O BookApp é um aplicativo desenvolvido com o objetivo de facilitar a busca, visualização e organização de livros favoritos de forma simples e intuitiva. A ideia central foi criar uma experiência fluida para o usuário, que pudesse explorar títulos diretamente da API do Google Books, acessar detalhes das obras e marcar aquelas que deseja manter salvas. Durante o desenvolvimento, buscou-se aplicar boas práticas de programação, como organização clara do código, uso de padrões de arquitetura e design, testes e separação de responsabilidades para tornar o projeto mais legível e de fácil manutenção.

Desde o início, o projeto foi dividido em três camadas principais: presentation, domain e data. Essa separação favorece a manutenção do código, evita acoplamento entre responsabilidades e facilita os testes unitários. A camada presentation é composta pelas telas do aplicativo (screens) e pelos componentes de interface criados com Jetpack Compose. Essa camada se comunica com ViewModels que, por sua vez, são responsáveis por gerenciar o estado da interface e acessar os casos de uso definidos na camada domain. Cada ViewModel expõe seus dados por meio de `StateFlow`, que é observado nas Composables para atualização reativa da UI.

A arquitetura adotada foi o padrão MVVM (Model-View-ViewModel). A ViewModel atua como intermediária entre a interface do usuário e as operações de negócio, garantindo que a UI permaneça desacoplada da lógica de aplicação. O modelo de dados está representado na camada domain, com uso de classes simples (data classes) e interfaces que representam os contratos dos repositórios. As implementações concretas desses repositórios estão localizadas na camada data, onde são feitas as chamadas HTTP via Retrofit, bem como o acesso ao banco de dados local via Room.

A injeção de dependência foi implementada usando o Hilt, o que permitiu uma construção de objetos automática, reduzindo o acoplamento e facilitando a testabilidade do sistema. Todas as dependências relevantes foram definidas em um módulo de DI, `AppModule`, anotado com `@Module` e `@InstallIn(SingletonComponent::class)`. Ali foram fornecidas instâncias do Retrofit, do Room Database, das interfaces de repositório e dos casos de uso, todos anotados com `@Provides` e `@Singleton` quando apropriado. ViewModels também são injetadas automaticamente através de `@HiltViewModel`, utilizando construtores com dependências.

Para garantir a qualidade do código, foram implementados testes unitários utilizando `junit`, `mockk` e `kotlinx-coroutines-test`. Foram testados os principais casos de uso da aplicação, como favoritar e desfavoritar livros, verificar se um livro é favorito, buscar livros por título e recuperar livros por ID. Além disso, testes de ViewModel foram escritos com uso de regras como `InstantTaskExecutorRule` para garantir a execução correta em ambiente de teste. As dependências foram mockadas quando necessário, permitindo a verificação de comportamentos isolados.

Diversos padrões de projeto foram aplicados ao longo da implementação. O padrão Repository permitiu isolar a origem dos dados, de forma que a camada de apresentação não precisa conhecer se os dados vêm da API ou do banco local. O padrão UseCase foi utilizado para encapsular regras de negócio em classes autônomas, facilitando o reuso e a testabilidade. O ViewModel, como padrão de gestão de estado, assegura que os dados da

UI sejam persistentes a mudanças de configuração. A injeção de dependência com Hilt também representa um padrão essencial para desacoplamento entre módulos.

A interface do aplicativo foi construída com Jetpack Compose e conta com três telas funcionais principais: a tela de lista de livros, onde o usuário pode pesquisar livros por título e visualizar resultados; a tela de detalhes do livro, que exibe informações completas sobre o livro selecionado (capa, título, autores, descrição, botão de favoritos); e a tela de favoritos, que apresenta todos os livros salvos localmente. As imagens dos livros são carregadas com a biblioteca Coil, e a navegação entre telas é gerenciada com `NavHost`, incluindo passagem de argumentos como `bookId`.

Além disso, o projeto conta com persistência local dos favoritos utilizando o Room, com entidades, DAOs e repositório local configurados corretamente. Todo o fluxo de dados é assíncrono, utilizando coroutines e flow para garantir responsividade e reatividade na interface.

Durante o desenvolvimento, enfrentamos desafios como a integração correta da API do Google Books, que por vezes retornava dados incompletos ou inconsistentes, exigindo tratamento cuidadoso no parsing das respostas JSON. Também houve a necessidade de adaptação de URLs de imagens, substituindo `http` por `https` para garantir carregamento adequado via Coil. A navegação entre telas exigiu atenção especial para manipulação do backstack e passagem segura de parâmetros.

Outra etapa importante foi a implementação progressiva dos casos de uso, ajustando a ViewModel e os estados observáveis conforme cada funcionalidade era construída. A decisão de utilizar um banco de dados local veio da necessidade de manter livros favoritos mesmo após o fechamento do aplicativo, reforçando o uso prático de Room. Cada repositório foi cuidadosamente testado em isolamento, o que facilitou a depuração e garantiu comportamento previsível.

O processo de organização dos pacotes e arquivos refletiu a arquitetura limpa, com cada componente posicionado de acordo com sua responsabilidade. A modularização foi pensada para permitir expansão futura — como, por exemplo, filtros por categoria ou recursos offline mais avançados.

Por fim, a escrita dos testes exigiu o uso de recursos como `runBlockingTest`, `advanceUntilIdle` e o uso de `FakeRepository` para simular comportamentos esperados.