as a MongoDB server on Ubuntu). Additionally, they can be configured, for example, to forward some ports from within the container to the host, or to mount a storage volume in the container that stores data on the host machine. By default, a container is isolated from the host machine, so if we want to access ports or storage from it on the host, we need to tell Docker to allow this.

## Installing Docker

The easiest way to set up the Docker platform for local development is using Docker Desktop. It can be downloaded from the official Docker website (<https://www.docker.com/products/docker-desktop/>). Follow the instructions to install it and start the Docker engine. After installation, you should have a `docker` command available in your Terminal. Run the following command to verify that it is working properly:

```
$ docker -v
```

This command should output the Docker version, like in the following example:

```
Docker version 24.0.6, build 980b856
```

After installing and starting Docker, we can move on to creating a container.

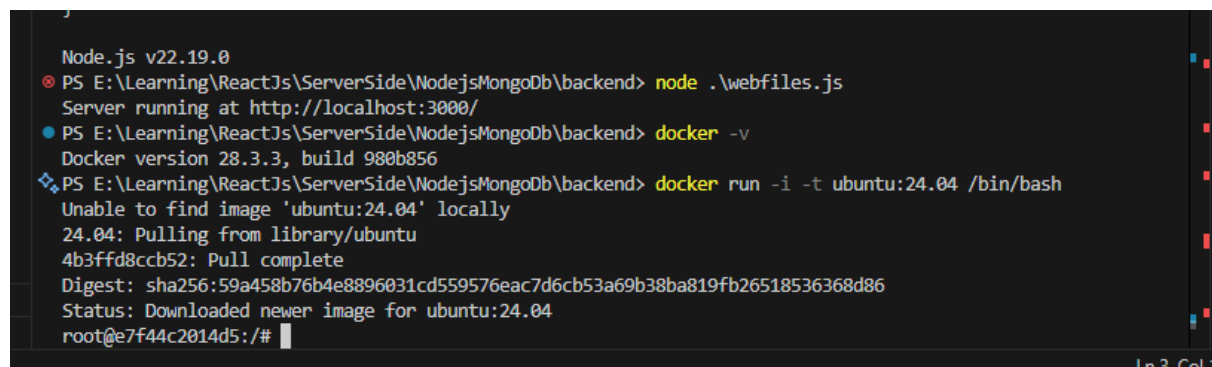
## Creating a container

Docker Client can instantiate a container from an image via the `docker run` command. Let's now create an `ubuntu` container and run a shell (`/bin/bash`) in it:

```
$ docker run -i -t ubuntu:24.04 /bin/bash
```

Note

The `:24.04` string after the image name is called the **tag**, and it can be used to pin images to certain versions. We use tags to pull specific versions of images so that the steps are reproducible even when new versions are released. By default, if no tag is specified, Docker will attempt to use the `latest` tag.

A terminal window showing a series of commands and their outputs. The first command is `node .\webfiles.js`, which outputs "Server running at http://localhost:3000/". The second command is `docker -v`, which outputs "Docker version 28.3.3, build 980b856". The third command is `docker run -i -t ubuntu:24.04 /bin/bash`, which outputs "Unable to find image 'ubuntu:24.04' locally", "24.04: Pulling from library/ubuntu", "4b3ffd8ccb52: Pull complete", "Digest: sha256:59a458b76b4e8896031cd559576eac7d6cb53a69b38ba819fb26518536368d86", "Status: Downloaded newer image for ubuntu:24.04", and "root@e7f44c2014d5:/#".

```
Node.js v22.19.0
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> node .\webfiles.js
Server running at http://localhost:3000/
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> docker -v
Docker version 28.3.3, build 980b856
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> docker run -i -t ubuntu:24.04 /bin/bash
Unable to find image 'ubuntu:24.04' locally
24.04: Pulling from library/ubuntu
4b3ffd8ccb52: Pull complete
Digest: sha256:59a458b76b4e8896031cd559576eac7d6cb53a69b38ba819fb26518536368d86
Status: Downloaded newer image for ubuntu:24.04
root@e7f44c2014d5:/#
```

A new shell will open. We can verify that this shell is running in the container by executing the following command to see which operating system is running:

```
$ uname -a
root@e7f44c2014d5:/# uname -a
Linux e7f44c2014d5 6.6.87.2-microsoft-standard-WSL2 #1 SMP PREEMPT_DYNAMIC Thu Jun  5 18:30:46 UTC 2025 x86_
64 x86_64 x86_64 GNU/Linux
root@e7f44c2014d5:/#
```

You can now type the following command to exit the shell and the container:

```
$ exit
```

The `docker run` command does the following:

- If you have never run a container based on the `ubuntu` image before, Docker will start by pulling the image from the Docker registry (this is equivalent to executing `docker pull ubuntu`).
- After the image is downloaded, Docker creates a new container (the equivalent to executing `docker container create`).
- Then, Docker configures a read-write filesystem for the container and creates a default network interface.
- Finally, Docker starts the container and executes the specified command. In our case, we specified the `/bin/bash` command. Because we passed the `-i` (keeps `STDIN` open) and `-t` (allocates a pseudo-tty) options, Docker attaches the container's shell to our currently running Terminal, allowing us to use the container as if we were directly accessing a Terminal on our host machine.

As we can see, Docker is very useful for creating self-contained environments for our apps and services to run in. Later, we are going to learn how to package our own apps in Docker containers. For now, we are only going to use Docker to run services without having to install them on our host system.

## Accessing Docker via VS Code

We can also access Docker via the VS Code extension. To do so, click the Docker icon in the left sidebar of VS Code. The Docker sidebar will open, showing you a list

of containers, images, registries, networks, volumes, contexts, and relevant resources:

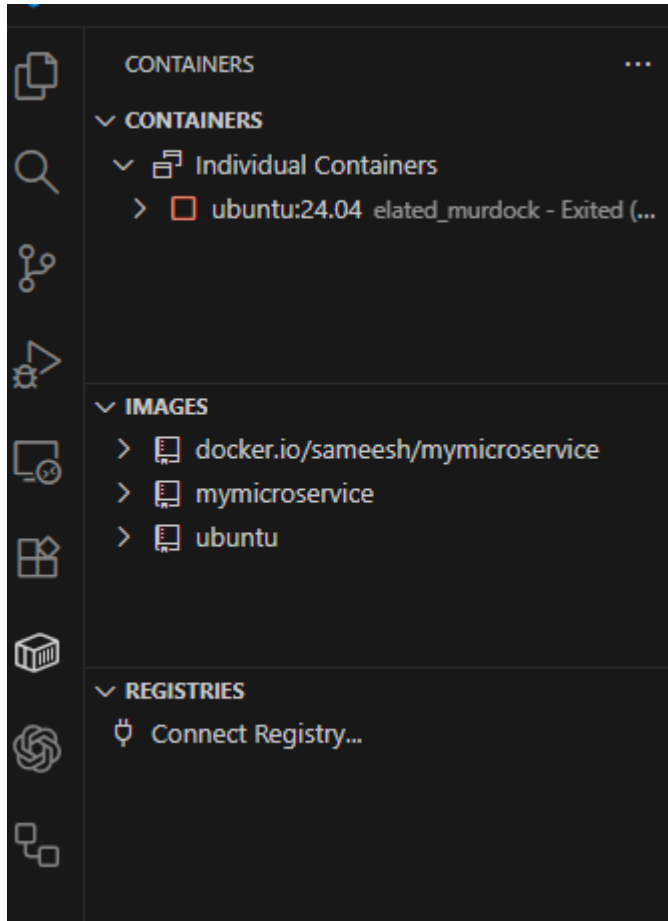


Figure 2.6 – The Docker sidebar in VS Code

Here, you can see which containers are stopped and which ones are running. You can right-click on a container to start, stop, restart, or remove it. You can also view its logs to debug what is going on inside the container. Additionally, you can attach a shell to the container to get access to its operating system.

Now that we know the essentials of Docker, we can create a container for our MongoDB database server

# Introducing MongoDB, a document database

MongoDB, is the most popular NoSQL database. Unlike **Structured Query Language (SQL)** databases (such as MySQL or PostgreSQL), NoSQL means that the database specifically does not use SQL to query the database. Instead, NoSQL databases have various other ways to query the database and often have a vastly different structure of how data is stored and queried.

The following main types of NoSQL databases exist:

- Key-value stores (for example, Valkey/Redis)
- Column-oriented databases (for example, Amazon Redshift)
- Graph-based databases (for example, Neo4j)
- Document-based databases (for example, MongoDB)

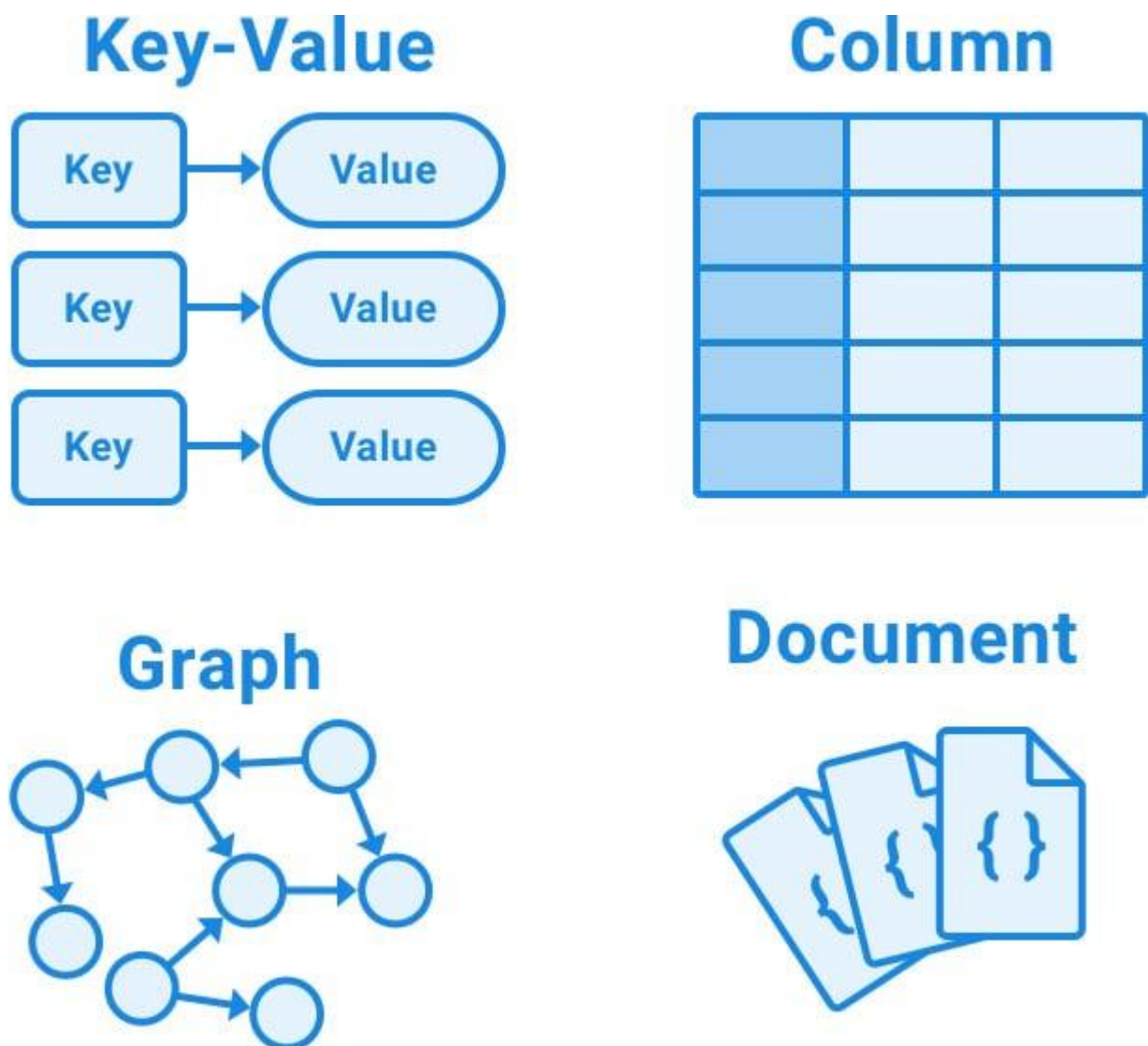


Figure 2.7 – Overview of NoSQL databases

MongoDB is a document-based database, which means that each entry in the database is stored as a document. In MongoDB, these documents are basically JSON objects (internally, they are stored as BSON – a binary JSON format to save space and improve performance, among other advantages). Instead, SQL databases store data as rows in tables. As such, MongoDB provides a lot more flexibility. Fields can be freely added or left out in documents. The downside of such a structure is that we do not have a consistent schema for documents. However, this can be solved by using libraries, such as Mongoose, which we will learn later.

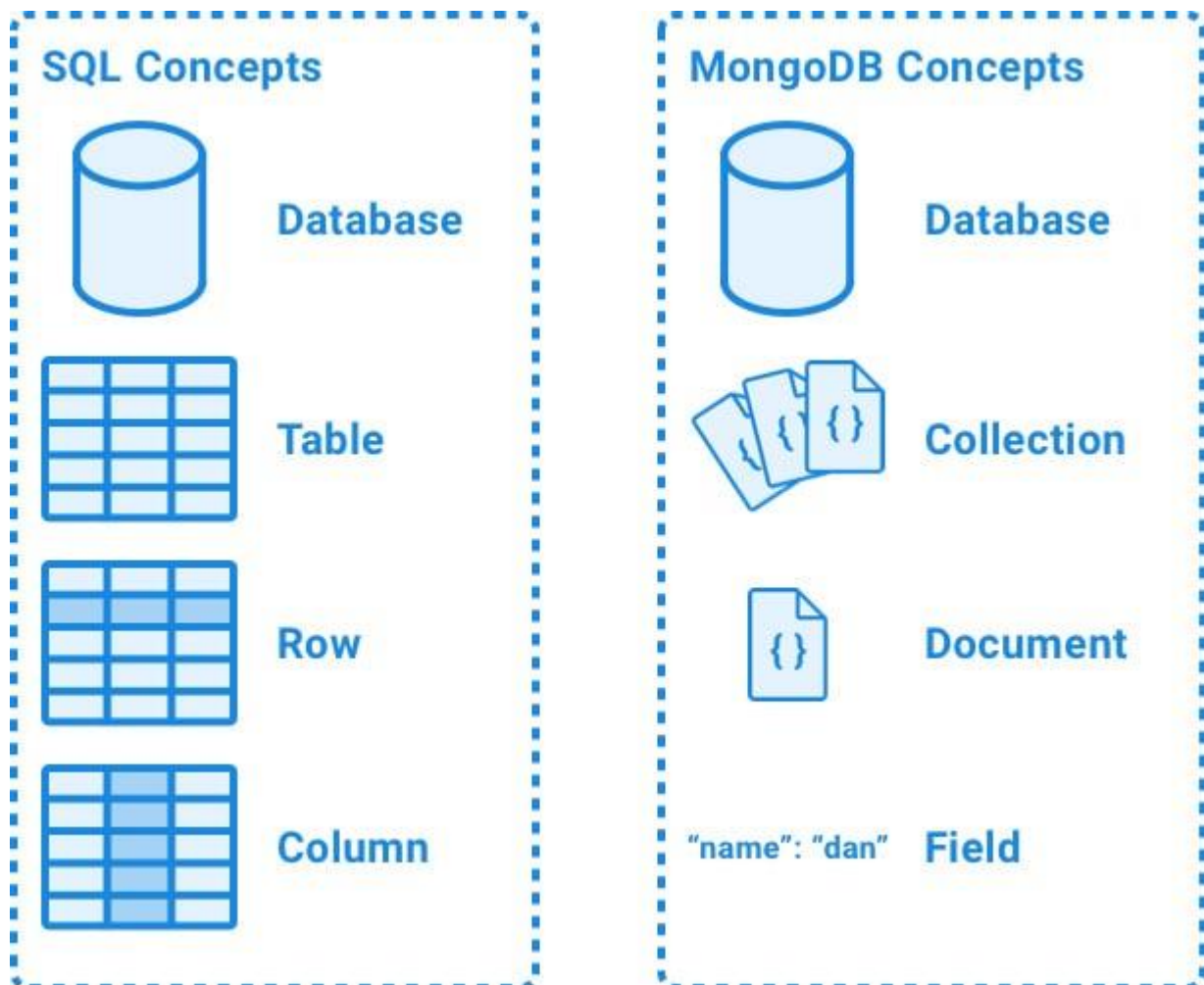


Figure 2.8 – Comparison between MongoDB and SQL databases

MongoDB is also based on a JavaScript engine. Since version 3.2, it has been using SpiderMonkey (the JavaScript engine that Firefox uses) instead of V8. Nevertheless, this still means we can execute JavaScript code in MongoDB. For example, we can use JavaScript in the **MongoDB Shell** to help with administrative tasks. Again, we must be careful with this, though, as the MongoDB environment is vastly different from a browser or Node.js environment.

In this section, we will first learn how to set up a MongoDB server using Docker. Then, we will learn more about MongoDB and how to access it directly using the MongoDB Shell for the administration of our database and the data. We are also going to learn how to use VS Code to access MongoDB. At the end of this section, you will have an understanding of how CRUD operations work in MongoDB.

Note

CRUD is an acronym for create, read, update, and delete, which are the common operations that backend services usually provide.

