Documentação

1. Visão Geral do Projeto

O **LivrariaAppV2** é um aplicativo móvel desenvolvido para a plataforma Android, construído integralmente utilizando a linguagem de programação **Kotlin** e o framework de UI **Jetpack Compose**. Este aplicativo simula um sistema de livraria robusto, oferecendo funcionalidades tanto para usuários comuns quanto para administradores.

1. Visão Geral do Projeto

O **LivrariaAppV2** é um aplicativo móvel desenvolvido para a plataforma Android, construído integralmente utilizando a linguagem de programação **Kotlin** e o framework de UI **Jetpack Compose**. Este aplicativo simula um sistema de livraria robusto, oferecendo funcionalidades tanto para usuários comuns quanto para administradores.

Objetivos Principais:

- **Catálogo de Livros:** Exibir uma vasta coleção de livros de forma organizada.
- **Pesquisa Dinâmica:** Permitir que os usuários encontrem livros rapidamente por título ou autor.
- **Gestão de Favoritos:** Capacitar os usuários a marcar e gerenciar seus livros preferidos, com persistência por usuário.
- **Autenticação e Autorização:** Implementar um sistema de login e registro, distinguindo papéis de usuário (usuário comum vs. administrador) para controle de acesso a funcionalidades específicas.
- **Gerenciamento Administrativo:** Fornecer aos administradores a capacidade de adicionar novos livros, editar detalhes de livros existentes e gerenciar o status de ativação/desativação de livros.
- **Persistência de Dados:** Armazenar informações de usuários e livros localmente no dispositivo.

Tecnologias-Chave:

- **Jetpack Compose:** Para a construção da interface do usuário declarativa.

- **Room Persistence Library:** Para o gerenciamento de banco de dados SQLite local.
- **Kotlin Coroutines & Flow:** Para programação assíncrona reativa e gerenciamento de streams de dados.
- **Jetpack Navigation Compose:** Para a navegação entre as telas do aplicativo.
- **Jetpack ViewModel & LiveData/StateFlow:** Para o gerenciamento do estado da UI de forma otimizada para o ciclo de vida.
- **DataStore Preferences:** Para armazenamento persistente e assíncrono de pares chave-valor, utilizado na gestão de favoritos.
- **BCrypt (JBCrypt):** Para hashing seguro de senhas.
- **Coil:** Para carregamento e exibição eficiente de imagens de rede.

2. Arquitetura do Aplicativo (MVVM)

O LivrariaAppV2 adota a arquitetura **MVVM (Model-View-ViewModel)**, que promove a separação de preocupações e a testabilidade do código.

- **Model (Camada de Dados):**
  - Representa os dados e a lógica de negócios.
  - No projeto, inclui as classes de entidades (`Book`, `User`), os DAOs (`BookDao`, `UserDao`), o banco de dados Room (`AppDatabase`), o `FavoriteManager` e as classes de repositório (`AuthRepository`, `BookRepository`).
  - É responsável por buscar, armazenar e manipular os dados, isolando a lógica de acesso a dados do resto do aplicativo.
- **View (Camada de UI - Jetpack Compose):**
  - Representa a interface do usuário.
  - No projeto, são as Composable Functions nas telas (`LoginScreen`, `HomeScreen`, `BookDetailsScreen`, etc.) e componentes de UI (`BookCard`, `SearchBar`).
  - É responsável por observar os dados expostos pelo ViewModel e exibir o estado atual da UI. Ela não contém lógica de negócios ou acesso direto a dados.
- **ViewModel (Camada de Lógica da UI):**

- Atua como um intermediário entre a View e o Model.
- No projeto, são as classes `AuthViewModel` e `BookViewModel`.
- Mantém o estado da UI e a lógica de apresentação, expondo dados e ações para a View através de `StateFlow`s. Ele não possui referência direta à View, garantindo que não haja vazamentos de memória e facilitando a testabilidade.
- Inicia operações assíncronas (via Coroutines) para interagir com os Repositórios e atualizar o estado da UI.

Essa separação clara entre as camadas melhora a manutenibilidade, escalabilidade e testabilidade do aplicativo.

3. Configurações e Inicialização Global

3.1. Classe `Application` (`LivrariaAppV2Application.kt` - Pag 24)

Esta classe estende `android.app.Application` e serve como o ponto de entrada global do aplicativo.

- `onCreate()`:
  - É o primeiro método a ser chamado quando o processo do aplicativo é criado.
  - **Configuração do Coil:** O bloco `Coil.setImageLoader(imageLoader)` inicializa a biblioteca Coil para carregamento de imagens. O `DebugLogger()` é habilitado, o que é extremamente útil durante o desenvolvimento para diagnosticar problemas de carregamento de imagens, exibindo informações detalhadas no Logcat sobre o ciclo de vida e o desempenho do carregamento de cada imagem. Em um ambiente de produção, este logger seria geralmente desabilitado para otimização.

3.2. Atividade Principal (`MainActivity.kt` - Pag 25)

A `MainActivity` é a única atividade no aplicativo e é responsável por configurar o ambiente inicial, instanciar as dependências da camada de dados/repositório e configurar o sistema de navegação da interface do usuário.

- `onCreate(savedInstanceState: Bundle?)`:
  - **Criação de Dependências de Camada de Dados:**

- `AppDatabase.getDatabase(applicationContext)`: Obtém a instância singleton do banco de dados Room. O `applicationContext` é usado para evitar vazamentos de memória associados ao contexto da Atividade.
- `userDao` e `bookDao`: Obtêm as interfaces DAO para interagir com as tabelas de usuários e livros, respectivamente.
- `AuthRepository(userDao)`: Instancia o repositório de autenticação, injetando o `UserDao`.
- `BookRepository(bookDao)`: Instancia o repositório de livros, injetando o `BookDao`.
- `FavoriteManager(applicationContext)`: Instancia o gerenciador de favoritos, passando o `applicationContext` para acesso ao DataStore.

- **Criação de `ViewModelFactory`s:**
  - `AuthViewModelFactory(authRepository)`: Cria um factory para `AuthViewModel`, injetando `AuthRepository`.
  - `BookViewModelFactory(bookRepository, favoriteManager, authViewModel.currentUser)`: Cria um factory para `BookViewModel`. É crucial notar que `authViewModel.currentUser` (um `StateFlow<User?>`) é passado *como uma dependência*. Isso permite que `BookViewModel` observe o estado de autenticação do usuário em tempo real, sem criar uma dependência circular direta entre os ViewModels.
- `setContent`: Bloco principal do Jetpack Compose que define a hierarquia da UI.
  - `LivrariaAppV2Theme`: Aplica o tema visual definido para o aplicativo.
  - `Surface`: Um contêiner que define a superfície primária da UI, usando a cor de fundo do tema.
  - **Instanciação dos ViewModels:**
    - `val authViewModel: AuthViewModel = viewModel(factory = authViewModelFactory)`: Instancia o `AuthViewModel` usando o factory pré-definido.
    - `val bookViewModel: BookViewModel = viewModel(factory = bookViewModelFactory)`: Instancia o `BookViewModel`, aproveitando o factory que já recebeu o `currentUser` do `authViewModel`. O uso de `viewModel()` dentro de um `@Composable` garante que o ViewModel sobreviva a mudanças de configuração e seja associado ao ciclo de vida correto.

        - `rememberNavController()`: Cria e lembra uma instância de `NavController` para gerenciar a navegação.
        - `AppNavHost`: O componente central de navegação, que recebe o `navController` e as instâncias de `authViewModel` e `bookViewModel` para que as telas dentro do grafo de navegação possam acessá-los.

4. Configuração do Projeto Gradle (`build.gradle.kts (app)` - Pag 26)

Este arquivo define as configurações de build e as dependências do módulo do aplicativo.

- `plugins` **bloco**:
    - `alias(libs.plugins.android.application)`: Plugin para aplicações Android.
    - `alias(libs.plugins.kotlin.android)`: Plugin para o suporte a Kotlin.
    - `id("kotlin-kapt")`: Plugin essencial para o processamento de anotações do Kotlin (utilizado pelo Room para gerar código).
    - `alias(libs.plugins.kotlin.compose)`: Plugin para habilitar o Jetpack Compose.
- `android` **bloco**:
    - `namespace`: Define o namespace do pacote Java/Kotlin.
    - `compileSdk`: Versão do SDK Android que o projeto usa para compilar (API 36).
    - `defaultConfig`:
        - `applicationId`: ID único do aplicativo.
        - `minSdk`: Versão mínima do SDK Android suportada (API 24, Android 7.0 Nougat).
        - `targetSdk`: Versão do SDK Android para a qual o aplicativo é otimizado (API 36).
        - `versionCode` e `versionName`: Versão interna e externa do aplicativo.
        - `testInstrumentationRunner`: Runner de teste para instrumentação.
        - `vectorDrawables { useSupportLibrary = true }`: Habilita o uso de `VectorDrawable`s em versões mais antigas do Android.
    - `buildTypes`: Configura diferentes tipos de build (e.g., `release`, `debug`).

        - `release`: Define configurações para o build de lançamento (minificação desabilitada, uso de `proguardFiles`).
    - `compileOptions`: Define a versão do Java para a compilação (Java 11).
    - `kotlinOptions`: Define a versão da JVM target para o Kotlin (JVM 11).
    - `buildFeatures { compose = true }`: Sinaliza ao Gradle que este módulo usa o Jetpack Compose.
    - `composeOptions { kotlinCompilerExtensionVersion = "1.6.10" }`: Especifica a versão do compilador do Compose, garantindo compatibilidade entre o Compose e o Kotlin.
    - `packaging`: Configura como os recursos são empacotados, excluindo arquivos específicos para evitar conflitos de licença ou metadados desnecessários.
- `dependencies` **bloco**: Lista de todas as bibliotecas de terceiros e módulos internos.
    - **Dependências do Compose**: Essenciais para o funcionamento do Jetpack Compose (e.g., `ui`, `graphics`, `material3`, `activity-compose`). `platform(libs.androidx.compose.bom)` garante que todas as bibliotecas Compose usem versões compatíveis entre si.
    - **Room (SQLite)**:
        - `implementation(libs.androidx.room.runtime)`: Biblioteca principal do Room.
        - `kapt(libs.androidx.room.compiler)`: Processador de anotações que gera o código necessário para o Room.
        - `implementation(libs.androidx.room.ktx)`: Extensões Kotlin para Room, incluindo suporte a Coroutines e Flow.
    - **Coroutines**:
        - `implementation(libs.kotlinx.coroutines.core)`: Coroutines para lógica agnóstica de plataforma.
        - `implementation(libs.kotlinx.coroutines.android)`: Extensões de Coroutines para Android, incluindo dispatchers de UI.
    - **ViewModel e LiveData/StateFlow**:
        - `implementation(libs.androidx.lifecycle.viewmodel.ktx)`: Extensões Kotlin para ViewModel.

        - `implementation(libs.androidx.lifecycle.viewmodel.compose)`: Extensões específicas para usar ViewModels em Composable functions (`viewModel()`).
    - **Navigation Compose**:
        - `implementation(libs.androidx.navigation.compose)`: Biblioteca para gerenciar a navegação em Jetpack Compose.
    - **Material Icons Extended**:
        - `implementation(libs.androidx.compose.material.icons.extended)`: Fornece um conjunto maior de ícones do Material Design.
    - **Hashing de senha (BCrypt)**:
        - `implementation(libs.jbcrypt)` e `implementation("org.mindrot:jbcrypt:0.4")`: Bibliotecas para hashing de senhas usando o algoritmo BCrypt, essencial para segurança ao armazenar credenciais de usuário.
    - **Carregamento de imagens (Coil)**:
        - `implementation(libs.coil.compose)`: Biblioteca leve e rápida para carregamento de imagens assíncrono em Jetpack Compose.
    - **DataStore (para FavoriteManager)**:
        - `implementation(libs.androidx.datastore.preferences)`: Biblioteca para armazenamento persistente de dados pequenos e tipados de forma assíncrona.
    - **Dependências de teste**: Bibliotecas para realizar testes unitários (JUnit) e de instrumentação (AndroidX Test, Compose Test).

5. Camada de Dados (Model)

Esta camada é responsável por encapsular a lógica de acesso a dados e as regras de negócio relacionadas aos dados.

5.1. Modelos de Dados (`data.model`)

Definem as estruturas dos objetos que representam os dados do aplicativo.

- **`Book.kt`**:Kotlin

```
@Entity(tableName = "books")
data class Book(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val title: String,
    val author: String,
```

```
    val description: String,
    val price: Double,
    val quantity: Int,
    val imageUrl: String = "",
    val isActive: Boolean = true // Indica se o livro está ativo/disponível
)
```

  - `@Entity(tableName = "books")`: Anotação Room que mapeia esta classe para uma tabela SQLite chamada "books".
  - `@PrimaryKey(autoGenerate = true)`: Define `id` como a chave primária auto-gerada.
  - `isActive: Boolean`: Campo booleano crucial que permite aos administradores "desativar" um livro sem removê-lo completamente. Livros inativos não são visíveis para usuários comuns no catálogo.

