

Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



# Clean Architecture en Flutter ❤ — ¿Cómo implementarla en el 2023?



Daniel Herrera Sánchez · Follow

Published in Bancolombia Tech

13 min read · Jan 21, 2023

Share

••• More

Hace un tiempo atrás, hablamos de arquitectura limpia en Flutter. Desde entonces ha cambiado la tecnología, los paquetes y algunos conceptos se refinaron. Por ello, en este artículo te hablaré de cómo puedes implementarla este año.



Dart menor nos explica arquitectura limpia

Antes de comenzar, si sientes que ya conoces los conceptos básicos de arquitectura limpia, puedes continuar al siguiente tema, allí empezaremos a ver a profundidad

## Clean Architecture aplicado en Flutter.

### Arquitectura de Software

Es la parte del desarrollo de soluciones que permite estructurar nuestros proyectos y definir los lineamientos que tendremos que seguir en durante la implementación de los componentes de nuestra solución. Su principal objetivo es que el desarrollo sea **fácil de implementar, operar y mantener**.

Cuando logramos diseñar una buena arquitectura de software, a nuestra solución le será fácil adaptarse al cambio de forma oportuna sin alterar su estabilidad.

También, la arquitectura de nuestra solución debería buscar la independencia entre sus diferentes capas. Por ejemplo, si en la evolución de nuestra solución, debemos cambiar la base de datos, una API o cualquier otro componente de nuestro desarrollo; esto no debería afectar la capa de presentación. La arquitectura solo deberá verse afectada cuando cambian de manera drástica las reglas del negocio.

### Clean Architecture no es tu única alternativa

Antes de entrar en el detalle de esta forma de diseñar tus arquitecturas; te quiero comentar que: *Clean Architecture no es una bala de plata*; es decir, no estás obligado, ni es necesario aplicarla a todas tus soluciones. No obstante, **todas tus soluciones deberían tener una buena arquitectura de software**.

En el caso de Flutter recomiendo analizar en cada escenario, cuál tipo de arquitectura se ajusta más a tus necesidades. Existen muchas podemos utilizar como:

- Arquitectura hexagonal (también conocida como puertos y adaptadores), desarrollada por Alistair Cockburn y adoptada por Steve Freeman y Nat Pryce en su maravilloso libro *Growing Object Oriented Software with Tests*.
- DCI de James Coplien y Trygve Reenskaug
- BCE presentada por Ivar Jacobson de su libro Software orientado a objetos.

En otro artículo profundizaremos más sobre el tema de selección de arquitecturas.

### Clean Architecture

Si traducimos en Traductor de Google: *Clean Architecture*, nos arroja lo siguiente:

DETECTOR IDIOMA [INGLÉS](#) [ESPAÑOL](#) [FRANCÉS](#)

[ESPAÑOL](#) [INGLÉS](#) [FRANCÉS](#)

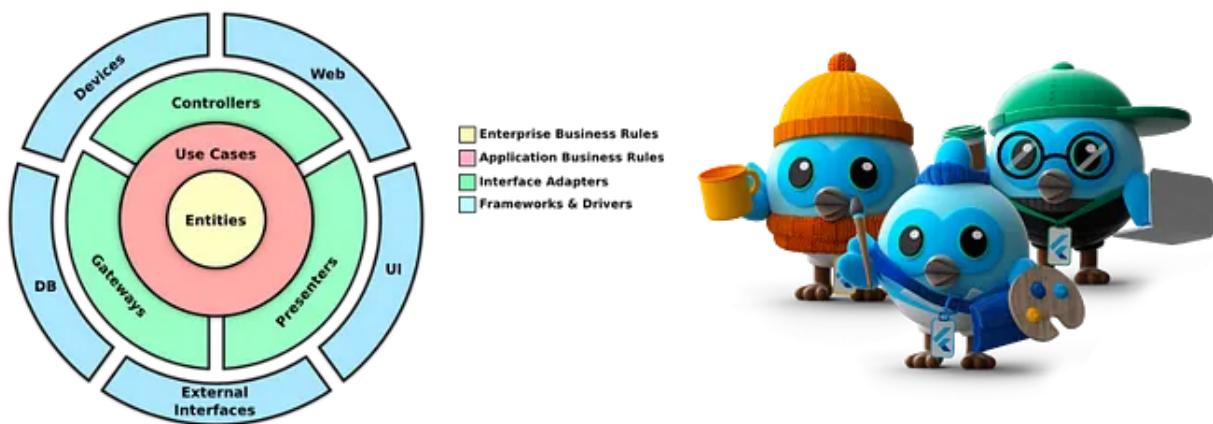
Clean Architecture ×

Traducir del: [español](#)

182 / 5.000

110% real, no fake 🎉🎉

En efecto: Es un arquitectura de software diseñada por Robert C. Martin, que consiste en conjunto de capas bien definidas, las cuales están centradas en el dominio (el negocio y sus reglas). Puedes encontrar su libro acá: [Clean Architecture: A Craftsman's Guide to Software Structure and Design, First Edition.](#)



El diagrama anterior es el más común a la hora de hablar de Arquitectura limpia. Este nos indica cómo, el Tío Bob (apodo de Robert), describe que debemos dividir nuestras soluciones en su libro. Este parte desde las capas del negocio como, entidades y casos de uso; para luego llegar a la capa de presentación.

Es importante que sepas que el Tío Bob nos enseña en su libro algunos conceptos sujetos a su interpretación. Por lo tanto, podemos deducir que, **no existe una única forma de implementar Clean Architecture**. Lo que sí podemos afirmar, es que **busca la segregación de responsabilidades y centralizar nuestra solución en el dominio**.

### ¿Qué implica centralizar una solución en el dominio?

El dominio es la capa definida por el **negocio**, es decir, no debe estar atada a ningún concepto técnico y dentro de ella vemos las entidades y los casos de uso.

Por ejemplo, pensemos en una solución como Spotify. **El equipo de negocio debe definir lo que quiere**, es decir, es el negocio quién decide cuáles son las entidades relevantes y sus casos de uso como:

- Un usuario accederá a la app para escuchar música.
- Deberá registrarse y autenticarse.
- Podrá crear listas reproducción propias.
- Tendrá la opción de darle me gusta a una canción.
- Navegará a través de la aplicación buscando por artista, álbum, canción, o podcast, etc.

Además de las funcionalidades que se agregarán, el equipo de negocio también deberá definir las reglas de la aplicación, como si el mismo usuario puede ser artista y consumidor al mismo tiempo o deberá crear dos perfiles distintos.

Por lo tanto, tiene mucho sentido decir que **debes centrar las soluciones en el dominio**. Para una correcta definición de entidades y **casos de uso** se hace muy útil realizar sesiones de Domain Driven Design. Te dejo un artículo que entra en detalle en el tema ([enlace](#)).

Pero, ¿qué son las entidades y casos de uso?

## Entidades

# Capa de Dominio



Son los objetos comerciales de la aplicación. Encapsulan las reglas más generales y de alto nivel. Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, estos objetos no deberían verse afectados por un cambio en la navegación o la seguridad de la página. Ningún cambio operativo en ninguna aplicación en particular debería afectar la capa de entidad.

## Casos de uso

# Capa de Dominio



El software en la capa de casos de uso contiene reglas comerciales específicas de la aplicación. Allí se encapsulan e implementan todos los casos de uso del sistema. Estos casos de uso organizan el flujo de datos hacia y desde las entidades, además, ordenan a esas entidades que usen sus reglas comerciales críticas para lograr los objetivos del caso de uso.

En esta capa no esperamos que los cambios afecten a las entidades y tampoco esperamos que esta capa se vea afectada por cambios en las externalidades como: la base de datos, la interfaz de usuario o cualquiera de los marcos comunes.

La capa de casos de uso deberá estar aislada de tales preocupaciones. Por eso sería una mala práctica que un caso de uso se nombre: *ApacheSMSSender*; este debería llamarse: *NotificationsSender*.

Sin embargo, esperamos que los cambios en el funcionamiento de la aplicación sí afecten los casos de uso y, por lo tanto, su software. Si los detalles de un caso de uso cambian, algo de código en esta capa se verá afectado.

Teniendo esto en mente, nos queda faltando saber a qué capa pertenece el software destinado a conectarnos con las APIs y la capa de presentación. A partir de este momento comienza la interpretación. Con esto me refiero a que la forma de agrupar o desacoplar estos elementos puede ser diferente siempre y cuando no vulnere las definiciones establecidas.

