Complete Guide: Metro App Authentication System

Table of Contents

---

1. Project Architecture Overview

Your app follows a **layered architecture**:

```
UI Layer (Widgets)
    ↓
State Management Layer (Cubit)
    ↓
Service Layer (AuthService)
    ↓
Network Layer (ApiClient)
    ↓
Server (Backend API)
```

Why this structure?

- **Separation of Concerns**: Each layer has one responsibility
- **Testability**: You can test each layer independently
- **Maintainability**: Easy to update or replace parts without affecting others

---

2. User Model (`user_model.dart`)

What is a Model?

A model is a **blueprint** for your data. It defines what a User looks like in your app.

Code Breakdown:

```dart
class User extends Equatable {
```

What is Equatable?

- Equatable is a package that helps compare objects easily
- Without it: `user1 == user2` compares memory addresses (always false)
- With it: `user1 == user2` compares actual values (true if values match)

```dart
final String id;
final String email;
final String name;
```

Why `final`?

- These values cannot be changed after creation (immutable)
- This prevents accidental modifications and makes your app more predictable

```dart
const User({required this.id, required this.email, required this.name});
```

Constructor:

- Creates a new User object
- `required` means you MUST provide these values
- `this.id` is shorthand for assigning the parameter to the field

```dart
```

```
@override
List<Object?> get props => [id, email, name];
```

For Equatable:

- Tells Equatable which fields to compare
- Two users are equal if their id, email, AND name match

```dart
factory User.fromJson(Map<String, dynamic> json) {
  return User(
    id: json['_id'] ?? json['id'],
    email: json['email'],
    name: json['name'],
  );
}
```

fromJson Factory:

- Converts JSON (from API) into a User object
- `json['_id'] ?? json['id']`: Use `_id` if it exists, otherwise use `id`
- **Example**:

```json
{"_id": "123", "email": "user@example.com", "name": "John"}
```

Becomes:

```dart
User(id: "123", email: "user@example.com", name: "John")
```

```dart
Map<String, dynamic> toJson() {
  return {'id': id, 'email': email, 'name': name};
}
```

toJson Method:

- Converts User object into JSON (for sending to API)
- Opposite of `fromJson`

```dart
User copyWith({String? id, String? email, String? name}) {
  return User(
    id: id ?? this.id,
    email: email ?? this.email,
    name: name ?? this.name,
  );
}
```

copyWith Method:

- Creates a copy of the user with some fields changed
- `??` means "if null, use the original value"
- **Example**:

```dart
  User newUser = currentUser.copyWith(name: "New Name");
  // Only name changes, id and email stay the same
```

AuthResult Class:

```dart
class AuthResult {
  final bool success;
  final String message;
  final User? user;
```

Purpose: Wraps the result of authentication

- `success`: Did login work?
- `message`: What happened? (e.g., "Login Successful" or "Invalid password")
- `user?`: The user data (null if login failed)

---

3. API Configuration (`api_config.dart`)

```dart
class ApiConfig {
  static const String baseUrl = 'https://metrodb-production.up.railway.app';
```

Base URL: The main address of your backend server

```dart
  static const Duration connectTimeout = Duration(seconds: 30);
  static const Duration receiveTimeout = Duration(seconds: 30);
```

Timeouts:

- `connectTimeout`: How long to wait when connecting to server
- `receiveTimeout`: How long to wait for server's response
- After 30 seconds, it gives up and throws an error

```dart
  static const String loginEndpoint = '/api/v1/users/login';
```

Endpoint: The specific path for login

- Full URL becomes: `https://metrodb-production.up.railway.app/api/v1/users/login`

Why use constants?

- Easy to change in one place
- Prevents typos
- Makes code cleaner

---

4. Storage Service (`storage_service.dart`)

What is SharedPreferences?

- A way to save simple data on the device (like a tiny database)
- Data persists even after closing the app

- Perfect for storing tokens, user info, settings

Code Breakdown:

```dart
SharedPreferences? _prefs;
```

Why nullable (`?`)?

