Arquitectura CMI V0.1.1 Experimental - Backend

Autor Diego Moreano Merino
Desarrollador, Utoqing App

Figure 1

Guía Oficial de la Arquitectura Modular y Escalable para Proyectos Backend

Introducción

La Arquitectura CMI (Capas, Módulos y Contenedores Recursivos) es una propuesta moderna diseñada para enfrentar los desafíos de organización, escalabilidad y mantenibilidad en proyectos de software backend, especialmente en entornos donde el crecimiento acelerado, la modularidad y el trabajo colaborativo son constantes.

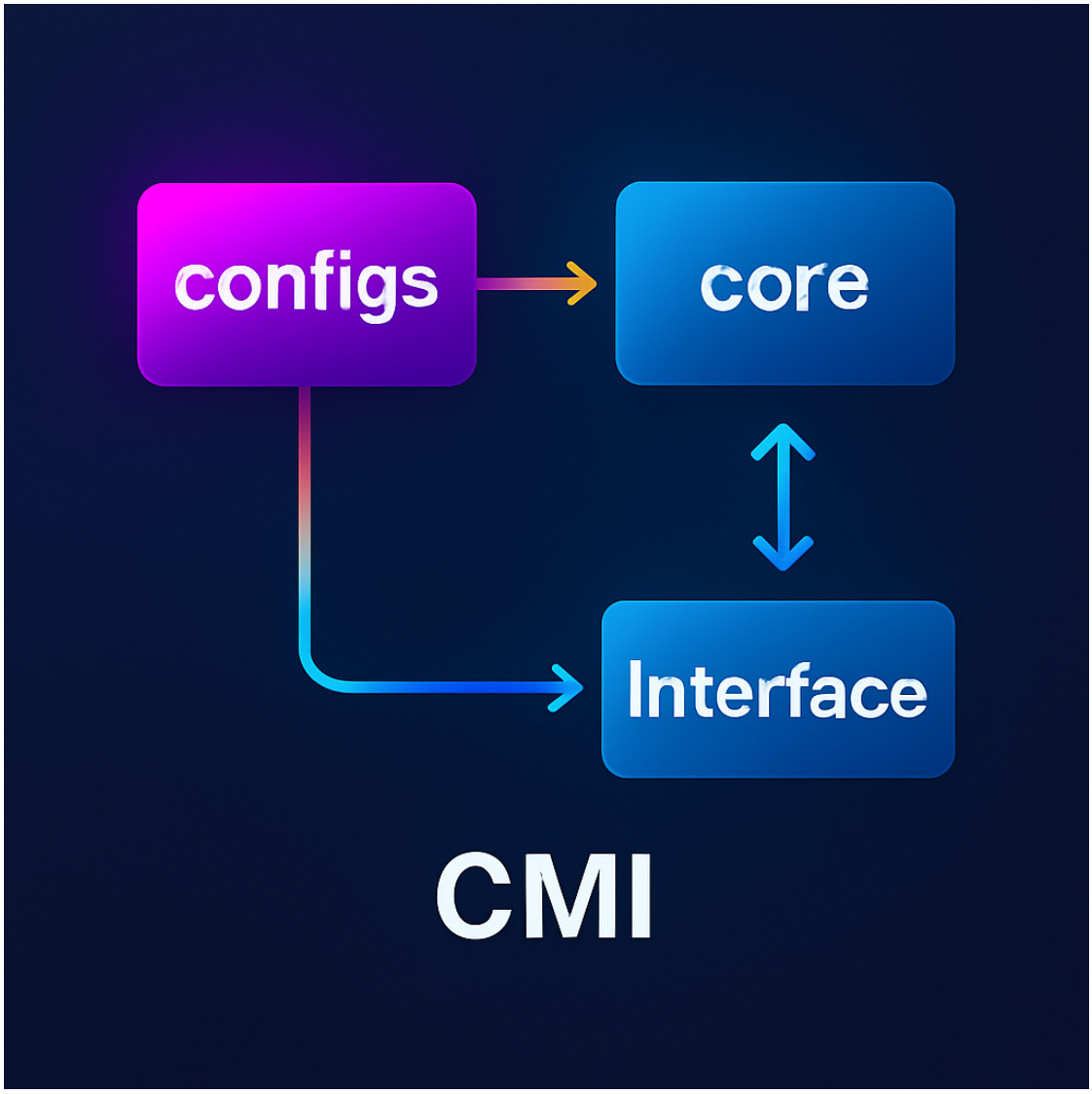
A diferencia de arquitecturas tradicionales como MVC o Clean Architecture, CMI establece una estructura jerárquica clara basada en tres pilares fundamentales: **Capas**, **Módulos** y **Contenedores**.

Cada elemento posee límites bien definidos y responsabilidades específicas, lo que garantiza un bajo acoplamiento y una alta cohesión en todo el proyecto backend.

CMI no solo facilita el desarrollo inicial, sino que también prepara el proyecto para su evolución a largo plazo, permitiendo integrar nuevas funcionalidades, servicios o tecnologías sin comprometer la estabilidad ni generar desorden.

Esta guía está dirigida a desarrolladores backend, arquitectos de software y equipos que buscan una solución flexible y robusta para gestionar servidores, microservicios, API REST, controladores de eventos, CLI, o cualquier otro punto de entrada de backend.

Al recorrer este documento, descubrirás cómo aplicar CMI para lograr proyectos más ordenados, sostenibles y preparados para el futuro.



¿Por qué CMI?

En el desarrollo backend, es común enfrentarse a problemas como:

- Código desorganizado y difícil de escalar.
- Dificultad para integrar nuevos endpoints, servicios o funcionalidades sin romper lo existente.
- Proyectos donde distintos equipos pisan el trabajo de otros por falta de una estructura clara.
- Mantenimiento costoso debido a la alta dependencia entre componentes, modelos y controladores.