- `User.kt`:Kotlin

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val username: String,
    val passwordHash: String, // Senha armazenada como hash
    val role: String // Ex: "user", "admin"
)
```

  - `@Entity(tableName = "users")`: Mapeia para a tabela SQLite "users".
  - `passwordHash: String`: Armazena a senha do usuário já criptografada com BCrypt, nunca a senha em texto claro, para segurança.
  - `role: String`: Define o papel do usuário no sistema, controlando o acesso a funcionalidades administrativas.

5.2. Banco de Dados Room (`AppDatabase.kt` , DAOs)

Room é a camada de abstração do SQLite que facilita o trabalho com o banco de dados local.

- `AppDatabase.kt`:Kotlin

```
@Database(entities = [User::class, Book::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun bookDao(): BookDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
```

```
            val instance = Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "livraria_database"
            )
            .fallbackToDestructiveMigration() // Para facilitar desenvolvimento, em produção usar migrations
            .build()
            INSTANCE = instance
            instance
        }
    }
}
```

- `@Database(...)`: Anotação que define as entidades pertencentes a este banco de dados (`User`, `Book`), a `version` (para controle de migrações) e `exportSchema`.
- **Métodos Abstratos para DAOs**: `userDao()` e `bookDao()` fornecem acesso às interfaces DAO.
- **Padrão Singleton (`companion object`)**: O método `getDatabase()` implementa o padrão Singleton para garantir que apenas uma instância do banco de dados seja criada, otimizando recursos e evitando inconsistências.
  - `@Volatile INSTANCE`: Garante que a variável `INSTANCE` seja sempre lida da memória principal e não de um cache de thread.
  - `synchronized(this)`: Bloco para garantir que a criação do banco de dados seja thread-safe.
  - `fallbackToDestructiveMigration()`: Usado para desenvolvimento, permite que o Room recrie o banco de dados se a versão for alterada. Em produção, isso levaria à perda de dados; migrações incrementais seriam preferidas.

- `BookDao.kt`:Kotlin

```
@Dao
interface BookDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertBook(book: Book)@Update
    suspend fun updateBook(book: Book)@Query("SELECT * FROM books ORDER BY title ASC")
    fun getAllBooks(): Flow<List<Book>> // Observa mudanças na lista de livros

    @Query("SELECT * FROM books WHERE id = :bookId")
    fun getBookById(bookId: Int): Flow<Book?> // Observa um livro específico

    @Query("UPDATE books SET isActive = 0 WHERE id = :bookId")
    suspend fun deactivateBook(bookId: Int)@Query("UPDATE books SET isActive = 1 WHERE id = :bookId")
    suspend fun activateBook(bookId: Int)
}
```

- `@Dao`: Anotação que marca a interface como um Data Access Object.
- `@Insert`, `@Update`: Anotações Room para operações básicas de banco de dados. `onConflict = OnConflictStrategy.REPLACE` indica que se um livro com o mesmo ID for inserido, o existente será substituído.
- `@Query(...)`: Permite definir consultas SQL personalizadas.
  - `getAllBooks()` e `getBookById()`: Retornam `Flow<T>`, o que significa que eles emitem novos valores sempre que os dados subjacentes no banco de dados mudam, permitindo uma UI reativa.
- `suspend`: Indica que as funções são suspendíveis e podem ser chamadas de uma coroutine.

- `UserDao.kt`:Kotlin