- SharedPreferences needs to be initialized asynchronously
- Starts as null until `_init()` is called

```dart
Future<void> _init() async {
  _prefs ??= await SharedPreferences.getInstance();
}
```

Lazy Initialization:

- `??=` means "if _prefs is null, then assign it"
- Only initializes once, even if called multiple times
- `async/await`: Waits for SharedPreferences to be ready

```dart
static const String _tokenKey = 'auth_token';
static const String _userIdKey = 'user_id';
```

Keys: Like variable names for stored data

- `static const`: Same for all instances, never changes

```dart
Future<void> saveToken(String token) async {
  await _init();
  await _prefs!.setString('auth_token', token);
}
```

Saving Token:

- `!` tells Dart "I promise _prefs is not null"
- `await _init()`: Make sure SharedPreferences is ready
- Saves the authentication token to device

```dart
String? getToken() {
  return _prefs?.getString('auth_token');
}
```

Getting Token:

- `?.` (safe navigation): Returns null if _prefs is null
- Returns the saved token or null

```dart
bool isLoggedIn() {
  final token = getToken();
  return token != null && token.isNotEmpty;
}
```

Check Login Status:

- If there's a token, user is logged in
- Used to decide if user should see login screen or home screen

```dart
```

```
Future<void> saveUserData({
  required String id,
  required String email,
  required String name,
}) async {
  await _init();
  _prefs!
    ..setString(_userIdKey, id)
    ..setString(_userEmailKey, email)
    ..setString(_userNameKey, name);
}
```

Cascade Notation (`..`):

- Performs multiple operations on the same object
- Same as:

```
dart

  _prefs!.setString(_userIdKey, id);
  _prefs!.setString(_userEmailKey, email);
  _prefs!.setString(_userNameKey, name);
```

---

5. API Client (`api_client.dart`)

What is Dio?

- A powerful HTTP client for Flutter
- Makes API requests (GET, POST, PUT, DELETE)
- Handles errors, interceptors, timeouts

Code Breakdown:

```
dart

final Dio _dio = Dio();
```

Creating Dio Instance:

- `_dio`: The underscore makes it private to this class

- This instance will make all your HTTP requests

```dart
ApiClient() {
  _setupBaseOptions();
  _setupInterceptors();
}
```

Constructor:

- Runs automatically when you create `ApiClient()`
- Sets up Dio configuration and interceptors

Base Options:

```dart
_dio.options = BaseOptions(
  baseUrl: baseUrl,
  connectTimeout: ApiConfig.connectTimeout,
  receiveTimeout: ApiConfig.receiveTimeout,
```

Configuration:

- `baseUrl`: Prepended to all requests
- Example: `post('/login')` becomes `https://metrodb.../login`

```dart
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
```

Default Headers:

- `Content-Type`: Tells server we're sending JSON
- `Accept`: Tells server we expect JSON back

```dart
```

```
validateStatus: (status) {
  return status != null && status < 500;
},
```

Status Validation:

- Normally, Dio throws errors for 400-499 status codes
- This says "only throw errors for 500+ (server errors)"
- Allows manual handling of 401 (Unauthorized), 400 (Bad Request), etc.

Interceptors:

What are Interceptors?

- Code that runs BEFORE request or AFTER response
- Like middleware in Express.js
- Perfect for logging, adding tokens, error handling

dart

```
onRequest: (options, handler) async {
  print('REQUEST[${options.method}] => PATH: ${options.path}');
  print('REQUEST DATA: ${options.data}');
```

Request Interceptor - Logging:

- Runs before every request
- Prints what you're sending (helpful for debugging)

dart

```
  final token = await StorageService().getToken();
  if (token != null && token.isNotEmpty) {
    options.headers['Authorization'] = 'Bearer $token';
    print('Authorization header added');
  }
```

Adding Authentication Token:

- Gets saved token from storage
- Adds it to request headers

- `Bearer`: Standard format for JWT tokens
- Server uses this to verify you're logged in

```dart
  return handler.next(options);
}
```

Continue Request:

- Passes modified request to next step

```dart
onResponse: (response, handler) {
  print('RESPONSE[${response.statusCode}] => DATA: ${response.data}');
  return handler.next(response);
},
```

Response Interceptor:

- Runs after every successful response
- Logs what you received

```dart
onError: (DioException error, handler) {
  print('ERROR[${error.response?.statusCode}] => MESSAGE: ${error.message}');
  print('ERROR RESPONSE: ${error.response?.data}');
  print('ERROR TYPE: ${error.type}');
  return handler.next(error);
},
```

Error Interceptor:

- Runs when request fails
- Logs error details (status code, message, type)

HTTP Methods:

```dart
```

```dart
Future<Response> get(String path, {Map<String, dynamic>? query}) async {
  try {
    return await _dio.get(path, queryParameters: query);
  } catch (e) {
    print('GET Error: $e');
    rethrow;
  }
}
```

GET Request:

- Used for retrieving data
- `queryParameters`: Adds `?key=value` to URL
- `rethrow`: Passes error to caller to handle

```dart
Future<Response> post(String path, {dynamic data}) async {
  try {
    return await _dio.post(path, data: data);
  } catch (e) {
    print('POST Error: $e');
    rethrow;
  }
}
```