## Setting up a MongoDB server

Before we can start using MongoDB, we need to set up a server. Since we already have Docker installed, we can make things easier for ourselves by running MongoDB in a Docker container. Doing so also allows us to have separate, clean MongoDB instances for our apps by creating separate containers. Let's get started with the steps:

1. Make sure Docker Desktop is running and Docker is started. You can verify this by running the following command, which lists all running containers:

```
$ docker ps
```

If Docker is not started properly, you will get a **Cannot connect to the Docker daemon** error. In that case, make sure Docker Desktop is running and the Docker Engine is not paused.

If Docker is started properly, you will see the following output:

CONTAINER
ID IMAGE COMMAND CREATED STATUS PORTS NAMES

If you already have some containers running, it will be followed by a list of started containers.

2. Run the following Docker command to create a new container with a MongoDB server:

```
$ docker run -d --name dbserver -p 27017:27017 --restart unless-stopped mongo:6.0.4
```

The `docker run` command creates and runs a new container. The arguments are as follows:

- `-d`: Runs the container in the background (daemon mode).

- `--name`: Specifies a name for the container. In our case, we named it `dbserver`.
  - `-p`: Maps a port from the container to the host. In our case, we map the default MongoDB server port `27017` in the container to the same port on our host. This allows us to access the MongoDB server running within our container from outside of it. If you already have a MongoDB server running on that port, feel free to change the first number to some other port, but make sure to also adjust the port number from `27017` to your specified port in the following guides.
  - `--restart unless-stopped`: Makes sure to automatically start (and restart) the container unless we manually stop it. This ensures that every time we start Docker, our MongoDB server will already be running.
  - `mongo`: This is the image name. The `mongo` image contains a MongoDB server.
3. Install the MongoDB Shell on your host system (not within the container) by following the instructions on the MongoDB website (<https://www.mongodb.com/docs/mongodb-shell/install/>).
  4. On your host system, run the following command to connect to the MongoDB server using the MongoDB Shell (`mongosh`). After the hostname and port, we specify a database name. We are going to call our database `mongotest`:

```
$ mongosh mongodb://localhost:27017/ mongotest
```

You will see some output from the database server, and at the end, we get a shell running on our selected database, as can be seen by the `mongotest >` prompt. Here, we can enter commands to be executed on our database. Interestingly, MongoDB, like Node.js, also exposes a JavaScript engine, but with yet another different environment. So, we can run JavaScript code, such as the following:

```
mongotest> console.log("test")
```

The following figure shows JavaScript code being executed in the MongoDB Shell:

```
test> exit

C:\Users\sameesh>mongosh mongodb://localhost:27017/ mongotest
Error: ENOENT: no such file or directory, open 'C:\Users\sameesh\mongotest'

C:\Users\sameesh>mongosh mongodb://localhost:27017/mongotest
Current Mongosh Log ID: 68e9ca902e97d4adcdebea3
Connecting to:      mongodb://localhost:27017/mongotest?directConnection=true&serverSelectionTimeoutMS=2000&appName=
mongosh+2.5.8
Using MongoDB:      8.0.1
Using Mongosh:      2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

-----
  The server generated these startup warnings when booting
  2025-09-12T10:09:32.943+05:30: Access control is not enabled for the database. Read and write access to data and conf
  igation is unrestricted
  -----

mongotest> console.log("test")
test
mongotest>
```

Figure 2.9 – Connecting to our MongoDB database server running in a Docker container

Now that we have a shell connected to our MongoDB database server, we can start practicing running commands directly on the database.

## Running commands directly on the database

Before we get started creating a backend service that interfaces with MongoDB, let's spend some time getting familiar with MongoDB itself via the MongoDB Shell. The MongoDB Shell is very important for debugging and doing maintenance tasks on the database, so it is a good idea to get to know it well.

### Creating a collection and inserting and listing documents

**Collections** in MongoDB are the equivalent of tables in relational databases. They store documents, which are like JSON objects. To make it easier to understand, a collection can be seen as a very large JSON array containing JSON objects. Unlike simple arrays, collections support the creation of indices, which speed up the lookup of certain fields in documents. In MongoDB, a collection is automatically created when we attempt to insert a document into it or create an index for it.

Let's use the MongoDB Shell to insert a document into our database in the `users` collection:

1. To insert a new user document into the `users` collection, run the following command in the MongoDB Shell:

```
> db.users.insertOne({ username: 'dan', fullName: 'Daniel Bugl', age: 26 })
```

Commands that access the database are prefixed with `db`, then the collection name follows, and finally comes the operation, all separated by periods.

#### Note

While `insertOne()` allows us to insert a single document into the collection, there is also an `insertMany()` method, where we can pass an array of documents to add to the collection.

2. We can now list all documents from the `users` collection by running the following command:

```
> db.users.find()
```

Doing so will return an array with our previously inserted document:

```
[
  {
    _id: ObjectId("6405f062b0d06adeaeefc3bc"),
    username: 'dan',
    fullName: 'Daniel Bugl',
    age: 26
  }
]
```

As we can see, MongoDB automatically created a unique ID (`ObjectId`) for our document. This ID consists of 12 bytes in hexadecimal format (so each byte is displayed as two characters). The bytes are defined as follows:

- The first 4 bytes are a timestamp, representing the creation of the ID measured in seconds since the Unix epoch
- The next 5 bytes are a random value unique to the machine and currently running database process

- The last 3 bytes are a randomly initialized incrementing counter

## Note

The way `ObjectId` identifiers are generated in MongoDB ensures that IDs are unique, avoiding ID collisions even when two ids are generated at the same time from different instances, without requiring a form of communication between the instances, which would slow down the creation of documents when scaling the database.

## Querying and sorting documents

Now that we have inserted some documents, we can query them by accessing different fields from the object. We can also sort the list of documents returned from MongoDB. Follow these steps:

1. Before we get started querying, let's insert two more documents into our `users` collection:

```
2. > db.users.insertMany([
3.   { username: 'jane', fullName: 'Jane Doe', age: 32 },
4.   { username: 'john', fullName: 'John Doe', age: 30 }
5. ])
```

5. Now we can start querying for a certain username by using `findOne` and passing an object with the `username` field. When using `findOne`, MongoDB will return the first matching object:

```
> db.users.findOne({ username: 'jane' })
```

6. We can also query for full names, or any other field in the documents from the collection. When using `find`, MongoDB will return an array of all matches:

```
> db.users.find({ fullName: 'Daniel Bugl' })
```

7. An important thing to watch out for is that when querying an `ObjectId`, we need to wrap the ID string with an `ObjectId()` constructor, as follows:

```
> db.users.findOne({ _id: ObjectId('6405f062b0d06adeaeefc3bc') })
```

Make sure to change the string passed to the `ObjectId()` constructor to a valid `ObjectId` returned from the previous commands.

8. MongoDB also provides certain query operators, prefixed by `$`. For example, we can find everyone above the age of 30 in our collection by using the `$gt` operator, as follows:

```
> db.users.find({ age: { $gt: 30 } })
```

You will notice that `John Doe` does not get returned, because his age is exactly 30. If we want to match ages greater than or equal to 30, we need to use the `$gte` operator.

9. If we want to sort our results, we can use the `.sort()` method after `.find()`. For example, we can return all items from the `users` collection, sorted by age ascending (`1` for ascending, `-1` for descending):

```
> db.users.find().sort({ age: 1 })
```

## Updating documents

To update a document in MongoDB, we combine the arguments from the query and insert operations into a single operation. We can use the same criteria to filter documents as we did for `find()`. To update a single field from the document, we use the `$set` operator:

1. We can update the `age` field for the user with the username `dan` as follows:

```
> db.users.updateOne({ username: 'dan' }, { $set: { age: 27 } })
```

## Note

Just like `findOne`, `updateOne` only updates the first matching document. If we want to update all documents that match, we can use `updateMany`.

MongoDB will return an object with information about how many documents matched (`matchedCount`), how many were modified (`modifiedCount`), and how many were upserted (`upsertedCount`).

2. The `updateOne` method accepts a third argument, which is an `options` object. One useful option here is the `upsert` option, which, if set to `true`, will insert a document if it does not exist yet, and update it if it does exist already. Let's first try to update a non-existent user with `upsert: false`:

```
> db.users.updateOne({ username: 'new' }, { $set: { fullName: 'New User' } })
```

3. Now we set `upsert` to `true`, which inserts the user:

```
> db.users.updateOne({ username: 'new' }, { $set: { fullName: 'New User' } }, { upsert: true })
```

#### Note

If you want to remove a field from a document, use the `$unset` operator. If you want to replace the whole document with a new document, you can use the `replaceOne` method and pass a new document as the second argument to it.

#### Deleting documents

To delete documents from the database, MongoDB provides the `deleteOne` and `deleteMany` methods, which have a similar API to the `updateOne` and `updateMany` methods. The first argument is, again, used to match documents.

Let's say the user with the username `new` wants to delete their account. To do so, we need to remove them from the `users` collection. We can do so as follows:

```
> db.users.deleteOne({ username: 'new' })
```

That's all there is to it! As you can see, it is very simple to do CRUD operations in MongoDB if you already know how to work with JSON objects and JavaScript, making it the perfect database for a Node.js backend.

Now that we have learned how to access MongoDB using the MongoDB Shell, let's learn about accessing it from within VS Code.

## Accessing the database via VS Code

Up until now, we have been using the Terminal to access the database. We installed a MongoDB extension for VS Code. We can now use this extension to access our database in a more visual way:

1. Click on the MongoDB icon on the left sidebar (it should be a leaf icon) and click on the **Add Connection** button:

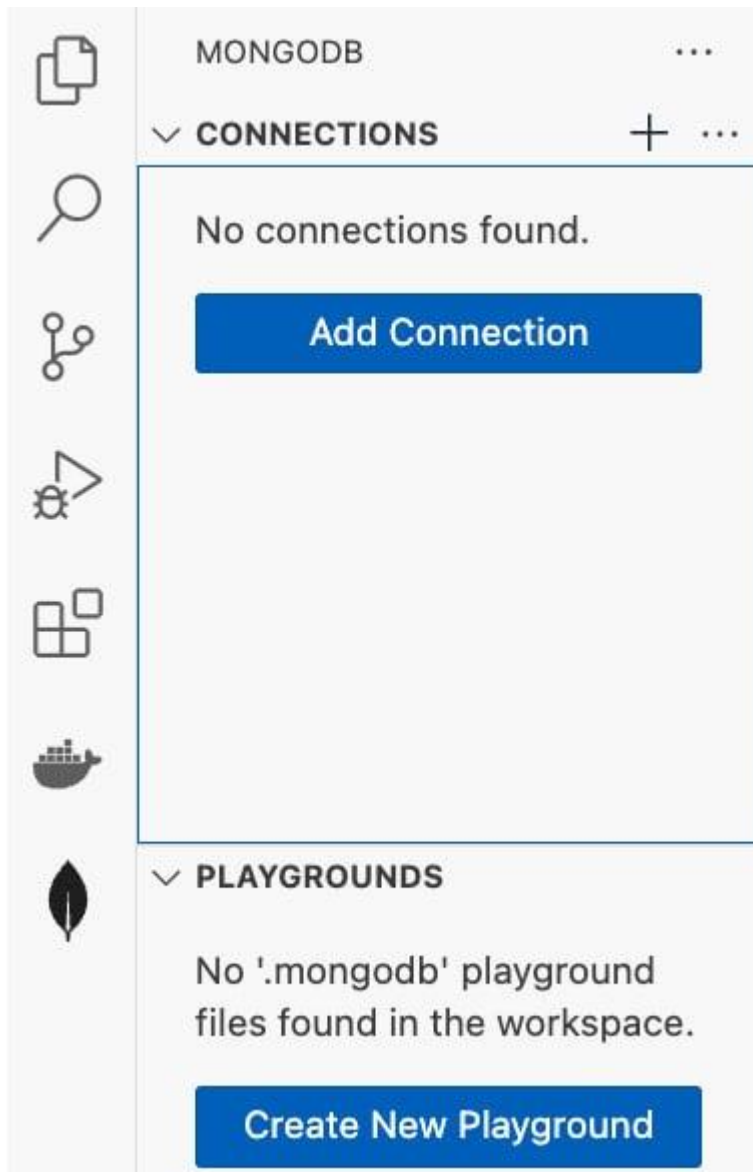


Figure 2.10 – The MongoDB sidebar in VS Code

2. A new **MongoDB** tab will open up. In this tab, click on **Connect** in the **Connect with Connection String** box:

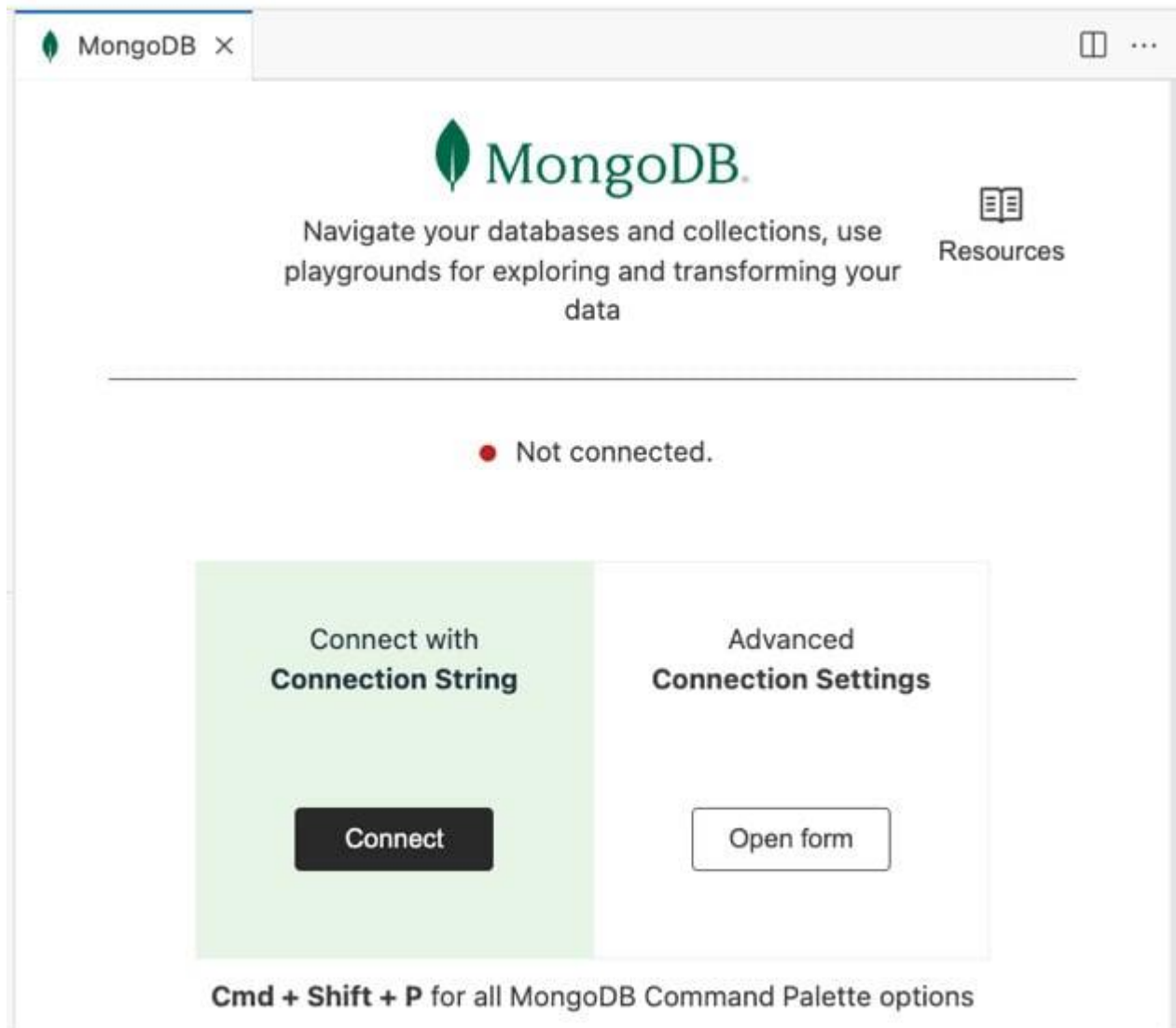


Figure 2.11 – Adding a new MongoDB connection in VS Code

3. A popup should open at the top. In this popup, enter the following connection string to connect to your local database:

```
mongodb://localhost:27017/
```

4. Press Return/Enter to confirm. A new connection will be listed in the MongoDB sidebar. You can browse the tree to view databases, collections, and documents. For example, click the first document to view it:

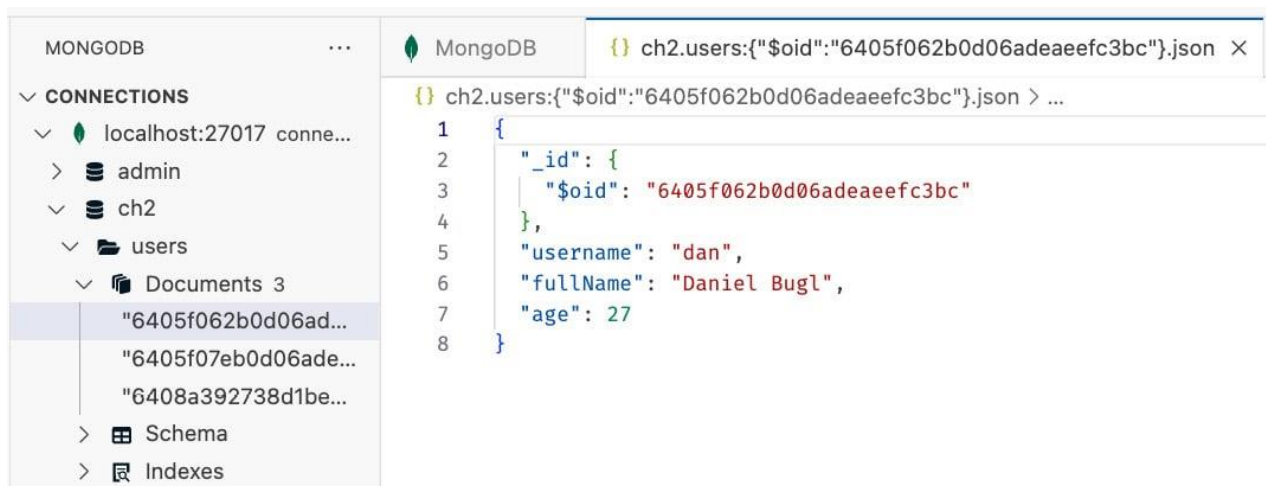


Figure 2.12 – Viewing a document in the MongoDB extension in VS Code

### Tip

You can also directly edit a document by editing a field in VS Code and saving the file. The updated document will automatically be saved to the database.