La Arquitectura CMI surge como respuesta a estos desafíos, ofreciendo:

- Una estructura clara y jerárquica, basada en **Capas**, **Módulos** y **Contenedores**.
- Flexibilidad para adaptarse a cambios de requisitos o tecnologías (bases de datos, motores de eventos, servicios externos).
- Escalabilidad, permitiendo que los servicios crezcan de manera ordenada y distribuida.
- Mantenibilidad, facilitando que nuevos desarrolladores comprendan el sistema rápidamente.
- Reutilización efectiva de lógica, modelos de dominio, validadores, adaptadores y controladores.

Comparativa con otras arquitecturas

Table 1. *Comparativa con otras arquitecturas*

Característica	**Clean Architecture**	**Hexagonal**	**MVC**	**CMI**
Separación de capas	Sí	Sí	Sí	Sí
Modularidad avanzada	Parcial	Alta	Limitada	**Muy Alta**
Flexibilidad	Alta	Alta	Baja	**Muy Alta**
Reutilización	Media	Alta	Baja	**Alta**
Complejidad inicial	Alta	Baja	Baja	Media

Continued on next page

Table 1. *Comparativa con otras arquitecturas* (Continued)

Adaptable a microservicios	Parcial	Sí	No	**Total**
Escalabilidad	Alta	Alta	Baja	**Muy Alta**

¿Cuándo usar CMI?

CMI es ideal para:

- Sistemas backend donde se prevé un crecimiento constante, como plataformas SaaS, APIs REST, GraphQL, WebSockets o microservicios.
- Equipos de desarrollo backend que buscan una estructura clara para trabajar de forma colaborativa y distribuida.
- Aplicaciones que requieren una alta mantenibilidad, modularidad y control de responsabilidades.
- Entornos donde se desea una gestión eficiente de lógica de negocio, acceso a datos y servicios externos desacoplados.
- Proyectos que planean escalar o cambiar partes tecnológicas (por ejemplo, cambiar de base de datos o sistema de autenticación) sin reestructurar toda la lógica.

Este documento presenta la versión **0.1.1 experimental** de l*a Arquitectura CMI para Backend, que está en constante evolución, por lo que se recomienda a los desarrolladores aportar sugerencias o adaptaciones según sus experiencias en distintos entornos tecnológicos.*

Filosofía y Principios de la Arquitectura CMI

Filisofia CMI

La Arquitectura CMI nace con una visión clara: ofrecer un marco estructurado que garantice orden, cohesión y escalabilidad sostenible en sistemas backend.

A diferencia de enfoques más permisivos, CMI establece normas precisas que previenen el desorden típico que surge en entornos con crecimiento acelerado, múltiples servicios o equipos grandes.

CMI entiende que un sistema exitoso no solo debe “funcionar”, sino que debe ser **fácil de mantener, extender y adaptar** a nuevas necesidades sin comprometer su estabilidad.

Por ello, propone una estructura jerárquica basada en **Capas, Módulos y Contenedores**, donde cada estructura tiene responsabilidades bien definidas.

Aunque CMI es estricta en su núcleo, permite **flexibilidad controlada**, siempre bajo principios de claridad, cohesión y separación de intereses.

Principios Fundamentales

Single Responsibility Principle (SRP)

La arquitectura CMI aplica el principio de responsabilidad única, el cual establece que cada Capa, Módulo y Contenedor en un proyecto debe tener una única responsabilidad o motivo para cambiar. Esto significa que cada elemento debe hacer una sola cosa y hacerla bien.

Cada archivo, contenedor, módulo o capa debe cumplir una única función definida. Por ejemplo: los elementos que transforman datos se agrupan en adapters/, las estructuras de lógica en consumers/, y las definiciones de contrato en rules/. Esto evita mezclar propósitos dentro de un mismo archivo o estructura, reduciendo el riesgo de errores y mejorando la legibilidad.

Open/Closed Principle (OCP)

La arquitectura CMI aplica el principio de abierto/cerrado, que sugiere que los distintos elementos deben estar abiertos para la extensión, pero cerrados para la modificación. Es decir, se debe poder agregar una nueva funcionalidad a una parte existente sin cambiar todo su código original.

En CMI, esto se logra creando nuevos archivos o contenedores que amplían comportamientos existentes mediante nuevos adaptadores, nuevos consumidores, o nuevas definiciones de contratos. Así, se puede evolucionar una funcionalidad sin afectar el comportamiento anterior ni los módulos que ya la utilizan.

Liskov Substitution Principle (LSP)

La arquitectura CMI aplica el principio de sustitución de Liskov, el cual indica que las clases derivadas o subtipos deben poder ser sustituidos por sus clases base o tipos sin alterar el correcto funcionamiento de la aplicación. Esto significa que un objeto de una clase hija debe ser intercambiable con un objeto de la clase padre sin que el usuario del objeto necesite saber la diferencia.

Esto se implementa utilizando interfaces en core/rules/, que permiten múltiples implementaciones dentro del proyecto sin alterar otras estructuras que las consumen. Cualquier elemento que dependa de una abstracción puede cambiar la implementación concreta sin alterar su comportamiento general.

Interface Segregation Principle (ISP)

La arquitectura CMI aplica este principio, que sugiere que es mejor tener múltiples interfaces específicas y pequeñas en lugar de una única interfaz general. Las clases no deberían verse obligadas a depender de métodos que no utilizan.

Los contratos definidos en rules/services/ y rules/consumers/ están divididos por función específica. Esto permite que cada implementación se enfoque en una única funcionalidad sin tener que acoplarse a métodos que no le corresponden. Además, mejora la claridad del sistema y la capacidad de mantenerlo y extenderlo.

