Chapitre 5 : Communication Réseau sous Android

Les APIs Android avec Retrofit

Découvrez comment Retrofit transforme la façon dont vos applications Android communiquent avec les serveurs et consomment des données en temps réel.

5.1 Architecture Client-Serveur : Les Fondations

Le Modèle Client-Serveur

Dans une application Android moderne, votre appareil agit comme un **client** qui envoie des requêtes à un **serveur distant**. Le serveur traite ces demandes et renvoie les données nécessaires.

Cette architecture permet de :

- Centraliser les données et la logique métier
- Synchroniser les informations entre plusieurs appareils
- Réduire la charge de traitement sur le mobile

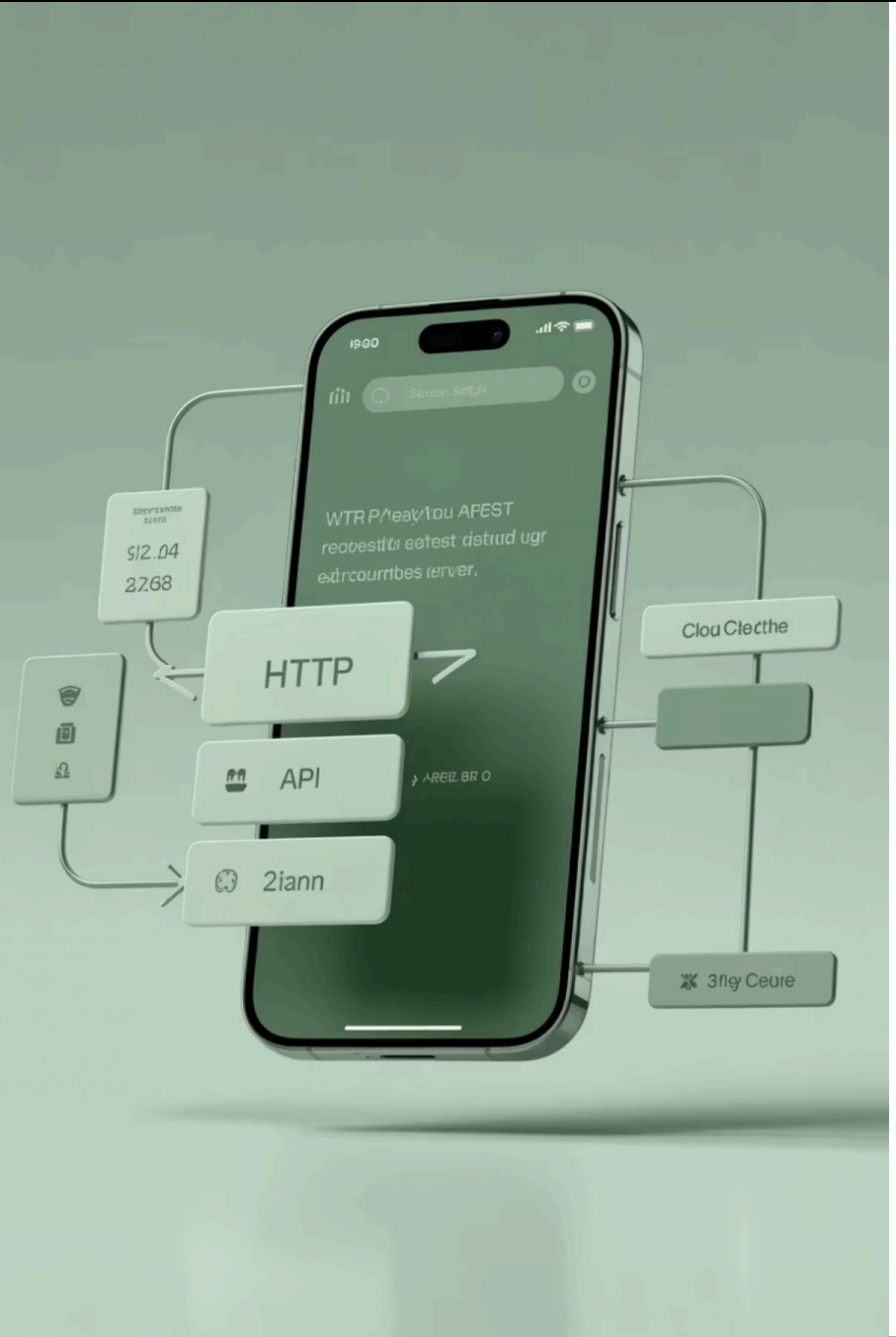
Qu'est-ce qu'une API ?

Une **API (Application Programming Interface)** est un ensemble de points d'accès (endpoints) qui permettent à votre application de communiquer avec le serveur.

Exemple d'endpoint :

```
https://api.exemple.com/users
```

Les données échangées sont généralement au format **JSON**, un format texte structuré, facile à lire et à manipuler.



Communication Client-Serveur en Action

Client Android

Envoie des requêtes HTTP/HTTPS avec paramètres

Réseau Internet

Transfert sécurisé des données via protocoles standards

Serveur API REST

Traite, stocke et renvoie les réponses JSON/XML

Ce schéma illustre le flux de communication bidirectionnel entre une application Android et un serveur distant, fondement de toute application mobile moderne connectée.

Fonctionnement dans une application Android

Interface Utilisateur

Le client Android gère l'interface utilisateur, les interactions locales et l'affichage des données. Il offre une expérience fluide et réactive à l'utilisateur, même pendant les communications réseau.

Délégation au Serveur

Les traitements lourds, les calculs complexes et la gestion centralisée des données sont intelligemment délégués au serveur. Cela permet d'alléger l'application mobile et de préserver la batterie de l'appareil.

