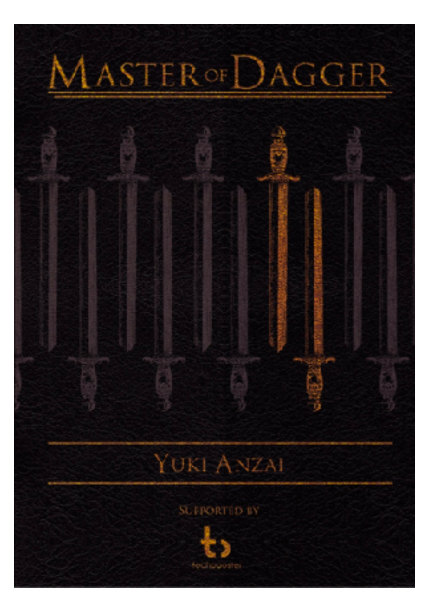
アプリをリリースできる状態に保ったまま段階的にリファクタリングするための戦略と戦術

Strategies and tactics for incremental refactoring

Yuki Anzai (@yanzm)
DroidKaigi 2024

Yuki Anzai

- Google Developer Expert for Android
- X (twitter) : @yanzm
- blog : y-anz-m.blogspot.com
- uPhyca Inc,



リファクタリング

リファクタリング

Code refactoring is the process of restructuring existing source code **without changing its external behavior**.

https://en.wikipedia.org/wiki/Code_refactoring

なぜリファクタリングしたくなるのか？

なぜリファクタリングしたくなるのか？

- 古いライブラリを置き換えたい

なぜリファクタリングしたくなるのか？

- 古いライブラリを置き換えたい ← 外的要因

なぜリファクタリングしたくなるのか？

- Coroutine
- Hilt
- ViewModel
- Jetpack Compose
- 公式が推奨している構成にしたい
- …

なぜリファクタリングしたくなるのか？

- Coroutine
- Hilt
- ViewModel
- Jetpack Compose
- 公式が推奨している構成にしたい
- …

← これらで何かを

解決・改善

したい

リファクタリングで何を解決したいのか

リファクタリングで何を解決したいのか

- わかりにくさ

わかりにくいってなんだろう？

わかりにくいってなんだろう？

- 前提知識が使えない

前提知識

- 一般的な知識
- ソフトウェア開発としての知識
- Android Platform の知識
- Android アプリ開発の知識
- プロジェクト特有の知識

前提知識

- **一般的な知識**
    - **単語の意味**
- ソフトウェア開発としての知識
- Android Platform の知識
- Android アプリ開発の知識
- プロジェクト特有の知識

前提知識

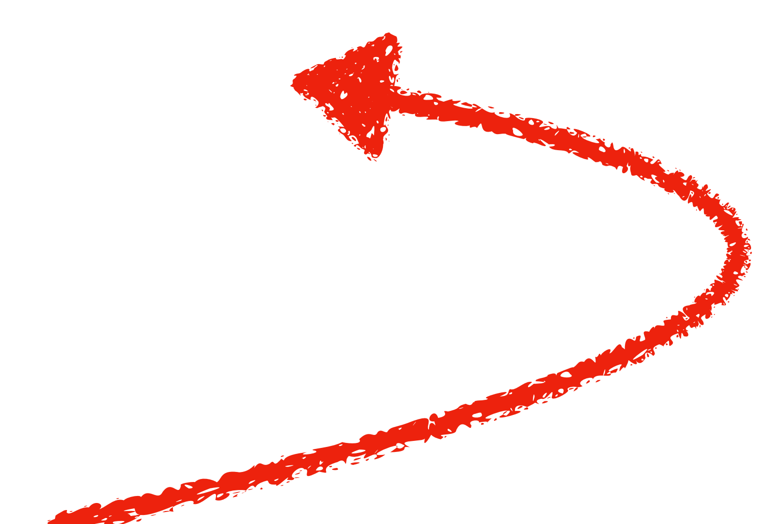
- 一般的な知識
- ソフトウェア開発としての知識
- **Android Platform の知識**
  - **Activity, Fragment, …**
- Android アプリ開発の知識
- プロジェクト特有の知識

前提知識

- 一般的な知識
- ソフトウェア開発としての知識
- Android Platform の知識
- **Android アプリ開発の知識**
  - **よく使われるライブラリや公式おすすめの構成**
- プロジェクト特有の知識

前提知識

- 一般的な知識
- ソフトウェア開発としての知識
- **Android Platform の知識**
- **Android アプリ開発の知識**
- プロジェクト特有の知識



前提知識

- 一般的な知識
- ソフトウェア開発としての知識
- Android Platform の知識
- Android アプリ開発の知識
- **プロジェクト特有の知識**

わかりにくいってなんだろう？

- 前提知識が使えない
- **認知能力の限界**
  - マジカルナンバー

わかりにくいってなんだろう？

- 前提知識が使えない
- 認知能力の限界
  - マジカルナンバー
    - **グループ化**
    - **階層化**

モジュールが多すぎる問題

- [IMO] （トップレベルの）モジュール数を7個以下に抑えたい

モジュールが多すぎる問題

- [IMO]（トップレベルの）モジュール数を7個以下に抑えたい
- グループ化
  - core/model, …
  - feature/…

モジュールが多すぎる問題

- [IMO] （トップレベルの）モジュール数を7個以下に抑えたい
- グループ化
- モジュールの必要性
  - ビルドの高速化
  - 依存方向の強制
  - internal を使った実装のカプセル化

リファクタリングの戦略

リファクタリングの戦略

- 戦略1：まず環境を整えよ
- 戦略2：順番を吟味せよ、末端から攻略せよ
- 戦略3：いつでも中断できる単位ですすめよ

戦略1：まず環境を整えよ

- CI (少なくともビルドとユニットテストは…)
- version catalogs
- format check tool
- build-logic (すでにモジュールがたくさんあるなら)
- ❌ 最初に必要になりそうなモジュールを全部用意する

戦略2：順番を吟味せよ、末端から攻略せよ

- なぜ順番が重要なのか？
  - アプリをリリースできる状態に保ったままリファクタリングするにはテストが必要

戦略2：順番を吟味せよ、末端から攻略せよ

- なぜ順番が重要なのか？
  - アプリをリリースできる状態に保ったままリファクタリングするにはテストが必要
- A のリファクタリングには A の unit test が必要
  - A が依存している B のリファクタリングをしないと A の unit test が書けない
  - B → A の順番でリファクタリングしないといけない

戦略2：順番を吟味せよ、末端から攻略せよ

- 末端とは
  - 依存しているものがない → テストできる
    - 例）サーバーやデータベースと通信する部分
  - 依存されていない → 影響範囲が小さい → 手動テストで確認することが可能
    - 例）UI の特定のパーツ

戦略3：いつでも中断できる単位ですすめよ

- いつでも中断できる単位で段階的に進める

Model Project

Model Project

- 一部 Kotlin でほぼ Java
- Volley (Deprecated libraries)
- ほとんどのロジックが Activity や Fragment、カスタム View に書かれている
- ViewModels, Hilt, Compose なし
- Test なし
- Single module

リファクタリングしたいところ

- Java → Kotlin
- Volley → Retrofit
- 巨大な Activity, Fragment の分割
  - ロジックを適切なクラスに切り出す
  - ViewModels 導入
  - Repositories 導入
- Multi module
  - version catalogs 導入
- Hilt 導入
- Compose 導入
  - Design System
- Tests 導入