```
@Dao
interface UserDao {
  @Insert
  suspend fun insertUser(user: User)@Query("SELECT * FROM users WHERE username = :username LIMIT 1")
  suspend fun getUserByUsername(username: String): User?
}
```

  - `@Insert`: Para inserir novos usuários.
  - `@Query(...)`:
    - `getUserByUsername()`: Busca um usuário pelo nome de usuário. Retorna `User?` (nulo se não encontrado) e é `suspend` porque é uma operação de banco de dados que pode demorar.

5.3. Gerenciador de Favoritos (`FavoriteManager.kt` - Pag 11)

Este componente gerencia a lista de livros favoritos de cada usuário de forma persistente e assíncrona, utilizando **DataStore Preferences**.

- **Propriedades**:
  - `Context.dataStore`: Uma propriedade de extensão que cria uma instância de `DataStore<Preferences>` para armazenar pares chave-valor. O nome "livraria_preferences" é o nome do arquivo subjacente.
- `addFavoriteBook(userId: String, bookId: Int)`:
  - Obtém a chave DataStore específica para o ID do usuário (`FAVORITES_KEY_PREFIX + userId`).
  - Usa `dataStore.edit { preferences -> ... }` para realizar uma transação atômica.

    - Recupera o conjunto atual de IDs de favoritos (armazenados como `Set<String>`).
    - Adiciona o `bookId` (convertido para `String`) ao conjunto.
    - Salva o conjunto atualizado de volta nas preferências.
- **`removeFavoriteBook(userId: String, bookId: Int)`**:
    - Similar ao `addFavoriteBook`, mas remove o `bookId` do conjunto existente.
- **`getFavoriteBookIds(userId: String)`**:
    - Obtém a chave DataStore para o usuário.
    - Retorna um `Flow<Set<Int>>` observando as preferências. O `map` transforma o `Set<String>` (armazenado no DataStore) em `Set<Int>` (usado na lógica do aplicativo), e trata o caso de não haver favoritos (retorna um conjunto vazio).

6. Camada de Repositórios

Os repositórios atuam como uma fonte única de verdade para os dados, abstraindo a origem dos dados (banco de dados, rede, etc.) dos ViewModels.

6.1. Repositório de Autenticação (`AuthRepository.kt` - Pag 9)

Gerencia as operações de autenticação e registro de usuários.

- **Construtor**: Recebe uma instância de `UserDao` como dependência.
- **`loginUser(username: String, passwordPlain: String)`**:
    - Busca o `User` pelo `username` no `UserDao`.
    - Se o usuário for encontrado, utiliza `BCrypt.checkpw()` para comparar a `passwordPlain` (senha em texto claro fornecida pelo usuário) com o `passwordHash` (hash armazenado no banco de dados). Esta é uma operação criptográfica segura.
    - Retorna o `User` se a senha for válida, caso contrário, retorna `null`.
- **`registerUser(username: String, passwordPlain: String)`**:
    - Primeiro, verifica se já existe um usuário com o `username` fornecido para evitar duplicidade.
    - Se não existir, gera um hash seguro da `passwordPlain` usando `BCrypt.hashpw()` com um "sal" aleatório (`BCrypt.gensalt()`). O sal é crucial para a segurança,

pois impede ataques de "rainbow table".

- Cria um novo objeto `User` com o nome de usuário, o hash da senha e define o `role` como "user" (padrão para novos registros).
- Insere o novo usuário no `UserDao`.
- Retorna `true` se o registro for bem-sucedido, `false` caso contrário (usuário já existe).

6.2. Repositório de Livros (`BookRepository.kt` - Pag 8)

Gerencia as operações relacionadas aos dados dos livros.

- **Construtor**: Recebe uma instância de `BookDao`.
- **`getAllBooks()`**: Delega a chamada para `bookDao.getAllBooks()`, retornando um `Flow<List<Book>>`.
- **`getBookById(bookId: Int)`**: Delega a chamada para `bookDao.getBookById(bookId)`, retornando um `Flow<Book?>`.
- **`insertBook(book: Book)`**: Delega a chamada para `bookDao.insertBook(book)`.
- **`updateBook(book: Book)`**: Delega a chamada para `bookDao.updateBook(book)`.
- **`deactivateBook(bookId: Int)`**: Delega a chamada para `bookDao.deactivateBook(bookId)`.
- **`activateBook(bookId: Int)`**: Delega a chamada para `bookDao.activateBook(bookId)`.

7. Camada de ViewModels

Os ViewModels contêm a lógica de apresentação e gerenciam o estado da UI, expondo dados e ações para as telas via `StateFlow`.

7.1. AuthViewModel (`AuthViewModel.kt` - Pag 20)

Gerencia o estado de autenticação do usuário.

- **`_currentUser: MutableStateFlow<User?>`**: Um `MutableStateFlow` privado que armazena o `User` atualmente logado. É exposto como `currentUser: StateFlow<User?>` para a UI, permitindo que ela observe as mudanças no estado de autenticação.
- **`_loginError: MutableStateFlow<String?>`**: Armazena mensagens de erro de login, exposto como `loginError`.
- **`_registrationError: MutableStateFlow<String?>`**: Armazena mensagens de erro de registro, exposto como `registrationError`.

- `login(username: String, passwordPlain: String, onLoginSuccess: () -> Unit)`:
  - Limpa erros de login anteriores.
  - Inicia uma coroutine no `viewModelScope` (associado ao ciclo de vida do ViewModel).
  - Chama `authRepository.loginUser()`.
  - Se o login for bem-sucedido (`user != null`), atualiza `_currentUser` e executa o callback `onLoginSuccess`.
  - Se falhar, define a mensagem em `_loginError`.
- `register(username: String, passwordPlain: String, onRegistrationSuccess: () -> Unit)`:
  - Limpa erros de registro anteriores.
  - Valida a entrada (`username` e `passwordPlain` não podem ser vazios, `passwordPlain` com no mínimo 6 caracteres). Se a validação falhar, define `_registrationError` e retorna.
  - Chama `authRepository.registerUser()`.
  - Se o registro for bem-sucedido, chama o próprio `login()` para logar o usuário recém-registrado e, em seguida, executa `onRegistrationSuccess`.
  - Se o registro falhar (ex: nome de usuário já existe), define `_registrationError`.
- `logout()`: Simplesmente define `_currentUser.value = null`, deslogando o usuário.
- `clearLoginError()` / `clearRegistrationError()`: Métodos públicos para limpar as mensagens de erro, geralmente chamados após a exibição do erro na UI.

7.2. AuthViewModelFactory (`AuthViewModelFactory.kt` - Pag 21)

Um `ViewModelProvider.Factory` customizado para `AuthViewModel`.

- `create(modelClass: Class<T>)`: Este método é invocado pelo sistema `viewModel()` do Compose quando um `AuthViewModel` é solicitado. Ele verifica se a `modelClass` é `AuthViewModel` e, se for, retorna uma nova instância de `AuthViewModel`, injetando o `AuthRepository` fornecido no construtor do factory. Isso é uma forma manual de injeção de dependência para ViewModels.

7.3. BookViewModel (`BookViewModel.kt` - Pag 22)

Gerencia o estado e as operações relacionadas aos livros, incluindo pesquisa e favoritos.

- **Construtor**: Recebe `bookRepository`, `favoriteManager` e, **criticamente**, `currentUserFlow: StateFlow<User?>` (o `currentUser` do `AuthViewModel`). Isso permite que `BookViewModel` reaja a mudanças no usuário logado.
- `_searchQuery: MutableStateFlow<String>`: Mantém o termo de busca atual, que é exposto como `searchQuery`.
- `_allBooks`: Um `Flow` que observa todos os livros diretamente do `bookRepository.getAllBooks()`.
- `favoriteBookIds: StateFlow<Set<Int>>`:
  - **Mecanismo:** Utiliza `currentUserFlow.flatMapLatest { ... }`.
  - **Lógica:** Quando `currentUserFlow` emite um novo usuário (ou nulo), `flatMapLatest` cancela a coroutine anterior e inicia uma nova.
    - Se `currentUser` **não for nulo**: Ele chama `favoriteManager.getFavoriteBookIds(currentUser.id.toString())` para obter os favoritos *desse usuário específico*.
    - Se `currentUser` **for nulo**: Ele emite um `flowOf(emptySet())`, indicando que não há favoritos para um usuário não logado.
  - `stateIn(...)`: Converte o `Flow` resultante em um `StateFlow`, garantindo que os dados sejam compartilhados e sobrevivam ao ciclo de vida.
- `books: StateFlow<List<Book>>`:
  - **Mecanismo:** Utiliza `combine(_allBooks, _searchQuery, currentUserFlow) { ... }`.
  - **Lógica:** Sempre que `_allBooks`, `_searchQuery` ou `currentUserFlow` emitem um novo valor, a função lambda é executada.
    - Primeiro, filtra `allBooks` com base em `_searchQuery` (título ou autor, ignorando maiúsculas/minúsculas).
    - Em seguida, verifica se o `currentUser` é um administrador (`isAdmin`).
    - Se `isAdmin` for `true`, todos os livros filtrados pela busca são retornados.
    - Se `isAdmin` for `false` (usuário comum), os livros são *adicionalmente* filtrados para incluir apenas aqueles onde `it.isActive` é `true`, garantindo que livros desativados não sejam exibidos para usuários comuns.
  - `stateIn(...)`: Converte o `Flow` combinado em um `StateFlow`.

- `favoriteBooks: StateFlow<List<Book>>`:
  - **Mecanismo:** Utiliza `combine(_allBooks, favoriteBookIds, currentUserFlow) { ... }`.
  - **Lógica:** Similar a `books`, mas focado nos favoritos.
    - Filtra `allBooks` para incluir apenas aqueles cujo `id` está presente em `favoriteBookIds` (que já reflete os favoritos do usuário atual).
    - Aplica o filtro `isActive` para usuários não-administradores, garantindo que mesmo um livro favorito desativado não seja exibido para usuários comuns.
  - `stateIn(...)`: Converte em `StateFlow`.
- `getBookById(bookId: Int)`: Apenas delega ao repositório.
- `addBook(book: Book)` / `updateBook(book: Book)` / `deactivateBook(bookId: Int)` / `activateBook(bookId: Int)`: Envolve as chamadas do repositório em `viewModelScope.launch { ... }` para execução assíncrona.
- `setSearchQuery(query: String)`: Atualiza o valor de `_searchQuery`, o que automaticamente re-emite o `books` `StateFlow` devido ao `combine` e atualiza a UI.
- `toggleFavorite(bookId: Int)`:
  - Inicia uma coroutine.
  - Obtém o `currentUser` usando `currentUserFlow.firstOrNull()`.
  - **Crucial:** Se um usuário estiver logado, ele recupera o `currentFavorites` *do* *`favoriteBookIds`* *`StateFlow`* (que já é específico do usuário).
  - Chama `favoriteManager.removeFavoriteBook()` ou `favoriteManager.addFavoriteBook()` com o `userId` e `bookId` apropriados. Se o usuário não estiver logado, uma mensagem de erro pode ser exibida (comentada no código).

7.4. BookViewModelFactory (`BookViewModelFactory.kt` - Pag 23)

Factory customizado para `BookViewModel`.

- `create(modelClass: Class<T>)`: Fornece as dependências (`bookRepository`, `favoriteManager`, e `currentUserFlow`) ao construtor do `BookViewModel` quando uma instância é solicitada. A passagem de `currentUserFlow` (do `AuthViewModel`) é um exemplo de injeção de dependência de `StateFlow` entre ViewModels.

8. Camada de Interface do Usuário (UI)

As telas são construídas com Jetpack Compose e observam os `StateFlow`s dos ViewModels para exibir dados e reagir a eventos do usuário.

8.1. Adicionar Livro (`AddBookScreen.kt` - Pag 12)

- **Componível**: `@Composable fun AddBookScreen(...)`
- **Funcionalidade**: Permite que usuários administradores adicionem novos livros ao catálogo.
- **Elementos da UI**:
  - `Scaffold` com `TopAppBar` (título "Adicionar Novo Livro" e botão de "Voltar").
  - `Column` centralizada contendo `OutlinedTextField`s para:
    - `title` (Título)
    - `author` (Autor)
    - `description` (Descrição)
    - `price` (Preço)
    - `quantity` (Quantidade)
    - `imageUrl` (URL da Imagem)
  - `Button("Adicionar Livro")`: Ao clicar, cria um objeto `Book` a partir dos inputs e chama `bookViewModel.addBook()`. Em seguida, navega de volta.
- **Observações**: Campos `price` e `quantity` deveriam ter validação de entrada para garantir que sejam números válidos.

8.2. Card de Livro (`BookCard.kt` - Pag 10)

Um componente reutilizável para exibir informações concisas de um livro em listas.

- **Componível**: `@Composable fun BookCard(...)`
- **Parâmetros**: Recebe um objeto `Book`, callbacks `onBookClick` e `onToggleFavorite`, e um booleano `isFavorite` para controlar a cor do ícone de favorito.
- **Layout**: Utiliza `Card` do Material3 com um `Row` para organizar imagem e texto.

    - **Imagem**: `AsyncImage(model = book.imageUrl, ...)` do Coil, exibe a imagem do livro. Inclui um `contentDescription` para acessibilidade.
    - **Informações do Livro**: `Column` para `title`, `author`, `price`, `quantity`. O preço é formatado para moeda.
    - **Ícone de Favorito**: `IconButton` com `Icons.Filled.Favorite`. A cor do ícone (`tint`) muda para `MaterialTheme.colorScheme.error` se `isFavorite` for `true`, indicando que está marcado como favorito. O `onClick` chama `onToggleFavorite`.
    - **Status Inativo**: Um `Text` adicional "Inativo" é exibido com um fundo vermelho opaco se `book.isActive` for `false`, alertando o usuário sobre o status do livro.
- **Interatividade**:
    - O `Card` inteiro tem um `clickable` que chama `onBookClick(book.id)`, permitindo a navegação para os detalhes do livro.
    - O `IconButton` permite alternar o status de favorito.

8.3. Tela de Catálogo (`CatalogScreen.kt` - Pag 13)

Exibe a lista principal de livros, permitindo rolagem e interação.

- **Componível**: `@Composable fun CatalogScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `onBookClick` e `paddingValues`.
- **Coleta de Dados**:
    - `val books by bookViewModel.books.collectAsState(initial = emptyList())`: Observa o `StateFlow` `books` do `BookViewModel`, que já contém a lógica de filtragem por busca e por status `isActive` (para não-administradores).
    - `val favoriteBookIds by bookViewModel.favoriteBookIds.collectAsState(initial = emptySet())`: Observa os IDs dos livros favoritos do usuário atual.
- **Layout**:
    - `if (books.isEmpty())`: Exibe uma mensagem "Nenhum livro encontrado." se a lista estiver vazia.
    - `LazyColumn`: Componente de Compose para listas roláveis de alta performance, que só renderiza os itens visíveis.
        - `contentPadding` e `verticalArrangement`: Adicionam espaçamento ao redor e entre os itens.

    - `items(books) { book -> ... }`: Itera sobre a lista de `books`.
        - Para cada `book`, renderiza um `BookCard`, passando o `book`, o `onBookClick`, o `onToggleFavorite` (que chama `bookViewModel.toggleFavorite`), e `isFavorite = book.id in favoriteBookIds`.

8.4. Tela de Detalhes/Edição de Livro (`EditBookScreen.kt` - Pag 14)

Permite visualizar os detalhes de um livro e, para administradores, editar suas propriedades.

- **Componível**: `@Composable fun EditBookScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `authViewModel`, `bookId` (do Navigation) e `onBack`.
- **Coleta de Dados**:
    - `val book by bookViewModel.getBookById(bookId).collectAsState(initial = null)`: Observa o livro específico do `BookViewModel`.
    - `val currentUser by authViewModel.currentUser.collectAsState()`: Observa o usuário logado para determinar permissões.
    - `val isAdmin = currentUser?.role == "admin"`: Calcula se o usuário é administrador.
- **Estado Local**: Variáveis `mutableStateOf` para armazenar o estado editável dos campos do livro (título, autor, etc.). `LaunchedEffect(book)` é usado para preencher esses estados quando o `book` é carregado ou muda.
- **`Scaffold` e `TopAppBar`**:
    - Título "Detalhes do Livro" ou "Editar Livro" (para admin).
    - Botão "Voltar" (`ArrowBack`).
    - **Ação para Administrador**: Um `IconButton` para salvar (`Save`) é visível apenas se `isAdmin` for `true`.
