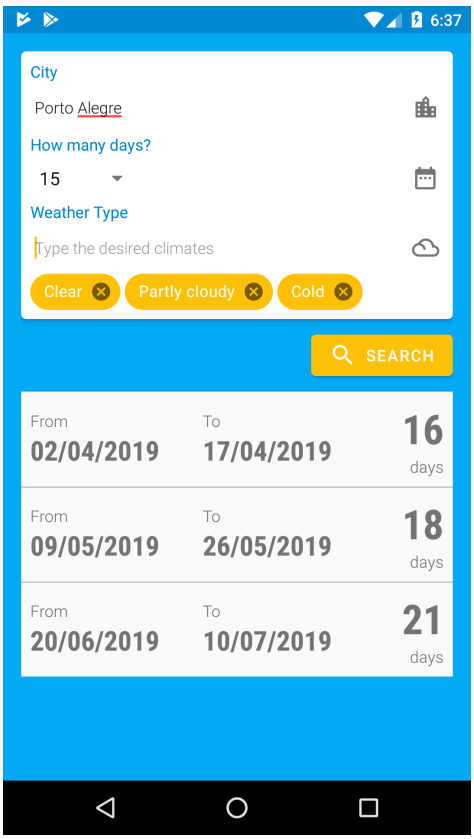
Vacation Planner

Autor: Nelson Glauber (nglaubervasc@gmail.com)

O **Vacation Planner** tem o objetivo de ajudar a planejar as férias do usuário. O aplicativo pesquisa na internet quais os melhores períodos do ano para aproveitar melhor a cidade de destino baseando-se na quantidade de dias e dos tipos de climas especificados.
Por exemplo, digamos que o usuário queira tirar quinze dias de férias a qualquer momento do próximo ano e está planejando visitar o Rio de Janeiro. A aplicação é capaz de obter todos os intervalos de datas maiores ou iguais ao número de dias de férias inseridos que satisfaçam os tipos de clima especificados.

Esse documento apresenta a uma visão geral da solução proposta para esse aplicativo. O projeto foi desenvolvido utilizando a linguagem Kotlin (1.3.21) no Android Studio 3.3.1. A figura a seguir mostra o Vacation Planner em execução.



Vacation Planner em execução

Como executar o projeto

Para visualizar a aplicação em perfeito funcionamento, é preciso executar o Stubby4j seguindo as instruções informadas no email do desafio proposto.
Primeiramente descompacte o arquivo zip do projeto no local de sua preferência. Em seguida abra o projeto normalmente no Android Studio. Para executar o projeto no emulador do Android, não é necessário nenhuma configuração

adicional. Caso queira utilizar um dispositivo real ou o emulador Genymotion é necessário informar o endereço da máquina que está hospedando o web service no arquivo na constante API_BASE_URL no seguinte arquivo do projeto:
TwVacationPlanner/data_remote/src/main/java/br/com/nglauber/twvacationplanner/data/remote/service/RemoteWebService.kt
Caso queira executar a aplicação sem o Android Studio, acesse via terminal a pasta onde o projeto foi descompactado e digite o seguinte comando para baixar o Gradle Wrapper:

```
gradle wrapper
```

Em seguida, abra o emulador do Android (AVD ou Genymotion). Para instalar o aplicativo no dispositivo conectado, digite:

```
./gradlew installDebug
```

Para executar os testes de UI, utilize o comando:

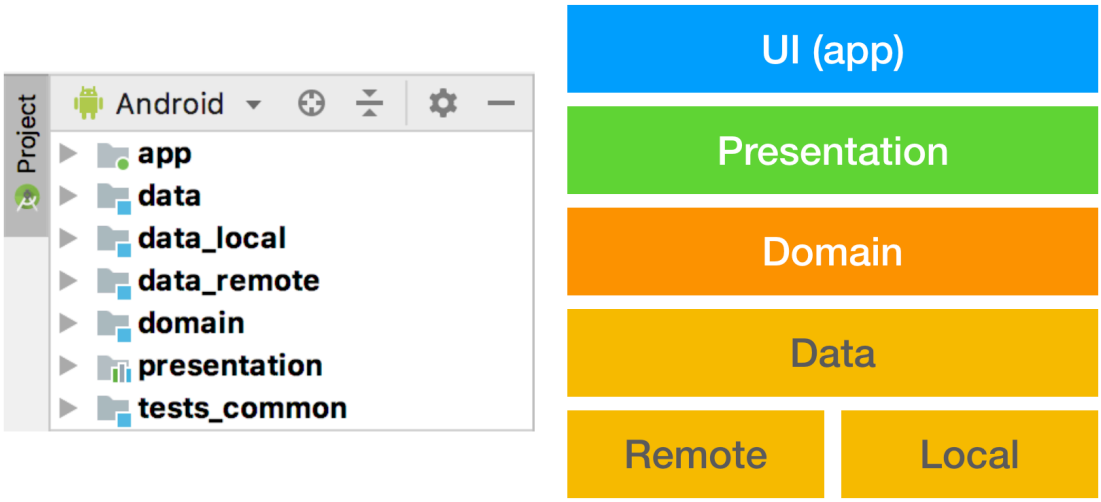
```
./gradlew connectedAndroidTest
```

Para rodar os testes unitários, digite:

```
./gradlew test
```

Arquitetura

A arquitetura utilizada segue os princípios do padrão **Clean Architecture** (http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html). Na camada de apresentação foi utilizado o padrão **MVVM** (*Model-View-ViewModel*).
A estrutura do projeto é apresentada na figura a seguir.



Visão Geral do Projeto

A decisão de separar a aplicação em módulos traz diversas vantagens como: o desacomplamento de código e dependências; facilidade de reuso; tempo de compilação; melhor distribuição do trabalho no caso do trabalho em equipe; etc.
Cada um dos módulos serão explicados nas próximas seções.

Módulo data (Kotlin)

Motivação

Esse módulo possui classes básicas e interfaces que serão utilizadas pelos demais módulos da aplicação. Nele também é definido, de forma abstrata, a parte de persistência da aplicação representada pelos repositórios.

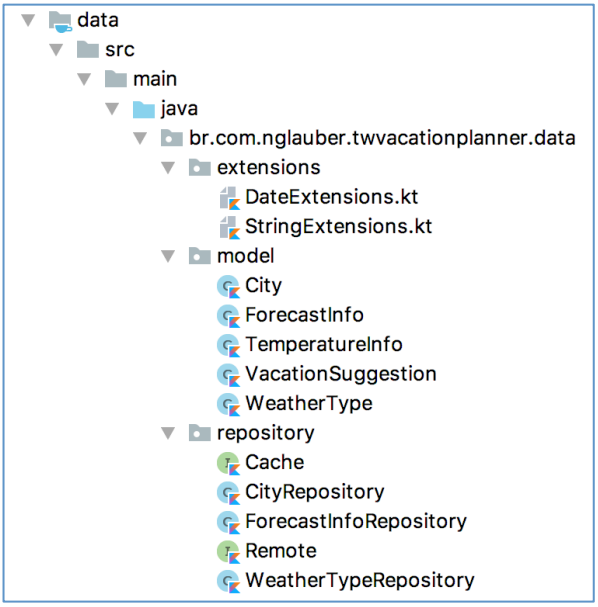
Organização

No pacote **extensions** estão declaradas algumas *extension functions*. Em **model** ficam as classes básicas que representam informações da aplicação (DTO – *Data Transfer Objects*).

No pacote **repository** foram definidos os três repositórios da aplicação que são responsáveis por fornecer os dados para as camadas superiores.

- Um repositório permite realizar a busca por cidades;
- Outro pesquisa os tipos de clima (*Cold, Hot, Cloudy*, ...);
- E o outro traz a informação da previsão do tempo de todos os dias de um determinado ano.

Os dados que serão obtidos do web service estão definidos na interface **Remote**.
Para evitar que a busca seja feita várias vezes no servidor, foi declarada a interface **Cache** para implementar um mecanismo básico de cache.
Todos os repositórios utilizam apenas as interfaces. As respectivas implementações são feitas em outros módulos permitindo o maior desacoplamento do código.



Estrutura do módulo data.

Dependências

Nesse módulo já é definida a utilização do **RX Java** (https://github.com/ReactiveX/RxJava) para o processamento do fluxo assíncrono dos dados por parte dos repositórios.