リファクタリングの順番

1. 環境の設定
   1. CI
   2. gradle, AGP version の更新
   3. Version catalogs
   4. Hilt
2. Volley → Retrofit
   1. Module 追加
   2. Repository, ViewModel 導入
3. Compose
   1. Design system 整備

戦略1：まず環境を整えよ

Strategy1：setup Environment first

CI

CI環境のセットアップ

- Github actions
  - https://github.com/android/nowinandroid/blob/main/.github/workflows/Build.yaml
  - `run: ./gradlew :app:testDebug`
  - `run: ./gradlew :app:assemble`
- Bitrise
- …

参考

```yaml
steps:
  - name: Checkout
    uses: actions/checkout@…

  - name: Set up JDK 17
    uses: actions/setup-java@…
    with:
      distribution: 'zulu'
      java-version: 17

  - name: Setup Gradle
    uses: gradle/actions/setup-gradle@…
    with:
      validate-wrappers: true
      gradle-home-cache-cleanup: true

  - name: Run local tests
    if: always()
    run: ./gradlew :app:testDebug

  - name: Build all build type and flavor permutations
    run: ./gradlew :app:assemble
```

:app:assembleDebug

for only debug
build type

gradle と AGP version の更新

gradle と AGP version の更新

- Gradle : 8+
- Android Gradle Plugin : 8+

AGP version の更新

- 7.x → 7.4.1
  - namespace
- 7.4.1 → 8.0.2 → 8.x
  - https://developer.android.com/build/releases/past-releases/agp-8-0-0-release-notes#default-changes
  - `android.enableR8.fullMode = false`
  - `android.nonTransitiveRClass = false # if uses resources defined other modules`

Version catalogs

Version catalogs

- https://developer.android.com/build/migrate-to-catalogs
- https://github.com/android/nowinandroid/blob/main/gradle/libs.versions.toml
- 定義名のアルファベット順に並べるのがおすすめ

```
androidx-activity = { module = "androidx.activity:activity-ktx", version.ref = "android
androidx-activity-compose = { module = "androidx.activity:activity-compose", version.re
androidx-annotation = { module = "androidx.annotation:annotation", version.ref = "andro
…
```

```
androidx-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "ktx" }

androidx-ktx = { module = "androidx.core:core-ktx", version.ref = "ktx" }
```

Add format check tool

Add format check tool

- spotless
  - https://github.com/diffplug/spotless
  - 使用する Kotlin formatter library を選べる（ktfmt, ktlint, …）
- ktlint
- …

Add spotless to version catalogs

```
[versions]
…
ktlint = "1.3.1"
…
spotless = "6.25.0"

…

[plugins]
…
spotless = { id = "com.diffplug.spotless", version.ref = "spotless" }
```

build.gradle (project level)

```
plugins {
    …
    alias(libs.plugins.spotless)
}

spotless {
    kotlin {
        …
    }
    kotlinGradle {
        …
    }
    format("xml") {
        …
    }
}
```

```
spotless {
    kotlin {
        target("**/*.kt")
        targetExclude(
            "**/build/**/*.kt",
            "app/src/main/java/com/sample/existing/**/*.kt",
        )
        ktlint(libs.versions.ktlint.get())
            .editorConfigOverride(
                …
            )
    }
    kotlinGradle {
        target("*.gradle.kts")
        ktlint(libs.versions.ktlint.get())
    }
    format("xml") {
        target("**/*.xml")
        targetExclude(
            "**/build/**/*.xml",
            "app/src/main/res/layout/*.xml",
        )
    }
}
```

.editorconfig

```
[*.{kt,kts}]
…
ktlint_standard_class-signature=disabled # not applied with spotless
```

```
spotless {
    kotlin {
        …
        ktlint(libs.versions.ktlint.get())
            .editorConfigOverride(
                mapOf(
                    "ktlint_standard_function-signature" to "disabled",
                ),
            )
    }

    …
}
```

Add format check to CI

```yaml
steps:
  …

  - name: Setup Gradle
    uses: gradle/actions/setup-gradle@v4
    with:
      validate-wrappers: true
      gradle-home-cache-cleanup: true

  - name: Check spotless
    run: ./gradlew spotlessCheck

  - name: Run local tests
    if: always()
    run: ./gradlew :app:testDebug
```