    - **Toggle Ativar/Desativar**: Um `IconButton` com ícones `PowerSettingsNew` (ativo) ou `PowerOff` (inativo) é visível apenas para administradores. O `onClick` alterna o status `isActive` do livro chamando `bookViewModel.activateBook()` ou `deactivateBook()`. A cor do ícone (`tint`) muda com o status.
- **`Column` Principal**: Exibe os `OutlinedTextField`s para os detalhes do livro.

    - `readOnly = !isAdmin`: Os campos de texto são `readOnly` (somente leitura) se o usuário não for um administrador, impedindo edições não autorizadas.
    - Exibe `AsyncImage` para a capa do livro.
- **Toast para Sucesso**: Um `LaunchedEffect(saveSuccess)` pode ser usado para exibir um `Toast` quando a atualização é bem-sucedida.

8.5. Tela de Livros Favoritos (`FavoriteBooksScreen.kt` - Pag 15)

Exibe a lista de livros que o usuário logado marcou como favoritos.

- **Componível**: `@Composable fun FavoriteBooksScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `onBookClick`, `onBack`.
- **Coleta de Dados**:
    - `val favoriteBooks by bookViewModel.favoriteBooks.collectAsState(initial = emptyList())`: Observa o `StateFlow` `favoriteBooks` do `BookViewModel`, que já está filtrado para mostrar apenas os favoritos do usuário logado e aplicando o filtro `isActive` para usuários comuns.
- **Layout**:
    - `Scaffold` com `TopAppBar` (título "Meus Favoritos" e botão "Voltar").
    - `Column` principal.
    - `if (favoriteBooks.isEmpty())`: Se a lista de favoritos estiver vazia, exibe uma mensagem informativa.
    - `LazyColumn`: Exibe a lista de `favoriteBooks`.
        - Cada `book` é renderizado com um `BookCard`.
        - `isFavorite = true` é passado, pois todos os livros nesta tela *são* favoritos.
        - `onToggleFavorite` permite que o usuário desmarque um favorito diretamente desta tela.

8.6. Tela Principal do Aplicativo (`HomeScreen.kt` - Pag 16)

A tela principal que o usuário vê após o login, agindo como um painel de controle.

- **Componível**: `@Composable fun HomeScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `bookViewModel`, callbacks para navegação (`onNavigateToAddBook`, `onBookClick`, `onLogout`, `onNavigateToFavoriteBooks`).

- **Coleta de Dados**:
  - `val currentUser by authViewModel.currentUser.collectAsState()`: Observa o usuário atual para determinar o papel.
  - `val isAdmin = currentUser?.role == "admin"`: Define a permissão administrativa.
  - `val searchQuery by bookViewModel.searchQuery.collectAsState()`: Observa o termo de busca atual.
- `Scaffold`: Estrutura a tela.
  - `topBar` **(TopAppBar)**:
    - Título "Livraria App".
    - `actions` **(Ícones na barra superior)**:
      - `IconButton(onClick = onNavigateToFavoriteBooks)`: Ícone de **Favoritos** (`Icons.Filled.Favorite`).
      - `if (isAdmin) { IconButton(onClick = onNavigateToAddBook) { Icon(Icons.Filled.Add, ...) } }`: Ícone de **Adicionar Livro** (`Icons.Filled.Add`) **visível apenas para administradores**.
      - `IconButton(onClick = onLogout) { Icon(Icons.Filled.ExitToApp, ...) }`: Ícone de **Sair** (`Icons.Filled.ExitToApp`) para deslogar.
    - Cores personalizadas para a `TopAppBar`.
  - **Remoção de** `bottomBar` **e** `floatingActionButton`: Essas seções foram removidas do `Scaffold`, simplificando a navegação e consolidando as ações principais na `TopAppBar`.
- `Column` **Principal**:
  - `SearchBar`: Componente reutilizável para a funcionalidade de busca.
    - `query = searchQuery`, `onQueryChange = { bookViewModel.setSearchQuery(it) }`: Vincula a entrada da barra de busca ao `searchQuery` do `BookViewModel`, acionando a filtragem.
  - `CatalogScreen`: Exibe o catálogo de livros, recebendo o `bookViewModel` e os callbacks de clique.

8.7. Tela de Login (`LoginScreen.kt` - Pag 17)

A primeira tela que os usuários veem para acessar o aplicativo.

- **Componível**: `@Composable fun LoginScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `onLoginSuccess`, `onNavigateToRegister`.
- **Estado Local**: `username` e `password` são gerenciados por `mutableStateOf`.
- **Tratamento de Erro**:
  - `val loginError by authViewModel.loginError.collectAsState()`: Observa o `loginError` do ViewModel.
  - `LaunchedEffect(loginError)`: Um efeito lateral que é disparado sempre que `loginError` muda.
    - `loginError?.let { ... }`: Se houver um erro, exibe um `Toast` com a mensagem de erro e então chama `authViewModel.clearLoginError()` para limpar o erro no ViewModel, evitando que o Toast apareça novamente após uma recriação da Composable.
- **Layout**:
  - `Column` centralizada com `Arrangement.Center` e `Alignment.CenterHorizontally`.
  - `Text("Bem-vindo à Livraria!")` com estilo de título.
  - `OutlinedTextField`s para "Nome de Usuário" e "Senha". A senha usa `PasswordVisualTransformation()` para ocultar os caracteres.
  - `Button("Entrar")`: `onClick` chama `authViewModel.login(username, password, onLoginSuccess)`.
  - `TextButton("Não tem uma conta? Cadastre-se")`: `onClick` chama `onNavigateToRegister`.
- **Preview**: `@Preview` para visualização rápida da tela no Android Studio.

8.8. Tela de Registro (`RegisterScreen.kt` - Pag 18)

Permite que novos usuários criem uma conta.

- **Componível**: `@Composable fun RegisterScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `onRegistrationSuccess`, `onBack`.
- **Estado Local**: `username`, `password`, `confirmPassword`.
- **Tratamento de Erro**: Similar ao `LoginScreen`, usando `LaunchedEffect` para observar `registrationError` e exibir `Toast`s.
- **Layout**:
  - `Scaffold` com `TopAppBar` (título "Criar Nova Conta" e botão "Voltar").

    - `Column` centralizada com `OutlinedTextField`s para nome de usuário, senha e confirmação de senha (ambas com `PasswordVisualTransformation`).
    - `Button("Registrar")`:
        - `onClick` contém uma validação local: `if (password == confirmPassword)`.
        - Se as senhas coincidirem, chama `authViewModel.register(...)`.
        - Se não coincidirem, exibe um `Toast` de "As senhas não coincidem!".
- **Preview**: `@Preview` para visualização.

8.9. Barra de Busca (`SearchBar.kt` - Pag 19)

Um componente de UI genérico para funcionalidade de busca.

- **Componível**: `@Composable fun SearchBar(...)`
- **Parâmetros**: `query` (o termo de busca atual), `onQueryChange` (callback para quando o termo de busca muda), `modifier`.
- **Layout**:
    - `OutlinedTextField`: Um campo de texto com bordas e rótulo.
        - `value = query`, `onValueChange = onQueryChange`: Vincula o valor do campo ao `query` e chama o callback `onQueryChange` quando o texto é modificado.
        - `placeholder`: Texto de dica "Buscar livros por título ou autor...".
        - `leadingIcon`: Exibe um ícone de busca (`Icons.Default.Search`).
        - `trailingIcon`: Exibe um ícone de "fechar" (`Icons.Default.Close`) **apenas se a `query` não estiver vazia**, permitindo limpar a busca rapidamente.
        - `singleLine = true`: Garante que o campo de texto seja uma única linha.
        - `colors`: Personaliza as cores da barra de busca usando o tema Material3.

9. Sistema de Navegação (`AppNavHost.kt`)

Gerencia as transições entre as diferentes telas (Composables) do aplicativo.

- **Componível**: `@Composable fun AppNavHost(...)`
- **Parâmetros**: Recebe `navController`, `authViewModel`, `bookViewModel`.

- `NavHost`: O principal componente de navegação, que define o grafo de navegação.
  - `navController`: A instância do controlador de navegação.
  - `startDestination = "login"`: A tela inicial do aplicativo.
  - **Definição das Rotas**:
    - `composable("login") { ... }`: Define a rota de login.
      - Chama `LoginScreen`, passando `authViewModel` e callbacks para navegação (e.g., `onLoginSuccess = { navController.navigate("home") { ... } }`).
    - `composable("register") { ... }`: Define a rota de registro.
      - Chama `RegisterScreen`, passando `authViewModel` e callbacks para navegação (`onRegistrationSuccess = { navController.popBackStack() }`, `onBack = { navController.popBackStack() }`).
    - `composable("home") { ... }`: Define a rota da tela principal.
      - Chama `HomeScreen`, passando `authViewModel`, `bookViewModel` e todos os callbacks necessários para navegação para outras telas (adicionar livro, favoritos, logout, detalhes do livro).
    - `composable("addBook") { ... }`: Define a rota para adicionar livro.
    - `composable("editBook/{bookId}", arguments = listOf(navArgument("bookId") { type = NavType.IntType })) { backStackEntry -> ... }`: Define a rota para editar livro.
      - **Argumentos de Navegação**: `arguments = listOf(navArgument("bookId") { ... })` especifica que esta rota espera um argumento inteiro chamado `bookId`.
      - `val bookId = backStackEntry.arguments?.getInt("bookId")`: Extrai o `bookId` dos argumentos da `backStackEntry`.
      - Chama `EditBookScreen`, passando o `bookId` extraído.
    - `composable("favoriteBooks") { ... }`: Define a rota para os livros favoritos.

10. Fluxos de Interação Exemplo

10.1. Fluxo de Autenticação (Login)

1. **Usuário abre o App**: `MainActivity` inicializa e `AppNavHost` exibe `LoginScreen`.

2. **Usuário insere credenciais**: Digita `username` e `password` nos `OutlinedTextField`s da `LoginScreen` (Pag 17).
3. **Clica em "Entrar"**:
    - `LoginScreen` chama `authViewModel.login(username, password, onLoginSuccess)`.
    - `AuthViewModel` (Pag 20) chama `authRepository.loginUser()`.
    - `AuthRepository` (Pag 9) busca o usuário no `UserDao` e verifica a senha com BCrypt.
    - **Sucesso**: `AuthViewModel` atualiza `_currentUser`, e o callback `onLoginSuccess` é executado.
        - `AppNavHost` (`onLoginSuccess`) navega para a rota "home" (`HomeScreen`).
    - **Falha**: `AuthViewModel` define `_loginError`.
        - `LoginScreen` observa `loginError` (`LaunchedEffect`) e exibe um `Toast` com a mensagem de erro.

10.2. Fluxo de Registro

1. **Usuário está na `LoginScreen`**: Clica em "Não tem uma conta? Cadastre-se" (Pag 17).
2. **Navegação**: `LoginScreen` chama `onNavigateToRegister`, e `AppNavHost` navega para `RegisterScreen`.
3. **Usuário preenche dados**: Insere `username`, `password`, `confirmPassword` na `RegisterScreen` (Pag 18).
4. **Clica em "Registrar"**:
    - `RegisterScreen` valida se `password` e `confirmPassword` são iguais.
    - Se iguais, chama `authViewModel.register(username, password, onRegistrationSuccess)`.
    - `AuthViewModel` (Pag 20) realiza validações internas (vazio, comprimento da senha).
    - `AuthViewModel` chama `authRepository.registerUser()`.
    - `AuthRepository` (Pag 9) verifica se o nome de usuário já existe, gera hash da senha e insere no `UserDao`.
    - **Sucesso**: `AuthViewModel` chama `login()` automaticamente. Após o login, `onRegistrationSuccess` é executado.

    - `AppNavHost` (`onRegistrationSuccess`) faz `navController.popBackStack()` (retorna para a tela de login, que imediatamente redireciona para Home se o login automático funcionar) ou pode ser adaptado para ir direto para Home.
- **Falha**: `AuthViewModel` define `_registrationError`.
    - `RegisterScreen` observa `registrationError` e exibe um `Toast`.

10.3. Fluxo de Gerenciamento de Livros (Admin)

1. **Admin logado na `HomeScreen`**: Ícone "Adicionar Livro" está visível (Pag 16).
2. **Admin clica em "Adicionar Livro"**:
    - `HomeScreen` chama `onNavigateToAddBook`, e `AppNavHost` navega para `AddBookScreen`.
3. **Admin preenche detalhes e clica "Adicionar Livro"**:
    - `AddBookScreen` (Pag 12) cria um objeto `Book` e chama `bookViewModel.addBook(book)`.
    - `BookViewModel` (Pag 22) chama `bookRepository.insertBook(book)`.
    - `BookRepository` (Pag 8) insere o livro no `BookDao`.
    - O `_allBooks` `Flow` no `BookViewModel` é automaticamente atualizado, e `books` `StateFlow` é re-emitido, atualizando a `CatalogScreen` em `HomeScreen`.
4. **Admin clica em um `BookCard`**:
    - `BookCard` (Pag 10) chama `onBookClick(bookId)`.
    - `AppNavHost` navega para `editBook/{bookId}`, passando o `bookId`.
5. **Admin edita detalhes ou status `isActive`**:
    - `EditBookScreen` (Pag 14) exibe os detalhes do livro.
    - Admin modifica campos ou clica no ícone de ativar/desativar.
    - **Salvar**: Clica no ícone de salvar na `TopAppBar`. `EditBookScreen` chama `bookViewModel.updateBook(editedBook)` (se for edição de detalhes) ou `bookViewModel.activateBook()` / `deactivateBook()` (se for toggle de status).
    - `BookViewModel` chama o método correspondente no `BookRepository`, que atualiza o `BookDao`.

- A `CatalogScreen` (Pag 13) e a `FavoriteBooksScreen` (Pag 15) automaticamente se atualizam devido à observação dos `Flow`s.

10.4. Fluxo de Favoritos (Usuário Comum)

1. **Usuário comum logado na `HomeScreen`**: Vê o catálogo de livros na `CatalogScreen`.
2. **Usuário clica no ícone de coração em um `BookCard`**:
   - `BookCard` (Pag 10) chama `onToggleFavorite(book.id)`.
   - `BookViewModel` (Pag 22) observa `toggleFavorite(bookId)`.
   - `BookViewModel` verifica o `currentUser` e o `favoriteBookIds` atual, então chama `favoriteManager.addFavoriteBook()` ou `removeFavoriteBook()`.
   - `FavoriteManager` (Pag 11) atualiza o `DataStore`.
   - O `favoriteBookIds` `StateFlow` no `BookViewModel` é re-emitido, fazendo com que o `BookCard` na `CatalogScreen` atualize sua cor de coração, e o `favoriteBooks` `StateFlow` (usado na `FavoriteBooksScreen`) também é atualizado.
3. **Usuário clica no ícone de Favoritos na `TopAppBar`**:
   - `HomeScreen` (Pag 16) chama `onNavigateToFavoriteBooks`.
   - `AppNavHost` navega para `FavoriteBooksScreen`.
4. **`FavoriteBooksScreen` exibe livros favoritos**:
   - A tela observa `bookViewModel.favoriteBooks` e exibe apenas os livros que o usuário marcou como favoritos.

11. Tecnologias Utilizadas (Aprofundamento)

- **Jetpack Compose**: Sistema de UI moderno e declarativo para Android.
  - **Benefícios**: Reduz boilerplate, melhora a legibilidade, e simplifica o desenvolvimento de UIs complexas.
- **Room Persistence Library**: Camada de abstração sobre SQLite.
  - **Benefícios**: Objeto-relacional mapping (ORM), verificação de SQL em tempo de compilação, integração com `Flow` para reatividade, minimiza código repetitivo.
- **Kotlin Coroutines & Flow**:

    - **Coroutines**: Para operações assíncronas e não bloqueantes, como acesso a banco de dados e hashing de senha.
    - **Flow**: Stream assíncrono de dados que pode emitir múltiplos valores ao longo do tempo. Ideal para observar mudanças no banco de dados (`BookDao`, `UserDao`) e preferências de usuário (`FavoriteManager`), permitindo que a UI se atualize automaticamente.
- **Jetpack Navigation Compose**: Gerencia a navegação entre as telas.
    - **Benefícios**: Define o grafo de navegação, passa argumentos de forma segura, gerencia a pilha de backstack e simplifica as transições.
- **Jetpack ViewModel**:
    - **Benefícios**: Mantém o estado da UI de forma otimizada para o ciclo de vida, sobrevivendo a mudanças de configuração (rotações de tela), e facilita a separação de preocupações (lógica da UI separada da View).
- **DataStore Preferences**:
    - **Benefícios**: Alternativa moderna ao `SharedPreferences`, opera de forma assíncrona com `Flow` (Kotlin Coroutines) e garante consistência transacional para pequenas quantidades de dados.
- **BCrypt (JBCrypt)**: Algoritmo de hashing de senha.
    - **Benefícios**: Forte contra ataques de força bruta devido ao seu fator de trabalho (iterações configuráveis), e incorpora um "sal" para proteger contra ataques de rainbow table. Essencial para segurança.
- **Coil**: Biblioteca de carregamento de imagens.
    - **Benefícios**: Rápida, leve, moderna, otimizada para Compose (`AsyncImage`), e oferece funcionalidades como cache de memória e disco, transformações de imagem e placeholders.

12. Próximos Passos e Oportunidades de Melhoria

Ainda que funcional, o LivrariaAppV2 pode ser aprimorado em diversas áreas para um ambiente de produção.

- **Validação de Entrada Robusta**:
    - Implementar validação em tempo real para todos os campos de entrada (ex: senhas com caracteres especiais, nomes de usuário sem espaços, URLs de imagem válidas, preço/quantidade como números positivos).

    - Fornecer feedback visual claro para o usuário sobre campos inválidos, não apenas `Toast`s.
- **Tratamento de Estado de Carregamento e Erros (UI)**:
    - Exibir indicadores de carregamento (`CircularProgressIndicator`) para operações de longa duração (login, registro, busca de dados).
    - Utilizar `Snackbar` para mensagens de erro/sucesso temporárias que podem ser dispensadas.
    - Implementar telas de erro mais amigáveis com opções de "Tentar Novamente" para falhas de rede ou banco de dados.
- **Persistência de Sessão do Usuário**:
    - Atualmente, o usuário é deslogado ao fechar o aplicativo. Para manter o login, o `AuthViewModel` precisaria salvar o ID ou um token do usuário no `DataStore` (ou `SharedPreferences criptografados`) e carregá-lo na inicialização do aplicativo.
- **Função "Esqueci a Senha"**:
    - Implementar um fluxo para redefinição de senha (em um aplicativo real, isso envolveria comunicação com um backend).
- **Refatoração do Login Automático Após Registro**:
    - Embora conveniente para o desenvolvimento, em um aplicativo real, após o registro, o ideal seria que o usuário fosse redirecionado para a tela de login para um login explícito.
- **Internacionalização (i18n)**:
    - Externalizar todas as strings da UI para arquivos de recursos (`strings.xml`) para facilitar a tradução do aplicativo para diferentes idiomas.
- **Acessibilidade (a11y)**:
    - Revisar e garantir que todos os elementos da UI tenham `contentDescription`s adequadas, permitindo que usuários com deficiência visual possam usar o aplicativo com leitores de tela.
- **Testes Abrangentes**:
    - **Testes Unitários**: Cobrir a lógica de negócios em ViewModels, Repositórios e DAOs.

    - **Testes de Integração**: Verificar a interação entre as camadas (ex: Repositório e Room, ViewModel e Repositório).
    - **Testes de UI (Instrumentados/Compose Tests)**: Validar o comportamento da interface do usuário e as transições entre telas.
- **Segurança Adicional**:
    - Considerar o uso de `AndroidKeystore` para armazenar chaves criptográficas, se informações mais sensíveis além das senhas forem persistidas localmente.
    - Para um ambiente de produção, o "sal" para o BCrypt deveria ser gerado e gerenciado de forma mais robusta.
- **Lógica de Migração de Banco de Dados**:
    - Substituir `fallbackToDestructiveMigration()` no `AppDatabase` por migrações incrementais (`Room.addMigrations()`) para preservar os dados dos usuários em futuras atualizações do aplicativo.
- **Gerenciamento de Imagens**:
    - Permitir que o usuário envie imagens (em vez de apenas URLs), o que envolveria permissões de armazenamento e upload para um serviço de armazenamento em nuvem.
- **Permissões de Usuário (Admin)**:
    - Uma tela de gerenciamento de usuários para administradores, permitindo-lhes alterar papéis de usuário ou desativar contas.

Objetivos Principais:

- **Catálogo de Livros:** Exibir uma vasta coleção de livros de forma organizada.
- **Pesquisa Dinâmica:** Permitir que os usuários encontrem livros rapidamente por título ou autor.
- **Gestão de Favoritos:** Capacitar os usuários a marcar e gerenciar seus livros preferidos, com persistência por usuário.
- **Autenticação e Autorização:** Implementar um sistema de login e registro, distinguindo papéis de usuário (usuário comum vs. administrador) para controle de acesso a funcionalidades específicas.

- **Gerenciamento Administrativo:** Fornecer aos administradores a capacidade de adicionar novos livros, editar detalhes de livros existentes e gerenciar o status de ativação/desativação de livros.
- **Persistência de Dados:** Armazenar informações de usuários e livros localmente no dispositivo.

Tecnologias-Chave:

- **Jetpack Compose:** Para a construção da interface do usuário declarativa.
- **Room Persistence Library:** Para o gerenciamento de banco de dados SQLite local.
- **Kotlin Coroutines & Flow:** Para programação assíncrona reativa e gerenciamento de streams de dados.
- **Jetpack Navigation Compose:** Para a navegação entre as telas do aplicativo.
- **Jetpack ViewModel & LiveData/StateFlow:** Para o gerenciamento do estado da UI de forma otimizada para o ciclo de vida.
- **DataStore Preferences:** Para armazenamento persistente e assíncrono de pares chave-valor, utilizado na gestão de favoritos.
- **BCrypt (JBCrypt):** Para hashing seguro de senhas.
- **Coil:** Para carregamento e exibição eficiente de imagens de rede.

2. Arquitetura do Aplicativo (MVVM)

O LivrariaAppV2 adota a arquitetura **MVVM (Model-View-ViewModel)**, que promove a separação de preocupações e a testabilidade do código.

- **Model (Camada de Dados):**
  - Representa os dados e a lógica de negócios.
  - No projeto, inclui as classes de entidades (`Book`, `User`), os DAOs (`BookDao`, `UserDao`), o banco de dados Room (`AppDatabase`), o `FavoriteManager` e as classes de repositório (`AuthRepository`, `BookRepository`).
  - É responsável por buscar, armazenar e manipular os dados, isolando a lógica de acesso a dados do resto do aplicativo.
- **View (Camada de UI - Jetpack Compose):**

    - Representa a interface do usuário.
    - No projeto, são as Composable Functions nas telas (`LoginScreen`, `HomeScreen`, `BookDetailsScreen`, etc.) e componentes de UI (`BookCard`, `SearchBar`).
    - É responsável por observar os dados expostos pelo ViewModel e exibir o estado atual da UI. Ela não contém lógica de negócios ou acesso direto a dados.
- **ViewModel (Camada de Lógica da UI):**
    - Atua como um intermediário entre a View e o Model.
    - No projeto, são as classes `AuthViewModel` e `BookViewModel`.
    - Mantém o estado da UI e a lógica de apresentação, expondo dados e ações para a View através de `StateFlow`s. Ele não possui referência direta à View, garantindo que não haja vazamentos de memória e facilitando a testabilidade.
    - Inicia operações assíncronas (via Coroutines) para interagir com os Repositórios e atualizar o estado da UI.

Essa separação clara entre as camadas melhora a manutenibilidade, escalabilidade e testabilidade do aplicativo.

3. Configurações e Inicialização Global

3.1. Classe `Application` (`LivrariaAppV2Application.kt` - Pag 24)

Esta classe estende `android.app.Application` e serve como o ponto de entrada global do aplicativo.

- `onCreate()`:
    - É o primeiro método a ser chamado quando o processo do aplicativo é criado.
    - **Configuração do Coil:** O bloco `Coil.setImageLoader(imageLoader)` inicializa a biblioteca Coil para carregamento de imagens. O `DebugLogger()` é habilitado, o que é extremamente útil durante o desenvolvimento para diagnosticar problemas de carregamento de imagens, exibindo informações detalhadas no Logcat sobre o ciclo de vida e o desempenho do carregamento de cada imagem. Em um ambiente de produção, este logger seria geralmente desabilitado para otimização.

3.2. Atividade Principal (`MainActivity.kt` - Pag 25)

A `MainActivity` é a única atividade no aplicativo e é responsável por configurar o ambiente inicial, instanciar as dependências da camada de dados/repositório e configurar o sistema de navegação da interface do usuário.

- **`onCreate(savedInstanceState: Bundle?)`**:
  - **Criação de Dependências de Camada de Dados:**
    - `AppDatabase.getDatabase(applicationContext)`: Obtém a instância singleton do banco de dados Room. O `applicationContext` é usado para evitar vazamentos de memória associados ao contexto da Atividade.
    - `userDao` e `bookDao`: Obtêm as interfaces DAO para interagir com as tabelas de usuários e livros, respectivamente.
    - `AuthRepository(userDao)`: Instancia o repositório de autenticação, injetando o `UserDao`.
    - `BookRepository(bookDao)`: Instancia o repositório de livros, injetando o `BookDao`.
    - `FavoriteManager(applicationContext)`: Instancia o gerenciador de favoritos, passando o `applicationContext` para acesso ao DataStore.
  - **Criação de `ViewModelFactory`s:**
    - `AuthViewModelFactory(authRepository)`: Cria um factory para `AuthViewModel`, injetando `AuthRepository`.
    - `BookViewModelFactory(bookRepository, favoriteManager, authViewModel.currentUser)`: Cria um factory para `BookViewModel`. É crucial notar que `authViewModel.currentUser` (um `StateFlow<User?>`) é passado *como uma dependência*. Isso permite que `BookViewModel` observe o estado de autenticação do usuário em tempo real, sem criar uma dependência circular direta entre os ViewModels.
  - **`setContent`**: Bloco principal do Jetpack Compose que define a hierarquia da UI.
    - `LivrariaAppV2Theme`: Aplica o tema visual definido para o aplicativo.
    - `Surface`: Um contêiner que define a superfície primária da UI, usando a cor de fundo do tema.
    - **Instanciação dos ViewModels:**

        - `val authViewModel: AuthViewModel = viewModel(factory = authViewModelFactory)`: Instancia o `AuthViewModel` usando o factory pré-definido.
        - `val bookViewModel: BookViewModel = viewModel(factory = bookViewModelFactory)`: Instancia o `BookViewModel`, aproveitando o factory que já recebeu o `currentUser` do `authViewModel`. O uso de `viewModel()` dentro de um `@Composable` garante que o ViewModel sobreviva a mudanças de configuração e seja associado ao ciclo de vida correto.
    - `rememberNavController()`: Cria e lembra uma instância de `NavController` para gerenciar a navegação.
    - `AppNavHost`: O componente central de navegação, que recebe o `navController` e as instâncias de `authViewModel` e `bookViewModel` para que as telas dentro do grafo de navegação possam acessá-los.

4. Configuração do Projeto Gradle (`build.gradle.kts (app)` - Pag 26)

Este arquivo define as configurações de build e as dependências do módulo do aplicativo.

- `plugins` **bloco**:
    - `alias(libs.plugins.android.application)`: Plugin para aplicações Android.
    - `alias(libs.plugins.kotlin.android)`: Plugin para o suporte a Kotlin.
    - `id("kotlin-kapt")`: Plugin essencial para o processamento de anotações do Kotlin (utilizado pelo Room para gerar código).
    - `alias(libs.plugins.kotlin.compose)`: Plugin para habilitar o Jetpack Compose.
- `android` **bloco**:
    - `namespace`: Define o namespace do pacote Java/Kotlin.
    - `compileSdk`: Versão do SDK Android que o projeto usa para compilar (API 36).
    - `defaultConfig`:
        - `applicationId`: ID único do aplicativo.
        - `minSdk`: Versão mínima do SDK Android suportada (API 24, Android 7.0 Nougat).

        - `targetSdk`: Versão do SDK Android para a qual o aplicativo é otimizado (API 36).
        - `versionCode` e `versionName`: Versão interna e externa do aplicativo.
        - `testInstrumentationRunner`: Runner de teste para instrumentação.
        - `vectorDrawables { useSupportLibrary = true }`: Habilita o uso de `VectorDrawable`s em versões mais antigas do Android.
    - `buildTypes`: Configura diferentes tipos de build (e.g., `release`, `debug`).
        - `release`: Define configurações para o build de lançamento (minificação desabilitada, uso de `proguardFiles`).
    - `compileOptions`: Define a versão do Java para a compilação (Java 11).
    - `kotlinOptions`: Define a versão da JVM target para o Kotlin (JVM 11).
    - `buildFeatures { compose = true }`: Sinaliza ao Gradle que este módulo usa o Jetpack Compose.
    - `composeOptions { kotlinCompilerExtensionVersion = "1.6.10" }`: Especifica a versão do compilador do Compose, garantindo compatibilidade entre o Compose e o Kotlin.
    - `packaging`: Configura como os recursos são empacotados, excluindo arquivos específicos para evitar conflitos de licença ou metadados desnecessários.
- `dependencies` **bloco**: Lista de todas as bibliotecas de terceiros e módulos internos.
    - **Dependências do Compose**: Essenciais para o funcionamento do Jetpack Compose (e.g., `ui`, `graphics`, `material3`, `activity-compose`). `platform(libs.androidx.compose.bom)` garante que todas as bibliotecas Compose usem versões compatíveis entre si.
    - **Room (SQLite)**:
        - `implementation(libs.androidx.room.runtime)`: Biblioteca principal do Room.
        - `kapt(libs.androidx.room.compiler)`: Processador de anotações que gera o código necessário para o Room.
        - `implementation(libs.androidx.room.ktx)`: Extensões Kotlin para Room, incluindo suporte a Coroutines e Flow.

- **Coroutines**:
  - `implementation(libs.kotlinx.coroutines.core)`: Coroutines para lógica agnóstica de plataforma.
  - `implementation(libs.kotlinx.coroutines.android)`: Extensões de Coroutines para Android, incluindo dispatchers de UI.
- **ViewModel e LiveData/StateFlow**:
  - `implementation(libs.androidx.lifecycle.viewmodel.ktx)`: Extensões Kotlin para ViewModel.
  - `implementation(libs.androidx.lifecycle.viewmodel.compose)`: Extensões específicas para usar ViewModels em Composable functions (`viewModel()`).
- **Navigation Compose**:
  - `implementation(libs.androidx.navigation.compose)`: Biblioteca para gerenciar a navegação em Jetpack Compose.
- **Material Icons Extended**:
  - `implementation(libs.androidx.compose.material.icons.extended)`: Fornece um conjunto maior de ícones do Material Design.
- **Hashing de senha (BCrypt)**:
  - `implementation(libs.jbcrypt)` e `implementation("org.mindrot:jbcrypt:0.4")`: Bibliotecas para hashing de senhas usando o algoritmo BCrypt, essencial para segurança ao armazenar credenciais de usuário.
- **Carregamento de imagens (Coil)**:
  - `implementation(libs.coil.compose)`: Biblioteca leve e rápida para carregamento de imagens assíncrono em Jetpack Compose.
- **DataStore (para FavoriteManager)**:
  - `implementation(libs.androidx.datastore.preferences)`: Biblioteca para armazenamento persistente de dados pequenos e tipados de forma assíncrona.
- **Dependências de teste**: Bibliotecas para realizar testes unitários (JUnit) e de instrumentação (AndroidX Test, Compose Test).

5. Camada de Dados (Model)

Esta camada é responsável por encapsular a lógica de acesso a dados e as regras de negócio relacionadas aos dados.

5.1. Modelos de Dados (`data.model`)

Definem as estruturas dos objetos que representam os dados do aplicativo.

- `Book.kt` :Kotlin

