Integração com APIs (Backend)

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

A autenticação em aplicações móveis é fundamentalmente uma dança entre cliente e servidor. O aplicativo Flutter que construímos é apenas metade da equação. A outra metade é o backend que valida credenciais, emite tokens, gerencia sessões e protege recursos. Nesta seção, vamos aprender como fazer nosso aplicativo Flutter comunicar-se efetivamente com APIs de backend, tratando todos os cenários possíveis desde requisições bem-sucedidas até falhas de rede e tokens expirados.

Vamos construir uma camada de integração com APIs que é robusta o suficiente para lidar com a realidade de redes móveis instáveis, flexível o suficiente para adaptar-se a diferentes backends, e clara o suficiente para que outros desenvolvedores entendam e mantenham. O objetivo é criar abstrações que isolem a complexidade de comunicação HTTP do resto da aplicação, permitindo que a lógica de negócio se concentre no que fazer com os dados, não em como obtê-los.

Escolhendo a Biblioteca HTTP

O Flutter oferece várias opções para fazer requisições HTTP. A biblioteca padrão http é simples e adequada para necessidades básicas, mas para aplicações de produção com autenticação, recomendo fortemente usar o pacote dio. O Dio oferece recursos essenciais que facilitam muito a vida do desenvolvedor, incluindo interceptors que permitem adicionar tokens automaticamente em todas as requisições, transformers que convertem automaticamente respostas JSON em objetos Dart, tratamento robusto de erros com tipos específicos de exceção, e suporte a cancelamento de requisições e timeouts configuráveis.

Vamos adicionar o Dio ao projeto:

```yaml
dependencies:
  flutter:
    sdk: flutter
  dio: ^5.4.0
```

A configuração inicial do Dio envolve criar uma instância com configurações base que serão aplicadas a todas as requisições:

```dart
```

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

class DioClient {
  late final Dio _dio;

  DioClient({String? baseUrl}) {
    _dio = Dio(
      BaseOptions(
        // URL base do seu backend
        baseUrl: baseUrl ?? 'https://api.exemplo.com',

        // Timeout padrão para conexão (10 segundos)
        connectTimeout: const Duration(seconds: 10),

        // Timeout padrão para receber resposta (30 segundos)
        receiveTimeout: const Duration(seconds: 30),

        // Headers que serão enviados em todas as requisições
        headers: {
          'Content-Type': 'application/json',
          'Accept': 'application/json',
        },

        // Validar status codes - considera apenas 2xx como sucesso
        validateStatus: (status) {
          return status != null && status >= 200 && status < 300;
        },
      ),
    );

    // Adiciona interceptors
    _dio.interceptors.add(_loggingInterceptor());
    _dio.interceptors.add(_authInterceptor());
  }