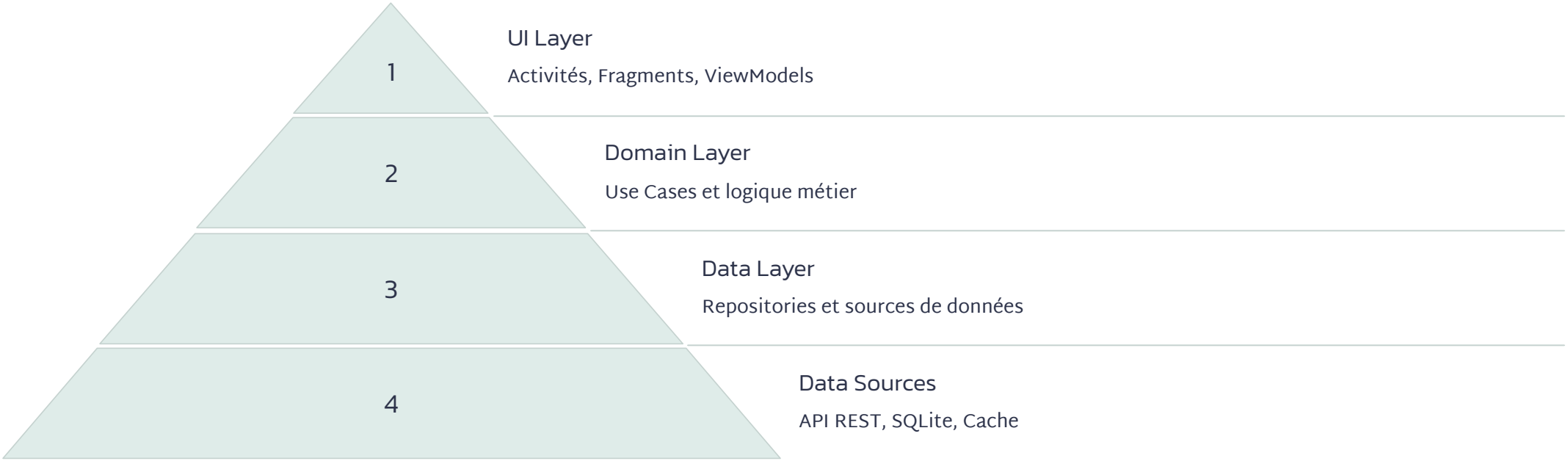
Protocoles Standards

La communication s'effectue via des protocoles éprouvés comme HTTP/HTTPS pour la sécurité, avec des formats d'échange comme JSON ou XML pour structurer les données de manière lisible et standardisée.

Outils Modernes

Des bibliothèques comme Retrofit ou Volley facilitent la récupération asynchrone des données, permettant de maintenir l'interface utilisateur réactive pendant les appels réseau.

Les couches clés dans une architecture Android moderne



Couche UI

Gère l'affichage et l'état de l'interface avec les Activités, Fragments et ViewModels qui maintiennent les données survivant aux changements de configuration.

Couche Domaine

Encapsule la logique métier dans des Use Cases réutilisables, indépendants de la technologie et faciles à tester unitairement.

Cette séparation claire en couches facilite considérablement la maintenance, les tests et l'évolutivité de l'application. Chaque couche a une responsabilité bien définie et communique avec les autres via des interfaces claires.

Avantages et défis de l'architecture client-serveur

Avantages majeurs

Centralisation des données

Une source unique de vérité pour toutes les applications, facilitant la cohérence et la synchronisation multi-plateformes.

Sécurité renforcée

Les données sensibles restent sur le serveur. Authentification, autorisation et chiffrement sont gérés côté serveur.

Évolutivité

Possibilité d'augmenter la puissance serveur sans modifier les applications clientes. Scaling horizontal et vertical facilités.

Défis à relever

Latence réseau

Gestion intelligente des délais de réponse, mise en cache locale, optimisation des requêtes et feedback utilisateur pendant le chargement.

Sécurité des échanges

Implémentation SSL/TLS, authentification OAuth, tokens JWT, protection contre les attaques MITM et injection SQL.

Multi-appareils

Compatibilité avec diverses versions Android, différentes tailles d'écran, gestion hors-ligne et synchronisation différée.

Bonnes pratiques essentielles : Sécuriser systématiquement les API avec authentification robuste, implémenter une gestion complète des erreurs réseau avec retry automatique, et optimiser les performances via la compression des données et le batching des requêtes.

5.2 HTTP : Le Protocole de Communication

Définition Simple

HTTP signifie **HyperText Transfer Protocol**. C'est littéralement la langue que parlent votre application et le serveur pour échanger des informations.

Chaque fois que vous consultez une page web, envoyez un message ou rafraîchissez votre fil d'actualité, c'est HTTP qui travaille en coulisses.

Comment Ça Fonctionne ?

- Votre application envoie une **requête HTTP**
- Le serveur reçoit et traite cette requête
- Le serveur renvoie une **réponse HTTP**
- Votre application affiche les données reçues

Les Méthodes HTTP : Les Verbes d'Action

Les méthodes HTTP sont comme des verbes dans une phrase. Elles indiquent au serveur quelle action vous souhaitez effectuer sur les données.

GET

Récupérer des données sans les modifier

Ex : Afficher la liste des films

POST

Envoyer de nouvelles données au serveur

Ex : Créer un nouveau compte

PUT

Modifier des données existantes

Ex : Mettre à jour un profil

DELETE

Supprimer des données

Ex : Retirer un produit du panier

Anatomie d'une Requête HTTP

Une requête HTTP est composée de plusieurs éléments structurés qui indiquent au serveur exactement ce que vous voulez faire.

La Méthode

GET, POST, PUT ou DELETE

L'URL

https://api.example.com/movies

L'adresse de la ressource demandée

Les En-têtes (Headers)

```
Content-Type: application/json
Authorization: Bearer token...
```

Métadonnées sur la requête

Le Corps (Body)

```
{ "name": "Salah" }
```

Les données envoyées (POST/PUT uniquement)

Structure d'une Réponse HTTP

Ce Que le Serveur Renvoie

Quand le serveur traite votre requête, il vous renvoie une réponse structurée contenant :

1 **Un Code HTTP**

Indique si tout s'est bien passé (200) ou s'il y a eu un problème (404, 500...)

2 **Un Corps de Réponse**

Les données demandées, souvent au format JSON

```
{ "id": 1, "title": "M" }
```

Headers et Codes de Réponse HTTP

Headers Essentiels

Les **headers** transportent des métadonnées importantes avec chaque requête :

Content-Type : Indique le format des données

```
Content-Type: application/json
```

Authorization : Authentifie l'utilisateur

```
Authorization: Bearer token123
```

Codes de Réponse

- **200 OK** : Requête réussie
- **201 Created** : Ressource créée avec succès
- **400 Bad Request** : Requête invalide
- **401 Unauthorized** : Authentification requise
- **404 Not Found** : Ressource introuvable
- **500 Internal Server Error** : Erreur serveur

Comprendre ces codes permet de gérer correctement les erreurs et d'afficher des messages pertinents à l'utilisateur.

5.3 Utilisation des web services (REST, JSON..)

Les Fondamentaux des APIs REST

JSON comme Format d'Échange

JSON (JavaScript Object Notation) est le format standard pour l'échange de données. Léger et lisible, il structure les données sous forme de paires clé-valeur et s'intègre parfaitement avec les data classes Kotlin.

```
{
  "id": 1,
  "nom": "Exemple"
}
```

Notion d'Endpoint

Un endpoint représente une URL spécifique exposant une ressource. Par exemple, `/api/users` pour accéder aux utilisateurs ou `/api/posts/1` pour un article spécifique.

Chaque endpoint définit les opérations possibles et la structure des données échangées.

