BookMyShow - User API

- BookMyShow - User API
    - Serialisers
    - Views
        - Create a user
        - Read, update and delete a user
        - Register the URLs
        - Testing the API
        - Modifying the response
    - Password Hashing
        - Why is Password Hashing Done?
        - Salt
        - Bcrypt
        - Using Bcrypt in Django
    - Conclusion

Serialisers

In our previous session, we created all the models for BookMyShow using Django ORM. Now, we will create the User API using Django REST Framework. To start with, we will create the serialisers for the User model. Serialisers are used to convert the data from the model into a format that can be sent over the network.

Serialisation

Serialisation and deserialisation are common terms used when data is sent over the network. Serialisation is the process of converting data into a format that can be sent over the network. Deserialisation is the process of converting the data back into the objects that can be used by the application. Another set of terms for the same are marshalling and unmarshalling.

Create a new file `serialisers.py` in the `users` app. Add the following code to it:

```
from rest_framework import serializers

class UserSerialiser(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = '__all__'
```

Here, we are creating a serialiser for the `User` model:

- We are using the `ModelSerializer` class provided by Django REST Framework.
- We are specifying the model that we want to serialise and the fields that we want to include in the serialised data.
- In this case, we are including all the fields.

Views

Views are the functions that are called when a request is made to the API. They are similar to the controller layer.
The request comes to the view, the view processes the request, queries the model, serialises the data and sends it back as a response.
Django REST Framework provides a class called `ModelViewSet` that can be used to create views for the models among other classes.

Create a user

Let's start with creating a view to create a user. Add the following code to `views.py`:

```
from rest_framework import generics

class UserListCreateView(generics.ListCreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Here, we are creating a view called `UserListCreateView` that inherits from `ListCreateAPIView`. This class provides the following functionalities:

- It handles creating a new user when a `POST` request is made to the API.
- Listing all the users when a `GET` request is made to the API.

Read, update and delete a user

Similar to the `ListCreateAPIView`, Django REST Framework provides a class called `RetrieveUpdateDestroyAPIView` that can be used to create views for the models to handle URLs that contain the primary key of the model. Add the following code to `views.py`:

```
from rest_framework import generics

class UserRetrieveUpdateDestroyView(generics.RetrieveUpdateDestroyAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Register the URLs

Now, we need to register the URLs for the views that we created. Add the following code to `urls.py`:

```
from django.urls import path

urlpatterns = [
    path('user/', UserListCreateView.as_view(), name='user-list-create'),
    path('user/<int:pk>/', UserRetrieveUpdateDestroyView.as_view(), name='user-retrieve-u
]
```

This sets up two endpoints:

- `/users/`: Handles creating new users (POST) and listing all users (GET).
- `/users/<id>/`: Handles retrieving (GET), updating (PUT/PATCH), and deleting (DELETE) a specific user.

Testing the API

Now, we can test the API using Postman. Start the server using `python manage.py runserver`. Open Postman and create a new request with the following details:

- URL: `http://localhost:8000/user/`
- Request type: `POST`
- Body: `raw` - `JSON`
- Body content:

```
{
    "name": "Tantia Tope",
    "email": "t@t.com",
    "password": "1857"
}
```

Click on `Send`. You should see the following response:

```
{
    "id": 1,
    "name": "Tantia Tope",
    "email": "t@t.com",
    "password": "1857"
}
```

Now, let's try to retrieve the user that we just created. Create a new request with the following details:

- URL: `http://localhost:8000/user/1/`
- Request type: `GET`
- Click on `Send`. You should see the following response:

```
{
    "id": 1,
    "name": "Tantia Tope",
    "email": "t@t.com",
    "password": "1857"
}
```

Modifying the response

We wouldn't want to send the password back to the user. Let's modify the response to remove the password. Add the following code to `serialisers.py`:

```
from rest_framework import serializers

class AllUserFieldsSerialiser(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = '__all__'

class UserNoPasswordSerialiser(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'name', 'email']
```

You also have to override the `get_serializer_class` method in the views to specify which serialiser to use. Add the following code to `views.py`:

```
class UserListCreateView(generics.ListCreateAPIView):
    queryset = User.objects.all()

    def get_serializer_class(self):
        if self.request.method == "POST":
            return AllUserFieldsSerialiser
        return UserNoPasswordSerialiser
```

Now if you try to retrieve the user, you won't see the password in the response.

Password Hashing

Password hashing is a security practice used to protect user passwords by converting them into a hashed, irreversible form. The purpose is to prevent attackers from easily retrieving the original passwords even if they gain access to the hashed values. Hashing is commonly used in authentication systems to store and verify passwords securely.

Why is Password Hashing Done?

The primary reasons for password hashing are:

1. **Security:** Hashing adds a layer of security by making it computationally infeasible for attackers to reverse the process and obtain the original password from the hashed value.
2. **Protection Against Data Breaches:** In the event of a data breach, if the stored passwords are hashed, even if the hashed values are exposed, attackers cannot directly use them to access user accounts.
3. **Uniformity:** Hashing provides a standardized representation for passwords, making it easier to manage and compare password data.

Salt

A salt is a random value that is unique for each user. It is combined with the user's password before hashing. Salting is done to address the vulnerability of hash tables and rainbow table attacks.

- **Hash Tables:** Attackers can use precomputed tables (rainbow tables) to quickly look up the hash of a common password. Salting prevents this by ensuring that even if two users have the same password, their hashes will be different due to unique salts.
- **Rainbow Tables:** These are precomputed tables of hash values for a large set of possible passwords. Adding a salt makes it impractical to create comprehensive rainbow tables because each user's salt requires a separate table.

Bcrypt

Bcrypt is a key derivation function designed specifically for password hashing. It incorporates a work factor, which adjusts the computational cost of the hashing process. Bcrypt is considered more secure than traditional hashing algorithms like MD5 or SHA-1 because:

1. **Work Factor:** Bcrypt allows you to adjust the number of iterations (work factor), making it computationally expensive and slowing down brute-force attacks.
2. **Salt Inclusion:** Bcrypt automatically generates and manages salts for each password, eliminating the need for manual salt management.

3. **Adaptability:** As computing power increases, the work factor can be easily adjusted to maintain a sufficient level of security.

When using Bcrypt in Django, the `make_password` function from `django.contrib.auth.hashers` automatically generates a salt and hashes the password with the Bcrypt algorithm. The `check_password` function is used to verify a password against its hashed value.

In summary, password hashing, the use of salts, and the choice of a secure hashing algorithm like Bcrypt are essential components of good password security practices. They protect user credentials and contribute to the overall security of authentication systems.

Using Bcrypt in Django

Django provides a `make_password` function that can be used to hash the password. Add the following code to `views.py`:

```
from django.contrib.auth.hashers import make_password

class UserListCreateView(generics.ListCreateAPIView):
    queryset = User.objects.all()

    def get_serializer_class(self):
        if self.request.method == "POST":
            return AllUserFieldsSerialiser
        return UserNoPasswordSerialiser

    def perform_create(self, serializer):
        serializer.save(password=make_password(serializer.validated_data['password']))
```

Here, we are overriding the `perform_create` method to hash the password before saving the user. Now, if you try to create a user, you will see that the password is hashed in the database.

In the `settings.py` file, add the following code:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
]
```

You might have to install `bcrypt` using `pip install bcrypt`.
This will ensure that the password is hashed using Bcrypt.

Conclusion

In this session, we created the User API using Django REST Framework. We also learnt about password hashing and how to use Bcrypt in Django. In the next session, we will create the Booking API.