POST Request:

- Used for creating/sending data
- `data`: Request body (usually JSON)

---

6. Authentication Service (`auth_service.dart`)

Purpose:

Handles all authentication logic (login, validation, error handling)

Code Breakdown:

```dart
```

```
final ApiClient _apiClient = ApiClient();
final StorageService _storage = StorageService();
```

Dependencies:

- Uses ApiClient to make requests
- Uses StorageService to save data

```
dart

Future<AuthResult> signIn({
  required String email,
  required String password,
}) async {
```

Sign In Method:

- `Future<AuthResult>`: Returns result after async operation
- Takes email and password

Validation:

```
dart

if (!_isValidEmail(email)) {
  return AuthResult(success: false, message: 'Invalid Email');
}
if (password.isEmpty || password.length < 6) {
  return AuthResult(
    success: false,
    message: 'Password must be at least 6 characters',
  );
}
```

Client-Side Validation:

- Checks email format and password length
- Returns early if invalid (saves API call)

```
dart
```

```
bool _isValidEmail(String email) {
  return RegExp(r'^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$').hasMatch(email);
}
```

Email Regex:

- `RegExp`: Regular expression (pattern matching)
- Checks if email has proper format: `something@domain.com`

Making the Request:

dart

```
final response = await _apiClient.post(
  _loginEndpoint,
  data: {
    'email': email,
    'password': password,
  },
);
```

API Call:

- Sends POST request to `/api/v1/users/login`
- Body contains email and password as JSON

Handling Success:

dart

```
if (response.statusCode == 200 || response.statusCode == 201) {
  final data = response.data;

  if (data['token'] != null) {
    await _storage.saveToken(data['token']);
    print('Token saved: ${data['token']}');
  }
```

Status Codes:

- `200`: OK (success)
- `201`: Created (also success)

- Saves the authentication token to device

```dart
final userJson = data['data']['user'];
final user = User.fromJson(userJson);

await _storage.saveUserData(
  id: user.id,
  email: user.email,
  name: user.name,
);
```

Parsing Response:

- Expected structure: `{token: "...", data: {user: {...}}}`
- Converts JSON to User object
- Saves user info to device

```dart
return AuthResult(
  success: true,
  message: 'Login Successful',
  user: user,
);
```

Success Result:

- Returns success with user data
- UI will use this to navigate to home screen

Error Handling:

```dart
} on DioException catch (e) {
  print('DioException caught: ${e.type}');
  print('DioException message: ${e.message}');
  print('DioException response: ${e.response?.data}');
```

Catching Dio Errors:

- `on DioException`: Only catches Dio-specific errors
- Logs detailed error info

```dart
if (e.response != null) {
  final statusCode = e.response!.statusCode;
  final data = e.response!.data;

  if (statusCode == 401) {
    String errorMessage = 'Invalid email or password';

    if (data is Map && data['message'] != null) {
      errorMessage = data['message'];
    } else if (data is String) {
      errorMessage = data;
    }

    return AuthResult(success: false, message: errorMessage);
  }
```

401 Unauthorized:

- Wrong email or password
- Extracts error message from response
- `data is Map`: Checks if data is a Map type

```dart
} else if (e.type == DioExceptionType.connectionTimeout) {
  return AuthResult(
    success: false,
    message: 'Connection timeout. Please check your internet connection.',
  );
} else if (e.type == DioExceptionType.connectionError) {
  return AuthResult(
    success: false,
    message: 'Connection error. Please check your internet connection.',
  );
}
```

Network Errors:

- `connectionTimeout`: Took too long to connect
- `connectionError`: No internet or can't reach server

---

7. State Management with Cubit

What is State Management?

- **State**: The data that describes your app at any moment
- **State Management**: How you update and share that data
- **Example**: Is user logged in? Is loading? What's the error?

What is Cubit?

- A simpler version of BLoC (Business Logic Component)
- Manages state and business logic
- **Flow**: Event → Cubit → New State → UI Updates

Login States (`login_state.dart`)

dart

```dart
abstract class LoginState extends Equatable {
  const LoginState();

  @override
  List<Object> get props => [];
}
```

Base State:

- `abstract`: Cannot be instantiated directly
- All login states extend this

dart

```
class LoginInitial extends LoginState {}
```

Initial State:

- User hasn't tried to login yet
- UI shows login form (not loading, no errors)

dart

```
class LoginLoading extends LoginState {}
```

Loading State:

- Login request is in progress
- UI shows loading spinner, disables button

dart

```
class LoginSuccess extends LoginState {
  final User user;

  const LoginSuccess(this.user);
  @override
  List<Object> get props => [user];
}
```