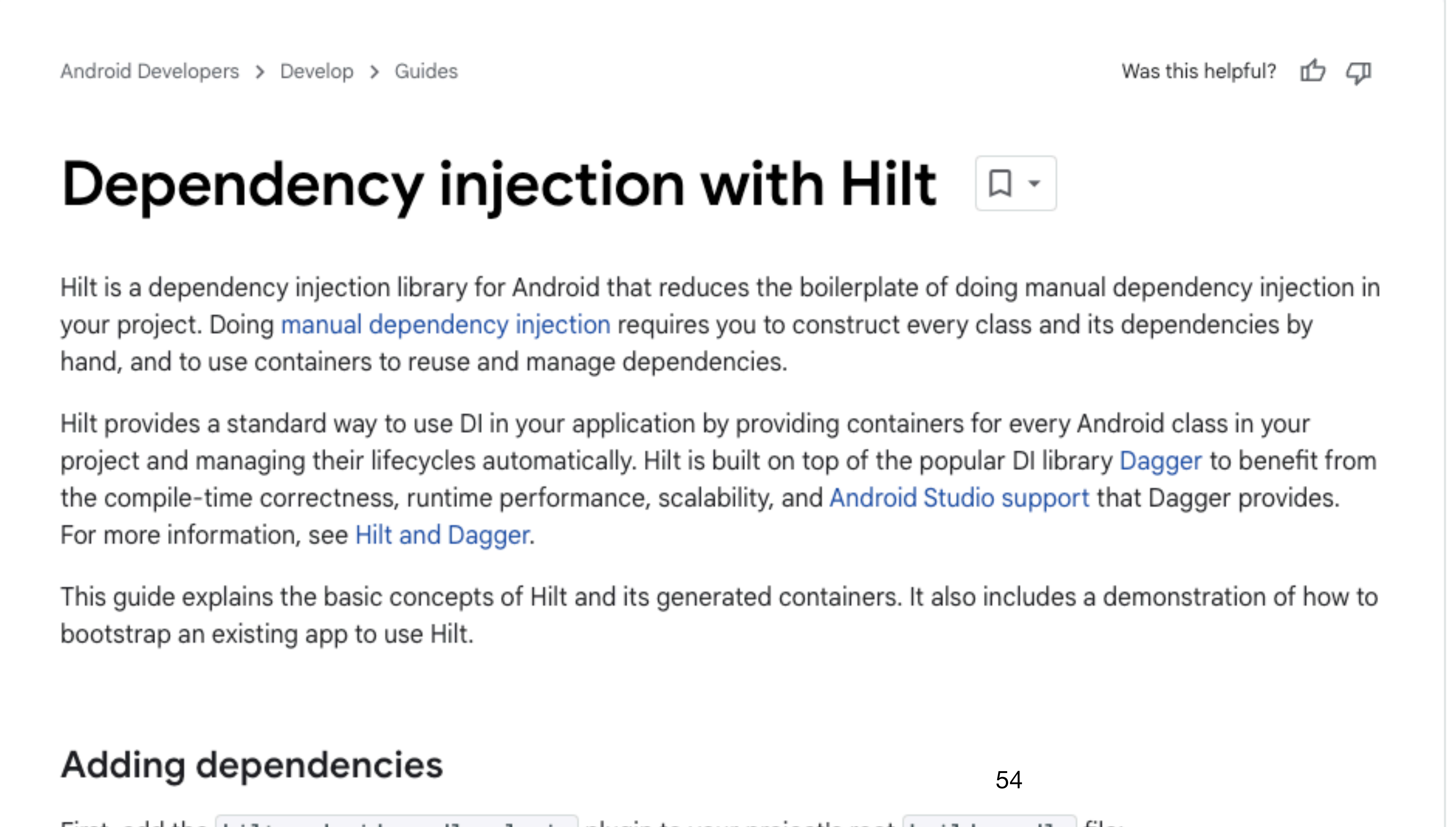
Add hilt

Hilt？

- すでに hilt を導入していたり、coin とか他の DI ライブラリを導入している場合 → 先に進む
- すでに hilt を使っている → 先に進む
- dagger も hilt も使ってない → hilt を導入する
- dagger を使っているがまだ hilt に移行していない → 段階的移行は可能
  - 他のリファクタリングの前に hilt への移行を完了することをおすすめします

Add hilt to the project

- https://developer.android.com/training/dependency-injection/hilt-android



libs.versions.toml

```toml
[versions]
…
hilt = “2.52"
kotlin = "2.0.10"
ksp = "2.0.10-1.0.24"
…

[libraries]
…
hilt-android = { module = "com.google.dagger:hilt-android", version.ref = "hilt" }
hilt-compiler = { module = "com.google.dagger:hilt-compiler", version.ref = "hilt" }
…

[plugins]
…
hilt = { id = "com.google.dagger.hilt.android", version.ref = "hilt" }
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
…
```

project level build.gradle

```
plugins {
    …
    alias(libs.plugins.hilt) apply false
    alias(libs.plugins.ksp) apply false
}
```

module level build.gradle

```
plugins {
    …
    alias(libs.plugins.hilt)
    alias(libs.plugins.ksp)
}

dependencies {
    …
    implementation(libs.hilt.android)
    ksp(libs.hilt.compiler)
}
```

```kotlin
@HiltAndroidApp
class MyApplication : Application() {
    …
}
```

AndroidManifest.xml

```xml
<application
    android:name=".MyApplication"
    …>

    …

</application>
```

戦略2：順番を吟味せよ、末端から攻略せよ

Start refactoring from the edge

As-Is

app module

JSON

```
{
  “name” : “…”,
  …
}
```

サーバーと
通信する部分

ItemResponse

UI

To-Be

core module

Item

JSON

api module

app module

Item

Item

(Old)
ItemResponse

サーバーと
通信する部分

Repository

ViewModel

UI

As-Is

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        …

        VolleyUtil.getInstance(this).request(
                new ItemsRequest(
                        response -> {
                            recyclerViewAdapter.submitList(response);
                        },
                        error -> {
                            …
                        }
                )
        );
    }
    …
}
```

api, core module

core module

api module

```
dependencies {
    implementation(project(":core"))
    …
}
```

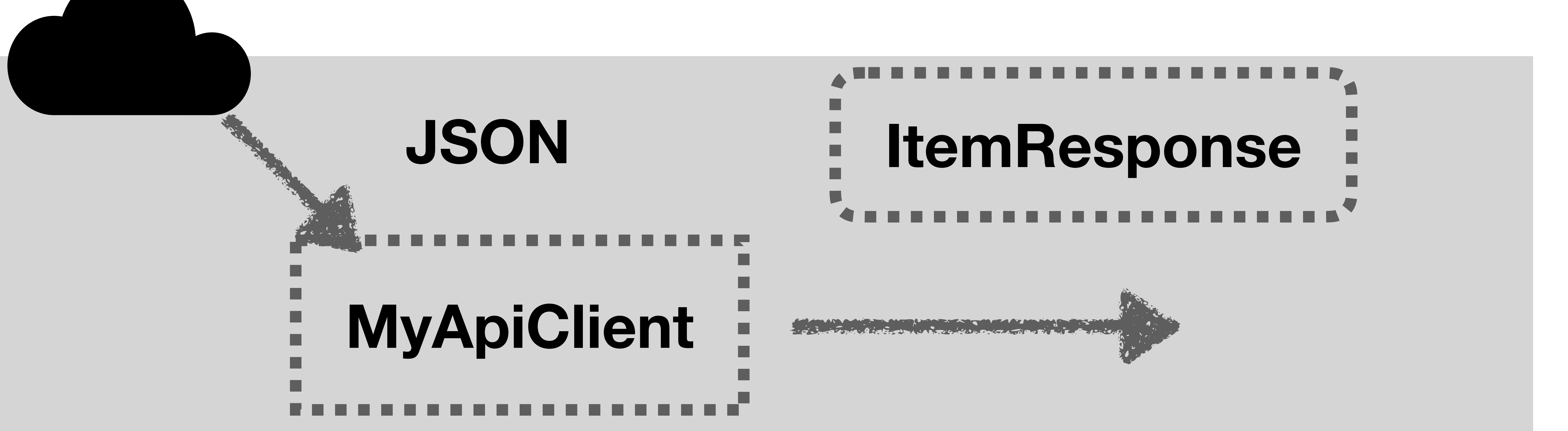
app module

```
dependencies {
    implementation(project(":core"))
    implementation(project(":api"))
    …
}
```

api module

1. **internal** Response classes
2. **internal** ApiClient (Retrofit)
3. Api interface and **internal** implementation of Api interface (wrapper of ApiClient)

```kotlin
internal interface MyApiClient {

    @GET("/items")
    suspend fun getItems(): List<ItemResponse>
}


@Serializable
internal data class ItemResponse(
    @SerialName("id") val id: Int,
    @SerialName("name") val name: String,
)
```



api module

```kotlin
interface MyApi {

    suspend fun getItems(): List<Item>
}


internal class DefaultMyApi(
    private val apiClient: MyApiClient
) : MyApi {

    override suspend fun getItems(): List<Item> {
        val response = apiClient.getItems()
        return response.map { ItemMapper(it) }
    }
}
```

core module

Item

MyApi

JSON

ItemResponse

implements

Item

MyApiClient

DefaultMyApi

api module

e.g.

success

```
{
  "status" : "OK",
  "data" : {
      ...
  }
}
```

error

```
{
  "status" : "NG",
  "message" : "invalid request parameter"
}
```

e.g.

success

```
{
  "status" : "OK",
  "data" : {
      ...
  }
}
```

error

```
{
  "status" : "NG",
  "message" : "invalid request parameter"
}
```

```
@Serializable
internal data class DataResponse(
    @SerialName("status") val status: String,
    @SerialName("message") val message: String?,
    @SerialName("data") val data: Data?,
) {

