Motorcycle Shop Management

Release 1.0.0

Sou Chanrojame, Yin Sothkry, Orn Pheakdey, Long Neron

Feb 12, 2026

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Motorcycle Shop Management Flutter app provides a complete e-commerce solution for motorcycle browsing and purchasing.

1.1 App Overview

This Flutter application connects to a Spring Boot backend API to provide:

- Product browsing by category
- User authentication
- Shopping cart functionality
- Order placement
- AI-powered chat assistance

1.2 Features

- User login and registration with profile images
- Product catalog with categories (Sport, Cruiser, Off-Road, Scooter, Touring)
- Shopping cart with quantity management
- Favorites list
- Product upload with image
- AI chat assistant (SSE streaming)
- Order history and checkout
- Profile editing and password change

1.3 Technology Stack

- **Flutter SDK**: 3.8.1
- **Dart**: ^3.8.1
- **HTTP Package**: ^1.6.0 for API communication
- **Image Picker**: ^1.2.1 for photo uploads
- **Flutter Client SSE**: ^2.0.3 for chat streaming

- **Markdown Widget**: ^2.3.2+8 for chat messages

1.4 App Architecture

1.4.1 File Structure

```
lib/
├── main.dart              # App entry and routes
├── config.dart            # API URL and utilities
├── home.dart              # Bottom navigation
├── start.dart             # Landing screen
├── login.dart             # Login UI
├── user.dart              # Registration UI
├── controller/
│   ├── cart_controller.dart
│   └── favorites_controller.dart
└── tab/
    ├── home.dart              # Home tab
    ├── motorcycle.dart        # Product grid/details
    ├── upload.dart            # Product upload
    ├── chat.dart              # AI chat
    ├── setting.dart           # Settings/orders
    ├── cart_screen.dart
    ├── favorites_screen.dart
    └── edit_profile.dart
```

1.5 Screens

Public Screens

- `start.dart` - Landing page with login/register buttons
- `login.dart` - Username/password login
- `user.dart` - Registration form

Main App Screens (5 tabs)

1. **Home** - Search, categories, featured products
2. **Motorcycles** - Product grid with details
3. **Upload** - Add new motorcycle with image
4. **Chat** - AI assistant with streaming responses
5. **Settings** - Profile, orders, FAQ, logout

1.6 Controllers

Two singleton controllers manage app state:

- `CartController` - Shopping cart items and total
- `FavoritesController` - Favorite products list

1.7 API Communication

The app communicates with a REST API at the URL configured in `config.dart`:

```
var config = {
  "apiUrl": "http://10.0.2.2:8080",  // Android emulator
  // "apiUrl": "https://your-api.com",  // Production
};
```

1.7.1 Key Endpoints

- `POST /auth/login` - User login
- `POST /user` - User registration
- `GET /product` - List products
- `POST /product` - Upload product with image
- `POST /order` - Create order
- `GET /order/user/{id}` - User order history
- `POST /chat` - AI chat stream (SSE)

1.8 Getting Started

See *Quick Start* for quick setup.
See *Installation* for setup instructions.

CHAPTER

TWO

QUICK START

2.1 Quick Setup

These commands clone the motorcycle-management repository, navigate into its folder, fetch all required Flutter packages/dependencies, and finally build + run the app on a connected Android/iOS device or emulator.

```
git clone https://github.com/james5635/motorcycle-management.git
cd motorcycle-management
flutter pub get
flutter run
```

CHAPTER

THREE

INSTALLATION

3.1 Requirements

- Flutter SDK ^3.8.1
- Android Studio or VS Code
- Android SDK (for Android)
- Xcode (for iOS, macOS only)

3.2 Setup

3.2.1 1. Install Dependencies

```
flutter pub get
```

3.2.2 2. Configure API URL

Edit `lib/config.dart`:

```
var config = {
  // Android emulator
  "apiUrl": "http://10.0.2.2:8080",

  // iOS simulator
  // "apiUrl": "http://localhost:8080",

  // Physical device (use your computer's IP)
  // "apiUrl": "http://192.168.1.100:8080",
};
```

3.2.3 3. Run the App

Android:

```
flutter run
```

iOS (macOS only):

```
flutter run
```

3.3 Troubleshooting

Build fails

```
flutter clean
flutter pub get
```

Gradle issues (Android)

```
cd android
./gradlew clean
cd ..
flutter run
```

iOS CocoaPods issues

```
cd ios
rm -rf Pods Podfile.lock
pod install
cd ..
```

CHAPTER

FOUR

CONFIGURATION

4.1 API Configuration

The `lib/config.dart` file contains the API base URL:

```
var config = {
  "apiUrl": "http://10.0.2.2:8080",
};
```

4.1.1 Environment URLs

- **Android emulator**: `http://10.0.2.2:8080`
- **iOS simulator**: `http://localhost:8080`
- **Physical device**: Use your computer's IP (e.g., `http://192.168.1.100:8080`)
- **Production**: Your deployed API URL

4.2 Utility Functions

4.2.1 formatPrice

Removes trailing .0 from prices:

```
String formatPrice(dynamic price) {
  if (price == null) return '0';
  double numPrice = price is num
      ? price.toDouble()
      : double.tryParse(price.toString()) ?? 0;
  if (numPrice == numPrice.truncate()) {
    return numPrice.truncate().toString();
  }
  return numPrice.toStringAsFixed(2);
}
```

Examples:

- `formatPrice(1500)` → `"1500"`
- `formatPrice(1500.50)` → `"1500.50"`
- `formatPrice(null)` → `"0"`

4.2.2 calculateStars

Calculates star rating (1-5) based on price digit sum:

```
int calculateStars(dynamic price) {
  if (price == null) return 1;
  int sum = 0;
  String priceStr = price is num
      ? price.toInt().toString()
      : (double.tryParse(price.toString()) ?? 0).toInt().toString();

  for (int i = 0; i < priceStr.length; i++) {
    sum += int.tryParse(priceStr[i]) ?? 0;
  }

  if (sum == 0) return 1;
  while (sum > 5) sum -= 5;
  return sum;
}
```

4.3 FAQ Data

Static FAQ content for the settings screen:

```
var faq = [
  {
    "question": "What is Motorcycle Shop Management?",
    "answer": "Motorcycle Shop Management is a Flutter project...",
  },
  // ... more FAQs
];
```

4.4 Dependencies

From `pubspec.yaml`:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^1.6.0
  bcrypt: ^1.2.0
  image_picker: ^1.2.1
  http_parser: ^4.1.2
  flutter_client_sse: ^2.0.3
  markdown_widget: ^2.3.2+8
```

CHAPTER

FIVE

PROJECT STRUCTURE

5.1 Directory Layout

```
lib/
├── controller/           # State management
│   ├── cart_controller.dart
│   └── favorites_controller.dart
├── tab/                  # Screen implementations
│   ├── home.dart
│   ├── motorcycle.dart
│   ├── upload.dart
│   ├── chat.dart
│   ├── setting.dart
│   ├── cart_screen.dart
│   ├── favorites_screen.dart
│   └── edit_profile.dart
├── config.dart           # API config and utilities
├── home.dart             # Main navigation
├── login.dart            # Login screen
├── main.dart             # App entry point
├── start.dart            # Landing page
└── user.dart             # Registration screen
```

5.2 Key Files

main.dart

App entry point with MaterialApp and route definitions.

config.dart

- `config` map with API URL
- `formatPrice()` utility
- `calculateStars()` utility
- `faq` list for settings

home.dart

Main scaffold with:

- BottomNavigationBar (5 tabs)

- IndexedStack for tab content
- User data passed to screens

controller/cart_controller.dart

Singleton controller for cart state:

- `addToCart()`
- `removeFromCart()`
- `updateQuantity()`
- `clearCart()`
- `items` getter
- `totalAmount` getter
- `itemCount` getter

controller/favorites_controller.dart

Singleton controller for favorites:

- `addToFavorites()`
- `removeFromFavorites()`
- `toggleFavorite()`
- `isFavorite()`
- `favorites` getter

tab/home.dart

Home tab with:

- Search bar
- Promo carousel (auto-scroll)
- Categories row
- Featured products
- Most popular products
- Recommended grid

tab/motorcycle.dart

Product browsing:

- `ProductGridScreen` - 2-column grid
- `ProductCard` - Product display
- `ProductDetailScreen` - Details with hero animation
- `_showSpecsDialog` - Specifications popup

tab/upload.dart

Product upload form:

- Image picker
- Text fields (name, brand, price, etc.)

- Category dropdown
- Condition dropdown
- Multipart upload

tab/chat.dart

AI chat with:

- Message list
- Text input
- SSE streaming
- Markdown rendering
- Copy/like/dislike buttons

tab/setting.dart

Settings screen with:

- User profile header
- Edit profile
- Change password
- My favorites
- My orders
- FAQ
- Terms
- Logout

tab/cart_screen.dart

Shopping cart UI:

- Cart items list
- Quantity controls
- Remove items
- Total amount
- Checkout button

tab/favorites_screen.dart

Favorites grid:

- 2-column grid of favorites
- Remove button
- Empty state

tab/edit_profile.dart

Profile editing:

- Form with validation
- Image picker

- Multipart upload

5.3 Platform Directories

android/ - Android configuration

- `app/src/main/AndroidManifest.xml`
- `app/build.gradle`

ios/ - iOS configuration

- `Runner/Info.plist`
- `Podfile`

5.4 Configuration Files

pubspec.yaml

Dependencies and app metadata:

- Flutter SDK: ^3.8.1
- http: ^1.6.0
- image_picker: ^1.2.1
- flutter_client_sse: ^2.0.3
- markdown_widget: ^2.3.2+8
- bcrypt: ^1.2.0
- http_parser: ^4.1.2

CHAPTER

SIX

ARCHITECTURE

6.1 App Structure

The app uses a simple layered architecture:

```
UI Layer (Screens/Widgets)
       ↕
Controller Layer (ChangeNotifier)
       ↕
Data Layer (HTTP API)
```

6.2 State Management

Controllers use Flutter's `ChangeNotifier`:

- `CartController` - Manages shopping cart state
- `FavoritesController` - Manages favorites list

Both use the singleton pattern for app-wide state.

6.3 Navigation

Named routes defined in `main.dart`:

```
routes: {
  '/home': (context) => HomePage(),
  '/login': (context) => LoginPage(),
  '/user': (context) => UserPage(),
  '/start': (context) => StartPage(),
}
```

Main navigation uses `IndexedStack` with 5 tabs managed in `home.dart`.

6.4 Data Flow

```
User Action → Controller → API Call
                              ↓
UI Updates ← notifyListeners() ← Response
```

6.5 API Integration

All API calls use the `http` package with base URL from `config.dart`:

- GET requests for fetching data
- POST requests with JSON or multipart
- PUT requests for updates
- SSE for chat streaming

CHAPTER

SEVEN

AUTHENTICATION

7.1 Screens

start.dart - Landing page

- Login button → `/login`
- Create Account button → `/user`

login.dart - Login form

Fields:

- Username (TextFormField with validation)
- Password (TextFormField with obscure toggle)

Validation:

- Username: required
- Password: required, min 6 characters

user.dart - Registration form

Fields:

- Full Name (required, min 2 chars)
- Password (required, min 6 chars)
- Email (required, valid format)
- Phone Number (required, valid format)
- Address (required, min 5 chars)
- Role (dropdown: customer, admin, guest)
- Profile Image (optional, picked from gallery)

7.2 API Endpoints

Login

```
POST /auth/login
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
  "username": "user",
  "password": "pass"
}
```

Register

```
POST /user
Content-Type: multipart/form-data

Parts:
- user: JSON string
- profileImage: file (optional)
```

7.3 User Data Flow

1. Login/Register returns user data
2. Data passed via navigation arguments
3. Retrieved in `home.dart` with `ModalRoute.of(context)!.settings.arguments`
4. User ID used for orders, profile, etc.

7.4 Password Change

In `tab/setting.dart`:

- `ChangePasswordScreen` widget
- Form with new password and confirm
- PUT request to `/user/{id}`

7.5 Logout

Clears navigation stack and returns to `/start`:

```
Navigator.pushNamedAndRemoveUntil(
  context,
  '/start',
  (_) => false,
);
```

CHAPTER

EIGHT

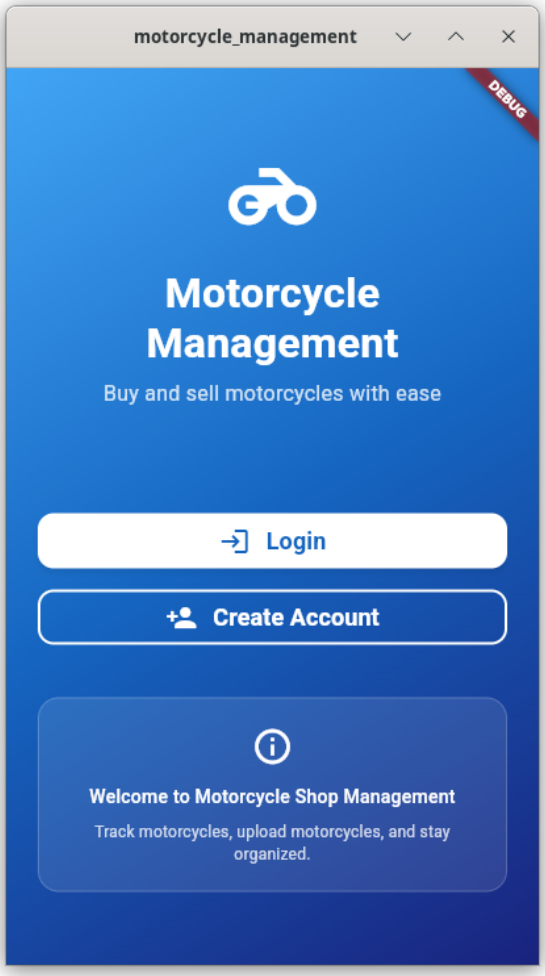
SCREENS

8.1 Navigation Structure

Bottom navigation with 5 tabs in `home.dart`:

1. Home - Browse and search
2. Motorcycles - Product grid
3. Upload - Add products
4. Chat - AI assistant
5. Settings - Profile and orders

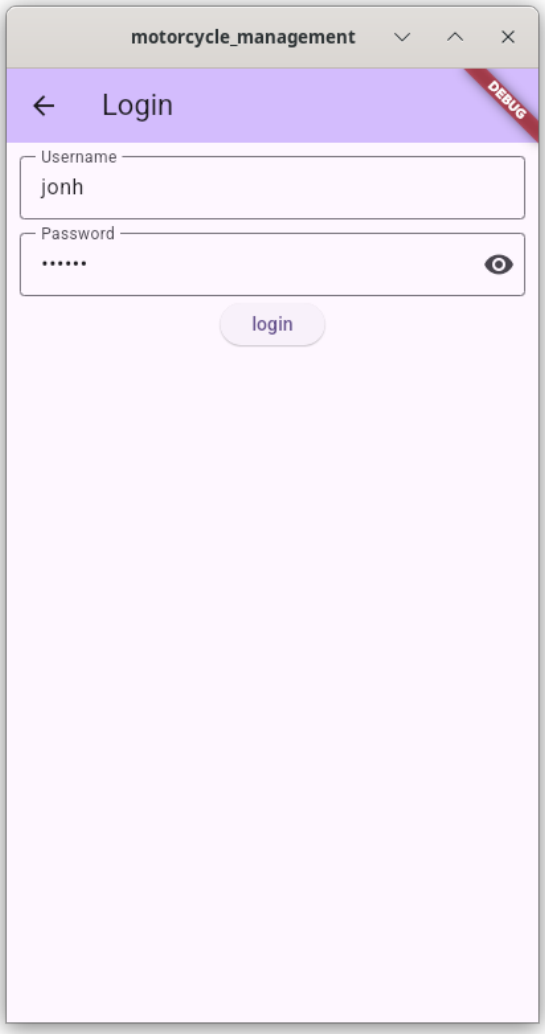
8.2 Start Page (start.dart)



Landing screen with:

- Gradient background
- App title and description
- Login button → `/login`
- Create Account button → `/user`

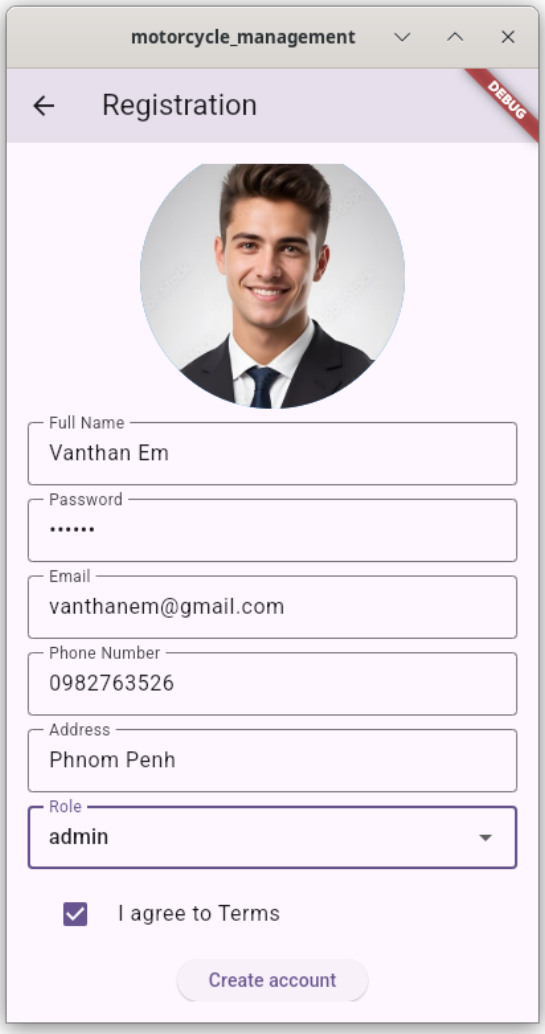
8.3 Login Page (login.dart)



Login form:

- Username field
- Password field with visibility toggle
- Login button → POST /auth/login
- Error handling with SnackBar

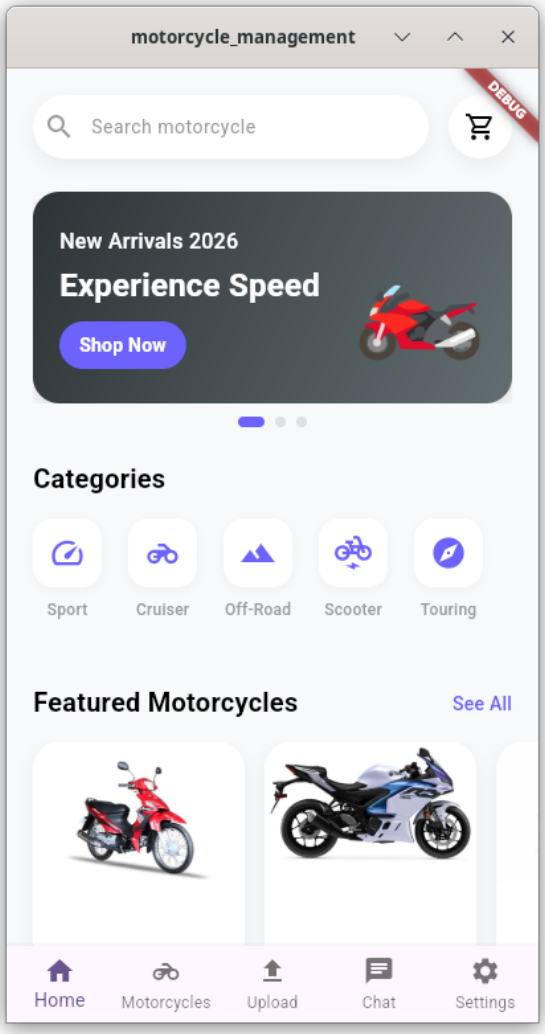
8.4 Registration (user.dart)



Registration form with:

- Full name
- Password
- Email
- Phone number
- Address
- Role dropdown (customer/admin/guest)
- Profile image picker
- Register button → POST /user

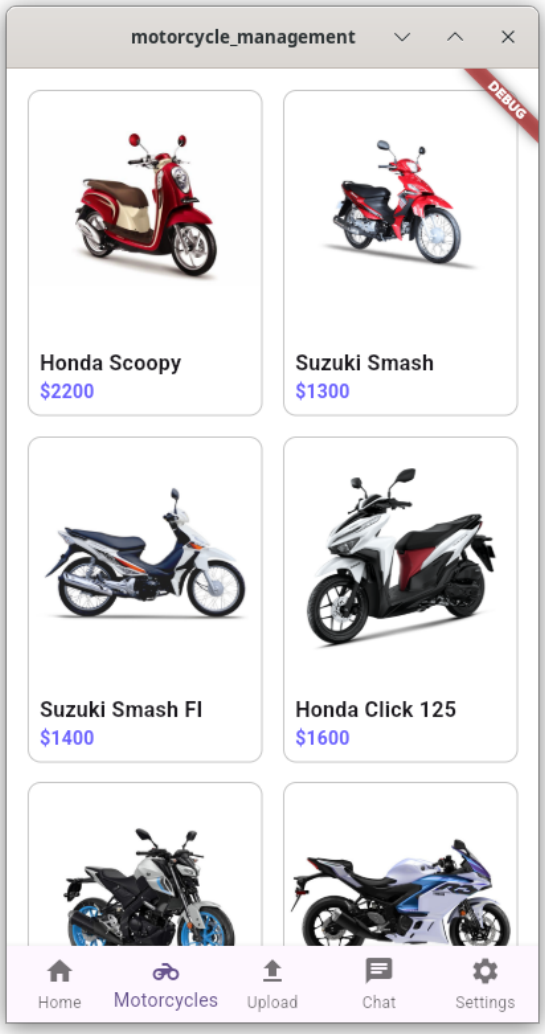
8.5 Home Tab (tab/home.dart)



Features:

- Search bar (submits to SearchResultScreen)
- Cart icon button
- Promo carousel (auto-scroll every 5s)
- Categories (Sport, Cruiser, Off-Road, Scooter, Touring)
- Featured motorcycles horizontal scroll
- Most popular section
- Recommended grid (2 columns)

8.6 Motorcycles Tab (tab/motorcycle.dart)



ProductGridScreen

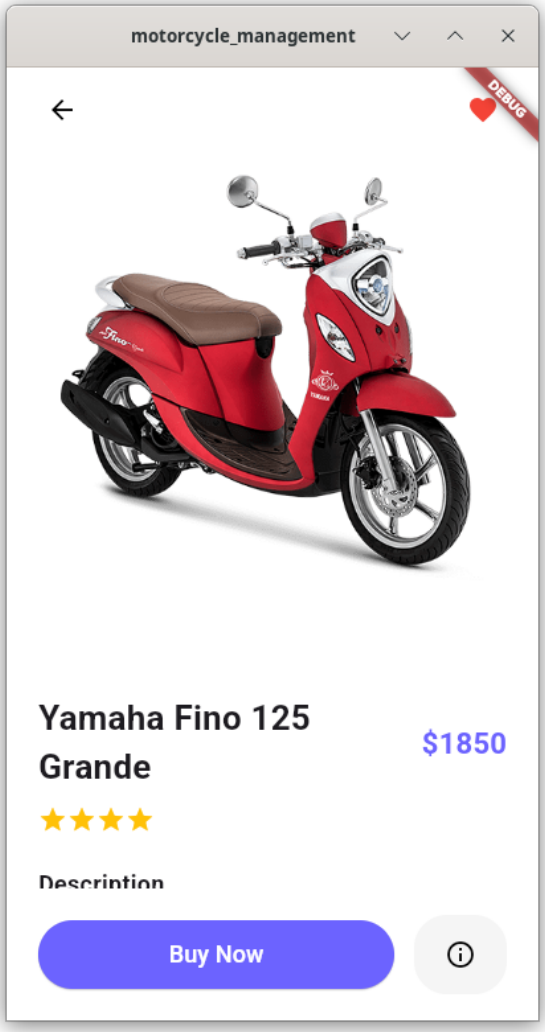
2-column grid of product cards:

- Fetches products from GET /product
- Shows loading spinner
- Each card: image, name, price

ProductCard

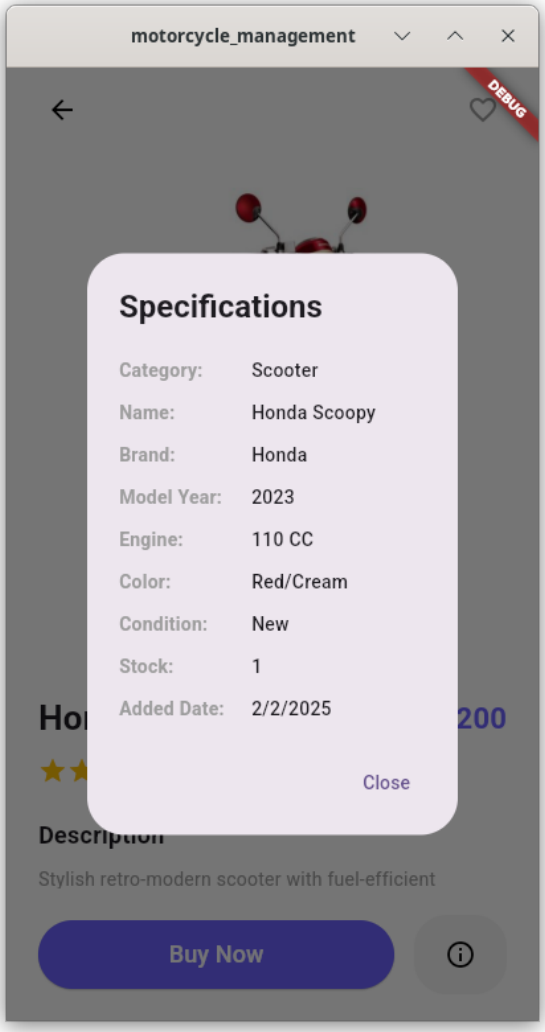
- GestureDetector for tap
- Hero animation for image
- Border decoration
- Price formatted with formatPrice()

ProductDetailScreen



- Large image with Hero animation
- Back button (circle)
- Favorite toggle button
- Product name and price
- Star rating (from calculateStars())
- Description
- Buy Now button → adds to cart
- Info button → shows specifications dialog

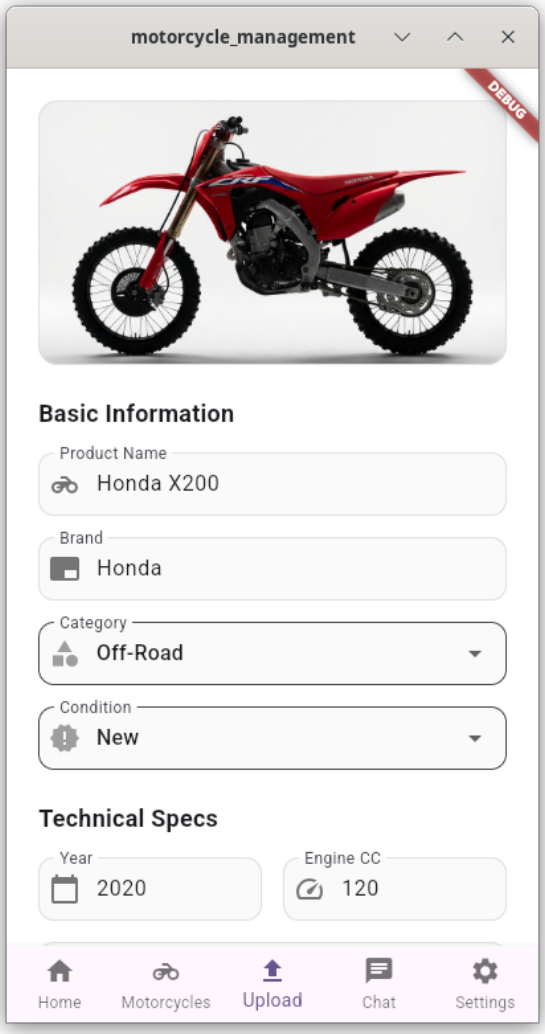
Specifications Dialog



Shows in popup:

- Category name (fetched from /category)
- Name, Brand, Model Year
- Engine CC, Color, Condition
- Stock quantity
- Added date

8.7 Upload Tab (tab/upload.dart)

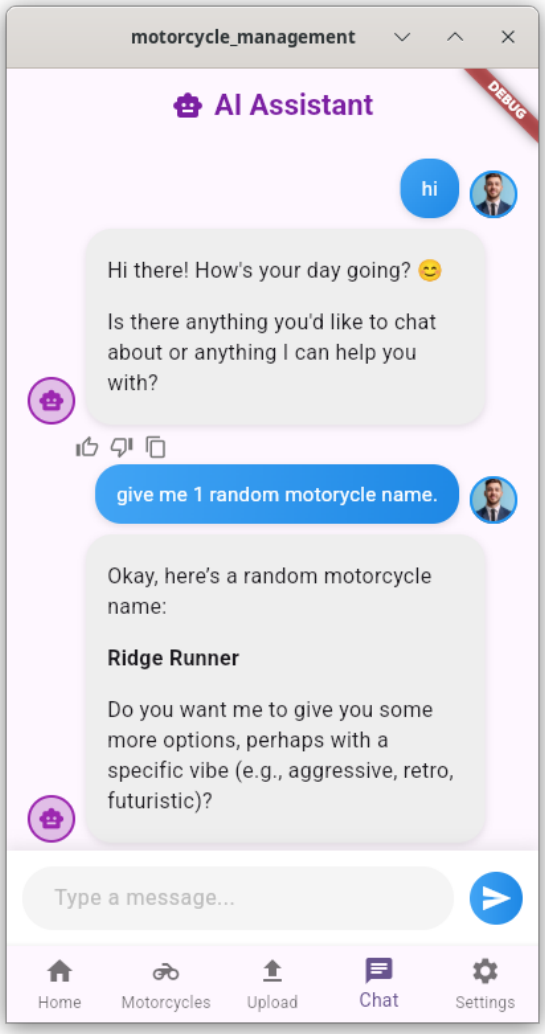


Form for adding new motorcycles:

- Image picker (tap container)
- Name, Brand, Description
- Category dropdown (fetched from /category)
- Condition dropdown (New/Used)
- Model Year, Engine CC, Color
- Price, Stock
- Upload button → POST /product