Méthodes HTTP

- **GET** : Récupérer des données
- **POST** : Créer une ressource
- **PUT** : Modifier entièrement
- **DELETE** : Supprimer une ressource

Ces verbes HTTP correspondent aux opérations CRUD standard.

Pour tester vos compétences, **JSONPlaceholder** offre une API REST fictive gratuite avec des endpoints pour posts, users, comments, etc. Idéal pour l'apprentissage et le prototypage.

Retrofit ? C'est quoi et pourquoi ?

Retrofit est une bibliothèque développée par Square qui transforme radicalement la façon dont nous interagissons avec les APIs REST dans Android. Elle offre une approche élégante et type-safe pour effectuer des appels réseau.

Code simplifié

Retrofit réduit considérablement le boilerplate code nécessaire pour les appels réseau. Les annotations déclaratives comme @GET et @POST rendent le code intuitif et facile à maintenir.

Conversion automatique

La conversion JSON vers objets Kotlin/Java est entièrement automatisée grâce aux convertisseurs intégrés. Plus besoin de parser manuellement chaque réponse JSON.

Type-safety

Retrofit garantit la sécurité des types à la compilation, réduisant ainsi les erreurs runtime et améliorant la robustesse de votre application.

Performance optimale

Basé sur OkHttp, Retrofit bénéficie d'optimisations avancées comme le pooling de connexions, le cache HTTP et la gestion intelligente des timeouts.

Comment Retrofit fonctionne-t-il ?

01

Création du proxy dynamique

À l'exécution, Retrofit utilise la réflexion Java pour créer un proxy dynamique de votre interface API. Ce proxy intercepte tous les appels de méthodes définis dans l'interface.

02

Analyse des annotations

Retrofit parcourt les annotations (@GET, @POST, @Path, @Query, etc.) pour construire la requête HTTP appropriée avec les paramètres, headers et body nécessaires.

03

Transformation en requête HTTP

Les appels de méthodes sont convertis en requêtes HTTP complètes avec l'URL, la méthode, les en-têtes et le corps de la requête correctement formatés.

04

Gestion du threading

Retrofit gère automatiquement l'exécution sur des threads d'arrière-plan, évitant les blocages du thread principal (UI). Avec les coroutines Kotlin, cette gestion devient encore plus élégante.

05

Conversion JSON automatique

Les réponses JSON sont automatiquement désérialisées en objets Kotlin/Java grâce aux convertisseurs (Gson, Moshi, Jackson), et inversement pour les requêtes sortantes.

Cycle complet d'une requête Retrofit

Application Android → Retrofit → OkHttp → API REST → OkHttp → Gson/Moshi → Application Android

1. Application Android
   (UI / ViewModel)
2. Retrofit
   (Interface API + Annotations)
3. OkHttp (HTTP Client)
   (Envoie la requête HTTP au serveur)
4. API REST Server
   (Traite requêtes & renvoie JSON)
5. OkHttp Response
   (Réponse brute)
6. Gson/Moshi Converter
   (Convertit JSON → objets Java/Kotlin)
7. Application Android
   (UI / ViewModel)

01
L'utilisateur déclenche une action

02
Retrofit construit la requête via les annotations

03
OkHttp envoie la requête HTTP

04
Le serveur renvoie une réponse JSON

05
OkHttp reçoit la réponse

06
Retrofit utilise Gson/Moshi pour convertir JSON → Objets

07
L'application met à jour l'UI

Architecture de communication avec Retrofit

L'application Android envoie une requête HTTP via Retrofit, qui contacte l'API REST sur le serveur. Le serveur traite la demande et renvoie une réponse JSON que Retrofit convertit automatiquement en objets utilisables dans votre code.

Application Android

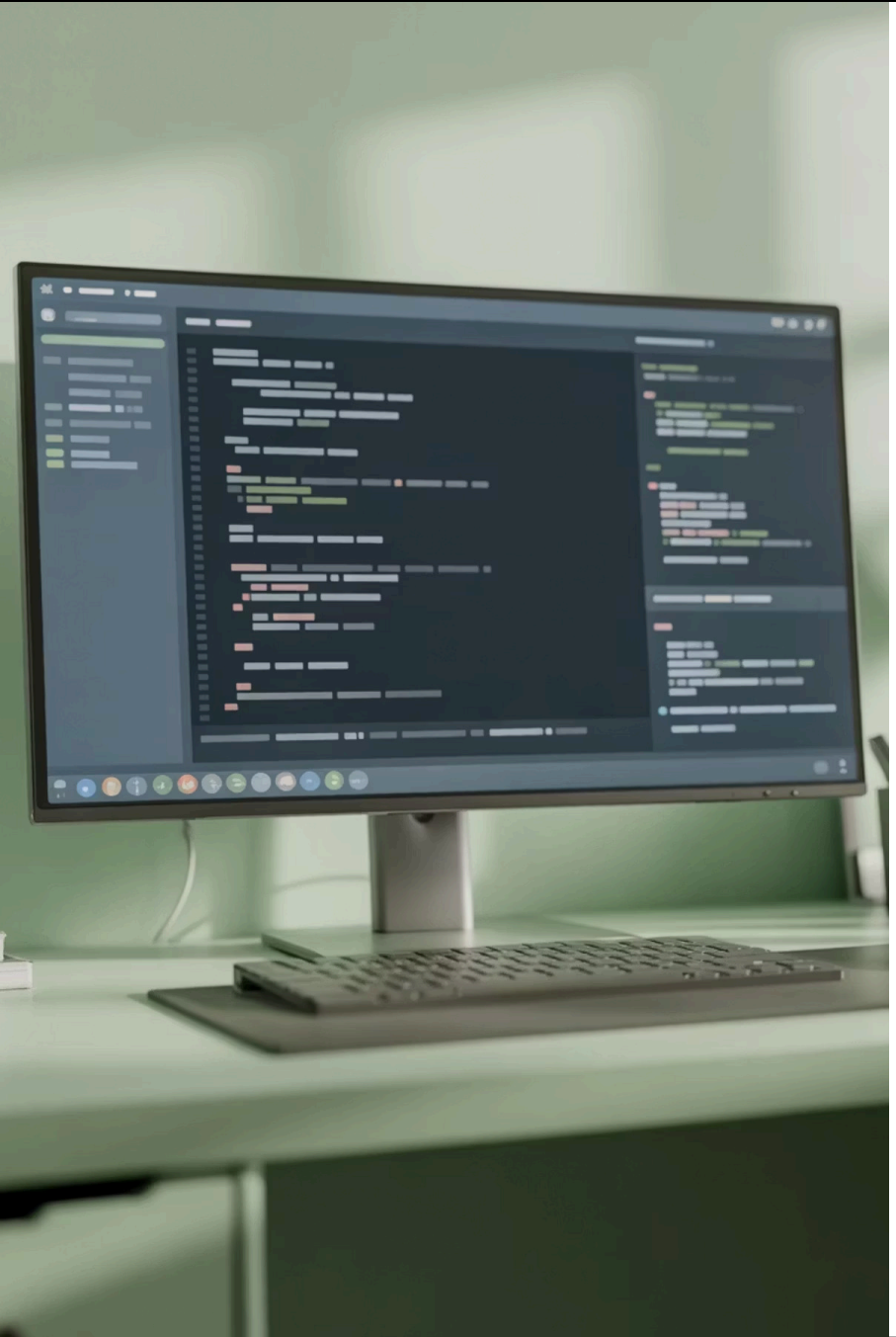
Interface utilisateur qui déclenche les requêtes