    @Serializable
    data class Data(
        @SerialName("id") val id: Int,
        …
    )
}
```

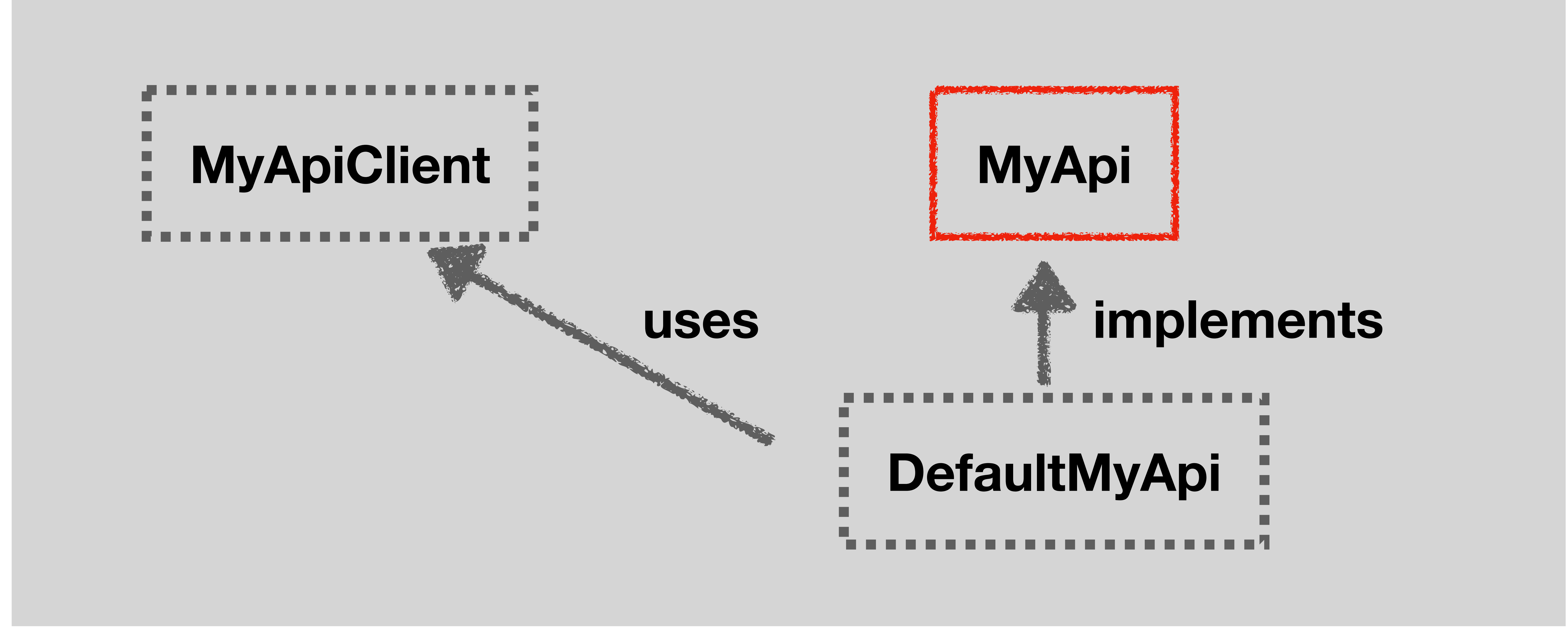
e.g.

```
internal class DefaultMyApi(
    private val apiClient: MyApiClient
) : MyApi {

    override suspend fun getData(): Data {
        val response = apiClient.getData()
        if (response.status != "OK") {
            throw MyApiException.InvalidRequest(
               message = "status = ${response.status}, message = ${response.mess
            )
        }

        val data = response.data
            ?: throw MyApiException.InvalidResponse(
               message = "data was null",
            )

        return DataMapper(data)
    }
}
```

defined in core module

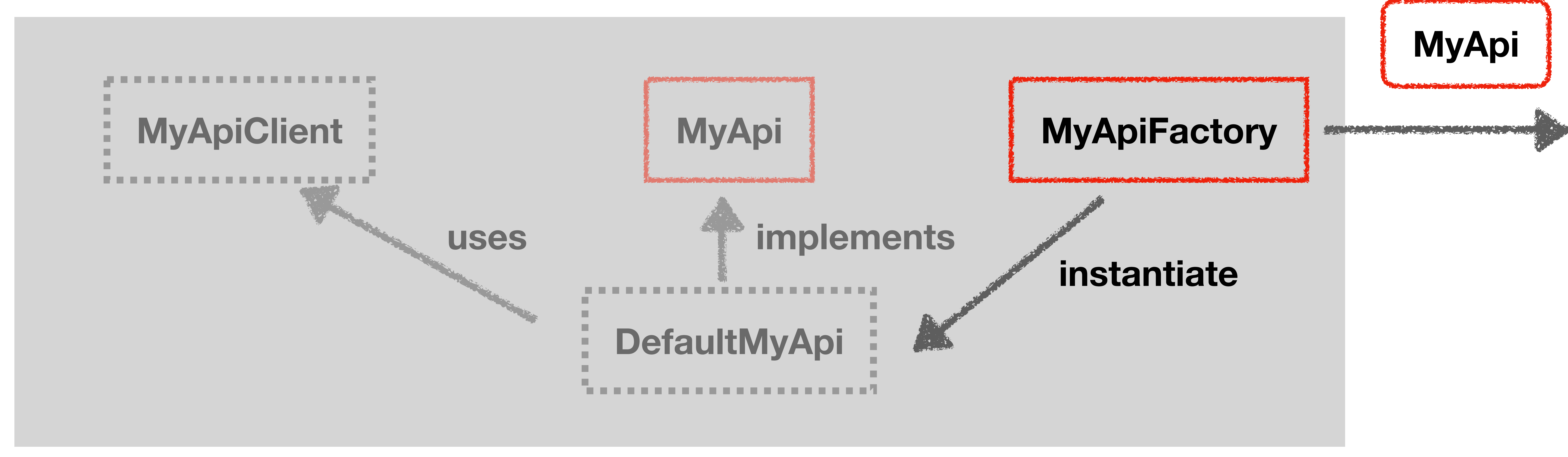
MyApi インスタンスの生成

api module



MyApi インスタンスの生成

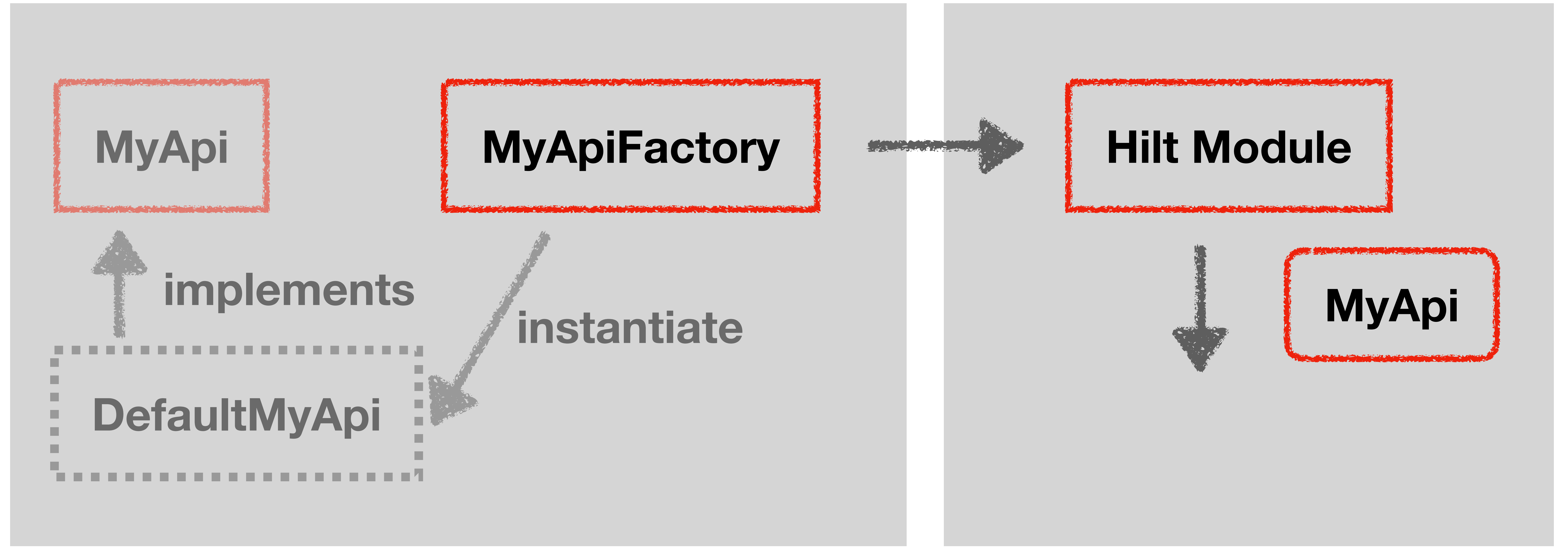
api module



api module

