7

Authentication and Authorization

The concept of authentication (proving that the user is who they claim to be) and authorization (making sure that the authenticated user should or should not be able to perform certain operations on our API) is very complex, and several good (and thick) books are dedicated to it. In this chapter, we will explore the topics of authentication and authorization from a very practical standpoint and from our FARM-stack perspective.

We will begin with a very simple yet robust and extensible setup for our FastAPI backend, based on JWT – arguably the most popular authentication method in the last couple of years. Then, we will see how to integrate our JWT-based authentication methods into React, leveraging some of React's coolest and newest features – namely Hooks, Context, and React Router.

The topics we will cover in this chapter are the following:

- The user model and how it relates to our other resources
- A brief overview of the JWT authentication mechanism – the big picture
- An overview of the authentication and authorization tools that FastAPI has to offer
- How to protect our routes, routers, or the entire app
- Various solutions for authenticating with React

By the end of this chapter, you should have a solid grasp of authentication methods that both FastAPI on the backend and React on the frontend have to offer, and you will be able to authenticate users and control what they can and cannot do within your application with granularity and precision.

Technical requirements

To run the sample application in this chapter, you should have both Node.js and Python installed on your local computer. The requirements are identical to those in the previous chapter, and the new packages that we will install will be pointed out. You should have a newer Python installation (version 3.6 or newer) and Node.js installation (version 14 or more). Your computer should be able to handle a couple of Visual Studio Code instances open at the same time, along with a couple of terminals and maybe a dozen browser tabs.

Understanding JSON Web Token – our key to security

HTTP is a stateless protocol, and that fact alone implies several important consequences. One of them is that if we want to persist some kind of **state** between requests, we must resort to a mechanism that will be able to **remember** who the logged-in user was, what the selected cars during a previous browser session were, or what the site preferences were.

Broadly speaking, there are many strategies that we can employ when performing authentication. Credential-based authentication requires the user to enter some personal credentials, usually a username or an email and a password. A new method that has gained some traction over the last years is the concept of a passwordless login – once the user creates an account, they are emailed a magic link that is used for authenticating a session, without the need to type (and remember!) passwords. Biometric passwords use some bio-feature of the user, such as a fingerprint, while social authentications use the user's account on social networks (Google, Facebook, or LinkedIn) to associate the user with their account. In this chapter, we will consider a classic personal credentials method – when a user registers, they get to provide an email and choose a password and, optionally, a username.

While there are different ways of maintaining the identity of a user across different parts of an app, **JSON Web Token** (**JWT**) is arguably the most common and popular method of connecting frontend applications (React, Vue.js, and Angular) or mobile apps with an API (in our case, a REST API). JWT is nothing but a standard – a way of structuring a big string composed of seemingly random characters and numbers.

JWT contains three parts – the header, the payload, and the signature. The header hosts metadata about the token itself: the algorithm used for signing the token and the type of the token.

The payload is the most interesting part. It contains the data (claims): the ID of the user (or the username) and the **Issued at** (**iat**) field, the date and time of issuing the token, the expiry (the time at which the token ceases to be valid), and optionally, other fields. The *payload* is decodable and readable by everyone. There is a very useful site – `https://jwt.io` – that enables us to play with tokens and see how they look.

Finally, probably the most important part of the token is the signature – the part of the token that guarantees the claims made by the token, so to speak. The signature is reproduced (calculated) and compared with the original, thus preventing the modification of the claims. Put simply, if a JWT token which can be easily "read," claims that the username is John, we could tamper with it and modify the username to be Rita, but by doing so, we would alter the signature, which wouldn't match anymore, rendering the said token invalid. It is really a simple yet ingenious mechanism if you think about it.

The token is hence able to completely replace the authentication data – user or email and password combinations that do not need to go flying over the wire more than once.

In this section, we have learned what JWT is, what the logic behind it is, and why you might want to resort to it for your authentication and authorization system. In the forthcoming sections, we will address how to implement a JSON Web Token – based authentication flow in our app.

FastAPI backend with users and relationships

Web applications (or mobile apps, for that matter) are not very useful if they are not secure – we keep hearing about tiny errors in the authentication implementations that ended up with hundreds of thousands or even millions of compromised accounts that might include sensitive and valuable information.

FastAPI is based on *OpenAPI* – previously known as *Swagger* – an open specification for crafting APIs. As such, OpenAPI enables us to define various security schemes, compatible with the various protocols (`apiKey`, `http`, `oauth2`, `openIdConnect`, and so on). While the FastAPI documentation website provides an excellent and detailed tutorial on creating an authentication flow, it is based on the `oauth2` protocol, which uses form data for sending the credentials (username and password).

There are literally dozens of ways you could implement some type of security for your API, but what I really want to accomplish in this chapter is just to give you an idea of what the viable options are and to create a simple authentication system based on JWT and JSON as the transport mechanism, a workflow that is easily extendable to fit your future needs, and one that provides just enough moving parts to be able to see the mechanism itself. In the following sections, we will devise a simple user model that will enable us to have an authentication flow. We will then learn how to encode the user data into a JWT token and how to require the token for accessing the protected routes.

Creating a User model and MongoDB relationships

In order to be able to even discuss the concepts of authenticating users, we have to introduce the entity of users to our app – up until now, we have only seen how to perform CRUD operations on a single entity (cars). A real application will probably have at least a couple of models, and the user's model is certainly going to be mandatory. While you could store various data in the user's model, it really depends on your needs; for a small application, a couple of fields will suffice – an email and/or username, a password, maybe some role (regular user, admin, or editor), and so on. For a publishing platform, you would want to add a short bio, maybe an image, and so on.

Modeling data with MongoDB is inherently different from modeling relational databases, as discussed in *Chapter 2*, *Setting Up the Document Store with MongoDB*, and the driving idea is to think of queries upfront and model your relationships, taking into account the queries that your app is going to be making most frequently.

First of all, what are our **requirements?** Well, our stakeholders are quite happy with the previous CRUD application, and eventually, they want to turn it into a public website – the cars should be displayed for potential customers, while the pages for inserting new cars and updating or deleting the existing ones should be protected. Two types of users are envisioned for the moment: salespersons – employees that can insert new cars and edit and delete "their" own cars (that is, the company cars for which they are responsible), and admins – a couple of managers who will oversee the whole process and who should be able to perform all the operations, regardless of whose entity it is. In order to keep things as simple as possible, I will make a simple reference-based model; the car will simply have an additional field – such as a foreign key – with the ID of the user, very similar to a relational database model. We could embed a list of all the users' cars into the user model, but in this app, this will be more than enough.