```
@Entity(tableName = "books")
data class Book(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val title: String,
    val author: String,
    val description: String,
    val price: Double,
    val quantity: Int,
    val imageUrl: String = "",
    val isActive: Boolean = true // Indica se o livro está ativo/disponível
)
```

  - `@Entity(tableName = "books")` : Anotação Room que mapeia esta classe para uma tabela SQLite chamada "books".
  - `@PrimaryKey(autoGenerate = true)` : Define `id` como a chave primária auto-gerada.
  - `isActive: Boolean` : Campo booleano crucial que permite aos administradores "desativar" um livro sem removê-lo completamente. Livros inativos não são visíveis para usuários comuns no catálogo.

- `User.kt` :Kotlin

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val username: String,
    val passwordHash: String, // Senha armazenada como hash
    val role: String // Ex: "user", "admin"
)
```

  - `@Entity(tableName = "users")` : Mapeia para a tabela SQLite "users".
  - `passwordHash: String` : Armazena a senha do usuário já criptografada com BCrypt, nunca a senha em texto claro, para segurança.
  - `role: String` : Define o papel do usuário no sistema, controlando o acesso a funcionalidades administrativas.

5.2. Banco de Dados Room (`AppDatabase.kt` , DAOs)

Room é a camada de abstração do SQLite que facilita o trabalho com o banco de dados local.

- `AppDatabase.kt` :Kotlin

```
@Database(entities = [User::class, Book::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun bookDao(): BookDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "livraria_database"
                )
                .fallbackToDestructiveMigration() // Para facilitar desenvolvimento, em produção usar migrations
                .build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

  - `@Database(...)`: Anotação que define as entidades pertencentes a este banco de dados (`User` , `Book`), a `version` (para controle de migrações) e `exportSchema` .
  - **Métodos Abstratos para DAOs**: `userDao()` e `bookDao()` fornecem acesso às interfaces DAO.
  - **Padrão Singleton (`companion object`)**: O método `getDatabase()` implementa o padrão Singleton para garantir que apenas uma instância do banco de dados seja criada, otimizando recursos e evitando inconsistências.
    - `@Volatile INSTANCE`: Garante que a variável `INSTANCE` seja sempre lida da memória principal e não de um cache de thread.
    - `synchronized(this)` : Bloco para garantir que a criação do banco de dados seja thread-safe.
    - `fallbackToDestructiveMigration()` : Usado para desenvolvimento, permite que o Room recrie o banco de dados se a versão for alterada. Em produção, isso levaria à perda de dados; migrações incrementais seriam preferidas.

- `BookDao.kt` :Kotlin

```
@Dao
interface BookDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertBook(book: Book)@Update
    suspend fun updateBook(book: Book)@Query("SELECT * FROM books ORDER BY title ASC")
    fun getAllBooks(): Flow<List<Book>> // Observa mudanças na lista de livros

    @Query("SELECT * FROM books WHERE id = :bookId")
    fun getBookById(bookId: Int): Flow<Book?> // Observa um livro específico

    @Query("UPDATE books SET isActive = 0 WHERE id = :bookId")
    suspend fun deactivateBook(bookId: Int)@Query("UPDATE books SET isActive = 1 WHERE id = :bookId")
    suspend fun activateBook(bookId: Int)
}
```

  - `@Dao` : Anotação que marca a interface como um Data Access Object.
  - `@Insert` , `@Update` : Anotações Room para operações básicas de banco de dados. `onConflict = OnConflictStrategy.REPLACE` indica que se um livro com o mesmo ID for inserido, o existente será substituído.
  - `@Query(...)` : Permite definir consultas SQL personalizadas.
    - `getAllBooks()` e `getBookById()` : Retornam `Flow<T>` , o que significa que eles emitem novos valores sempre que os dados subjacentes no banco de dados mudam, permitindo uma UI reativa.
  - `suspend` : Indica que as funções são suspendíveis e podem ser chamadas de uma coroutine.
- `UserDao.kt` :Kotlin