Testes

Foram implementados alguns testes unitários utilizando:

- **JUnit4** (https://junit.org/junit4/);
- **Mockito-Kotlin** (https://github.com/nhaarman/mockito-kotlin).

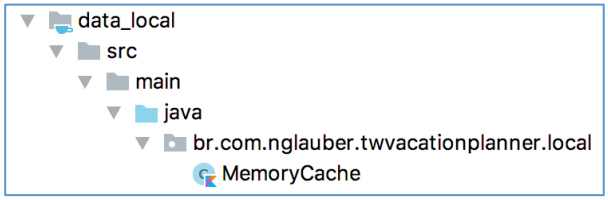
Módulo data_local (Kotlin)

Motivação

O objetivo desse módulo é prover uma implementação de persistência local para cache que foi definido no módulo **data**. Entretanto, a implementação atual apenas realiza o armazenamento das informações em memória.

Organização

A aplicação possui apenas uma classe para salvar informações em memória. Ela utiliza *generics* para permitir salvar qualquer tipo de objeto.



Estrutura do módulo data_local.

Dependências
Assim como o módulo **data**, esse módulo também utiliza o RX Java.

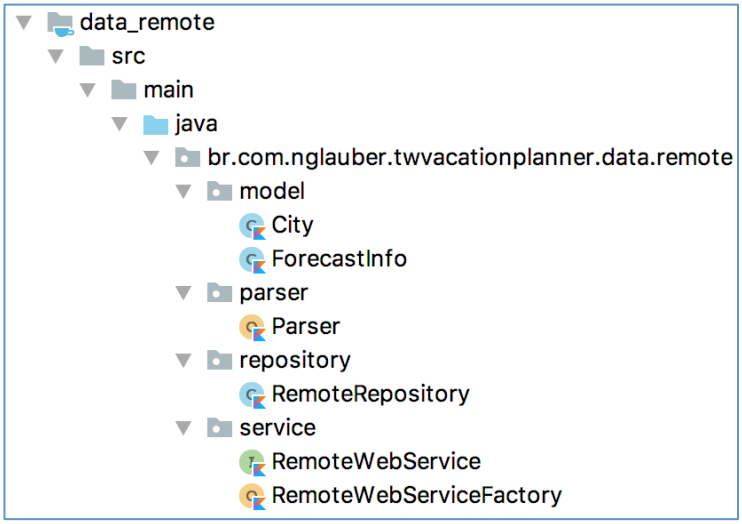
Testes
Foram implementados alguns testes unitários utilizando **JUnit4** (https://junit.org/junit4/).

Módulo data_remote (Kotlin)

Motivação
Nesse módulo é feita a implementação do serviço remote que fornece informações para os repositórios definidos no módulo data.
Para encontrar a lista de cidades disponíveis, é feita uma requisição HTTP (GET) para o endereço **<ip_do_servidor>:8882/cities/**
A lista de condições climáticas existentes é obtida ao realizar uma requisição para o endereço **<ip_do_servidor>:8882/weather/**
Por fim, para descobrir os climas diários de uma determinada cidade em um determinado ano, o endereço é **<ip_do_servidor>:8882/cities/<id_cidade>/year/<ano>**

Organização
No pacote **model** temos algumas classes duplicadas do módulo **data**. A razão disso é que, apesar de armazenarem as mesmas informações, as classes desse módulo utilizam anotações que não são necessárias no módulo **data**.
A classe responsável por realizar a conversão dos objetos das classes do pacote **model** para a classe correspondente do módulo **data** está definida no pacote **parser**.

A implementação real do repositório remoto definido no módulo data está localizada no pacote **repository**. E por fim, no pacote **service** ficam as classes que definem o acesso ao serviço HTTP.



Estrutura do módulo data_remote.

Dependências

Nesse módulo foram utilizadas as seguintes bibliotecas:

- Retrofit (https://github.com/square/retrofit) acesso ao serviço web;
- Adaptador do Retrofit para RXJava (https://github.com/square/retrofit/tree/master/retrofit-adapters/rxjava2);
- Conversor do Retrofit de JSON para objetos utilizando a biblioteca Gson (https://github.com/square/retrofit/tree/master/retrofit-converters/gson).