Dependency Inversion Principle (DIP)

La arquitectura CMI aplica el principio de inversión de dependencias, lo que establece que los elementos de alto nivel no deben depender de los elementos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

Esto se implementa en la capa core/ utilizando clases abstractas en el módulo rules/, lo que permite que las dependencias sean invertidas y desacopladas. Las implementaciones se resuelven en core/use/, manteniendo el proyecto organizado y permitiendo pruebas unitarias sin acoplamiento directo a detalles técnicos.

Principio de Separación de Intereses (Separation of Concerns)

La arquitectura CMI aplica el principio que implica dividir un proyecto en distintas Capas, Módulos y Contenedores, donde cada una aborda una preocupación específica o una funcionalidad concreta. La separación de intereses mejora la modularidad del proyecto, permitiendo que los desarrolladores trabajen en partes individuales del proyecto sin necesidad de comprender toda la aplicación en su totalidad. Esto también facilita la identificación y corrección de errores, ya que los problemas suelen estar localizados en un solo lugar.

Principio DRY (Don't Repeat Yourself)

La arquitectura CMI aplica el principio DRY, que busca evitar la duplicación de código o lógica. Cada parte de conocimiento o funcionalidad debe tener una representación única y sin duplicados en el proyecto. Siguiendo este principio, se reduce la cantidad de código redundante, lo que facilita la modificación y el mantenimiento. Si una pieza de lógica necesita ser cambiada, solo hay que hacerlo en un lugar, minimizando el riesgo de inconsistencias y errores.

Principio KISS (Keep It Simple, Stupid)

La arquitectura CMI aplica el principio que enfatiza la importancia de mantener el diseño y la implementación de la aplicación lo más simple y directa posible. La simplicidad reduce la complejidad y facilita la comprensión, el mantenimiento y

la modificación del código. Evita agregar funcionalidades complejas de forma innecesaria que puedan complicar el desarrollo o dificultar el entendimiento.

Encapsulamiento

La arquitectura CMI aplica el principio de encapsulamiento para ocultar los detalles internos de un objeto y exponer solo lo necesario a través de una interfaz pública. Esto protege la integridad del estado interno del objeto y reduce el acoplamiento entre diferentes partes de la aplicación. Al encapsular los datos y las operaciones, se permite un mayor control sobre cómo las partes interactúan entre sí, lo que mejora la modularidad y la seguridad de la aplicación.

Los enrutadores son obligatorios y controlan qué archivos se exponen; el acceso directo a archivos internos está prohibido. Esto aplica tanto a Capas como a Módulos o Contenedores.

Alta Cohesión y Bajo Acoplamiento

La arquitectura CMI aplica el principio de alta cohesión, que se refiere a la medida en que los elementos dentro de una Capa, Módulo o Contenedor están relacionados y trabajan juntos para cumplir una función específica.

Una Capa, Módulo o Contenedor con alta cohesión es más fácil de entender, mantener y reutilizar. Por otro lado, el bajo acoplamiento indica que los diferentes niveles tienen pocas o nulas dependencias entre sí, lo que permite que se modifiquen o evolucionen de manera independiente, reduciendo el impacto de los cambios en la aplicación.

Principio de Inversión de Control (IoC)

La arquitectura CMI aplica el principio de inversión de control, el cual sugiere que el flujo de control en una aplicación debe ser manejado por un gestor de dependencias o configuración, en lugar de que las estructuras internas lo hagan directamente. Esto significa que, en lugar de que los elementos internos creen sus dependencias, estas se inyectan desde fuera, respetando los contratos definidos.

Esto promueve la flexibilidad y permite la creación de aplicaciones más modulares, al separar la creación de los objetos de su uso.

Modularidad

La arquitectura CMI aplica el principio de modularidad que divide una aplicación en Capas, Módulos y Contenedores, independientes y autónomos, donde cada nivel encapsula una funcionalidad específica y bien definida. Esto facilita el desarrollo, la prueba, el mantenimiento y la reutilización de código.

Resumen visual de principios aplicados en CMI

Table 2. *Resumen visual de principios aplicados en CMI*

Principio	Descripción breve	Aplicación en CMI
SRP (Single Responsibility Principle)	Una clase o módulo debe tener una única responsabilidad.	Cada archivo, contenedor, módulo o capa cumple una única función definida. Ej: adapters/ solo transforma datos, consumers/ solo ejecutan lógica.
OCP (Open/Closed Principle)	Abierto a extensión, cerrado a modificación.	Se agregan nuevas funcionalidades creando nuevos archivos o módulos sin modificar los existentes.
LSP (Liskov Substitution Principle)	Una subclase puede sustituir a su clase base sin romper el proyecto.	Las interfaces en rules/ permiten múltiples implementaciones en use/ sin afectar otras estructuras del proyecto.
ISP (Interface Segregation Principle)	Es mejor tener interfaces pequeñas y específicas.	Los contratos en rules/services/ y rules/-consumers/ se dividen por función específica, evitando dependencias innecesarias.

Continued on next page

Table 2. *Resumen visual de principios aplicados en CMI* (Continued)

DIP (Dependency Inversion Principle)	Las dependencias se dirigen hacia abstracciones, no implementaciones.	Las estructuras que consumen lógica dependen exclusivamente de contratos definidos en rules/, lo que permite una implementación desacoplada en use/.
Separación de Intereses (SoC)	Cada parte del sistema se enfoca en una preocupación específica.	Las capas (configs/, core/, interface/) separan configuración, lógica y entrada, evitando cruces de responsabilidades.
DRY (Don't Repeat Yourself)	Evitar duplicación de lógica o estructuras.	Lógica reutilizable ubicada en shared/ (como middlewares, validadores o helpers), contratos únicos en rules/, y transformaciones centralizadas en adapters/.
KISS (Keep It Simple, Stupid)	Mantener el diseño lo más simple posible.	Se evita crear estructuras innecesarias. Cada contenedor tiene una responsabilidad clara y limitada.