Retrofit Client

Gère la communication et la conversion des données

API REST

Serveur qui traite et renvoie les données JSON



Configuration de Retrofit

Mettons en place Retrofit dans votre projet Android en quelques étapes simples mais essentielles pour garantir une configuration optimale.

Étape 1 : Ajouter les dépendances

La première étape consiste à déclarer les dépendances nécessaires dans votre fichier build.gradle au niveau du module app. Ces bibliothèques travaillent ensemble pour offrir une solution complète de communication réseau.

Retrofit Core

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
```

La bibliothèque principale qui fournit le client HTTP type-safe

Gson Converter

```
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
```

Convertit automatiquement les réponses JSON en objets Kotlin/Java

N'oubliez pas de synchroniser votre projet après avoir ajouté ces dépendances !

Exemple de fichier gradle avec les 2 elements :

Ouvrez le fichier `build.gradle` (niveau module) et ajoutez ces dépendances essentielles :

```
dependencies {
    implementation "com.squareup.retrofit2:retrofit:2.9.0"
    implementation "com.squareup.retrofit2:converter-gson:2.9.0"
}
```

Le logging-interceptor est optionnel mais fortement recommandé pour le débogage de vos appels réseau.

Étape 2 : Connexion Réseau & Permissions

Permission INTERNET Obligatoire

Toute application Android nécessitant une connexion réseau doit déclarer la permission `android.permission.INTERNET` dans le fichier AndroidManifest.xml :

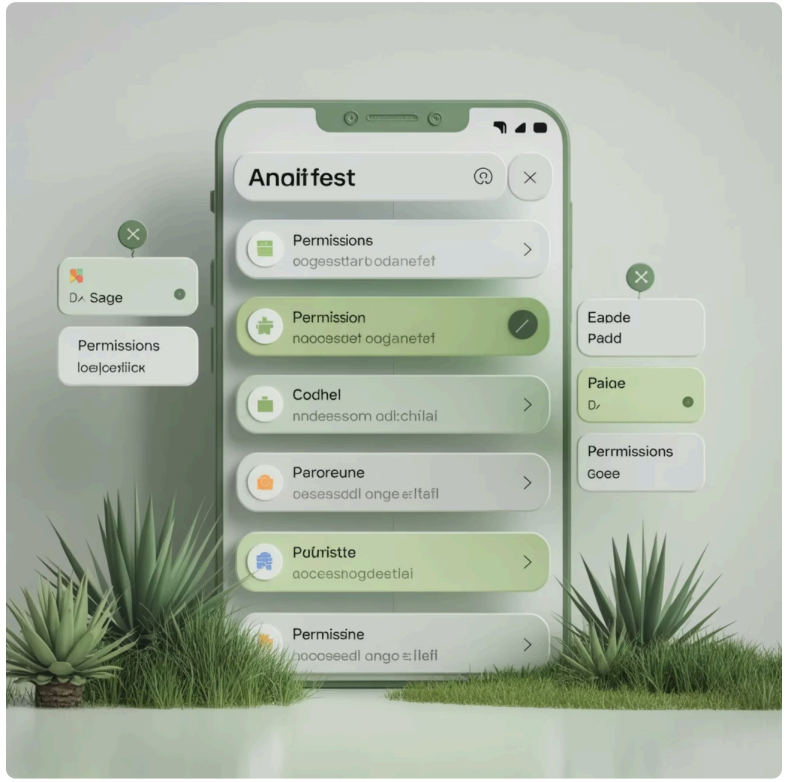
```
<uses-permission android:name="android.permission.INTERNET" />
```

Cette permission est automatiquement accordée à l'installation, sans demande explicite à l'utilisateur.

Pour les applications qui nécessitent une connectivité constante, vous pouvez également ajouter la permission ACCESS_NETWORK_STATE pour vérifier l'état de la connexion.

HTTPS Obligatoire depuis Android 9

Google impose l'utilisation de connexions sécurisées HTTPS par défaut depuis Android 9 (API 28). Les requêtes HTTP en clair sont bloquées pour protéger la confidentialité des utilisateurs.



Étape 3 : Modèle de données (Data Class)

Les data classes Java représentent la structure des données JSON reçues de l'API. Elles doivent correspondre exactement au format JSON pour que la conversion automatique fonctionne correctement.

Exemple JSON reçu