  // Getter para acessar a instância do Dio
  Dio get instance => _dio;
}
```

Esta configuração estabelece valores padrão sensatos para todas as requisições. O baseUrl evita repetir a URL completa em cada requisição. Os timeouts previnem que requisições fiquem penduradas indefinidamente se a rede estiver lenta ou o servidor não responder. Os headers padrão indicam que estamos enviando e esperando receber JSON, que é o formato universal para APIs REST modernas.

Estruturando o Auth Service

O Auth Service é a camada que faz as chamadas específicas de autenticação para o backend. Ele encapsula toda a lógica de comunicação HTTP, deixando o resto da aplicação trabalhar com objetos de domínio limpos sem se preocupar com detalhes de rede.

Vamos construir um AuthService completo que cobre todas as operações de autenticação:

dart

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

class AuthService {
  final Dio _dio;
  final SecureStorageService _storage;

  AuthService({
    required Dio dio,
    required SecureStorageService storage,
  })  : _dio = dio,
        _storage = storage;

  /// Faz login com email e senha
  /// Retorna o usuário autenticado ou lança exceção em caso de erro
  Future<User> login({
    required String email,
    required String password,
  }) async {
    try {
      // Faz requisição POST para o endpoint de login
      final response = await _dio.post(
        '/auth/login',
        data: {
          'email': email,
          'password': password,
        },
      );

      // Extrai dados da resposta
      final data = response.data as Map<String, dynamic>;

      // Salva tokens recebidos no armazenamento seguro
      await _storage.saveTokens(
        accessToken: data['access_token'] as String,
        refreshToken: data['refresh_token'] as String,
      );

      // Converte dados do usuário para objeto de domínio
      final userData = data['user'] as Map<String, dynamic>;
      return User.fromJson(userData);

    } on DioException catch (e) {
      // Trata diferentes tipos de erro HTTP
      throw _handleDioError(e);
    }
  }
```

```dart
/// Registra novo usuário
/// Retorna o usuário criado ou lança exceção
Future<User> register({
  required String email,
  required String password,
  String? name,
}) async {
  try {
    final response = await _dio.post(
      '/auth/register',
      data: {
        'email': email,
        'password': password,
        if (name != null) 'name': name,
      },
    );

    final data = response.data as Map<String, dynamic>;

    // Dependendo do backend, pode retornar tokens imediatamente
    // ou exigir confirmação de email primeiro
    if (data.containsKey('access_token')) {
      await _storage.saveTokens(
        accessToken: data['access_token'] as String,
        refreshToken: data['refresh_token'] as String,
      );
    }

    final userData = data['user'] as Map<String, dynamic>;
    return User.fromJson(userData);

  } on DioException catch (e) {
    throw _handleDioError(e);
  }
}

/// Solicita recuperação de senha
/// Envia email com link de redefinição
Future<void> resetPassword(String email) async {
  try {
    await _dio.post(
      '/auth/reset-password',
      data: {'email': email},
    );
    // Sucesso é silencioso - não retorna nada
  } on DioException catch (e) {
```

```
    throw _handleDioError(e);
  }
}

/// Atualiza senha usando token de recuperação
/// Token geralmente vem do link clicado no email
Future<void> updatePassword({
  required String token,
  required String newPassword,
}) async {
  try {
    await _dio.post(
      '/auth/update-password',
      data: {
        'token': token,
        'password': newPassword,
      },
    );
  } on DioException catch (e) {
    throw _handleDioError(e);
  }
}

/// Obtém dados do usuário atualmente autenticado
/// Usa o access token armazenado automaticamente pelo interceptor
Future<User?> getCurrentUser() async {
  try {
    final response = await _dio.get('/auth/me');
    final data = response.data as Map<String, dynamic>;
    return User.fromJson(data);
  } on DioException catch (e) {
    // Se falhar, provavelmente token é inválido
    return null;
  }
}

/// Faz logout revogando o refresh token no servidor
Future<void> logout() async {
  try {
    final refreshToken = await _storage.getRefreshToken();

    if (refreshToken != null) {
      await _dio.post(
        '/auth/logout',
        data: {'refresh_token': refreshToken},
      );
    }
```

```
    } catch (e) {
      // Mesmo se falhar no servidor, continua com logout local
      print('Erro ao revogar token no servidor: $e');
    } finally {
      // Sempre limpa tokens localmente
      await _storage.clearAll();
    }
  }

  /// Renova access token usando refresh token
  /// Chamado automaticamente pelo interceptor quando access token expira
  Future<Map<String, String>> refreshAccessToken() async {
    try {
      final refreshToken = await _storage.getRefreshToken();

      if (refreshToken == null) {
        throw Exception('Não há refresh token disponível');
      }

      final response = await _dio.post(
        '/auth/refresh',
        data: {'refresh_token': refreshToken},
      );

      final data = response.data as Map<String, dynamic>;

      return {
        'access_token': data['access_token'] as String,
        'refresh_token': data['refresh_token'] as String,
      };

    } on DioException catch (e) {
      throw _handleDioError(e);
    }
  }
}
```

Cada método segue um padrão consistente. Primeiro, tenta fazer a requisição HTTP apropriada. Se bem-sucedido, processa a resposta, salva tokens se necessário, converte dados JSON para objetos de domínio, e retorna o resultado. Se falhar, captura a DioException e a converte em uma exceção de domínio mais significativa usando o método handleDioError.

Note como usamos try-catch especificamente para DioException, que é o tipo de exceção que o Dio lança para erros relacionados a requisições HTTP. Outras exceções, como problemas de parsing de JSON ou erros de lógica, propagam naturalmente porque não as capturamos. Isto é intencional: queremos tratar erros HTTP de forma específica, mas deixar outros erros serem capturados em níveis superiores.

