Connecting SQL to my Flutter Project Tutorial

1. First off we made our database, with columns and appropriate fields in SSMS
2. We then made our Backend to our Flutter App which allows us to connect our App to SQL
3. Making Our Backend
    a. Three Main Components
        i. Model
            1. Match it to your SQL table/columns
        ii. Context Class
            1. The class help links our backend to the actual SQL database of which we use a DbSet to help map our database table to our backend of which we can query from
            2. Ex:
                a. public DbSet<Paint> Paintings { get; set; }
            3. This then helps connect to our Program.cs file of which tells ASP.NET Core to create and inject our context using our connectionString
                a. The connection string then will come from your appsettings.json and be formatted as such

                "ConnectionStrings": {
                "DefaultConnection":
                "Server=CS-20\\SQLEXPRESS;Initial Catalog=PaintAppDatabase;TrustServerCertificate=True;Trusted_Connection=True;"
                },
            4.
        iii. Controller
            1. Here we can create our endpoints to interact with our database, having the ability to get data, delete, or add
4. **Focus of Tutorial** → How we Interact with SQL and our Backend in Android Studio
    a. Making a Model Class
        i. WHY?
            1. Your Flutter app talks to the backend using HTTP, and the backend sends JSON data back.
                a. So… You need a Dart model to:
                    i. Parse that JSON into usable Flutter objects
                    ii. Display it on the UI (ListView, etc.)
        ii. Code and Functionality
            1. First Half of Model Class

```
a. class Painting {
b.  final int? PaintId;
c.  final String Title;
```

```
  final String Description;
  final String PaintingUrl;

  Painting({
    required this.PaintId,
    required this.Title,
    required this.Description,
    required this.PaintingUrl,
  });
```

m. Here we are matching our model in our Backend and then making a constructor to use these parameters in our Flutter code

2. Second Half

```
factory Painting.fromJson(Map<String, dynamic> json) {
 return Painting(
   PaintId: json['paintId'],
   Title: json['title'] ?? 'unknown',
   Description: json['description'] ?? 'unknown',
   PaintingUrl: json['paintingUrl'] ?? 'unknown',
 );
}
```

i. A factory constructor lets you return a new instance of a class, but here we our transforming the json data to a Dart object we can understand and use better

j. The function itself creates and returns a Painting object using the values extracted from the json map

k. The ?? operators default to “unknown” if the value is null or missing

b. NextPage file and how it wraps this whole process

i. Dependencies

1. import 'dart:convert'; - helps to decode JSON data
2. import 'dart:io'; - helps to override security restrictions
3. import 'package:http/http.dart' as http; - Makes HTTP calls
4. import 'package:http/io_client.dart'; - Used with custom HttpClient
5. import 'models/painting.dart'; - The model that represents a Painting

ii. getPaintings Function

```
final client = HttpClient()
 ..badCertificateCallback = (cert, host, port) => true;
final ioClient = IOClient(client);
```

4. Because of issues connecting to my backend where it uses Https, my code became functional when we created an HttpClient that

ignores SSL certificate errors. We then turned converted the HttpClient into an IO-compatible client you can use such as http.get()

```
final url = Uri.parse('http://10.0.2.2:5041/paint/GetPainting');
```

6. Builds the URL to your backend's GET endpoint.
7. 10.0.2.2 lets Android Emulator talk to your PC's localhost.
8. 5041 is the port your ASP.NET backend is running on.

```
try {
  final response = await ioClient.get(url);
  if (response.statusCode == 200) {
    List<dynamic> jsonList = jsonDecode(response.body);
    return jsonList.map((json) => Painting.fromJson(json)).toList();
  } else {
    throw Exception('Failed to load paintings');
  }
} catch (e) {
  throw Exception("Fetch failed: $e");
}
}
```

1. The first line here we are sending a GET request to the backend of which using an if statement, check if we get a 200 OK response
2. If it is successful, we then decode the JSON response into a Dart list of dynamic maps of which we then convert into a Painting object via the fromJson() constructor from our model class
3. Otherwise we signal we failed to load paintings and use a catch to check for errors

xv. Our State Class

```
List<Painting> paintings = [];
bool isLoading = true;
```

3. We set an empty array to hold our paintings and have a loading spinner while our app fetches the paintings

```
@override
void initState() {
 super.initState();
 loadPaintings();
}
```

9. This function is called as soon as the screen loads of which will trigger a loadPaintings() function that will fetch the paintings data

10. loadPaintings function

```
Future<void> loadPaintings() async {
  try {
```

```
    paintings = await getPaintings();
  } catch (e) {
    print("Error loading paintings: $e");
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: $e')),
    );
  } finally {
    setState(() {
      isLoading = false;
    });
  }
}
```

o. On third line here we call our getPaintings function which as we said sends an HTTP GET request to our backend and converts it into Painting objects
p. If it does not work, we log the error to the console and then to help the user see if there is an error as well, we display an error in the snackbar so they aren't staring at an empty screen for so long
q. We use a finally block here specifically for our isLoading so it guarantees the page responds despite whether or not the getPainting call works so the user isn't stuck on an infinite loading screen - basically updates the page's ui regardless

11. Build Wigdet/ UI Layout

```
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: const Text('Painting Catalog'),
     ),
     body: isLoading
         ? const Center(child:
CircularProgressIndicator())
         : paintings.isEmpty
         ? const Center(child: Text("No paintings
found."))
         : ListView.builder(
       itemCount: paintings.length,
       itemBuilder: (context, index) {
         final painting = paintings[index];
         return ListTile(
           title: Text(painting.Title),
           subtitle: Text(painting.Description),
```

```
r.                );
s.              },
t.            ),
u.        );
v.    }
w.  }
```

x. Starting at the body, we basically say if isLoading is true we show the loading spinner
y. If we see an empty painting list, we simply give a message saying “No Paintings Found”
z. ListView.builder block
    i. Creates what is only currently visible on screen, not everything in the memory
    ii. We use .builder here since it allows us to dynamically build our ui as we scroll down
    iii. Our itemCount: paintings.length tells the app to display however many items there are and make a list with that many rows which also controls how many times the builder function is called
    iv. Our itemBuilder is a function that is ran for every item in the list of which returns each item with its Title and Description