```
{
  "id": 12345,
  "title": "Inception",
  "poster": "/poster.jpg",
  "rating": 8.8,
  "releaseDate": "2010-07-16",
  "overview": "A thief who steals..."
}
```

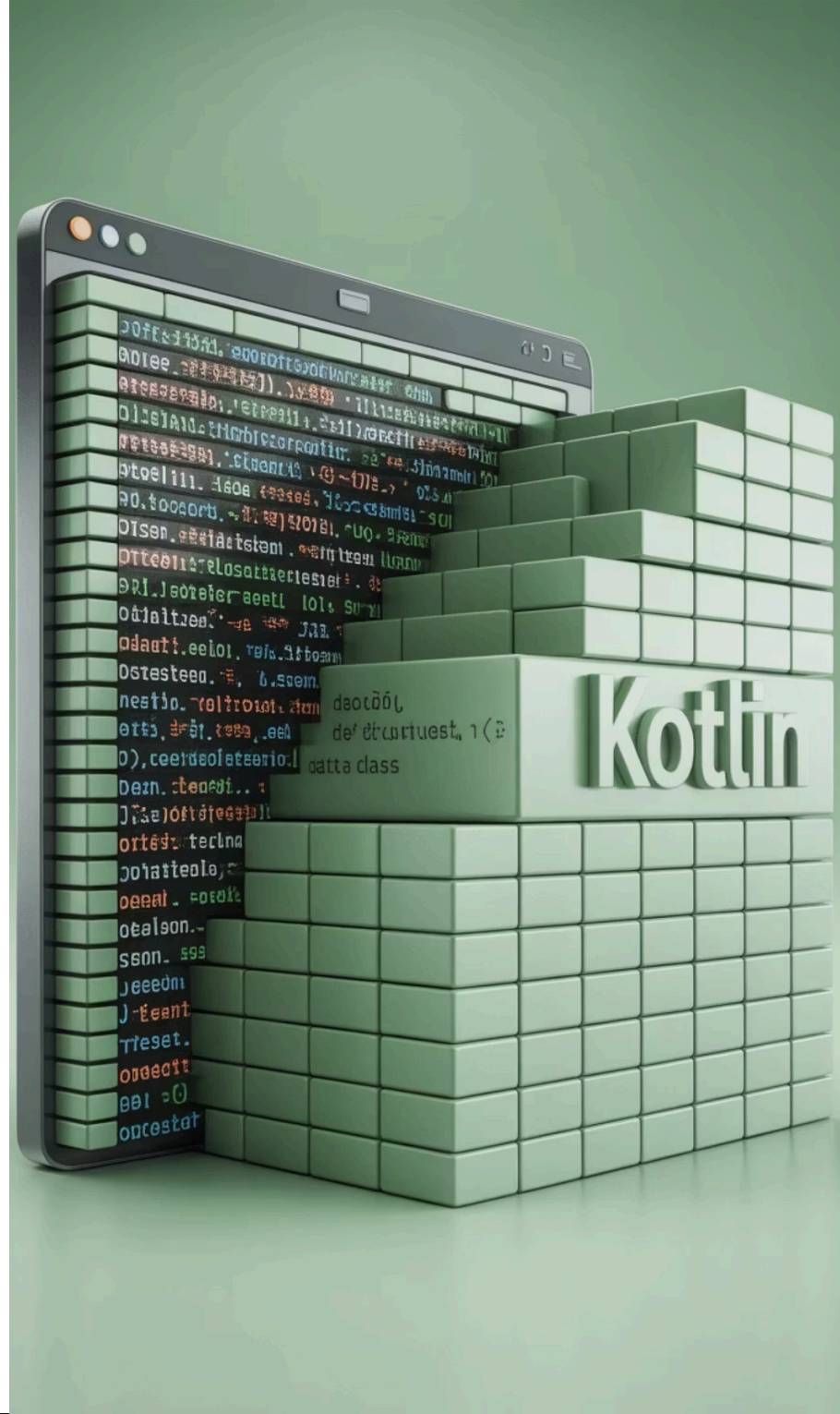
Data class correspondante

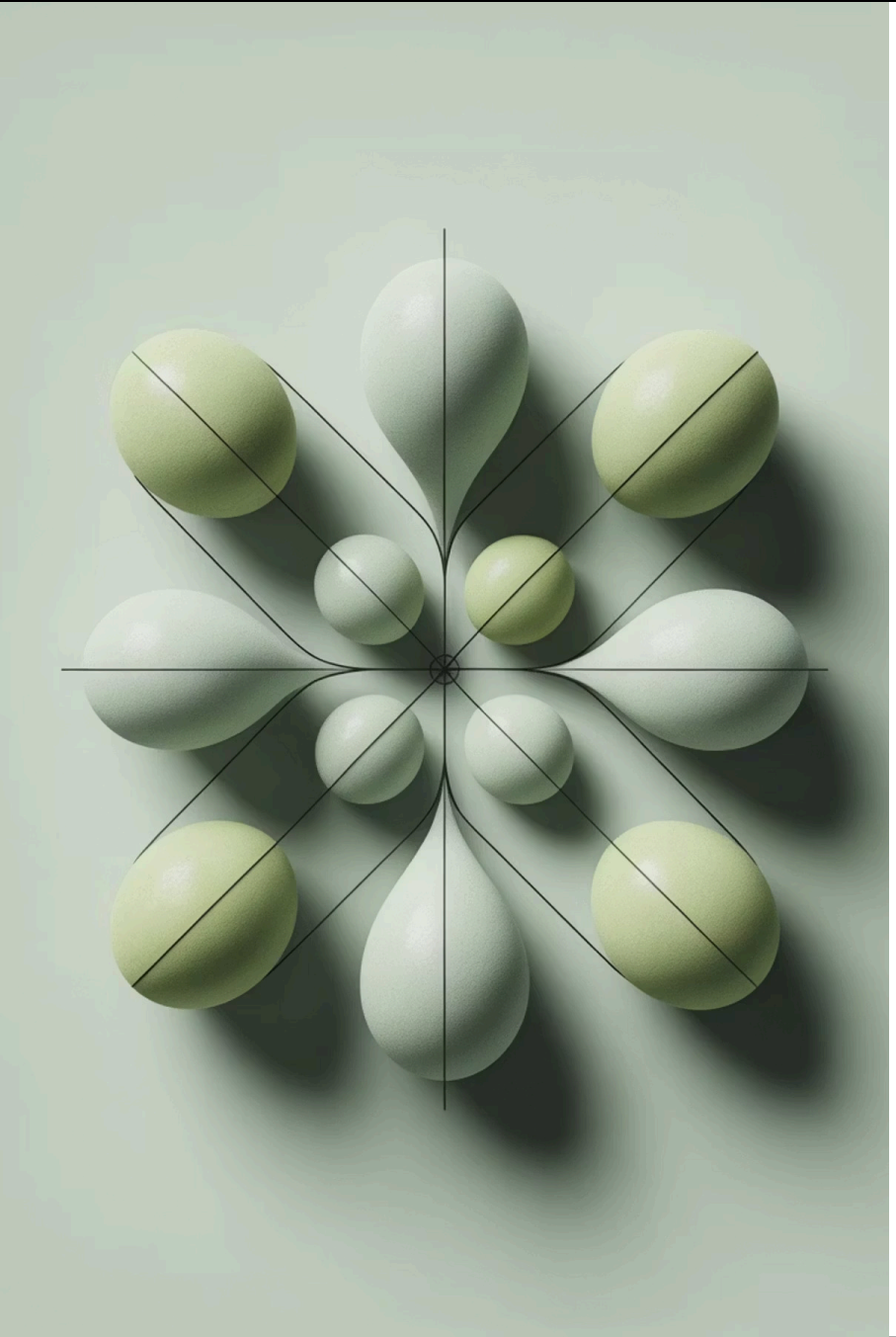
```
public class Movie {
  private int id;
  private String title;
  private String poster;
  private double rating;