Let's begin with the models of our application. We should probably apply the same structure as we did for the routers – create a `/models` directory and two files (`users.py` and `cars.py`) – but in order to keep the project as simple as possible, I am going to put them together in a single `models.py` file. This should be avoided in cases where you have more than two models!

Let's begin with `main.py`, the entry point of our application, which will be very similar to the one used in the previous chapter:

```
from decouple import config
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from motor.motor_asyncio import AsyncIOMotorClient
from routers.cars import router as cars_router
from routers.users import router as users_router

DB_URL = config('DB_URL', cast=str)
DB_NAME = config('DB_NAME', cast=str)
```

I have just added a new router – the one that we will be creating right now:

```
origins = [
    "*"
]
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
```

```
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"]
)
@app.on_event("startup")
async def startup_db_client():
    app.mongodb_client = AsyncIOMotorClient(DB_URL)
    app.mongodb = app.mongodb_client[DB_NAME]

@app.on_event("shutdown")
async def shutdown_db_client():
    app.mongodb_client.close()

app.include_router(cars_router, prefix="/cars", tags=["cars"])
app.include_router(users_router, prefix="/users",
tags=["users"])
```

The `main.py` file is practically unaltered, and that is one of the benefits of having a modular structure for our app. We just mounted the additional `/users` router, while maintaining the same logic – connect the database client on startup, disconnect on shutdown, and load the database variables using `decouple`.

Let's create our `models.py` file now. The following code is almost identical to the one we wrote for our CRUD app in *Chapter 5*, *Building the Backend for Our Application* – we declare the imports and create `MongoBaseModel` in order to flatten `ObjectId` into a string:

```
from enum import Enum
from bson import ObjectId
from typing import Optional
from pydantic import EmailStr, Field, BaseModel, validator
from email_validator import validate_email, EmailNotValidError
```

We imported the `email_validator` package that is needed for, well, email validation:

```

class PyObjectId(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate
```

```
    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid objectid")
        return ObjectId(v)
    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class MongoBaseModel(BaseModel):
    id: PyObjectId = Field(default_factory=PyObjectId,
      alias="_id")
    class Config:
        json_encoders = {ObjectId: str}
```

Notice that we imported the email validator package since it is not part of Pydantic – you should install it with the following:

```
pip install  email-validator
```

It is a package needed for Pydantic to validate the email addresses since we want to require a valid email address when the user registers. Although I will not implement a client-side registration flow in this chapter, I will create the user creation route, and it will require a valid email address. Who knows, maybe the owner of the company decides to introduce user accounts for customers later on?

On to the same file, `models.py`, and to the actual user model. We are defining two roles – a salesperson and an admin – and a very basic user model containing only the username, email, password, and role fields. Since the email field is the only one that cannot be directly validated from Pydantic, we add a simple validation method by using the email validator package. It is really simple – it just returns an error if the value provided isn't a valid email:

```
class Role(str, Enum):
    SALESPERSON = "SALESPERSON"
    ADMIN = "ADMIN"

class UserBase(MongoBaseModel):
    username: str = Field(..., min_length=3, max_length=15)
    email: str = Field(...)
```

```
    password: str = Field(...)
    role: Role

    @validator("email")
    def valid_email(cls, v):
        try:
            email = validate_email(v).email
            return email
        except EmailNotValidError as e:
            raise EmailNotValidError
```

Since we have already seen how Pydantic handles validation, this should be pretty self-explanatory. I defined two roles – a salesperson and an admin – and the users must fit into one of these two roles. We are now ready to define our user model and a couple of helper models:

```
class UserBase(MongoBaseModel):
    username: str = Field(..., min_length=3, max_length=15)
    email: str = EmailStr(...)
    password: str = Field(...)
    role: Role
    @validator("email")
    def valid_email(cls, v):
        try:
            email = validate_email(v).email
            return email
        except EmailNotValidError as e:
            raise EmailNotValidError
class LoginBase(BaseModel):
    email: str = EmailStr(...)
    password: str = Field(...)
class CurrentUser(BaseModel):
    email: str = EmailStr(...)
    username: str = Field(...)
    role: str = Field(...)
```

`UserModel` is simple enough: we require a username with a length between 3 and 15 characters, a valid email address, a password, and a role. I have added two additional models: `LoginBase` for the login route, and `CurrentUser`, which contains the data that we will extract from the model when we want to check who is currently making the requests. Moving on to the `Cars` model that I decided to put in the same `models.py` file, little has changed:

```
class CarBase(MongoBaseModel):
    brand: str = Field(..., min_length=3)
    make: str = Field(..., min_length=1)
    year: int = Field(..., gt=1975, lt=2023)
    price: int = Field(...)
    km: int = Field(...)
    cm3: int = Field(..., gt=600, lt=8000)

class CarDB(CarBase):
    owner: str = Field(...)
class CarUpdate(MongoBaseModel):
    price: Optional[int] = None
```

The base model is intact, with all the features that we had earlier (mileage, year of production, and so on). I have just added a new model, called `CarDB`, which extends the `CarBase` model and adds an owner field – that is, `id` of the user assigned to the car, and since we converted all our MongoDB `ObjectIds` to strings, it is a string as well. The `CarUpdate` model contains only the optional price update.

It is important to point out that this model is a great oversimplification of a real system. We would probably want to add a list of car IDs as a field in the User model, we would have a bunch of `DateTime` fields denoting the moment when the car was put up for sale, sold, reserved, and so on. However, in this chapter, I only want to implement a rather simple JWT-based authentication system and keep only the bare minimum functionality needed in order to have a working mechanism.

On to the authentication file, aptly called `authentication.py`. Let's quickly go over the requirements of our authentication mechanism:

- Once the user submits the registration form and sends us their password, the password should be hashed and only then inserted into the database
- We should have a function ready to compare this stored password hash with the subsequent passwords submitted during the login phase in order to verify whether they match
- We should be able to create/encode JWT tokens and decode them with a custom expiry time and with a payload containing the *user ID*

- Finally, we should have a function that accepts the request through dependency injection and returns either the ID of the user making the request or a message such as *invalid token* or *token expired*

The following mechanism is inspired and adapted from a YouTube video (`https://www.youtube.com/watch?v=xZnOoO3ImSY`), which offers an alternative and simpler approach than the one proposed in the FastAPI documentation.