## **Capa de Infraestructura**

# Capa de Infraestructura



Contiene los siguientes elementos:

- **Driven adapters**: serán los adaptadores los que nos permitirán conectarnos hacia el exterior, conforme a las necesidades que tengamos. En este tendremos todos nuestros repositorios de datos y escritura a estos.
- **Entry points**: esta capa es cuando exponemos servicios. Para el mundo front-end no haremos uso de esta. Es común encontrarla en una solución back-end.
- **Helpers**: es una capa dedicada a ayudar a sus hermanos (miembros de la capa de infraestructura) con transformaciones, operaciones, funciones útiles para la capa de infraestructura.

## Capa de presentación

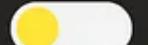
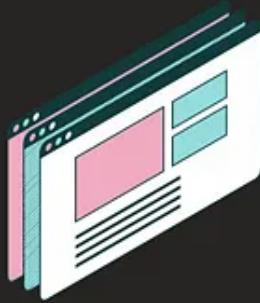
Open in app ↗



Search



# Capa de Presentación



MENU



+ CREATE



Implementa todo el software relacionado a los elementos visuales con los que el usuario final interactuará.

En algunas ocasiones cuando hablo del tema me suelen preguntar si es bien tener una capa llamada *Widgets* para los elementos visuales reutilizables dentro de la aplicación. Para este tipo de componentes sugiero que mejor construyas tu sistema de diseño, te dejo un artículo en el que hago un mayor acercamiento a esta idea ([enlace](#)).

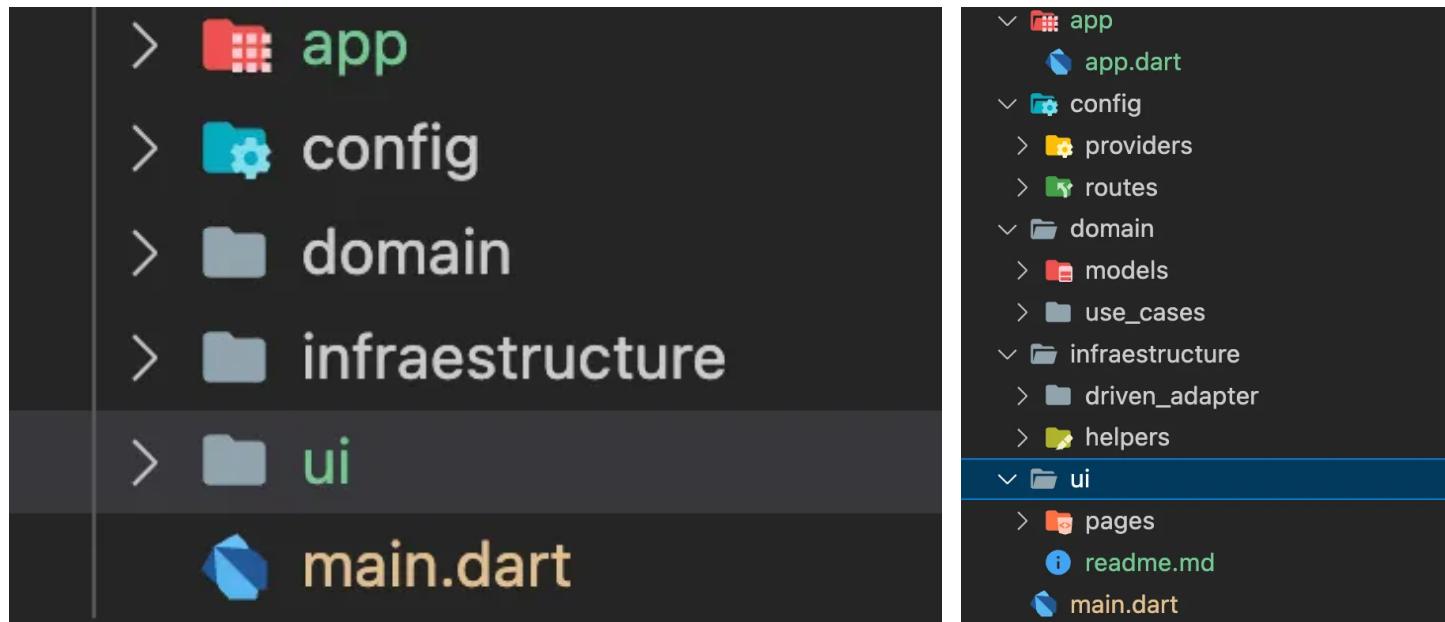
## Implementemos arquitectura limpia en Flutter

Bueno ahora sí llegó la hora de la verdad. Vamos a aplicar lo aprendido, pero antes te debo comentar algo muy importante: existen distintos niveles de desacoplos de este tipo de arquitectura. Vamos a ver sus variantes más conocidas.

### Primera Forma: Segregación por carpetas

Como vimos en la teoría, contamos con tres capas principales: dominio, infraestructura y presentación. Podemos separar estas capas, utilizando diferentes estrategias.

La primera que analizaremos es separar las capas por carpetas, algo como lo siguiente:



Estructura arquitectura limpia separada por carpetas

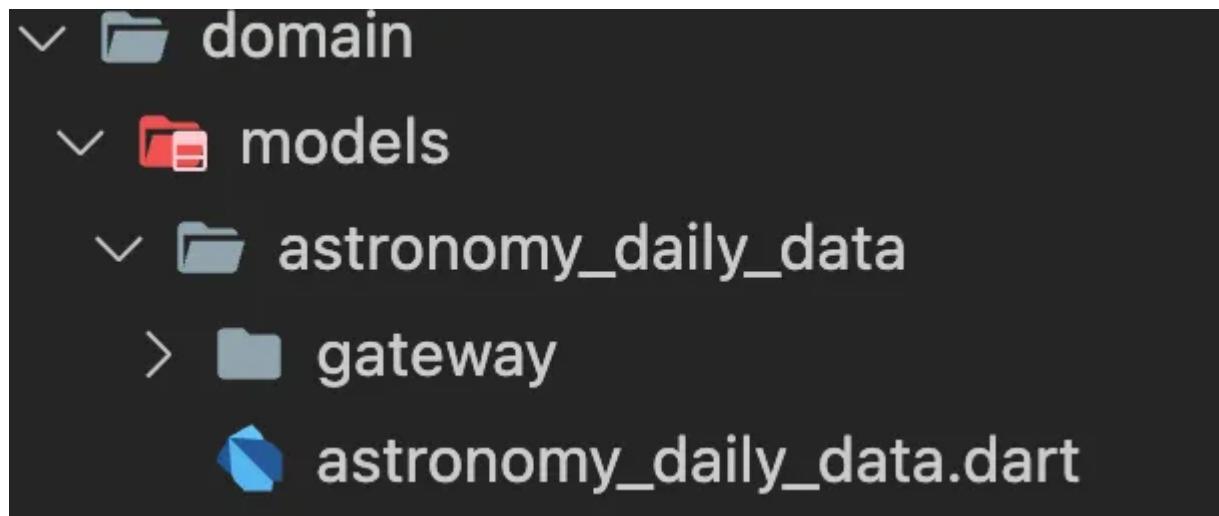
Como vemos tenemos carpetas por cada una de las capas, sin embargo su contenido debe respetar los principios de la arquitectura limpia. Por ejemplo, **bajo ningún motivo la capa de dominio dependerá de infraestructura**, tampoco la capa de presentación dependerá de ella. Logrando así un **desacople en la solución**.

Podemos considerar la capa de Configuración como una capa más externa que nos ayudará con las configuraciones de nuestra aplicación.

Apliquemos esto a un ejemplo sencillo para que podamos interiorizar su implementación. Para ello emplearé de ejemplo la siguiente API, la cual devuelve un dato de astronomía diferente cada día: <https://go-apod.herokuapp.com/apod>.

Para el ejemplo utilizaré *Riverpod* como gestor de estados y la inversión de dependencias (ahora mas adelante veremos su importancia).