Success State:

- Login succeeded
- Contains the logged-in user
- UI navigates to home screen

dart

```
class LoginFailure extends LoginState {
  final String error;
  const LoginFailure({required this.error});

  @override
  List<Object> get props => [error];
}
```

Failure State:

- Login failed
- Contains error message
- UI shows error (e.g., "Invalid password")

Login Cubit (`login_cubit.dart`)

dart

```
class LoginCubit extends Cubit<LoginState> {
  final AuthService _authService;

  LoginCubit(this._authService) : super(LoginInitial());
```

Constructor:

- Takes `AuthService` as dependency (Dependency Injection)
- `super(LoginInitial())`: Starts in Initial state

dart

```
Future<void> signIn({required String email, required String password}) async {
  emit(LoginLoading());
  print("Login Started");
```

Sign In Method:

- `emit(LoginLoading())`: Changes state to Loading
- This triggers UI rebuild (shows spinner)

```dart
try {
  final result = await _authService.signIn(
    email: email,
    password: password,
  );
  print("Result: ${result.success}");
```

Call Auth Service:

- Waits for login result
- `await`: Pauses execution until result arrives

```dart
  if (result.success && result.user != null) {
    print("Emitting Success");
    emit(LoginSuccess(result.user!));
  } else {
    print("Emitting LoginFailure");
    emit(LoginFailure(error: result.message));
  }
```

Handle Result:

- If success: Emit Success state with user
- If failure: Emit Failure state with error message
- UI listens to these states and reacts accordingly

```dart
  } catch (e) {
    emit(LoginFailure(error: e.toString()));
  }
}
```

Error Handling:

- Catches any unexpected errors
- Emits Failure state

```dart
void reset() {
  emit(LoginInitial());
}
```

Reset Method:

- Returns to Initial state
- Used when navigating back to login screen

User States (`user_state.dart`)

```dart
class UserInitial extends UserState {}
```

Initial: App just started, user status unknown

```dart
class UserLoading extends UserState {}
```

Loading: Checking if user is logged in

```dart
class UserLoaded extends UserState {
  final User user;
  const UserLoaded(this.user);
}
```

Loaded: User is logged in, contains user data

```dart
class UserError extends UserState {
  final String message;
  const UserError(this.message);
}
```

Error: Something went wrong loading user

```dart
class UserLoggedOut extends UserState {}
```

Logged Out: User is not logged in

User Cubit (`user_cubit.dart`)

```dart
class UserCubit extends Cubit<UserState> {
  final StorageService _storageService;

  UserCubit(this._storageService) : super(UserInitial());
```

Purpose: Manages global user state

```dart
Future<void> loadUser() async {
  try {
    emit(UserLoading());
    final userData = await _storageService.getUserData();
```

Load User:

- Called when app starts
- Checks if user data is saved on device

```dart
```

```
  if (userData != null) {
    final user = User(
      id: userData['id']!,
      email: userData['email']!,
      name: userData['name']!,
    );
    emit(UserLoaded(user));
  } else {
    emit(UserLoggedOut());
  }
```

Result:

- If data exists: User is logged in
- If not: User needs to log in

dart

```
void setUser(User user) {
  emit(UserLoaded(user));
}
```

Set User:

- Called after successful login
- Updates global user state

dart

```
Future<void> logout() async {
  try {
    await _storageService.clearAll();
    emit(UserLoggedOut());
  } catch (e) {
    emit(UserError("Failed to logout: $e"));
  }
}
```

Logout:

- Clears all saved data (token, user info)

- Changes state to LoggedOut
- UI redirects to login screen

```dart
User? get currentUser {
  if (state is UserLoaded) {
    return (state as UserLoaded).user;
  }
  return null;
}
```

Getter:

- Easy way to access current user
- Returns null if not logged in

```dart
String get userName {
  if (state is UserLoaded) {
    return (state as UserLoaded).user.name;
  }
  return "Guest";
}
```

User Name Getter:

- Returns user's name or "Guest"
- Used in UI to display greeting

---

8. How Everything Works Together

Complete Login Flow:

1. **User Enters Credentials**

```
User types email and password in UI
```

2. **UI Calls LoginCubit**

```dart
loginCubit.signIn(email: email, password: password);
```

3. **LoginCubit Emits Loading**

```dart
emit(LoginLoading());
// UI shows spinner
```

4. **LoginCubit Calls AuthService**

```dart
final result = await _authService.signIn(email, password);
```

5. **AuthService Validates Input**

```dart
if (!_isValidEmail(email)) return error;
if (password.length < 6) return error;
```