The MongoDB extension is very useful for debugging our database, as it lets us visually spot problems and quickly make edits to documents. Additionally, we can right-click on **Documents** and **Search for documents...** to open a new window where we can run MongoDB queries, just like we did in the Terminal. The queries can be executed on the database by clicking on the **Play** button in the top right. You may need to confirm a dialog with **Yes**, and then the results will show in a new pane, as can be seen in the following screenshot:

The screenshot shows the MongoDB Playground interface in VS Code. The left pane contains the MongoDB Playground code, and the right pane shows the results of the query.

```

1 // MongoDB Playground
2 // Use Ctrl+Space inside a snippet or
3
4 // The current database to use.
5 use('ch2');
6
7 // Search for documents in the current
8 db.getCollection('users')
9 .find(
10 {
11   /*
12    * Filter
13    * fieldA: value or expression
14    */
15 },
16 {
17   /*
18    * Projection
19    * _id: 0, // exclude _id
20    * fieldA: 1 // include field
21    */
22 }
23 )
24 .sort({
25   /*
26    * fieldA: 1 // ascending
27    * fieldB: -1 // descending
28    */
29 });

```

The right pane shows the results of the query, which are three documents from the 'users' collection:

```

1 [
2   {
3     "_id": {
4       "$oid": "6405f062b0d06adeaeefc3b"
5     },
6     "username": "dan",
7     "fullName": "Daniel Bugl",
8     "age": 27
9   },
10  {
11    "_id": {
12      "$oid": "6405f07eb0d06adeaeefc3b"
13    },
14    "username": "jane",
15    "fullName": "Jane Doe",
16    "age": 32
17  },
18  {
19    "_id": {
20      "$oid": "6408a392738d1be75915796"
21    },
22    "username": "john",
23    "fullName": "John Doe",
24    "age": 30
25  }
26 ]

```

Figure 2.13 – Querying MongoDB in VS Code

Now that we have learned the basics of using and debugging MongoDB databases, we can start integrating our database in a Node.js backend service, instead of simply storing and reading information from files.

## Accessing the MongoDB database via Node.js

We are now going to create a new web server that, instead of returning users from a JSON file, returns the list of users from our previously created `users` collection:

1. In the `ch2` folder, open a Terminal. Install the `mongodb` package, which contains the official MongoDB driver for Node.js:

```
$ npm install mongodb@6.3.0
```

2. Create a new `backend/mongodbweb.js` file and open it. Import the following:

```
3. import { createServer } from 'node:http'
   import { MongoClient } from 'mongodb'
```

4. Define the connection URL and database name and then create a new MongoDB client:

```
5. const url = 'mongodb://localhost:27017/'
6. const dbName = 'mongotest'
   const client = new MongoClient(url)
```

7. Connect to the database and log a message after we are connected successfully, or when there is an error with the connection:

```
8. try {
9.   await client.connect()
10.  console.log('Successfully connected to database!')
11. } catch (err) {
12.  console.error('Error connecting to database:', err)
13. }
```

13. Next, create an HTTP server, like we did before:

```
const server = createServer(async (req, res) => {
```

14. Then, select the database from the client, and the `users` collection from the database:

```
15.  const db = client.db(dbName)
    const users = db.collection('users')
```

16. Now, execute the `find()` method on the `users` collection. In the MongoDB Node.js driver, we also need to call the `toArray()` method to resolve the iterator to an array:

```
const usersList = await users.find().toArray()
```

17. Finally, set the status code and response header, and return the users list:

```
18.  res.statusCode = 200
19.  res.setHeader('Content-Type', 'application/json')
20.  res.end(JSON.stringify(usersList))
    })
```

21. Now that we have defined our server, copy over the code from before to listen to `localhost` on port `3000`:

```
22. const host = 'localhost'
23. const port = 3000
24. server.listen(port, host, () => {
25.   console.log(`Server listening on http://${host}:${port}`)
  })
```

26. Run the server by executing the script:

```
$ node backend/mongodbweb.js
```

27. Open `http://localhost:3000` in your browser and you should see the list of users from our database being returned:



Figure 2.14 – Our first Node.js service retrieving data from a MongoDB database!

As we have seen, we can use similar methods that we have used in the MongoDB Shell in Node.js as well. However, the APIs of the `node:http` module and the `mongodb` package are very low-level, requiring a lot of code to create an HTTP API and talk to the database.

In the next chapter, we are going to learn about libraries that abstract these processes to allow for easier creation of HTTP APIs and handling of the database. These libraries are Express and Mongoose. Express is a web framework that allows us to easily define API routes and handle requests. Mongoose allows us to create schemas for documents in our database to more easily create, read, update, and delete objects.

## Summary

In this chapter, we learned how to use Node.js to develop scripts that can run on a server. We also learned how to create containers with Docker, and how MongoDB works and can be interfaced with. At the end of this chapter, we even successfully created our first simple backend service using Node.js and MongoDB!

## Building and Deploying Our First Full-Stack Application with a REST API

In this part, we are going to be building and deploying our first full-stack application with a **REST API**. We will start by implementing a backend service using **Express** and **Mongoose ODM**. Then, we will create unit tests for it using **Jest**. After that, we will create a frontend with **React** and integrate it with our backend service using **TanStack Query**. Finally, we will deploy the application using **Docker** and learn how to set up a CI/CD pipeline.

This part includes the following chapters:

## Implementing a Backend Using Express, Mongoose ODM, and Jest

After learning the basics of Node.js and MongoDB, we will now put them into practice by building our first backend service using Express to provide a REST API, Mongoose **object data modeling (ODM)** to interface with MongoDB, and Jest to test our code. We will first learn how to structure a backend project using an architectural pattern. Then, we will create database schemas using Mongoose. Next, we will

make service functions to interface with the database schemas and write tests for them using Jest. Then, we will learn what REST is and when it is useful. Finally, we provide a REST API and serve it using Express. At the end of this chapter, we will have a working backend service to be consumed by a frontend developed in the next chapter.

In this chapter, we are going to cover the following main topics:

- Designing a backend service
- Creating database schemas using Mongoose
- Developing and testing service functions
- Providing a REST API using Express

## Designing a backend service

To design our backend service, we are going to use a variation of an existing architectural pattern called **model–view–controller (MVC)** pattern. The MVC pattern consists of the following parts:

- **Model:** Handles data and basic data logic
- **Controller:** Controls how data is processed and displayed
- **View:** Displays the current state

In traditional full-stack applications, the backend would render and display the frontend completely, and an interaction would usually require a full-page refresh. The MVC architecture was designed mainly for such applications. However, in modern applications, the frontend is usually interactive and rendered in the backend only through server-side rendering. In modern applications, we thus often distinguish between the actual backend service(s) and the backend for frontend (which handles static site generation and server-side rendering):

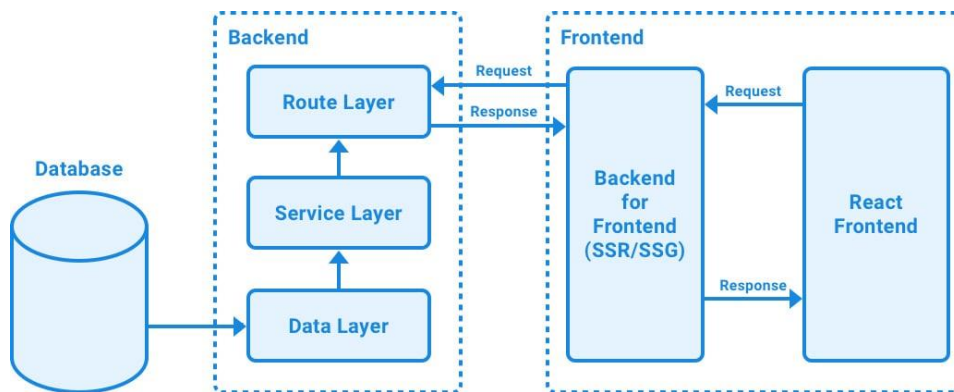


Figure 3.1 – A modern full-stack architecture, with a single backend service and a frontend with server-side rendering (SSR) and static-site generation (SSG)

For modern applications, the idea is that the backend service only deals with processing and serving requests and data and does not render the user interface anymore. Instead, we have a separate application that handles the frontend and server-side rendering of user interfaces specifically. To adapt to this change, we adjust the MVC architectural pattern to a data-service-route pattern for the backend service as follows:

- **Route layer:** Defines routes that consumers can access and handles user input by processing the request parameters and body and then calling service functions
- **Service layer:** Provides service functions, such as **create–read–update–delete (CRUD)** functions, which access the database through the data layer
- **Data layer:** Only deals with accessing the database and does basic validation to ensure that the database is consistent

This separation of concerns works best for services that only expose routes and do not deal with rendering user interfaces. Each layer in this pattern only deals with one step in processing the request.

After learning about the design of backend services, let's get started creating a folder structure reflecting what we have learned.

Our first application is going to be a blog application. For such an application, we are going to need the API to be able to do the following:

- Get a list of posts
- Get a single post
- Create a new post
- Update an existing post
- Delete an existing post

To provide these functions, we first need to create a database schema to define what a blog post object should look like in our database. Then, we need service functions to handle CRUD functionality for blog posts. Finally, we are going to define our REST API to query, create, update, and delete blog posts.

## Creating database schemas using Mongoose

Before we can get started defining the database schemas, we first need to set up Mongoose itself. Mongoose is a library that simplifies MongoDB object modeling by reducing the boilerplate code needed to interface with MongoDB. It also includes common business logic such as setting `createdAt` and `updatedAt` timestamps automatically and validation and type casting to keep the database state consistent.

Follow these steps to set up the `mongoose` library:

1. First, install the `mongoose` library:

```
$ npm install mongoose@8.0.2
```

2. Create a new `src/db/init.js` file and import `mongoose` there:

```
import mongoose from 'mongoose'
```

3. Define and export a function that will initialize the database connection:

```
export function initDatabase() {
```

4. First, we define `DATABASE_URL` to point to our local MongoDB instance running via Docker and specify `blog` as the database name:

```
const DATABASE_URL = 'mongodb://localhost:27017/blog'
```

The connection string is similar to what we used in the previous chapter when directly accessing the database via Node.js.

5. Then, add a listener to the `open` event on the Mongoose connection so that we can show a log message once we are connected to the database:

```
6. mongoose.connection.on('open', () => {  
7.   console.info('successfully connected to database:',  
   DATABASE_URL)  
   })
```

8. Now, use the `mongoose.connect()` function to connect to our MongoDB database and return the `connection` object:

```
9.   const connection = mongoose.connect(DATABASE_URL)  
10.  return connection  
   }
```

11. Create a new `src/example.js` file and import and run the `initDatabase` function there:

```
12. import { initDatabase } from './db/init.js'  
   initDatabase()
```

13. Run the `src/example.js` file using Node.js to see Mongoose successfully connecting to our database:

```
$ node src/example.js
```

As always, you can stop the server by pressing Ctrl + C in the Terminal.

We can see our log message being printed to the Terminal, so we know that Mongoose was able to successfully connect to our database! If there is an error, for example, because Docker (or the container) is not running, it will hang for a while and then throw an error about the connection being refused (`ECONNREFUSED`). In that case, make sure the Docker MongoDB container is running properly and can be connected to.

## Defining a model for blog posts

After initializing the database, the first thing we should do is define the data structure for blog posts. Blog posts in our system should have a title, an author, contents, and some tags associated with the post. Follow these steps to define the data structure for blog posts:

1. Create a new `src/db/models/` folder.
2. Inside that folder, create a new `src/db/models/post.js` file, import the `mongoose` and the `Schema` classes:

```
import mongoose, { Schema } from 'mongoose'
```

3. Define a new schema for posts:

```
const postSchema = new Schema({
```

4. Now specify all properties of a blog post and the corresponding types. We have a required `title`, an `author`, and `contents`, which are all strings:

```
5.   title: { type: String, required: true },
```

```
6.   author: String,  
    contents: String,
```

7. Lastly, we have `tags`, which are a string array:

```
8.   tags: [String],  
  })
```

9. Now that we have defined the schema, we can create a Mongoose model from it by using the `mongoose.model()` function:

```
export const Post = mongoose.model('post', postSchema)
```

### Note

The first argument to `mongoose.model()` specifies the name of the collection. In our case, the collection will be called `posts` because we specified `post` as the name. In Mongoose models, we need to specify the name of the document in singular form.

Now that we have defined the data structure and model for blog posts, we can start using it to create and query posts.

## Using the blog post model

After creating our model, let's try using it! For now, we are simply going to access it in the `src/example.js` file because we have not defined any service functions or routes yet:

1. Import the `Post` model in the `src/example.js` file:

```
2. import { initDatabase } from './db/init.js'
   import { Post } from './db/models/post.js'
```

3. The `initDatabase()` function we defined earlier is an `async` function, so we need to `await` it; otherwise, we would be attempting to access the database before we are connected to it:

```
await initDatabase()
```

4. Create a new blog post by calling `new Post()`, defining some example data:

```
5. const post = new Post({
6.   title: 'Hello Mongoose!',
7.   author: 'Daniel Bugl',
8.   contents: 'This post is stored in a MongoDB database using
   Mongoose.',
9.   tags: ['mongoose', 'mongodb'],
   })
```

10. Call `.save()` on the blog post to save it to the database:

```
await post.save()
```

11. Now we can use the `.find()` function to list all posts, and log the result:

```
12. const posts = await Post.find()
    console.log(posts)
```

13. Run the example script to see our post being inserted and listed:

```
$ node src/example.js
```

You will get the following result after running the preceding script:

```
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> node example.js
tags: [ 'mongoose', 'mongodb' ],
createdAt: 2025-10-14T00:28:05.092Z,
updatedAt: 2025-10-14T00:28:05.198Z,
__v: 0
},
{
  _id: new ObjectId('68ed9a2ccf78debcdf840bc5'),
  title: 'Hello Mongoose!',
  author: 'Daniel Bugl',
  contents: 'This post is stored in a MongoDB database using Mongoose.',
  tags: [ 'mongoose', 'mongodb' ],
  createdAt: 2025-10-14T00:32:44.771Z,
  updatedAt: 2025-10-14T00:32:44.771Z,
  __v: 0
}
```

Figure 3.2 – Our first document inserted via Mongoose!

As you can see, using Mongoose is very similar to using MongoDB directly. However, it offers us some wrappers around models for convenience, making it easier to deal with documents.