## Capa de Dominio



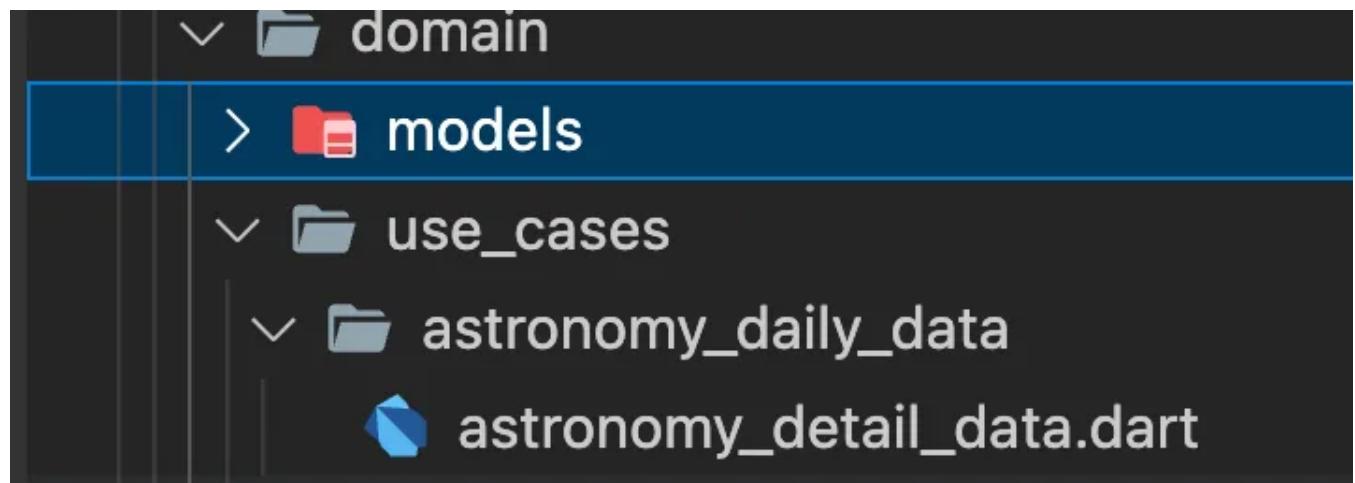
Comencemos creando nuestra identidad:

En este caso creamos la identidad *AstronomyDailyData*. Los atributos principales de una identidad deben salir de una sesión de *Domain Driven Design* con el negocio.

Un siguiente punto a crear es una clase abstracta que encapsule la definiciones más no la implementación. En mi caso los suelo llamar *gateway*, sin embargo otros lo pueden llamar *repository* o de alguna otra forma. Lo importante es que cuentes con este tipo de clase para que dependas de la abstracción mas no de la implementación. Esto es muy útil para mantener la estabilidad de tu código pese al cambio de las externalidades.

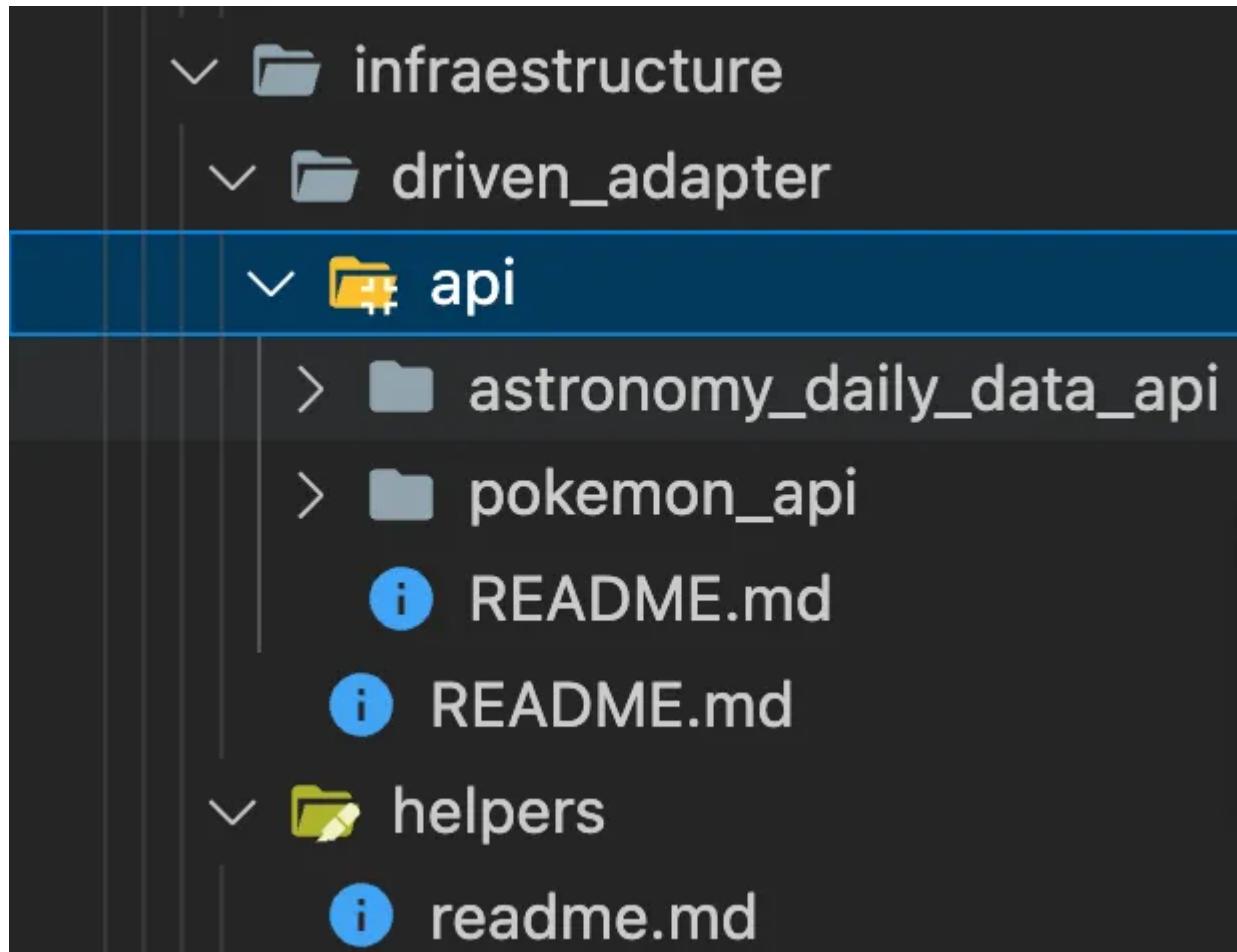
Lo que indica esta clase es que deseamos que, cuando alguien ejecute la acción *getAstronomyDayliData* pueda obtener un futuro del tipo *AstronomyDailyData*. Esto

permite que el caso de uso pueda estructurarse de la siguiente forma:



Como vemos, el caso de uso depende de la abstracción es decir del *gateway* y en ningún momento de la implementación. Esto permite que más adelante podamos crear una clase que extienda de *AstronomyDailyDataGateway* para que implemente la lógica necesaria para obtener la información esperada. También, permite que ante un cambio en la implementación la capa de dominio no se vea afectada.

### Capa de Infraestructura

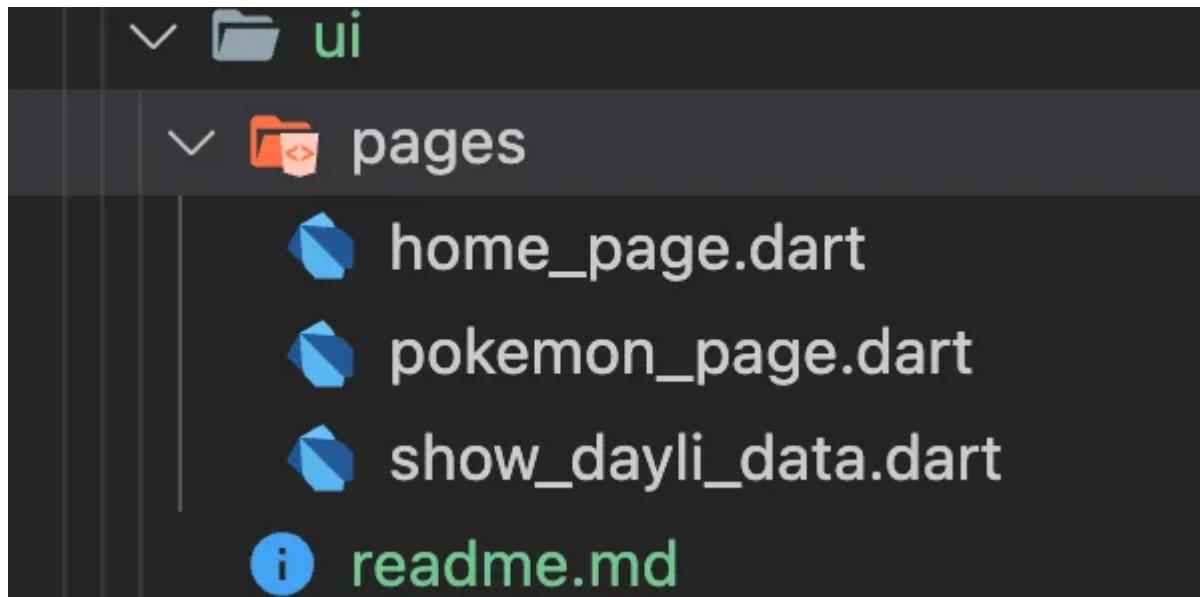


Como vimos en la teoría; en esta sección nos conectaremos con los elementos externos. Par el caso del ejemplo, en esta capa tendríamos la conexión con la API que obtendrá la información deseada:

Como vemos, la clase que implementa el caso de uso se extiende del *gateway* de forma que implementa los métodos del mismo. En este escenario, toma la información de un servicio.