Tratamento de Erros HTTP

Diferentes códigos de status HTTP comunicam diferentes situações, e precisamos traduzi-los em ações apropriadas. Vamos implementar um tratador de erros abrangente que mapeia status codes para exceções de domínio significativas:

dart

```dart
/// Exceção customizada para erros de autenticação
class AuthException implements Exception {
  final String message;
  final String? code;
  final int? statusCode;

  AuthException({
    required this.message,
    this.code,
    this.statusCode,
  });

  @override
  String toString() => message;
}

/// Converte DioException em AuthException com mensagem apropriada
AuthException _handleDioError(DioException error) {
  switch (error.type) {
    case DioExceptionType.connectionTimeout:
    case DioExceptionType.sendTimeout:
    case DioExceptionType.receiveTimeout:
      return AuthException(
        message: 'Tempo de conexão esgotado. Verifique sua internet.',
        code: 'timeout',
      );

    case DioExceptionType.connectionError:
      return AuthException(
        message: 'Não foi possível conectar ao servidor. Verifique sua internet.',
        code: 'connection_error',
      );

    case DioExceptionType.badResponse:
      // Erro de resposta do servidor, analisa status code
      return _handleResponseError(error.response);

    case DioExceptionType.cancel:
      return AuthException(
        message: 'Requisição cancelada',
        code: 'cancelled',
      );

    default:
      return AuthException(
        message: 'Erro inesperado. Por favor, tente novamente.',
```

```
        code: 'unknown',
      );
    }
  }

  /// Trata erros baseados no status code da resposta
  AuthException _handleResponseError(Response? response) {
    if (response == null) {
      return AuthException(
        message: 'Erro ao processar resposta do servidor',
        code: 'no_response',
      );
    }

    final statusCode = response.statusCode ?? 0;
    final data = response.data;

    // Tenta extrair mensagem de erro do corpo da resposta
    String? errorMessage;
    if (data is Map<String, dynamic>) {
      errorMessage = data['message'] as String? ??
                     data['error'] as String?;
    }

    switch (statusCode) {
      case 400:
        return AuthException(
          message: errorMessage ?? 'Dados inválidos. Verifique as informações.',
          code: 'bad_request',
          statusCode: 400,
        );

      case 401:
        return AuthException(
          message: errorMessage ?? 'Email ou senha incorretos.',
          code: 'unauthorized',
          statusCode: 401,
        );

      case 403:
        return AuthException(
          message: errorMessage ?? 'Acesso negado.',
          code: 'forbidden',
          statusCode: 403,
        );

      case 404:
```

```
      return AuthException(
        message: errorMessage ?? 'Recurso não encontrado.',
        code: 'not_found',
        statusCode: 404,
      );

    case 409:
      return AuthException(
        message: errorMessage ?? 'Este email já está cadastrado.',
        code: 'conflict',
        statusCode: 409,
      );

    case 422:
      return AuthException(
        message: errorMessage ?? 'Dados inválidos.',
        code: 'unprocessable_entity',
        statusCode: 422,
      );

    case 429:
      return AuthException(
        message: errorMessage ?? 'Muitas tentativas. Aguarde alguns minutos.',
        code: 'too_many_requests',
        statusCode: 429,
      );

    case 500:
    case 502:
    case 503:
      return AuthException(
        message: 'Erro no servidor. Tente novamente mais tarde.',
        code: 'server_error',
        statusCode: statusCode,
      );

    default:
      return AuthException(
        message: errorMessage ?? 'Erro ao processar requisição.',
        code: 'unknown_error',
        statusCode: statusCode,
      );
  }
}
```

Este tratamento de erros é abrangente e amigável ao usuário. Primeiro, diferenciamos entre tipos de erro do Dio como timeout, erro de conexão, ou erro de resposta do servidor. Para erros de resposta, analisamos o status code e retornamos mensagens específicas para cada cenário. Também tentamos extrair mensagens de erro do corpo da resposta quando o servidor as fornece, dando prioridade a mensagens específicas do servidor sobre mensagens genéricas do cliente.

O resultado é que quando algo dá errado, o usuário recebe uma mensagem clara e acionável como "Email ou senha incorretos" ou "Muitas tentativas. Aguarde alguns minutos", ao invés de mensagens técnicas confusas como "HTTP 401" ou "DioException: Bad response".

Interceptors: Automatizando Tarefas Comuns

Interceptors são uma das funcionalidades mais poderosas do Dio. Eles permitem interceptar requisições antes de serem enviadas e respostas antes de serem processadas, permitindo adicionar comportamentos globais sem duplicar código em cada chamada de API.

Vamos implementar dois interceptors essenciais: um para logging que ajuda durante desenvolvimento, e um para autenticação que adiciona tokens automaticamente e os renova quando expiram.