Let's begin building our `authentication.py` file. First, we need to install a couple of libraries needed for the JWT authentication mechanism. `Pyjwt` is a library for encoding and decoding JWTs, while `passlib` is a library for hashing strings. Stop your FastAPI server and, in the active environment, insert the following command:

```
pip install pyjwt passlib['bcrypt']
```

Now we are ready to declare our imports in the `authentication.py` file as follows:

```
import jwt
from fastapi import HTTPException, Security
from fastapi.security import HTTPAuthorizationCredentials,
HTTPBearer
from passlib.context import CryptContext
from datetime import datetime, timedelta
```

As we said earlier, `jwt` is here to enable us to encode and decode JWTs, while FastAPI provides us with the bulk of the needed functionality. `HTTPException` is going to take care of cases in which the token is not valid – effectively turning exceptions in code into valid HTTP responses – while `Security` is used for authorization and for highlighting routes that will need a user to be authenticated in the automatic documentation. `HTTPBearer` is a FastAPI class ensuring that the HTTP request has the appropriate authentication header, while `HTTPAuthorizationCredentials` is the object type returned from the dependency injection.

`CryptContext` is used for creating a context for hashing passwords and it lives under `passlib.context`. Finally, we imported some `datetime` utilities for signing the token and giving it the desired expiry date.

After having declared our imports, it is time to create a class I will call `Authorization`, which will expose methods responsible for all the needed authentication steps:

```
class Authorization():
    security = HTTPBearer()
    pwd_context = CryptContext(schemes=["bcrypt"],
      deprecated="auto")
    secret = 'FARMSTACKsecretString'
```

We are instantiating FastAPI's simplest authentication – `HTTPBearer` – and creating a password context with `CryptContext`, using the `bcrypt` algorithm. We also need a secret string that could be generated automatically for increased security. Next, we will take care of hashing the password:

```
    def get_password_hash(self, password):
        return self.pwd_context.hash(password)

    def verify_password(self, plain_password, hashed_password):
        return self.pwd_context.verify(plain_password, hashed_
            password)
```

These rather simple functions ensure that the user's password is hashed and that it can be verified by comparing it to the plain text version. The second function returns a simple true or false value. We are now at the heart of the class – creating the JWT:

```
    def encode_token(self, user_id):
        payload = {
            'exp': datetime.utcnow() + timedelta(days=0,
minutes=35),
            'iat': datetime.utcnow(),
            'sub': user_id
        }
        return jwt.encode(
            payload,
            self.secret,
            algorithm='HS256'
        )
```

The preceding function does the bulk of the work – it takes `user_id` as the sole parameter and puts it in the `sub` section of the payload. Bear in mind that we could encode more information in the JWT – the user's role or username for instance. In that case, the `sub` section would have a structure of a dictionary and the JWT would be considerably longer. The expiry time is set to 35 minutes, while `issued at time` is set to the moment of JWT creation. Finally, the function uses the `jwt.encode` method to encode the token. We provide the algorithm (`HS256`) and a secret as arguments.

The decode part of the class is very similar; we just reverse the process and provide exceptions in case they are needed:

```
    def decode_token(self, token):
        try:
            payload = jwt.decode(token, self.secret,
```

```
algorithms=['HS256'])
            return payload['sub']
        except jwt.ExpiredSignatureError:
            raise HTTPException(status_code=401,
detail='Signature has expired')
        except jwt.InvalidTokenError as e:
            raise HTTPException(status_code=401,
detail='Invalid token')
```

The `decode_token` function returns just the `sub` part of the token – in our case, the user's ID – while we provide appropriate exceptions in case the token is not valid or if it has expired. Finally, we create our `auth_wrapper` function that will be used for dependency injection in the routes. If the function returns the user's ID, the route will be accessible; otherwise, we will get `HTTPException`:

```
    def auth_wrapper(self, auth: HTTPAuthorizationCredentials =
Security(security)):
        return self.decode_token(auth.credentials)
```

The `authorization.py` file is under 40 lines long but packs quite a punch – it enables us to protect routes by leveraging the excellent FastAPI's dependency injection mechanism.

Let's dive into the *users* router and put our authentication logic to the test. In the `routers` folder, create a `users.py` file and begin with the imports and class instantiations:

```
from fastapi import APIRouter, Request, Body, status,
HTTPException, Depends
from fastapi.encoders import jsonable_encoder
from fastapi.responses import JSONResponse
from models import UserBase, LoginBase, CurrentUser
from authentication import AuthHandler

router = APIRouter()
auth_handler = AuthHandler()
```

After the standard FastAPI imports, including `jsonable_encoder` and `JSONResponse`, we import our user models and the `AuthHandler` class from `authorization.py`. We then proceed to create the router that will be responsible for all the user's routes and an instance of `AuthHandler`. Let's begin with a registration route, so we can create some users and test them with a REST client:

```
@router.post("/register", response_description="Register user")
async def register(request: Request, newUser: UserBase =
```

```
Body(...)) -> UserBase:

    newUser.password = auth_handler.get_password_hash(newUser.
        password)
    newUser = jsonable_encoder(newUser)
```

The register route, which will be available at the `/users/register` URL, takes in a request and a `newUser` instance, modeled by Pydantic's `UserBase` class, through the body of the request. The first thing that we do is replace the password with the hashed password and convert the Pydantic model into a `jsonable_encoder` instance.

Now, we perform the standard registration checks – the email and the username should be available; otherwise, we throw an exception, notifying the user that the username or password has already been taken:

```
    if (
        existing_email := await request.app.mongodb["users"].
            find_one({"email": newUser["email"]}) is not None):
        raise HTTPException(
            status_code=409, detail=f"User with email
                {newUser['email']} already exists"
        )
    if (
        existing_username := await request.app.mongodb["users"]
            .find_one({"username": newUser["username"]}) is not
                None):
        raise HTTPException(
            status_code=409, detail=f"User with username
                {newUser['username']} already exists",
        )
```

The previous functions could and should be refactored to allow for further checks, but I want them to be as explicit as possible. The final part of the function is trivial; we just need to insert the user into MongoDB! You can see this in the following code:

```
    user = await request.app.mongodb["users"].insert_
        one(newUser)
    created_user = await request.app.mongodb["users"].find_one(
        {"_id": user.inserted_id}
    )
```

```
    return JSONResponse(status_code=status.HTTP_201_CREATED,
        content=created_user)
```