Lo interesante es que no importa la forma en que obtengamos los datos (la implementación), podremos obtener como resultado la información deseada y si cambia la implementación no se ve afectada la capa de dominio.

## Capa de Presentación



En la capa de Presentación solo debe existir código responsable de dibujar elementos en pantalla. Por ejemplo, veamos la implementación de la pantalla que muestra el dato diario:

Como podemos observar, la capa de Presentación no depende de infraestructura ni de una implementación del caso de uso. Sencillamente, espera que de alguna forma se le inyecte la información que necesita. En este escenario un:

*Future<AstronomyDailyData>.*

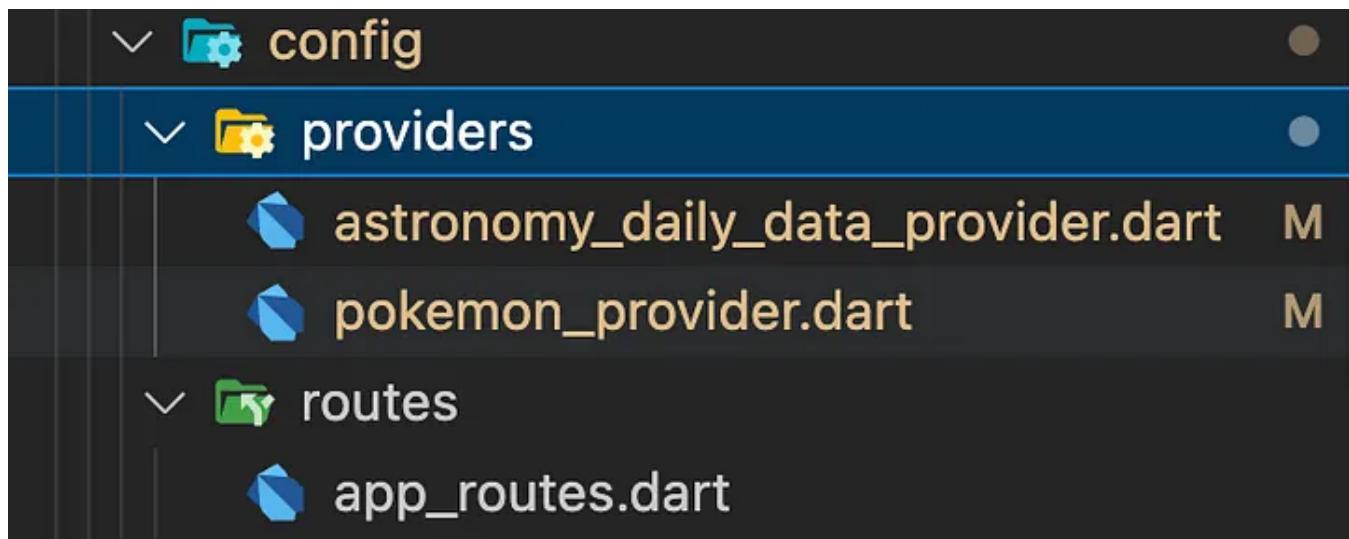
Si revisamos en detalle, tenemos algunos elementos *front* que provienen de la librería *design\_system\_weincode*. Como sugerencia, cuando tengas elementos reutilizables *front*, podrías crear una librería con los mismos, la cual te será útil para esta y otras aplicaciones ([enlace](#)).

### **Inversión de dependencias**

En este momento la capa de presentación no tiene acceso a la información que necesita y tampoco tenemos relacionado quién implementa el caso de uso.

Al comienzo del ejemplo comenté que haría uso de *riverpod*. Vamos a crear un *provider* que nos ofrezca el caso de uso *AstronomyDailyDataUseCase*.

En este caso la variable *AstronomyDailyDataProvider* nos está diciendo que, el caso de uso será resuelto por el *AstronomyDailyDataApi*. Esto es algo útil para una capa de Configuración aplicativa, porque si quisiéramos modificar quién implementa el caso de uso, no se debería ver afectada ni la capa de Presentación ni mucho menos la capa de Dominio. Por lo tanto, decidí ingresar los *providers* en una capa externa que denominamos Configuración.

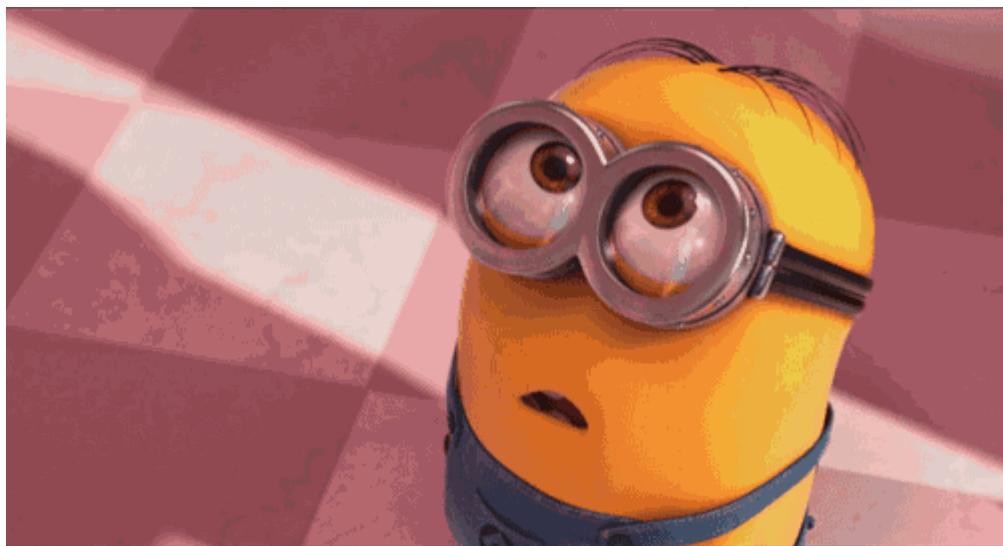


Ahora solo nos falta configurar que desde nuestra aplicación hagamos uso de los *providers*, de la siguiente manera:

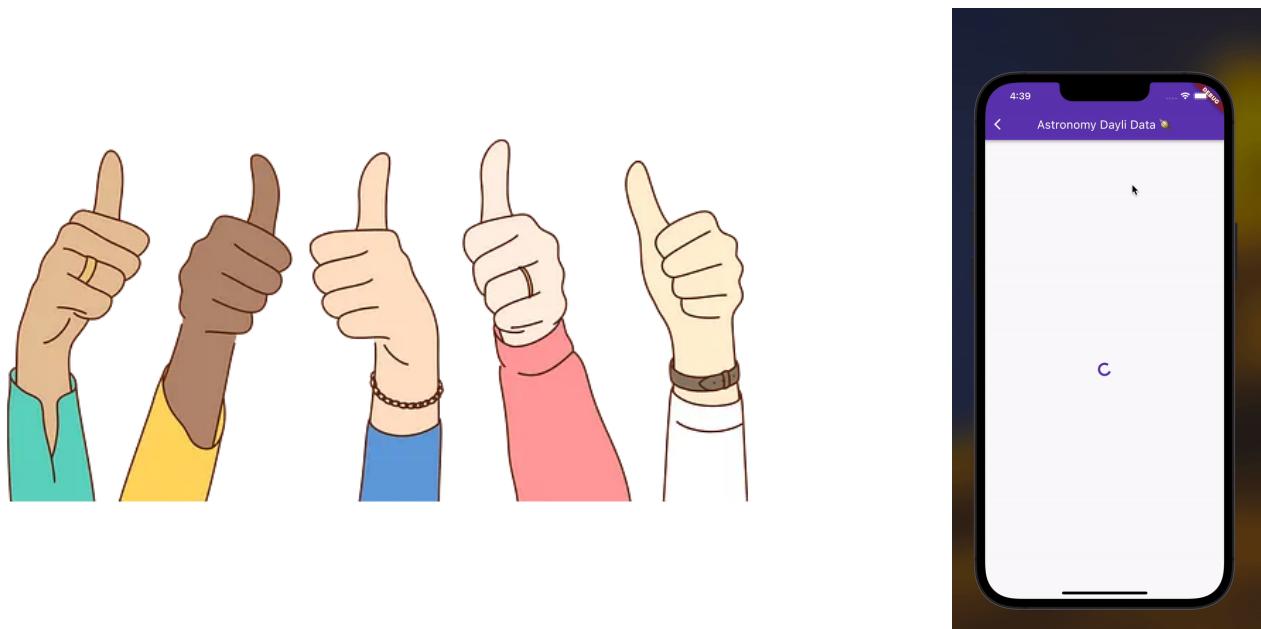
Como vemos, desde la definición de nuestro *MaterialApp* esta es la capa más externa aplicativa. En este sitio es donde podremos inyectarle a la capa de presentación lo que necesita para dibujar.