Logging Interceptor

Este interceptor registra todas as requisições e respostas, facilitando debugging durante desenvolvimento:

dart

```dart
Interceptor _loggingInterceptor() {
  return InterceptorsWrapper(
    onRequest: (options, handler) {
      print('🔵 REQUEST[${options.method}] => PATH: ${options.path}');
      print('Headers: ${options.headers}');
      print('Data: ${options.data}');
      return handler.next(options);
    },
    onResponse: (response, handler) {
      print('🟢 RESPONSE[${response.statusCode}] => PATH: ${response.requestOptions.path}');
      print('Data: ${response.data}');
      return handler.next(response);
    },
    onError: (error, handler) {
      print('🔴 ERROR[${error.response?.statusCode}] => PATH: ${error.requestOptions.path}');
      print('Message: ${error.message}');
      return handler.next(error);
    },
  );
}
```

Este interceptor é simples mas extremamente útil. Durante desenvolvimento, você pode ver exatamente quais requisições estão sendo feitas, com quais dados, e quais respostas estão retornando. Os emojis coloridos facilitam encontrar logs específicos rapidamente no console. Em produção, você removeria este interceptor ou o substituiria por um que envia logs para um serviço de monitoramento ao invés de imprimi-los no console.

Auth Interceptor

Este é o interceptor mais importante para autenticação. Ele adiciona automaticamente o access token em todas as requisições, detecta quando o token expira, e tenta renová-lo automaticamente:

dart

```
Interceptor _authInterceptor() {
  return InterceptorsWrapper(
    onRequest: (options, handler) async {
      // Obtém access token do armazenamento seguro
      final accessToken = await _storage.getAccessToken();

      // Se há token e a requisição não é para renovação
      // (evita loop infinito)
      if (accessToken != null &&
          !options.path.contains('/auth/refresh')) {
        // Adiciona token no header Authorization
        options.headers['Authorization'] = 'Bearer $accessToken';
      }

      return handler.next(options);
    },

    onError: (error, handler) async {
      // Se erro é 401 (não autorizado), tenta renovar token
      if (error.response?.statusCode == 401) {
        try {
          // Obtém refresh token
          final refreshToken = await _storage.getRefreshToken();

          if (refreshToken == null) {
            // Não há refresh token, usuário precisa fazer login
            return handler.reject(error);
          }

          // Tenta renovar access token
          final response = await _dio.post(
            '/auth/refresh',
            data: {'refresh_token': refreshToken},
            options: Options(
              headers: {
                'Authorization': 'Bearer $refreshToken',
              },
            ),
          );

          // Extrai novos tokens da resposta
          final data = response.data as Map<String, dynamic>;
          final newAccessToken = data['access_token'] as String;
          final newRefreshToken = data['refresh_token'] as String;

          // Salva novos tokens
```

```
        await _storage.saveTokens(
          accessToken: newAccessToken,
          refreshToken: newRefreshToken,
        );

        // Atualiza token na requisição original
        error.requestOptions.headers['Authorization'] =
          'Bearer $newAccessToken';

        // Tenta novamente a requisição original com novo token
        final clonedRequest = await _dio.fetch(error.requestOptions);

        // Retorna resposta da requisição que deu certo
        return handler.resolve(clonedRequest);

      } catch (refreshError) {
        // Renovação falhou, usuário precisa fazer login
        await _storage.clearAll();
        return handler.reject(error);
      }
    }

    // Para outros erros, apenas propaga
    return handler.next(error);
  },
 );
}
```

Este interceptor implementa renovação automática de tokens de forma transparente. Quando uma requisição retorna 401 indicando que o access token expirou, o interceptor automaticamente usa o refresh token para obter um novo access token, salva o novo token, e tenta novamente a requisição original. Do ponto de vista do código que fez a chamada inicial, a requisição simplesmente funciona, sem necessidade de código especial para lidar com tokens expirados.

A chave para este padrão funcionar corretamente é evitar loop infinito. Se a própria requisição de refresh falhar com 401, não tentamos renová-la novamente, apenas propagamos o erro. Também evitamos adicionar token na requisição de refresh em si, pois essa requisição usa o refresh token de forma diferente.

