

```
async get_user_details(user: User = Depends(get_user)) :
    return user
```

The route function here is dependent on the `get_user` function, which serves as its dependency. What this means is that to access the preceding route, the `get_user` dependency must be satisfied.

The `Depends` class, which is imported from the FastAPI library, is responsible for taking the function passed as the argument and executing it when the endpoint is called, automatically making available to the endpoint, the return value of the function passed to it.

Now that you have an idea of how a dependency is created and how it's used, let's build the authentication dependency for the event planner application.

Securing the application with OAuth2 and JWT

In this section, we'll build out the authentication system for the event planner application. We'll be making use of the OAuth2 password flow, which requires the client to send a username and password as form data. The username in our case is the email used when creating an account.

When the form data is sent to the server from the client, an **access token**, which is a signed JWT, is sent as a response. Usually, a background check is done to validate the credentials sent to the server before creating a token to allow further authorization. To authorize the authenticated user, the JWT is prefixed with Bearer when sent via the header to authorize the action on the server.

What Is a JWT and Why Is It Signed?

A JWT is an encoded string usually containing a dictionary housing a payload, a signature, and its algorithm. JWTs are signed using a unique key known only to the server and client to avoid the encoded string being tampered with by an external body.



Figure 7.1 – Authentication flow

Now that we have an idea of how the authentication flow works, let's create the necessary folder and files required to set up an authentication system in our application:

1. In the project folder, create the `auth` folder first:

```
(venv)$ mkdir auth
```

2. Next, create the following files in the `auth` folder:

```
(venv)$ cd auth && touch {__init__,jwt_  
handler,authenticate,hash_password}.py
```

The preceding command creates four files:

- `jwt_handler.py`: This file will contain the functions required to encode and decode the JWT strings.
- `authenticate.py`: This file will contain the `authenticate` dependency, which will be injected into our routes to enforce authentication and authorization.
- `hash_password.py`: This file will contain the functions that will be used to encrypt the password of a user during sign-up and compare passwords during sign-in.
- `__init__.py`: This file indicates the contents of the folder as a module.

Now that the files have been created, let's build the individual components. We'll start by creating the components for hashing user passwords.

Hashing passwords

In the previous chapter, we stored user passwords in plain text. This is a highly insecure and prohibited practice when building APIs. Passwords are to be encrypted or hashed using appropriate libraries. We'll be encrypting the user passwords using `bcrypt`.

Let's install the `passlib` library. This library houses the `bcrypt` hashing algorithm, which we'll be using for hashing user passwords:

```
(venv)$ pip install passlib[bcrypt]
```

Now that we have installed the library, let's create the functions for hashing the passwords in `hash_password.py`:

```
from passlib.context import CryptContext  
  
pwd_context = CryptContext(schemes=["bcrypt"],
```

```
deprecated="auto")

class HashPassword:
    def create_hash(self, password: str):
        return pwd_context.hash(password)

    def verify_hash(self, plain_password: str,
hashed_password: str):
        return pwd_context.verify(plain_password,
hashed_password)
```

In the preceding code block, we start by importing `CryptContext`, which takes the `bcrypt` scheme for hashing the strings passed to it. The context is stored in the `pwd_context` variable, giving us access to the methods required for executing our task.

The `HashPassword` class is then defined and contains two methods, `create_hash` and `verify_hash`:

- The `create_hash` method takes a string and returns the hashed value.
- `verify_hash` takes the plain password and the hashed password and compares them. The function returns a Boolean value indicating whether the values passed are the same or not.