**Nota:** ¿Qué pasa si cambia el API?, ¿qué parte del proyecto cambia?

Solo sería necesario cambiar una línea de código en la capa de Configuración, todo lo demás sigue de la misma forma. No cambia ni la capa de Presentación, ni la de Dominio. Únicamente se agrega la nueva implementación en infraestructura y se inyecta en la capa de configuración.



Si ejecutamos la aplicación obtenemos lo siguiente :



Para este ejemplo construí la conexión con dos API bajo arquitectura limpia. Espero que puedas mirar en detalle la solución. Te comarto el [enlace](#) al repositorio en GitHub (recuerda darle estrellitas por gratitud al contenido).

**Reto:** Haz un *Fork* al proyecto y realiza la implementación para una tercera API.

### **Segunda Forma: Segregación de las capas por librerías**

Sé que el artículo está largo, pero en ocasiones nos toca prolongarnos en estos temas. Espero estés disfrutando y aprendiendo mucho. 😊🔥❤️

Para comenzar, entendamos por qué nace la necesidad de segregar las capas por librerías. En proyectos grandes, constantemente ingresan nuevos integrantes al equipo y es necesario mantener la calidad de software. Por lo tanto, es necesario

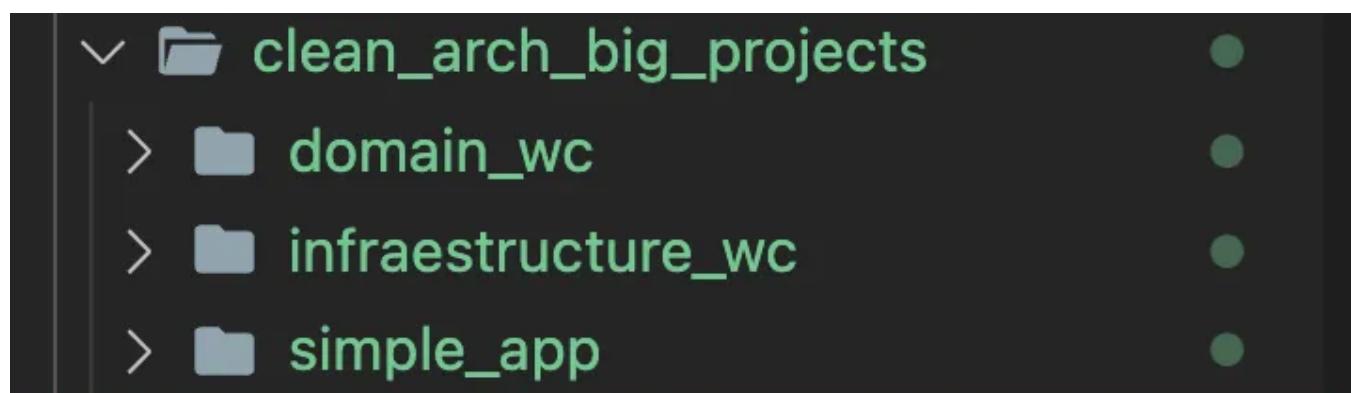
hacer que sea fácil de entender la arquitectura del proyecto para un nuevo desarrollador que se una.

Ante esto surge algunas inquietudes: ¿te imaginas a alguien junior teniendo que intervenir todas las capas del proyecto?, ¿qué tan fácil sería equivocarse o frustrarse?

Para enfrentarnos a esta posibilidad sería más fácil asignar a nuestros desarrolladores junior intervenir solo la capa de Presentación, comunicándoles qué información van a dibujar en pantalla. Por el otro lado, delegaríamos a otros desarrolladores más experimentados la capa de Dominio y la de Infraestructura. Lo anterior es una exageración pero podría suceder.

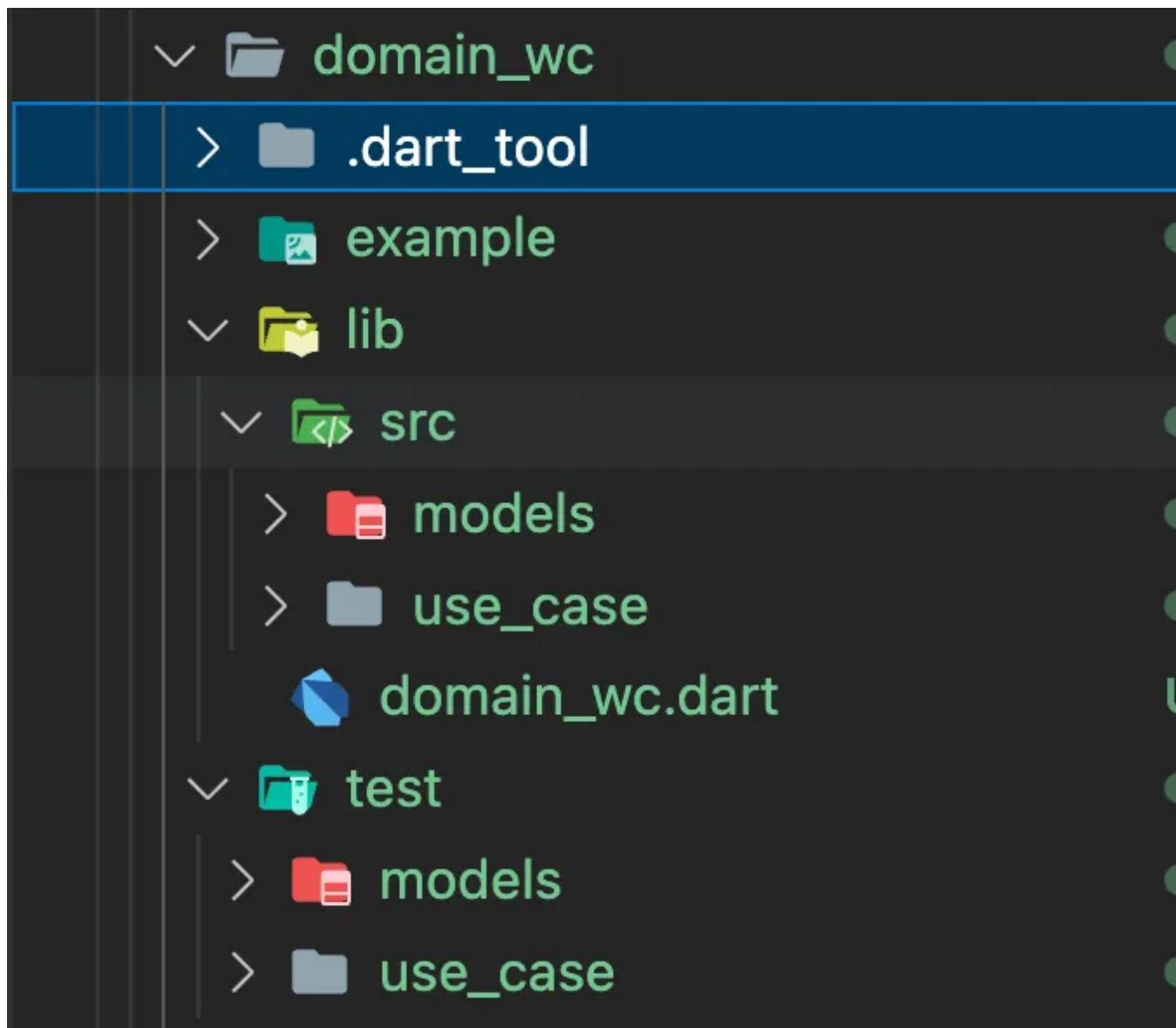
Por otro lado, piensa que tienes que realizar un cambio en tu capa de Presentación. ¿Por qué se ejecutan las pruebas unitarias de tu capa de Dominio, si solo modificaste la de Presentación? lo mismo aplica si solo modificaste la de Infraestructura. Teniendo en el radar este y otros criterios algunos equipos podrían pensar en separar su aplicación por librerías.