We return the standard `201 CREATED` status code, and we are now ready to perform some basic tests using *HTTPie*, our command-line REST client. Let's try and create a user as follows:

```
(venv) λ http POST 127.0.0.1:8000/users/register
username="bill" password="bill" role="ADMIN" email="koko@gmail.
    com"
{
    "_id": "629333d7e33842d9499e6ac7",
    "email": "koko@gmail.com",
    "password": "$2b$12$HKGcr5CnxV7coSMgx41gRu34Q11Qb.
        m5XZHlX1tslH8ppqlVB2oJK",
    "role": "ADMIN",
    "username": "bill"
}
```

We get a new user with a hashed password, a role, and `_id`. Of course, we wouldn't want to send the password back to the user, even if it is hashed, but you already have the knowledge to create a new Pydantic model that returns all the fields except the password. Let's move on to the login route – it is very similar to what you might have already used with Flask or Express.js. We receive the email and password (we could have opted for a username) and, first, we try to find the user by email. After that, we compare the password with our hashing function:

```
@router.post("/login", response_description="Login user")
async def login(request: Request, loginUser: LoginBase =
Body(...)) -> str:
    user = await request.app.mongodb["users"].find_
        one({"email": loginUser.email})
    if (user is None) or (
        not auth_handler.verify_password(loginUser.password,
            user["password"])
    ):
        raise HTTPException(status_code=401, detail="Invalid
            email and/or password")
    token = auth_handler.encode_token(user["_id"])
    response = JSONResponse(content={"token": token})
    return response
```

If the user exists and the password passes the hash verification, we create a token and return it as a JSON response. This precious token will then be responsible for authentication all over our app and it will be the only data that needs to be sent to the server with every request. We can test the login route as well by hitting the `/users/login` route with the appropriate credentials:

```
λ http POST http://127.0.0.1:8000/users/login email="tanja@
gmail.com" password="tanja"
HTTP/1.1 200 OK
content-length: 184
content-type: application/json
date: Wed, 01 Jun 2022 20:13:32 GMT
server: uvicorn
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJleHAiOjE2NTQxMTY1MTMsImlhdCI6MTY1NDExNDQxMywic3
ViIjoiNjI4OTQyODU3YTBjYmZlNGE2MzQwNDdkIn0.v1FTBM0wIlDKUw_
VVCJlsSItM58sDzDnwGbzyDKs_pc"
}
```

We got the token back! If you want, you can try the same route with the wrong username/password combination and check the response.

We will need one final route in the `users` router: the `/me` route. This route is not supposed to be called directly and generate a page, only to be used as a helper – a way of verifying the currently logged user making the request. The `/me` route should not accept any parameters except the authentication dependency – the perfect opportunity to test our authentication wrapper:

```
@router.get("/me", response_description="Logged in user data")
async def me(request: Request, userId=Depends(auth_handler.
auth_wrapper)):
    currentUser = await request.app.mongodb["users"].find_
        one({"_id": userId})
    result = CurrentUser(**currentUser).dict()
    result["id"] = userId
    return JSONResponse(status_code=status.HTTP_200_OK,
        content=result)
```

This route is pretty simple: if the provided token is valid and not expired, `auth_wrapper` will return `userId` – the ID of the user making the request. Otherwise, it will return an HTTP exception. In this route, I have added a database call in order to retrieve the desired data about the user, according to the `CurrentUser` model.

We could have encoded all this data in the token and avoided the trip to the database, but I wanted to leave the JWT as thin as possible.

Now, we can test the `/me` route. First, let's log in with our previously registered user:

```
(venv) λ http POST 127.0.0.1:8000/users/login password="bill"
email="koko@gmail.com"
HTTP/1.1 200 OK
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOj
E2NTM4Mzk1NTksImlhdCI6MTY1MzgzNzQ1OSwic3ViIjoiNjI5MzMzZDdlMzM
4NDJkOTQ5OWU2YWM3In0.
ajpoftEFBWcfn2XClJqPDNcJMaS6OujZpaU8bCv0BNE"
}
```

Copy this token and provide it to the `/me` route:

```
(venv) λ http GET 127.0.0.1:8000/users/me "Authorization:
Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOj
E2NTM4Mzk1NTksImlhdCI6MTY1MzgzNzQ1OSwic3ViIjoiNjI5MzMzZDdlMzM4
NDJkOTQ5OWU2YWM3In0.
ajpoftEFBWcfn2XClJqPDNcJMaS6OujZpaU8bCv0BNE"
HTTP/1.1 200 OK
{
    "email": "koko@gmail.com",
    "id": "629333d7e33842d9499e6ac7",
    "role": "ADMIN",
    "username": "bill"
}
```

If you test the route without the bearer token, you will get a `Not Authenticated` error and you will be back at square one.

Finally, I will show how to insert the authentication dependency into the `/cars` router (or really, any other router that you might need to create). Since it will be a pretty long file, I will not explain all of it – I will rather focus on the logic used to perform authentication and authorization on a couple of routes, while the entire file is available in the book's GitHub repository. Let's see the imports for the `/cars` router:

```
from typing import List, Optional
from fastapi import APIRouter, Request, Body, status,
```

```
HTTPException, Depends
from fastapi.encoders import jsonable_encoder
from fastapi.responses import JSONResponse

from models import CarBase, CarDB, CarUpdate
from authentication import AuthHandler

router = APIRouter()
auth_handler = AuthHandler()
```

This part is almost identical to the `users` router – we import our Pydantic models and instantiate the router and the authentication handler class.

Our Pydantic `cars` model has changed; now it has something called `owner`, basically just the ID of the user that is supposed to sell the car. We will provide this information to the database through our authentication dependency injection. When the user attempts to insert a new instance of the entity (a car), they will have to be authenticated in order to proceed. If they are authenticated, we will just take their ID and set it as the `owner` field value:

```
@router.post("/", response_description="Add new car")
async def create_car(
    request: Request,
    car: CarBase = Body(...),
    userId=Depends(auth_handler.auth_wrapper),
):
    car = jsonable_encoder(car)
    car["owner"] = userId
    new_car = await request.app.mongodb["cars2"].insert_
        one(car)
    created_car = await request.app.mongodb["cars2"].find_one(
        {"_id": new_car.inserted_id}
    )
    return JSONResponse( status_code=status.HTTP_201_CREATED,
        content=created_car)
```

The simplest case would be the path corresponding to GET `/cars` – a route that would list all available cars, with some pagination implemented through query strings. Let's say that we want only logged users (so salespersons or admins) to access this route.

