# **How the Logged-In User and BMI Records Are Connected in My Full-Stack App**

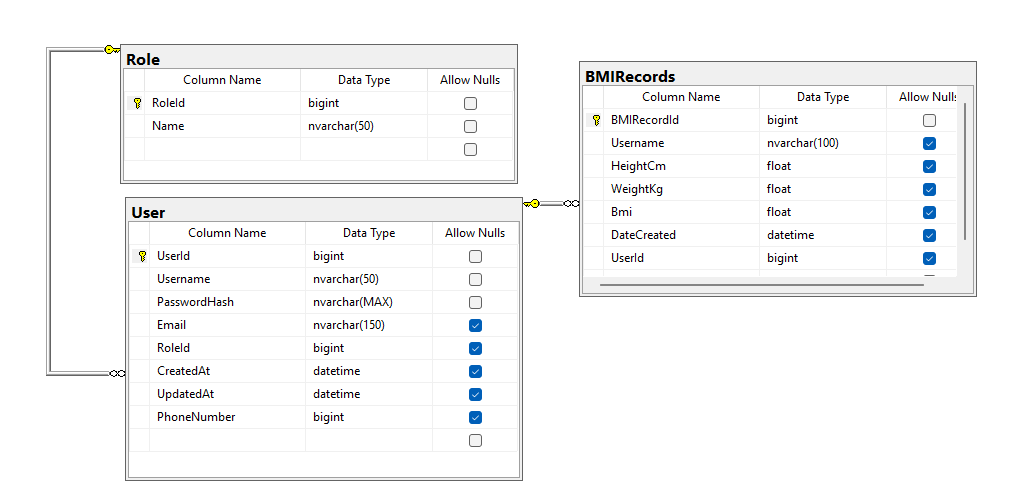
## Overview

Here’s the breakdown of how I got BMI records to connect to the right logged-in user in my app. I used:

* Flutter for the frontend (UI and logic)
* ASP.NET Core for the backend (API + authentication)
* SQL Server for the database (stores users and their BMI info)

The whole idea is that when someone logs in and adds a BMI, it’s saved to *their* account. No random data is floating around. Just personalized, trackable info.

This diagram shows how everything is connected — each BMI record has a UserId that links to the User table, and users have a RoleId too.



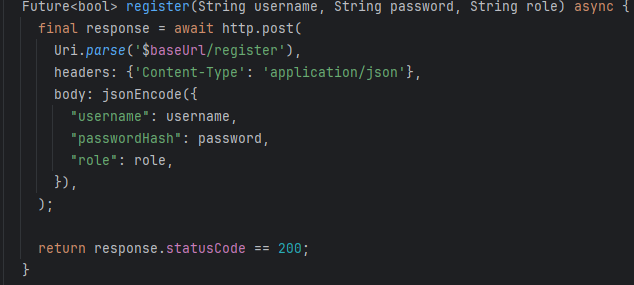
## Step 1: Registering a New User

### Flow

### The Flutter app sends the user’s info (like username and password) to the backend.

* The backend checks if the username already exists.
* If not, it hashes the password, creates the user in the database, and assigns a default role.
* Then it returns a JWT token, so the user is instantly logged in.

When someone registers, we’re not just storing a password directly — we hash it using BCrypt. That means even if someone broke into the database, they wouldn’t get plain passwords. I also chose to return a JWT right away after registration so the user is instantly logged in and doesn’t need to sign in again. That keeps the UX smooth.



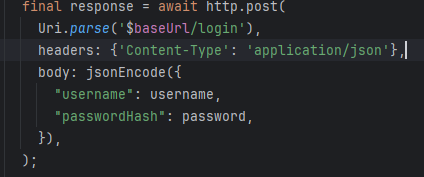
Why: This shows how data gets sent from Flutter to the backend — it’s the “how” behind the POST request.

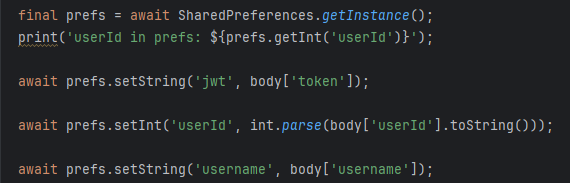
## Step 2. Logging In + Getting the Token

* User types in their credentials and taps login.
* The backend checks if the username exists and if the password is correct.
* If it all checks out, the backend gives back a JWT.
* Flutter stores that token to use later (like when saving BMI).

This login system uses the same logic — we check if the user exists, and then verify the password using BCrypt again. If it works, we build a JWT. The cool part is the token contains their UserId in the claims — that’s how we later figure out whose data is whose.