Para este escenario tomé la API de *pokemons* y le apliqué la metodología.



Para no sonar redundante solo explicaré los puntos que cambian versus la anterior forma.

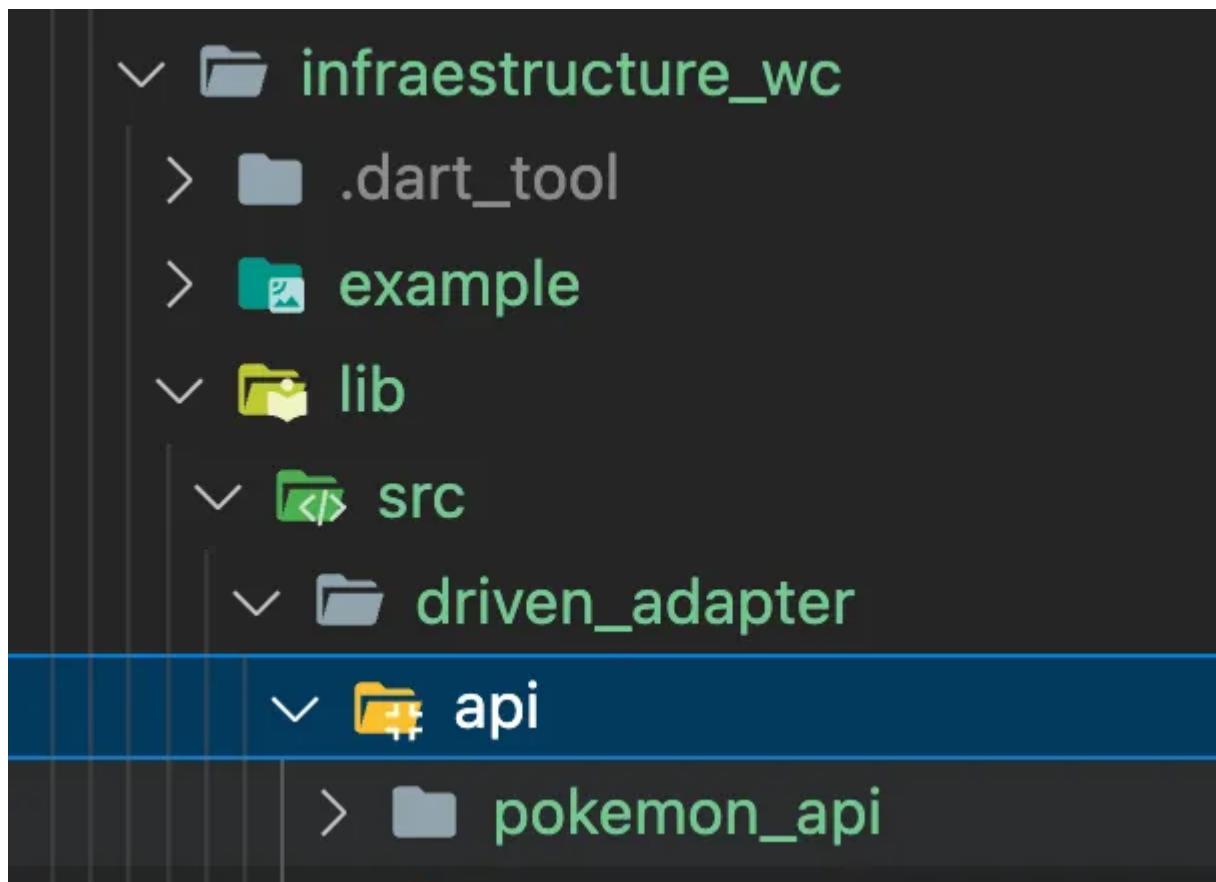
### Capa de dominio



Como ves, en este caso, la capa de Dominio es una librería *dart*. Esto indica que no tiene porque depender de ninguna de las clases del SDK de Flutter. Tiene sus propias pruebas. Y en el *pubspec.yaml* se configura para que sea publicado como paquete en *pub.dev*:

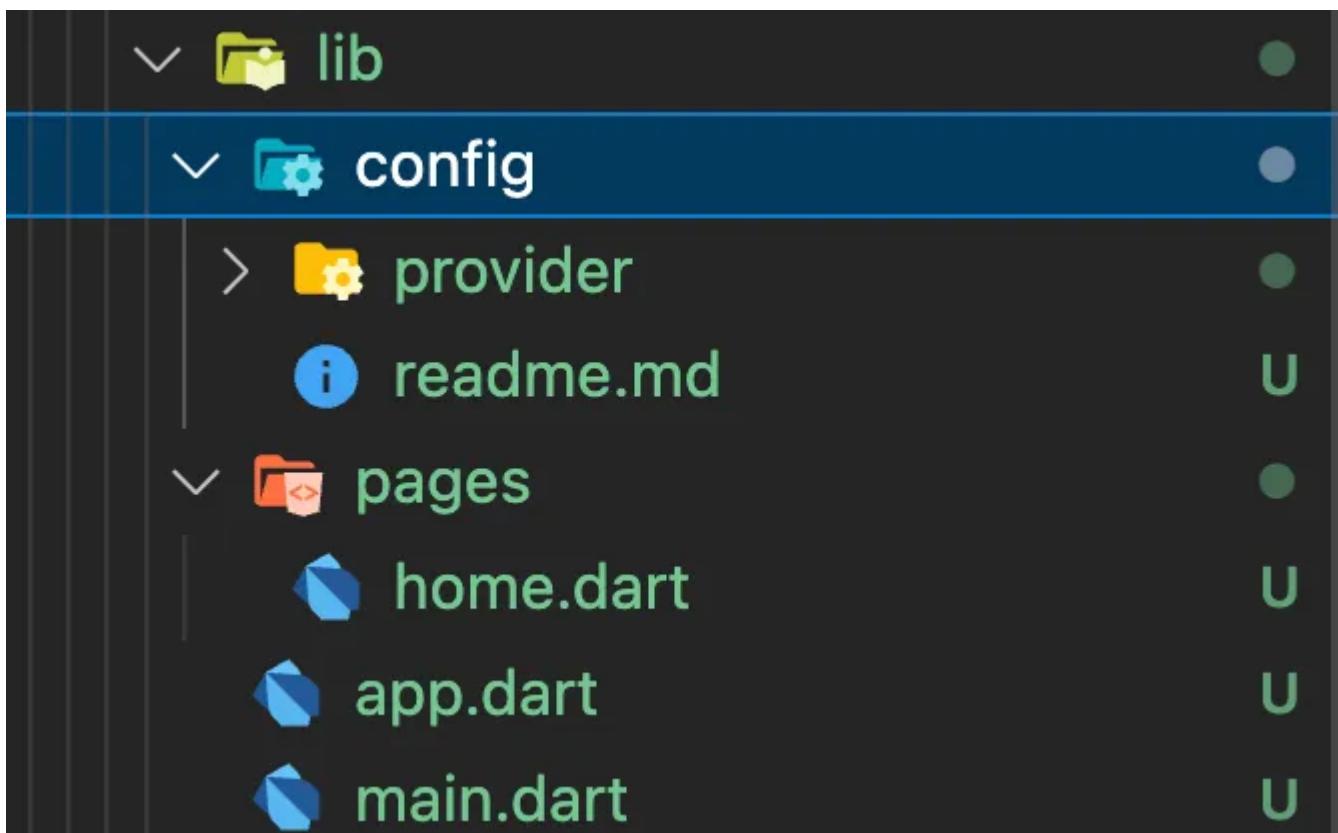
Al ser el proyecto central, será el primero que salga a producción, pues todos los demás dependen de él. Pero Daniel, ¿qué hago si quiero que mis reglas de negocio no sean públicas? Puedes publicar el paquete en la versión empresarial de *pub.dev*. Depender de la librería a través de un repositorio privado de GitHub o hasta importar con la ruta relativa si está en el mismo repositorio.