All we need to do is inject the authentication wrapper into the dependency, and FastAPI has only two choices: either the token is valid and fresh and we get a user, or `HTTPException` is raised; it is really as simple as that. So, let's create our route for listing cars – we're assuming that only registered users can access this route:

```
@router.get("/", response_description="List all cars")
async def list_all_cars(
    request: Request,
    min_price: int = 0,
    max_price: int = 100000,
    brand: Optional[str] = None,
    page: int = 1,
    userId=Depends(auth_handler.auth_wrapper),
) -> List[CarDB]:
    RESULTS_PER_PAGE = 25
    skip = (page - 1) * RESULTS_PER_PAGE
    query = {"price": {"$lt": max_price, "$gt": min_price}}
    if brand:
        query["brand"] = brand
    full_query = (
        request.app.mongodb["cars2"]
        .find(query)
        .sort("_id", -1)
        .skip(skip)
        .limit(RESULTS_PER_PAGE)
    )
    results = [CarDB(**raw_car) async for raw_car in full_
        query]
    return results
```

While the whole function performs some pagination (25 hardcoded results per page) and has some nifty options for filtering by price and by brand, the gist of authentication logic is in the bold line. Also, please note that I have created a separate MongoDB collection and named it `cars2`, just to differentiate it from the collection used in the previous chapter, while using the same database.

Finally, let's examine the route for editing a car (just the price, in our case). We want only the owner of the car to be able to edit the price and, additionally, any admin can also step in and update the price. For this case, it would have been wise if we had encoded the role of the user as well in the JWT, as it would save us a trip to the database, but I just want to make you aware of some decisions and trade-offs that you are bound to make during the development of the API:

```
@router.patch("/{id}", response_description="Update car")
async def update_task(
    id: str,
    request: Request,
    car: CarUpdate = Body(...),
    userId=Depends(auth_handler.auth_wrapper),
):
    user = await request.app.mongodb
      ["users"].find_one({"_id": userId})
    findCar = await request.app.mongodb
      ["cars2"].find_one({"_id": id})

    if (findCar["owner"] != userId) and user["role"] !=
      "ADMIN":
        raise HTTPException(
            status_code=401, detail="Only the owner or an
              admin can update the car"
        )
    await request.app.mongodb["cars2"].update_one(
        {"_id": id}, {"$set": car.dict(exclude_unset=True)}
    )
    if (car := await request.app.mongodb
      ["cars2"].find_one({"_id": id})) is not None:
        return CarDB(**car)
    raise HTTPException
      (status_code=404, detail=f"Car with {id} not found")
```

In this route handler, we first get the user making the request, and then we locate the car to be edited. Finally, we perform a check: if the owner of the car is *not* the user making the request and this user is not an admin, we throw an exception. Otherwise, we perform the update. FastAPI's dependency injection is a simple and powerful mechanism that really shines in the authentication and authorization domain!

In this section, we have created a simple but efficient authentication system on our FastAPI backend, we have created a JWT generator and we are able to verify the tokens, we have protected some routes, and provided the routes needed for creating (registering) new users and logging in. It is now time to see how things work on the frontend!

Authenticating the users in React

As with the other aspects of security, authentication in React is a huge topic and is beyond the scope of this book. In this section, I will give you just a very basic mechanism that enables us to have a simple authentication flow on the client side. Everything will revolve around the JWT and the way we decide to handle it. In this chapter, we are going to store it just in memory.

The internet and the specialized literature are full of debates on what is the optimal solution for storing authentication data – in our case, the JWT token. As always, there are pros and cons to each solution and at the beginning of this section.

Cookies have been around for a very long time – they can store data in key-value pairs in the browser and they are readable both from the browser and the server. Their popularity coincided with the classic server-side rendered websites. However, they can store a very limited amount of data, and the structure of said data has to be very simple.

Localstorage and Session Storage were introduced with HTML5 as a way to address the need for storing complex data structures in single-page applications, among other things. Their capacity is around 10 MB, depending on the browser's implementation, compared to 4 KB of cookie capacity. Session storage data persists through a session, while local storage remains in the browser, even after it is closed and reopened, until manually deleted. Both can host complex JSON data structures.

Storing JWT in localstorage is nice, it's easy, and it allows for a great user experience and developer experience. It is, however, frowned upon since it opens the application to a wide array of vulnerabilities, since they can be accessed by any client-side JavaScript running in the browser.

The majority of authorities on the subject suggest storing JWT in HTTP – only cookies, cookies that cannot be accessed through JavaScript and require the frontend and the backend to run on the same domain. This can be accomplished in different ways, through routing requests, using a proxy, and so on. Another popular strategy is the use of so-called refresh tokens – we issue one token upon login and then this token is used to generate other (refresh) tokens automatically, allowing us to mitigate between security and user experience.

In this section, I will build a very simple and minimalistic React app that will just barely meet the requirements; some routes and pages should be protected unless the user logs in. I will not persist the JWT in any way – when the user refreshes the application, they are logged out. Not the most pleasant user experience, but that is not the issue right now.

Let's proceed step by step. We have our FastAPI backend running, and we are ready to create our simple frontend:

1. Navigate to your `/chapter7` directory and, from the terminal, create a React app:

```
npx create-react-app frontend
```

2. Change the directory into the frontend and install Tailwind CSS:

```
npm install -D tailwindcss postcss@latest autoprefixer
```

3. Initialize Tailwind with the following command:

```
npx tailwindcss init -p
```

4. Now, it is time to edit `postcss.config.js`:

```
module.exports = {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

5. Finally, delete everything in the `src/index.css` file and replace the content with the following:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

These steps should be familiar by now, but now I want to take the process one step further. Tailwind has gained in popularity over the last few years, and different UI kits and utilities are based on the basic Tailwind classes. One of the most popular and usable ones is called DaisyUI (`https://daisyui.com`), and we will use it for prototyping our app.

6. The installation process is similar to Tailwind itself. In the terminal, type the following:

```
npm i daisyui
```

7. When completed, we need to register `daisyui` as a Tailwind plugin in `tailwind.config.js` as follows:

```
module.exports = {
  //...
  plugins: [require("daisyui")],
}
```

8. Finally, delete all the unneeded files (such as `App.css` and `Logo.svg`) and reduce your `App.js` file to the following in order to test that React has picked up the UI dependencies:

```
function App() {
  return (
    <div className="App bg-zinc-500 min-h-screen flex
        flex-col justify-center items-center">
      <button class="btn btn-primary">It works!</button>
    </div>
  );
}
export default App;
```

9. Now, we can test the app and see that both Tailwind and DaisyUI are functioning correctly – you should be able to see a pretty empty page with a styled button. I had to run the following again:

```
npm install postcss@latest
```

Maybe by the time you are reading this, the fix will not be necessary anymore.

10. For authentication purposes, we will dive a bit deeper into the React Router 6 and we will take advantage of some of its new features and components. Stop the terminal process and bravely install the router:

```
npm install react-router-dom@6
```

11. We are going to set the router up in the `index.js` file as follows:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import {BrowserRouter, Routes, Route} from 'react-router-
    dom'
import App from './App';
const root = ReactDOM.createRoot(document.
```

```
getElementById('root'));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <Routes>
        <Route path='/*' element={<App />} />
      </Routes>
    </BrowserRouter>
  </React.StrictMode>
);
```

We are wrapping everything in the router so that it "covers" the whole application and the path is a catch-all asterisk (`*`), while the element that needs to be provided to the router is the root `App.js` component. Now comes the tedious part of defining all the possible routes and components, but again, we are going to use React Router's new features – nested routes. Instead of wrapping each and every component into a `Layout` component – containing the common web page elements such as navigation or footer – we are going to use the Router's `Outlet` component, which just fills the component with the content of the nested component that matches the URL pattern.

12. Let's create a components folder under `/src` and build generic `Header.jsx` and `Footer.jsx` components, making use of our React ES6 Visual Studio Code extension (by typing `_rafce`):

```
const Header = () => {
  return <div>Header</div>;
};
export default Header;
```

13. Following the exact same procedure, create the following components in the `/src/components` folder: `Footer`, `HomePage`, `Login`, and `Register`, containing just a `div` returning the component's name. `Layout.jsx` will make use of the nested routing:

```
import { Outlet } from "react-router-dom";
import Header from "./Header";
import Footer from "./Footer";

const Layout = () => {
  return (
    <div className="App flex flex-col min-h-screen">
      <Header />
```

```
        <main className="flex-1 min-h-full flex flex-col
            align-middle justify-center items-center">
          <Outlet />
        </main>
        <Footer />
      </div>
    );
  };
  export default Layout;
```

The `Layout` component is simple but very useful: it makes use of the `Outlet` component that acts as a high-order component, effectively wrapping the contained routes and adding the header and the footer. I have made the page full-height using Tailwind's classes and set the display to flex. The main section is set to `flex-1`, to take up all the remaining space.

The `App.js` file is now updated as follows:

```
import {Route, Routes} from "react-router-dom"
import Layout from "./components/Layout";
import Login from "./components/Login";
import Register from "./components/Register";
import HomePage from "./components/HomePage";

function App() {
  return (
    <Routes>
      <Route path="/" element={<Layout />}>
        <Route path="/" element={<HomePage />} />
        <Route path="login" element={<Login/>} />
        <Route path="register" element={<Register/>} />
      </Route>
    </Routes>
  );
}
export default App;
```

A keen observer will immediately notice that the `Route` element that uses the `Layout` component as the rendering element is *not self-closing* – it, in fact, encloses all the remaining routes, channeling them, and in the process, adding the `Header` and `Footer` components. Excellent and elegant! You can manually try to change the URL; navigate to `/login` or `/register` or simply `/` (the root URL of the React site), and see whether the middle section updates. The router is set up and working. We will add more routes for the CRUD operations on cars, but they will be protected – the user will have to provide credentials in the form of a valid JWT token in order to access them (and even if they could access the React routes, without a token, the operations couldn't be performed on the backend). It is time to introduce another React hook – `useContext`.

Context is a way of solving the problem known as prop-drilling in React when a component that is located down in the component tree needs to accept props through a series of components – parents that do not essentially need them. Context is a method of sharing values (strings, numeric values, lists, and objects) with all the components that are enclosed in a context provider. The `useContext` Hook – used to interact with the context – is one of the coolest features of the new Hook-based React, and something that can handle lots of common design problems.

Using context is a bit particular, not like the `useState` or `useEffect` hook, as it involves a bit more moving parts, but we will use the simplest version, coupled with a custom Hook, for easier access.

The first step is to create a context, using `createContext` provided by React. This function accepts default arguments, so you could provide it with, for instance, a dictionary: `{username:"Marko"}`. This argument will only be used unless no value is provided otherwise. Even functions for setting or modifying the context values can be passed to the context – and that is precisely what we are going to do. We can set up an `auth` value that will store the logged-in user's data (if any), but also a `setAuth` function, called when the user logs in, that will set the user data. We could also use this function for logging the user out, by simply setting the context value of `auth` to a `null` value.

The second step is to use a context provider – a React component that allows other components to consume our context. All the consumers that are wrapped inside the provider will re-render once the context (the provider's value) changes. The provider is the vehicle for providing the context value(s) to the child component, instead of props.

Now comes the Hook, `useContext`, which takes a context as an argument and makes it available to the component. We will use it for accessing the context. Let's move on to the example, as it will become clearer. Follow these steps:

1. I will create the simplest possible context with a single state variable called `auth` (with a `useState` Hook, `setAuth`) in the `/src/context/AutProvider.js` file:

```
import { createContext, useState } from "react";
const AuthContext = createContext({})
export const AuthProvider = ({children}) => {
    const [auth, setAuth] = useState({
```

```
    })
    return <AuthContext.Provider value={{auth, setAuth}}>
        {children}
    </AuthContext.Provider>
}
export default AuthContext
```

2. Now, we can wrap our `Router` routes in the `index.js` file and make `auth` and `setAuth` available to all the routes. Edit the `index.js` file:

```
import { AuthProvider } from './context/AuthProvider';
 …
<React.StrictMode>
  <BrowserRouter>
   <AuthProvider>
    <Routes>
      <Route path='/*' element={<App />} />
    </Routes>
    </AuthProvider>
  </BrowserRouter>
</React.StrictMode>
```

Finally, since we do not want to have to import both the `AuthContext` provider and `useContext` in every component, we will create a simple utility Hook that will import the context for us.