Now that we have created a class to handle the hashing of passwords, let's update the sign-up route to hash the user password before storing it in the database:

routes/users.py

```
from auth.hash_password import HashPassword
from database.connection import Database

user_database = Database(User)
hash_password = HashPassword()

@user_router.post("/signup")
async def sign_user_up(user: User) -> dict:
    user_exist = await User.find_one(User.email ==
```

```

user.email)

if user_exist:
    raise HTTPException(
        status_code=status.HTTP_409_CONFLICT,
        detail="User with email provided exists
        already."
    )

hashed_password = hash_password.create_hash(
user.password)
user.password = hashed_password
await user_database.save(user)
return {
    "message": "User created successfully"
}

```

Now that we have updated the user sign-up route to hash the password before saving, let's create a new user to confirm. In a terminal window, start the application:

```

(venv)$ python main.py
INFO:      Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C
to quit)
INFO:      Started reloader process [8144] using statreload
INFO:      Started server process [8147]
INFO:      Waiting for application startup.
INFO:      Application startup complete.

```

In another terminal window, start the MongoDB instance:

```
$ mongod --dbpath database --port 27017
```

Next, let's create a new user:

```

$ curl -X 'POST' \
'http://0.0.0.0:8080/user/signup' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "email": "reader@packt.com",

```

```
"password": "exemplary"
},'
```

We get a success response from the request above:

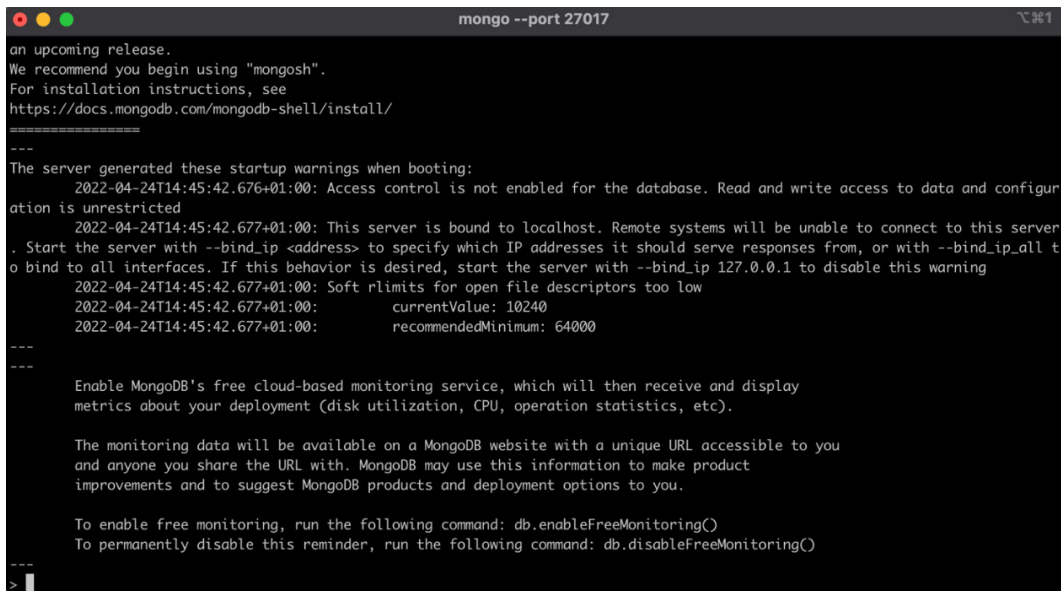
```
{
  "message": "User created successfully"
}
```

Now that we have created a user, let's verify that the password sent to the database was hashed. To do that, we'll create an interactive MongoDB session that allows us to run commands from within the database.

In a new terminal window, run the following commands:

```
$ mongo --port 27017
```

An interactive MongoDB session is started:



```
an upcoming release.
We recommend you begin using "mongosh".
For installation instructions, see
https://docs.mongodb.com/mongodb-shell/install/
=====
---
The server generated these startup warnings when booting:
  2022-04-24T14:45:42.676+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
  2022-04-24T14:45:42.677+01:00: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
  2022-04-24T14:45:42.677+01:00: Soft rlimits for open file descriptors too low
  2022-04-24T14:45:42.677+01:00:           currentValue: 10240
  2022-04-24T14:45:42.677+01:00:           recommendedMinimum: 64000
---
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Figure 7.2 – An interactive MongoDB session

With the interactive session running, run the series of commands to switch to the planner database and retrieve all user records:

```
> use planner
> db.users.find({})
```