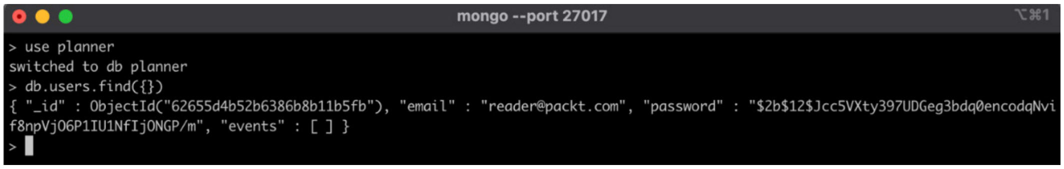


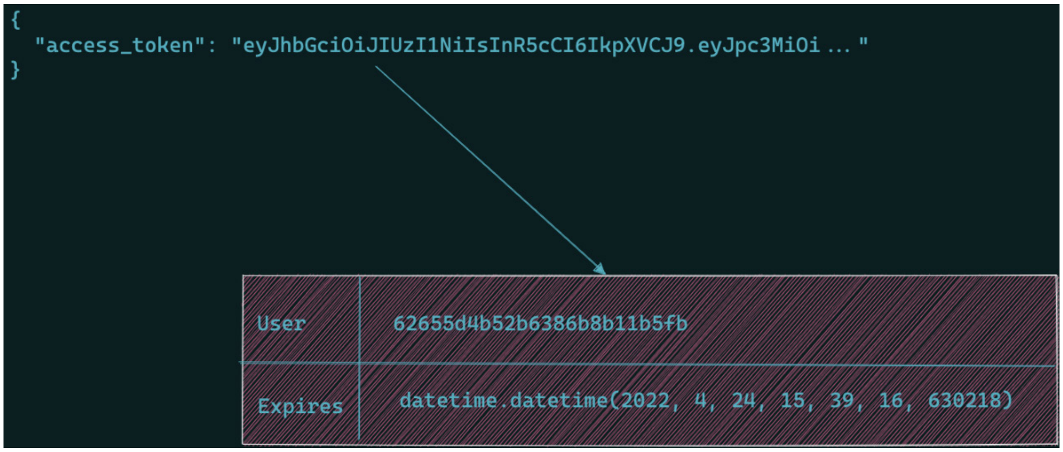
```
> use planner
switched to db planner
> db.users.find({})
{ "_id" : ObjectId("62655d4b52b6386b8b11b5fb"), "email" : "reader@packt.com", "password" : "$2b$12$Jcc5VXty397UDGeg3bdq0encodqNvi
f8npVjO6P1IU1NfIjONGP/m", "events" : [ ] }
> 
```

Figure 7.3 – Result from the find all users query

The preceding command returns the list of users, and we can now confirm that the user's password was hashed before it was stored in the database. Now that we have successfully built the components for securely storing user passwords, let's build the components for creating and verifying JWTS.

Creating and verifying access tokens

Creating a **JWT takes us** a step closer to securing our application. The token's payload will comprise the user ID and an expiry time before encoding in the long string as shown here:



```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOi ... "
}
```

Figure 7.4 – Anatomy of a JWT

Earlier on, we learned why JWTs are signed. JWTs are signed with a secret key known only to the sender and the receiver. Let's update the `Settings` class in `database/database.py` as well as the environment file, `.env`, to include a `SECRET_KEY` variable, which will be used to sign the JWTs:

database/database.py

```
class Settings(BaseSettings):
    SECRET_KEY: Optional[str] = None
```

.env

```
SECRET_KEY=HI5HL3V3L$3CR3T
```

With that in place, add the following imports in `jwt_handler.py`:

```
import time
from datetime import datetime

from fastapi import HTTPException, status
from jose import jwt, JWTError
from database.database import Settings
```

In the preceding code block, we have imported the `time` modules, the `HTTPException` class, as well as the status from FastAPI. We also imported the `jose` library responsible for encoding and decoding JWTs and the `Settings` class.

Next, we'll create an instance of the `Settings` class so we can retrieve the `SECRET_KEY` variable and create the function responsible for creating the token:

```
settings = Settings()

def create_access_token(user: str) -> str:
    payload = {
        "user": user,
        "expires": time.time() + 3600
    }

    token = jwt.encode(payload, settings.SECRET_KEY,
    algorithm="HS256")
    return token
```

In the preceding code block, the function takes a string argument, which is passed into the `payload` dictionary. The `payload` dictionary contains the user and the expiry time, which is returned when a JWT is decoded.

The `expires` value is set to an hour from the time of creation. The payload is then passed to the `encode()` method, which takes three parameters:

- **Payload**: A dictionary containing the values to be encoded.
- **Key**: The key used to sign the payload.
- **Algorithm**: The algorithm used in signing the payload. The default and most common is the **HS256** algorithm.

Next, let's create a function to verify the authenticity of a token sent to our application:

```
def verify_access_token(token: str) -> dict:
    try:
        data = jwt.decode(token, settings.SECRET_KEY,
        algorithms=["HS256"])

        expire = data.get("expires")

        if expire is None:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="No access token supplied"
            )
        if datetime.utcnow() >
        datetime.utcfromtimestamp(expire):
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Token expired!"
            )
        return data

    except JWTError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Invalid token"
        )
```

In the preceding code block, the function takes the token string as the argument and runs several checks in the `try` block. The function first checks the expiry time of the token. If there's no expiry time, then no token was supplied. The second check is the validity of the token – an exception is thrown to inform the user of the token expiration. If the token is valid, the decoded payload is returned.

In the `except` block, a bad request exception is thrown for any JWT error.

Now that we have implemented the functions for creating and verifying the tokens sent to the application, let's create the function that validates user authentication and serves as the dependency.

Handling user authentication

We have successfully implemented the components for hashing and comparing passwords as well as components for creating and decoding JWTs. Let's implement the dependency function that will be injected into the event routes. This function will serve as the single source of truth for retrieving a user for an active session.

In `auth/authenticate.py`, add the following:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

from auth.jwt_handler import verify_access_token

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/user/signin")

async def authenticate(token: str = Depends(oauth2_scheme)) ->
str:
    if not token:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Sign in for access"
        )

    decoded_token = verify_access_token(token)
    return decoded_token["user"]
```

In the preceding code block, we start by importing the necessary dependencies:

- `Depends`: This injects `oauth2_scheme` into the function as a dependency.
- `OAuth2PasswordBearer`: This class tells the application that a security scheme is present.
- `verify_access_token`: This function, defined in the creating and verifying access token section will be used to check the validity of the token.

We then define the token URL for the OAuth2 scheme and the `authenticate` function. The `authenticate` function takes the token as the argument. The function has the OAuth scheme injected into it as a dependency. The token is decoded, and the user field of the payload is returned if the token is valid, otherwise, the adequate error responses are returned as defined in the `verify_access_token` function.

Now that we have successfully created the dependency for securing the routes, let's update the authentication flow in the routes, as well as injecting the `authenticate` function into the event routes.

Updating the application

In this section, we'll update the routes to use the new authentication model. Lastly, we'll update the POST route for adding an event to populate the events field in the user's record.

Updating the user sign-in route

In `routes/users.py`, update the imports:

```
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
from auth.jwt_handler import create_access_token

from models.users import User
```

We have imported the `OAuth2PasswordRequestForm` class from FastAPI's security module. This will be injected into the sign-in route to retrieve the credentials sent over: username and password. Let's update the `sign_user_in()` route function:

```
async def sign_user_in(user: OAuth2PasswordRequestForm =
Depends()) -> dict:
    user_exist = await User.find_one(User.email ==
```