6. **AuthService Calls ApiClient**

```dart
final response = await _apiClient.post('/login', data: {...});
```

7. **ApiClient Makes HTTP Request**

```
ApiClient → Interceptor (adds token) → Dio → Server
```

8. **Server Responds**

```json
```

```
{
 "token": "eyJhbGciOiJIUzI1NiIs...",
 "data": {
  "user": {
   "_id": "123",
   "email": "user@example.com",
   "name": "John Doe"
  }
 }
}
```

9. **AuthService Processes Response**

```dart
await _storage.saveToken(data['token']);
final user = User.fromJson(data['data']['user']);
await _storage.saveUserData(...);
return AuthResult(success: true, user: user);
```

10. **LoginCubit Emits Success**

```dart
emit(LoginSuccess(result.user!));
// UI navigates to home
```

11. **UserCubit Updates Global State**

```dart
userCubit.setUser(user);
// Now user is available app-wide
```

App Startup Flow:

1. **App Starts**

```dart
```

```
main() → runApp(MyApp())
```

2. **Provide Cubits**

dart

```
MultiBlocProvider(
  providers: [
    BlocProvider(create: (_) => UserCubit(StorageService())),
    BlocProvider(create: (_) => LoginCubit(AuthService())),
  ],
)
```

3. **UserCubit Loads User**

dart

```
userCubit.loadUser();
```

4. **Check Storage**

dart

```
final userData = await _storageService.getUserData();
```

5. **Decide Initial Screen**

dart

```
if (userData != null) {
  emit(UserLoaded(user));
  // Navigate to Home
} else {
  emit(UserLoggedOut());
  // Navigate to Login
}
```

Where Each Part is Used:

User Model: Everywhere you need user data

- LoginCubit (stores logged-in user)
- UserCubit (manages global user)
- Profile Screen (displays user info)
- API requests (sends user data)

ApiClient: All API requests

- AuthService (login, register)
- Metro Service (get stations, routes)
- Booking Service (create bookings)
- Profile Service (update profile)

StorageService: Persisting data

- Login (save token and user)
- App startup (check if logged in)
- Logout (clear data)
- Settings (save preferences)

AuthService: Authentication operations

- Login screen
- Register screen
- Password reset

LoginCubit: Login screen only

- Manages login process
- Handles login errors
- Navigates on success

UserCubit: Throughout entire app

- App startup (load user)
- All screens (access user data)
- Logout (clear user)
- Profile updates (refresh user)

Key Concepts Summary

Async/Await:

```dart
Future<String> fetchData() async {
  // async: This function returns a Future
  final result = await someAsyncOperation();
  // await: Wait for result before continuing
  return result;
}
```

Null Safety:

```dart
String? name;        // Can be null
String name2 = "John"; // Cannot be null
name2 = null;        // ERROR!

final value = name ?? "Default"; // If name is null, use "Default"
```

State Management Flow:

```
User Action → Cubit Method → emit(NewState) → UI Rebuilds
```

Dependency Injection:

```dart
class LoginCubit {
  final AuthService _authService; // Dependency

  LoginCubit(this._authService);  // Injected via constructor
}
```

Why Use Cubit?

- **Predictable**: Same input = same output
- **Testable**: Easy to test each state

- **Reactive**: UI automatically updates
- **Scalable**: Easy to add new features

Common Patterns

Error Handling Pattern:

```dart
try {
  final result = await riskyOperation();
  // Handle success
} on SpecificException catch (e) {
  // Handle specific error
} catch (e) {
  // Handle any other error
}
```

State Pattern:

```dart
if (state is UserLoaded) {
  // Access user data
  final user = (state as UserLoaded).user;
}
```

Repository Pattern (Your Architecture):

```
UI → Cubit → Service → ApiClient → Server
```

Next Steps for Learning:

1. **Add print statements** to see the flow
2. **Trigger errors** intentionally to see error handling
3. **Add new features** (register, forgot password)
4. **Create new cubits** for other features (metro routes, bookings)

5. **Write tests** for your services and cubits

Questions to Test Your Understanding:

1. What happens if you remove `await` before `_authService.signIn()`?
2. Why do we save the token to storage instead of keeping it in memory?
3. What's the difference between `LoginCubit` and `UserCubit`?
4. Why use `Equatable` for states?
5. What happens if the user closes the app after logging in?

Answers:

1. Code continues before login finishes, `result` would be a Future instead of AuthResult
2. Memory is cleared when app closes, storage persists
3. LoginCubit: temporary (login process), UserCubit: permanent (app-wide user state)
4. So Flutter knows when to rebuild widgets (compares states)
5. UserCubit loads saved data on startup, user stays logged in!