Continued on next page

Table 2. *Resumen visual de principios aplicados en CMI* (Continued)

Encapsulamiento	Ocultar detalles internos y exponer solo lo necesario.	Los enrutadores definen qué archivos se exponen. El acceso directo a estructuras internas está prohibido. Todos los elementos deben exponerse a través de su enrutador correspondiente, incluso a nivel de módulos y capas.
Alta cohesión / Bajo acoplamiento	Agrupar elementos relacionados y reducir dependencias externas.	services/, consumers/, adapters/, etc., están organizados según su función, permitiendo modificar uno sin afectar otros contenedores del proyecto.
IoC (Inversión de Control)	Las dependencias se inyectan, no se crean dentro del proyecto.	Las estructuras internas no crean sus propias dependencias, sino que las reciben desde configuraciones externas o contenedores superiores.

Conceptos Fundamentales

La Arquitectura CMI se basa en una estructura jerárquica compuesta por **Capas**, **Módulos**, **Contenedores** y **Enrutadores**. Estos elementos son la base de la organización y deben aplicarse de forma estricta para garantizar la coherencia, el orden y la escalabilidad del proyecto.

A continuación, se definen cada uno de estos conceptos esenciales.

Capas

Las **Capas** en la Arquitectura CMI representan la división estructural más rígida y fundamental. Esta separación no es arbitraria, sino el resultado de una

necesidad clara: mantener el orden a nivel macro en cualquier proyecto backend, sin importar su tamaño o complejidad.

En muchas arquitecturas flexibles, es común que las responsabilidades se mezclen con el tiempo, especialmente bajo presión de crecimiento o cambios rápidos. Este desorden puede empezar de forma imperceptible —una función de lógica en una configuración global, una validación dentro de un origen de datos— pero a medida que el proyecto evoluciona, estas pequeñas decisiones generan una estructura difícil de mantener, escalar y entender.

CMI resuelve este problema desde el inicio estableciendo un sistema de Capas fijas e inalterables que separan claramente:

- La configuración técnica global (configs/).
- La lógica de negocio y los datos del proyecto (core/).
- Los puntos de entrada del proyecto, como API, CLI o controladores de eventos (interface/).

¿Por qué CMI impone capas estrictas?

- Para evitar que los desarrolladores "personalicen" la estructura según criterios subjetivos.
- Para garantizar que, sin importar quién trabaje en el proyecto o cuánto crezca, siempre exista una organización reconocible y coherente.
- Para asegurar que cada cambio se realice dentro del contexto correcto, minimizando riesgos de errores estructurales.

Esta rigidez no es una limitación, sino una herramienta para **proteger la salud del proyecto a largo plazo**.

Consecuencias de ignorar la separación de capas

- Dificultad para localizar o modificar funciones específicas.
- Aumento del acoplamiento entre partes de la aplicación.
- Crecimiento desordenado, donde la configuración, la lógica y la interfaz se entrelazan sin control.
- Refactorizaciones costosas en etapas avanzadas del desarrollo.

Módulos

Los **Módulos** en la Arquitectura CMI son la respuesta a uno de los problemas más comunes en el desarrollo de software: el crecimiento descontrolado dentro de una misma área funcional. Aunque las **Capas** establecen una división clara a nivel macro, es dentro de cada **Capa** donde el código tiende a expandirse rápidamente, y sin una estructura adecuada, este crecimiento puede convertirse en caos.

Aquí es donde entran los módulos. Su propósito es **aislar funcionalidades específicas** dentro de cada **Capa**, garantizando que cada bloque de código tenga un espacio definido, evitando concentraciones excesivas de archivos y responsabilidades difusas.

¿Por qué existen los Módulos en CMI?

- Para dividir de manera lógica y ordenada las diferentes áreas funcionales dentro de cada **Capa**.
- Para aplicar de forma efectiva el principio de **separación de responsabilidades**, no solo a nivel de clases o funciones, sino también en la organización del proyecto.
- Para facilitar la escalabilidad, permitiendo que cada funcionalidad crezca de forma independiente sin afectar otras partes de la aplicación.
- Para evitar el llamado "*cajón de sastre*", donde todo termina en una sola carpeta sin criterio.

La diferencia clave con otras arquitecturas

En muchas arquitecturas, la creación de **Módulos** queda a criterio del equipo, lo que puede derivar en estructuras inconsistentes entre proyectos o incluso dentro del mismo equipo de trabajo.

CMI estandariza esta práctica definiendo **Módulos obligatorios** (llamados Módulos estándar) y establece criterios claros para la creación de Módulos adicionales solo cuando es estrictamente necesario.

Errores comunes al gestionar Módulos

- No utilizar **Módulos**, dejando crecer desordenadamente las **Capas**.
- Crear **Módulos** por costumbre sin una necesidad real, generando estructuras vacías o innecesarias.
- Duplicar funcionalidades entre Módulos por falta de claridad en sus responsabilidades.
- Omitir los enrutadores dentro de los Módulos, lo que rompe el control de acceso al código interno.

¿Cuándo crear un módulo adicional?

Aunque CMI permite la creación de módulos adicionales, esta decisión debe ser siempre **justificada**. La creación de un nuevo módulo debe responder a una necesidad clara de aislamiento funcional que no pueda ser resuelta mediante contenedores dentro de un módulo existente.

Crear módulos por "comodidad" o para evitar pensar en la estructura es uno de los errores más costosos a largo plazo.