```
object MyApiFactory {

   fun create(
        baseUrl: String,
        tokenProvider: () -> String?,
        …
    ): MyApi {
        val apiClient = Retrofit.Builder()
            .baseUrl(baseUrl)
            .client(
                OkHttpClient.Builder()
                    .addInterceptor { chain ->
                        …
                    }
                    .build(),
            )
            …
            .build()
            .create(MyApiClient::class.java)

        return DefaultMyApi(apiClient)
    }
}
```

MyApi のインスタンスを hilt で管理する



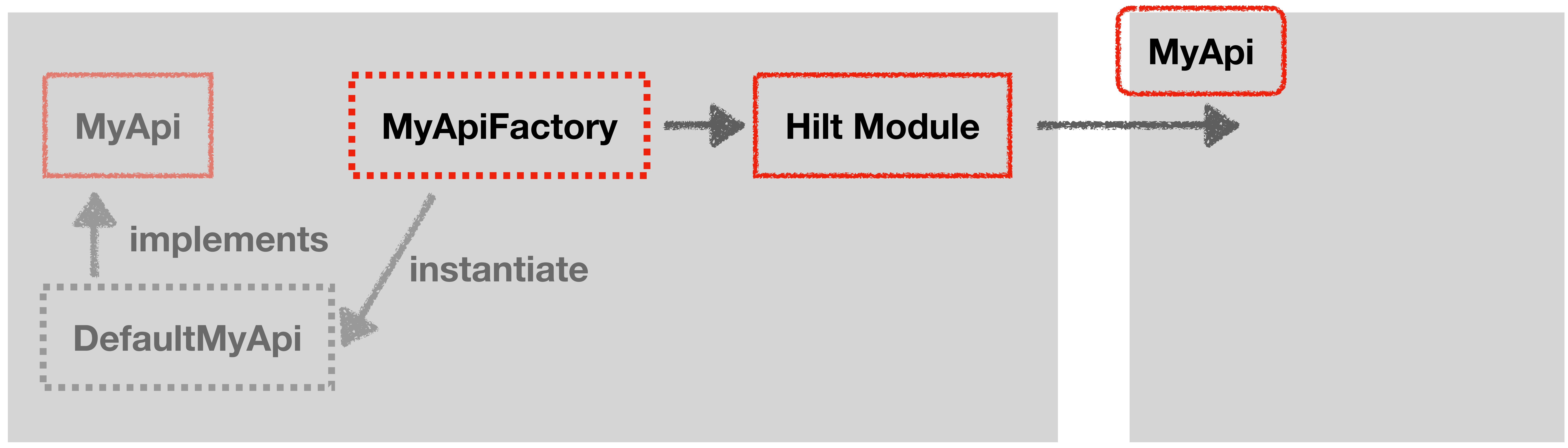
api module

```kotlin
object MyApiFactory {

    …

    fun create(
        …
    ): MyApi {
        …
    }
}
```

app module

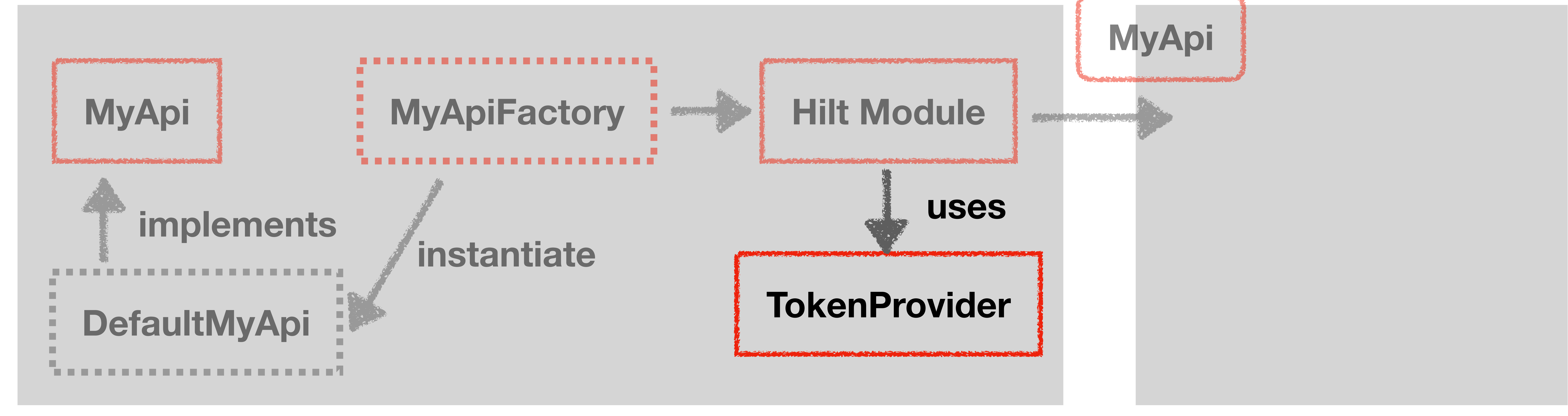
```kotlin
@InstallIn(SingletonComponent::class)
@Module
object AppModule {

    @Singleton
    @Provides
    fun provideMyApi(
        …
    ): MyApi {
        return MyApiFactory.create(
            tokenProvider = { … },
            …
        )
    }
}
```

internal MyApiFactory



api module