## Defining creation and last update dates in the blog post

You may have noticed that we have not added any dates to our blog post. So, we do not know when a blog post is created or when it was last updated. Mongoose makes implementing such functionality simple, let's try it out now:

1. Edit the `src/db/models/post.js` file and add a second argument to the `new Schema()` constructor. The second argument specifies options for the schema. Here, we set the `timestamps: true` setting:

```
2. const postSchema = new Schema(
3.   {
4.     title: String,
5.     author: String,
6.     contents: String,
7.     tags: [String],
8.   },
9.   { timestamps: true },
)
```

10. Now all we need to do is create a new blog post by running the example script, and we will see that the last post inserted now has `createdAt` and `updatedAt` timestamps:

```
$ node src/example.js
```

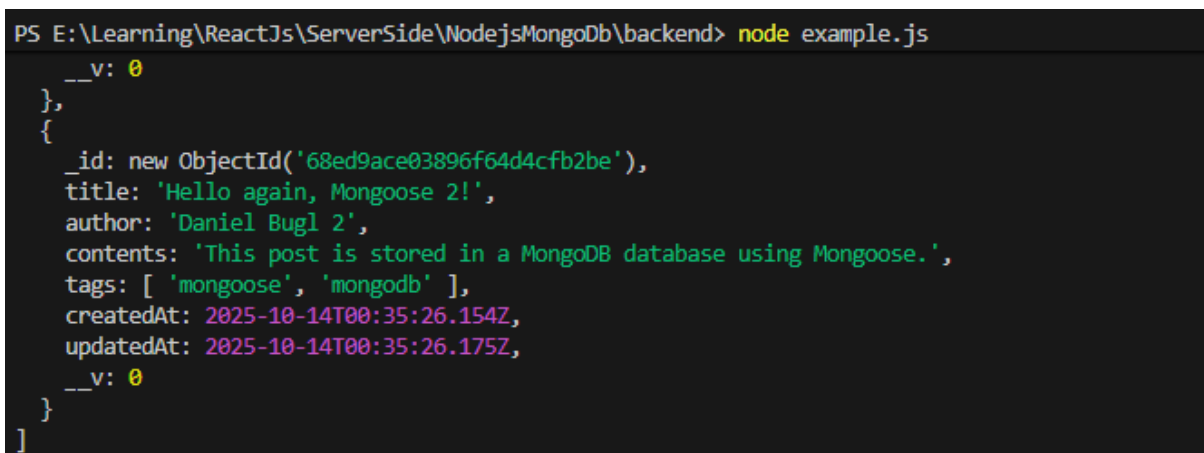
11. To see if the `updatedAt` timestamp works, let's try updating the created blog post by using the `findByIdAndUpdate` method. Save the result of `await post.save()` in a `createdPost` constant, then add the following code close to the end of the `src/example.js` file, before the `Post.find()` call:

```
12. const createdPost = await post.save()
13. await Post.findByIdAndUpdate(createdPost._id, {
14.   $set: { title: 'Hello again, Mongoose!' },
    })
```

15. Run the server again to see the blog posts being updated:

```
$ node src/example.js
```

You will get three posts, and the last one of them now looks as follows:



```
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> node example.js
[
  {
    __v: 0
  },
  {
    _id: new ObjectId('68ed9ace03896f64d4cfb2be'),
    title: 'Hello again, Mongoose 2!',
    author: 'Daniel Bugl 2',
    contents: 'This post is stored in a MongoDB database using Mongoose.',
    tags: [ 'mongoose', 'mongodb' ],
    createdAt: 2025-10-14T00:35:26.154Z,
    updatedAt: 2025-10-14T00:35:26.175Z,
    __v: 0
  }
]
```

Figure 3.3 – Our updated document with the automatically updated timestamps

As we can see, using Mongoose makes dealing with MongoDB documents much more convenient! Now that we have defined our database model, let's start developing (and writing tests for) service functions!

Code for the example can be viewed at

<https://github.com/ensatetechnologies/NodejsMongoDb>

## Developing and testing service functions

Up until now, we have always been testing code by putting it in the `src/example.js` file. Now, we are going to write some service functions and learn how to write actual tests for them by using Jest.

### Setting up the test environment

First, we are going to set up our test environment by following these steps:

1. Install `jest` and `mongodb-memory-server` as dev dependencies:

```
2. $ npm install --save-dev jest@29.7.0 \
  mongodb-memory-server@9.1.1
```

Jest is a test runner used to define and execute unit tests. The `mongodb-memory-server` library allows us to spin up a fresh instance of a MongoDB database, storing data only in memory, so that we can run our tests on a fresh database instance.

3. Create a `src/test/` folder to put the setup for our tests in.
4. In this folder, create a `src/test/globalSetup.js` file, where we will import `MongoMemoryServer` from the previously installed library:

```
import { MongoMemoryServer } from 'mongodb-memory-server'
```

5. Now define a `globalSetup` function, which creates a memory server for MongoDB:

```
6. export default async function globalSetup() {
  const instance = await MongoMemoryServer.create({
```

7. When creating the `MongoMemoryServer`, set the binary version to `6.0.4`, which is the same version that we installed for our Docker container:

```
8.     binary: {
9.       version: '6.0.4',
10.    },
    })
```

11. We will store the MongoDB instance as a global variable to be able to access it later in the `globalTeardown` function:

```
global.__MONGOINSTANCE = instance
```

12. We will also store the URL to connect to our test instance in the `DATABASE_URL` environment variable:

```
13.   process.env.DATABASE_URL = instance.getUri()
    }
```

14. Edit `src/db/init.js` and adjust the `DATABASE_URL` to come from the environment variable so that our tests will be using the correct database:

```
15. export function initDatabase() {
    const DATABASE_URL = process.env.DATABASE_URL
```

16. Additionally, create a `src/test/globalTeardown.js` file to stop the MongoDB instance when our tests are finished and add the following code inside it:

```
17. export default async function globalTeardown() {
18.   await global.__MONGOINSTANCE.stop()
    }
```

19. Now, create a `src/test/setupFileAfterEnv.js` file. Here, we will define a `beforeAll` function to initialize our database connection in Mongoose before all tests run and an `afterAll` function to disconnect from the database after all tests finish running:

```
20. import mongoose from 'mongoose'
21. import { beforeAll, afterAll } from '@jest/globals'
22. import { initDatabase } from '../db/init.js'
23. beforeAll(async () => {
24.   await initDatabase()
25. })
26. afterAll(async () => {
27.   await mongoose.disconnect()
    })
```

28. Then, create a new `jest.config.json` file in the root of our project where we will define the config for our tests. In the `jest.config.json` file, we first set the test environment to `node`:

```
29. {  
    "testEnvironment": "node",
```

30. Next, tell Jest to use the `globalSetup`, `globalTeardown`, and `setupFileAfterEnv` files we created earlier:

```
31.   "globalSetup": "<rootDir>/src/test/globalSetup.js",  
32.   "globalTeardown": "<rootDir>/src/test/globalTeardown.js",  
33.   "setupFilesAfterEnv":  
    [ "<rootDir>/src/test/setupFileAfterEnv.js" ]  
}
```

#### Note

In this case, `<rootDir>` is a special string that automatically gets resolved to the root directory by Jest. You do not need to manually fill in a root directory here.

13. Finally, edit the `package.json` file and add a `test` script, which will run Jest:

```
14.   "scripts": {  
15.     "test": "NODE_OPTIONS=--experimental-vm-modules jest",  
16.     "lint": "eslint src",  
17.     "prepare": "husky install"  
    },
```

#### Note

At the time of writing, the JavaScript ecosystem is still in the process of moving to the **ECMAScript module (ESM)** standard. However, Jest does not support it yet by default, so we need to pass the `--experimental-vm-modules` option when running Jest.

14. If we attempt running this script now, we will see that there are no tests found, but we can still see that Jest is set up and working properly:

```
$ npm test
```

If it's showing the following error : `'NODE_OPTIONS' is not recognized as an internal or external command`,

```
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend> npm test

> test
> NODE_OPTIONS=--experimental-vm-modules jest

'NODE_OPTIONS' is not recognized as an internal or external command,
operable program or batch file.
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb\backend>
```

Then run **npm install --save-dev cross-env**

Then in package.config add

```
{
  "type": "module",
  "scripts": {
    "test": "cross-env NODE_OPTIONS=--experimental-vm-modules jest --passWithNoTests"
  },
  "dependencies": {
    "mongodb": "^6.3.0",
    "mongodb-memory-server": "^9.1.1",
    "mongoose": "^8.0.2"
  },
  "devDependencies": {
    "cross-env": "^7.0.3",
    "jest": "^29.7.0"
  }
}
```

### npm test

```
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb> npm test

> test
> cross-env NODE_OPTIONS=--experimental-vm-modules jest --passWithNoTests

No tests found, exiting with code 0
```

Figure 3.4 – Jest is set up successfully, but we have not defined any tests yet

Now that our test environment is set up, we can start writing our service functions and unit tests. It is always a good idea to write unit tests right after writing service functions, as it means we will be able to debug them right away while still having their intended behavior fresh in our minds.

## Writing our first service function: createPost

For our first service function, we are going to make a function to create a new post. We can then write tests for it by verifying that the create function creates a new post with the specified properties. Follow these steps:

1. Create a new `src/services/posts.js` file.
2. In the `src/services/posts.js` file, first import the `Post` model:

```
import { Post } from '../db/models/post.js'
```

3. Define a new `createPost` function, which takes an object with `title`, `author`, `contents`, and `tags` as arguments and creates and returns a new post:

```
4. export async function createPost({ title, author, contents,
   tags }) {
5.   const post = new Post({ title, author, contents, tags })
6.   return await post.save()
   }
```

We specifically listed all properties that we want the user to be able to provide here instead of simply passing the whole object to the `new Post()` constructor. While we need to type more code this way, it allows us to have control over which properties a user should be able to set. For example, if we later add permissions to the database models, we may be accidentally allowing users to set those permissions here, if we forget to exclude those properties. For those security reasons, it is always good practice to have a list of allowed properties instead of simply passing down the whole object.

After writing our first service function, let's continue by writing test cases for it.

## Defining test cases for the createPost service function

To test if the `createPost` function works as expected, we are going to define unit test cases for it using Jest:

1. Create a new `src/__tests__` folder, which will contain all test definitions.

#### Note

Alternatively, test files can also be co-located with the related files that they are testing. However, we use the `__tests__` directory to make it easier to distinguish tests from other files.

2. Create a new `src/__tests__/posts.test.js` file for our tests related to posts. In this file, start by importing `mongoose` and the `describe`, `expect`, and `test` functions from `@jest/globals`:

```
3. import mongoose from 'mongoose'
   import { describe, expect, test } from '@jest/globals'
```

4. Also import the `createPost` function from our services and the `Post` model:

```
5. import { createPost } from '../services/posts.js'
   import { Post } from '../db/models/post.js'
```

6. Then, use the `describe()` function to define a new test. This function describes a group of tests. We can call our group `creating posts`:

```
describe('creating posts', () => {
```

7. Inside the group, we will define a test by using the `test()` function. We can pass an `async` function here to be able to use `async/await` syntax. We call the first test `creating posts with all parameters should succeed`:

```
test('with all parameters should succeed', async () => {
```

8. Inside this test, we will use the `createPost` function to create a new post with some parameters:

```
9.     const post = {
10.       title: 'Hello Mongoose!',
11.       author: 'Daniel Bugl',
12.       contents: 'This post is stored in a MongoDB database
   using Mongoose.',
13.       tags: ['mongoose', 'mongodb'],
14.     }

   const createdPost = await createPost(post)
```

15. Then, verify that it returns a post with an ID by using the `expect()` function from Jest and the `toBeInstanceOf` matcher to verify that it is an `ObjectId`:

```
expect(createdPost._id).toBeInstanceOf(mongoose.Types.ObjectId)
```

16. Now use Mongoose directly to find the post with the given ID:

```
const foundPost = await Post.findById(createdPost._id)
```

17. We `expect()` the `foundPost` to equal an object containing at least the properties of the original post object we defined. Additionally, we expect the created post to have `createdAt` and `updatedAt` timestamps:

```
18.   expect(foundPost).toEqual(expect.objectContaining(post))
19.   expect(foundPost.createdAt).toBeInstanceOf(Date)
20.   expect(foundPost.updatedAt).toBeInstanceOf(Date)
    })
```

21. Additionally, define a second test, called `creating posts without title should fail`. As we defined the `title` to be required, it should not be possible to create a post without one:

```
22.   test('without title should fail', async () => {
23.     const post = {
24.       author: 'Daniel Bugl',
25.       contents: 'Post with no title',
26.       tags: ['empty'],
    }
```

27. Use a `try/catch` construct to catch the error and `expect()` the error to be a Mongoose `ValidationError`, which tells us that the title is required:

```
28.     try {
29.       await createPost(post)
30.     } catch (err) {
31.       expect(err).toBeInstanceOf(mongoose.Error.ValidationError)
32.       expect(err.message).toContain(`title` is required)
33.     }
    })
```

34. Finally, make a test called `creating posts with minimal parameters should succeed` and only enter the `title`:

```
35.   test('with minimal parameters should succeed', async () =>
36.     {
37.       const post = {
38.         title: 'Only a title',
39.       }
40.       const createdPost = await createPost(post)
41.       expect(createdPost._id).toBeInstanceOf(mongoose.Types.ObjectId)
42.     })
```

Package.config

```
{
  "type": "module",
  "jest": {
    "testEnvironment": "node",
    "setupFilesAfterEnv": []
  },
  "scripts": {
    "test": "cross-env NODE_OPTIONS=--experimental-vm-modules jest --passWithNoTests"
  },
  "dependencies": {
    "mongodb": "^6.3.0",
    "mongodb-memory-server": "^9.1.1",
```

```
"dotenv": "^16.4.5",  
  
"mongoose": "^8.0.2"  
  
},  
  
"devDependencies": {  
  
  "cross-env": "^7.0.3",  
  
  "jest": "^29.7.0"  
  
}  
  
}
```

```
Init.js  
  
import 'dotenv/config'  
  
import mongoose from 'mongoose'  
  
  
export function initDatabase() {  
  
  const DATABASE_URL = process.env.DATABASE_URL ||  
'mongodb://127.0.0.1:27017/blog'  
  
  mongoose.connection.on('open', () => {  
  
    console.info('successfully connected to database:', DATABASE_URL)  
  
  })  
  
  return mongoose.connect(DATABASE_URL)  
  
}
```

```
export async function closeDatabase() {  
  
  if (mongoose.connection.readyState) {  
  
    await mongoose.connection.close()  
  
  }  
  
}
```

Now that we have defined our tests, run the script we defined earlier:

```
$ npm test
```

```
PASS backend/__tests__/posts.test.js  
  creating posts  
    ✓ with all parameters should succeed (65 ms)  
    ✓ without title should fail (6 ms)  
    ✓ with minimal parameters should succeed (7 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      3 passed, 3 total  
Snapshots:  0 total  
Time:       1.611 s, estimated 3 s  
Ran all test suites.
```

As we can see, using unit tests we can do isolated tests on our service functions without having to define and manually access routes or write some manual test setups. These tests also have the added advantage that when we change code later, we can ensure that the previously defined behavior did not change by re-running the tests.

## Defining a function to list posts

After defining a function to create posts, we are now going to define an internal `listPosts` function, which allows us to query posts and define a sort order. Then, we are going to use this function to define `listAllPosts`, `listPostsByAuthor`, and `listPostsByTag` functions:

1. Edit the `src/services/posts.js` file and define a new function at the end of the file.

The function accepts a `query` and an `options` argument (with `sortBy` and `sortOrder` properties). With `sortBy`, we can define which field we want to sort by, and the `sortOrder` argument allows us to specify whether posts should be sorted in ascending or descending order. By default, we list all posts (empty object as query) and show the newest posts first (sorted by `createdAt`, in `descending` order):

```
async function listPosts(  
  query = {},  
  { sortBy = 'createdAt', sortOrder = 'descending' } = {},  
) {
```

2. We can use the `.find()` method from our Mongoose model to list all posts, passing an argument to sort them:

```
3.   return await Post.find(query).sort({ [sortBy]: sortOrder })  
   }
```

4. Now we can define a function to list all posts, which simply passes an empty object as query:

```
5. export async function listAllPosts(options) {  
6.   return await listPosts({}, options)  
   }
```

7. Similarly, we can create a function to list all posts by a certain author by passing `author` to the query object:

```
8. export async function listPostsByAuthor(author, options) {  
9.   return await listPosts({ author }, options)  
   }
```

10. Lastly, define a function to list posts by tag:

```
11. export async function listPostsByTag(tags, options) {  
12.   return await listPosts({ tags }, options)  
   }
```

In MongoDB, we can simply match strings in an array by matching the string as if it was a single value, so all we need to do is add a query for `tags: 'nodejs'`. MongoDB will then return all documents that have a `'nodejs'` string in their `tags` array.

### Note

The `{ [variable]: ... }` operator resolves the string stored in the `variable` to a key name for the created object. So, if our variable contains `'createdAt'`, the resulting object will be `{ createdAt: ... }`.

After defining the list post function, let's also write test cases for it.

## Defining test cases for list posts

Defining test cases for list posts is similar to create posts. However, we now need to create an initial state where we already have some posts in the database to be able to test the list functions. We can do this by using the `beforeEach()` function, which executes some code before each test case is executed. We can use the `beforeEach()` function for a whole test file or only run it for each test inside a `describe()` group. In our case, we are going to define it for the whole file, as the sample posts will come in handy when we test the delete post function later:

1. Edit the `src/__tests__/posts.js` file, adjust the `import` statement to import the `beforeEach` function from `@jest/globals` and import the various functions to list posts from our services:

```
2. import { describe, expect, test, beforeEach } from  
   '@jest/globals'  
3. import { createPost,  
4.   listAllPosts,  
5.   listPostsByAuthor,  
6.   listPostsByTag,  
   } from '../services/posts.js'
```

7. At the end of the file, define an array of sample posts:

```
8. const samplePosts = [  
9.   { title: 'Learning Redux', author: 'Daniel Bugl', tags:  
    ['redux'] },  
10.  { title: 'Learn React Hooks', author: 'Daniel Bugl', tags:  
    ['react'] },  
11.  {  
12.    title: 'Full-Stack React Projects',  
13.    author: 'Daniel Bugl',  
14.    tags: ['react', 'nodejs'],  
15.  },  
16.  { title: 'Guide to TypeScript' },  
  ]
```

17. Now, define an empty array, which will be populated with the created posts.

Then, define a `beforeEach` function, which first clears all posts from the database and clears the array of created sample posts and then creates the sample posts in the database again for each of the posts defined in the array earlier. This ensures that we have a consistent state of the database before each test case runs and that we have an array to compare against when testing the list post functions:

```
18. let createdSamplePosts = []  
19. beforeEach(async () => {  
20.   await Post.deleteMany({})  
21.   createdSamplePosts = []  
22.   for (const post of samplePosts) {  
23.     const createdPost = new Post(post)  
24.     createdSamplePosts.push(await createdPost.save())  
25.   }  
  })
```

To ensure that our unit tests are modular and independent from each other, we insert posts into the database directly by using Mongoose functions (instead of the `createPost` function).