All fields validated (required).

8.8 Chat Tab (tab/chat.dart)



AI chat interface:

- Message list (user + bot)
- User messages: blue gradient, right side
- Bot messages: grey, left side, markdown
- Thinking indicator with spinner
- Streaming text display
- Text input field
- Send/stop button

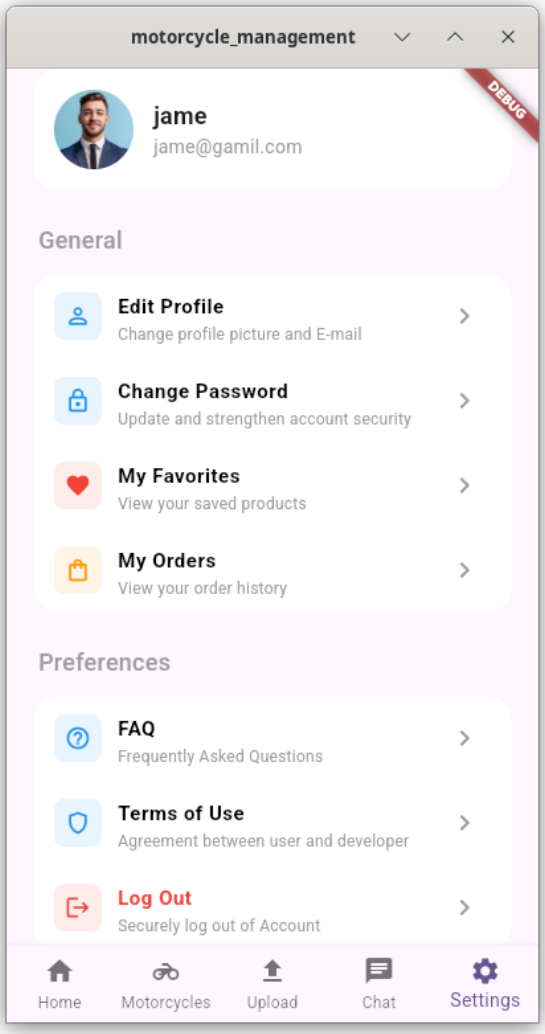
Message actions (bot messages only):

- Like button (green when selected)
- Dislike button (red when selected)

- Copy to clipboard

Uses SSE (Server-Sent Events) for streaming responses.

8.9 Settings Tab (tab/setting.dart)



ProfileSettingScreen

Profile header:

- Avatar image
- Full name
- Email

General section:

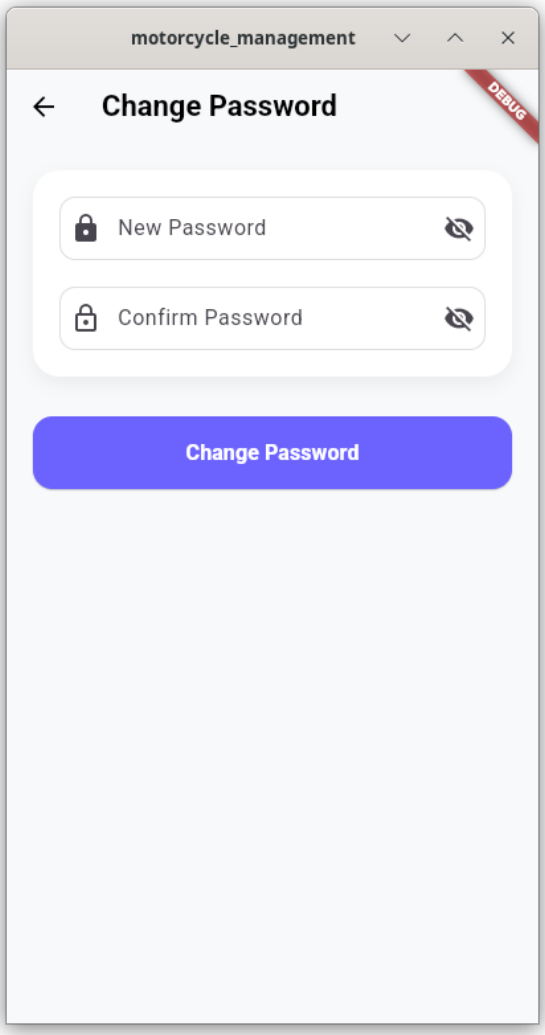
- Edit Profile → EditProfileScreen
- Change Password → ChangePasswordScreen
- My Favorites → FavoritesScreen

- My Orders → OrdersListScreen

Preferences section:

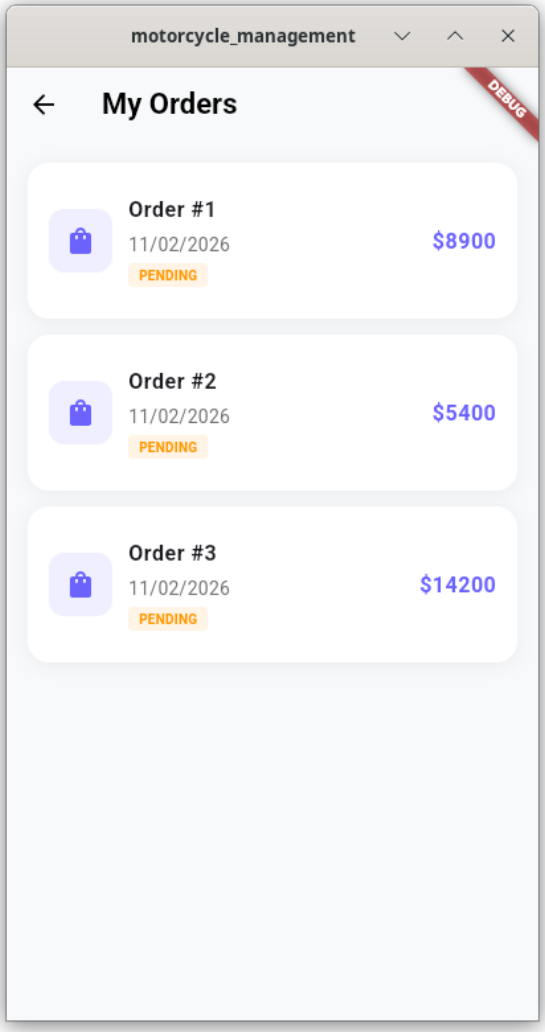
- FAQ → FAQScreen
- Terms of Use → AlertDialog
- Logout → confirmation dialog

ChangePasswordScreen



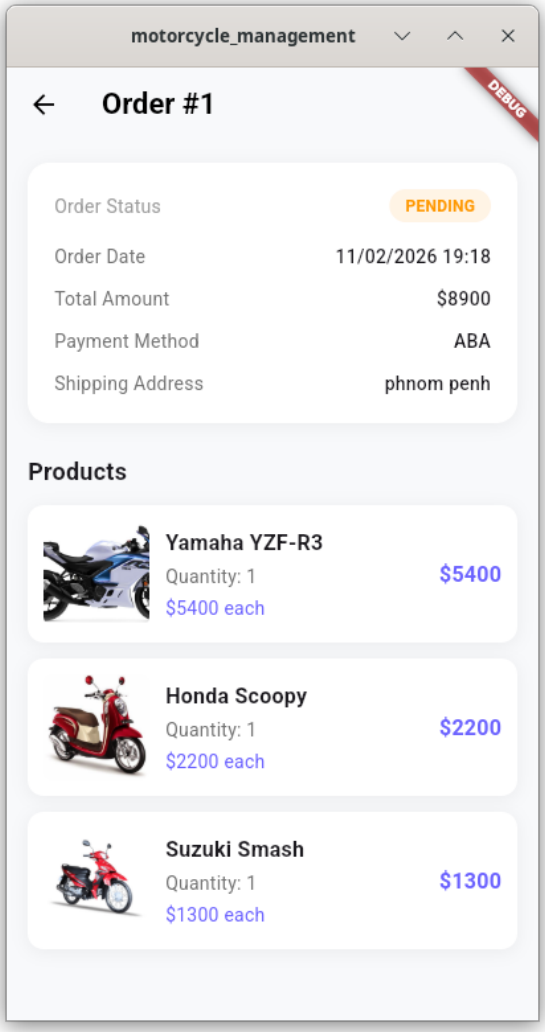
- New password field
- Confirm password field
- Change button → PUT /user/{id}

OrdersListScreen



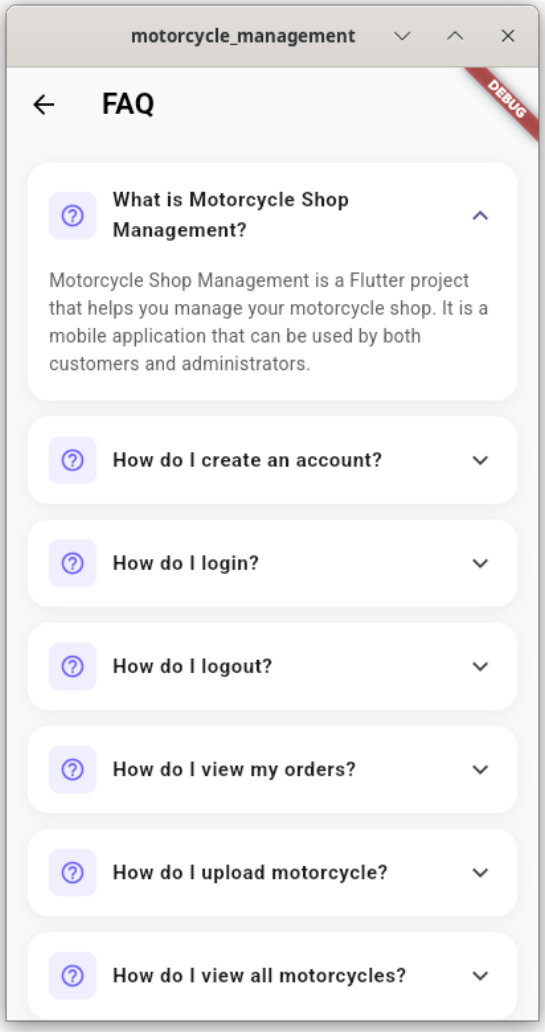
- List of user orders
- Order number, date, status, total
- Status badge (orange for pending, green for completed)
- Tap → OrderDetailScreen

OrderDetailScreen



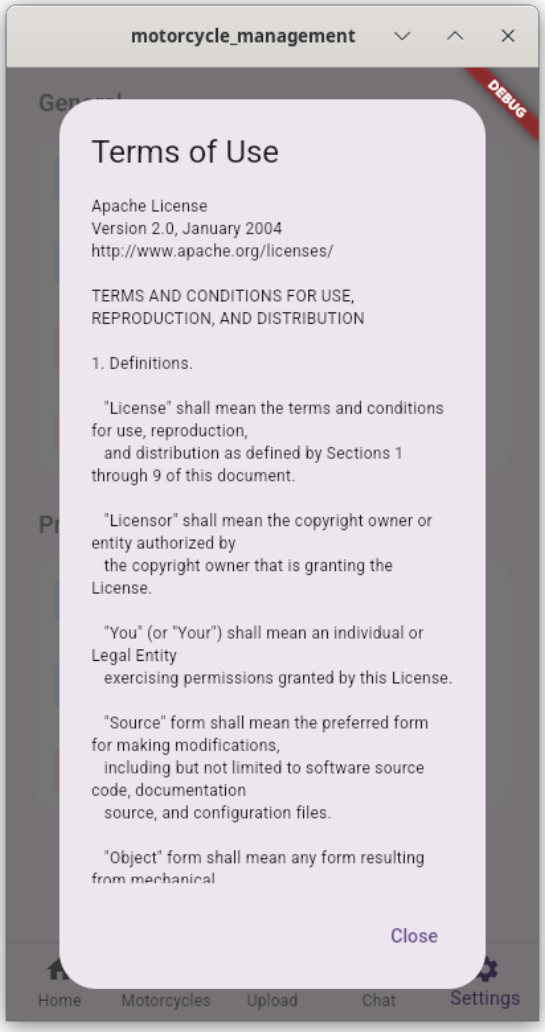
- Order status
- Order date, total, payment method, shipping address
- Products in order (with images)

FAQScreen



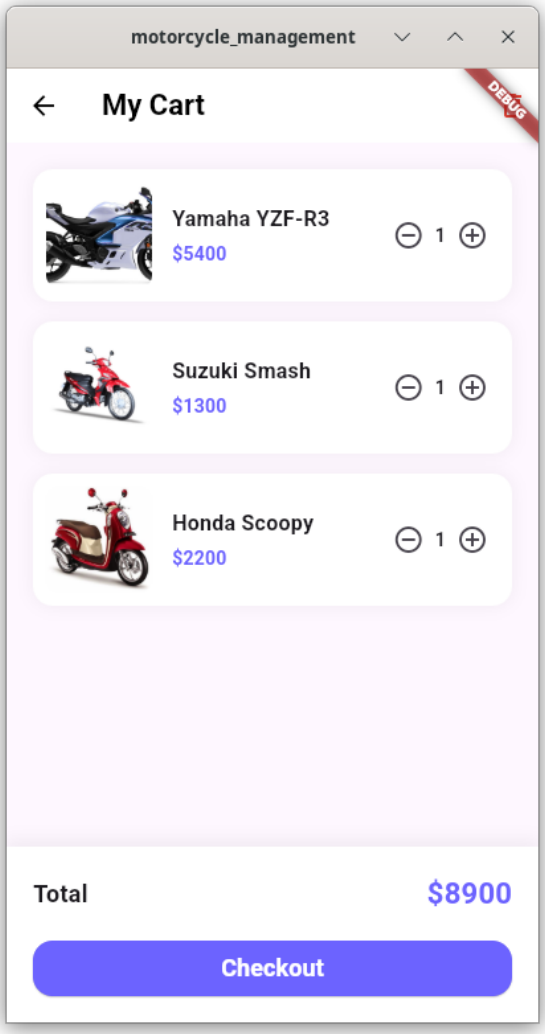
- Expandable list of FAQ items
- Question and answer format

Terms Dialog



- Shows Apache License 2.0 text
- Scrollable content

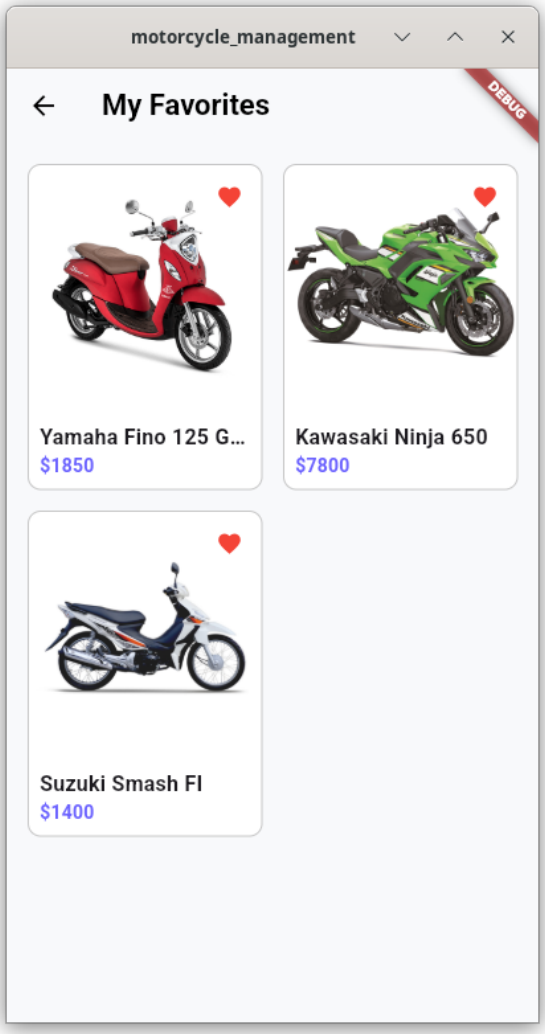
8.10 Cart Screen (tab/cart_screen.dart)



Shopping cart:

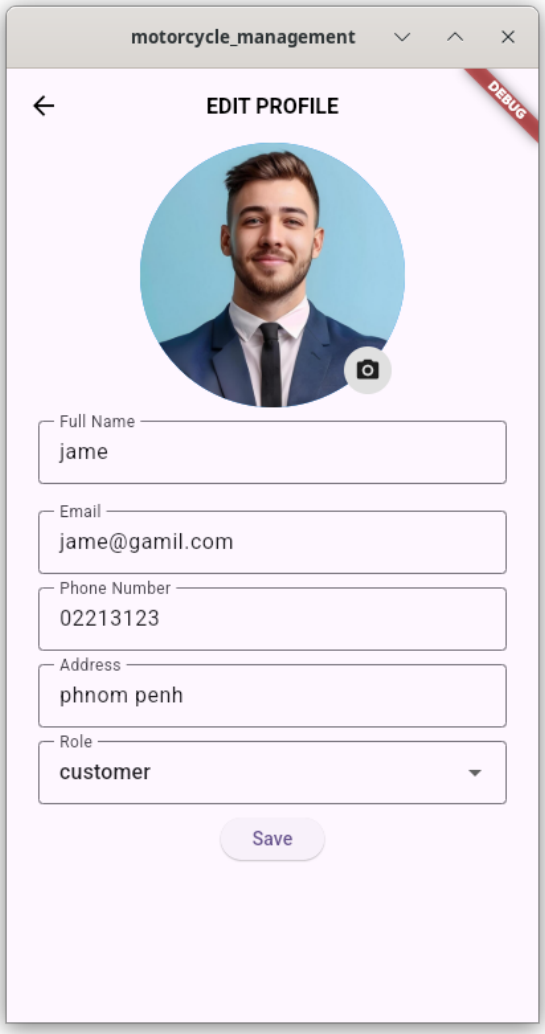
- List of cart items
- Product image, name, price
- Quantity buttons (+/-)
- Remove from cart
- Clear cart button (in app bar)
- Total amount
- Checkout button → POST /order

8.11 Favorites Screen (tab/favorites_screen.dart)



- 2-column grid of favorites
- Heart icon to remove
- Empty state with icon and text
- Tap → ProductDetailScreen

8.12 Edit Profile (tab/edit_profile.dart)



Form with:

- Profile image (tap to change)
- Full name
- Email
- Phone number
- Address
- Role dropdown
- Save button → PUT /user/{id}

CHAPTER

NINE

CONTROLLERS

9.1 Overview

Two singleton controllers manage app state using ChangeNotifier.

9.2 CartController

File: `lib/controller/cart_controller.dart`

Purpose: Manage shopping cart items

9.2.1 State

