JWT Authentication Tutorial – Fullstack App (ASP.NET Core + Flutter)

This tutorial explains how JWT-based authentication works in a fullstack app using an ASP.NET Core backend and a Flutter frontend. It covers how users register or log in, how JWT tokens are generated and validated, and how protected data is accessed securely using those tokens.

You'll learn:

- How the backend (AuthController) handles secure login and registration.
- How JWT tokens are created using user identity and role.
- How the frontend (Flutter) saves and reuses tokens for authenticated requests.
- How protected endpoints are tested and how user data is displayed using the stored token.

This guide breaks down the key classes and methods involved in the process, showing both the purpose and functionality of each piece. By the end, you'll have a clear understanding of how user sessions are created, stored, and used securely in your app—from the moment a user types in their password to when they see personalized content on the homepage.

`AuthController.cs`

Class Overview

Handles authentication logic: login and register. Uses dependency injection for database access (`MyContext`), config values (`IConfiguration`), and password hashing (`IPasswordHasher<UserModel>`).

`Login` Method

```
[HttpPost("login")]
public IActionResult Login([FromBody] LoginRequest request)
```

What it does:

- Validates input.
- Looks up the user in the database.
- Verifies the password using a hashed comparison.
- Retrieves the user's role.
- Creates a JWT token containing user claims.
- Returns the token and basic user info to the client.

Why it's important:
Allows existing users to securely log in and access protected backend routes using a JWT.

`Register` Method

```
[HttpPost("register")]
public IActionResult Register([FromBody] RegisterRequest model)
```

What it does:

- Validates that all required fields are provided.
- Checks if the username already exists in the database.
- Fetches the corresponding role from the `Role` table.
- Hashes the password and saves a new `UserModel` to the database.
- Generates a JWT token with the new user's claims.

- Returns the token and user info so the frontend can treat this as an automatic login.

Why it's important:
Creates a new user account while securing the password and immediately authenticates the user with a token.

`UserController.cs` – Tutorial Notes

Class Overview

Handles user-related operations like retrieving, creating, updating, and deleting users. Also includes a JWT-protected test route (`SayHello`).

`GetUsers()`

```
[HttpGet("[action]")]
public IActionResult GetUsers()
```

What it does:

- Retrieves all users from the database.

Why it matters:
Useful for admin views or debugging when needing to inspect all user records.

`GetUserById(long id)`

```
[HttpGet("[action]")]
public IActionResult GetUserById(long id)
```

What it does:

- Retrieves a user by their ID.
- Returns 404 if the user doesn't exist.

Why it matters:
Used when you want to view or edit details of a specific user.

`CreateUser(UserModel user)`

```
[HttpPost("[action]")]
public IActionResult CreateUser([FromBody] UserModel user)
```

What it does:

- Adds a new user to the database.
- Checks for duplicates based on username.

Why it matters:
Lets you insert users directly (outside the auth flow). Could be used by an admin interface.

`UpdateUser(long id, UserModel model)`

```
[HttpPut("[action]")]
public IActionResult UpdateUser(long id, [FromBody] UserModel model)
```

What it does:

- Updates user fields (username, email, etc.) based on their ID.
- Ensures the ID in URL matches the one in the request body.

Why it matters:
Supports profile editing or admin updates. This does not re-hash the password—be careful with that.

`DeleteUser(long id)`

```
[HttpDelete("[action]")]
public IActionResult DeleteUser(long id)
```

What it does:

- Finds a user by ID and removes them from the database.

Why it matters:
Provides backend support for user account deletion (either by admin or user self-deletion flow).

`SayHello()`

```
[Authorize]
[HttpGet("hello")]
public IActionResult SayHello()
```

What it does:

- Returns a simple message, but only if the request includes a valid JWT token.

Why it matters:
This is a protected route used to confirm that authentication is working correctly.

`MyContext.cs`

csharp
CopyEdit

```
using Comp375BackEnd.Models;
```

```
using Microsoft.EntityFrameworkCore;

namespace Comp375BackEnd.Data
{
    public class MyContext : DbContext
    {
        public MyContext(DbContextOptions<MyContext> options) :
base(options) { }

        public DbSet<UserModel> User { get; set; }
        public DbSet<RoleModel> Role { get; set; }
    }
}
```

Why: This class is needed to configure how your app connects to and interacts with the database.

What: It extends `DbContext` and exposes two tables, `User` and `Role`, so Entity Framework can use them for querying and saving.

How: When injected into controllers, it allows code to access and manipulate user and role data without writing raw SQL.