26. Now that we have some sample posts ready, let's write our first test case, which should simply list all posts. We will define a new test group for `listing`

`posts` and a test to verify that all sample posts are listed by the `listAllPosts()` function:

```
27. describe('listing posts', () => {
28.   test('should return all posts', async () => {
29.     const posts = await listAllPosts()
30.     expect(posts.length).toEqual(createdSamplePosts.length)
    })
  })
```

31. Next, make a test that verifies that the default sort order shows newest posts first. We sort the `createdSamplePosts` array manually by `createdAt` (descending) and then compare the sorted dates to those returned from the `listAllPosts()` function:

```
32.   test('should return posts sorted by creation date
    descending by default', async () => {
33.     const posts = await listAllPosts()
34.     const sortedSamplePosts = createdSamplePosts.sort(
35.       (a, b) => b.createdAt - a.createdAt,
36.     )
37.     expect(posts.map((post) => post.createdAt)).toEqual(
38.       sortedSamplePosts.map((post) => post.createdAt),
39.     )
    })
```

## Note

The `.map()` function applies a function to each element of an array and returns the result. In our case, we select the `createdAt` property from all elements of the array. We cannot directly compare the arrays with each other because Mongoose returns documents with a lot of additional information in hidden metadata, which Jest will attempt to compare.

6. Additionally, define a test case where the `sortBy` value is changed to `updatedAt`, and the `sortOrder` value is changed to `ascending` (showing oldest updated posts first):

```
7.   test('should take into account provided sorting options',
    async () => {
8.     const posts = await listAllPosts({
9.       sortBy: 'updatedAt',
```

```
10.     sortOrder: 'ascending',
11.   })
12.   const sortedSamplePosts = createdSamplePosts.sort(
13.     (a, b) => a.updatedAt - b.updatedAt,
14.   )
15.   expect(posts.map((post) => post.updatedAt)).toEqual(
16.     sortedSamplePosts.map((post) => post.updatedAt),
17.   )
    })
```

18. Then, add a test to ensure that listing posts by author works:

```
19.   test('should be able to filter posts by author', async ()
    => {
20.     const posts = await listPostsByAuthor('Daniel Bugl')
21.     expect(posts.length).toBe(3)
    })
```

#### Note

We are controlling the test environment by creating a specific set of sample posts before each test case runs. We can make use of this controlled environment to simplify our tests. As we already know that there are only three posts with that author, we can simply check if the function returned exactly three posts. Doing so keeps our tests simple, and they are still safe because we control the environment completely.

8. Finally, add a test to verify that listing posts by tag works:

```
9.   test('should be able to filter posts by tag', async () => {
10.     const posts = await listPostsByTag('nodejs')
11.     expect(posts.length).toBe(1)
12.   })
    })
```

13. Run the tests again and watch them all pass:

```
$ npm test
```

```
PS E:\Learning\ReactJs\ServerSide\NodejsMongoDb> npm test
  ✓ with minimal parameters should succeed (12 ms)
listing posts
  ✓ should return all posts (16 ms)
  ✓ should return posts sorted by creation date descending
  ✓ should take into account provided sorting options (16 ms)
  ✓ should be able to filter posts by author (12 ms)
  ✓ should be able to filter posts by tag (12 ms)

Test Suites: 1 passed, 1 total
Tests:      8 passed, 8 total
Snapshots:  0 total
Time:       1.566 s, estimated 2 s
Ran all test suites.
```

As we can see, for some tests, we need to prepare an initial state. In our case, we only had to create some posts, but this initial state may become more sophisticated. For example, on a more advanced blogging platform, it may be necessary to create a user account first, then create a blog on the platform, and then create blog posts for that blog. In that case, we could create test utility functions, such as `createTestUser`, `createTestBlog`, `createTestPost` and import them in our tests. We can then use these functions in `beforeEach()` across multiple test files instead of manually doing it every single time. Depending on your application structure, different test utility functions may be needed, so feel free to define them as you see fit.

After defining the test cases for the list posts function, let's continue by defining the get single post, update post, and delete post functions.

## Defining the get single post, update and delete post functions

The get single post, update and delete post functions can be defined very similarly to the list posts function. Let's do that quickly now:

1. Edit the `src/services/posts.js` file and define a `getPostById` function as follows:
2. export async function `getPostById(postId)` {
3.   return await `Post.findById(postId)`
- }

It may seem a bit trivial to define a service function that just calls `Post.findById`, but it is good practice to define it anyway. Later, we may want to add some additional restrictions, such as access control. Having the service function allows us to change it only in one place and we do not have to worry about forgetting to add it somewhere. Another benefit is that if we, for example, want to change the database provider later, the developer only needs to worry about getting the service functions working again, and they can be verified with the test cases.

4. In the same file, define the `updatePost` function. It will take an ID of an existing post, and an object of parameters to be updated. We are going to use the `findOneAndUpdate` function from Mongoose, together with the `$set` operator, to change the specified parameters. As a third argument, we provide an options object with `new: true` so that the function returns the modified object instead of the original:

```
5. export async function updatePost(postId, { title, author,
6.   contents, tags }) {
7.   return await Post.findOneAndUpdate(
8.     { _id: postId },
9.     { $set: { title, author, contents, tags } },
10.    { new: true },
11.  )
12. }
```

11. In the same file, also define a `deletePost` function, which simply takes the ID of an existing post and deletes it from the database:

```
12. export async function deletePost(postId) {
13.   return await Post.deleteOne({ _id: postId })
14. }
```

### Tip

Depending on your application, you may want to set a `deletedOn` timestamp instead of deleting it right away. Then, set up a function that gets all documents that have been deleted for more than 30 days and delete them. Of course, this means that we need to always filter out already deleted posts in the `listPosts` function and that we need to write test cases for this behavior!

4. Edit the `src/__tests__/posts.js` file and import the `getPostById` function:

```
5. getPostById,  
   } from '../services/posts.js'
```

6. Add tests for getting a post by ID and failing to get a post because the ID did not exist in the database:

```
7. describe('getting a post', () => {  
8.   test('should return the full post', async () => {  
9.     const post = await getPostById(createdSamplePosts[0]._id)  
10.    expect(post.toObject()).toEqual(createdSamplePosts[0].toObject())  
11.  })  
12.   test('should fail if the id does not exist', async () => {  
13.     const post = await  
14.       getPostById('000000000000000000000000')  
15.     expect(post).toEqual(null)  
16.   })  
17. })
```

In the first test, we use `.toObject()` to convert the Mongoose object with all its internal properties and metadata to a **plain old JavaScript object (POJO)** so that we can compare it to the sample post object by comparing all properties.

16. Next, import the `updatePost` function:

```
17. updatePost,  
   } from '../services/posts.js'
```

18. Then, add tests for updating a post successfully. We add one test to verify that the specified property was changed and another test to verify that it does not interfere with other properties:

```
19. describe('updating posts', () => {  
20.   test('should update the specified property', async () => {  
21.     await updatePost(createdSamplePosts[0]._id, {  
22.       author: 'Test Author',  
23.     })  
24.     const updatedPost = await  
25.       Post.findById(createdSamplePosts[0]._id)  
26.     expect(updatedPost.author).toEqual('Test Author')
```

```
26.   })
27.   test('should not update other properties', async () => {
28.     await updatePost(createdSamplePosts[0]._id, {
29.       author: 'Test Author',
30.     })
31.     const updatedPost = await
      Post.findById(createdSamplePosts[0]._id)
32.     expect(updatedPost.title).toEqual('Learning Redux')
  })
```

33. Additionally, add a test to ensure the `updatedAt` timestamp was updated. To do so, first convert the `Date` objects to numbers by using `.getTime()`, and then we can compare them by using the `expect(...).toBeGreaterThan(...)` matcher:

```
34.   test('should update the updatedAt timestamp', async () => {
35.     await updatePost(createdSamplePosts[0]._id, {
36.       author: 'Test Author',
37.     })
38.     const updatedPost = await
      Post.findById(createdSamplePosts[0]._id)
39.     expect(updatedPost.updatedAt.getTime()).toBeGreaterThan(
40.       createdSamplePosts[0].updatedAt.getTime(),
41.     )
  })
```

42. Also add a failing test to see if the `updatePost` function returns `null` when no post with a matching ID was found:

```
43.   test('should fail if the id does not exist', async () => {
44.     const post = await updatePost('000000000000000000000000',
45.     {
46.       author: 'Test Author',
47.     })
48.     expect(post).toEqual(null)
  })
```

49. Lastly, import the `deletePost` function:

```
50.   deletePost,
  } from '../services/posts.js'
```

51. Then, add tests for successful and unsuccessful deletes by checking if the post was deleted and verifying the returned `deletedCount`:

```
52. describe('deleting posts', () => {
53.   test('should remove the post from the database', async ()
    => {
54.     const result = await
      deletePost(createdSamplePosts[0]._id)
55.     expect(result.deletedCount).toEqual(1)
56.     const deletedPost = await
      Post.findById(createdSamplePosts[0]._id)
57.     expect(deletedPost).toEqual(null)
58.   })
59.   test('should fail if the id does not exist', async () => {
60.     const result = await
      deletePost('000000000000000000000000')
61.     expect(result.deletedCount).toEqual(0)
62.   })
  })
```

63. Finally, run all tests again; they should all pass:

```
$ npm test
```

```
PS E:\Learning\ReactJS\ServerSide\NodejsMongoDb> npm test
  ✓ should update the specified property (18 ms)
  ✓ should not update other properties (14 ms)
  ✓ should update the updatedAt timestamp (18 ms)
  ✓ should fail if the id does not exist (12 ms)
deleting posts
  ✓ should remove the post from the database (12 ms)
  ✓ should fail if the id does not exist (12 ms)

Test Suites: 1 passed, 1 total
Tests:       16 passed, 16 total
Snapshots:   0 total
Time:        1.661 s, estimated 2 s
Ran all test suites.
```

Writing tests for service functions may be tedious, but it will save us a lot of time in the long run. Adding additional functionality later, such as access control, may change the basic behavior of the service functions. By having the unit tests, we can ensure that we do not break existing behavior when adding new functionality.

## Using the Jest VS Code extension

Up until now, we have run our tests by using Jest via the Terminal. There is also a Jest extension for VS Code, which we can use to make running tests more visual. The extension is especially helpful for larger projects where we have many tests in multiple files. Additionally, the extension can automatically watch and re-run tests if we change the definitions. We can install the extension as follows:

1. Go to the **Extensions** tab in the VS Code sidebar.
2. Enter `Orta.vscode-jest` in the search box to find the Jest extension.
3. Install the extension by pressing the **Install** button.
4. Now go to the newly added test icon on the sidebar (it should be a chemistry flask icon):

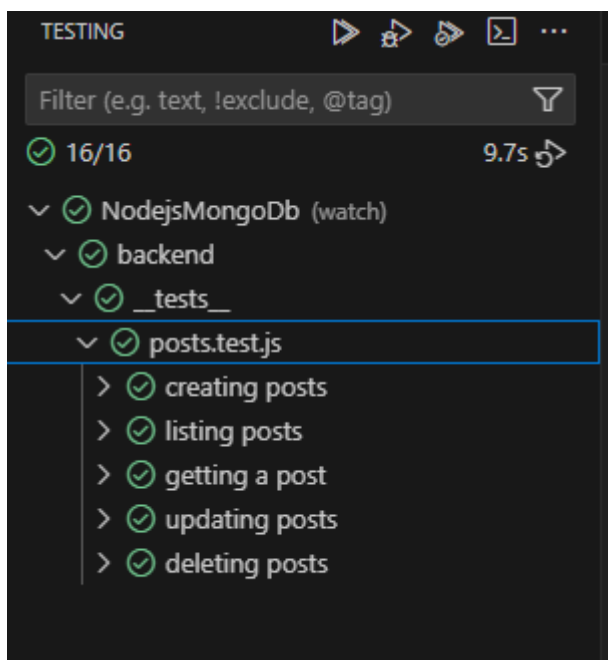


Figure 3.6 – The Testing tab in VS Code provided by the Jest extension

The Jest extension provides us an overview of all tests that we have defined. We can hover over them and press on the **Play** icon to re-run a specific test. By default, the Jest extension enables **auto-run-watch** (as can be seen in Figure 3.6). If **auto-run-watch** is enabled, the extension will re-run tests automatically when test definition files are saved. That's pretty handy!

Now that we have defined and tested our service functions, we can start using them when defining routes, which we are going to do next!

## Providing a REST API using Express

Having our data and service layers set up, we have a good framework for being able to write our backend. However, we still need an interface that lets users access our backend. This interface will be a **representational state transfer (REST)** API. A REST API provides a way to access our server via HTTP requests, which we can make use of when we develop our frontend.

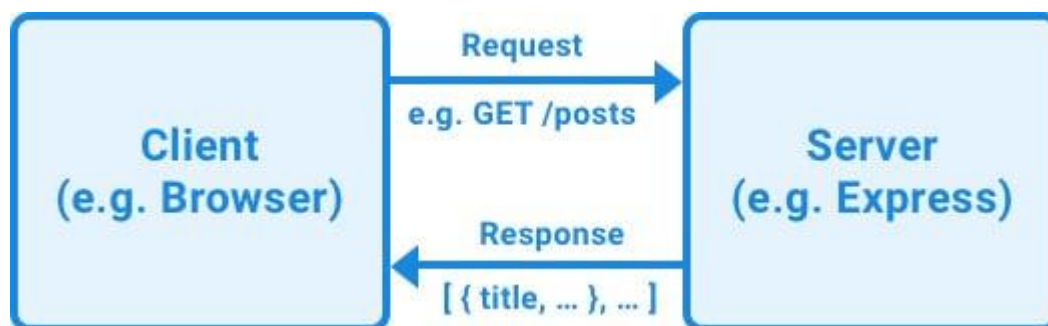


Figure 3.7 – The interaction between client and server using HTTP requests

As we can see, clients can send requests to our backend server, and the server will respond to them. There are five commonly used methods in a REST-based architecture:

- **GET:** This is used to read resources. Generally, it should not influence the database state and, given the same input, it should return the same output (unless the database state was changed through other requests). This behavior is called **idempotence**. In response to a successful GET request, a server usually returns the resource(s) with a 200 OK status code.
- **POST:** This is used to create new resources, from the information provided in the request body. In response to a successful POST request, a server usually either returns the newly created object with a 201 Created status code or

returns an empty response (with 201 Created status code) with a URL in the **Location** header that points to the newly created resource.

- **PUT:** This is used to update an existing resource with a given ID, replacing the resource completely with the new data provided in the request body. In some cases, it can also be used to create a new resource with a client-specified ID. In response to a successful PUT request, a server either returns the updated resource with a 200 OK status code, 204 No Content if it does not return the updated resource, or 201 Created if it created a new resource.
- **PATCH:** This is used to modify an existing resource with a given ID, only updating the fields specified in the request body instead of replacing the whole resource. In response to a successful PATCH request, a server either returns the updated resource with 200 OK or 204 No Content if it does not return the updated resource.
- **DELETE:** This is used to delete a resource with a given ID. In response to a successful DELETE request, a server either returns the deleted resource with 200 OK or 204 No Content if it does not return the deleted resource.

HTTP REST API routes are usually defined in a folder-like structure. It is always a good idea to prefix all routes with `/api/v1/` (`v1` being the version of the API definition, starting with `1`). If we want to change the API definition later, we can then easily run `/api/v1/` and `/api/v2/` in parallel for a while until everything is migrated.

## Defining our API routes

Now that we have learned how HTTP REST APIs work, let's start by defining routes for our backend, covering all functionality we have already implemented in the service functions:

- `GET /api/v1/posts`: Get a list of all posts
- `GET /api/v1/posts?sortBy=updatedAt&sortOrder=ascending`: Get a list of all posts sorted by `updatedAt` (ascending)

Note

Everything after the `?` symbol is called a query string and follows the format `key1=value1&key2=value2&...`. The query string can be used to provide additional optional parameters to a route.

- `GET /api/v1/posts?author=daniel`: Get a list of posts by author “daniel”
- `GET /api/v1/posts?tag=react`: Get a list of posts with the tag `react`
- `GET /api/v1/posts/:id`: Get a single post by ID
- `POST /api/v1/posts`: Create a new post
- `PATCH /api/v1/posts/:id`: Update an existing post by ID
- `DELETE /api/v1/posts/:id`: Delete an existing post by ID

As we can see, by putting together our already developed service functions and what we have learned about REST APIs, we can easily define routes for our backend. Now that we have defined our routes, let’s set up Express and our backend server to be able to expose those routes.

#### Note

This is just one example of how a REST API can be designed. It is intended as an example to get you started with full-stack development. Later, on your own time, feel free to check out other resources, such as <https://standards.rest>, to deepen your knowledge of REST API designs.

## Setting up Express

Express is a web application framework for Node.js. It provides utility functions to easily define routes for REST APIs and serve HTTP servers. Express is also very extensible, and there are many plugins for it in the JavaScript ecosystem.

#### Note

While Express is the most well-known framework at the time of writing, there are also newer ones, such as Koa (<https://koajs.com>) or Fastify (<https://fastify.dev>). Koa is designed by the team behind Express but aims to be smaller, more expressive, and

more robust. Fastify focuses on efficiency and low overhead. Feel free to check these out on your own time to see if they fit your requirements better.