```dart
final List<Map<String, dynamic>> _items = [];
```

9.2.2 Methods

addToCart(product)

Adds product or increments quantity:

```dart
void addToCart(Map<String, dynamic> product) {
  int index = _items.indexWhere(
    (item) => item['productId'] == product['productId'],
  );

  if (index != -1) {
    _items[index]['quantity'] =
      (_items[index]['quantity'] ?? 1) + 1;
  } else {
    final newProduct = Map<String, dynamic>.from(product);
    newProduct['quantity'] = 1;
    _items.add(newProduct);
  }

  notifyListeners();
}
```

removeFromCart(product)

Removes item completely:

```
void removeFromCart(Map<String, dynamic> product) {
  _items.removeWhere(
    (item) => item['productId'] == product['productId']
  );
  notifyListeners();
}
```

updateQuantity(product, quantity)

Updates item quantity:

```
void updateQuantity(Map<String, dynamic> product, int quantity) {
  int index = _items.indexWhere(
    (item) => item['productId'] == product['productId'],
  );

  if (index != -1) {
    if (quantity <= 0) {
      _items.removeAt(index);
    } else {
      _items[index]['quantity'] = quantity;
    }
    notifyListeners();
  }
}
```

clearCart()

Removes all items:

```
void clearCart() {
  _items.clear();
  notifyListeners();
}
```

9.2.3 Getters

```
// List of cart items
List<Map<String, dynamic>> get items => _items;

// Total quantity of all items
int get itemCount => _items.fold(
  0,
  (sum, item) => sum + (item['quantity'] as int),
);

// Total price
double get totalAmount => _items.fold(0, (sum, item) {
  double price = double.tryParse(item['price'].toString()) ?? 0.0;
  int quantity = item['quantity'] as int;
  return sum + (price * quantity);
});
```

9.2.4 Usage

```
class _CartScreenState extends State<CartScreen> {
  final CartController _cartController = CartController();

  @override
  void initState() {
    super.initState();
    _cartController.addListener(() {
      if (mounted) setState(() {});
    });
  }

  @override
  void dispose() {
    _cartController.removeListener(() {});
    super.dispose();
  }
}
```

9.3 FavoritesController

File: `lib/controller/favorites_controller.dart`

Purpose: Manage favorite products

9.3.1 State

```
final List<Map<String, dynamic>> _favorites = [];
```

9.3.2 Methods

addToFavorites(product)

Adds if not already present:

```
void addToFavorites(Map<String, dynamic> product) {
  if (!isFavorite(product)) {
    _favorites.add(Map<String, dynamic>.from(product));
    notifyListeners();
  }
}
```

removeFromFavorites(product)

```
void removeFromFavorites(Map<String, dynamic> product) {
  _favorites.removeWhere(
    (item) => item['productId'] == product['productId']
  );
  notifyListeners();
}
```

isFavorite(product)

```
bool isFavorite(Map<String, dynamic> product) {
  return _favorites.any(
    (item) => item['productId'] == product['productId']
  );
}
```

toggleFavorite(product)

```
void toggleFavorite(Map<String, dynamic> product) {
  if (isFavorite(product)) {
    removeFromFavorites(product);
  } else {
    addToFavorites(product);
  }
}
```

clearFavorites()

```
void clearFavorites() {
  _favorites.clear();
  notifyListeners();
}
```

9.3.3 Getters

```
List<Map<String, dynamic>> get favorites => _favorites;
int get favoriteCount => _favorites.length;
```

9.4 Singleton Pattern

Both controllers use singleton pattern:

```
class CartController extends ChangeNotifier {
  static final CartController _instance =
    CartController._internal();

  factory CartController() => _instance;

  CartController._internal();
}
```

This ensures the same instance is used throughout the app.

CHAPTER

TEN

API INTEGRATION

10.1 Base URL

From `lib/config.dart`:

```
String get baseUrl => config['apiUrl']!;
```

10.2 API Endpoints

10.2.1 Authentication

POST /auth/login

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

POST /user (Registration)

Multipart request:

```
var request = http.MultipartRequest('POST',
  Uri.parse('$baseUrl/user'));

// JSON part
request.files.add(
  http.MultipartFile.fromString(
    'user',
    jsonEncode({
      'username': fullName,
      'password': password,
      'email': email,
      'phoneNumber': phone,
      'address': address,
      'role': selectedRole,
    }),
```

(continues on next page)

(continued from previous page)

```
    contentType: http.MediaType('application', 'json'),
  ),
);

// Image part (optional)
if (imageFile != null) {
  request.files.add(
    await http.MultipartFile.fromPath(
      'profileImage',
      imageFile.path,
    ),
  );
}

var response = await request.send();
```

PUT /user/{id} (Update)

Same multipart format as POST.

10.2.2 Products

GET /product

```
final response = await http.get(
  Uri.parse('$baseUrl/product'),
);

if (response.statusCode == 200) {
  return jsonDecode(response.body);
}
```

POST /product (Upload)

Multipart with product data and image:

```
var request = http.MultipartRequest(
  'POST',
  Uri.parse('$baseUrl/product'),
);

// Product JSON
request.files.add(
  http.MultipartFile.fromString(
    'product',
    jsonEncode({
      'categoryId': categoryId,
      'name': name,
      'description': description,
      'price': price,
      'stockQuantity': stock,
      'brand': brand,
      'modelYear': year,
```

(continues on next page)

(continued from previous page)

```
      'engineCc': engine,
      'color': color,
      'conditionStatus': condition,
    }),
    contentType: http.MediaType('application', 'json'),
  ),
);

// Product image
request.files.add(
  await http.MultipartFile.fromPath(
    'productImage',
    imageFile.path,
    contentType: MediaType('image', 'jpeg'),
  ),
);
```

10.2.3 Categories

GET /category

```
final response = await http.get(
  Uri.parse('$baseUrl/category'),
);

// Returns list of categories for dropdown
```

10.2.4 Orders

POST /order

```
final response = await http.post(
  Uri.parse('$baseUrl/order'),
  headers: {'Content-Type': 'application/json'},
  body: jsonEncode({
    'userId': userId,
    'orderDate': DateTime.now().toIso8601String(),
    'totalAmount': total,
    'status': 'pending',
    'shippingAddress': address,
    'paymentMethod': 'ABA',
  }),
);
```

GET /order/user/{userId}

```
final response = await http.get(
  Uri.parse('$baseUrl/order/user/$userId'),
);
```

10.2.5 Order Items

POST /orderitem

```
await http.post(
  Uri.parse('$baseUrl/orderitem'),
  headers: {'Content-Type': 'application/json'},
  body: jsonEncode({
    'orderId': orderId,
    'productId': productId,
    'quantity': quantity,
    'unitPrice': price,
  }),
);
```

10.2.6 Chat

POST /chat (SSE streaming)

```
import 'package:flutter_client_sse/flutter_client_sse.dart';

final stream = SSEClient.subscribeToSSE(
  method: SSERequestType.POST,
  url: '$baseUrl/chat',
  header: {'Content-Type': 'application/json'},
  body: {'prompt': userMessage},
);

stream.listen(
  (event) {
    if (event.data != '[DONE]') {
      setState(() {
        _response += event.data ?? '';
      });
    }
  },
);
```

10.3 Error Handling

Basic pattern used in app:

```
try {
  final response = await http.get(Uri.parse(url));

  if (response.statusCode == 200) {
    return jsonDecode(response.body);
  } else {
    throw Exception('Failed: ${response.statusCode}');
  }
} catch (e) {
  // Show error to user
  ScaffoldMessenger.of(context).showSnackBar(
```

(continues on next page)

(continued from previous page)

```
    SnackBar(content: Text('Error: $e')),
  );
  return [];
}
```

10.4 Image URLs

Product images served from:

```
'${config['apiUrl']}/uploads/${product['imageUrl']}'
```

User profile images:

```
'${config['apiUrl']}/uploads/${user['profileImageUrl']}'
```

CHAPTER

ELEVEN

DEPLOYMENT

11.1 Build APK (Android)

Debug build:

```
flutter build apk --debug
```

Release build:

```
flutter build apk --release
```

Output: `build/app/outputs/flutter-apk/app-release.apk`

11.2 Build App Bundle (Play Store)

```
flutter build appbundle --release
```

Output: `build/app/outputs/bundle/release/app-release.aab`

11.3 Build iOS (macOS only)

```
flutter build ios --release
```

Then archive in Xcode and upload to App Store Connect.

11.4 Configuration

Before building:

1. Update version in `pubspec.yaml`:

   ```
   version: 1.0.0+1
   ```

2. Update API URL in `lib/config.dart` to production:

   ```
   var config = {
     "apiUrl": "https://your-api.com",
   };
   ```

3. Verify icons are set:

- Android: `android/app/src/main/res/`
- iOS: `ios/Runner/Assets.xcassets/AppIcon.appiconset/`

11.5 Signing

11.5.1 Android

Create `android/key.properties`:

```
storePassword=your-password
keyPassword=your-password
keyAlias=your-alias
storeFile=/path/to/keystore.jks
```

Update `android/app/build.gradle`:

```
signingConfigs {
    release {
        keyAlias keystoreProperties['keyAlias']
        keyPassword keystoreProperties['keyPassword']
        storeFile file(keystoreProperties['storeFile'])
        storePassword keystoreProperties['storePassword']
    }
}
```

11.5.2 iOS

Configure in Xcode:

- Select Runner
- Signing & Capabilities
- Select your team
- Set unique Bundle ID

11.6 Troubleshooting

Build fails

```
flutter clean
flutter pub get
```

Android: Gradle issues

```
cd android
./gradlew clean
cd ..
```

iOS: CocoaPods issues

```
cd ios
rm -rf Pods Podfile.lock
pod install
cd ..
```

CHAPTER

TWELVE

TROUBLESHOOTING

12.1 Build Issues

flutter pub get fails

```
flutter clean
rm -f pubspec.lock
flutter pub get
```

Android build fails

```
cd android
./gradlew clean
cd ..
flutter pub get
flutter run
```

iOS build fails

```
cd ios
rm -rf Pods Podfile.lock
pod deintegrate
pod install
cd ..
flutter clean
flutter run
```

12.2 API Issues

API calls not working

Check config.dart:

```
// Android emulator
"apiUrl": "http://10.0.2.2:8080"

// iOS simulator
"apiUrl": "http://localhost:8080"

// Physical device
"apiUrl": "http://YOUR_COMPUTER_IP:8080"
```

Images not loading

Add error builder:

```
Image.network(
  url,
  errorBuilder: (context, error, stackTrace) {
    return Icon(Icons.error);
  },
)
```

CORS errors (web)

The app uses mobile-specific features (image picker) that don't work on web. Test on mobile emulator or device.

12.3 State Issues

UI not updating

Ensure listener is set up:

```
@override
void initState() {
  super.initState();
  controller.addListener(() {
    if (mounted) setState(() {});
  });
}

@override
void dispose() {
  controller.removeListener(() {});
  super.dispose();
}
```

12.4 Common Errors

Null check operator on null

Add null checks:

```
if (data != null) {
  return data.length;
}
return 0;
```

setState after dispose

Always check mounted:

```
if (mounted) {
  setState(() {});
}
```

Image picker crashes (iOS)

Add to ios/Runner/Info.plist:

```
<key>NSPhotoLibraryUsageDescription</key>
<string>Upload profile and product images</string>
<key>NSCameraUsageDescription</key>
<string>Take photos for uploads</string>
```

12.5 Debugging

Enable verbose logging:

```
flutter run -v
```

View logs:

```
flutter logs
```

Hot restart during development:

```
Press 'r' in terminal
```

CHAPTER

THIRTEEN

GALLERY

13.1 Additional Collection of Images

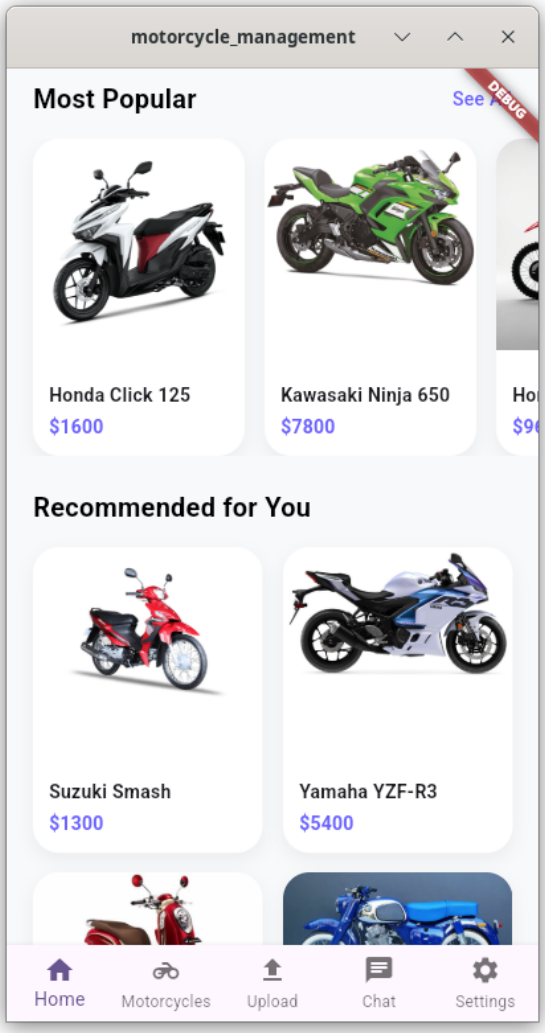


Fig. 1: home screen with popular

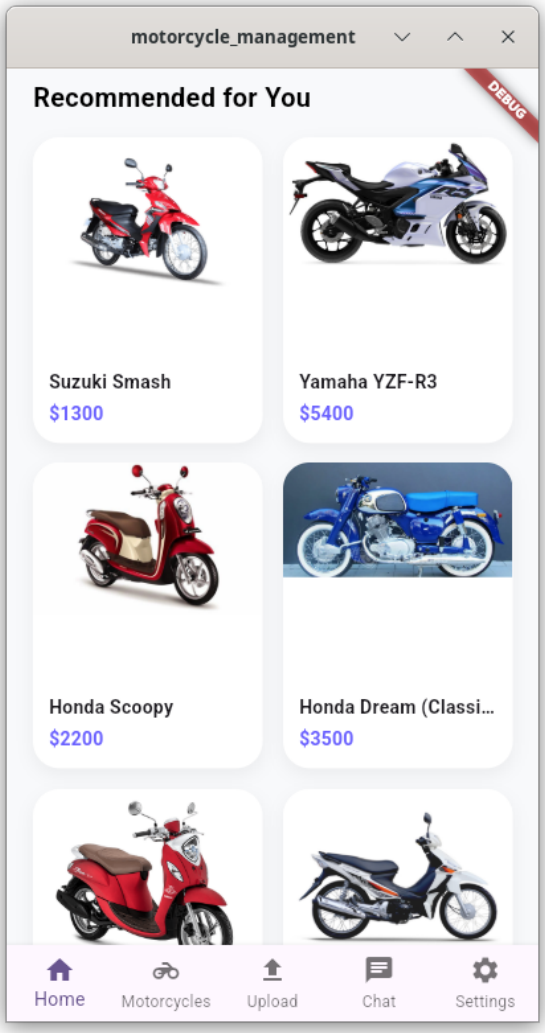


Fig. 2: home screen with recommend

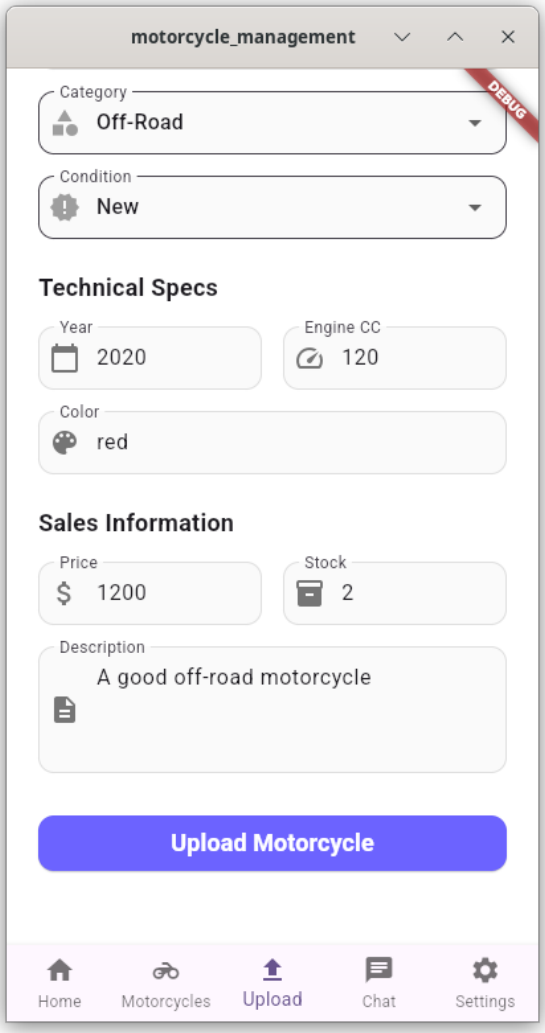


Fig. 3: upload with user input

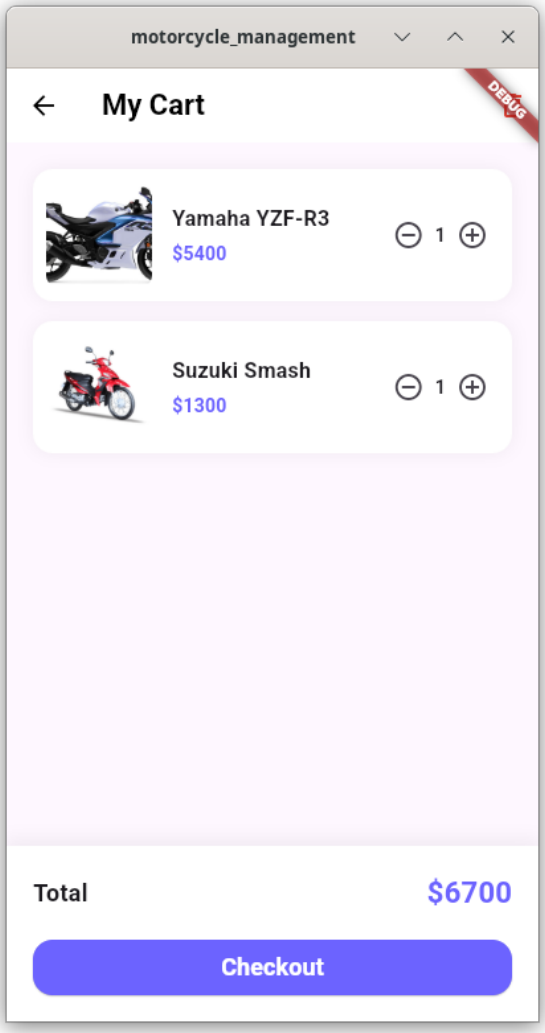


Fig. 4: cart screen

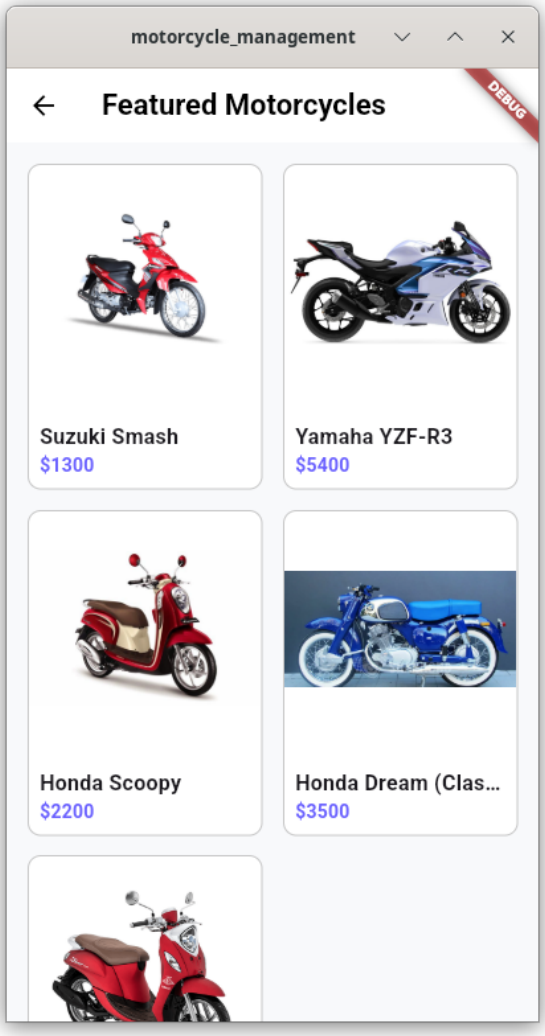


Fig. 5: featured motorcycle screen

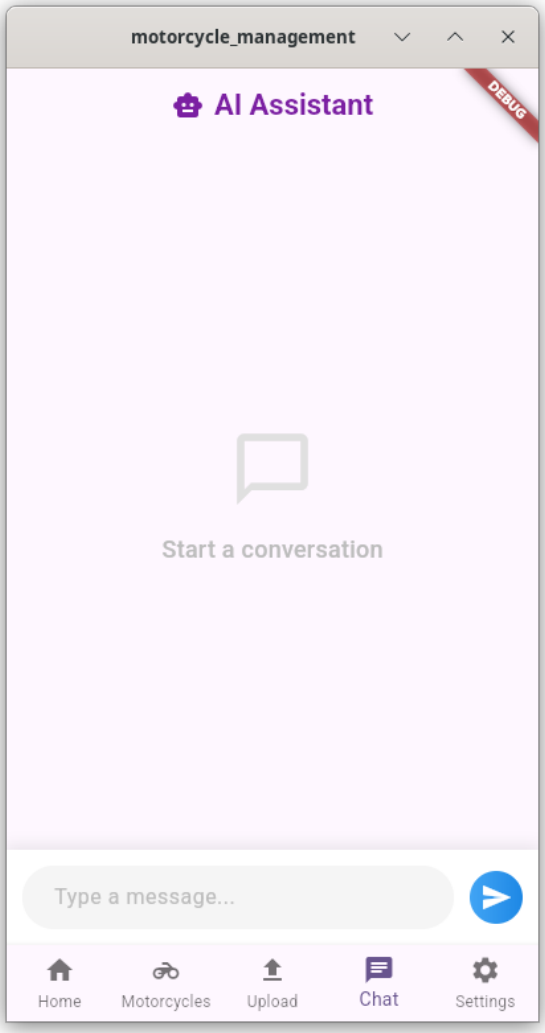


Fig. 6: chat screen with no conversation yet

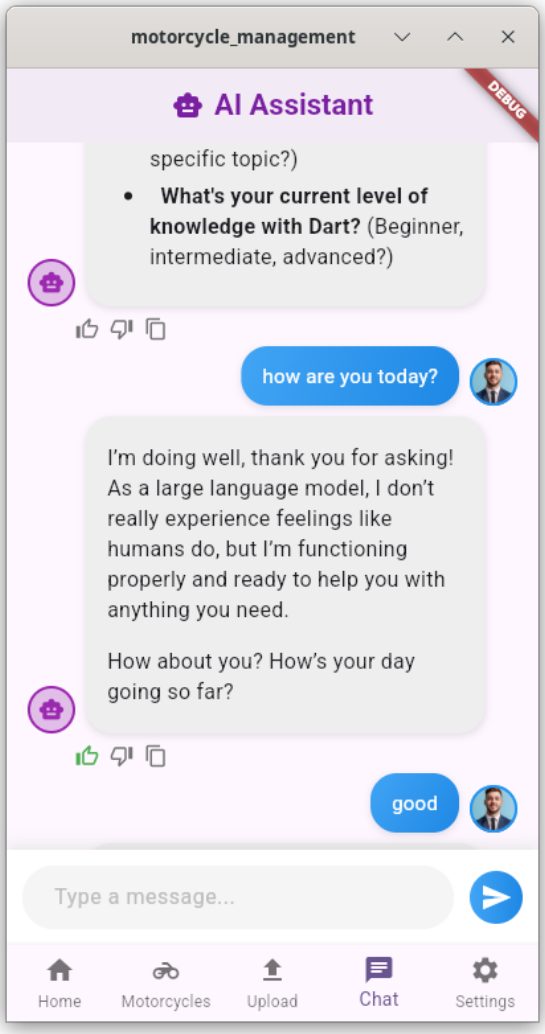


Fig. 7: chat screen with AI assistant conversation

CHAPTER

FOURTEEN

MOTORCYCLE SHOP MANAGEMENT - SPRING BOOT BACKEND

14.1 Backend Introduction

The Spring Boot backend provides the REST API and data persistence layer for the Motorcycle Shop Management application.

14.1.1 Overview

This backend application is built with Spring Boot and provides a comprehensive REST API for managing:

- User authentication and management
- Product catalog with categories
- Order processing and management
- AI-powered chat assistance
- File uploads (profile images, product images)

14.1.2 Architecture

The backend follows a layered architecture pattern:

```
Controller Layer
    |
    v
Service Layer
    |
    v
Repository Layer (Spring Data JPA)
    |
    v
Database (H2/MySQL)
```

14.1.3 Technology Stack

Core Framework

- **Spring Boot**: 4.0.1 - Application framework
- **Java**: 21 - Programming language
- **Gradle**: Build tool with Kotlin DSL

Data Layer

- **Spring Data JPA**: Data access and ORM
- **Hibernate**: JPA provider
- **H2 Database**: In-memory development database
- **MySQL**: Production database support

Security

- **Spring Security**: Authentication and authorization
- **BCrypt**: Password hashing

AI Integration

- **Spring AI**: AI framework integration
- **Ollama**: Local LLM integration (Gemma 3)

Utilities

- **Lombok**: Boilerplate code reduction
- **Jackson**: JSON processing
- **Guava**: Google utilities

14.1.4 Key Features

REST API

Full RESTful API with endpoints for:

- User CRUD operations
- Product management
- Category management
- Order processing
- Authentication
- AI chat streaming

Database

- JPA entities with automatic schema generation
- Support for H2 (development) and MySQL (production)
- Database seeding for initial data

Security

- BCrypt password encryption
- Spring Security configuration
- Session-based authentication
- CORS configuration for frontend access

File Handling

- Multipart file uploads for images
- UUID-based file naming to prevent collisions
- Static resource serving

AI Chat

- Server-Sent Events (SSE) streaming
- Ollama integration for local LLM
- Support for Gemma 3 models

14.1.5 API Base URL

The backend runs on port 8080 by default:

```
http://localhost:8080
```

All API endpoints are prefixed with the base URL. For example:

- `GET http://localhost:8080/user` - List all users
- `POST http://localhost:8080/auth/login` - User login
- `GET http://localhost:8080/product` - List products

14.1.6 Integration with Frontend

The Flutter mobile app communicates with this backend via HTTP requests. The backend provides:

- JSON responses for all API calls
- Proper HTTP status codes
- Error handling with descriptive messages
- Image URLs for serving uploaded files

14.1.7 Next Steps

- *Backend Installation* - Set up the development environment
- *Backend Project Structure* - Understand the codebase organization
- *Backend Configuration* - Configure application properties

14.2 Backend Installation

This guide walks you through setting up the Spring Boot backend development environment.

14.2.1 Prerequisites

Before you begin, ensure you have the following installed:

Required Software

- **Java Development Kit (JDK) 21** or higher

- **Gradle** (or use the included Gradle wrapper)
- **Git** for version control

Optional for AI Features

- **Ollama** - For local AI chat functionality

14.2.2 Install Java

Ubuntu/Debian

```
sudo apt update
sudo apt install openjdk-21-jdk
```

macOS (using Homebrew)

```
brew install openjdk@21
```

Windows

Download and install from Oracle JDK or use OpenJDK.

Verify installation:

```
java -version
# Should show Java 21
```

14.2.3 Clone and Setup

1. Navigate to the backend directory:

```
cd backend
```

2. The project includes Gradle wrapper, so you don't need to install Gradle separately.

14.2.4 Build the Project

Using Gradle wrapper (recommended):

```
./gradlew build
```

On Windows:

```
gradlew.bat build
```

14.2.5 Run the Application

Development Mode

Start the application with hot reload:

```
./gradlew bootRun
```

The application will start on `http://localhost:8080`

Using the Run Script

A convenience script is provided:

```
chmod +x scripts/run.sh
./scripts/run.sh
```

14.2.6 Verify Installation

Once running, test the API:

```
curl http://localhost:8080/user
```

You should receive a JSON response with user data (or an empty array if no users exist).

14.2.7 Setup Ollama (Optional)

For AI chat functionality:

1. Install Ollama from https://ollama.com
2. Pull the required model:

```
ollama pull gemma3:270m
```

3. Ensure Ollama is running on port 11434 (default)
4. The backend will automatically connect to Ollama if available

14.2.8 IDE Configuration

IntelliJ IDEA

1. Open the `backend` folder in IntelliJ
2. IntelliJ will automatically detect the Gradle project
3. Import the project as a Gradle project
4. Enable annotation processing for Lombok: * Settings > Build > Annotation Processors * Enable annotation processing

VS Code

1. Install the Extension Pack for Java
2. Install the Gradle for Java extension
3. Open the `backend` folder
4. The project will be recognized automatically

14.2.9 Development Configuration

The application uses H2 in-memory database by default for development. Data is seeded automatically on startup via `DatabaseSeeder`.

To view the H2 console during development:

1. Navigate to `http://localhost:8080/h2-console`
2. Use the following credentials: * JDBC URL: `jdbc:h2:mem:testdb` * Username: `sa` * Password: (leave empty)

14.2.10 Common Issues

Port 8080 Already in Use

```
# Find process using port 8080
lsof -i :8080
# Kill the process
kill -9 <PID>
```

Or change the port in `application.properties`:

```
server.port=8081
```

Build Fails Due to Java Version

Ensure JAVA_HOME is set correctly:

```
export JAVA_HOME=/usr/lib/jvm/java-21-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
```

14.2.11 Next Steps

- *Backend Configuration* - Learn about application configuration
- *Backend Project Structure* - Explore the codebase structure

14.3 Backend Project Structure

Understanding the organization of the Spring Boot backend codebase.

14.3.1 Directory Layout

```
backend/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           └── demo/
│   │   │               ├── category/          # Category module
│   │   │               ├── controller/        # REST controllers
│   │   │               ├── order/             # Order module
│   │   │               ├── product/           # Product module
│   │   │               ├── user/              # User module
│   │   │               ├── DemoApplication.java
│   │   │               ├── SecurityConfig.java
│   │   │               ├── WebConfig.java
│   │   │               └── DatabaseSeeder.java
│   │   └── resources/
│   │       ├── application.properties
│   │       └── productSeed.json
│   └── test/
│       └── java/
├── build.gradle.kts
```

(continues on next page)

(continued from previous page)

```
├── settings.gradle.kts
├── gradlew
└── scripts/
    └── run.sh
```

14.3.2 Package Structure

The application uses a package-by-feature organization:

Domain Packages

Each domain has its own package containing entity, repository, service, and DTO classes:

User Package (`com.example.demo.user`)

```
user/
├── User.java              # JPA Entity
├── UserRepository.java    # Data access
├── UserService.java       # Business logic
├── CreateUserDto.java     # Create request DTO
└── UpdateUserDto.java     # Update request DTO
```

Product Package (`com.example.demo.product`)

```
product/
├── Product.java           # JPA Entity
├── ProductRepository.java # Data access
├── ProductService.java    # Business logic
├── CreateProductDto.java  # Create request DTO
└── UpdateProductDto.java  # Update request DTO
```

Category Package (`com.example.demo.category`)

```
category/
├── Category.java          # JPA Entity
├── CategoryRepository.java
├── CategoryService.java
├── CreateCategoryDto.java
└── UpdateCategoryDto.java
```

Order Package (`com.example.demo.order`)

```
order/
├── Order.java             # JPA Entity
├── OrderItem.java         # Order item entity
├── OrderRepository.java
├── OrderItemRepository.java
├── OrderService.java
├── OrderItemService.java
├── CreateOrderDto.java
├── UpdateOrderDto.java
├── CreateOrderItemDto.java
└── UpdateOrderItemDto.java
```

Controllers

All REST controllers are in the `controller` package:

```
controller/
├── AuthController.java      # Authentication endpoints
├── UserController.java      # User CRUD operations
├── ProductController.java   # Product management
├── CategoryController.java  # Category management
├── OrderController.java     # Order processing
├── OrderItemController.java # Order items
└── ChatController.java      # AI chat endpoints
```

Configuration Classes

DemoApplication.java

The main entry point with H2 server configuration:

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean(initMethod = "start", destroyMethod = "stop")
    public Server h2Server() throws SQLException {
        return Server.createTcpServer("-tcp", "-tcpAllowOthers", "-tcpPort", "9092");
    }
}
```

SecurityConfig.java

Spring Security configuration with BCrypt password encoding:

```
@Configuration
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(12);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) {
        // CSRF disabled for API access, all requests permitted
    }
}
```

WebConfig.java

Web configuration for serving static files:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
```

(continues on next page)

(continued from previous page)