UserModel.cs

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Comp375BackEnd.Models
{
    [Table("User")]
    public class UserModel
    {
        [Key]
        public long UserId { get; set; }
        public string? Username { get; set; }
```

```
        public string? PasswordHash { get; set; }
        public string? Email { get; set; }
        public string? PhoneNumber { get; set; }
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
        public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;

        [ForeignKey("Role")]
        public long? RoleId { get; set; }
    }
}
```

Why: This class defines the structure of a user in your database.

What: It includes properties like `Username`, `PasswordHash`, and `RoleId`, and links to the `Role` table via a foreign key.

How: Entity Framework uses these properties to build and manage the `User` table and its relationships at runtime.

RoleModel.cs

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Comp375BackEnd.Models
{
    [Table("Role")]
    public class RoleModel
    {
        [Key]
        public long RoleId { get; set; }
        public string Name { get; set; }
    }
}
```

Why: The role model helps classify users by access level or function, like Admin or Guest.

What: It maps to the `Role` table and holds a unique ID and role name.

How: When users are created, they reference a `RoleId`, which links back to this model for role-based logic and JWT claims.

LoginRequest.cs

```
namespace Comp375BackEnd.Models.Auth
{
    public class LoginRequest
    {
        public string Username { get; set; } = string.Empty;
        public string Password { get; set; } = string.Empty;
    }
}
```

Why: This class is used to receive login data from the frontend in a structured way.

What: It holds the `Username` and `Password` properties that are passed into the login endpoint.

How: When the login API is called, the JSON body is automatically deserialized into this class for validation and authentication.

RegisterRequest.cs

```
namespace Comp375BackEnd.Models.Auth
{
    public class RegisterRequest
    {
        public string Username { get; set; } = string.Empty;
        public string Password { get; set; } = string.Empty;
        public string Role { get; set; } = "User";
    }
}
```

Why: This class structures the data sent from the frontend when creating a new account.

What: It captures the username, password, and role for registration and ensures required fields are present.

How: The backend binds this class from the request body and uses it to validate, create, and store the new user securely.

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=CS-10\\SQLEXPRESS1;Initial
Catalog=MovieDB;Integrated Security=True;Connect
Timeout=30;Encrypt=True;Trust Server Certificate=True;Application
Intent=ReadWrite;Multi Subnet Failover=False"
  },
  "Jwt": {
    "Key": "a-string-secret-at-least-256-bits-long",
    "Issuer": "MovieRentalAPI",
    "Audience": "MovieRentalClient"
  }
}
```

Why: This configuration file stores essential app settings like the database connection and JWT security keys.

What: The `"ConnectionStrings"` section defines how to connect to your SQL Server database. The `"Jwt"` section stores the signing key, issuer, and audience values for generating and validating tokens.

How: These values are injected into the app through `IConfiguration` and are used in places like `MyContext`, `AddAuthentication()`, and your `AuthController` to enforce secure login/register logic.

`LoginPage` – Handles User Authentication

Purpose:
This class presents a login form and handles user authentication by verifying credentials against the backend. If login succeeds, it navigates to the `HomePage`.

Key Methods:

- `_login()`
  This function starts by showing a loading indicator. It sends the username and password to the backend via `AuthService.login`. If the backend returns a 200 response with a valid token, it stores the token, username, and role in `SharedPreferences`.
  The method also uses a double-delay system to confirm whether navigation to `HomePage` occurred. If not, it throws a user-facing error. This guards against rare edge cases where the app might silently fail to navigate.

- `build()`
  The UI collects username and password from the user and triggers `_login` when the user taps the Sign In button. Navigation to the Register page is also handled here.

Why It Matters:
This page connects frontend credentials with backend authentication and manages token storage, enabling secure session continuity throughout the app.

`RegisterPage` – Handles New Account Creation

Purpose:
This class allows new users to register. It sends the credentials and role selection to the backend, receives a token, and logs the user in immediately.

Key Methods:

- `_register()`
  This method works similarly to `_login`, but instead uses `AuthService.register`. After receiving a success response, it stores the returned user details and token in

`SharedPreferences`. It navigates to the `HomePage` just like login, with the same navigation safeguard. If the user already exists, it shows a snackbar message with the backend's error.

- `build()`
  Presents form fields for username, password, and role selection. Also includes basic UI feedback and connects to `LoginPage` for users who already have an account.

Why It Matters:
This method mirrors the login logic, demonstrating how new users are registered and authenticated seamlessly in one flow using the same JWT pipeline.

`ProfileView` – Retrieves and Displays Auth Data

Purpose:
This screen shows the current user's username and role stored locally, as well as a logout option.

Key Methods:

- `_loadProfile()`
  Retrieves the `username` and `role` from `SharedPreferences`, which were previously saved by either login or register methods. This confirms that local state is synchronized with the user session.

- `build()`
  Displays the stored username and role. Also contains a logout button that clears all stored JWT and user info, then returns to the login screen.

Why It Matters:
This verifies the authentication flow worked by checking that token-related data was saved and persists across app restarts. It's a lightweight user identity confirmation screen.

Summary of JWT Flow in Frontend

1. **Login/Register send credentials → backend.**

2. **Backend validates → returns JWT and user info.**
3. **Frontend saves JWT + user info with `SharedPreferences`.**
4. **User is redirected to `HomePage` with identity shown.**
5. **Other parts of the app (like ProfileView or protected endpoints) use the stored JWT to confirm session identity.**

HomePage Class (Flutter Frontend)

Purpose

This screen is the first page users see after logging in or registering. It welcomes them by name and includes a button to test whether the JWT token from the login was successfully stored and used to access protected backend endpoints.

Core JWT Logic & Backend Integration

`testProtectedEndpoint()` – Secure API Test

dart
CopyEdit