## **Capa de Infraestructura**



Al igual que el anterior, es una librería *dart* en la cual configuramos todas las conexiones contra terceros. También debe definirse un correcto método de publicación.

### Capa de Presentación



Como era de esperarse en este caso, la aplicación final tendrá código relacionado a dibujar información al cliente final. De forma que, la aplicación resultante se torna más sencilla de soportar y administrar. Esto sirve sobre todo para grandes proyectos.

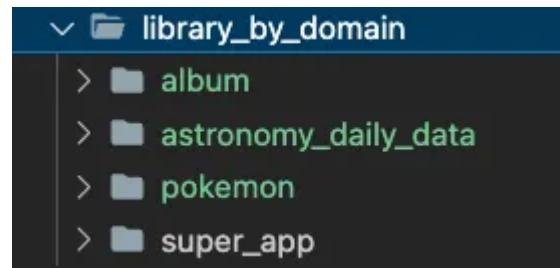
**Dato curioso:** en este método la capa de presentación no debería tener dependencias de conexión con terceros como la librería http.

Te dejo el [enlace](#) para el ejemplo completo.

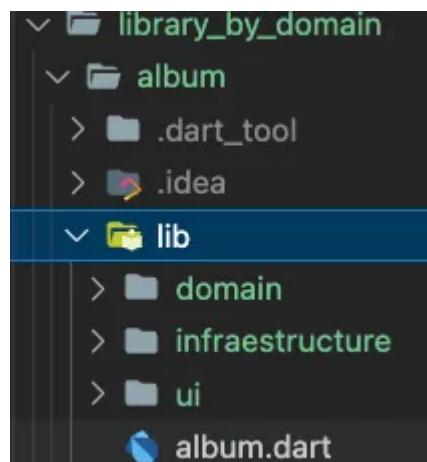
### Tercera forma: Segregación por dominio empleando librerías

Aquí buscaremos crear librerías completas por dominio, es decir, crearemos una especie de “*microapps*” (una por cada dominio) y la aplicación final es una especie de “*superapp*” que consume las mismas.

Este método es muy útil en grandes proyectos que quieran segregar la implementación. De esta forma la aplicación final no es un monolito, sino que está construida a partir de pequeñas soluciones. Esto optimiza la resolución de errores, pero tiene un alto impacto en el tiempo que se invierte para crear una funcionalidad.



En este caso se construyó una librería por cada uno de los dominios:



Dentro de cada *microapp* existiría una estructura de arquitectura limpia donde se implementa todos los casos de uso.

La aplicación final, en su archivo de dependencias, puede depender de estas librerías desde un repositorio privado, *pub.dev* o un *path* relativo:

En este caso, el código de la aplicación contenedora de las demás reduce en gran medida su tamaño. Te dejo el ejemplo completo para que lo puedas examinar en detalle ([enlace](#)).



## Conclusión

En este artículo vimos cómo implementar arquitectura limpia en Flutter. Además, repasamos los conceptos relevantes. Espero que puedas escoger la metodología que más se acopla a tus necesidades. Existen muchas otras formas de implementar arquitectura limpia pero te compartí las que he aplicado en diferentes proyectos. Si el contenido te gustó no olvides regalarnos +50 🌟 aplausos y ⭐ en el repositorio de GitHub.

Repositorio con los ejemplos:

[arquitectura\\_referencia\\_flutter/Clean\\_arch\\_2023\\_examples at main · ...](#)

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

[github.com](https://github.com)

Flutter

Clean Architecture

Architecture

Development

Platform

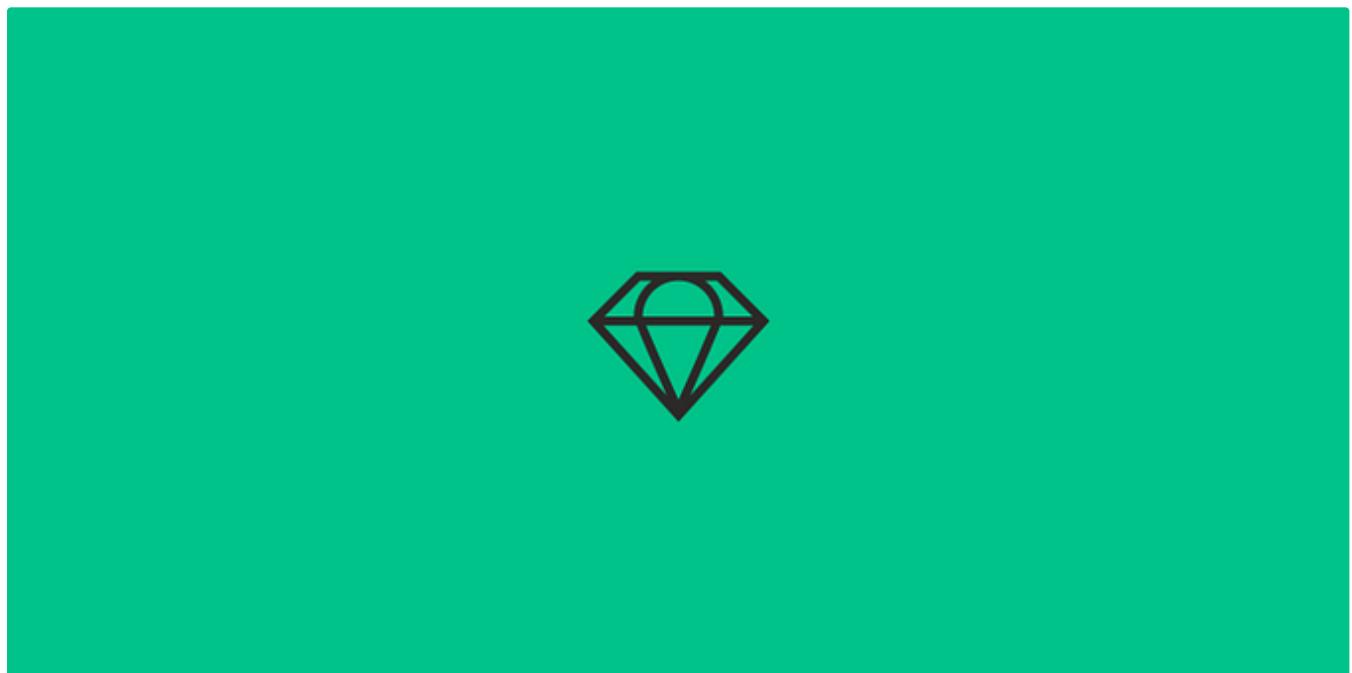
[Follow](#)

## Written by Daniel Herrera Sánchez

338 Followers · Writer for Bancolombia Tech

Flutter,Dart@GoogleDevExpert ❤ • 🔴Youtube Channel Weicode • 🧑FlutterMedellin Community Lead •  
🧑Angular Content creator • Speaker • GitHub: weincoder

More from Daniel Herrera Sánchez and Bancolombia Tech



Daniel Herrera Sánchez in Bancolombia Tech

### Flutter Provider: What is it, what is it for, and how to use it?

Flutter has developed libraries that promote our digital solutions. One of them the most powerful is called Provider. I will explain its...

5 min read · Aug 16, 2022



656



4



...



Daniel Herrera Sánchez in Bancolombia Tech

## Flutter Provider: Qué es, para qué sirve y cómo utilizarlo

Flutter tiene librerías de desarrollo que apalancan nuestras soluciones digitales, una de ellas, muy poderosa, es denominada Provider. En...

6 min read · Mar 14, 2022



...



Andrés Mauricio Gómez P in Bancolombia Tech

## Clean Architecture—Aislando los detalles

Una arquitectura que aísla al dominio de los detalles tecnológicos, aportando a la mantenibilidad y evolución de un componente de software

11 min read · Mar 3, 2020

👏 542

💬 4



...

Daniel Herrera Sánchez

**Lleva tus aplicaciones  
Flutter al siguiente  
nivel con Gemini ♦**



Febrero - 2024

 Daniel Herrera Sánchez

## **Lleva tus aplicaciones Flutter al siguiente nivel con Gemini ♦**

¿Te imaginas un compañero que sea capaz de acompañarte en tus desarrollos, contenido escrito, pero que también te ayude a mejorar tu...

8 min read · Feb 19, 2024

👏 176

💬



...