Contenedores

En la Arquitectura CMI, los **Contenedores** son mucho más que simples carpetas; son una herramienta fundamental para mantener el orden a nivel micro dentro de cada Módulo. Su función es organizar de manera precisa los diferentes tipos de archivos que componen una funcionalidad, asegurando que cada elemento esté ubicado en el lugar correcto según su responsabilidad.

A medida que un proyecto crece, la tendencia natural es que los archivos comiencen a mezclarse si no existe una guía estricta. Los contenedores en CMI evitan este problema, estableciendo un sistema claro para clasificar y aislar controladores, validadores, transformadores, rutas, middlewares, entre otros.

¿Por qué son esenciales los contenedores en CMI?

- Permiten que cualquier desarrollador pueda identificar rápidamente dónde encontrar o colocar un archivo.
- Refuerzan el principio de **alta cohesión**, agrupando elementos relacionados en un solo lugar.
- Evitan la creación de carpetas arbitrarias basadas en preferencias personales o criterios inconsistentes.
- Preparan la estructura para el crecimiento futuro, incluso si al inicio algunos contenedores están vacíos.
- Facilitan la mantenibilidad y la escalabilidad a largo plazo, especialmente en equipos colaborativos.

Sin contenedores bien definidos, un proyecto puede volverse caótico en poco tiempo, dificultando tanto la comprensión como la extensión de su funcionalidad.

Tipos de Contenedores en CMI

CMI no deja a criterio del desarrollador cómo organizar los archivos. Define contenedores específicos para cada propósito:

- **Contenedores Estándar:** De uso obligatorio. Organizan los elementos más comunes como rules/, consumers/, services/, adapters/, origins/ y router/.
- **Contenedores de Dominio:** Agrupan funcionalidades específicas dentro de Módulos que requieren varios grupos de contenedores, como en módulos complejos con múltiples orígenes o adaptadores.
- **Contenedores Adicionales:** Utilizados solo cuando es necesario organizar elementos que no encajan en los contenedores estándar, como cli/ o shared/.
- **Recursividad en Contenedores:** Permite mantener el orden en niveles más profundos cuando la complejidad lo requiere. Su uso debe ser justificado y controlado.

Errores comunes al trabajar con contenedores

- Ignorar los contenedores estándar por considerar que "no son necesarios en proyectos pequeños".
- Crear carpetas personalizadas donde ya existe un contenedor estándar definido por la arquitectura.
- Mezclar diferentes tipos de archivos en un mismo contenedor (por ejemplo, colocar adaptadores dentro de services/).
- Abusar de la recursividad, generando estructuras innecesariamente profundas que afectan la legibilidad.
- No utilizar el contenedor router/, comprometiendo el control de exposición del módulo.

Enrutadores

En la Arquitectura CMI, los enrutadores son uno de los elementos más distintivos y esenciales para mantener el control, la cohesión y la integridad de la estructura del proyecto.

A diferencia de otras arquitecturas donde los archivos se importan libremente desde cualquier ubicación, CMI establece que toda interacción entre capas, módulos o contenedores debe realizarse exclusivamente a través de sus enrutadores, los cuales definen de forma precisa qué elementos pueden ser accesibles desde el exterior y cuáles deben permanecer encapsulados.

Un enrutador no es simplemente un archivo de exportación, sino un mecanismo estructural normativo que permite garantizar el bajo acoplamiento entre estructuras y controlar la exposición externa de los elementos internos de cada contenedor, como servicios, adaptadores, consumidores o validadores.

En todos los casos, los enrutadores deben limitarse a exponer únicamente los elementos internos que han sido diseñados para ser accedidos desde fuera del contenedor o módulo.

Un enrutador no solo organiza las exportaciones, sino que define el límite explícito de acceso dentro del sistema, reforzando el bajo acoplamiento estructural.

Esto asegura que el proyecto conserve una estructura mantenible, donde ninguna función, clase o tipo salga del ámbito de su dominio sin una intención clara y controlada.

¿Por qué son fundamentales los enrutadores en CMI?

- **Control de acceso**: Evitan que los desarrolladores accedan directamente a archivos internos, protegiendo la estructura de dependencias desordenadas.
- **Encapsulamiento real**: Permiten exponer solo lo necesario, manteniendo ocultos los detalles de implementación que no deben ser utilizados fuera de su contexto.
- **Facilitan el mantenimiento**: Si la estructura interna de una capa, módulo o contenedor cambia, mientras el enrutador mantenga su contrato, el resto de la aplicación no se verá afectada.
- **Claridad en las dependencias**: Cuando un archivo importa desde un enrutador, queda claro que está accediendo a una interfaz pública controlada, no al "interior" de la aplicación.
- • **Refuerzo de límites jerárquicos:** Los enrutadores definen con precisión los puntos de entrada permitidos en cada nivel estructural. Desde la capa configs/ hasta los handlers de interface/, todo acceso debe realizarse mediante su enrutador correspondiente, evitando saltos estructurales o referencias cruzadas no autorizadas.

Tipos de Enrutadores en CMI

CMI establece el uso de enrutadores en distintos niveles:
ser el **único punto de acceso autorizado** para su contexto.
Enrutadores Normales:

Son los enrutadores utilizados en cualquier nivel estructural de CMI (capas, módulos o contenedores). Controlan qué elementos son accesibles externamente y cuáles permanecen encapsulados, actuando como punto de acceso único a su contexto.

Errores comunes al gestionar enrutadores

- **Omitir enrutadores donde son obligatorios**: No crear un enrutador en una capa, módulo o contenedor que expone elementos hacia otras estructuras, violando la regla de control de acceso. *(Nota: los contenedores genéricos son la*

única excepción, ya que no requieren enrutador propio si no que usan el de su padre).