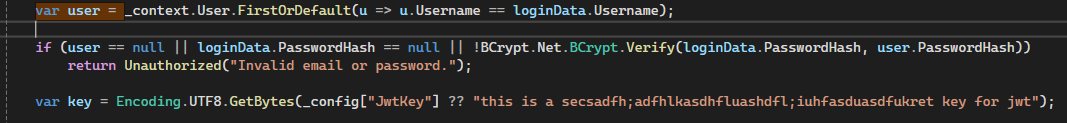
Flutter stores this JWT in SharedPreferences, which is like a little local drawer where I stash user info for future requests





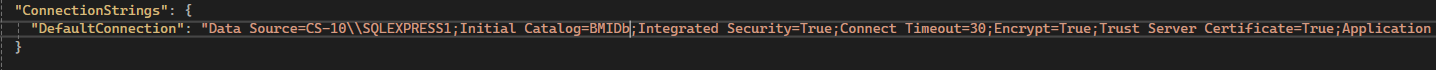
That chunk is inside your login() method — after getting a successful response from the backend. This is how you're storing the data locally so that you can access it later in other parts of the app (like when saving a BMI or checking who's logged in).

Why: This shows how the app sends the login request and stores the token + user info for later use.

Why: The token proves “this is me” every time the user wants to save or view data. Without it, the backend wouldn’t know who’s asking.

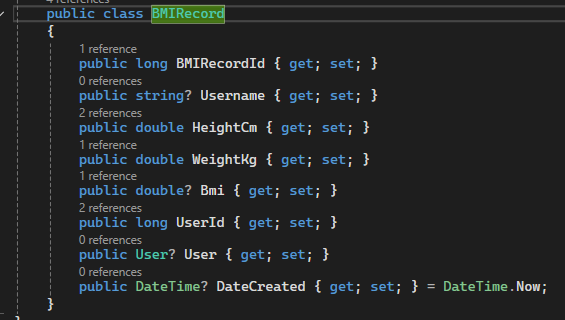
Program.cs:

This connects your app to the SQL Server database using the connection string from appsettings.json.

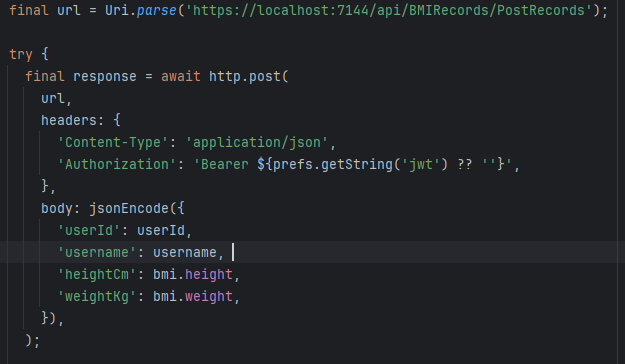


That’s the connection to your BMIDb, which has the User and BMIRecords tables.

In Flutter, we make a POST request to /api/BMIRecords. I send that JSON payload from the Flutter frontend, but in the .NET backend, it gets mapped to the BMIRecord class.



Inside bmi\_submit\_button.dart, the code grabs the values from the bmiController, builds a JSON object, and makes a POST request to the backend:

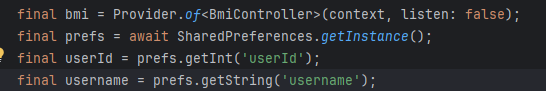
Why the Token?

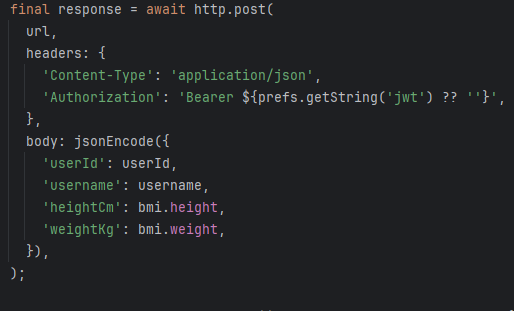
The Authorization: Bearer $token part is key — the backend reads that JWT token and pulls the UserId out of it. That’s how it knows which user to assign the BMI record to.

#### Why Do It This Way?

* It keeps records personalized to each logged-in user.
* No one can submit data for someone else.
* The backend stays secure and only accepts data from valid users.

**Submitting BMI Data (Frontend → Backend)**



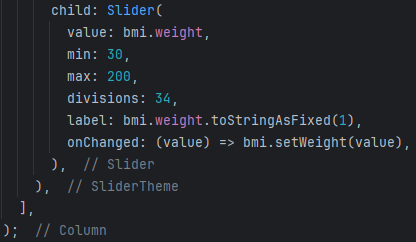


### **Why This Matters**

This is the exact moment the BMI gets submitted — tied to the logged-in user.