```
interface TokenProvider {
    fun provide(): String?
}
```



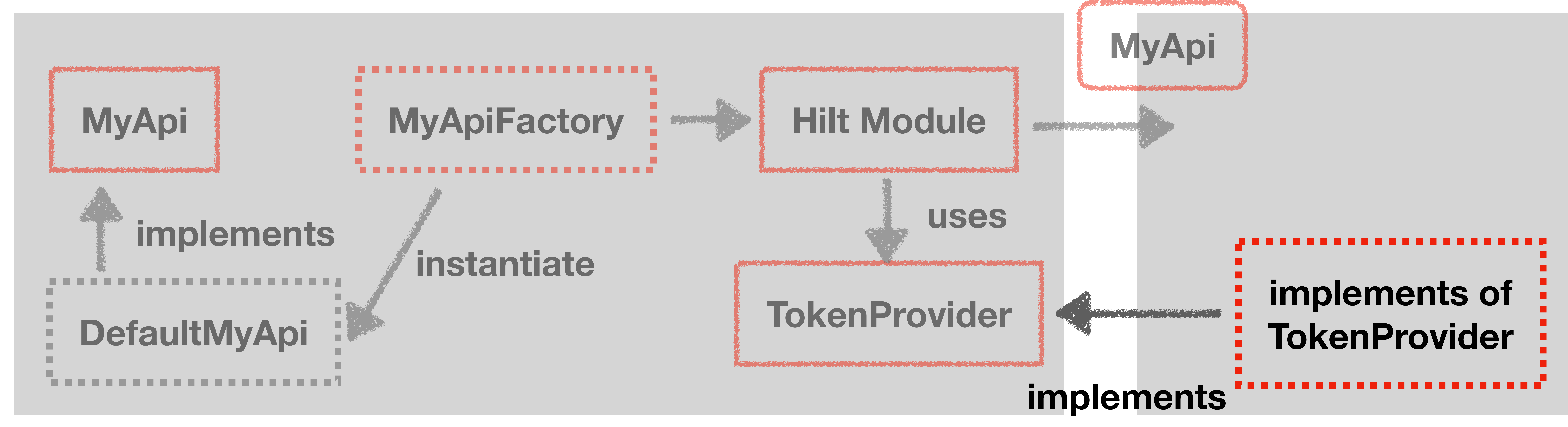
api module

```kotlin
interface TokenProvider {
    fun provide(): String?
}


internal object MyApiFactory {
    …
}
```

```kotlin
@InstallIn(SingletonComponent::class)
@Module
object ApiModule {

    @Singleton
    @Provides
    fun provideMyApi(
        tokenProvider: TokenProvider,
        …
    ): MyApi {
        return MyApiFactory.create(
            tokenProvider::provide,
            …
        )
    }
}
```

app module

```
class DefaultTokenProvider(
    …
) : TokenProvider {
    override fun provide(): String? {
        return …
    }
}
```



app module

```
@InstallIn(SingletonComponent::class)
@Module
object AppModule {

    @Singleton
    @Provides
    fun provideTokenProvider(
        …
    ): TokenProvider {
        return DefaultTokenProvider(
            …
        )
    }
}
```

Test

- ItemResponse
- ItemMapper
- DefaultMyApi
- **MyApiFactory, MyApi with MockWebServer of OkHttp**

api/src/test/resources/items.json

```json
[
  {
    "id": "item id",
    "name": "item name"
  }
]
```

api/src/test/kotlin/...

```kotlin
internal object TestResourceReader {

    fun readFileAsString(
        fileName: String,
        charset: Charset = Charset.defaultCharset(),
    ): String {
        return javaClass.classLoader!!.getResource(fileName).readText(charset)
    }
}
```

```kotlin
@Test
fun items_success() = runTest {
    val server = MockWebServer().apply {
        enqueue(
            MockResponse().setBody(
                TestResourceReader.readFileAsString("items.json")
            )
        )
        start()
    }

    val baseUrl = server.url("/").toString()
    val token = UUID.randomUUID().toString()

    val api = MyApiFactory.create(
        baseUrl = baseUrl,
        tokenProvider = { token }
    )

    val items = api.getItems()
```

```
    val items = api.getItems()

    assertEquals(
        listOf(
            Item(
                id = ItemId("item id"),
                name = "item name"
            )
        ),
        items
    )

    val request = server.takeRequest()
    assertEquals("/items", request.path)
    assertEquals(
        "Bearer $token",
        request.getHeader("Authorization")
    )

    server.shutdown()
}
```

app module

Repository

```kotlin
interface ItemRepository {
    suspend fun getItems(): ApiResult<List<Item>>
}


class DefaultItemRepository @Inject constructor(
    private val myApi: MyApi
) : ItemRepository {

    override suspend fun getItems(): ApiResult<List<Item>> {
        return try {
            val result = myApi.getItems()
            ApiResult.Success(result)
        } catch (e: MyApiException) {
            ApiResult.Error(e)
        }
    }
}
```

Test

```
class DefaultItemRepositoryTest {

    private lateinit var repository: ItemRepository
    private lateinit var api: MyApi

    @Before
    fun setup() {
        api = mockk()
        repository = DefaultItemRepository(api)
    }

    @Test
    fun success() = runTest {
        coEvery { api.getItems() } returns listOf(…)

        val result = repository.getItems()
        assertEquals(ApiResult.Success(listOf(…)), result)
    }

    @Test
    fun error() = runTest {
        …
    }
}
```

```
@InstallIn(SingletonComponent::class)
@Module
interface BindModule {

    @Singleton
    @Binds
    fun bindItemRepository(
        defaultImplementation: DefaultItemRepository
    ): ItemRepository
}
```

```
class DefaultItemRepository @Inject constructor(
    private val myApi: MyApi
) : ItemRepository {

    …
}
```

core module

Item

JSON

Item

MyApi

Repository

api module

app module

core module

Item

JSON

Item

(Old)
ItemResponse

MyApi

Repository

UI

api module

app module

core module

Item

JSON

Item

Item

(Old)
ItemResponse

MyApi

Repository

ViewModel

UI

convert Item to (Old) ItemResponse

api module

app module

```kotlin
fun Item.toItemResponse(): ItemResponse {
    return ItemResponse(
        id = id.value,
        iconUrl = iconUrl,
        name = title,
    )
}
```

```kotlin
@Deprecated("use Item in core module")
data class ItemResponse(
    …
)
```

```kotlin
class ItemTest {

    @Test
    fun toItemResponse() {
        val item = Item(…)

        val itemResponse = item.toItemResponse()

        assertEquals(
            ItemResponse(…),
            itemResponse
        )
    }
}
```

```kotlin
@HiltViewModel
class MainViewModel @Inject constructor(
    private val itemRepository: ItemRepository,
) : ViewModel() {

    fun request(
        onSuccess: (List<ItemResponse>) -> Unit,
        onError: (Exception) -> Unit,
    ) {
        viewModelScope.launch {
            when (val result = itemRepository.getItems()) {
                is ApiResult.Error -> {
                    onError(result.e)
                }

                is ApiResult.Success -> {
                    onSuccess(result.data.map { it.toItemResponse() })
                }
            }
        }
    }
}
```

```
@AndroidEntryPoint
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        …

        MainViewModel viewModel = new ViewModelProvider(this).get(MainViewModel

        viewModel.request(
        VolleyUtil.getInstance(this).request(
                new ItemsRequest(
                        response -> {
                            recyclerViewAdapter.submitList(response);
                            return Unit.INSTANCE;
                        },
                        error -> {
                            …
                            return Unit.INSTANCE;
                        }
                )
        );
    }
    …
```

```kotlin
@HiltViewModel
class MainViewModel @Inject constructor(
    private val itemRepository: ItemRepository,
) : ViewModel() {

    fun request(
        onSuccess: (List<ItemResponse>) -> Unit,
        onError: (Exception) -> Unit,
    ) {
        viewModelScope.launch {
            when (val result = itemRepository.getItems()) {
                is ApiResult.Error -> {
                    onError(result.e)
                }

                is ApiResult.Success -> {
                    onSuccess(result.data.map { it.toItemResponse() })
                }
            }
        }
    }
}
```

