Authentication

1. Implementing User Authentication with `passport-firebase-jwt`

Step 1: Set Up Firebase Admin SDK

First, install `firebase-admin` and `passport-firebase-jwt`

```
npm install firebase-admin passport passport-firebase-jwt
```

initialize the Firebase Admin SDK.

download the service account key from Firebase Console (Project Settings → Service Accounts) and place it securely in your project.

```
// firebase/firebase.service.ts
import * as admin from 'firebase-admin';

admin.initializeApp({
  credential: admin.credential.cert(require('./path/to/ser
viceAccountKey.json')),
});

export default admin;
```

Step 2: Create the Firebase JWT Strategy

set up a custom **Passport strategy** using `passport-firebase-jwt`.

This will allow Firebase ID tokens (issued by Firebase Authentication) to be verified by your NestJS server.

```
// auth/strategies/firebase-jwt.strategy.ts
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy, ExtractJwt } from 'passport-firebase-jwt';
import * as admin from '../../firebase/firebase.service';

@Injectable()
export class FirebaseJwtStrategy extends PassportStrategy(Strategy, 'firebase-jwt') {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    });
  }

  async validate(token: string) {
    try {
      // Verify the token with Firebase Admin
      const decodedToken = await admin.auth().verifyIdToken(token);
      return decodedToken; // Attach the decoded token to request.user
    } catch (error) {
      throw new UnauthorizedException('Invalid Firebase ID token');
    }
  }
}
```

- `ExtractJwt.fromAuthHeaderAsBearerToken()` extracts the JWT from the `Authorization` header.
- `verifyIdToken` uses Firebase Admin SDK to verify the ID token, ensuring it's issued by Firebase.

Step 3: Configure the Firebase JWT Strategy in the Auth Module

register the `FirebaseJwtStrategy` in your `AuthModule` to make it available across your application.

```
// auth/auth.module.ts
import { Module } from '@nestjs/common';
import { FirebaseJwtStrategy } from './strategies/firebase
-jwt.strategy';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports: [PassportModule],
  providers: [FirebaseJwtStrategy],
})
export class AuthModule {}
```

Step 4: Protect Routes with Firebase JWT Authentication

With `FirebaseJwtStrategy` configured, we can now use it to protect routes. Set up a route that requires authentication in your `AuthController` by using the `@UseGuards(AuthGuard('firebase-jwt'))` decorator.

```
// auth/auth.controller.ts
import { Controller, Get, UseGuards } from '@nestjs/commo
n';
import { AuthGuard } from '@nestjs/passport';

@Controller('auth')
export class AuthController {
  // Protected route
  @UseGuards(AuthGuard('firebase-jwt'))
  @Get('profile')
  getProfile(@Req() req) {
    return req.user; // req.user contains the decoded Fire
base user data
```

```
  }
}
```

This `getProfile` route will now be accessible only if the request contains a valid Firebase ID token in the `Authorization` header.

Step 5: Testing the Authentication Flow

To test the setup:

1. Obtain an ID token by logging in on the client side (e.g., in a mobile app or website) using Firebase Authentication.
2. Include the ID token in the `Authorization` header (format: `Bearer <id_token>`) when making requests to your NestJS backend.

Using Firebase Custom Tokens

Firebase **Custom Tokens** are useful in cases where:

1. You have a custom or enterprise authentication system and want to integrate it with Firebase.
2. You want to allow users authenticated through an external system (like LDAP or a third-party OAuth provider) to interact with Firebase.

Generating a Custom Token

To create a custom token in NestJS, use the Firebase Admin SDK's `createCustomToken` method:

```
// auth/auth.service.ts
import { Injectable } from '@nestjs/common';
import * as admin from '../firebase/firebase.service';

@Injectable()
export class AuthService {
  async generateCustomToken(uid: string): Promise<string>
{
    try {
      const customToken = await admin.auth().createCustomT
oken(uid);
      return customToken;
    } catch (error) {
      throw new Error('Error generating custom token');
    }
  }
}
```

Once generated, send the custom token to the client. On the client side, they can use Firebase's `signInWithCustomToken` method to authenticate:

```
// Client-side code
import { getAuth, signInWithCustomToken } from 'firebase/a
uth';

const auth = getAuth();
signInWithCustomToken(auth, customToken)
  .then((userCredential) => {
    console.log('User signed in:', userCredential.user);
  })
  .catch((error) => {
    console.error('Error signing in with custom token:', e
rror);
  });
```

Choosing Between ID Tokens and Custom Tokens

- **ID Tokens**: Use these if you're using Firebase's built-in providers (email/password, Google, Facebook, etc.). They're issued automatically when a user signs in through Firebase.
- **Custom Tokens**: Use these if you're integrating an external authentication system or need to create server-side users who need access to Firebase services.

https://github.com/starcharles/passport-firebase?tab=readme-ov-file https://stackoverflow.com/questions/58788289/firebase-how-to-generate-access-token-from-username-and-password-on-the-server

https://stackoverflow.com/questions/50192454/firebase-acces-and-id-tokens

https://firebase.google.com/docs/auth/admin/verify-id-tokens