```
Future<void> testProtectedEndpoint() async {
  final prefs = await SharedPreferences.getInstance();
  final token = prefs.getString('jwt');

  final response = await http.get(
    Uri.parse('http://10.0.2.2:5000/user/hello'),
    headers: {
      'Authorization': 'Bearer $token',
      'Content-Type': 'application/json',
    },
  );

  if (response.statusCode == 200) {
    _showSnackBar("Backend says: ${response.body}");
  } else {
```

```
    _showSnackBar("Failed: ${response.statusCode}");
  }
}
```

- **Why:** This function tests your token by calling a secure route on the backend. If the token is valid, the backend responds.
- **How:** The token is retrieved from local storage (`SharedPreferences`) and sent via the `Authorization` header.
- **Importance:** Confirms that your frontend is properly authenticated and that the backend is validating the JWT token as expected.

`SharedPreferences.getString('jwt')`

dart
CopyEdit

```
final token = prefs.getString('jwt');
```

- **Why:** This loads the JWT token saved during login or registration.
- **How:** Uses Flutter's local storage system to persist the token.
- **Importance:** Without this line, the frontend can't access protected backend routes after login.

`widget.username`

dart
CopyEdit

```
Text(
  widget.username,
  style: TextStyle(
    color: Colors.white,
    fontSize: 40,
    fontWeight: FontWeight.bold,
  ),
```

```
),
```

- **Why:** Displays the user's name after login.
- **How:** The login/register screen passes this via the `HomePage(username: ...)` constructor.
- **Importance:** It's cosmetic but confirms to the user that they are successfully logged in.

UI & Structure (Trimmed to JWT-Relevant Parts)

Protected Route Test Button

dart
CopyEdit

```
ElevatedButton(
  onPressed: testProtectedEndpoint,
  style: ElevatedButton.styleFrom(backgroundColor: Colors.deepPurple),
  child: const Text("Test Protected Backend", style: TextStyle(color:
Colors.white)),
),
```

- Allows the user to test if the current session is authenticated by calling the `/user/hello` backend route.

Dark Mode Toggle and Logout

dart
CopyEdit

```
IconButton(
  icon: Icon(
    _darkMode ? Icons.dark_mode : Icons.light_mode,
    color: Colors.white,
  ),
  onPressed: () => setState(() => _darkMode = !_darkMode),
```

```
),

IconButton(
  icon: const Icon(Icons.logout, color: Colors.white),
  onPressed: () => Navigator.pop(context),
),
```

- Not related to JWT, but allows toggling UI mode and logging out (which simply navigates back and forgets the token).

Why It Matters

- `HomePage` is your **proof of login success**. If `testProtectedEndpoint()` works, your JWT is valid and accepted by the backend.
- If this call fails, it's the first place you should check (maybe token expired, not saved, or invalid).
- It's the bridge between **frontend identity** and **backend verification**, making it a critical checkpoint in your app's secure flow.

`pubspec.yaml`

Purpose:

This file manages dependencies, assets, and settings for your Flutter app. It tells Flutter which libraries to use, what assets to include, and how to build your app. Keeping it organized ensures your app compiles and runs properly across devices.

Code (key parts only):

```
name: sql_flutter_project
description: "A new Flutter project."
version: 1.0.0+1

environment:
```

```
  sdk: ^3.6.2
```

Purpose: These lines set the project name, description, version, and the Dart SDK version required.
Why it matters: This metadata is essential for project management and compatibility.
How it works: Flutter reads this info during `flutter build` or `flutter run` to configure the app version and supported SDK.

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.6
  provider: ^6.1.2
  flutter_slidable: ^2.0.0
  shared_preferences: ^2.2.2
  animated_text_kit: ^4.2.2
  cupertino_icons: ^1.0.8
```

Purpose: These are the libraries your app depends on.
Why it matters: You use `http` to talk to your backend, `shared_preferences` for storing tokens, and UI libraries like `animated_text_kit`.
How it works: Running `flutter pub get` fetches and installs these packages so you can import and use them in your Dart files.