Parsing de JSON e Models

Respostas de APIs REST geralmente vêm em formato JSON. Precisamos converter este JSON em objetos Dart tipados que nosso código pode usar. Esta conversão acontece em classes de modelo com métodos fromJson e toJson:

dart

```dart
class UserModel {
  final String id;
  final String email;
  final String? name;
  final String? photoUrl;
  final DateTime createdAt;

  UserModel({
    required this.id,
    required this.email,
    this.name,
    this.photoUrl,
    required this.createdAt,
  });

  /// Cria instância a partir de JSON
  factory UserModel.fromJson(Map<String, dynamic> json) {
    return UserModel(
      id: json['id'] as String,
      email: json['email'] as String,
      name: json['name'] as String?,
      photoUrl: json['photo_url'] as String?,
      createdAt: DateTime.parse(json['created_at'] as String),
    );
  }

  /// Converte instância para JSON
  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'email': email,
      if (name != null) 'name': name,
      if (photoUrl != null) 'photo_url': photoUrl,
      'created_at': createdAt.toIso8601String(),
    };
  }

  /// Converte model para entidade de domínio
  User toDomain() {
    return User(
      id: id,
      email: email,
      name: name,
      photoUrl: photoUrl,
    );
```

```
  }
}
```

O método fromJson é um factory constructor que recebe um mapa String para dynamic, que é o tipo que json.decode retorna. Extraímos cada campo do mapa, fazendo cast para o tipo apropriado. Para campos opcionais, usamos cast para tipo nullable. Para campos que precisam de conversão como datas, aplicamos a conversão necessária.

O método toJson faz o oposto, convertendo a instância de volta para um mapa que pode ser serializado para JSON. Usamos sintaxe de spread condicional para incluir campos opcionais apenas se não forem null, mantendo o JSON limpo.

O método toDomain converte o model em uma entidade de domínio pura. Esta separação entre model e entidade é importante em arquiteturas limpas. O model representa como os dados existem na API, a entidade representa como os dados existem no domínio da aplicação. Se a estrutura da API mudar, só precisamos atualizar o model e a conversão, não toda a aplicação.

Requisições Autenticadas

Depois que o usuário faz login e temos tokens salvos, todas as requisições para endpoints protegidos precisam incluir o access token. Graças ao nosso auth interceptor, isto acontece automaticamente. Vamos ver exemplos de como fazer requisições autenticadas para outros endpoints:

dart

```dart
class UserRepository {
  final Dio _dio;

  UserRepository(this._dio);

  /// Obtém perfil do usuário
  /// Token é adicionado automaticamente pelo interceptor
  Future<User> getProfile() async {
    try {
      final response = await _dio.get('/user/profile');
      final data = response.data as Map<String, dynamic>;
      return UserModel.fromJson(data).toDomain();
    } on DioException catch (e) {
      throw _handleDioError(e);
    }
  }

  /// Atualiza perfil do usuário
  Future<User> updateProfile({
    String? name,
    String? photoUrl,
  }) async {
    try {
      final response = await _dio.put(
        '/user/profile',
        data: {
          if (name != null) 'name': name,
          if (photoUrl != null) 'photo_url': photoUrl,
        },
      );

      final data = response.data as Map<String, dynamic>;
      return UserModel.fromJson(data).toDomain();
    } on DioException catch (e) {
      throw _handleDioError(e);
    }
  }

  /// Deleta conta do usuário
  Future<void> deleteAccount() async {
    try {
      await _dio.delete('/user/account');
    } on DioException catch (e) {
      throw _handleDioError(e);
    }
```

```
  }
}
```

Note como não há código especial para adicionar tokens. Simplesmente fazemos as requisições normalmente e o interceptor cuida de tudo. Se o token expirar durante qualquer uma dessas chamadas, o interceptor renova automaticamente e a requisição funciona. Este é o poder de interceptors: comportamentos transversais que aplicam a todas as requisições sem código duplicado.

Conclusão da Seção

Nesta seção, construímos uma camada completa de integração com APIs para autenticação. Aprendemos a usar o Dio com configurações apropriadas, estruturar um AuthService que encapsula todas as chamadas de autenticação, tratar erros HTTP de forma abrangente convertendo-os em exceções de domínio significativas, e implementar interceptors que automatizam tarefas como adicionar tokens e renová-los quando expiram.

O resultado é uma camada de rede robusta que isola o resto da aplicação de complexidades de comunicação HTTP. O código de negócio simplesmente chama métodos do AuthService e recebe objetos de domínio ou exceções compreensíveis, sem precisar saber nada sobre requisições HTTP, parsing de JSON ou renovação de tokens. Esta separação de responsabilidades mantém o código limpo, testável e manutenível.

Na próxima seção, vamos focar em navegação e proteção de rotas, aprendendo como controlar o acesso a diferentes partes do aplicativo baseado no estado de autenticação, e como implementar deep linking para suportar redirecionamentos de emails de confirmação e recuperação de senha.

Fim da Seção 8: Integração com APIs (Backend)