Testes

Foram implementados alguns testes unitários utilizando:

- **JUnit4** (https://junit.org/junit4/);
- **Mockito-Kotlin** (https://github.com/nhaarman/mockito-kotlin).

Módulo domain (Kotlin)

Motivação

Cada ação realizada pela aplicação está mapeada em um caso de uso. Cada caso de uso é declarado em uma classe. O objetivo do caso de uso é isolar a "lógica de negócio" das demais tarefas da aplicação (como "lógica de persistência" e "lógica de UI").

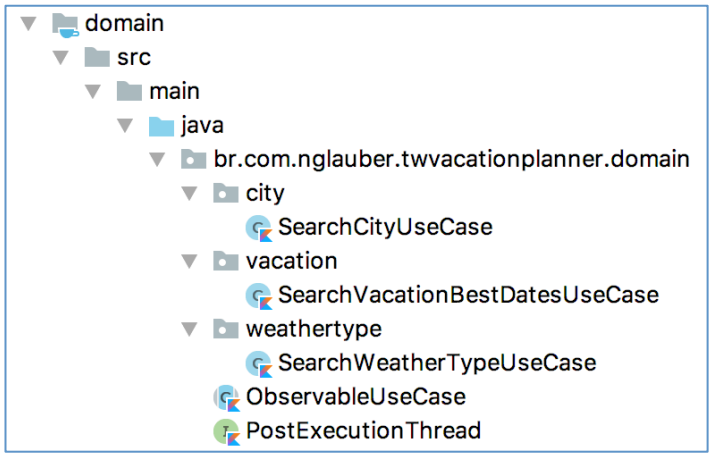
Organização

A classe **ObservableUseCase** define um caso de uso que retorna um objeto observável do RX Java. Esse projeto possui três casos de uso:

- Busca de cidades;
- Pesquisa de tipos climáticos;

- Procura de datas disponíveis para o usuário tirar férias.

A interface **PostExecutionThread** permite criar uma abstração para a *thread* onde os resultados são exibidos. Essencialmente ela utilizará a *Main Thread* do Android, mas como esse é um módulo Kotlin (e não Android) isso não é preciso ser declarado aqui.



Estrutura do módulo domain.

Dependências

Assim como os demais módulos, esse módulo também utiliza o RX Java.

Testes

Foram implementados alguns testes unitários utilizando:

- **JUnit4** (https://junit.org/junit4/);
- **Mockito-Kotlin** (https://github.com/nhaarman/mockito-kotlin).

Módulo presentation (Android)

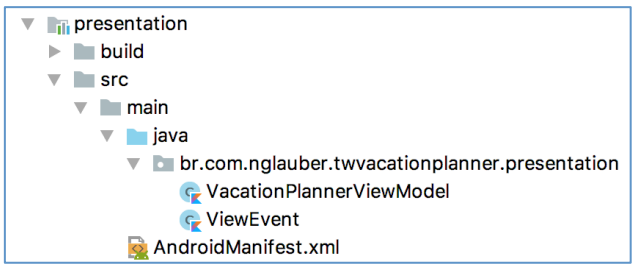
Motivação

O objetivo desse módulo é isolar a lógica de apresentação da sua exibição propriamente dita. É muito comum ter essa camada junto da UI, mas optei por essa abordagem pois ela fica independente da UI. Com isso, poderíamos reutilizar esse módulo para uma aplicação Android TV, Android Auto ou quem sabe até para Wear OS.

Organização

Como a aplicação só possui uma tela, esse módulo só possui uma classe que representa o View-Model (do padrão MVVM). É ela a responsável por manter a lógica de apresentação e serve de intermediário entre a UI e a camada de domínio.

A classe ViewEvent abstrai eventos de UI para exibir três estados: carregando, sucesso ou erro.



Estrutura do módulo presentation.

Dependências

Esse módulo utiliza as bibliotecas de *View Model* e *Live Data* (https://developer.android.com/topic/libraries/architecture/) do Jetpack *Architecture Components*.

O ViewModel foi escolhido por facilitar a retenção do estado tela durante mudanças de configuração (como rotacionar a tela). O LiveData foi utilizado para que a UI possa receber atualizações automáticas vindas do View Model. O RX Java poderia fazer esse papel, mas optei pelo Live Data pelo fato de ser "*life-cycle aware*", ou seja, as atualizações de tela só são realizadas quando a activity estiver nos estados de *started* ou *resumed*.