```
@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)@Query("SELECT * FROM users WHERE username = :username LIMIT
1")
    suspend fun getUserByUsername(username: String): User?
}
```

  - `@Insert` : Para inserir novos usuários.
  - `@Query(...)` :
    - `getUserByUsername()` : Busca um usuário pelo nome de usuário. Retorna `User?` (nulo se não encontrado) e é `suspend` porque é uma operação de banco de dados que pode demorar.

5.3. Gerenciador de Favoritos (`FavoriteManager.kt` - Pag 11)

Este componente gerencia a lista de livros favoritos de cada usuário de forma persistente e assíncrona, utilizando **DataStore Preferences**.

- **Propriedades**:
  - `Context.dataStore`: Uma propriedade de extensão que cria uma instância de `DataStore<Preferences>` para armazenar pares chave-valor. O nome "livraria_preferences" é o nome do arquivo subjacente.
- **`addFavoriteBook(userId: String, bookId: Int)`**:
  - Obtém a chave DataStore específica para o ID do usuário (`FAVORITES_KEY_PREFIX + userId`).
  - Usa `dataStore.edit { preferences -> ... }` para realizar uma transação atômica.
  - Recupera o conjunto atual de IDs de favoritos (armazenados como `Set<String>`).
  - Adiciona o `bookId` (convertido para `String`) ao conjunto.
  - Salva o conjunto atualizado de volta nas preferências.
- **`removeFavoriteBook(userId: String, bookId: Int)`**:
  - Similar ao `addFavoriteBook`, mas remove o `bookId` do conjunto existente.
- **`getFavoriteBookIds(userId: String)`**:
  - Obtém a chave DataStore para o usuário.
  - Retorna um `Flow<Set<Int>>` observando as preferências. O `map` transforma o `Set<String>` (armazenado no DataStore) em `Set<Int>` (usado na lógica do aplicativo), e trata o caso de não haver favoritos (retorna um conjunto vazio).

6. Camada de Repositórios

Os repositórios atuam como uma fonte única de verdade para os dados, abstraindo a origem dos dados (banco de dados, rede, etc.) dos ViewModels.

6.1. Repositório de Autenticação (`AuthRepository.kt` - Pag 9)

Gerencia as operações de autenticação e registro de usuários.

- **Construtor**: Recebe uma instância de `UserDao` como dependência.
- **`loginUser(username: String, passwordPlain: String)`**:

    - Busca o `User` pelo `username` no `UserDao`.
    - Se o usuário for encontrado, utiliza `BCrypt.checkpw()` para comparar a `passwordPlain` (senha em texto claro fornecida pelo usuário) com o `passwordHash` (hash armazenado no banco de dados). Esta é uma operação criptográfica segura.
    - Retorna o `User` se a senha for válida, caso contrário, retorna `null`.
- **`registerUser(username: String, passwordPlain: String)`**:
    - Primeiro, verifica se já existe um usuário com o `username` fornecido para evitar duplicidade.
    - Se não existir, gera um hash seguro da `passwordPlain` usando `BCrypt.hashpw()` com um "sal" aleatório (`BCrypt.gensalt()`). O sal é crucial para a segurança, pois impede ataques de "rainbow table".
    - Cria um novo objeto `User` com o nome de usuário, o hash da senha e define o `role` como "user" (padrão para novos registros).
    - Insere o novo usuário no `UserDao`.
    - Retorna `true` se o registro for bem-sucedido, `false` caso contrário (usuário já existe).

6.2. Repositório de Livros (`BookRepository.kt` - Pag 8)

Gerencia as operações relacionadas aos dados dos livros.

- **Construtor**: Recebe uma instância de `BookDao`.
- **`getAllBooks()`**: Delega a chamada para `bookDao.getAllBooks()`, retornando um `Flow<List<Book>>`.
- **`getBookById(bookId: Int)`**: Delega a chamada para `bookDao.getBookById(bookId)`, retornando um `Flow<Book?>`.
- **`insertBook(book: Book)`**: Delega a chamada para `bookDao.insertBook(book)`.
- **`updateBook(book: Book)`**: Delega a chamada para `bookDao.updateBook(book)`.
- **`deactivateBook(bookId: Int)`**: Delega a chamada para `bookDao.deactivateBook(bookId)`.
- **`activateBook(bookId: Int)`**: Delega a chamada para `bookDao.activateBook(bookId)`.

7. Camada de ViewModels

Os ViewModels contêm a lógica de apresentação e gerenciam o estado da UI, expondo dados e ações para as telas via `StateFlow`.

7.1. AuthViewModel (`AuthViewModel.kt` - Pag 20)

Gerencia o estado de autenticação do usuário.

- **`_currentUser: MutableStateFlow<User?>`**: Um `MutableStateFlow` privado que armazena o `User` atualmente logado. É exposto como `currentUser: StateFlow<User?>` para a UI, permitindo que ela observe as mudanças no estado de autenticação.
- **`_loginError: MutableStateFlow<String?>`**: Armazena mensagens de erro de login, exposto como `loginError`.
- **`_registrationError: MutableStateFlow<String?>`**: Armazena mensagens de erro de registro, exposto como `registrationError`.
- **`login(username: String, passwordPlain: String, onLoginSuccess: () -> Unit)`**:
  - Limpa erros de login anteriores.
  - Inicia uma coroutine no `viewModelScope` (associado ao ciclo de vida do ViewModel).
  - Chama `authRepository.loginUser()`.
  - Se o login for bem-sucedido (`user != null`), atualiza `_currentUser` e executa o callback `onLoginSuccess`.
  - Se falhar, define a mensagem em `_loginError`.
- **`register(username: String, passwordPlain: String, onRegistrationSuccess: () -> Unit)`**:
  - Limpa erros de registro anteriores.
  - Valida a entrada (`username` e `passwordPlain` não podem ser vazios, `passwordPlain` com no mínimo 6 caracteres). Se a validação falhar, define `_registrationError` e retorna.
  - Chama `authRepository.registerUser()`.
  - Se o registro for bem-sucedido, chama o próprio `login()` para logar o usuário recém-registrado e, em seguida, executa `onRegistrationSuccess`.
  - Se o registro falhar (ex: nome de usuário já existe), define `_registrationError`.
- **`logout()`**: Simplesmente define `_currentUser.value = null`, deslogando o usuário.

- `clearLoginError()` / `clearRegistrationError()`: Métodos públicos para limpar as mensagens de erro, geralmente chamados após a exibição do erro na UI.

7.2. AuthViewModelFactory (`AuthViewModelFactory.kt` - Pag 21)

Um `ViewModelProvider.Factory` customizado para `AuthViewModel`.

- `create(modelClass: Class<T>)`: Este método é invocado pelo sistema `viewModel()` do Compose quando um `AuthViewModel` é solicitado. Ele verifica se a `modelClass` é `AuthViewModel` e, se for, retorna uma nova instância de `AuthViewModel`, injetando o `AuthRepository` fornecido no construtor do factory. Isso é uma forma manual de injeção de dependência para ViewModels.

7.3. BookViewModel (`BookViewModel.kt` - Pag 22)

Gerencia o estado e as operações relacionadas aos livros, incluindo pesquisa e favoritos.

- **Construtor**: Recebe `bookRepository`, `favoriteManager` e, **criticamente**, `currentUserFlow: StateFlow<User?>` (o `currentUser` do `AuthViewModel`). Isso permite que `BookViewModel` reaja a mudanças no usuário logado.
- `_searchQuery: MutableStateFlow<String>`: Mantém o termo de busca atual, que é exposto como `searchQuery`.
- `_allBooks`: Um `Flow` que observa todos os livros diretamente do `bookRepository.getAllBooks()`.
- `favoriteBookIds: StateFlow<Set<Int>>`:
  - **Mecanismo:** Utiliza `currentUserFlow.flatMapLatest { ... }`.
  - **Lógica:** Quando `currentUserFlow` emite um novo usuário (ou nulo), `flatMapLatest` cancela a coroutine anterior e inicia uma nova.
    - Se `currentUser` **não for nulo**: Ele chama `favoriteManager.getFavoriteBookIds(currentUser.id.toString())` para obter os favoritos *desse usuário específico*.
    - Se `currentUser` **for nulo**: Ele emite um `flowOf(emptySet())`, indicando que não há favoritos para um usuário não logado.
  - `stateIn(...)`: Converte o `Flow` resultante em um `StateFlow`, garantindo que os dados sejam compartilhados e sobrevivam ao ciclo de vida.
- `books: StateFlow<List<Book>>`:

    - **Mecanismo:** Utiliza `combine(_allBooks, _searchQuery, currentUserFlow) { ... }`.
    - **Lógica:** Sempre que `_allBooks`, `_searchQuery` ou `currentUserFlow` emitem um novo valor, a função lambda é executada.
        - Primeiro, filtra `allBooks` com base em `_searchQuery` (título ou autor, ignorando maiúsculas/minúsculas).
        - Em seguida, verifica se o `currentUser` é um administrador (`isAdmin`).
        - Se `isAdmin` for `true`, todos os livros filtrados pela busca são retornados.
        - Se `isAdmin` for `false` (usuário comum), os livros são *adicionalmente* filtrados para incluir apenas aqueles onde `it.isActive` é `true`, garantindo que livros desativados não sejam exibidos para usuários comuns.
    - `stateIn(...)`: Converte o `Flow` combinado em um `StateFlow`.
- `favoriteBooks: StateFlow<List<Book>>`:
    - **Mecanismo:** Utiliza `combine(_allBooks, favoriteBookIds, currentUserFlow) { ... }`.
    - **Lógica:** Similar a `books`, mas focado nos favoritos.
        - Filtra `allBooks` para incluir apenas aqueles cujo `id` está presente em `favoriteBookIds` (que já reflete os favoritos do usuário atual).
        - Aplica o filtro `isActive` para usuários não-administradores, garantindo que mesmo um livro favorito desativado não seja exibido para usuários comuns.
    - `stateIn(...)`: Converte em `StateFlow`.
- `getBookById(bookId: Int)`: Apenas delega ao repositório.
- `addBook(book: Book)` / `updateBook(book: Book)` / `deactivateBook(bookId: Int)` / `activateBook(bookId: Int)`: Envolve as chamadas do repositório em `viewModelScope.launch { ... }` para execução assíncrona.
- `setSearchQuery(query: String)`: Atualiza o valor de `_searchQuery`, o que automaticamente re-emite o `books` `StateFlow` devido ao `combine` e atualiza a UI.
- `toggleFavorite(bookId: Int)`:
    - Inicia uma coroutine.

- Obtém o `currentUser` usando `currentUserFlow.firstOrNull()`.
- **Crucial:** Se um usuário estiver logado, ele recupera o `currentFavorites` *do* `favoriteBookIds` `StateFlow` (que já é específico do usuário).
- Chama `favoriteManager.removeFavoriteBook()` ou `favoriteManager.addFavoriteBook()` com o `userId` e `bookId` apropriados. Se o usuário não estiver logado, uma mensagem de erro pode ser exibida (comentada no código).

7.4. BookViewModelFactory (`BookViewModelFactory.kt` - Pag 23)

Factory customizado para `BookViewModel`.

- `create(modelClass: Class<T>)`: Fornece as dependências (`bookRepository`, `favoriteManager`, e `currentUserFlow`) ao construtor do `BookViewModel` quando uma instância é solicitada. A passagem de `currentUserFlow` (do `AuthViewModel`) é um exemplo de injeção de dependência de `StateFlow` entre ViewModels.

8. Camada de Interface do Usuário (UI)

As telas são construídas com Jetpack Compose e observam os `StateFlow`s dos ViewModels para exibir dados e reagir a eventos do usuário.

8.1. Adicionar Livro (`AddBookScreen.kt` - Pag 12)

- **Componível**: `@Composable fun AddBookScreen(...)`
- **Funcionalidade**: Permite que usuários administradores adicionem novos livros ao catálogo.
- **Elementos da UI**:
  - `Scaffold` com `TopAppBar` (título "Adicionar Novo Livro" e botão de "Voltar").
  - `Column` centralizada contendo `OutlinedTextField`s para:
    - `title` (Título)
    - `author` (Autor)
    - `description` (Descrição)
    - `price` (Preço)
    - `quantity` (Quantidade)
    - `imageUrl` (URL da Imagem)

- `Button("Adicionar Livro")`: Ao clicar, cria um objeto `Book` a partir dos inputs e chama `bookViewModel.addBook()`. Em seguida, navega de volta.

- **Observações**: Campos `price` e `quantity` deveriam ter validação de entrada para garantir que sejam números válidos.

8.2. Card de Livro (`BookCard.kt` - Pag 10)

Um componente reutilizável para exibir informações concisas de um livro em listas.

- **Componível**: `@Composable fun BookCard(...)`
- **Parâmetros**: Recebe um objeto `Book`, callbacks `onBookClick` e `onToggleFavorite`, e um booleano `isFavorite` para controlar a cor do ícone de favorito.
- **Layout**: Utiliza `Card` do Material3 com um `Row` para organizar imagem e texto.
  - **Imagem**: `AsyncImage(model = book.imageUrl, ...)` do Coil, exibe a imagem do livro. Inclui um `contentDescription` para acessibilidade.
  - **Informações do Livro**: `Column` para `title`, `author`, `price`, `quantity`. O preço é formatado para moeda.
  - **Ícone de Favorito**: `IconButton` com `Icons.Filled.Favorite`. A cor do ícone (`tint`) muda para `MaterialTheme.colorScheme.error` se `isFavorite` for `true`, indicando que está marcado como favorito. O `onClick` chama `onToggleFavorite`.
  - **Status Inativo**: Um `Text` adicional "Inativo" é exibido com um fundo vermelho opaco se `book.isActive` for `false`, alertando o usuário sobre o status do livro.
- **Interatividade**:
  - O `Card` inteiro tem um `clickable` que chama `onBookClick(book.id)`, permitindo a navegação para os detalhes do livro.
  - O `IconButton` permite alternar o status de favorito.

8.3. Tela de Catálogo (`CatalogScreen.kt` - Pag 13)

Exibe a lista principal de livros, permitindo rolagem e interação.

- **Componível**: `@Composable fun CatalogScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `onBookClick` e `paddingValues`.

- **Coleta de Dados**:
  - `val books by bookViewModel.books.collectAsState(initial = emptyList())`: Observa o `StateFlow` `books` do `BookViewModel`, que já contém a lógica de filtragem por busca e por status `isActive` (para não-administradores).
  - `val favoriteBookIds by bookViewModel.favoriteBookIds.collectAsState(initial = emptySet())`: Observa os IDs dos livros favoritos do usuário atual.
- **Layout**:
  - `if (books.isEmpty())`: Exibe uma mensagem "Nenhum livro encontrado." se a lista estiver vazia.
  - `LazyColumn`: Componente de Compose para listas roláveis de alta performance, que só renderiza os itens visíveis.
    - `contentPadding` e `verticalArrangement`: Adicionam espaçamento ao redor e entre os itens.
    - `items(books) { book -> ... }`: Itera sobre a lista de `books`.
      - Para cada `book`, renderiza um `BookCard`, passando o `book`, o `onBookClick`, o `onToggleFavorite` (que chama `bookViewModel.toggleFavorite`), e `isFavorite = book.id in favoriteBookIds`.

8.4. Tela de Detalhes/Edição de Livro (`EditBookScreen.kt` - Pag 14)

Permite visualizar os detalhes de um livro e, para administradores, editar suas propriedades.

- **Componível**: `@Composable fun EditBookScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `authViewModel`, `bookId` (do Navigation) e `onBack`.
- **Coleta de Dados**:
  - `val book by bookViewModel.getBookById(bookId).collectAsState(initial = null)`: Observa o livro específico do `BookViewModel`.
  - `val currentUser by authViewModel.currentUser.collectAsState()`: Observa o usuário logado para determinar permissões.
  - `val isAdmin = currentUser?.role == "admin"`: Calcula se o usuário é administrador.

- **Estado Local**: Variáveis `mutableStateOf` para armazenar o estado editável dos campos do livro (título, autor, etc.). `LaunchedEffect(book)` é usado para preencher esses estados quando o `book` é carregado ou muda.
- `Scaffold` e `TopAppBar`:
  - Título "Detalhes do Livro" ou "Editar Livro" (para admin).
  - Botão "Voltar" (`ArrowBack`).
  - **Ação para Administrador**: Um `IconButton` para salvar (`Save`) é visível apenas se `isAdmin` for `true`.
  - **Toggle Ativar/Desativar**: Um `IconButton` com ícones `PowerSettingsNew` (ativo) ou `PowerOff` (inativo) é visível apenas para administradores. O `onClick` alterna o status `isActive` do livro chamando `bookViewModel.activateBook()` ou `deactivateBook()`. A cor do ícone (`tint`) muda com o status.
- `Column` **Principal**: Exibe os `OutlinedTextField`s para os detalhes do livro.
  - `readOnly = !isAdmin`: Os campos de texto são `readOnly` (somente leitura) se o usuário não for um administrador, impedindo edições não autorizadas.
  - Exibe `AsyncImage` para a capa do livro.
- **Toast para Sucesso**: Um `LaunchedEffect(saveSuccess)` pode ser usado para exibir um `Toast` quando a atualização é bem-sucedida.

8.5. Tela de Livros Favoritos (`FavoriteBooksScreen.kt` - Pag 15)

Exibe a lista de livros que o usuário logado marcou como favoritos.

- **Componível**: `@Composable fun FavoriteBooksScreen(...)`
- **Parâmetros**: Recebe `bookViewModel`, `onBookClick`, `onBack`.
- **Coleta de Dados**:
  - `val favoriteBooks by bookViewModel.favoriteBooks.collectAsState(initial = emptyList())`: Observa o `StateFlow` `favoriteBooks` do `BookViewModel`, que já está filtrado para mostrar apenas os favoritos do usuário logado e aplicando o filtro `isActive` para usuários comuns.
- **Layout**:
  - `Scaffold` com `TopAppBar` (título "Meus Favoritos" e botão "Voltar").
  - `Column` principal.

- `if (favoriteBooks.isEmpty())`: Se a lista de favoritos estiver vazia, exibe uma mensagem informativa.
- `LazyColumn`: Exibe a lista de `favoriteBooks`.
  - Cada `book` é renderizado com um `BookCard`.
  - `isFavorite = true` é passado, pois todos os livros nesta tela *são* favoritos.
  - `onToggleFavorite` permite que o usuário desmarque um favorito diretamente desta tela.