```kotlin
@HiltViewModel
class MainViewModel @Inject constructor(
    private val itemRepository: ItemRepository,
) : ViewModel() {

    private val _items = MutableLiveData<ApiResult<List<ItemResponse>>>()
    val items: LiveData<ApiResult<List<ItemResponse>>>
        get() = _items

    fun request() {
        viewModelScope.launch {
            when (val result = itemRepository.getItems()) {
                is ApiResult.Error -> _items.postValue(result)
                is ApiResult.Success -> _items.postValue(
                    ApiResult.Success(result.data.map { it.toItemResponse() })
                )
            }
        }
    }
}
```

```java
@AndroidEntryPoint
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        …

        MainViewModel viewModel = new ViewModelProvider(this).get(MainViewModel.class)

        viewModel.getItems().observe(
            this,
            result -> {
                if (result instanceof ApiResult.Success<List<ItemResponse>> response)
                    adapter.submitList(response.getData());
                } else if (result instanceof ApiResult.Error error) {
                    …
                }
            }
        );

        viewModel.request();
    }
    …
}
```

Kotlin化

Kotlin 化の戦略

- 大きいファイルのまま Kotlin 化しない
  - ~ 300 行ぐらいに減らしてから

Kotlin 化の順番

1. static なクラスを top-level に移動

```
public class ItemAdapter extends ListAdapter<ItemResponse, ItemViewHolder> {
    …

    public static class ItemHolder extends RecyclerView.ViewHolder {
        …
    }
}

public class ItemHolder extends RecyclerView.ViewHolder {
    …
}
```

Kotlin 化の順番

1. static なクラスを top-level に移動
2. static じゃない inner class を static にして top-level に移動

```java
public class MainActivity extends AppCompatActivity {
    …

    private void onClickItem(Item item) {
        …
    }

    class ItemAdapter extends ListAdapter<Item, ItemViewHolder> {
        …

        @Override
        public void onBindViewHolder(@NonNull ItemViewHolder holder, int position) {
            holder.itemView.setOnClickListener(view -> {
              onClickItem(getItem(position));
            });
        }
    }
}
```

```java
public class MainActivity extends AppCompatActivity {
    …

    private void onClickItem(Item item) {
        …
    }

    static class ItemAdapter extends ListAdapter<Item, ItemViewHolder> {
        private final OnClickListener onClickItemListener;

        protected ItemAdapter3(OnClickListener listener) {
            …
            this.onClickItemListener = listener;
        }

        @Override
        public void onBindViewHolder(@NonNull ItemViewHolder holder, int position) {
            holder.itemView.setOnClickListener(view -> {
                onClickItemListener.onClick(getItem(position));
            });
        }
    }
}
```

Kotlin 化の順番

1. static なクラスを top-level に移動
2. static じゃない inner class を static にして top-level に移動
3. このクラスの責務ではない処理を、別クラスを作って委譲

Kotlin 化の順番

1. static なクラスを top-level に移動
2. static じゃない inner class を static にして top-level に移動
3. このクラスの責務ではない処理を、別クラスを作って委譲する
4. Helper 的な class/object を kotlin で用意し、300 行くらいになるまでメソッドを移動する

Kotlin 化の順番

1. static なクラスを top-level に移動
2. static じゃない inner class を static にして top-level に移動
3. このクラスの責務ではない処理を、別クラスを作って委譲する
4. Helper 的な class/object を kotlin で用意し、300 行くらいになるまでメソッドを移動する
5. kotlin 化し、helper クラスと合体

Kotlin 化の commit

1. 中身が Java のまま拡張子を .java から .kt に変えて commit する

```
% mv ItemAdapter.java ItemAdapter.java.kt
% git add -A
% git commit -m 'rename ItemAdapter.java to ItemAdapter.kt'
```

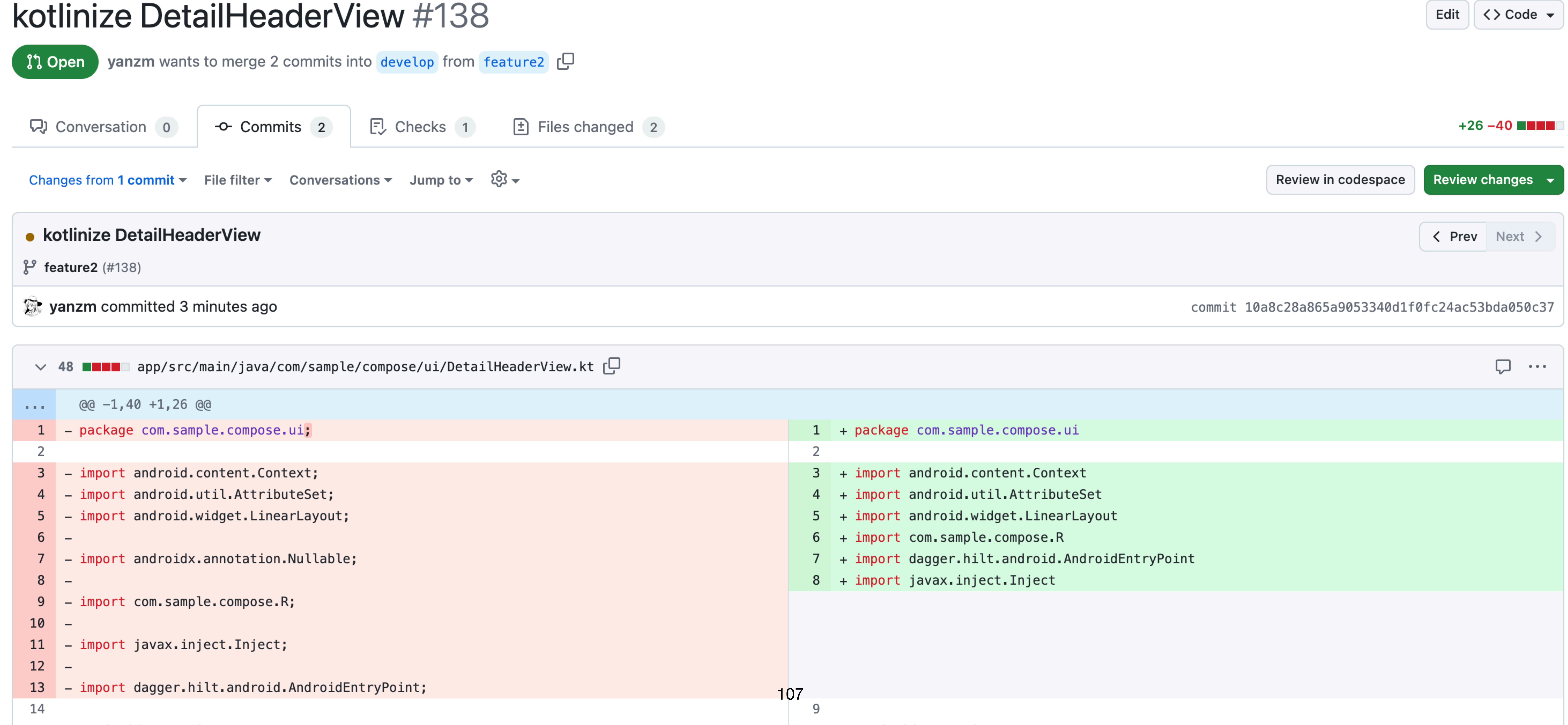
2. 拡張子を .kt から .java に戻す（commit しない）