See all from Daniel Herrera Sánchez

See all from Bancolombia Tech

## Recommended from Medium



 Henry Ifebunandu

### Supercharge Your Flutter Apps with Google's App Architecture.

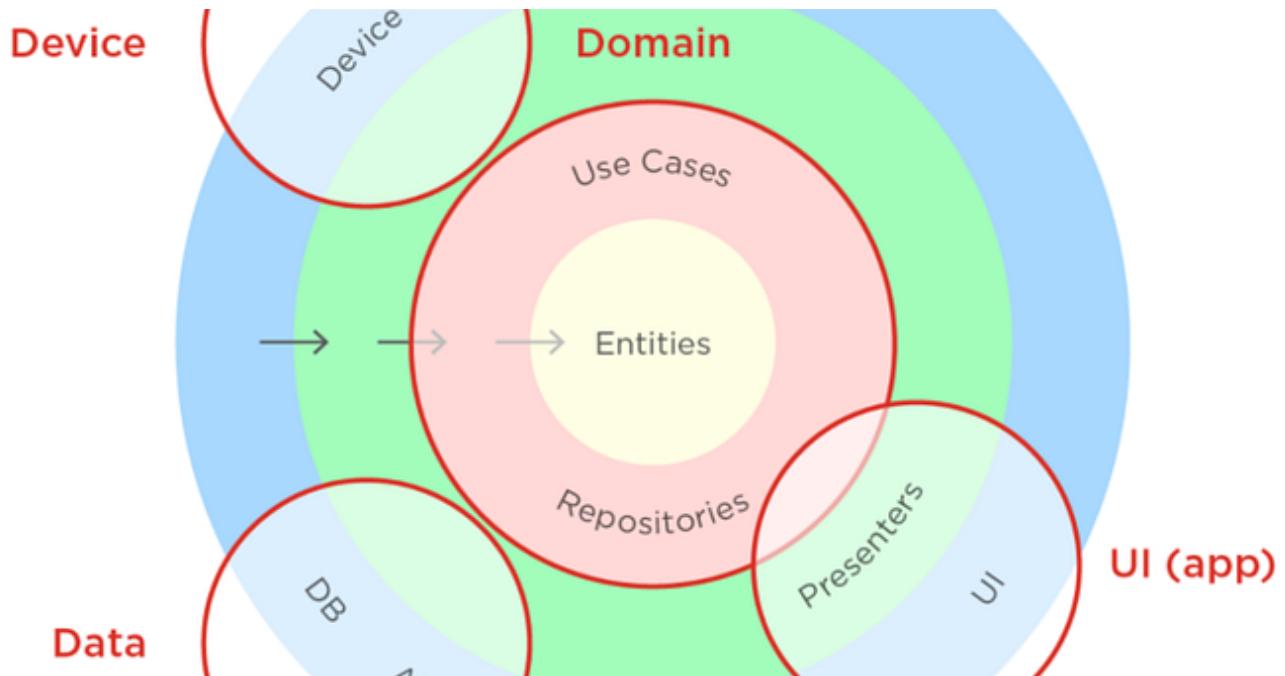
Explore the powerful use of Google's App Architecture and learn how it can elevate your Flutter app development. Discover best practices.

10 min read · Sep 28, 2023

 282



...



 Max Herrera

## Flutter Clean Architecture MVVM

Have you ever wondered the importance of looking at how to work with clean code architectures in flutter?

8 min read · Oct 15, 2023

 152 

 +

...

## Lists



### Growth Marketing

10 stories · 59 saves



### Stories to Help You Grow as a Software Developer

19 stories · 850 saves



### My Kind Of Medium (All-Time Faves)

67 stories · 225 saves



### Modern Marketing

70 stories · 447 saves



Shivam Kumar Nayak

## Understanding the Differences Between Bloc and Riverpod State Management in Flutter with Examples

Introduction:

5 min read · Oct 18, 2023



22



...

The screenshot displays a mobile application interface for inventory and sales management. The top section has a blue header with the title "Inventory and sales management". Below the header is a sidebar menu listing various features: Payments, Excel, PDF, Analytics, Product Management, Monthly Sales, Profit/Loss Calculation, Expense Management, and Call, SMS, Email, WhatsApp. At the bottom of the sidebar are icons for Flutter, Firebase, Android, iOS, and camera. The main area of the screen shows a grid of 16 smaller screens, each demonstrating a different feature of the app. These include: 1. Product Management (listing products like shirts, cameras, headphones). 2. Sales Performance (bar chart showing monthly sales). 3. Profitability (line chart showing profit over time). 4. Inventory Levels (table and bar chart showing item availability). 5. Product Details (product card for a camera). 6. Payments (screens for PayPal and Credit Card processing). 7. Customers (list of customers). 8. Sales Quotations (customer details and quotation forms). 9. Product Categories (list of categories like Electronics, Apparel, Home Goods). 10. Company Info (general company details). 11. Reports (various reports and analytics). 12. Expenses (expense tracking). 13. Profit/Loss (calculated financial results). 14. Monthly Sales (monthly sales summary). 15. Product Management (another view of product listing). 16. Analytics (another view of performance and reporting).

code.market

## Top 11 Free & Premium Flutter Card Templates to Elevate Your UI Design

5 min read · Jan 31, 2024



7



...



Ashish Sharma

## Exploring ISAR: A Step-by-Step Guide to Database Management in Flutter

As a Flutter developer, you understand the importance of efficient data management in your applications. Enter ISAR, a high-performance...

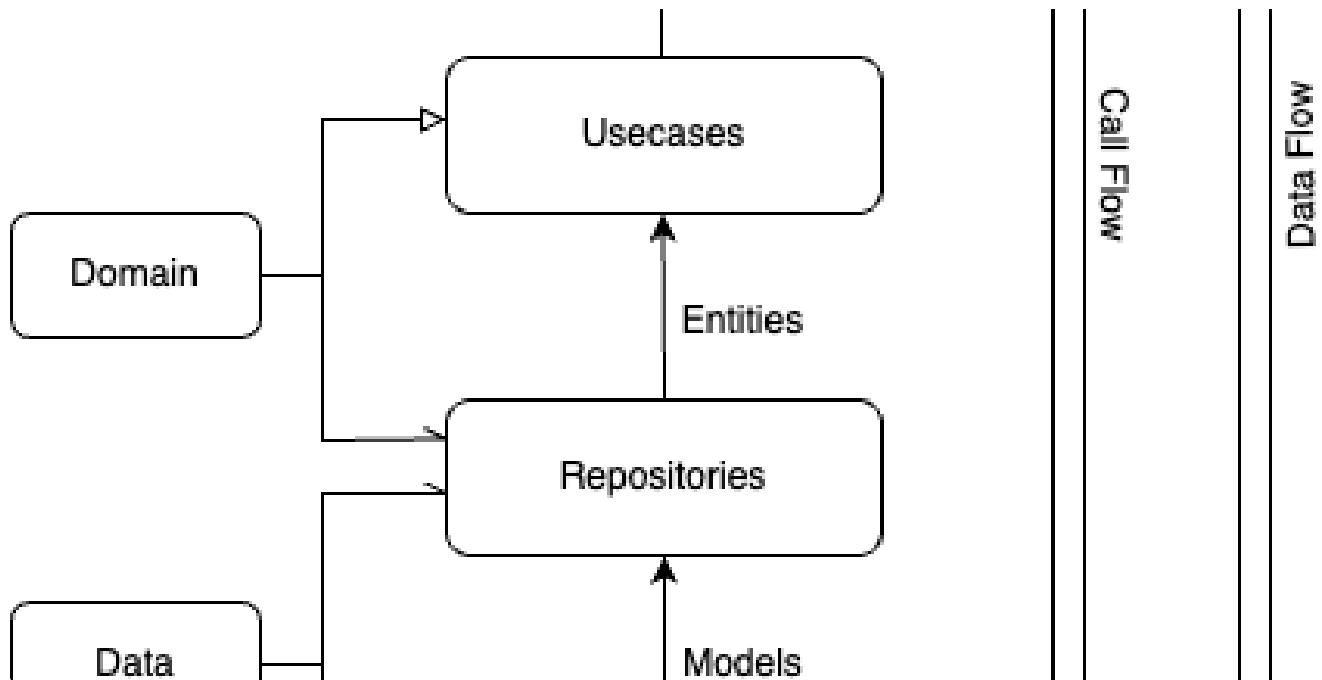
3 min read · Sep 2, 2023



31



...



Dineshkumar Atti

## Flutter Clean 🚗 Architecture-Bloc Pattern

It makes our life easier when we have to change our software in the future. A good plan for the app but also for specific features is the...

7 min read · Nov 6, 2023

51    1

+    ...

See more recommendations