---

yaml
CopyEdit

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^5.0.0
```

Purpose: These packages are only needed for testing and code linting.
Why it matters: Linting helps enforce best coding practices, and tests help ensure your app works as expected.
How it works: They run during development and test stages but are not included in the final app.

```
flutter:
  uses-material-design: true
  assets:
    -
assets/transparent-happy-young-hispanic-school-boy-with-thumbs-up-png.
webp
    - assets/stitch.gif
    - assets/profile.jpg
```

Purpose: This section declares design settings and assets used in the UI.
Why it matters: You need to register image assets here before using them in the app, or they won't show up.
How it works: Flutter loads these assets from the `assets/` directory and makes them available in the `Image.asset()` widget.

JWT Authentication Flow (Frontend + Backend)

---

1. User logs in or registers (Flutter UI)

LoginPage / RegisterPage

When the user submits their credentials:

dart
CopyEdit

```
final success = await AuthService().login(username, password);
// or
final success = await AuthService().register(username, password,
role);
```

This sends an HTTP request to your backend at `/auth/login` or `/auth/register`.

---

2. Flutter calls AuthService (HTTP POST)

dart
CopyEdit
```dart
final response = await http.post(
  Uri.parse('$baseUrl/Login'),
  headers: {'Content-Type': 'application/json'},
  body: jsonEncode({"username": username, "password": password}),
);
```

If successful, you get a response like:

json
CopyEdit
```json
{
  "token": "<JWT_TOKEN>",
  "user": {
    "userId": 1,
    "username": "Sofiia",
    "role": "User"
  }
}
```

The token is saved using:

dart
CopyEdit
```dart
prefs.setString('jwt', body['token']);
prefs.setString('username', body['user']['username']);
prefs.setString('role', body['user']['role']);
```

---

3. Backend verifies credentials and creates token

Inside your `AuthController.cs`:

Login:

csharp
CopyEdit
```csharp
var user = _context.User.FirstOrDefault(u => u.Username ==
request.Username);
```

```
var result = _passwordHasher.VerifyHashedPassword(user,
user.PasswordHash, request.Password);
```

If the password is valid, the backend generates a token:

csharp
CopyEdit

```
var claims = new List<Claim> {
  new Claim(ClaimTypes.Name, user.Username),
  new Claim(ClaimTypes.Role, roleName)
};
var token = new JwtSecurityToken(
  claims: claims,
  signingCredentials: creds,
  expires: DateTime.Now.AddHours(1)
);
```

---

4. Frontend redirects user to homepage

After saving the token, the user is redirected:

dart
CopyEdit

```
Navigator.pushReplacement(
  context,
  MaterialPageRoute(builder: (context) => HomePage(username:
_usernameController.text)),
);
```

---

5. HomePage loads user info

The `username` is passed in and shown:

dart
CopyEdit

```
Text(widget.username) // e.g. "Welcome back, Sofiia"
```

This does not contact the backend again, it just uses what was stored.

---

6. Protected endpoint test with token

The `Test Protected Backend` button sends a token-authenticated request:

dart
CopyEdit
```
final token = prefs.getString('jwt');
final response = await http.get(
  Uri.parse('http://10.0.2.2:5000/user/hello'),
  headers: {'Authorization': 'Bearer $token'},
);
```

Backend checks the token because the endpoint has:

csharp
CopyEdit
```
[Authorize]
public IActionResult SayHello() => Ok("Hello from a protected backend route!");
```

If the token is valid, the response shows up in a `SnackBar`.

---

7. Logout clears token and user info

In `ProfileView`, logout clears storage:

dart
CopyEdit
```
await prefs.clear();
Navigator.pushReplacement(context, MaterialPageRoute(builder: (_) => LoginPage()));
```

Why it works

- Tokens are generated by the backend and signed with a secret.
- The frontend stores and reuses the token for future authenticated requests.
- Middleware on the backend (`UseAuthentication`) checks each token.
- If the token is expired, invalid, or missing, the backend returns 401.