```
% mv ItemAdapter.kt ItemAdapter.java
```

3. Kotlin 化して commit する

```
% git add -A
% git commit -m 'Kotlinize ItemAdapter'
```

Kotlin 化の commit



Compose への移行

1. AbstractComposeView を継承した CustomView

```
class DetailActivityComposeView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0,
) : AbstractComposeView(context, attrs, defStyleAttr) {

    @Composable
    override fun Content() {
        MyTheme {

        }
    }
}
```

ComposeView の限界

```kotlin
class DetailActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detail)

        val composeView = findViewById<ComposeView>(R.id.compose_view)
        composeView.setContent {
            MyTheme {
                Scaffold {
                    …
                }
            }
        }
    }
}
```

only with kotlin

2. 簡単な View を CustomView に置き換える

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout …>

    <TextView
        android:id="@+id/title_view"
        … />

    <net.yanzm.myapplication.DetailActivityComposeView
        android:id="@+id/compose_view"
        … />

</LinearLayout>
```

```
@Composable
fun DetailContent(
    title: String,
) {
    Text(
        text = title,
        …
    )
}

@Preview
@Composable
private fun Preview() {
    MyTheme {
        Surface {
            DetailContent(
                title = "title",
            )
        }
    }
}
```

```kotlin
class DetailActivityComposeView @JvmOverloads constructor(
    …
) : AbstractComposeView(context, attrs, defStyleAttr) {

    var title by mutableStateOf("")

    @Composable
    override fun Content() {
        MyTheme {
            Surface {
                DetailContent(
                    title = title,
                )
            }
        }
    }
}
```

```java
public class DetailActivity extends AppCompatActivity {

    private TextView titleView;
    private DetailActivityComposeView composeView;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        …

        titleView = findViewById(R.id.title_view);
        composeView = findViewById(R.id.compose_view);
    }

    private void update() {
        titleView.setText(…);
        composeView.setTitle(…);
    }
}
```

Compose への移行ステップ

1. AbstractComposeView を継承した CustomView を作る
2. 簡単な View を CustomView に置き換える
3. **隣接する View を CustomView に取り込む**
4. **全ての View が取り込まれるまでステップ3を繰り返す**

RecyclerView → Compose

RecyclerView → Compose

1. item layout in RecyclerView → Compose
2. RecyclerView → LazyColumn/LazyRow

item layout in RecyclerView → Compose

1. ViewHolder のリファクタリング
2. ViewHolder → Compose
    1. ViewHolder のレイアウトに対応する composable を作る
    2. 1 の composable を利用する CustomView を AbstractComposeView で作る
    3. 2 のCustomView を ViewHolder で使う
3. 全ての ViewHolder に対してステップ 2 を行う

ViewHolder のリファクタリング

- 生成メソッド
- bind メソッド

```java
public class ItemViewHolder extends RecyclerView.ViewHolder {
    …

    public static ItemViewHolder create(@NonNull ViewGroup parent) {
        View view = LayoutInflater.from(parent.getContext())
                .inflate(R.layout.list_item, parent, false);
        return new ItemViewHolder(view);
    }
}


public class ItemAdapter extends ListAdapter<Item, ItemViewHolder> {
    …

    @NonNull
    @Override
    public ItemViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType)
        return ItemViewHolder.create(parent);
    }
}
```

```java
public class ItemViewHolder extends RecyclerView.ViewHolder {

    private final TextView titleView;
    …

    public void bind(@NonNull String iconUrl, @NonNull String title) {
        …
        titleView.setText(title);
    }
    …
}

public class ItemAdapter extends ListAdapter<Item, ItemViewHolder> {

    …

    @Override
    public void onBindViewHolder(@NonNull ItemViewHolder holder, int position) {
        Item item = getItem(position);
        holder.bind(item.iconUrl, item.title);
    }
}
```

ViewHolder に対応する composable を作る

```
@Composable
fun ItemContent(
    iconUrl: String,
    title: String,
    modifier: Modifier = Modifier
) {
    Row(…) {
        AsyncImage(
            model = iconUrl,
            contentDescription = null,
            modifier = Modifier.size(40.dp)
        )
        Text(
            text = title,
            …
        )
    }
}
```

AbstractComposeView で CustomView を作る

```
class ItemView @JvmOverloads constructor(
    …
) : AbstractComposeView(context, attrs, defStyleAttr) {

    var iconUrl by mutableStateOf("")
    var title by mutableStateOf("")

    @Composable
    override fun Content() {
        MyTheme {
            Surface {
                ItemContent(
                    iconUrl = iconUrl,
                    title = title,
                )
            }
        }
    }
}
```

ViewHolder で CustomView を使う

```
public class ItemViewHolder extends RecyclerView.ViewHolder {

    private final ItemView itemView;

    private ItemViewHolder(@NonNull ItemView itemView) {
        super(itemView);
        this.itemView = itemView;
    }

    public void bind(@NonNull String iconUrl, @NonNull String title) {
        itemView.setIconUrl(iconUrl);
        itemView.setTitle(title);
    }

    public static ItemViewHolder create(@NonNull ViewGroup parent) {
        return new ItemViewHolder(new ItemView(parent.getContext()));
    }
}
```

RecyclerView → LazyColumn/LazyRow

1. RecyclerView に対応する composable を作る（LazyColumn/ LazyRow)
2. 1 の composable を利用する CustomView を AbstractComposeView で作る
3. Activity/Fragment で 2 の composable を使う

RecyclerView に対応する composable を用意

```
@Composable
fun ItemList(
    items: List<Item>
) {
    LazyColumn {
        items(
            items = items,
            key = { it.id.value },
            contentType = { "item" }
        ) { item ->
            ItemContent(
                iconUrl = item.iconUrl,
                title = item.title
            )
        }
    }
}
```

AbstractComposeView で CustomView を作る

```
class ItemListView @JvmOverloads constructor(
    …
) : AbstractComposeView(context, attrs, defStyleAttr) {

    var items by mutableStateOf<List<Item>>(emptyList())

    @Composable
    override fun Content() {
        MyTheme {
            Surface {
                ItemList(items)
            }
        }
    }
}
```

Activity/Fragment で CustomView を使う

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout …>

    …

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <net.yanzm.myapplication.ItemListView
        android:id="@+id/item_list_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Activity/Fragment で CustomView を使う

```
public class MainActivity extends AppCompatActivity {

    private RecyclerView recyclerView;
    private ItemAdapter adapter;
    private ItemListView itemListView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        recyclerView = findViewById(R.id.recycler_view);
        itemListView = findViewById(R.id.item_list_view);
    }

    …
    private void update(List<Item> items) {
        adapter.submitList(items);
        itemListView.setItems(items);
    }
}
```

まとめ

まとめ

- Android の共通の知識と人間の認知の限界を活用して、プロジェクトがよりわかりやすくなるようなリファクタリングをする
- リファクタリングの順番が重要、テストできる末端部分からはじめる
- AbstractComposeView を継承した CustomView を活用し、段階的に compose 化していく

Thank you