  @SerializedName("releaseDate")
  private String releaseDate;
  private String overview;

  public Movie(int id, String title, String poster,
               double rating, String releaseDate, String
overview) {
    this.id = id;
    this.title = title;
    this.poster = poster;
    this.rating = rating;
    this.releaseDate = releaseDate;
    this.overview = overview;
  }
  // Getters & Setters ici...
}
```

L'annotation **@SerializedName** permet de mapper des noms JSON différents vers des propriétés Kotlin avec un naming plus conventionnel.





Étape 4 : Créer l'instance Retrofit

La configuration de Retrofit suit le pattern Singleton pour garantir qu'une seule instance existe dans toute l'application. Cela optimise les ressources et assure une gestion cohérente des connexions réseau.

01

Définir la Base URL

L'URL de base de votre API REST (ex: "https://api.example.com/")

02

Ajouter le convertisseur

GsonConverterFactory pour la conversion JSON automatique

03

Builder l'instance

Utiliser Retrofit.Builder() pour créer l'instance finale

Exemple de la classe "client Retrofit"

```
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class RetrofitClient {
    private static final String BASE_URL = "https://jsonplaceholder.typicode.com/";
    private static Retrofit retrofit = null;

    public static ApiService getApiService() {
        if (retrofit == null) {
            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create()) // JSON → Java
                    .build();
        }
        return retrofit.create(ApiService.class);
    }
}
```

Cette configuration crée une instance Retrofit robuste.

Étape 5 : Interface de l'API (définir les endpoints)

L'interface ApiService utilise des annotations Retrofit pour définir les endpoints de votre API de manière déclarative. Chaque méthode correspond à un appel réseau spécifique.

Exemple d'interface complète

```
import java.util.List;
import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.DELETE;
import retrofit2.http.GET;
import retrofit2.http.POST;
import retrofit2.http.Path;

public interface ApiService {
    // GET /movies/popular
    @GET("movies/popular")
    Call<List<Movie>> getPopularMovies();

    // GET /movies/{id}
    @GET("movies/{id}")
    Call<Movie> getMovieDetails(@Path("id") int movieId);

    // POST /movies/favorite
    @POST("movies/favorite")
    Call<Void> addToFavorites(@Body Movie movie);

    // DELETE /movies/{id}
    @DELETE("movies/{id}")
    Call<Void> deleteMovie(@Path("id") int movieId);
}
```

Méthodes HTTP

@GET

Récupère des données depuis le serveur

@POST

Envoie de nouvelles données au serveur

@PUT

Met à jour des ressources existantes

@DELETE

Supprime des ressources du serveur

Modes d'envoi des paramètres :

@Path : Insère une valeur dynamique dans l'URL

@Query : Ajoute des paramètres à l'URL pour filtrer ou personnaliser la requête

@Field : Envoie des données dans le corps d'une requête POST, sous forme de formulaire (x-www-form-urlencoded)

@Body : Envoi d'un objet complet (JSON) - Sérialise automatiquement un objet Java en JSON (via Gson).

Étape 6 : Appel de l'API (ajouter dans la queue Retrofit)

```
public class MainActivity extends AppCompatActivity {
   ApiService apiService = ApiAdapter.retrofit.create(ApiService.class);
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState); setContentView(R.layout.activity_main); loadPopularMovies(); }
   private void loadPopularMovies() {
      apiService.getPopularMovies().enqueue(new Callback>() {
         @Override
         public void onResponse(Call> call, Response> response) {
            if (response.isSuccessful()) {
               List movies = response.body();
               // TODO: Update UI (RecyclerView, TextViews, etc.)
               Log.d("API", "Movies: " + movies.size());
            } else {
               Log.e("API", "Erreur : " + response.code());
            } }
         @Override
         public void onFailure(Call> call, Throwable t) {
            Log.e("API", "Erreur réseau : " + t.getMessage());
         }});} }
```

enqueue()

Lance l'appel API de manière asynchrone sans bloquer l'interface utilisateur

onResponse()

Méthode appelée automatiquement quand le serveur renvoie une réponse

onFailure()

Méthode exécutée en cas de problème réseau ou d'erreur serveur

Résumé et bonnes pratiques Retrofit

Sécurité et permissions

Permissions runtime

Demandez les permissions au moment opportun, expliquez leur nécessité à l'utilisateur avec des dialogues contextuels et pédagogiques.

Sécurisation des communications

Utilisez HTTPS exclusivement, implémentez le certificate pinning pour applications sensibles, chiffrez les données locales avec EncryptedSharedPreferences.

Gestion des tokens

Stockez les tokens d'authentification de manière sécurisée, implémentez un refresh automatique, et invalidez les sessions expirées.

Performance et optimisation

Caching intelligent

Configurez OkHttp cache pour réduire les appels réseau, définissez des stratégies de cache HTTP appropriées (max-age, must-revalidate).

Compression des données

Activez GZIP compression dans OkHttp, compressez les images avant upload, utilisez des formats modernes comme WebP.

Timeouts configurables

Ajustez les timeouts selon le type de requête : courts pour les requêtes légères, longs pour les uploads volumineux.

Tests exhaustifs

Utilisez MockWebServer pour simuler les réponses API, testez les cas d'erreur réseau, validez le comportement offline avec des tests instrumentés.

> 💡 Conseil Pro
>
> Créez une couche d'abstraction au-dessus de Retrofit avec des sealed classes pour représenter les états de réseau (Loading, Success, Error). Cela rend votre code plus testable, maintenable et facilite la gestion d'erreur centralisée dans l'UI.

Gestion des erreurs réseau

Une gestion robuste des erreurs est essentielle pour offrir une expérience utilisateur fluide même lorsque les choses ne se passent pas comme prévu. Retrofit et OkHttp fournissent plusieurs mécanismes pour détecter et gérer les erreurs.

200–299 : Succès

La requête a été traitée avec succès, les données sont disponibles

400–499 : Erreur client

Problème avec la requête (paramètres invalides, ressource introuvable)

500–599 : Erreur serveur

Le serveur a rencontré une erreur lors du traitement