- **Exposición indiscriminada**: Exportar todos los elementos internos sin control, rompiendo el principio de encapsulamiento y haciendo públicos detalles que deberían permanecer internos.
- **Acceso directo no autorizado**: Importar archivos internos directamente, sin pasar por el enrutador correspondiente, generando acoplamiento indebido.
- **Enrutador desactualizado**: No actualizar el enrutador al agregar, eliminar o modificar elementos expuestos, provocando errores de dependencia o accesos rotos.
- **Confundir rol de enrutador:** Incluir lógica funcional, inicializaciones, llamadas o cualquier otro tipo de código ejecutable dentro de un enrutador, cuando su único propósito es definir de forma ordenada qué elementos internos pueden ser accedidos desde el exterior de su estructura.

Relación Jerárquica

1. Capa
2. Módulo
3. Contenedor

Elementos: consumidores, reglas, servicios, adaptadores, orígenes

Cada nivel superior **desconoce los detalles internos** del nivel inferior gracias al principio de **abstracción**, comunicándose siempre a través de su **enrutador** correspondiente.

Ejemplo Visual

```
lib/
 configs/
     env/
     constants/
     logger/
     configs.dart                # Enrutador de Configs

 core/
     rules/
         data/
         services/
```

```
        consumers/
        rules.dart              # Enrutador de Rules

    uses/
        consumers/
        adapters/
        services/
        origins/
        uses.dart               # Enrutador de Use

    core.dart                   # Enrutador de Core

interface/
    http/
        routes/
        handlers/
        http.dart               # Enrutador del ómdulo HTTP

    cli/
        commands/
        cli.dart                # Enrutador del ómdulo CLI

    events/
        listeners/
        events.dart             # Enrutador del ómdulo
  Events

    shared/
        middlewares/
        shared.dart             # Enrutador del ómdulo
  Shared

    interface.dart              # Enrutador de Interface

main.dart
```

Organización del proyecto

En un proyecto backend estructurado bajo la arquitectura CMI, el orden no es una decisión opcional, sino una condición para su crecimiento, mantenibilidad y resistencia a largo plazo. La forma en que los elementos del proyecto se agrupan, se aíslan y se exponen no solo afecta la experiencia de desarrollo, sino que condiciona

la calidad de sus integraciones, su capacidad de escalar y la claridad con la que otros podrán intervenir sobre él.

A diferencia de otras arquitecturas más flexibles o ambiguas, CMI impone una jerarquía estricta desde su base. Esta jerarquía se expresa en Capas que no pueden omitirse ni redefinirse, cada una con un rol específico, límites estructurales definidos y relaciones permitidas claramente delimitadas. No basta con saber que existen Capas, Módulos o Contenedores: es necesario comprender cómo se relacionan en el espacio físico del proyecto, qué se permite dentro de cada uno y cómo se distribuyen sus responsabilidades.

Este capítulo no busca ejemplificar ni simular un caso real —eso le corresponde a una futura aplicación práctica—, sino dejar establecida la forma correcta de organizar un proyecto backend que adopta la arquitectura CMI. La claridad de esta organización permite que los flujos técnicos y funcionales operen sobre una base firme, legible y evolutiva.

La Capa CONFIGS

La Capa configs/ cumple una función silenciosa pero fundamental en la arquitectura CMI: proveer una fuente única, centralizada y confiable para todos los elementos de configuración técnica del proyecto. Aunque su contenido no ejecuta lógica del dominio, la calidad y organización de esta Capa determinan cuán consistente, adaptable y segura puede ser una base backend a medida que crece.

Esta Capa no contiene lógica de negocio ni implementaciones operativas, pero es invocada por todas las demás de forma indirecta o estructural. Desde el manejo de variables de entorno hasta la definición de constantes globales o la configuración del sistema de logs, configs/ actúa como el punto de partida de las condiciones bajo las cuales el proyecto se ejecuta.

La arquitectura CMI no deja este punto al azar. La existencia de esta Capa y su contenido estática pero organizadamente viva permite que las demás Capas permanezcan libres de decisiones contextuales como si están en un entorno de desarrollo o producción, qué claves están disponibles, qué nivel de log está habilitado o qué middlewares globales deben ser aplicados a la entrada.

Estructura recomendada

Dentro de la Capa configs/, CMI establece una estructura mínima clara, que puede extenderse si el proyecto lo justifica, pero que debe mantenerse siempre bajo control estricto.

- **env/**: Agrupa todo lo relacionado al manejo de variables de entorno. Aquí se ubican archivos como .env.defaults, funciones para cargar variables desde el entorno operativo y utilidades para acceder a ellas de forma controlada desde otras Capas.

- **constants/**: Contiene constantes globales utilizadas por distintos módulos. Estas pueden incluir códigos de error, mensajes comunes, claves simbólicas o valores predeterminados, que no deben ser codificados directamente dentro de core/ ni de interface/.

- **logger/**: Define el sistema de registros del proyecto. Incluye la configuración de niveles de severidad, formatos, salidas y cualquier integración opcional con herramientas externas de monitoreo o auditoría.

- **enrutador/**: Enrutador obligatorio. Define qué elementos de la configuración estarán disponibles para otras Capas. Nunca debe exponer la totalidad del contenido de configs/, sino únicamente lo necesario.

Reglas estructurales de la Capa CONFIGS

CMI impone una serie de restricciones a esta Capa para evitar su desnaturalización:

1. **No puede contener lógica de negocio.** Ninguna función dentro de configs/ debe procesar decisiones del dominio, validar operaciones ni interpretar datos de entrada.

2. **No puede importar desde core/ ni desde interface/.** Esta Capa debe ser completamente autónoma. No puede verse afectada por estructuras lógicas ni por mecanismos de entrada.