3. In the `/src/hooks` folder, create a file called `useAuth.js`:

```
import { useContext } from "react";
import AuthContext from "../context/AuthProvider";
const useAuth = () => {
    return useContext(AuthContext)
}
export default useAuth;
```

This setup might seem complicated, but it really isn't – we just had to create one context and one hook to facilitate our job. The benefit is that now we can cover the entire area of the app and set and get the value of our `auth` variable. Let's begin using our React authentication mechanism and create the `Login` component – the one that will actually get us logged in. For the form handling, I want to introduce a third-party package: *React-Form-Hook* (`https://react-hook-form.com/`).

We have already seen that manual form handling in React can get pretty tedious, and there are some excellent and battle-tested solutions. In this chapter, we will get to use the React form hook. Let's begin by installing it:

```
npm install react-hook-form
```

Restart the React server with `npm run start` and fire up the `Login.jsx` component. This will arguably be the most complex component logic-wise, so let's break it down:

```
import { useForm } from "react-hook-form";
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import useAuth from "../hooks/useAuth";
```

We import the `useForm` Hook, `useState` for some state variables, the `useNavigate` Hook from the router for redirecting after the login, and our `useAuth` Hook since we want to set the authentication context after a successful login. We then begin to draw our component and set up the Hook:

```
const Login = () => {
  const [apiError, setApiError] = useState();
  const { setAuth } = useAuth();
  let navigate = useNavigate();
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm();
```

The `ApiError` variable should be self-explanatory – I will use it to store potential errors generated from the backend in order to display them later. The `navigate` is necessary for programmatic navigation to different pages inside the router, while `react-form-hook` gives us several useful tools: `register` is used to register the form inputs with the instance of the Hook, `handleSubmit` is for, well, handling the submitting of the form, while `errors` will host the errors during the process. Let's continue with the code:

```
  const onFormSubmit = async (data) => {
    const response = await fetch("http://127.0.0.1:8000/users/
        login", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
```

```
      },
      body: JSON.stringify(data),
    });
    if (response.ok) {
      const token = await response.json();
      await getUserData(token["token"]);
    } else {
      let errorResponse = await response.json();
      setApiError(errorResponse["detail"]);
      setAuth(null);
    }
  };
  const onErrors = (errors) => console.error(errors);
```

The `onSubmit` is pretty similar to what we have already done manually: we send a `POST` request to the `/login` endpoint with the form data encoded as JSON. If everything is fine (an OK response, which is short for a response code in the 200–299 range), we proceed and get the token. We then feed this token to another function called `getUserData`. If the API sends any error, we take this error and put it in the `apiError` state variable. Remember, FastAPI has this nice `detail` key that contains the human-readable message error. The errors are simply sent to the console.

Let's take a look at the `getUserData` function – it is simply a call to the `/me` route on the backend:

```
  const getUserData = async (token) => {
    const response = await fetch("http://127.0.0.1:8000/users/
        me", {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`,
      },
    });
    if (response.ok) {
      let userData = await response.json();
      userData["token"] = token;
      setAuth(userData);
      setApiError(null);
      navigate("/", { replace: true });
```

```
    }
  };
```

This is the function that actually makes use of our token – we add it to the header of the request and if a user is retrieved (an OK response), we use that user's data to populate the `auth` object in `authContext`. Finally, we send the user to the home page with the help of the router. The remaining portion of the function is the markup and some utility classes:

```
return (
    <div className="mx-auto p-10 rounded-lg shadow-2xl">
      <h2 className="text-xl text-primary text-center font-bold
          my-2">
        Login page
      </h2>
      <form onSubmit={handleSubmit(onFormSubmit,
        onErrors)}>
        <div className="flex flex-col justify-center items-
          center">
          <input
            type="text"
            placeholder="email@email.com"
            className="input input-bordered input-accent w-
              full max-w-xs m-3"
            name="email"
            autoComplete="off"
            {...register("email", { required: "The email is
              required" })}
          />
          {errors?.email && errors.email.message}

          <input
            type="password"
            placeholder="your password"
            className="input input-bordered input-accent w-
             full max-w-xs m-3"
            name="password"
            {...register("password", { required: "The
```

```
            password is required" })}
          />
          {errors?.password && errors.password.message}
```

Finally, there is just some simple markup for displaying HTML elements:

```
          <button className="btn btn-outline btn-accent m-3
              btn-block">
            Login
          </button>
        </div>
      </form>
      {apiError && (
        <div className="alert alert-error shadow-lg">
          <div>

            <span>{apiError}</span>
          </div>
        </div>
      )}
    </div>
  );
};
export default Login;
```

It is important to note that each field in the form has a register prop that binds it to the form that is controlled by the React form hook. If we try to log in with a non-existing email or password, the API errors will be displayed, while if everything goes well, we should be redirected to the home page. In order to see the `auth` data, we can take a look at the React extension in Chrome after the redirect. In the **Components** tab, under **ContextProvider**, you should be able to see all the data stored in the `auth` object.

It will be difficult to continue developing without proper navigation, so let's visit the DaisyUI website and find a suitable navigation bar. After snooping around, I found the following solution that required some copying and some adjustments for the structure of the React Router's links:

```
import React from "react";
import { Link } from "react-router-dom";
import useAuth from "../hooks/useAuth";
```

```
const Header = () => {
  const { auth, setAuth } = useAuth();
  return (
    <div className="navbar bg-primary text-primary-content">
      <div className="flex-1">
        <Link className="btn btn-ghost normal-case text-xl"
            to="/">FARM Cars </Link>
        <span className="border-2 border-amber-500 p-1">
          {auth?.username
            ? `Logged in as ${auth?.username} - ${auth.role}`
            : "Not logged in"}
        </span>
      </div>
      <div className="flex-none">
        <ul className="menu menu-horizontal p-0">
          {!auth?.username && (
            <li className="mx-1">
              <Link to="/login">Login</Link>
            </li>
          )}
          {!auth?.username && (
            <li className="mx-1">
              <Link to="/register">Register</Link>
            </li>
          )}
          {auth?.username && (
            <li className="mx-1">
              <button className=" btn-warning">
                Logout <span className="font-semibold">{auth?.
                    username}</span>
              </button>
            </li>
          )}
        </ul>
      </div>
    </div>
```

```
  );
};
export default Header;
```

This is a regular navigation menu with a couple of context niceties: we import our `useAuth` Hook and immediately gain access to `authContext`. This enables us to conditionally show or hide the **Login and register** or **Logout** links. I added a small span inside the navbar to notify the user whether there's anybody logged in or not. Since the default theme is pretty bland, I am going to apply a DaisyUI theme – you can explore them on `https://daisyui.com/docs/themes/`. I like the autumn theme, so I am just going to find the `index.html` file and add `data-theme="autumn"` to the `html` opening tag.