```
apiService.getPopularMovies().enqueue(new Callback>() {
   @Override
   public void onResponse(Call> call, Response> response) {
      if (response.isSuccessful()) {  handleSuccess(response.body());
      } else if (response.code() == 401) {          showLoginScreen();
      } else if (response.code() == 404) {          showNotFoundError();
      } else if (response.code() >= 500) {          showServerError();
      } else {          showGenericError(response.code());
      }  }
   @Override
   public void onFailure(Call> call, Throwable t) {
      if (t instanceof IOException) {// Problème réseau : pas d'Internet, timeout, DNS, etc.
        showNetworkError("Vérifiez votre connexion Internet");
      } else {// Erreur inconnue (crash API, parsing JSON, etc.)
        showUnexpectedError(t.getMessage());
      }   }  });
```

Conclusion : Maîtriser les APIs Android avec Retrofit

Simplicité d'intégration

Retrofit transforme des opérations réseau complexes en appels de méthodes simples et lisibles. Son architecture basée sur les annotations réduit drastiquement le boilerplate code et accélère le développement.

Flexibilité architecturale

Compatible avec toutes les architectures modernes (MVVM, MVI, Clean Architecture), Retrofit s'intègre parfaitement avec Kotlin Coroutines, Flow, RxJava et s'adapte à vos choix technologiques.

Robustesse en production

Des millions d'applications en production font confiance à Retrofit. Sa gestion d'erreur mature, son système d'intercepteurs puissant et sa compatibilité avec OkHttp garantissent fiabilité et performance.

Perspectives d'avenir

La combinaison de Retrofit avec les APIs Android modernes ouvre des possibilités infinies : applications IoT connectant capteurs et cloud, solutions de réalité augmentée géolocalisée, systèmes de paiement mobile sécurisés, ou encore applications collaboratives en temps réel. Maîtriser Retrofit et l'écosystème Android vous positionne pour créer la prochaine génération d'applications mobiles innovantes.

> *"Le code que vous écrivez aujourd'hui avec Retrofit sera celui qui propulsera les applications de demain. Investissez dans la qualité, la sécurité et l'expérience utilisateur."*

Ressources pour aller plus loin

- Documentation officielle Retrofit : square.github.io/retrofit
- Android Developers Guide : developer.android.com
- Codelab Google Maps Platform

Expérience Utilisateur : Gestion du Chargement

01

Afficher un ProgressBar

Indiquez visuellement que l'application charge des données. L'utilisateur comprend qu'il doit attendre.

02

Gérer les Timeouts

Définissez un délai maximal (ex: 30 secondes). Si le serveur ne répond pas, affichez un message d'erreur clair.

03

Feedback sur les Erreurs

En cas d'échec (pas de connexion, erreur serveur), proposez à l'utilisateur de réessayer ou de consulter des données en cache.

Une bonne gestion de l'attente et des erreurs transforme une expérience frustrante en une interaction fluide et professionnelle.

5.4 Pourquoi le Multitâche est Essentiel ?

Le Thread Principal

Le **Thread Principal** (UI Thread) est le cœur vivant de votre application Android. Il gère l'affichage, les interactions utilisateur, les clics sur les boutons et toutes les mises à jour visuelles de l'écran.

Si vous effectuez un travail long sur ce thread, l'application se fige complètement et l'utilisateur voit apparaître le redoutable message ANR (Application Not Responding).

Appels réseau vers une API