3. **Su contenido debe ser pasivo.** Toda configuración debe estar disponible para ser accedida, pero no debe ejecutar lógica por defecto, salvo inicialización controlada.

4. **Todo acceso debe realizarse a través de su enrutador.** No se permite acceder directamente a archivos individuales dentro de configs/ sin pasar por su punto de exposición.

La Capa CORE

La Capa core/ es el corazón del proyecto. Es donde se define todo lo que la arquitectura protege: las reglas del dominio, los contratos que gobiernan las operaciones, las transformaciones necesarias para mantener integridad estructural, y las implementaciones que traducen esas reglas en acciones concretas. Esta Capa representa el núcleo lógico del proyecto y, como tal, está diseñada para ser completamente independiente de tecnologías específicas, mecanismos de entrada o decisiones contextuales.

Toda funcionalidad significativa del proyecto nace, se valida, se transforma y se regula dentro de esta Capa. Cualquier acción que tenga efecto sobre el estado del

dominio debe ser ejecutada desde aquí, y toda solicitud externa debe atravesar esta estructura para ser considerada válida. Esta es la única Capa donde la lógica del proyecto es plenamente expresada.

División interna obligatoria

La Capa core/ está dividida en dos Módulos obligatorios:

- **rules/**: Contiene contratos, definiciones abstractas, entidades del dominio, y operaciones representadas en su forma declarativa.
- **use/**: Implementa las operaciones definidas en rules/. Aquí se aplica la lógica real, se conectan orígenes de datos, se invocan servicios técnicos y se ejecutan flujos de negocio.

Cada uno de estos Módulos sigue sus propias reglas estructurales y su propia forma de encapsulamiento, pero ambos están completamente al servicio de una arquitectura modular, cohesionada y de bajo acoplamiento.

Módulo rules/: Definiciones del dominio. Este Módulo contiene todo lo que es abstracto, declarativo y representativo del dominio. No se permite aquí ningún tipo de lógica operativa, acceso a datos ni implementación de flujos. Su propósito es definir las reglas y estructuras que regirán el proyecto sin importar qué tecnología, cliente o entorno las utilice.

- **data/**: Entidades, objetos de transferencia de datos (DTOs), estructuras base, tipos o interfaces que representan el modelo del dominio.
- **services/**: Contratos de servicios técnicos que serán implementados en use/, tales como almacenamiento, autenticación, mensajería, etc.
- **consumers/**: Operaciones públicas del dominio. No contienen implementación, sino la definición de qué se espera que una operación haga.
- **enrutador/**: Punto de exposición. Solo los contratos estrictamente necesarios deben ser accesibles desde fuera del módulo.

Reglas del módulo rules/:

1. No se puede implementar lógica operativa.
2. No puede importar use/, configs/, ni interface/.
3. Todo lo que aquí se define debe ser reutilizable, estable y orientado al dominio.
4. El módulo debe exponer únicamente lo definido en su enrutador.

Módulo use/: Implementaciones y operaciones. Este módulo se encarga de aplicar, ejecutar y concretar las reglas definidas en rules/. Aquí se encuentra la lógica activa del proyecto. Este Módulo es el único que puede acceder a orígenes de datos, integrar servicios externos o coordinar la ejecución de múltiples reglas.

- **consumers/**: Operaciones públicas implementadas. Son las únicas estructuras del core/ que pueden ser invocadas directamente desde interface/. Aquí se combinan entidades, validaciones y orígenes de forma ordenada.
- **services/**: Implementaciones técnicas de los contratos definidos en rules/services/. Por ejemplo, una implementación concreta de un sistema de almacenamiento.
- **adapters/**: Transformadores entre estructuras internas del dominio y estructuras externas. Son el puente entre lo abstracto y lo concreto, lo técnico y lo semántico.
- **origins/**: Comunicación con bases de datos, archivos, colas, sockets u otros sistemas externos. Todo acceso a estos orígenes debe pasar por aquí.
- **enrutador/**: Punto de exposición obligatorio. Solo consumers/ debe estar disponible desde otras Capas.

Reglas del módulo use/:

1. Puede importar rules/, pero nunca configs/ ni interface/.
2. Solo consumers/ puede ser accedido desde fuera del core/.
3. Todo acceso a orígenes de datos debe pasar por origins/.
4. Los adaptadores deben ser utilizados para todo tipo de transformación entre estructuras internas y externas.
5. Ningún archivo puede ser accedido directamente si no ha sido expuesto por el enrutador del módulo.

La Capa INTERFACE

La Capa interface/ es la única puerta de entrada autorizada al proyecto. Su función no es transformar datos ni aplicar reglas, sino canalizar solicitudes externas hacia las operaciones públicas del núcleo lógico (core/) y devolver una respuesta conforme a los requerimientos de entrada.

Es aquí donde el proyecto se comunica con el mundo exterior: rutas HTTP, comandos de consola, listeners de eventos o cualquier otro mecanismo que active funcionalidades internas. Pero CMI no permite que esta Capa asuma más funciones

que las estrictamente necesarias. No debe contener lógica del dominio, no debe tomar decisiones estratégicas, y no debe modificar estructuras de negocio. Su único trabajo es **exponer**, **formatear** y **redirigir**.

Esta claridad funcional permite que el backend crezca, se integre o se reestructure sin temor a que el punto de entrada condicione el núcleo del sistema.