Before we can set up the routes, let's take some time to set up our Express application and backend server by following these steps:

1. First, install the `express` dependency:

```
$ npm install express@4.18.2
```

2. Create a new `src/app.js` file. This file will contain everything needed to set up our Express app. In this file, first import `express`:

```
import express from 'express'
```

3. Then create a new Express app, as follows:

```
const app = express()
```

4. Now we can define routes on the Express app. For example, to define a GET route, we can write the following code:

```
5. app.get('/', (req, res) => {  
6.   res.send('Hello from Express!')  
   })
```

7. We export the app to be able to use it in other files:

```
export { app }
```

8. Next, we need to create a server and specify a port, similar to what we did before when creating an HTTP server. To do so, we create a new `src/index.js` file. In this file, we import the Express app:

```
import { app } from './app.js'
```

9. Then, we define a port, make the Express app listen to it, and log a message telling us where the server is running:

```
10. const PORT = 3000  
11. app.listen(PORT)  
    console.info(`express server running on  
    http://localhost:${PORT}`)
```

12. Edit `package.json` and add a `start` script to run our server:

```
13.   "scripts": {  
      "start": "node src/index.js",
```

14. Run the backend server by executing the following command:

```
$ npm start
```

15. Now, navigate to <http://localhost:3000/> in your browser and you will see **Hello from Express!** Being printed, just like before with the plain http server:



Figure 3.8 – Accessing our first Express app from the browser!

That's all there is to setting up a simple Express app! We can now keep defining routes by using `app.get()` for GET routes, `app.post()` for POST routes, etc. However, before we start developing our routes, let's take some time to improve our development environment. First, we should make `PORT` and `DATABASE_URL` configurable so that we can change them without having to change the code. To do so, we are going to use environment variables.

## Using dotenv for setting environment variables

A good way to load environment variables is using `dotenv`, which loads environment variables from `.env` files into our `process.env`. This makes it easy to define environment variables for local development while keeping it possible to set them differently in, for example, a testing environment. Follow these steps to set up `dotenv`:

1. Install the `dotenv` dependency:

```
$ npm install dotenv@16.3.1
```

2. Edit `src/index.js`, import `dotenv` there, and call `dotenv.config()` to initialize the environment variables. We should do this before we call any other code in our app:

```
3. import dotenv from 'dotenv'  
   dotenv.config()
```

4. Now we can start replacing our static variables with environment variables. Edit `src/index.js` and replace the static port `3000` with `process.env.PORT`:

```
const PORT = process.env.PORT
```

5. We have already migrated the `initDatabase` function to use `process.env.DATABASE_URL` earlier when we set up Jest. Now, we can edit `src/index.js` and import `initDatabase` there:

```
import { initDatabase } from './db/init.js'
```

6. Adjust the existing code to first call `initDatabase`, and only when the database initialized, start the Express app. We can now also handle errors while connecting to the database by adding a try/catch block:

```
7. try {  
8.   await initDatabase()  
9.   const PORT = process.env.PORT  
10.  app.listen(PORT)  
11.  console.info(`express server running on  
    http://localhost:${PORT}`)  
12. } catch (err) {  
13.   console.error('error connecting to database:', err)  
14. }
```

14. Finally, create a `.env` file in the root of the project and define the two environment variables there:

```
15. PORT=3000  
    DATABASE_URL=mongodb://localhost:27017/blog
```

16. We should exclude the `.env` file from the Git repository, as it is only used for local development. Edit `.gitignore` and add `.env` to it in a new line:

```
.env
```

At the moment, we have no sensible information in our environment variables, but it is still a good practice to do this already now. Later, we may have some credentials in the environment variables, which we do not want to accidentally push to a Git repository.

17. To make it easier for someone to get started with our project, we can create a copy of our `.env` file and duplicate it to `.env.template`, making sure that it does not contain any sensitive credentials, of course! Sensitive credentials could instead be stored in, for example, a shared password manager.
18. If it is still running from before, stop the server (by pressing Ctrl + C in the Terminal) and start it again as follows:

```
$ npm start
```

You will get the following result:

```
○ → ~/D/F/ch3 ↵ main± > npm start

> ch3@0.0.0 start
> node src/index.js

successfully connected to database: mongodb://localhost:27017/ch3
express server running on http://localhost:3000
```

Figure 3.9 – Initializing the database connection and the Express server with environment variables

As we can see, `dotenv` makes it easy to maintain environment variables for development while still allowing us the possibility to change them in a continuous integration, testing, or production environment.

You may have noticed that we need to manually restart the server after making some changes. This is a stark contrast to the hot reloading we got out of the box from Vite, where any changes we make are applied to the frontend in the browser instantly. Let's now spend some time to improve the development experience by making the server auto-restart on changes.

## Using nodemon for easier development

To make our server auto-restart on changes, we can use the `nodemon` tool. The `nodemon` tool allows us to run our server, similarly to the node CLI command. However, it offers the possibility to auto-restart the server on changes to the source files.

1. Install the `nodemon` tool as a dev dependency:

```
$ npm install --save-dev nodemon@3.0.2
```

2. Create a new `nodemon.json` file in the root of your project and add the following contents to it:

```
3. {  
4.   "watch": ["../src", ".env", "package-lock.json"]  
   }
```

This makes sure that all code in the `src/` folder is watched for changes, and it will refresh if any files inside it are changed. Additionally, we specified the `.env` file in case environment variables are changed and the `package-lock.json` file in case packages are added or upgraded.

5. Now, edit `package.json` and define a new `"dev"` script that runs `nodemon`:

```
6.   "scripts": {  
     "dev": "nodemon src/index.js",
```

7. Stop the server (if it is currently running) and start it again by running the following command:

```
$ npm run dev
```

8. As we can see, our server is now running through `nodemon`! We can try it out by changing the port in the `.env` file:

```
9. PORT=3000  
   DATABASE_URL=mongodb://localhost:27017/blog
```

10. Edit `.env.template` as well to change the port to `3001`:

```
PORT=3000
```

11. Keep the server running.

```
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): src/**/*.env package.json
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
successfully connected to database: mongodb://localhost:27017/ch3
express server running on http://localhost:3000
[nodemon] restarting due to changes...
[nodemon] starting `node src/index.js`
successfully connected to database: mongodb://localhost:27017/ch3
express server running on http://localhost:3001
```

Figure 3.10 – Nodemon automatically restarting the server after we changed the port

After making the change, `nodemon` automatically restarted the server for us with the new port. We now have something like hot reloading, but for backend development—awesome! Now that we have improved the developer experience on the backend, let's start writing our API routes with Express. Keep the server running (via `nodemon`) to see it restart and update live while coding!

## Creating our API routes with Express

We can now start creating our previously defined API routes with express. We start by defining the GET routes:

1. Create a new `src/routes/posts.js` file and import the service functions there:

```
2. import {
3.   listAllPosts,
4.   listPostsByAuthor,
5.   listPostsByTag,
6.   getPostById,
   } from '../services/posts.js'
```

7. Now create and export a new function called `postsRoutes`, which takes the Express app as an argument:

```
export function postsRoutes(app) {
```

8. In this function, define the routes. Start with the `GET /api/v1/posts` route:

```
app.get('/api/v1/posts', async (req, res) => {
```

9. In this route, we need to make use of query params (`req.query` in Express) to map them to the arguments of our functions. We want to be able to add query params for `sortBy`, `sortOrder`, `author`, and `tag`:

```
10.     const { sortBy, sortOrder, author, tag } = req.query
        const options = { sortBy, sortOrder }
```

11. Before we call our service functions, which might throw an error if we pass invalid data to the database functions, we should add a try-catch block to handle potential errors properly:

```
    try {
```

12. We now need to check if the `author` or `tag` was provided. If both were provided, we return a `400 Bad Request` status code and a JSON object with an error message by calling `res.json()`:

```
13.         if (author && tag) {
14.             return res
15.                 .status(400)
                    .json({ error: 'query by either author or tag, not
both' })
```

16. Otherwise, we call the respective service function and return a JSON response in Express by calling `res.json()`. In case an error happened, we catch it, log it, and return a 500 status code:

```
17.         } else if (author) {
18.             return res.json(await listPostsByAuthor(author,
options))
19.         } else if (tag) {
20.             return res.json(await listPostsByTag(tag, options))
21.         } else {
22.             return res.json(await listAllPosts(options))
23.         }
24.     } catch (err) {
25.         console.error('error listing posts', err)
26.         return res.status(500).end()
27.     }
    })
```

28. Next, we define an API route to get a single post. We use the `:id` param placeholder to be able to access it as a dynamic parameter in the function:

```
app.get('/api/v1/posts/:id', async (req, res) => {
```

29. Now, we can access `req.params.id` to get the `:id` part of our route and pass it to our service function:

```
30.   const { id } = req.params
31.   try {
      const post = await getById(id)
```

32. If the result of the function is `null`, we return a 404 response because the post was not found. Otherwise, we return the post as a JSON response:

```
33.     if (post === null) return res.status(404).end()
34.     return res.json(post)
35.   } catch (err) {
36.     console.error('error getting post', err)
37.     return res.status(500).end()
38.   }
39. })
}
```

By default, Express will return the JSON response with status 200 OK.

40. After defining our GET routes, we still need to mount them in our app.

Edit `src/app.js` and import the `postsRoutes` function there:

```
import { postsRoutes } from './routes/posts.js'
```

41. Then, call the `postsRoutes(app)` function after initializing our Express app:

```
42. const app = express()
    postsRoutes(app)
```

43. Go to `http://localhost:3001/api/v1/posts` to see the route in action!

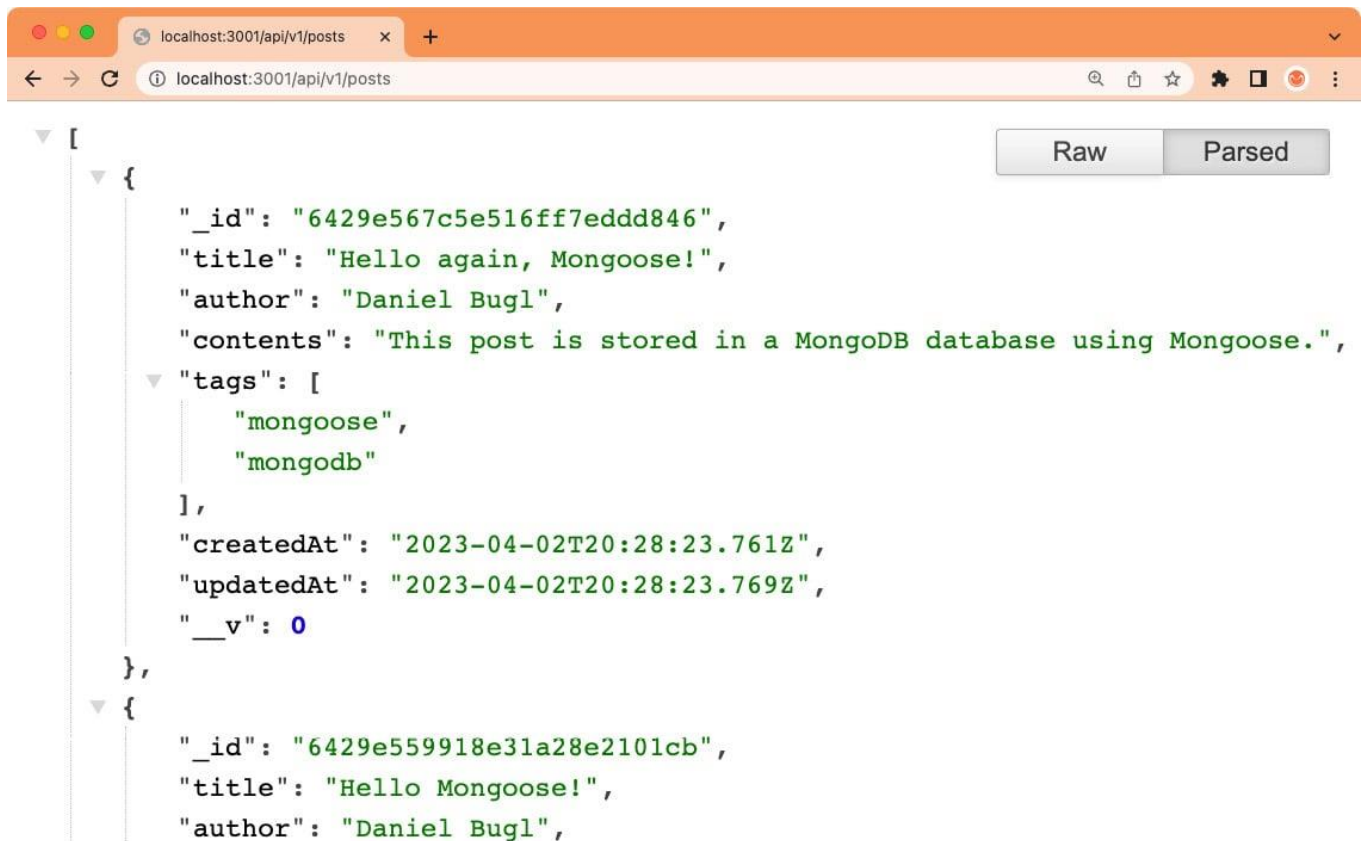


Figure 3.11 – Our first real API route in action!

### Tip

You can install a **JSON Formatter** extension in your browser to format the JSON response nicely, like in Figure 3.11.

After defining the GET routes, we need to define the POST routes. However, these accept a body, which will be formatted as JSON objects. As such, we need a way to parse this JSON body in Express.

### Defining routes with a JSON request body

To define routes with a JSON request body in Express, we need to use the **body-parser** module. This module detects if a client sent a JSON request (by looking at the **Content-Type** header) and then automatically parses it for us so that we can access the object in **req.body**.

1. Install the **body-parser** dependency:

```
$ npm install body-parser@1.20.2
```

2. Edit `src/app.js` and import the `body-parser` there:

```
import bodyParser from 'body-parser'
```

3. Now add the following code after our app is initialized to load the `body-parser` plugin as middleware into our Express app:

```
4. const app = express()
```

```
app.use(bodyParser.json())
```

#### Note

Middleware in Express allows us to do something before and after each request. In this case, `body-parser` is reading the JSON body for us, parsing it as JSON and giving us a JavaScript object that we can easily access from our route definitions. It should be noted that only routes defined after the middleware have access to it, so the order of defining middleware and routes is important!

4. After loading the `body-parser`, we edit `src/routes/posts.js` and import the service functions needed to make the rest of our routes:

```
5. createPost,  
6. updatePost,  
7. deletePost,  
} from '../services/posts.js'
```

8. Now, we define the `POST /api/v1/posts` route by using `app.post` and `req.body`, inside of the `postsRoutes` function:

```
9. app.post('/api/v1/posts', async (req, res) => {  
10.   try {  
11.     const post = await createPost(req.body)  
12.     return res.json(post)  
13.   } catch (err) {  
14.     console.error('error creating post', err)  
15.     return res.status(500).end()  
16.   }  
  })
```

17. Similarly, we can define the update route, where we need to make use of the `id` param and the request body:

```
18. app.patch('/api/v1/posts/:id', async (req, res) => {
19.   try {
20.     const post = await updatePost(req.params.id, req.body)
21.     return res.json(post)
22.   } catch (err) {
23.     console.error('error updating post', err)
24.     return res.status(500).end()
25.   }
  })
```

26. Finally, we define the delete route, which does not require the `body-parser`; we just need to get the `id` param here. We return 404 if the post was not found, and 204 No Content if the post was deleted successfully:

```
27. app.delete('/api/v1/posts/:id', async (req, res) => {
28.   try {
29.     const { deletedCount } = await
      deletePost(req.params.id)
30.     if (deletedCount === 0) return res.sendStatus(404)
31.     return res.status(204).end()
32.   } catch (err) {
33.     console.error('error deleting post', err)
34.     return res.status(500).end()
35.   }
  })
```

As we can see, Express makes defining and handling routes, requests, and responses much easier. It already detects and sets headers for us, and thus it can read and send JSON responses properly. It also allows us to change the HTTP status code easily.

Now that we finished defining the routes with a JSON request body, let's allow access to our routes from other URLs using **cross-origin resource sharing (CORS)**.

Allowing access from other URLs using CORS

Browsers have a safety feature to only allow us to access APIs on the same URL as the page we are currently on. To allow access to our backend from other URLs than the backend URL itself (for example, when we run the frontend on a different port in the next chapter), we need to allow CORS requests. Let's set that up now by using the `cors` library with Express:

1. Install the `cors` dependency:

```
$ npm install cors@2.8.5
```

2. Edit `src/app.js` and import `cors` there:

```
import cors from 'cors'
```

3. Now add the following code after our app is initialized to load the `cors` plugin as middleware into our Express app:

```
4. const app = express()
```

```
5. app.use(cors())
```

```
app.use(bodyParser.json())
```

Now that CORS requests are allowed, we can start trying out the routes in a browser!