* **grabbing login info from storage** (SharedPreferences)
* **making a secure API call** to your .NET backend
* including the JWT for authorization, and userId so the backend knows who this record belongs to
* Flutter grabs the userId, username, and token from SharedPreferences
* These values were saved when the user logged in earlier
* A POST request is made to /api/BMIRecords/PostRecords with all BMI data and token
* The backend uses this to validate the user and store the BMI under the correct account

**BMI Input Screen – Height and Weight Sliders**



* I used two sliders to make height and weight selection more visual and user-friendly
* bmi\_controller.dart tracks the state with Provider, so everything updates instantly
* The label on the slider updates live to show exact values
* It feels smooth and gives feedback to the user while selecting

## **Displaying the BMI Result to the User**

After a user submits their height and weight, the app calculates their BMI and navigates them to a results screen. This screen is more than just a number—it helps the user interpret their result visually and contextually.

### **1. Showing the BMI Value and Category**

Text(  
 bmi.toStringAsFixed(1),  
 style: TextStyle(fontSize: 48, fontWeight: FontWeight.bold, color: color),  
),  
Text(category, style: TextStyle(fontSize: 24, color: color)),

* **Why:**  
   Simply giving a number isn’t enough. Users need to understand what that number means, and whether it falls into a healthy or concerning range. The color provides quick visual context.
* **How:**  
   The BMI value is passed in from the previous screen and formatted to one decimal place. The getBmiCategory() and getBmiColor() methods determine what category the number falls into ("Normal", "Overweight", etc.) and which color to use to match that range. These are shown in large and styled text so they’re easily noticed.

### **2. Providing a Custom Health Tip**

Text(tip, style: const TextStyle(fontSize: 16), textAlign: TextAlign.center),

* **Why:**  
   Each user receives a tip tailored to their BMI category. This turns raw data into personalized advice, which is much more helpful than numbers alone.
* **How:**  
   The getBmiTip() method returns a short sentence based on the BMI value. For example, someone underweight will get a suggestion to eat nutrient-rich food. The advice is centered on the screen for clarity and emphasis.

### **3. Visual Scale Bar With Gradient and Pointer**

Stack(  
 alignment: Alignment.centerLeft,  
 children: [  
 Container(  
 height: 20,  
 width: double.infinity,  
 decoration: BoxDecoration(  
 borderRadius: BorderRadius.circular(12),  
 gradient: const LinearGradient(  
 colors: [Colors.orange, Colors.green, Colors.yellow, Colors.red],  
 stops: [0.1, 0.35, 0.65, 1.0],  
 ),  
 ),  
 ),  
 Positioned(  
 left: markerPosition,  
 child: const Icon(Icons.arrow\_drop\_up, size: 32, color: Colors.white),  
 ),  
 ],  
),

* **Why:**  
   A colored scale helps users visually interpret where their BMI stands. It’s a simple but powerful way to show whether their number is in a healthy range or not.
* **How:**  
   The bar uses a gradient to represent BMI ranges from low to high: orange (underweight), green (normal), yellow (overweight), and red (obese). The white arrow icon is placed using the result of getBmiMarkerPosition(), which maps the user's BMI to a horizontal position on the screen. That position reflects how far along the scale their result is.

### **4. Labeling the Scale Ranges**

Row(  
 mainAxisAlignment: MainAxisAlignment.spaceBetween,  
 children: const [  
 Text("Under"),  
 Text("Normal"),  
 Text("Over"),  
 Text("Obese"),  
 ],  
),

* **Why:**  
   Without labels, users wouldn’t know what each color section on the gradient represents. These provide that key context.
* **How:**  
   This row uses MainAxisAlignment.spaceBetween to evenly distribute four category labels underneath the scale. Each word matches a portion of the colored bar above it, clarifying the meaning of the pointer's position.

### **5. Adding an Option to Learn More**

TextButton(  
 onPressed: () {  
 Navigator.push(  
 context,  
 MaterialPageRoute(builder: (\_) => const BmiTipsPage()),  
 );  
 },  
 child: const Text("Learn more about BMI →"),  
),

* **Why:**  
   Some users may want more detailed guidance or health information after seeing their result. Offering an optional tips page helps them take the next step without overloading the main screen.
* **How:**  
   The button uses a Navigator.push() call to open a new screen (BmiTipsPage). This gives users the chance to explore more without leaving the app or losing their data.