Testes

Foram implementados alguns testes unitários utilizando:

- **JUnit4** (https://junit.org/junit4/);
- **Mockito-Kotlin** (https://github.com/nhaarman/mockito-kotlin);
- A biblioteca de **Core-Testing** do Jetpack .

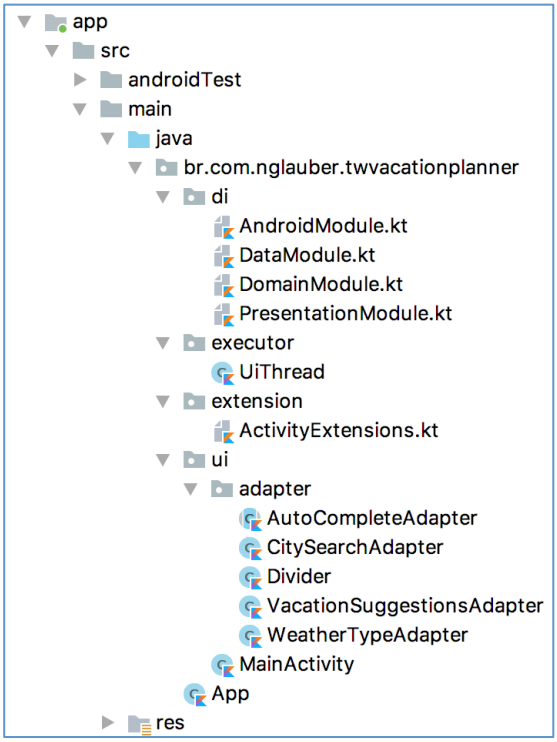
Módulo app (Android)

Motivação

Toda a interface gráfica da aplicação é definida nesse módulo.

Organização

A aplicação utiliza injeção de dependências onde cada módulo da aplicação recebe as instâncias necessárias para realizar o seu trabalho. Esse trabalho é feito no módulo **di** (*Dependency Injection*). Essa abordagem além de promover a centralização da criação dos serviços internos da aplicação, o desacoplamento do código, ainda facilitou bastante a escrita dos testes.



Estrutura do módulo app.

No pacote **executor** existe apenas a classe **UiThread** que implementa a interface **PostExecutionThread** do módulo **domain**.
Por fim, no pacote **ui** fica a Activity principal (e única) da aplicação. Os adapters para os componentes de listagem e suas classes auxiliaries ficam no pacote **adapter**.

Dependências

Além das tradicionais bibliotecas do Android (AppCompat, RecyclerView, ConstraintLayout e Material Design) foi utilizado o **Koin** (https://github.com/InsertKoinIO/koin) para injeção de dependências.

Testes

Foram implementados alguns testes de integração utilizando **Espresso** para validar os casos de exemplo enviados no email em que o desafio foi proposto.

Conclusão

Sem sombra de dúvidas esse foi um desafio interessante de ser implementado. Gostaria bastante de ter dedicado mais tempo a ele, pois creio que a implementação atual possui uma série de melhorias a serem feitas em cada um dos módulos. Na camada de UI, por exemplo, poderia haver uma opção onde o usuário possa selecionar vários tipos de clima ao mesmo tempo, sem a necessidade de digitar todos manualmente. Ou ainda, após realizar a busca, seria interessante ter um botão para limpar a busca atual, facilitando o início de uma nova busca.

O cache da aplicação poderia ser mais inteligente, salvando as informações no banco de dados ao invés de apenas em memória.
Em relação aos testes, foram escritos apenas os testes mais simples, utilizando basicamente JUnit, Mockito e Espresso. Apesar de simples, eles ajudaram a encontrar diversos bugs e melhorias bastante durante o desenvolvimento.

Referências

Curso do Joe Birch (@hitherejoe) no Caster.io
https://caster.io/courses/android-clean-architecture
https://github.com/hitherejoe/GithubTrending/

Meus repositórios
https://github.com/nglauber/books_jetpack
https://github.com/nglauber/tdcapp
https://github.com/nglauber/dominando_android2