### Trying out the routes

After defining our routes, we can try them out by using the `fetch()` function in the browser:

1. In your browser, go to `http://localhost:3001/`, open the console by right-clicking on a page and clicking **Inspect**, then go to the **Console** tab.
2. In the console, enter the following code to make a GET request to get all posts:

```
3. fetch('http://localhost:3001/api/v1/posts')
```

```
4. .then(res => res.json())
```

```
.then(console.log)
```

5. Now we can modify this code to make a POST request by specifying the `Content-Type` header to tell the server that we will be sending JSON and then sending a body with `JSON.stringify` (as the body has to be a string):

```
6. fetch('http://localhost:3001/api/v1/posts', {
7.   headers: { 'Content-Type': 'application/json' },
8.   method: 'POST',
9.   body: JSON.stringify({ title: 'Test Post' })
10. })
11.   .then(res => res.json())
   .then(console.log)
```

12. Similarly, we can also send a **PATCH** request, as follows:

```
13. fetch('http://localhost:3001/api/v1/posts/642a8b15950196ee8b3
    437b2', {
14.   headers: { 'Content-Type': 'application/json' },
15.   method: 'PATCH',
16.   body: JSON.stringify({ title: 'Test Post Changed' })
17. })
18.   .then(res => res.json())
   .then(console.log)
```

Make sure to replace the MongoDB IDs in the URL with the one returned from the **POST** request made before!

19. Finally, we can send a **DELETE** request:

```
20. fetch('http://localhost:3001/api/v1/posts/642a8b15950196ee8b3
    437b2', {
21.   method: 'DELETE',
22. })
23.   .then(res => res.status)
   .then(console.log)
```

24. When doing a GET request, we can see that our post has now been deleted again:

```
25. fetch('http://localhost:3001/api/v1/posts/642a8b15950196ee8b3
    437b2')
26.   .then(res => res.status)
   .then(console.log)
```

This request should now return a **404**.

Tip

Instead of the browser console, you can also use command line tools such as `curl` or apps such as Postman to make the requests. Feel free to use different tools to try out the requests if you are already familiar with them.

We have now successfully defined all routes needed to handle a simple blog post API!

## Summary

The first version of our backend service is now complete, allowing us to create, read, update, and delete blog posts via a REST API (using Express), which then get stored in a MongoDB database (using Mongoose). Additionally, we have created service functions with unit tests, defined using the Jest test suite. All in all, we managed to create a solid foundation for our backend in this chapter.

# Integrating a Frontend Using React and TanStack Query

After designing, implementing, and testing our backend service, it's now time to create a frontend to interface with the backend. First, we will start by setting up a full-stack React project based on the Vite boilerplate and the backend service created in the previous chapters. Then, we are going to create a basic user interface for our blog application. Finally, we will use TanStack Query, a data fetching library to handle backend state, to integrate the backend API into the frontend. By the end of this chapter, we will have successfully developed our first full-stack application!

In this chapter, we are going to cover the following main topics:

- Principles of React
- Setting up a full-stack React project
- Creating the user interface for our application
- Integrating the backend service using TanStack Query

## Principles of React

Before we start learning how to set up a full-stack React project, let's revisit the three fundamental principles of React. These principles allow us to easily write scalable web applications:

- **Declarative:** Instead of telling React how to do things, we tell it what we want it to do. As a result, we can easily design our applications and React will efficiently update and render just the right components when the data changes. For example, the following code, which duplicates strings in an array is imperative, which is the opposite of declarative:
  - `const input = ['a', 'b', 'c']`
  - `let result = []`
  - `for (let i = 0; i < input.length; i++) {`

- `result.push(input[i] + input[i])`
- `}`

```
console.log(result) // prints: [ 'aa', 'bb', 'cc' ]
```

As we can see, in imperative code, we need to tell JavaScript exactly what to do, step by step. However, with declarative code, we can simply tell the computer what we want, as follows:

```
const input = ['a', 'b', 'c']  
const result = input.map(str => str + str)  
console.log(result) // prints: ['aa', 'bb', 'cc']
```

In this declarative code, we tell the computer that we want to map each element of the `input` array from `str` to `str + str`. As you can see, declarative code is much more concise.

- **Component-based:** React encapsulates components that manage their own state and views and then allows us to compose them in order to create complex user interfaces.
- **Learn once, write anywhere:** React does not make assumptions about your technology stack and tries to ensure that you can develop apps without rewriting existing code as much as possible.

React's three fundamental principles make it easy to write code, encapsulate components, and share code across multiple platforms. Instead of reinventing the wheel, React tries to make use of existing JavaScript features as much as possible. As a result, we will learn software design patterns that will be applicable in many more cases than just designing user interfaces.

Now that we have learned the fundamental principles of React, let's get started setting up a full-stack React project!

## Creating the user interface for our application

When designing the structure of a frontend, we should also consider the folder structure, so that our app can grow easily in the future. Similar to how we did for the backend, we will also put all our source code into a `src/` folder. We can then group the files in separate folders for the different features. Another popular way to structure frontend projects is to group code by routes. Of course, it is also possible to mix them, for example, in Next.js projects we can group our components by features and then create another folder and file structure for the routes, where the components are used. For full-stack projects, it additionally makes sense to first separate our code by creating separate folders for the API integration and UI components.

Now, let's define the folder structure for our project:

1. Create a new `src/api/` folder.
2. Create a new `src/components/` folder.

#### Tip

It is a good idea to start with a simple structure at first, and only nest more deeply when you actually need it. Do not spend too much time thinking about the file structure when starting a project, because usually, you do not know upfront how files should be grouped, and it may change later anyway.

After defining the high-level folder structure for our projects, let's now take some time to consider the component structure.

## Component structure

Based on what we defined in the backend, our blog application is going to have the following features:

- Viewing a single post
- Creating a new post
- Listing posts
- Filtering posts

- Sorting posts

The idea of components in React is to have each component deal with a single task or UI element. We should try to make components as fine-grained as possible, in order to be able to reuse code. If we find ourselves copying and pasting code from one component to another, it might be a good idea to create a new component and reuse it in multiple other components.

Usually, when developing a frontend, we start with a UI mock-up. For our blog application, a mock-up could look as follows:

Title:

Author:

Create

---

author:

Sort By:  / Sort Order:

---

## Full-Stack React Projects

Let's become full-stack developers!

*Written by Daniel Bugl*

---

## Hello React!

---

Figure 4.1 – An initial mock-up of our blog application

Note

In this book, we will not cover UI or CSS frameworks. As such, the components are designed and developed without styling. Instead, the book focuses on the full-stack aspect of the integration of backends with frontends. Feel free to use a UI framework (such as MUI), or a CSS framework (such as Tailwind) to style the blog application on your own.

When splitting up the UI into components, we use the **single-responsibility principle**, which states that every module should have responsibility over a single encapsulated part of the functionality.

In our mock-up, we can draw boxes around each component and subcomponent, and give them names. Keep in mind that each component should have exactly one responsibility. We start with the fundamental components that make up the app:

Title:

Author:

Create

CreatePost

Filter by:  
author:

PostFilter

Sort By:  / Sort Order:

PostSort

## Full-Stack React Projects

Let's become full-stack developers!

*Written by Daniel Bugl*

Post

## Hello React!

Post

Figure 4.2 – Defining the fundamental components in our mock-up

We defined a `CreatePost` component, with a form to create a new post, a `PostFilter` component to filter the list of posts, a `PostSorting` component to sort posts, and a `Post` component to display a single post.

Now that we have defined our fundamental components, we are going to look at which components logically belong together, thereby forming a group: we can group the `Post` components together in `PostList`, then make an `App` component to group everything together and define the structure of our app.

Now that we are done with structuring our React components, we can move on to implementing the static React components.

## Implementing static React components

Before integrating with the backend, we are going to model the basic features of our application as static React components. Dealing with the static view structure of our application first makes sense, as we can play around and re-structure the application UI if needed, before adding integration to the components, which would make it harder and more tedious to move them around. It is also easier to deal only with the UI first, which helps us to get started quickly with projects and features. Then, we can move on to implementing integrations and handling state.

Let's get started implementing the static components now.

### The Post component

We have already thought about which elements a post has during the creation of the mock-up and the design of the backend. A post should have a `title`, `contents`, and an `author`.

Let's implement the `Post` component now:

1. First, create a new `src/components/Post.jsx` file.
2. In that file, import `PropTypes`:

```
import PropTypes from 'prop-types'
```

3. Define a function component, accepting `title`, `contents`, and `author` props:

```
export function Post({ title, contents, author }) {
```

4. Next, render all props in a way that resembles the mock-up:

```
5.   return (  
6.     <article>  
7.       <h3>{title}</h3>  
8.       <div>{contents}</div>  
9.       {author && (  
10.        <em>  
11.          <br />  
12.          Written by <strong>{author}</strong>  
13.        </em>  
14.      )}  
15.     </article>  
16.   )  
}
```

#### Tip

Please note that you should always prefer spacing via CSS, rather than using the `<br />` HTML tag. However, we are focusing on the UI structure and integration with the backend in this book, so we simply use HTML whenever possible.

5. Now, define `propTypes`, making sure only `title` is required:

```
6. Post.propTypes = {  
7.   title: PropTypes.string.isRequired,  
8.   contents: PropTypes.string,  
9.   author: PropTypes.string,  
}
```

#### Info

`PropTypes` are used to validate the props passed to React components and to ensure that we are passing the correct props when using JavaScript. When using a type-safe language, such as TypeScript, we can instead do this by directly typing the props passed to the component.

6. Let's test out our component by replacing the `src/App.jsx` file with the following contents:

```
7. import { Post } from './components/Post.jsx'
8. export function App() {
9.   return (
10.     <Post
11.       title='Full-Stack React Projects'
12.       contents="Let's become full-stack developers!"
13.       author='Daniel Bugl'
14.     />
15.   )
  }
```

16. Edit `src/main.jsx` and update the import of the `App` component, because we are now not using `export default` anymore:

```
import { App } from './App.jsx'
```

## Info

I personally tend to prefer not using default exports, as they make it harder to re-group and re-export components and functions from other files. Also, they allow us to change the names of the components, which could be confusing. For example, if we change the name of a component, the name when importing it is not changed automatically.

8. Also, remove the following line from `src/main.jsx`:

```
import './index.css'
```

9. Finally, we can delete the `index.css` and `App.css` files, as they are not needed anymore.

Now that our static `Post` component has been implemented, we can move on to the `CreatePost` component.

## The CreatePost component

We'll now implement a form to allow for the creation of new posts. Here, we provide fields for `author` and `title` and a `<textarea>` element for the contents of the blog post.

Let's implement the `CreatePost` component now:

1. Create a new `src/components/CreatePost.jsx` file.
  2. Define the following component, which contains a form to enter the title, author, and contents of a blog post:
- ```
3. export function CreatePost() {  
4.   return (  
5.     <form onSubmit={(e) => e.preventDefault()}>  
6.       <div>  
7.         <label htmlFor='create-title'>Title: </label>  
8.         <input type='text' name='create-title' id='create-  
title' />  
9.       </div>  
10.      <br />  
11.      <div>  
12.        <label htmlFor='create-author'>Author: </label>  
13.        <input type='text' name='create-author' id='create-  
author' />  
14.      </div>  
15.      <br />  
16.      <textarea />  
17.      <br />  
18.      <br />  
19.      <input type='submit' value='Create' />  
20.    </form>  
21.  )  
}
```
- [Copy](#)[Explain](#)

In the preceding code block, we defined an `onSubmit` handler and called `e.preventDefault()` on the event object to avoid a page refresh when the form is submitted.

22. Let's test the component out by replacing the `src/App.jsx` file with the following contents:

```
23. import { CreatePost } from './components/CreatePost.jsx'
24. export function App() {
25.   return <CreatePost />
    }
```

As you can see, the `CreatePost` component renders fine. We can now move on to the `PostFilter` and `PostSorting` components.

### Tip

If you want to test out multiple components at once and keep the tests around for later, or build a style guide for your own component library, you should look into Storybook (<https://storybook.js.org>), which is a useful tool to build, test, and document UI components in isolation.

### The PostFilter and PostSorting components

Similar to the `CreatePost` component, we will be creating two components that provide input fields to filter and sort posts. Let's start with `PostFilter`:

1. Create a new `src/components/PostFilter.jsx` file.
2. In this file, we import `PropTypes`:

```
import PropTypes from 'prop-types'
```

[Copy](#)[Explain](#)

3. Now, we define the `PostFilter` component and make use of the `field` prop:

```
4. export function PostFilter({ field }) {
5.   return (
6.     <div>
7.       <label htmlFor={`filter-${field}`}>{field}: </label>
8.       <input
9.         type='text'
10.        name={`filter-${field}`}
11.        id={`filter-${field}`}
12.      />
13.     </div>
14.   )
15. }
```

```
16. PostFilter.propTypes = {
17.   field: PropTypes.string.isRequired,
   }
```

Next, we are going to define the `PostSorting` component.

18. Create a new `src/components/PostSorting.jsx` file.

19. In this file, we create a `select` input to select which field to sort by. We also create another `select` input to select the sort order:

```
20. import PropTypes from 'prop-types'
21. export function PostSorting({ fields = [] }) {
22.   return (
23.     <div>
24.       <label htmlFor='sortBy'>Sort By: </label>
25.       <select name='sortBy' id='sortBy'>
26.         {fields.map((field) => (
27.           <option key={field} value={field}>
28.             {field}
29.           </option>
30.         ))}
31.       </select>
32.       {' / '}
33.       <label htmlFor='sortOrder'>Sort Order: </label>
34.       <select name='sortOrder' id='sortOrder'>
35.         <option value='ascending'>ascending</option>
36.         <option value='descending'>descending</option>
37.       </select>
38.     </div>
39.   )
40. }
41. PostSorting.propTypes = {
42.   fields: PropTypes.arrayOf(PropTypes.string).isRequired,
   }
```

Now we have successfully defined UI components to filter and sort posts. In the next step, we are going to create a `PostList` component to combine the filter and sorting with a list of posts.

### The PostList component

After implementing the other post-related components, we can now implement the most important part of our blog app, that is, the feed of blog posts. For now, the feed is simply going to show a list of blog posts.

Let's start implementing the `PostList` component now:

1. Create a new `src/components/PostList.jsx` file.
2. First, we import `Fragment`, `PropTypes`, and the `Post` component:
3. `import { Fragment } from 'react'`
4. `import PropTypes from 'prop-types'`  
`import { Post } from './Post.jsx'`
5. Then, we define the `PostList` function component, accepting a `posts` array as a prop. If `posts` is not defined, we set it to an empty array, by default:

```
export function PostList({ posts = [] }) {
```

6. Next, we render all posts by using the `.map` function and the spread syntax:

```
7.   return (  
8.     <div>  
9.       {posts.map((post) => (  
10.        <Post {...post} key={post._id} />  
11.      )}  
12.     </div>  
13.   )  
  }
```

We return the `<Post>` component for each post, and pass all the keys from the `post` object to the component as props. We do this by using the spread syntax, which has the same effect as listing all the keys from the object manually as props, like so:

```
<Post  
  title={post.title}  
  author={post.author}  
  contents={post.contents}  
/>
```

Note

If we are rendering a list of elements, we have to give each element a unique `key` prop. React uses this `key` prop to efficiently compute the difference between two lists when the data has changed.

We used the `map` function, which applies a function to all the elements of an array. This is similar to using a `for` loop and storing all the results, but it is more concise, declarative, and easier to read! Alternatively, we could do the following instead of using the `map` function:

```
let renderedPosts = []
let index = 0
for (let post of posts) {
  renderedPosts.push(<Post {...post} key={post._id} />)
  index++
}
return (
  <div>
    {renderedPosts}
  </div>
)
```

However, using this style is not recommended with React.

5. We also still need to define the prop types. Here, we can make use of the prop types from the `Post` component, by wrapping it inside the `PropTypes.shape()` function, which defines an object prop type:

```
6. PostList.propTypes = {
7.   posts:
  PropTypes.arrayOf(PropTypes.shape(Post.propTypes)).isRequired,
}
```

8. In the mock-up, we have a horizontal line after each blog post. We can implement this without an additional `<div>` container element, by using `Fragment`, as follows:

```
9.     {posts.map((post) => (
10.      <Fragment key={post._id}>
11.        <Post {...post} />
12.        <hr />
13.      </Fragment>

```

```
)))}
```

## Note

The `key` prop always has to be added to the uppermost parent element that is rendered within the `map` function. In this case, we had to move the `key` prop from the `Post` component to `Fragment`.

7. Again, we test our component by editing the `src/App.jsx` file:

```
8. import { PostList } from './components/PostList.jsx'
9. const posts = [
10.   {
11.     title: 'Full-Stack React Projects',
12.     contents: "Let's become full-stack developers!",
13.     author: 'Daniel Bugl',
14.   },
15.   { title: 'Hello React!' },
16. ]
17. export function App() {
18.   return <PostList posts={posts} />
19. }
```

Now we can see that our app lists all the posts that we defined in the `posts` array.

As you can see, listing multiple posts via the `PostList` component works fine. We can now move on to putting the app together.

## Putting the app together

After implementing all the components, we now have to put everything together in the `App` component. Then, we will have successfully reproduced the mock-up!

Let's start modifying the `App` component and putting our blog app together:

1. Open `src/App.jsx` and add imports for the `CreatePost`, `PostFilter`, and `PostSorting` components:

```
2. import { PostList } from './components/PostList.jsx'
3. import { CreatePost } from './components/CreatePost.jsx'
```

```
4. import { PostFilter } from './components/PostFilter.jsx'
   import { PostSorting } from './components/PostSorting.jsx'
```

5. Adjust the **App** component to contain all the components:

```
6. export function App() {
7.   return (
8.     <div style={{ padding: 8 }}>
9.       <CreatePost />
10.      <br />
11.      <hr />
12.      Filter by:
13.      <PostFilter field='author' />
14.      <br />
15.      <PostSorting fields={['createdAt', 'updatedAt']} />
16.      <hr />
17.      <PostList posts={posts} />
18.    </div>
19.  )
}
```

[Copy](#)[Explain](#)

20. After saving the file, the browser should automatically refresh, and we can now see the full UI:



Title:

Author:

Create

Filter by:  
author:

Sort By:  / Sort Order:

---

## Full-Stack React Projects

Let's become full-stack developers!