Our **Logout** button is not doing anything useful, so let's add a `logout` handler in the same `Header.jsx` file:

```
    let navigate = useNavigate();
    const logout = () =>{
        setAuth({})
        navigate("/login", {replace:true})
    }
```

And just add the `onClick` handler to the **Logout** button and set it to `{logout}`.

We have created a very simple authentication system, but we have no routes to protect, especially routes that involve cars: updating, adding, and deleting. That is the final part of the authentication system that I want to show here. There are many ways to prevent certain components from showing or displaying conditionally in React. An elegant way is making use of the React router again – with the use of outlets.

Simply put, we will make an authentication component that will just check for the presence of the `auth` data – if the data is present, you will be served the outlet, the enclosed protected routes, and corresponding components, and if not, the router will send you to the login page (or whatever page you wish).

Let's create a component called `RequiredAuthentication.jsx`:

```
import { useLocation, Navigate, Outlet } from "react-router-
dom";
import useAuth from "../hooks/useAuth";

const RequireAuthentication = () => {
  const { auth } = useAuth();
  const location = useLocation;
```

```
  return auth?.username ? <Outlet /> : <Navigate to="/login"
/>;
};
export default RequireAuthentication;
```

The component acts as a simple switch: if the username is present in the `auth` object, the outlet takes over and lets the client through to any route that is enclosed. Otherwise, it forces navigation to the `/login` route.

This isn't much different than some other approaches that use a simple functional component and then conditionally render the reserved output or the login route.

In order to be able to see our authentication logic in practice, we need at least one protected route. Let's create a new component and call it `CarList.jsx`. It will simply display all the cars in the database, but in order to be accessible, the user will have to be logged in – either as an admin or a salesperson. The `CarList` component has some standard imports and Hooks:

```
import { useEffect, useState } from "react"
import useAuth from "../hooks/useAuth"
import Card from "./Card"
const CarList = () => {
  const { auth } = useAuth()
  const [cars, setCars] = useState([]);
```

The `Card` component is really not important here – it is just a card element provided by DaisyUI, similar to the one we used in *Chapter 6, Building the Frontend of the Application* in order to display the car information. The `useAuth` hook provides us with a fast way to check for the authenticated user information through Context. The `useEffect` Hook is used to make a call to the FastAPI server and populate the `cars` array:

```
useEffect(() => {
    fetch("http://127.0.0.1:8000/cars/", {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${auth.token}`,
      },
    })
      .then((response) => response.json())
      .then((json) => {
        setCars(json);
```

```
      });
  }, []);
```

Finally, the JSX for returning the list of cars is just a map over the array of cars:

```
return (
    <div>
      <h2 className="text-xl text-primary text-center font-
         bold my-5">
        Cars Page
      </h2>
      <div className="mx-8 grid grid-cols-1 md:grid-cols-2
        gap-5 p-4">
        {cars &&
          cars.map((el) => {
            return <Card key={el._id} car={el} />;
          })}
      </div>
    </div>
  );
};
export default CarList;
```

In order to hook this component up with the application, we need to update the `App.js` file with the routes:

```
<Routes>
      <Route path="/" element={<Layout />}>
        <Route path="/" element={<HomePage />} />
        <Route path="login" element={<Login/>} />
        <Route path="register" element={<Register/>} />
        <Route element={<RequireAuthentication />}>
          <Route path="cars" element={<CarList/>} />
        </Route>
      </Route>
    </Routes>
```

Notice how we wrapped the `CarList` component inside the `RequireAuthentication` route: we could add other routes that need authentication in the same way, and we could also perform more granular control over which user can access which route. It is easy to edit the `RequireAuthentication` component and perform additional checks on the type of authenticated user – so we could have an area for admins only, but not for regular salespersons and so on.

Finally, let's update the `Header.jsx` component as well, in order to show the link to the newly created `/cars` route:

```
          {!auth?.username && (
            <li className="mx-1">
              <Link to="/register">Register</Link>
            </li>
          )}
          <li className="mx-1">
            <Link to="/cars">Cars</Link>
          </li>
```

I have left the link visible for all visitors – logged in or not – in order to showcase the authentication route's functionality; if you click the link without being logged in, you will be sent to the login page, otherwise, you should see a nice set of cards with the cars displayed.

There is really no need to present the remaining CRUD operations on the cars that should require authentication – we have already seen how the backend checks for the appropriate user by reading the JWT token, so it is just a matter of ensuring that the token is present and valid.

As I underlined earlier, authentication and authorization are probably the most fundamental and serious topics in any application, and they put before the developer and stakeholders a series of challenges and questions that need to be addressed early on. While external solutions (such as Auth0, AWS Cognito, Firebase, Okta, and others) provide robust and industrial strength security and features, your project might need a custom solution in which the ownership of data is under total control.

In these cases, it is important that you weigh up your options carefully, and who knows – maybe you will end up having to write your own authentication. Not all apps are made for banking, after all!

Summary

In this chapter, we have seen a very basic but quite representative implementation of an authentication mechanism. We have seen how FastAPI enables us to use standard-compliant authentication methods and we implemented one of the simplest possible yet effective solutions.

We have learned how elegant and flexible FastAPI and MongoDB are when it comes to defining granular roles and permissions, with the aid of Pydantic as the middleman. This chapter was focused exclusively on JWT tokens as the means of communication because it is the primary and most popular tool in single-page applications nowadays, and it enables great connectivity between services or microservices.

Finally, we created a simple React application and implemented a login mechanism that stores the user data in the state in memory. I have chosen not to show any solution of persisting the JWT token on purpose – the idea is just to see how a React application behaves with authenticated users and with those who are not. Using both localstorage and cookies has its pros and vulnerabilities (localstorage more so), but they both might be viable solutions for an application that has very light security requirements.

It is important to emphasize again that the FARM stack can be a great prototyping tool, so knowing your way around when creating an authentication flow, even if it is not ideal or absolutely bulletproof, might be just good enough to get you over that MVP hump in the race for the next great data-driven product! In the next chapter, we will see how we can integrate our MongoDB and FastAPI-based backend with a robust React framework – Next.js – and we will cover some standard web development tasks such as image and file uploads, authentication with `httpOnly` cookies, simple data visualizations, sending emails, and taking advantage of the flexibility of the stack.