Télécharger des données depuis un serveur distant peut prendre plusieurs secondes selon la connexion

Accès à la base de données

Les requêtes complexes sur de grandes bases de données locales nécessitent du temps de traitement

Traitement de fichiers volumineux

Lire ou écrire des images, vidéos ou documents lourds bloque l'exécution

Comment Android Gère le Multitâche

Android offre un système sophistiqué de gestion des threads pour maintenir votre application fluide et réactive, même pendant l'exécution de tâches complexes en arrière-plan.

1. **Action utilisateur**
   L'utilisateur clique sur un bouton pour lancer le chargement des données
2. **Thread en arrière-plan**
   L'appel API s'exécute dans un thread séparé, sans bloquer l'interface
3. **Retour sur le Thread Principal**
   Les données téléchargées reviennent automatiquement sur l'UI Thread pour mise à jour
4. **Affichage actualisé**
   Le RecyclerView ou l'interface se met à jour avec les nouvelles informations

La Magie de Retrofit

Avec Retrofit, vous n'avez pas besoin de créer ou gérer manuellement les threads. La bibliothèque s'occupe automatiquement de lancer l'appel réseau dans un thread séparé et de retourner le résultat sur le Thread Principal.

Cette abstraction permet aux développeurs débutants de se concentrer sur la logique métier plutôt que sur la complexité du multithreading Android.

Les Appels Asynchrones avec Retrofit

Appel asynchrone

La méthode `enqueue()` lance l'opération réseau sans bloquer l'UI

Réponse réussie

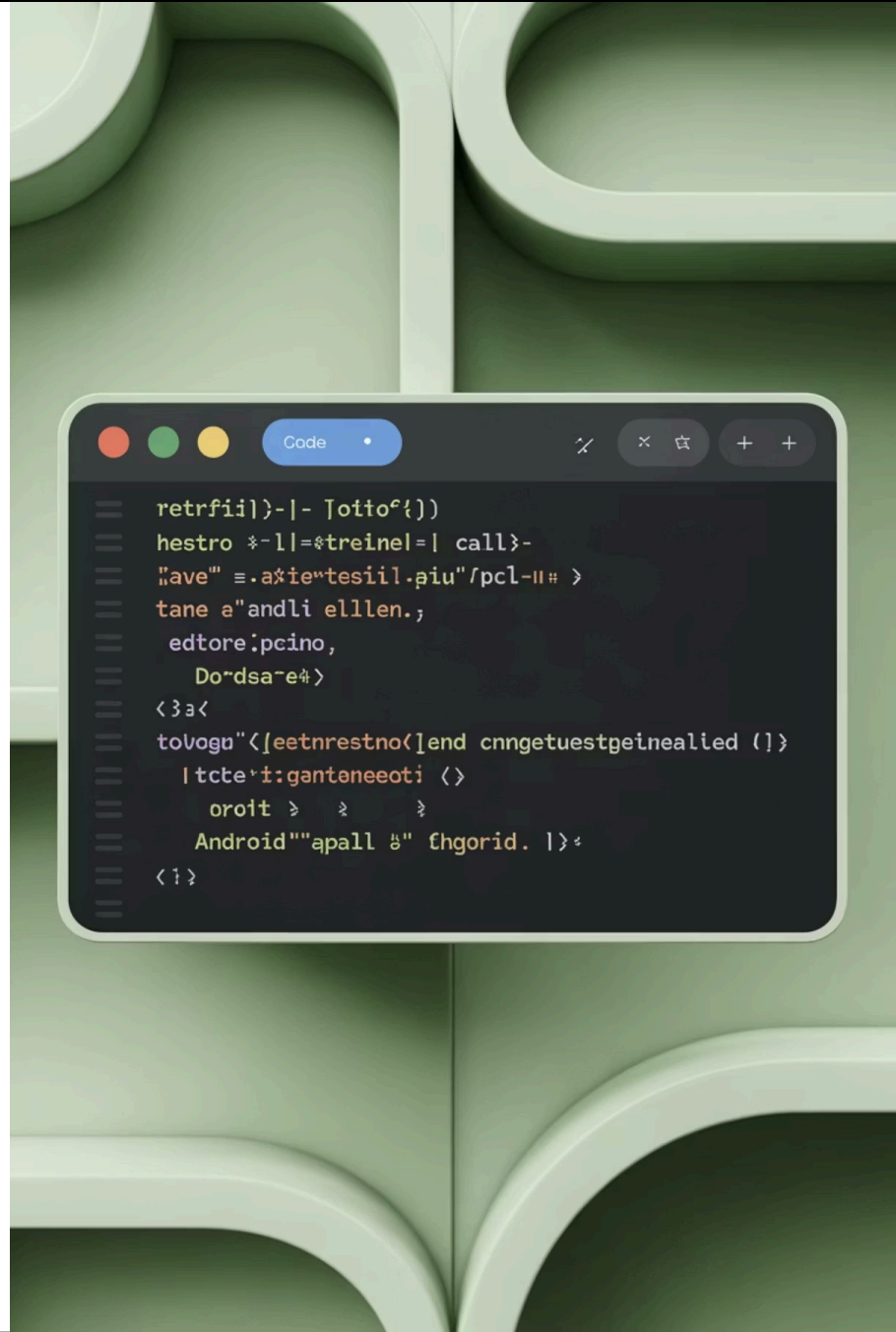
`onResponse()` reçoit les données quand le serveur répond

Gestion d'erreur

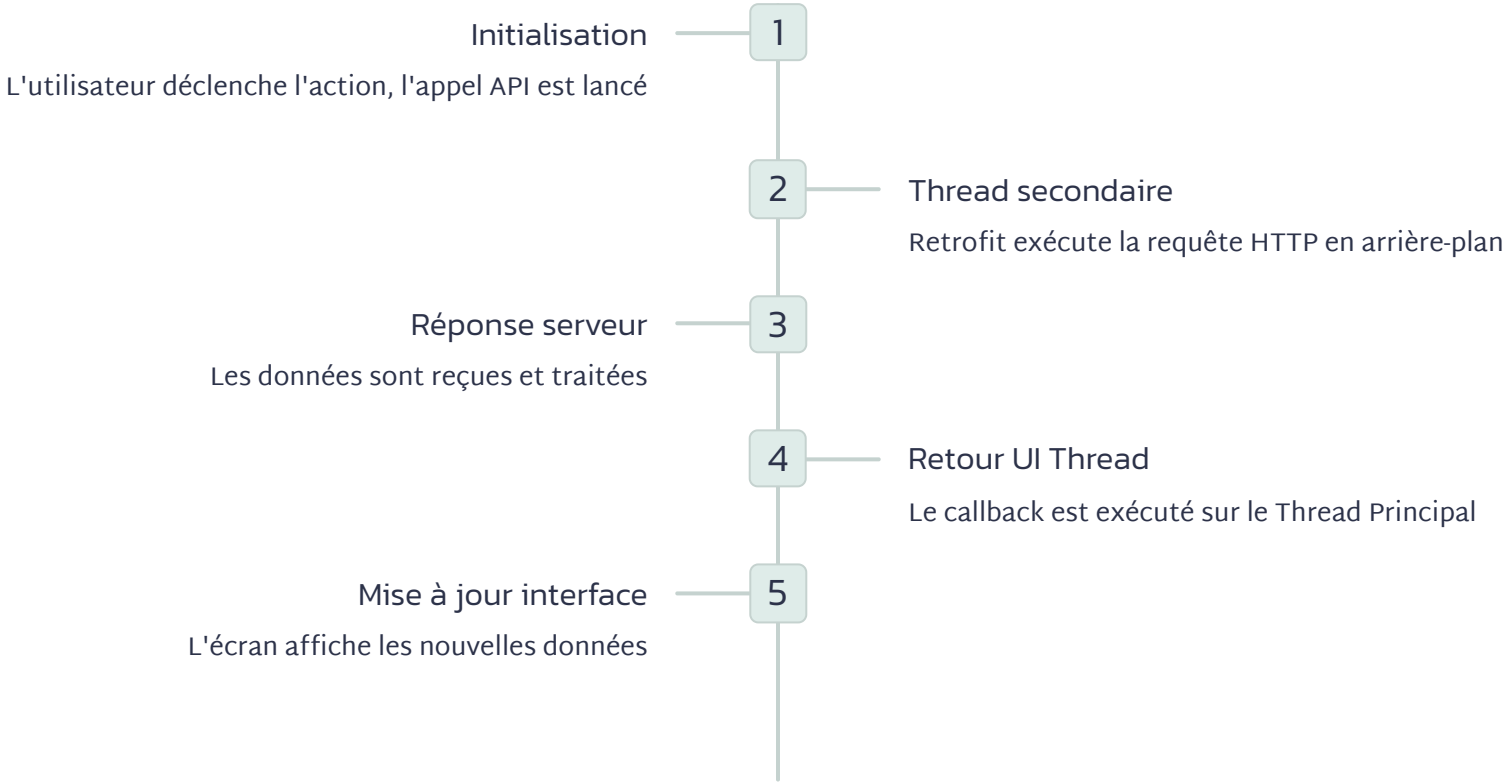
`onFailure()` capture les problèmes réseau ou serveur

Le pattern de **callback** utilisé par Retrofit est au cœur du multitâche en Android. Quand vous appelez `enqueue()`, Retrofit crée automatiquement un thread secondaire pour effectuer la requête HTTP. Pendant ce temps, votre application continue de fonctionner normalement.

Une fois que le serveur répond, Retrofit bascule automatiquement sur le Thread Principal pour exécuter votre callback `onResponse()` ou `onFailure()`. Vous pouvez donc mettre à jour l'interface utilisateur directement dans ces méthodes en toute sécurité.



Le Flux d'Exécution Asynchrone



Ce flux garantit que votre application reste réactive à tout moment. Pendant que les données sont téléchargées, l'utilisateur peut continuer à interagir avec l'interface, faire défiler du contenu ou même naviguer vers d'autres écrans.

L'architecture asynchrone de Retrofit transforme ce qui pourrait être un processus bloquant en une expérience utilisateur fluide et moderne.

Stratégies de Gestion des Erreurs

Toast pour erreurs mineures

Utilisez Toast.makeText() pour des messages brefs et non bloquants qui disparaissent automatiquement après quelques secondes.

TextView pour erreurs persistantes

Affichez un message dans un TextView qui reste visible jusqu'à ce que l'utilisateur réessaie ou change d'écran.

Dialog pour erreurs critiques

Présentez une boîte de dialogue avec options (réessayer, annuler) pour les situations nécessitant une action utilisateur.

Écran d'erreur complet

Pour les échecs majeurs, basculez vers un écran dédié avec illustration et bouton de nouvelle tentative.

Le choix de la méthode dépend de la criticité de l'erreur et de l'impact sur l'expérience utilisateur. Une erreur de chargement d'image peut justifier un simple Toast, tandis qu'un échec de connexion initial mérite un écran dédié.