8.6. Tela Principal do Aplicativo (`HomeScreen.kt` - Pag 16)

A tela principal que o usuário vê após o login, agindo como um painel de controle.

- **Componível**: `@Composable fun HomeScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `bookViewModel`, callbacks para navegação (`onNavigateToAddBook`, `onBookClick`, `onLogout`, `onNavigateToFavoriteBooks`).
- **Coleta de Dados**:
  - `val currentUser by authViewModel.currentUser.collectAsState()`: Observa o usuário atual para determinar o papel.
  - `val isAdmin = currentUser?.role == "admin"`: Define a permissão administrativa.
  - `val searchQuery by bookViewModel.searchQuery.collectAsState()`: Observa o termo de busca atual.
- `Scaffold`: Estrutura a tela.
  - `topBar` **(TopAppBar)**:
    - Título "Livraria App".
    - `actions` **(Ícones na barra superior)**:
      - `IconButton(onClick = onNavigateToFavoriteBooks)`: Ícone de **Favoritos** (`Icons.Filled.Favorite`).
      - `if (isAdmin) { IconButton(onClick = onNavigateToAddBook) { Icon(Icons.Filled.Add, ...) } }`: Ícone de **Adicionar Livro** (`Icons.Filled.Add`) **visível apenas para administradores**.
      - `IconButton(onClick = onLogout) { Icon(Icons.Filled.ExitToApp, ...) }`: Ícone de **Sair** (`Icons.Filled.ExitToApp`) para deslogar.

        - Cores personalizadas para a `TopAppBar`.
    - **Remoção de `bottomBar` e `floatingActionButton`**: Essas seções foram removidas do `Scaffold`, simplificando a navegação e consolidando as ações principais na `TopAppBar`.
- `Column` **Principal**:
    - `SearchBar`: Componente reutilizável para a funcionalidade de busca.
        - `query = searchQuery`, `onQueryChange = { bookViewModel.setSearchQuery(it) }`: Vincula a entrada da barra de busca ao `searchQuery` do `BookViewModel`, acionando a filtragem.
    - `CatalogScreen`: Exibe o catálogo de livros, recebendo o `bookViewModel` e os callbacks de clique.

8.7. Tela de Login (`LoginScreen.kt` - Pag 17)

A primeira tela que os usuários veem para acessar o aplicativo.

- **Componível**: `@Composable fun LoginScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `onLoginSuccess`, `onNavigateToRegister`.
- **Estado Local**: `username` e `password` são gerenciados por `mutableStateOf`.
- **Tratamento de Erro**:
    - `val loginError by authViewModel.loginError.collectAsState()`: Observa o `loginError` do ViewModel.
    - `LaunchedEffect(loginError)`: Um efeito lateral que é disparado sempre que `loginError` muda.
        - `loginError?.let { ... }`: Se houver um erro, exibe um `Toast` com a mensagem de erro e então chama `authViewModel.clearLoginError()` para limpar o erro no ViewModel, evitando que o Toast apareça novamente após uma recriação da Composable.
- **Layout**:
    - `Column` centralizada com `Arrangement.Center` e `Alignment.CenterHorizontally`.
    - `Text("Bem-vindo à Livraria!")` com estilo de título.
    - `OutlinedTextField`s para "Nome de Usuário" e "Senha". A senha usa `PasswordVisualTransformation()` para ocultar os caracteres.

    - `Button("Entrar")`: `onClick` chama `authViewModel.login(username, password, onLoginSuccess)`.
    - `TextButton("Não tem uma conta? Cadastre-se")`: `onClick` chama `onNavigateToRegister`.
- **Preview**: `@Preview` para visualização rápida da tela no Android Studio.

8.8. Tela de Registro (`RegisterScreen.kt` - Pag 18)

Permite que novos usuários criem uma conta.

- **Componível**: `@Composable fun RegisterScreen(...)`
- **Parâmetros**: Recebe `authViewModel`, `onRegistrationSuccess`, `onBack`.
- **Estado Local**: `username`, `password`, `confirmPassword`.
- **Tratamento de Erro**: Similar ao `LoginScreen`, usando `LaunchedEffect` para observar `registrationError` e exibir `Toast`s.
- **Layout**:
    - `Scaffold` com `TopAppBar` (título "Criar Nova Conta" e botão "Voltar").
    - `Column` centralizada com `OutlinedTextField`s para nome de usuário, senha e confirmação de senha (ambas com `PasswordVisualTransformation`).
    - `Button("Registrar")`:
        - `onClick` contém uma validação local: `if (password == confirmPassword)`.
        - Se as senhas coincidirem, chama `authViewModel.register(...)`.
        - Se não coincidirem, exibe um `Toast` de "As senhas não coincidem!".
- **Preview**: `@Preview` para visualização.

8.9. Barra de Busca (`SearchBar.kt` - Pag 19)

Um componente de UI genérico para funcionalidade de busca.

- **Componível**: `@Composable fun SearchBar(...)`
- **Parâmetros**: `query` (o termo de busca atual), `onQueryChange` (callback para quando o termo de busca muda), `modifier`.
- **Layout**:
    - `OutlinedTextField`: Um campo de texto com bordas e rótulo.
        - `value = query`, `onValueChange = onQueryChange`: Vincula o valor do campo ao `query` e chama o callback `onQueryChange` quando o texto é modificado.

        - `placeholder`: Texto de dica "Buscar livros por título ou autor...".
        - `leadingIcon`: Exibe um ícone de busca (`Icons.Default.Search`).
        - `trailingIcon`: Exibe um ícone de "fechar" (`Icons.Default.Close`) **apenas se a `query` não estiver vazia**, permitindo limpar a busca rapidamente.
        - `singleLine = true`: Garante que o campo de texto seja uma única linha.
        - `colors`: Personaliza as cores da barra de busca usando o tema Material3.

9. Sistema de Navegação (`AppNavHost.kt`)

Gerencia as transições entre as diferentes telas (Composables) do aplicativo.

- **Componível**: `@Composable fun AppNavHost(...)`
- **Parâmetros**: Recebe `navController`, `authViewModel`, `bookViewModel`.
- `NavHost`: O principal componente de navegação, que define o grafo de navegação.
    - `navController`: A instância do controlador de navegação.
    - `startDestination = "login"`: A tela inicial do aplicativo.
    - **Definição das Rotas**:
        - `composable("login") { ... }`: Define a rota de login.
            - Chama `LoginScreen`, passando `authViewModel` e callbacks para navegação (e.g., `onLoginSuccess = { navController.navigate("home") { ... } }`).
        - `composable("register") { ... }`: Define a rota de registro.
            - Chama `RegisterScreen`, passando `authViewModel` e callbacks para navegação (`onRegistrationSuccess = { navController.popBackStack() }`, `onBack = { navController.popBackStack() }`).
        - `composable("home") { ... }`: Define a rota da tela principal.
            - Chama `HomeScreen`, passando `authViewModel`, `bookViewModel` e todos os callbacks necessários para navegação para outras telas (adicionar livro, favoritos, logout, detalhes do livro).
        - `composable("addBook") { ... }`: Define a rota para adicionar livro.
        - `composable("editBook/{bookId}", arguments = listOf(navArgument("bookId") { type = NavType.IntType })) { backStackEntry -> ... }`: Define a rota para editar livro.

- **Argumentos de Navegação**: `arguments = listOf(navArgument("bookId") { ... })` especifica que esta rota espera um argumento inteiro chamado `bookId`.
- `val bookId = backStackEntry.arguments?.getInt("bookId")`: Extrai o `bookId` dos argumentos da `backStackEntry`.
- Chama `EditBookScreen`, passando o `bookId` extraído.

- `composable("favoriteBooks") { ... }`: Define a rota para os livros favoritos.

10. Fluxos de Interação Exemplo

10.1. Fluxo de Autenticação (Login)

1. **Usuário abre o App**: `MainActivity` inicializa e `AppNavHost` exibe `LoginScreen`.
2. **Usuário insere credenciais**: Digita `username` e `password` nos `OutlinedTextField`s da `LoginScreen` (Pag 17).
3. **Clica em "Entrar"**:
   - `LoginScreen` chama `authViewModel.login(username, password, onLoginSuccess)`.
   - `AuthViewModel` (Pag 20) chama `authRepository.loginUser()`.
   - `AuthRepository` (Pag 9) busca o usuário no `UserDao` e verifica a senha com BCrypt.
   - **Sucesso**: `AuthViewModel` atualiza `_currentUser`, e o callback `onLoginSuccess` é executado.
     - `AppNavHost` (`onLoginSuccess`) navega para a rota "home" (`HomeScreen`).
   - **Falha**: `AuthViewModel` define `_loginError`.
     - `LoginScreen` observa `loginError` (`LaunchedEffect`) e exibe um `Toast` com a mensagem de erro.

10.2. Fluxo de Registro

1. **Usuário está na `LoginScreen`**: Clica em "Não tem uma conta? Cadastre-se" (Pag 17).
2. **Navegação**: `LoginScreen` chama `onNavigateToRegister`, e `AppNavHost` navega para `RegisterScreen`.

3. **Usuário preenche dados**: Insere `username`, `password`, `confirmPassword` na `RegisterScreen` (Pag 18).
4. **Clica em "Registrar"**:
    - `RegisterScreen` valida se `password` e `confirmPassword` são iguais.
    - Se iguais, chama `authViewModel.register(username, password, onRegistrationSuccess)`.
    - `AuthViewModel` (Pag 20) realiza validações internas (vazio, comprimento da senha).
    - `AuthViewModel` chama `authRepository.registerUser()`.
    - `AuthRepository` (Pag 9) verifica se o nome de usuário já existe, gera hash da senha e insere no `UserDao`.
    - **Sucesso**: `AuthViewModel` chama `login()` automaticamente. Após o login, `onRegistrationSuccess` é executado.
        - `AppNavHost` (`onRegistrationSuccess`) faz `navController.popBackStack()` (retorna para a tela de login, que imediatamente redireciona para Home se o login automático funcionar) ou pode ser adaptado para ir direto para Home.
    - **Falha**: `AuthViewModel` define `_registrationError`.
        - `RegisterScreen` observa `registrationError` e exibe um `Toast`.

10.3. Fluxo de Gerenciamento de Livros (Admin)

1. **Admin logado na `HomeScreen`**: Ícone "Adicionar Livro" está visível (Pag 16).
2. **Admin clica em "Adicionar Livro"**:
    - `HomeScreen` chama `onNavigateToAddBook`, e `AppNavHost` navega para `AddBookScreen`.
3. **Admin preenche detalhes e clica "Adicionar Livro"**:
    - `AddBookScreen` (Pag 12) cria um objeto `Book` e chama `bookViewModel.addBook(book)`.
    - `BookViewModel` (Pag 22) chama `bookRepository.insertBook(book)`.
    - `BookRepository` (Pag 8) insere o livro no `BookDao`.
    - O `_allBooks` `Flow` no `BookViewModel` é automaticamente atualizado, e `books` `StateFlow` é re-emitido, atualizando a `CatalogScreen` em `HomeScreen`.

4. **Admin clica em um `BookCard`:**
    - `BookCard` (Pag 10) chama `onBookClick(bookId)`.
    - `AppNavHost` navega para `editBook/{bookId}`, passando o `bookId`.
5. **Admin edita detalhes ou status `isActive`:**
    - `EditBookScreen` (Pag 14) exibe os detalhes do livro.
    - Admin modifica campos ou clica no ícone de ativar/desativar.
    - **Salvar**: Clica no ícone de salvar na `TopAppBar`. `EditBookScreen` chama `bookViewModel.updateBook(editedBook)` (se for edição de detalhes) ou `bookViewModel.activateBook()` / `deactivateBook()` (se for toggle de status).
    - `BookViewModel` chama o método correspondente no `BookRepository`, que atualiza o `BookDao`.
    - A `CatalogScreen` (Pag 13) e a `FavoriteBooksScreen` (Pag 15) automaticamente se atualizam devido à observação dos `Flow`s.

10.4. Fluxo de Favoritos (Usuário Comum)

1. **Usuário comum logado na `HomeScreen`**: Vê o catálogo de livros na `CatalogScreen`.
2. **Usuário clica no ícone de coração em um `BookCard`:**
    - `BookCard` (Pag 10) chama `onToggleFavorite(book.id)`.
    - `BookViewModel` (Pag 22) observa `toggleFavorite(bookId)`.
    - `BookViewModel` verifica o `currentUser` e o `favoriteBookIds` atual, então chama `favoriteManager.addFavoriteBook()` ou `removeFavoriteBook()`.
    - `FavoriteManager` (Pag 11) atualiza o `DataStore`.
    - O `favoriteBookIds` `StateFlow` no `BookViewModel` é re-emitido, fazendo com que o `BookCard` na `CatalogScreen` atualize sua cor de coração, e o `favoriteBooks` `StateFlow` (usado na `FavoriteBooksScreen`) também é atualizado.
3. **Usuário clica no ícone de Favoritos na `TopAppBar`:**
    - `HomeScreen` (Pag 16) chama `onNavigateToFavoriteBooks`.
    - `AppNavHost` navega para `FavoriteBooksScreen`.
4. **`FavoriteBooksScreen` exibe livros favoritos**:

- A tela observa `bookViewModel.favoriteBooks` e exibe apenas os livros que o usuário marcou como favoritos.

11. Tecnologias Utilizadas (Aprofundamento)

- **Jetpack Compose**: Sistema de UI moderno e declarativo para Android.
  - **Benefícios**: Reduz boilerplate, melhora a legibilidade, e simplifica o desenvolvimento de UIs complexas.
- **Room Persistence Library**: Camada de abstração sobre SQLite.
  - **Benefícios**: Objeto-relacional mapping (ORM), verificação de SQL em tempo de compilação, integração com `Flow` para reatividade, minimiza código repetitivo.
- **Kotlin Coroutines & Flow**:
  - **Coroutines**: Para operações assíncronas e não bloqueantes, como acesso a banco de dados e hashing de senha.
  - **Flow**: Stream assíncrono de dados que pode emitir múltiplos valores ao longo do tempo. Ideal para observar mudanças no banco de dados (`BookDao`, `UserDao`) e preferências de usuário (`FavoriteManager`), permitindo que a UI se atualize automaticamente.
- **Jetpack Navigation Compose**: Gerencia a navegação entre as telas.
  - **Benefícios**: Define o grafo de navegação, passa argumentos de forma segura, gerencia a pilha de backstack e simplifica as transições.
- **Jetpack ViewModel**:
  - **Benefícios**: Mantém o estado da UI de forma otimizada para o ciclo de vida, sobrevivendo a mudanças de configuração (rotações de tela), e facilita a separação de preocupações (lógica da UI separada da View).
- **DataStore Preferences**:
  - **Benefícios**: Alternativa moderna ao `SharedPreferences`, opera de forma assíncrona com `Flow` (Kotlin Coroutines) e garante consistência transacional para pequenas quantidades de dados.
- **BCrypt (JBCrypt)**: Algoritmo de hashing de senha.
  - **Benefícios**: Forte contra ataques de força bruta devido ao seu fator de trabalho (iterações configuráveis), e incorpora um "sal" para proteger

contra ataques de rainbow table. Essencial para segurança.

- **Coil**: Biblioteca de carregamento de imagens.
  - **Benefícios**: Rápida, leve, moderna, otimizada para Compose (`AsyncImage`), e oferece funcionalidades como cache de memória e disco, transformações de imagem e placeholders.

12. Próximos Passos e Oportunidades de Melhoria

Ainda que funcional, o LivrariaAppV2 pode ser aprimorado em diversas áreas para um ambiente de produção.

- **Validação de Entrada Robusta**:
  - Implementar validação em tempo real para todos os campos de entrada (ex: senhas com caracteres especiais, nomes de usuário sem espaços, URLs de imagem válidas, preço/quantidade como números positivos).
  - Fornecer feedback visual claro para o usuário sobre campos inválidos, não apenas `Toast`s.
- **Tratamento de Estado de Carregamento e Erros (UI)**:
  - Exibir indicadores de carregamento (`CircularProgressIndicator`) para operações de longa duração (login, registro, busca de dados).
  - Utilizar `Snackbar` para mensagens de erro/sucesso temporárias que podem ser dispensadas.
  - Implementar telas de erro mais amigáveis com opções de "Tentar Novamente" para falhas de rede ou banco de dados.
- **Persistência de Sessão do Usuário**:
  - Atualmente, o usuário é deslogado ao fechar o aplicativo. Para manter o login, o `AuthViewModel` precisaria salvar o ID ou um token do usuário no `DataStore` (ou `SharedPreferences criptografados`) e carregá-lo na inicialização do aplicativo.
- **Função "Esqueci a Senha"**:
  - Implementar um fluxo para redefinição de senha (em um aplicativo real, isso envolveria comunicação com um backend).
- **Refatoração do Login Automático Após Registro**:

    - Embora conveniente para o desenvolvimento, em um aplicativo real, após o registro, o ideal seria que o usuário fosse redirecionado para a tela de login para um login explícito.
- **Internacionalização (i18n)**:
    - Externalizar todas as strings da UI para arquivos de recursos (`strings.xml`) para facilitar a tradução do aplicativo para diferentes idiomas.
- **Acessibilidade (a11y)**:
    - Revisar e garantir que todos os elementos da UI tenham `contentDescription` s adequadas, permitindo que usuários com deficiência visual possam usar o aplicativo com leitores de tela.
- **Segurança Adicional**:
    - Considerar o uso de `AndroidKeystore` para armazenar chaves criptográficas, se informações mais sensíveis além das senhas forem persistidas localmente.
    - Para um ambiente de produção, o "sal" para o BCrypt deveria ser gerado e gerenciado de forma mais robusta.
- **Lógica de Migração de Banco de Dados**:
    - Substituir `fallbackToDestructiveMigration()` no `AppDatabase` por migrações incrementais (`Room.addMigrations()`) para preservar os dados dos usuários em futuras atualizações do aplicativo.
- **Gerenciamento de Imagens**:
    - Permitir que o usuário envie imagens (em vez de apenas URLs), o que envolveria permissões de armazenamento e upload para um serviço de armazenamento em nuvem.
- **Permissões de Usuário (Admin)**:
    - Uma tela de gerenciamento de usuários para administradores, permitindo-lhes alterar papéis de usuário ou desativar contas.