*Written by Daniel Bugl*

---

## Hello React!

---

Figure 4.3 – Full implementation of our static blog app, according to the mock-up

As we can see, all of the static components that we defined earlier are rendered together in one `App` component. Our app now looks just like a mock-up. Next, we can move on to integrating our components with the backend service.

## Integrating the backend service using TanStack Query

After finishing creating all the UI components, we can now move on to integrating them with the backend we created in the previous chapter. For the integration, we

are going to use TanStack Query (previously called React Query), which is a data fetching library that can also help us with caching, synchronizing, and updating data from a backend.

TanStack Query specifically focuses on managing the state of fetched data (server state). While other state management libraries can also deal with server state, they specialize in managing client state instead. Server state has some stark differences from client state, such as the following:

- Being persisted remotely in a location the client does not control directly
- Requiring asynchronous APIs to fetch and update state
- Having to deal with shared ownership, which means that other people can change the state without your knowledge
- State becoming stale (“out of date”) at some point when changed by the server or other people

These challenges with server state result in issues such as having to cache, deduplicate multiple requests, update “out of date” state in the background, and so on.

TanStack Query provides solutions to these issues out of the box and thus makes dealing with server state simple. You can always combine it with other state management libraries that focus on client state as well. For use cases where the client state essentially just reflects the server state though, TanStack Query on its own can be good enough as a state management solution!

#### Note

The reason why React Query got renamed to TanStack Query is that the library now also supports other frameworks, such as Solid, Vue, and Svelte!

Now that you know why and how TanStack Query can help us integrate our frontend with the backend, let's get started using it!

## Setting up TanStack Query for React

To set up TanStack Query, we first have to install the dependency and set up a query client. The query client is provided to React through a context and will store information about active requests, cached results, when to periodically re-fetch data, and everything needed for TanStack Query to function.

Let's get started setting it up now:

1. Open a new Terminal (do not quit Vite!) and install the `@tanstack/react-query` dependency by running the following command in the root of our project:

```
$ npm install @tanstack/react-query@5.12.2
```

[Copy](#)[Explain](#)

We are now going to move our current `App` component to a new `Blog` component, as we are going to use the `App` component for setting up libraries and contexts instead.

2. Rename the `src/App.jsx` file to `src/Blog.jsx`.

Do not update imports yet. If VS Code asks you to update imports, click **No**.

3. Now, in `src/Blog.jsx`, change the function name from `App` to `Blog`:

```
export function Blog() {
```

[Copy](#)[Explain](#)

4. Create a new `src/App.jsx` file. In this file, import `QueryClient` and `QueryClientProvider` from TanStack React Query:

```
import { QueryClient, QueryClientProvider } from  
'@tanstack/react-query'
```

[Copy](#)[Explain](#)

5. Also, import the `Blog` component:

```
import { Blog } from './Blog.jsx'
```

6. Now, create a new query client:

```
const queryClient = new QueryClient()
```

7. Define the `App` component and render the `Blog` component wrapped inside `QueryClientProvider`:

```
8. export function App() {  
9.   return (  
10.     <QueryClientProvider client={queryClient}>  
11.       <Blog />  
12.     </QueryClientProvider>  
13.   )  
  }
```

That's all there is to setting up TanStack Query! We can now make use of it inside our **Blog** component (and its children).

## Fetching blog posts

The first thing we should do is fetch the list of blog posts from our backend. Let's implement that now:

1. First of all, in the second Terminal window opened (not where Vite is running), run the backend server (do not quit Vite!), as follows:

```
2. $ cd backend/  
   $ npm start
```

If you get an error, make sure Docker and MongoDB are running properly!

### Tip

If you want to develop the backend and frontend at the same time, you can start the backend using **npm run dev** to make sure it hot reloads when you change the code.

2. Create a **.env** file in the root of the project, and enter the following contents into it:

```
VITE_BACKEND_URL="http://localhost:3001/api/v1"
```

Vite supports **dotenv** out of the box. All environment variables that should be available to be accessed within the frontend need to be prefixed with **VITE\_**. Here, we set an environment variable to point to our backend server.

3. Create a new **src/api/posts.js** file. In this file, we are going to define a function to fetch posts, which accepts the query params for

the `/posts` endpoint as an argument. These query params are used to filter by author and tag and define sorting using `sortBy` and `sortOrder`:

```
export const getPosts = async (queryParams) => {
```

- Remember that we can use the `fetch` function to make a request to a server. We need to pass the environment variable to it and add the `/posts` endpoint. After the path, we add query params, which are prefixed with the `?` symbol:

```
5.   const res = await fetch(  
      `${import.meta.env.VITE_BACKEND_URL}/posts?` +
```

- Now we need to use the `URLSearchParams` class to turn an object into query params. That class will automatically escape the input for us and turn it into valid query params:

```
      new URLSearchParams(queryParams),
```

- Like we did before in the browser, we need to parse the response as JSON:

```
8.   )  
9.   return await res.json()  
   }
```

- Edit `src/Blog.jsx` and remove the sample `posts` array:

```
11.  const posts = [  
12.    {  
13.      title: 'Full-Stack React Projects',  
14.      contents: "Let's become full-stack developers!",  
15.      author: 'Daniel Bugl',  
16.    },  
17.    { title: 'Hello React!' },  
  ]
```

- Also, import the `useQuery` function from `@tanstack/react-query` and the `getPosts` function from our `api` folder in the `src/Blog.jsx` file:

```
19.  import { useQuery } from '@tanstack/react-query'  
20.  import { PostList } from './components/PostList.jsx'  
21.  import { CreatePost } from './components/CreatePost.jsx'  
22.  import { PostFilter } from './components/PostFilter.jsx'  
23.  import { PostSorting } from './components/PostSorting.jsx'  
    import { getPosts } from './api/posts.js'
```

24. Inside the `Blog` component, define a `useQuery` hook:

```
25. export function Blog() {  
26.   const postsQuery = useQuery({  
27.     queryKey: ['posts'],  
28.     queryFn: () => getPosts(),  
    })
```

The `queryKey` is very important in TanStack Query, as it is used to uniquely identify a request, among other things, for caching purposes. Always make sure to use unique query keys. Otherwise, you might see requests not triggering properly.

For the `queryFn` option, we just call the `getPosts` function, without query params for now.

29. After the `useQuery` hook, we get the posts from our query and fall back to an empty array if the posts are not loaded yet:

```
const posts = postsQuery.data ?? []
```

30. Check your browser, and you will see that the posts are now loaded from our backend!

Now that we have successfully fetched blog posts, let's get the filters and sorting working!

## Implementing filters and sorting

To implement filters and sorting, we need to handle some local state and pass it as query params to `postsQuery`. Let's do that now:

1. We start by editing the `src/Blog.jsx` file and importing the `useState` hook from React:

```
import { useState } from 'react'
```

2. Then we add state hooks for the `author` filter and the sorting options inside the `Blog` component, before the `useQuery` hook:

```
3.   const [author, setAuthor] = useState('')  
4.   const [sortBy, setSortBy] = useState('createdAt')
```

```
const [sortOrder, setSortOrder] = useState('descending')
```

- Then, we adjust `queryKey` to contain the query params (so that whenever a query param changes, TanStack Query will re-fetch unless the request is already cached). We also adjust `queryFn` to call `getPosts` with the relevant query params:

```
6. const postsQuery = useQuery({
7.   queryKey: ['posts', { author, sortBy, sortOrder }],
8.   queryFn: () => getPosts({ author, sortBy, sortOrder }),
  })
```

- Now pass the values and relevant `onChange` handlers to the filter and sorting components:

```
10. <PostFilter
11.   field='author'
12.   value={author}
13.   onChange={(value) => setAuthor(value)}
14. />
15. <br />
16. <PostSorting
17.   fields={['createdAt', 'updatedAt']}
18.   value={sortBy}
19.   onChange={(value) => setSortBy(value)}
20.   orderValue={sortOrder}
21.   onOrderChange={(orderValue) => setSortOrder(orderValue)}
  />
```

#### Note

For simplicity's sake, we are only using state hooks for now. A state management solution or context could make dealing with filters and sorting much easier, especially for larger applications. For our small blog application, it is fine to use state hooks though, as we are focusing mostly on the integration of the backend and frontend.

- Now, edit `src/components/PostFilter.jsx` and add the `value` and `onChange` props:

```
6. export function PostFilter({ field, value, onChange }) {
```

```
7.   return (
8.     <div>
9.       <label htmlFor={`filter-${field}`}>{field}: </label>
10.      <input
11.        type='text'
12.        name={`filter-${field}`}
13.        id={`filter-${field}`}
14.        value={value}
15.        onChange={(e) => onChange(e.target.value)}
16.      />
17.    </div>
18.  )
19. }
20. PostFilter.propTypes = {
21.   field: PropTypes.string.isRequired,
22.   value: PropTypes.string.isRequired,
23.   onChange: PropTypes.func.isRequired,
24. }
```

24. We also do the same for `src/components/PostSorting.jsx`:

```
25. export function PostSorting({
26.   fields = [],
27.   value,
28.   onChange,
29.   orderValue,
30.   onOrderChange,
31. }) {
32.   return (
33.     <div>
34.       <label htmlFor='sortBy'>Sort By: </label>
35.       <select
36.         name='sortBy'
37.         id='sortBy'
38.         value={value}
39.         onChange={(e) => onChange(e.target.value)}
40.       >
41.         {fields.map((field) => (
42.           <option key={field} value={field}>
43.             {field}
```

```

44.         </option>
45.     )})
46. </select>
47. {' / '}
48. <label htmlFor='sortOrder'>Sort Order: </label>
49. <select
50.     name='sortOrder'
51.     id='sortOrder'
52.     value={orderValue}
53.     onChange={(e) => onOrderChange(e.target.value)}
54. >
55.     <option value={'ascending'}>ascending</option>
56.     <option value={'descending'}>descending</option>
57. </select>
58. </div>
59. )
60. }
61. PostSorting.propTypes = {
62.   fields: PropTypes.arrayOf(PropTypes.string).isRequired,
63.   value: PropTypes.string.isRequired,
64.   onChange: PropTypes.func.isRequired,
65.   orderValue: PropTypes.string.isRequired,
66.   onOrderChange: PropTypes.func.isRequired,
  }

```

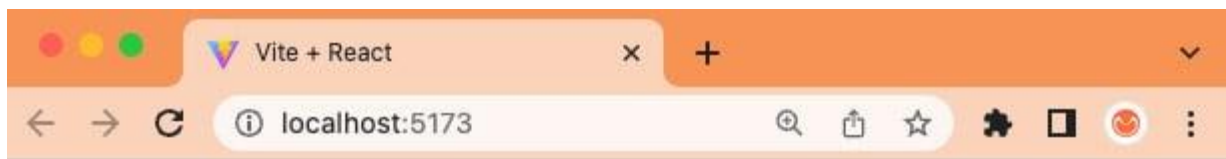
67. In your browser, enter **Daniel Bugl** as the author. You should see TanStack Query re-fetch the posts from the backend as you type, and once a match is found, the backend will return all posts by that author!

68. After testing it out, make sure to clear the filter again, so that newly created posts are not filtered by the author anymore later on.

### Tip

If you do not want to make that many requests to the backend, make sure to use a debouncing state hook, such as **useDebounce**, and then pass only the debounced value to the query param. If you are interested in gaining further knowledge about the **useDebounce** hook and other useful hooks, I recommend checking out my book titled **Learn React Hooks**.

The application should now look as follows, with the posts being filtered by the author entered in the field, and sorted by the selected field, in the selected order:



Title:

Author:

Create

---

Filter by:

author:

Sort By:  / Sort Order:

---

## Hello Mongoose!

*Written by Daniel Bugl*

---

## Hello again, Mongoose!

*Written by Daniel Bugl*

---

Figure 4.4 – Our first full-stack application – a frontend fetching posts from a backend!

Now that sorting and filtering are working properly, let's learn about mutations, which allow us to make requests to the server that change the state of the backend (for example, inserting or updating entries in the database).

## Creating new posts

We are now going to implement a feature to create posts. To do this, we need to use the `useMutation` hook from TanStack Query. While queries are meant to be idempotent (meaning that calling them multiple times should not affect the result), mutations are used to create/update/delete data or perform operations on the server. Let's get started using mutations to create new posts now:

1. Edit `src/api/posts.js` and define a new `createPost` function, which accepts a `post` object as an argument:

```
export const createPost = async (post) => {
```

2. We also make a request to the `/posts` endpoint, like we did for `getPosts`:

```
  const res = await  
  fetch(`${import.meta.env.VITE_BACKEND_URL}/posts`, {
```

3. However, now we also set `method` to a `POST` request, pass a header to tell the backend that we will be sending a JSON body, and then send our `post` object as a JSON string:

```
4.    method: 'POST',  
5.    headers: { 'Content-Type': 'application/json' },  
      body: JSON.stringify(post),
```

6. Like with `getPosts`, we also need to parse the response as JSON:

```
7.  })  
8.  return await res.json()  
  }
```

After defining the `createPost` API function, let's use it in the `CreatePost` component by creating a new mutation hook there.

9. Edit `src/components/CreatePost.jsx` and import the `useMutation` hook from `@tanstack/react-query`, the `useState` hook from `React`, and our `createPost` API function:

```
10. import { useMutation } from '@tanstack/react-query'
11. import { useState } from 'react'
    import { createPost } from '../api/posts.js'
```

12. Inside the `CreatePost` component, define state hooks for `title`, `author`, and `contents`:

```
13.   const [title, setTitle] = useState('')
14.   const [author, setAuthor] = useState('')
    const [contents, setContents] = useState('')
```

15. Now, define a mutation hook. Here, we are going to call our `createPost` function:

```
16.   const createPostMutation = useMutation({
17.     mutationFn: () => createPost({ title, author, contents
    }),
    })
```

18. Next, we are going to define a `handleSubmit` function, which will prevent the default submit action (which refreshes the page), and instead call `.mutate()` to execute the mutation:

```
19.   const handleSubmit = (e) => {
20.     e.preventDefault()
21.     createPostMutation.mutate()
    }
```

22. We add the `onSubmit` handler to our form:

```
    <form onSubmit={handleSubmit}>
CopyExplain
```

23. We also add the `value` and `onChange` props to our fields, as we did before for the sorting and filters:

```
24.     <div>
25.       <label htmlFor='create-title'>Title: </label>
26.       <input
27.         type='text'
```

```

28.         name='create-title'
29.         id='create-title'
30.     value={title}
31.     onChange={(e) => setTitle(e.target.value)}
32.     />
33. </div>
34. <br />
35. <div>
36.     <label htmlFor='create-author'>Author: </label>
37.     <input
38.         type='text'
39.         name='create-author'
40.         id='create-author'
41.         value={author}
42.         onChange={(e) => setAuthor(e.target.value)}
43.     />
44. </div>
45. <br />
46. <textarea
47.     value={contents}
48.     onChange={(e) => setContents(e.target.value)}
    />

```

49. For the submit button, we make sure it says **Creating...** instead of **Create** while we are waiting for the mutation to finish, and we also disable the button if no title was set (as it is required), or if the mutation is currently pending:

```

50.     <br />
51.     <br />
52.     <input
53.         type='submit'
54.         value={createPostMutation.isPending ? 'Creating...' : 'Create'}
55.         disabled={!title || createPostMutation.isPending}
    />

```

56. Lastly, we add a message below the submit button, which will be shown if the mutation is successful:

```

57.     {createPostMutation.isSuccess ? (
58.         <>
59.         <br />

```

```
60.         Post created successfully!
61.     </>
62.     ) : null}
    </form>
```

## Note

In addition to `isPending` and `isSuccess`, mutations also return `isIdle` (when the mutation is idle or in a fresh/reset state) and `isError` states. The same states can also be accessed from queries, for example, to show a loading animation while posts are fetching.

13. Now we can try adding a new post, and it seems to work fine, but the post list is not updating automatically, only after a refresh!

The issue is that the query key did not change, so TanStack Query does not refresh the list of posts. However, we also want to refresh the list when a new post is created. Let's fix that now.

## Invalidating queries

To ensure that the post list is refreshed after creating a new post, we need to invalidate the query. We can make use of the query client to do this. Let's do it now:

1. Edit `src/components/CreatePost.jsx` and import the `useQueryClient` hook:

```
import { useMutation, useQueryClient } from '@tanstack/react-query'
```

2. Use the query client to invalidate all queries starting with the `'posts'` query key. This will work with any query params to the `getPosts` request, as it matches all queries starting with `'posts'` in the array:

```
3. const queryClient = useQueryClient()
4. const createPostMutation = useMutation({
5.   mutationFn: () => createPost({ title, author, contents }),
6.   onSuccess: () => queryClient.invalidateQueries(['posts']),
   })
```

Try creating a new post, and you will see that it works now, even with active filters and sorting! As we can see, TanStack Query is great for handling server state with ease.

## Summary

In this chapter, we learned how to create a React frontend and integrate it with our backend using TanStack Query. We have covered the main functionality of our backend: listing posts with sorting, creating posts, and filtering by author. Dealing with tags and deleting and editing posts are similar to the already explained functionalities and are left as an exercise for you.