```
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/uploads/**")
            .addResourceLocations("file:uploads/");
    }
}
```

DatabaseSeeder.java

Initial data loading on application startup:

```
@Component
public class DatabaseSeeder implements CommandLineRunner {
    // Seeds users, categories, and products from JSON
}
```

14.3.3 Build Configuration

build.gradle.kts

The Gradle build file defines dependencies and build configuration:

```
plugins {
    java
    id("org.springframework.boot") version "4.0.1"
    id("io.spring.dependency-management") version "1.1.7"
}

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(21)
    }
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")
    implementation("org.springframework.boot:spring-boot-starter-webmvc")
    implementation("org.springframework.boot:spring-boot-starter-security")
    runtimeOnly("com.h2database:h2")
    runtimeOnly("com.mysql:mysql-connector-j")
    // ... more dependencies
}
```

14.3.4 Resources

application.properties

Configuration file for Spring Boot:

```
spring.application.name=demo
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=update
# ... more configuration
```

productSeed.json

JSON file containing initial product data for seeding the database.

14.3.5 Naming Conventions

- **Classes**: PascalCase (e.g., `UserService`)
- **Methods**: camelCase (e.g., `getUserById`)
- **Variables**: camelCase (e.g., `userRepository`)
- **Constants**: UPPER_SNAKE_CASE
- **Packages**: lowercase (e.g., `com.example.demo.user`)

14.3.6 Design Patterns Used

- **Repository Pattern**: Spring Data JPA repositories
- **Service Layer**: Business logic abstraction
- **DTO Pattern**: Data transfer objects for API requests
- **Builder Pattern**: Lombok `@Builder` for entity construction
- **Dependency Injection**: Spring IoC container

14.3.7 Next Steps

- *Backend Configuration* - Learn about application properties
- *Backend Entities* - Explore JPA entities
- *Backend Controllers* - Understand REST controllers

14.4 Backend Configuration

Comprehensive guide to configuring the Spring Boot backend application.

14.4.1 Application Properties

The main configuration file is located at `src/main/resources/application.properties`.

Basic Configuration

```
spring.application.name=demo
```

14.4.2 Database Configuration

H2 In-Memory Database (Development)

Default configuration for development:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.hibernate.ddl-auto=update
```

This configuration:

- Creates an in-memory H2 database named `testdb`

- Enables SQL logging for debugging
- Automatically creates/updates database schema

H2 Console Access

Access the H2 console at `http://localhost:8080/h2-console`

Connection details:

- **JDBC URL**: `jdbc:h2:mem:testdb`
- **Username**: `sa`
- **Password**: (leave empty)

MySQL Database (Production)

To use MySQL, comment out H2 and uncomment MySQL configuration:

```
# spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.url=jdbc:mysql://localhost:3306/db
spring.datasource.username=root
spring.datasource.password=your_password
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.hibernate.ddl-auto=update
```

Make sure to:

1. Create the database `db` in MySQL
2. Update the username and password
3. Include MySQL connector dependency (already in build.gradle.kts)

14.4.3 JPA/Hibernate Configuration

Schema Generation

```
spring.jpa.hibernate.ddl-auto=update
```

Options:

- `none`: No schema generation
- `update`: Update existing schema (recommended for development)
- `create`: Create schema on startup, destroy on shutdown
- `create-drop`: Create schema on startup, drop on shutdown
- `validate`: Validate schema, make no changes

SQL Logging

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Naming Strategy (Optional)

For legacy naming compatibility:

```
spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.
↪ImplicitNamingStrategyLegacyJpaImpl
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.
↪PhysicalNamingStrategyStandardImpl
```

14.4.4 Server Configuration

Port Configuration

Default port is 8080. To change:

```
server.port=8081
```

Session Configuration

```
server.servlet.session.cookie.same-site=lax
```

14.4.5 Security Configuration

Session-based authentication is configured in `SecurityConfig.java`:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12);
}
```

BCrypt strength (cost factor):

- Default: 10
- Recommended: 12-14
- Higher values = more secure but slower

14.4.6 Multipart File Upload

Configure file upload limits:

```
spring.servlet.multipart.max-file-size=5MB
spring.servlet.multipart.max-request-size=5MB
```

Adjust these values based on your maximum expected file size.

14.4.7 AI/Ollama Configuration

Ollama Base URL

```
spring.ai.ollama.base-url=http://localhost:11434
```

Change if Ollama runs on a different host or port.

Model Strategy

```
spring.ai.ollama.init.pull-model-strategy=when_missing
```

Options:

- `never`: Never pull models automatically
- `when_missing`: Pull only if model doesn't exist (recommended)
- `always`: Always pull on startup

Model Selection

Available models:

```
# Lightweight model (270M parameters)
spring.ai.ollama.chat.options.model=gemma3:270m

# Medium model (1B parameters)
# spring.ai.ollama.chat.options.model=gemma3:1b

# Alternative model
# spring.ai.ollama.chat.options.model=llama3.2:1b
```

14.4.8 Profile-Specific Configuration

Create separate configuration files for different environments:

application-dev.properties (Development)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=create-drop
logging.level.org.springframework=DEBUG
```

application-prod.properties (Production)

```
spring.datasource.url=jdbc:mysql://localhost:3306/production_db
spring.jpa.hibernate.ddl-auto=validate
logging.level.org.springframework=WARN
```

Activate a profile:

```
./gradlew bootRun --args='--spring.profiles.active=dev'
```

Or set environment variable:

```
export SPRING_PROFILES_ACTIVE=prod
```

14.4.9 Logging Configuration

Configure logging levels in `application.properties`:

```
logging.level.root=INFO
logging.level.com.example.demo=DEBUG
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate.SQL=DEBUG
```

14.4.10 Complete Configuration Example

```
spring.application.name=demo

# Database
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.hibernate.ddl-auto=update

# Security
server.servlet.session.cookie.same-site=lax

# File Upload
spring.servlet.multipart.max-file-size=5MB
spring.servlet.multipart.max-request-size=5MB

# AI/Ollama
spring.ai.ollama.base-url=http://localhost:11434
spring.ai.ollama.init.pull-model-strategy=when_missing
spring.ai.ollama.chat.options.model=gemma3:270m

# Logging
logging.level.com.example.demo=DEBUG
```

14.4.11 Environment Variables

You can use environment variables in `application.properties`:

```
spring.datasource.password=${DB_PASSWORD:default_password}
spring.ai.ollama.base-url=${OLLAMA_URL:http://localhost:11434}
```

Set environment variables:

```
export DB_PASSWORD=secret_password
export OLLAMA_URL=http://192.168.1.100:11434
```

14.4.12 Next Steps

- *Backend Entities* - Learn about JPA entities
- *Backend Security* - Security configuration details

14.5 Backend Entities

JPA entity classes define the data model for the Motorcycle Shop Management application.

14.5.1 Overview

The backend uses JPA (Jakarta Persistence API) with Hibernate as the provider. Entities map Java classes to database tables.

14.5.2 Common Annotations

Entity Declaration

```
@Entity
@Table(name = "TableName")
public class EntityName {
    // fields
}
```

Lombok Annotations

```
@Data                    // Generates getters, setters, toString, equals, hashCode
@Builder                 // Builder pattern
@NoArgsConstructor       // Empty constructor
@AllArgsConstructor      // All-args constructor
```

Primary Key

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Column Configuration

```
@Column(nullable = false, length = 100)
private String name;

@Column(columnDefinition = "TEXT")
private String description;

@Column(updatable = false)
private LocalDateTime createdAt;
```

14.5.3 Entity Relationships

Many-to-One

```
@ManyToOne
@JoinColumn(name = "foreign_key_column")
private RelatedEntity related;
```

14.5.4 User Entity

Stores user account information.

```
@Entity
@Table(name = "Users")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
```

(continues on next page)

(continued from previous page)

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(nullable = false, length = 100, unique = true)
    private String fullName;

    @Column(unique = true, nullable = false, length = 100)
    private String email;

    @Column(nullable = false)
    private String passwordHash;

    private String phoneNumber;

    @Column(columnDefinition = "TEXT")
    private String address;

    @Builder.Default
    private String role = "customer";

    private String profileImageUrl;

    @Builder.Default
    @Column(updatable = false)
    private LocalDateTime createdAt = LocalDateTime.now();
}
```

Fields

- `userId`: Primary key, auto-generated
- `fullName`: Username, unique, required (100 chars max)
- `email`: Unique, required (100 chars max)
- `passwordHash`: BCrypt hashed password
- `phoneNumber`: Optional contact number
- `address`: Optional, stored as TEXT
- `role`: User role, defaults to "customer"
- `profileImageUrl`: Filename of uploaded profile image
- `createdAt`: Timestamp, set on creation

14.5.5 Product Entity

Stores motorcycle product information.

```
@Entity
@Table(name = "Products")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
```

(continues on next page)

(continued from previous page)

```
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long productId;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    @Column(nullable = false, length = 150)
    private String name;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal price;

    @Builder.Default
    private Integer stockQuantity = 0;

    private String imageUrl;
    private String brand;
    private Integer modelYear;
    private Integer engineCc;
    private String color;

    @Builder.Default
    private String conditionStatus = "new";

    @Builder.Default
    @Column(updatable = false)
    private LocalDateTime createdAt = LocalDateTime.now();
}
```

Fields

- `productId`: Primary key
- `category`: Many-to-one relationship with Category
- `name`: Product name, required (150 chars max)
- `description`: Optional detailed description (TEXT)
- `price`: Decimal with 10 digits, 2 decimal places
- `stockQuantity`: Available quantity, defaults to 0
- `imageUrl`: Product image filename
- `brand`: Manufacturer brand
- `modelYear`: Manufacturing year
- `engineCc`: Engine displacement in CC
- `color`: Product color

- `conditionStatus`: “new” or “used”, defaults to “new”
- `createdAt`: Creation timestamp

14.5.6 Category Entity

Stores product categories.

```
@Entity
@Table(name = "Categories")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long categoryId;

    @Column(nullable = false, length = 50)
    private String name;

    @Column(columnDefinition = "TEXT")
    private String description;

    private String imageUrl;
}
```

Fields

- `categoryId`: Primary key
- `name`: Category name, required (50 chars max)
- `description`: Optional category description
- `imageUrl`: Category image filename

Default Categories

- Sport
- Cruiser
- Off-Road
- Scooter
- Touring

14.5.7 Order Entity

Stores customer orders.

```
@Entity
@Table(name = "Orders")
@Data
@Builder
@NoArgsConstructor
```

(continues on next page)

(continued from previous page)

```
@AllArgsConstructor
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderId;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Builder.Default
    private LocalDateTime orderDate = LocalDateTime.now();

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal totalAmount;

    @Builder.Default
    private String status = "pending";

    @Column(nullable = false, columnDefinition = "TEXT")
    private String shippingAddress;

    private String paymentMethod;
}
```

Fields

- `orderId`: Primary key
- `user`: Many-to-one relationship with User (required)
- `orderDate`: Order creation timestamp
- `totalAmount`: Order total (decimal)
- `status`: Order status (“pending”, “processing”, “shipped”, “delivered”, “cancelled”)
- `shippingAddress`: Delivery address (required)
- `paymentMethod`: Payment method used

14.5.8 OrderItem Entity

Stores individual items within an order.

```
@Entity
@Table(name = "OrderItems")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class OrderItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderItemId;
```

(continues on next page)

(continued from previous page)

```
    @ManyToOne
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;

    @ManyToOne
    @JoinColumn(name = "product_id", nullable = false)
    private Product product;

    @Column(nullable = false)
    private Integer quantity;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal unitPrice;
}
```

Fields

- `orderItemId`: Primary key
- `order`: Many-to-one relationship with Order
- `product`: Many-to-one relationship with Product
- `quantity`: Number of items
- `unitPrice`: Price per unit at time of order

14.5.9 Entity Diagram

```
+-------------+       +-------------+       +-------------+
|    User     |       |    Order    |       |  OrderItem  |
+-------------+       +-------------+       +-------------+
| PK userId   |<------| FK user_id  |       | PK itemId   |
| fullName    |       | PK orderId  |<------| FK order_id |
| email       |       | orderDate   |       | FK productId|
| passwordHash|       | totalAmount |       | quantity    |
| role        |       | status      |       | unitPrice   |
+-------------+       | shippingAddr|       +-------------+
                      +-------------+            |
                                                 |
                      +-------------+            |
                      |   Product   |<-----------+
                      +-------------+
                      | PK productId|
                      | name        |
                      | price       |
                      | FK category |
                      +-------------+
                            |
                            v
                      +-------------+
                      |   Category  |
                      +-------------+
                      | PK category |
```

(continues on next page)

(continued from previous page)

```
                        | name         |
                        +--------------+
```

14.5.10 Additional Entities

Review Entity

Stores product reviews (simplified version in codebase).

```
@Entity
public class Review {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long reviewId;

    private Integer rating;
    private String comment;

    @ManyToOne
    private User user;

    @ManyToOne
    private Product product;
}
```

Service Entity

Stores service appointments (simplified version).

```
@Entity
public class Service {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long serviceId;

    private String serviceType;
    private LocalDateTime appointmentDate;
    private String status;

    @ManyToOne
    private User user;
}
```

14.5.11 Best Practices

1. **Always use wrapper types** (Long, Integer) instead of primitives for IDs to allow null
2. **Use @Builder.Default** for default field values when using @Builder
3. **Set upddatable=false** on creation timestamps
4. **Use appropriate column types** (TEXT for long strings, DECIMAL for money)
5. **Add unique constraints** on fields that must be unique

6. **Use nullable=false** for required fields

14.5.12 Next Steps

- *Backend Controllers* - REST API controllers
- *Backend Services* - Business logic layer

14.6 Backend Controllers

REST controllers handle HTTP requests and define the API endpoints for the application.

14.6.1 Overview

Controllers use Spring's `@RestController` annotation and handle:

- Request mapping (URL paths and HTTP methods)
- Request parsing (JSON, form data, multipart)
- Response generation (JSON responses)
- HTTP status codes
- Error handling

14.6.2 Base Annotations

```
@RestController
@RequestMapping("/api-path")
public class ControllerName {
    // endpoints
}
```

Common Method Annotations:

- `@GetMapping` - Handle GET requests
- `@PostMapping` - Handle POST requests
- `@PutMapping` - Handle PUT requests
- `@DeleteMapping` - Handle DELETE requests
- `@PathVariable` - Extract URL parameters
- `@RequestBody` - Parse JSON request body
- `@RequestParam` - Extract query parameters
- `@RequestPart` - Handle multipart form data

14.6.3 AuthController

Handles user authentication.

Base Path: `/auth`

```java
@RestController
@RequestMapping("/auth")
public class AuthController {

    private final UserService userService;

    public AuthController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/login")
    public ResponseEntity<?> loginUser(@RequestBody Map<String, String> credentials) {
        String username = credentials.get("username");
        String password = credentials.get("password");

        if (username == null || password == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Map.of("error", "username or password cannot be null"));
        }

        return userService.loginUser(username, password);
    }
}
```

Endpoints

Method	Endpoint	Description
POST	`/auth/login`	Authenticate user with username and password

Request Body:

```
{
    "username": "jame",
    "password": "123456"
}
```

Response:

```
{
    "userId": 1
}
```

14.6.4 UserController

Handles user CRUD operations with profile image upload.

Base Path: `/user`

```java
@RestController
@RequestMapping("/user")
public class UserController {
```

(continues on next page)

(continued from previous page)

```
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUser() {
        return userService.getAllUser();
    }

    @GetMapping("/{id}")
    public User getUser(@PathVariable long id) {
        return userService.getUser(id);
    }

    @PostMapping(consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    public User createUser(
            @RequestPart("user") CreateUserDto userDto,
            @RequestPart(value = "profileImage", required = false) MultipartFile file)
            throws IOException {

        String uploadDir = "uploads/";
        Files.createDirectories(Paths.get(uploadDir));

        String filename = UUID.randomUUID() + "_" + file.getOriginalFilename();
        Path filePath = Paths.get(uploadDir, filename);
        Files.copy(file.getInputStream(), filePath);

        return userService.createUser(userDto, filename);
    }

    @PutMapping(value = "/{id}", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    public User updateUser(
            @PathVariable long id,
            @RequestPart("user") UpdateUserDto dto,
            @RequestPart(value = "profileImage", required = false) MultipartFile file)
            throws IOException {

        if (file != null && !file.isEmpty()) {
            // Handle file upload similar to create
        }

        return userService.updateUser(id, dto);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable long id) {
        userService.deleteUser(id);
    }
}
```

Endpoints

Method	Endpoint	Description
GET	`/user`	List all users
GET	`/user/{id}`	Get user by ID
POST	`/user`	Create new user with profile image
PUT	`/user/{id}`	Update user with optional image
DELETE	`/user/{id}`	Delete user

Create User Request (multipart/form-data):

- **user**: JSON string with user data
- **profileImage**: Image file (optional)

Example user JSON:

```
{
    "username": "john_doe",
    "email": "john@example.com",
    "password": "secure123",
    "phoneNumber": "1234567890",
    "address": "123 Main St",
    "role": "customer"
}
```

14.6.5 ProductController

Manages product catalog with image upload.

Base Path: `/product`

Key Endpoints:

Method	Endpoint	Description
GET	`/product`	List all products
GET	`/product/{id}`	Get product by ID
GET	`/product/category/{categoryId}`	Get products by category
POST	`/product`	Create product with image
PUT	`/product/{id}`	Update product
DELETE	`/product/{id}`	Delete product

14.6.6 CategoryController

Manages product categories.

Base Path: `/category`

Endpoints:

Method	Endpoint	Description
GET	`/category`	List all categories
GET	`/category/{id}`	Get category by ID
POST	`/category`	Create category
PUT	`/category/{id}`	Update category
DELETE	`/category/{id}`	Delete category

14.6.7 OrderController

Handles order processing.

Base Path: `/order`

Endpoints:

Method	Endpoint	Description
GET	`/order`	List all orders
GET	`/order/{id}`	Get order by ID
GET	`/order/user/{userId}`	Get orders for specific user
POST	`/order`	Create new order
PUT	`/order/{id}`	Update order
DELETE	`/order/{id}`	Delete order

Create Order Request:

```
{
    "userId": 1,
    "totalAmount": 15999.99,
    "shippingAddress": "123 Main St, City, Country",
    "paymentMethod": "credit_card",
    "items": [
        {
            "productId": 1,
            "quantity": 2,
            "unitPrice": 7999.99
        }
    ]
}
```

14.6.8 OrderItemController

Manages individual order items.

Base Path: `/order-item`

Endpoints:

Method	Endpoint	Description
GET	`/order-item`	List all order items
GET	`/order-item/{id}`	Get order item by ID
GET	`/order-item/order/{orderId}`	Get items for specific order
POST	`/order-item`	Create order item
PUT	`/order-item/{id}`	Update order item
DELETE	`/order-item/{id}`	Delete order item

14.6.9 ChatController

Provides AI chat functionality with Server-Sent Events (SSE).

Base Path: `/chat`

```
@RestController
@RequestMapping("/chat")
public class ChatController {

    private final ChatModel chatModel;

    public ChatController(ChatModel chatModel) {
        this.chatModel = chatModel;
    }

    @PostMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<String> generate(@RequestBody Map<String, String> request) {
        String prompt = request.get("prompt");

        if (prompt == null || prompt.isBlank()) {
            return Flux.just("error: prompt is required");
        }

        return chatModel.stream(prompt).concatWithValues("[DONE]");
    }
}
```

Endpoint:

Method	Endpoint	Content-Type	Description
POST	`/chat`	text/event-stream	Stream AI responses

Request:

```
{
    "prompt": "Tell me about motorcycles"
}
```

Response: SSE stream

```
data: Motorcycles are two-wheeled vehicles...
```

(continues on next page)

(continued from previous page)

```
data: They come in various types including...

data: [DONE]
```

14.6.10 HomeController

Simple controller for root path.

Base Path: `/`

```
@RestController
public class HomeController {

    @GetMapping("/")
    public String home() {
        return "Motorcycle Shop Management API";
    }
}
```

14.6.11 ResponseEntity Patterns

Use `ResponseEntity` for full control over responses:

```
// Success with data
return ResponseEntity.ok(user);

// Created
return ResponseEntity.status(HttpStatus.CREATED).body(user);

// Bad request
return ResponseEntity.badRequest()
    .body(Map.of("error", "Invalid input"));

// Not found
return ResponseEntity.status(HttpStatus.NOT_FOUND)
    .body(Map.of("error", "User not found"));

// Unauthorized
return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
    .body(Map.of("error", "Invalid credentials"));
```

14.6.12 Error Handling

Use try-catch for error handling:

```
@GetMapping("/{id}")
public ResponseEntity<?> getUser(@PathVariable long id) {
    try {
        User user = userService.getUser(id);
        return ResponseEntity.ok(user);
    } catch (NoSuchElementException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
```

(continues on next page)

(continued from previous page)

```
            .body(Map.of("error", "User not found"));
    }
}
```

14.6.13 CORS Configuration

CORS is configured in `WebConfig.java` to allow frontend access:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("GET", "POST", "PUT", "DELETE");
    }
}
```

14.6.14 Complete API Summary

Endpoint	Methods	Description
`/`	GET	API info
`/auth/login`	POST	User login
`/user`	GET, POST	List/Create users
`/user/{id}`	GET, PUT, DELETE	User operations
`/product`	GET, POST	List/Create products
`/product/{id}`	GET, PUT, DELETE	Product operations
`/category`	GET, POST	List/Create categories
`/category/{id}`	GET, PUT, DELETE	Category operations
`/order`	GET, POST	List/Create orders
`/order/{id}`	GET, PUT, DELETE	Order operations
`/order/user/{userId}`	GET	User orders
`/order-item`	GET, POST	List/Create items
`/chat`	POST	AI chat stream

14.6.15 Next Steps

- *Backend Services* - Business logic implementation
- *Backend AI Integration* - AI chat details

14.7 Backend Services

Services contain the business logic and act as an intermediary between controllers and repositories.

14.7.1 Overview

Services use the `@Service` annotation and handle:

- Business logic and rules
- Data transformation
- Transaction management
- Integration with repositories
- Password encoding
- Data validation

14.7.2 Service Pattern

```
@Service
public class ServiceName {

    @Autowired
    private RepositoryName repository;

    // Business methods
}
```

14.7.3 UserService

Manages user-related business logic including authentication.

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder encoder;

    public User createUser(CreateUserDto dto, String filename) {
        User user = User.builder()
            .fullName(dto.username())
            .email(dto.email())
            .passwordHash(encoder.encode(dto.password()))
            .phoneNumber(dto.phoneNumber())
            .address(dto.address())
            .role(dto.role())
            .profileImageUrl(filename)
            .build();
        return userRepository.save(user);
    }

    public List<User> getAllUser() {
        return (List<User>) userRepository.findAll();
    }
```

(continues on next page)

(continued from previous page)

```
    public User getUser(long id) {
        return userRepository.findById(id).get();
    }

    public User updateUser(long id, UpdateUserDto dto) {
        User user = userRepository.findById(id).get();

        if (dto.username().isPresent()) {
            user.setFullName(dto.username().get());
        }
        if (dto.email().isPresent()) {
            user.setEmail(dto.email().get());
        }
        if (dto.password().isPresent()) {
            user.setPasswordHash(encoder.encode(dto.password().get()));
        }
        if (dto.phoneNumber().isPresent()) {
            user.setPhoneNumber(dto.phoneNumber().get());
        }
        if (dto.address().isPresent()) {
            user.setAddress(dto.address().get());
        }
        if (dto.role().isPresent()) {
            user.setRole(dto.role().get());
        }
        if (dto.profileImageUrl().isPresent()) {
            user.setProfileImageUrl(dto.profileImageUrl().get());
        }

        return userRepository.save(user);
    }

    public void deleteUser(long id) {
        userRepository.deleteById(id);
    }

    public ResponseEntity<?> loginUser(String username, String password) {
        User user = userRepository.findByFullName(username);

        if (user == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Map.of("error", "Invalid username"));
        }

        if (!encoder.matches(password, user.getPasswordHash())) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Map.of("error", "Invalid password"));
        }

        return ResponseEntity.ok(Map.of("userId", user.getUserId()));
    }
```

(continues on next page)

(continued from previous page)

```
    public long count() {
        return userRepository.count();
    }
}
```

Key Methods

Method	Description
`createUser`	Hash password and save new user
`getAllUser`	Retrieve all users
`getUser`	Get user by ID
`updateUser`	Update user fields selectively
`deleteUser`	Remove user by ID
`loginUser`	Authenticate and return user ID
`count`	Get total user count

14.7.4 ProductService

Manages product catalog operations.

Key Methods:

Method	Description
`createProduct`	Create new product with category
`getAllProduct`	List all products
`getProduct`	Get product by ID
`getProductByCategory`	Filter products by category
`updateProduct`	Update product fields
`deleteProduct`	Remove product

Example Implementation:

```
public Product createProduct(CreateProductDto dto, String filename) {
    Category category = categoryRepository.findById(dto.categoryId())
        .orElseThrow();

    Product product = Product.builder()
        .category(category)
        .name(dto.name())
        .description(dto.description())
        .price(dto.price())
        .stockQuantity(dto.stockQuantity())
        .imageUrl(filename)
        .brand(dto.brand())
        .modelYear(dto.modelYear())
        .engineCc(dto.engineCc())
        .color(dto.color())
        .conditionStatus(dto.conditionStatus())
        .build();
```

(continues on next page)

(continued from previous page)

```
    return productRepository.save(product);
}
```

14.7.5 CategoryService

Manages product categories.

Key Methods:

Method	Description
`createCategory`	Create new category
`getAllCategory`	List all categories
`getCategory`	Get category by ID
`updateCategory`	Update category
`deleteCategory`	Remove category

14.7.6 OrderService

Handles order processing and management.

Key Methods:

Method	Description
`createOrder`	Create new order with user
`getAllOrder`	List all orders
`getOrder`	Get order by ID
`getOrderByUser`	Get orders for specific user
`updateOrder`	Update order details
`deleteOrder`	Remove order

14.7.7 OrderItemService

Manages individual items within orders.

Key Methods:

Method	Description
`createOrderItem`	Add item to order
`getAllOrderItem`	List all order items
`getOrderItem`	Get item by ID
`getOrderItemByOrder`	Get items for specific order
`updateOrderItem`	Update item details
`deleteOrderItem`	Remove item from order

14.7.8 DTOs (Data Transfer Objects)

DTOs define the structure of request data.

CreateUserDto

```
public record CreateUserDto(
    String username,
    String email,
    String password,
    String phoneNumber,
    String address,
    String role
) {}
```

UpdateUserDto

Uses `Optional` for partial updates:

```
public record UpdateUserDto(
    Optional<String> username,
    Optional<String> email,
    Optional<String> password,
    Optional<String> phoneNumber,
    Optional<String> address,
    Optional<String> role,
    Optional<String> profileImageUrl
) {
    public UpdateUserDto withProfileImageUrl(String filename) {
        return new UpdateUserDto(
            username, email, password, phoneNumber,
            address, role, Optional.of(filename)
        );
    }
}
```

CreateProductDto

```
public record CreateProductDto(
    Long categoryId,
    String name,
    String description,
    BigDecimal price,
    Integer stockQuantity,
    String brand,
    Integer modelYear,
    Integer engineCc,
    String color,
    String conditionStatus,
    LocalDateTime createdAt
) {}
```

CreateOrderDto

```
public record CreateOrderDto(
    Long userId,
    BigDecimal totalAmount,
    String shippingAddress,
    String paymentMethod,
    List<CreateOrderItemDto> items
) {}
```

CreateOrderItemDto

```
public record CreateOrderItemDto(
    Long productId,
    Integer quantity,
    BigDecimal unitPrice
) {}
```

14.7.9 Repositories

Repositories extend Spring Data JPA's `CrudRepository`:

```
public interface UserRepository extends CrudRepository<User, Long> {
    User findByFullName(String fullName);
}

public interface ProductRepository extends CrudRepository<Product, Long> {
    List<Product> findByCategory_CategoryId(Long categoryId);
}

public interface OrderRepository extends CrudRepository<Order, Long> {
    List<Order> findByUser_UserId(Long userId);
}
```

Repository Methods

Spring Data JPA automatically implements:

- `findAll()` - Get all entities
- `findById(id)` - Get by primary key
- `save(entity)` - Create or update
- `deleteById(id)` - Delete by ID
- `count()` - Count all entities

Custom query methods:

- `findByFullName` - Find user by username
- `findByCategory_CategoryId` - Find products by category
- `findByUser_UserId` - Find orders by user

14.7.10 Password Encoding

Services use `BCryptPasswordEncoder` for password security:

```
@Autowired
private PasswordEncoder encoder;

// Hash password
String hashed = encoder.encode(plainPassword);

// Verify password
boolean matches = encoder.matches(plainPassword, hashedPassword);
```

BCrypt strength is configured in `SecurityConfig`:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12); // strength factor
}
```

14.7.11 Transaction Management

Spring manages transactions automatically. For complex operations, use `@Transactional`:

```
@Transactional
public Order createOrderWithItems(CreateOrderDto dto) {
    Order order = createOrder(dto);

    for (CreateOrderItemDto itemDto : dto.items()) {
        createOrderItem(order, itemDto);
    }

    return order;
}
```

14.7.12 Best Practices

1. **Keep controllers thin** - Move business logic to services
2. **Use DTOs** for request/response data
3. **Handle nulls** with Optional for updates
4. **Hash passwords** before storing
5. **Validate input** before processing
6. **Use specific exceptions** for error handling
7. **Keep services stateless** - No instance variables except dependencies

14.7.13 Next Steps

- *Backend Security* - Security configuration
- *Backend AI Integration* - AI service integration

14.8 Backend Security

Security configuration for the Spring Boot backend application.

14.8.1 Overview

The backend uses **Spring Security** for authentication and authorization with the following features:

- BCrypt password hashing
- Session-based authentication
- CSRF protection configuration
- CORS support for frontend access
- H2 console access for development

14.8.2 Security Configuration

The main security configuration is in `SecurityConfig.java`:

```
@Configuration
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(12);
    }

    @Bean
    UserDetailsService userDetailsService(PasswordEncoder encoder) {
        String password = encoder.encode("abc@123");
        UserDetails user = User.withUsername("user")
            .password(password)
            .roles("ADMIN")
            .build();
        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
            throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().permitAll()
            )
            .headers(headers -> headers
                .frameOptions(FrameOptionsConfig::disable)
            );
        return http.build();
    }
}
```

14.8.3 Password Encoding

BCrypt Password Hashing

Passwords are hashed using BCrypt with a strength factor of 12:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12);
}
```

BCrypt features:

- **Salt generation**: Automatic random salt
- **Adaptive hashing**: Strength factor controls iterations
- **One-way function**: Cannot be decrypted
- **Built-in salt**: Salt is stored with the hash

Usage in Services:

```
@Autowired
private PasswordEncoder encoder;

// Hash password before storing
String hashedPassword = encoder.encode(plainPassword);

// Verify password during login
boolean isValid = encoder.matches(plainPassword, hashedPassword);
```

14.8.4 CSRF Configuration

CSRF protection is disabled for API endpoints:

```
.csrf(csrf -> csrf.disable())
```

Note: In production, consider enabling CSRF with proper token handling for state-changing operations.

14.8.5 Authorization

Current Configuration

All requests are permitted (development mode):

```
.authorizeHttpRequests(authorize -> authorize
    .anyRequest().permitAll()
)
```

Production Configuration

For production, implement role-based access:

```
.authorizeHttpRequests(authorize -> authorize
    // Public endpoints
    .requestMatchers("/auth/**").permitAll()
```

(continues on next page)

(continued from previous page)

```
    .requestMatchers("/product").permitAll()
    .requestMatchers("/category").permitAll()

    // Admin only
    .requestMatchers("/user/**").hasRole("ADMIN")
    .requestMatchers(HttpMethod.POST, "/product").hasRole("ADMIN")
    .requestMatchers(HttpMethod.PUT, "/product/**").hasRole("ADMIN")
    .requestMatchers(HttpMethod.DELETE, "/product/**").hasRole("ADMIN")

    // Authenticated users
    .requestMatchers("/order/**").authenticated()
    .requestMatchers("/chat").authenticated()

    // Deny all others
    .anyRequest().denyAll()
)
```

User Roles

Default roles in the application:

- **customer**: Standard user role
- **admin**: Administrative privileges

Role is stored in the `User` entity:

```
@Builder.Default
private String role = "customer";
```

14.8.6 H2 Console Security

H2 console frame options are disabled for development:

```
.headers(headers -> headers
    .frameOptions(FrameOptionsConfig::disable)
)
```

This allows the H2 console to be displayed in frames.

14.8.7 Session Management

Session Configuration

Session cookie settings in `application.properties`:

```
server.servlet.session.cookie.same-site=lax
```

Options for `same-site`:

- `strict`: Cookie sent only to same site
- `lax`: Cookie sent on top-level navigation (default)
- `none`: Cookie sent with all requests (requires secure)

Session Timeout

Configure session timeout (in minutes):

```
server.servlet.session.timeout=30m
```

14.8.8 CORS Configuration

Cross-Origin Resource Sharing is configured in `WebConfig.java`:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
            .allowedHeaders("*")
            .allowCredentials(true)
            .maxAge(3600);
    }
}
```

CORS settings:

- **allowedOrigins**: Frontend URL (`*` allows all in development)
- **allowedMethods**: HTTP methods permitted
- **allowedHeaders**: Headers allowed in requests
- **allowCredentials**: Allow cookies/auth headers
- **maxAge**: Cache preflight response (seconds)

14.8.9 Security Best Practices

1. **Use HTTPS** in production
2. **Enable CSRF** with proper token handling
3. **Set secure session cookies**:

```
server.servlet.session.cookie.secure=true
server.servlet.session.cookie.http-only=true
```

4. **Implement rate limiting** for login endpoints
5. **Use strong BCrypt strength** (12-14)
6. **Validate all input** to prevent injection attacks
7. **Use parameterized queries** (JPA handles this automatically)
8. **Log security events** for monitoring

14.8.10 Production Security Checklist

Item	Status
HTTPS enabled	[ ]
CSRF enabled	[ ]
Session secure flag	[ ]
Session http-only flag	[ ]
Rate limiting implemented	[ ]
Input validation	[ ]
SQL injection prevention	[ ]
XSS prevention	[ ]
Security headers	[ ]
Logging enabled	[ ]

14.8.11 Authentication Flow

Login Process

1. **Client sends credentials**:

```
POST /auth/login
{
    "username": "john",
    "password": "secret123"
}
```

2. **Server validates**:

```
public ResponseEntity<?> loginUser(String username, String password) {
    User user = userRepository.findByFullName(username);

    if (user == null) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
            .body(Map.of("error", "Invalid username"));
    }

    if (!encoder.matches(password, user.getPasswordHash())) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
            .body(Map.of("error", "Invalid password"));
    }

    return ResponseEntity.ok(Map.of("userId", user.getUserId()));
}
```

3. **Session created**: Spring Security manages session
4. **Cookie returned**: Session ID in cookie

Registration Process

1. **Client sends user data**:

```
POST /user
Content-Type: multipart/form-data

user: {"username":"john","email":"john@example.com","password":"secret123",...}
profileImage: [file]
```

2. **Server hashes password**:

```
String hashedPassword = encoder.encode(dto.password());
```

3. **User saved** to database
4. **Response** with created user (excluding password)

14.8.12 Security Headers

Recommended security headers for production:

```
http.headers(headers -> headers
    .frameOptions(frameOptions -> frameOptions.deny())
    .xssProtection(xss -> xss.disable())
    .contentSecurityPolicy(csp -> csp.policyDirectives("default-src 'self'"))
    .httpStrictTransportSecurity(hsts -> hsts.includeSubDomains(true).
↪maxAgeInSeconds(31536000))
);
```

14.8.13 Next Steps

- *Backend AI Integration* - AI chat security
- *Backend Deployment* - Production deployment

14.9 Backend AI Integration

The Spring Boot backend integrates with Ollama for AI-powered chat functionality.

14.9.1 Overview

The application uses **Spring AI** to connect with **Ollama**, a local LLM (Large Language Model) server. This enables:

- Real-time AI chat responses
- Server-Sent Events (SSE) streaming
- Local LLM execution (no external API calls)
- Customizable AI models

14.9.2 Technology Stack

Spring AI

Spring AI provides abstractions for AI model integration:

- **ChatModel**: Interface for chat completions
- **Prompt**: Input to the model

- **Generation**: Model response
- **Streaming**: Reactive stream support

Ollama

Ollama runs LLMs locally:

- Open-source LLM runner
- Supports various models (Gemma, Llama, etc.)
- REST API for model interaction
- Local execution (privacy, no internet required)

14.9.3 Dependencies

Add to `build.gradle.kts`:

```
dependencies {
    implementation(platform("org.springframework.ai:spring-ai-bom:2.0.0-M2"))
    implementation("org.springframework.ai:spring-ai-starter-model-ollama")
}
```

14.9.4 Configuration

Application Properties

Configure Ollama in `application.properties`:

```
# Ollama server URL
spring.ai.ollama.base-url=http://localhost:11434

# Model pull strategy
spring.ai.ollama.init.pull-model-strategy=when_missing

# AI Model selection
spring.ai.ollama.chat.options.model=gemma3:270m
```

Model Options

Available models:

Model	Size	Use Case
gemma3:270m	270M params	Fast responses, simple queries
gemma3:1b	1B params	Balanced speed/quality
llama3.2:1b	1B params	Alternative model
llama3.2	3B+ params	Higher quality, slower

14.9.5 ChatController

The controller handles AI chat requests:

```java
@RestController
@RequestMapping("/chat")
public class ChatController {

    private final ChatModel chatModel;

    public ChatController(ChatModel chatModel) {
        this.chatModel = chatModel;
    }

    @PostMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<String> generate(@RequestBody Map<String, String> request) {
        String prompt = request.get("prompt");

        if (prompt == null || prompt.isBlank()) {
            return Flux.just("error: prompt is required");
        }

        return chatModel.stream(prompt)
            .concatWithValues("[DONE]");
    }
}
```

Key Components:

- **ChatModel**: Injected Spring AI component
- **@PostMapping**: Accepts POST requests
- **produces**: Sets content-type to `text/event-stream`
- **Flux<String>**: Reactive stream of responses
- **stream(prompt)**: Stream tokens as they're generated
- **concatWithValues**: Append end marker

14.9.6 API Usage

Request

```
POST /chat
Content-Type: application/json
```

```
{
    "prompt": "What are the benefits of electric motorcycles?"
}
```

Response

Server-Sent Events stream:

```
data: Electric motorcycles offer several benefits:

data:
```

(continues on next page)

(continued from previous page)

```
data: 1. **Environmental Impact**: Zero emissions during operation

data: 2. **Lower Operating Costs**: Electricity is cheaper than gasoline

data: 3. **Reduced Maintenance**: Fewer moving parts, no oil changes

data: [DONE]
```

Each `data:` line is a separate SSE event. The client should:

1. Open an EventSource connection
2. Listen for `message` events
3. Append each chunk to the UI
4. Stop when `[DONE]` received

14.9.7 Setup Ollama

Installation

macOS/Linux:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Windows:

Download from https://ollama.com/download

Start Ollama

```
ollama serve
```

Default port: 11434

Pull Models

Download the model specified in configuration:

```
ollama pull gemma3:270m
```

Verify installation:

```
ollama list
```

14.9.8 Customizing AI Behavior

System Prompt

Configure system behavior in application properties:

```
spring.ai.ollama.chat.options.system-prompt=You are a helpful motorcycle shop␣
↪assistant.
```

Temperature

Control response creativity (0.0 - 1.0):

```
spring.ai.ollama.chat.options.temperature=0.7
```

- **0.0**: Deterministic, focused responses
- **0.7**: Balanced creativity
- **1.0**: Maximum creativity

Max Tokens

Limit response length:

```
spring.ai.ollama.chat.options.max-tokens=500
```

14.9.9 Error Handling

Ollama Not Running

```
@PostMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> generate(@RequestBody Map<String, String> request) {
    try {
        String prompt = request.get("prompt");
        if (prompt == null || prompt.isBlank()) {
            return Flux.just("error: prompt is required");
        }
        return chatModel.stream(prompt)
            .concatWithValues("[DONE]");
    } catch (Exception e) {
        return Flux.just("error: AI service unavailable");
    }
}
```

Model Not Found

If the configured model isn't available:

1. Check model name in properties
2. Pull the model: `ollama pull gemma3:270m`
3. Or change to available model

14.9.10 Performance Considerations

Model Size

- Smaller models (270M) are faster but less capable
- Larger models (1B+) are slower but higher quality
- Choose based on response time requirements

Hardware Requirements

- **CPU**: Any modern CPU works

- **RAM**: 2GB+ for small models, 8GB+ for larger
- **GPU**: Optional but speeds up inference

Caching

Consider implementing response caching for common queries:

```
@Cacheable("chat-responses")
public Flux<String> generate(String prompt) {
    // Check cache first
    // Return cached or generate new
}
```

14.9.11 Testing

Manual Test

```
curl -X POST http://localhost:8080/chat \
  -H "Content-Type: application/json" \
  -d '{"prompt":"Hello"}'
```

Integration Test

```
@SpringBootTest
class ChatControllerTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void testChatEndpoint() {
        Map<String, String> request = Map.of("prompt", "Hi");
        ResponseEntity<String> response = restTemplate.postForEntity(
            "/chat", request, String.class
        );
        assertEquals(HttpStatus.OK, response.getStatusCode());
    }
}
```

14.9.12 Next Steps

- *Backend Deployment* - Deploy with Ollama
- Ollama Documentation: https://ollama.com/docs

14.10 Backend Deployment

Guide for deploying the Spring Boot backend to various environments.

14.10.1 Overview

The Spring Boot application can be deployed in multiple ways:

- **JAR file**: Standalone executable
- **Docker container**: Containerized deployment
- **Traditional server**: Tomcat, Jetty
- **Cloud platforms**: AWS, Azure, Heroku

14.10.2 Building for Production

Create Executable JAR

```
./gradlew bootJar
```

The JAR file is created at:

```
build/libs/demo-0.0.1-SNAPSHOT.jar
```

Run the JAR:

```
java -jar build/libs/demo-0.0.1-SNAPSHOT.jar
```

14.10.3 Production Configuration

Create `application-prod.properties`:

```
# Application
spring.application.name=demo
server.port=8080

# Database - MySQL
spring.datasource.url=jdbc:mysql://${DB_HOST:localhost}:3306/${DB_NAME:motorcycle_
↪shop}
spring.datasource.username=${DB_USERNAME:root}
spring.datasource.password=${DB_PASSWORD}
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# Security
server.servlet.session.cookie.same-site=strict
server.servlet.session.cookie.secure=true
server.servlet.session.cookie.http-only=true

# File Upload
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB

# AI/Ollama
spring.ai.ollama.base-url=${OLLAMA_URL:http://localhost:11434}
```

(continues on next page)

(continued from previous page)

```
spring.ai.ollama.chat.options.model=gemma3:270m

# Logging
logging.level.root=WARN
logging.level.com.example.demo=INFO
logging.file.name=/var/log/motorcycle-shop/app.log
```

Environment Variables

Set required environment variables:

```
export SPRING_PROFILES_ACTIVE=prod
export DB_HOST=your-db-host
export DB_NAME=motorcycle_shop
export DB_USERNAME=db_user
export DB_PASSWORD=your_secure_password
export OLLAMA_URL=http://localhost:11434
```

Run with production profile:

```
java -jar -Dspring.profiles.active=prod build/libs/demo-0.0.1-SNAPSHOT.jar
```

14.10.4 Docker Deployment

Dockerfile

Create `Dockerfile` in project root:

```
# Build stage
FROM eclipse-temurin:21-jdk-alpine AS builder

WORKDIR /app
COPY build.gradle.kts settings.gradle.kts ./
COPY gradle ./gradle
COPY gradlew ./
COPY src ./src

RUN ./gradlew bootJar --no-daemon

# Runtime stage
FROM eclipse-temurin:21-jre-alpine

WORKDIR /app

# Create uploads directory
RUN mkdir -p uploads

# Copy JAR from builder
COPY --from=builder /app/build/libs/*.jar app.jar

# Expose port
EXPOSE 8080
```

(continues on next page)

(continued from previous page)

```
# Run application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Build Docker Image

```
docker build -t motorcycle-shop-backend .
```

Run Container

```
docker run -d \
  --name motorcycle-backend \
  -p 8080:8080 \
  -e SPRING_PROFILES_ACTIVE=prod \
  -e DB_HOST=host.docker.internal \
  -e DB_PASSWORD=secret \
  -v $(pwd)/uploads:/app/uploads \
  motorcycle-shop-backend
```

Docker Compose

Create `docker-compose.yml`:

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - DB_HOST=db
      - DB_NAME=motorcycle_shop
      - DB_USERNAME=root
      - DB_PASSWORD=password
      - OLLAMA_URL=http://ollama:11434
    depends_on:
      - db
      - ollama
    volumes:
      - ./uploads:/app/uploads

  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: motorcycle_shop
    ports:
      - "3306:3306"
    volumes:
```

(continues on next page)

(continued from previous page)

```
      - mysql-data:/var/lib/mysql

  ollama:
    image: ollama/ollama
    ports:
      - "11434:11434"
    volumes:
      - ollama-data:/root/.ollama

volumes:
  mysql-data:
  ollama-data:
```

Run with Docker Compose:

```
docker-compose up -d
```

14.10.5 Cloud Deployment

AWS Elastic Beanstalk

1. Create `.ebextensions/java.config`:

```
option_settings:
  aws:elasticbeanstalk:container:java:
    JVMOptions: -Xmx512m -Dspring.profiles.active=prod
  aws:elasticbeanstalk:application:environment:
    DB_HOST: your-rds-endpoint
    DB_PASSWORD: your-password
```

2. Package application:

```
zip -r deploy.zip build/libs/*.jar .ebextensions/
```

3. Upload to Elastic Beanstalk

Heroku

Create `Procfile`:

```
web: java -Dserver.port=$PORT -jar build/libs/demo-0.0.1-SNAPSHOT.jar
```

Create `system.properties`:

```
java.runtime.version=21
```

Deploy:

```
git push heroku main
```

Render/Railway

Create `render.yaml`:

```
services:
  - type: web
    name: motorcycle-shop-backend
    runtime: java
    buildCommand: ./gradlew bootJar
    startCommand: java -jar build/libs/demo-0.0.1-SNAPSHOT.jar
    envVars:
      - key: SPRING_PROFILES_ACTIVE
        value: prod
      - key: DB_HOST
        fromDatabase:
          name: motorcycle-db
          property: host
```

14.10.6 Traditional Server Deployment

Systemd Service

Create `/etc/systemd/system/motorcycle-shop.service`:

```
[Unit]
Description=Motorcycle Shop Backend
After=network.target

[Service]
Type=simple
User=appuser
WorkingDirectory=/opt/motorcycle-shop
ExecStart=/usr/bin/java -jar -Dspring.profiles.active=prod app.jar
Restart=always
Environment="DB_PASSWORD=secret"
Environment="OLLAMA_URL=http://localhost:11434"

[Install]
WantedBy=multi-user.target
```

Enable and start:

```
sudo systemctl enable motorcycle-shop
sudo systemctl start motorcycle-shop
sudo systemctl status motorcycle-shop
```

Nginx Reverse Proxy

Configure Nginx:

```
server {
    listen 80;
    server_name api.yourdomain.com;
```

(continues on next page)

(continued from previous page)

```
    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    client_max_body_size 10M;
}
```

SSL with Let's Encrypt

```
sudo certbot --nginx -d api.yourdomain.com
```

14.10.7 Pre-Deployment Checklist

Item	Status
Application tests passing	[ ]
Production configuration created	[ ]
Environment variables configured	[ ]
Database migrations applied	[ ]
SSL certificate installed	[ ]
Log rotation configured	[ ]
Health check endpoint working	[ ]
Backup strategy in place	[ ]
Monitoring configured	[ ]

14.10.8 Monitoring and Health Checks

Health Endpoint

Spring Boot Actuator provides health checks:

Add dependency:

```
implementation("org.springframework.boot:spring-boot-starter-actuator")
```

Access health check:

```
curl http://localhost:8080/actuator/health
```

Response:

```
{
    "status": "UP"
}
```

Logging

Monitor logs:

```
tail -f /var/log/motorcycle-shop/app.log
```

Or with Docker:

```
docker logs -f motorcycle-backend
```

14.10.9 Troubleshooting

Application Won't Start

Check:

1. Database connection
2. Environment variables
3. Port availability
4. Java version

Out of Memory

Increase heap size:

```
java -Xmx1g -jar app.jar
```

Database Connection Issues

Verify:

```
# Test MySQL connection
mysql -h your-host -u user -p

# Check firewall rules
telnet your-host 3306
```

Ollama Not Responding

Check:

```
curl http://localhost:11434/api/tags

# Restart Ollama
ollama serve
```