Estructura de entrada por tipo de canal

El proyecto puede exponer su funcionalidad a través de distintos canales, cada uno con su propia organización interna. Todos los canales deben estar organizados en Módulos independientes dentro de interface/.

a) Entrada HTTP (http/)

- **routes/**: Define las rutas, métodos, middlewares y versiones expuestas a través de la red.
- **handlers/**: Ejecutores que invocan core/use/consumers/ y formatean la respuesta. No deben contener lógica de negocio.
- **enrutador/**: Expone las rutas del módulo HTTP.

b) Entrada CLI (cli/)

- **commands/**: Agrupa los comandos del proyecto. Cada comando debe activar un consumer sin aplicar lógica intermedia.
- **options/**: Define las banderas, argumentos o configuraciones de entrada.
- **enrutador/**: Expone solo los comandos autorizados.

c) Entrada por eventos (events/)

- **listeners/**: Reaccionan ante eventos del sistema o externos. No procesan directamente datos del dominio.
- **enrutador/**: Expone los listeners activos.

d) Entrada compartida (shared/)

- **middlewares/**: Middlewares reutilizables que serán aplicados por múltiples puntos de entrada.
- **enrutador/**: Controla estrictamente qué puede ser accedido por otros Módulos de interface/.

Patrones de Diseño en CMI

CMI no es una arquitectura basada en patrones importados. Es una arquitectura basada en **estructura, jerarquía y encapsulamiento disciplinado**. Sin embargo, en su implementación, varios de sus componentes reproducen comportamientos que históricamente han sido definidos como "patrones de diseño".

Estos patrones **no se utilizan como herramientas externas** que el desarrollador puede aplicar o ignorar a voluntad, sino que **emergen como una consecuencia directa** de aplicar correctamente los principios y estructuras de CMI. De hecho, **funcionar fuera de ellos implicaría romper la arquitectura**, ya que el uso de enrutadores, adaptadores, delegación o separación de responsabilidades **no es una sugerencia**, sino parte del diseño formal.

Por tanto, aunque CMI no se presenta como una arquitectura "de patrones", su estructura reproduce muchos de ellos de forma natural. Reconocerlos permite comprender mejor **por qué CMI se comporta como lo hace**, y qué beneficios estructurales aporta: bajo acoplamiento, alta cohesión, control de visibilidad, entrada controlada y escalabilidad modular.

Este capítulo no busca enseñar los patrones desde cero, sino **mostrar cómo se manifiestan de forma obligatoria y coherente** dentro de un proyecto backend organizado bajo la arquitectura CMI.

Singleton

El patrón Singleton asegura que exista una única instancia de una clase a lo largo del proyecto y que pueda accederse a ella desde cualquier lugar que lo requiera, sin necesidad de múltiples construcciones. En la arquitectura CMI backend, este patrón se utiliza de manera controlada exclusivamente en la **Capa configs/**, donde ciertos servicios técnicos como el sistema de registros (logger) o el cargador de variables de entorno deben mantenerse únicos y accesibles desde cualquier punto del proyecto.

Este patrón **no debe utilizarse** en la Capa core/ ni en interface/. Usarlo fuera de configs/ representa una violación de los principios de encapsulamiento y responsabilidad única. En CMI, el Singleton no es una estrategia del desarrollador: es un comportamiento estructurado y limitado que aparece cuando se necesita garantizar consistencia de infraestructura sin romper la modularidad.

Ejemplo en CMI:

```dart
// configs/logger/logger_singleton.dart

class Logger {
  Logger._internal(); // Constructor privado
  static final Logger _instance = Logger._internal();

  factory Logger() => _instance;
```

```
8
9   void log(String message) {
10    print('[LOG] $message');
11  }
12 }
```

***Listing 1**. Singleton*

Factory

El patrón Factory permite crear objetos sin exponer directamente la lógica de construcción ni los detalles de sus dependencias. En lugar de instanciar clases manualmente, se delega la creación a una función o clase especializada, lo que permite cambiar la implementación de forma transparente y desacoplada.

En la arquitectura CMI backend, este patrón aparece de manera natural cuando una Capa superior necesita **obtener una implementación sin conocer sus detalles internos**, respetando el principio de inversión de dependencias. Se emplea, por ejemplo, cuando la Capa interface/ requiere usar una operación expuesta por core/, o cuando un consumer/ de core/use/ necesita construir un adapter/ o un service/ sin acoplarse a su implementación concreta.

Ejemplo en CMI:

```
1 // core/rules/services/storage_service_rule.dart
2
3 abstract class StorageService {
4   Future<void> save(String key, String value);
5 }
6
7 // core/use/services/local_storage_service.dart
8
9 import '../../rules/services/storage_service_rule.dart';
10
11 class LocalStorageService implements StorageService {
12   @override
13   Future<void> save(String key, String value) async {
14     // Guardado local simulado
15     print('Guardando [$key: $value]');
16   }
17 }
18 // core/use/services/factory_storage_service.dart
19
20 import 'local_storage_service.dart';
21 import '../../rules/services/storage_service_rule.dart';
```

```
22
23 StorageService createStorageService() {
24   return LocalStorageService();
25 }
```

Listing 2. *Factory*

En este ejemplo, createStorageService() actúa como una fábrica que entrega la implementación concreta sin exponer detalles técnicos. Si en el futuro se desea reemplazar LocalStorageService por otra implementación (como una basada en base de datos o en la nube), solo se modifica la fábrica.

Repository

El patrón Repository permite separar la lógica que accede a los datos del resto de la aplicación, encapsulando las operaciones técnicas de almacenamiento, consulta y transformación en una interfaz consistente. Esto facilita que otras estructuras del proyecto trabajen con datos sin preocuparse por la fuente, el formato ni la infraestructura.

En la arquitectura CMI backend, este patrón está implícitamente presente en los contenedores origins/ y services/ del módulo core/use/. Las operaciones de lectura o escritura se realizan a través de contratos definidos en core/rules/, y las implementaciones reales —ya sea hacia una base de datos, un archivo o una API— se encapsulan y aíslan dentro de use/.

Este patrón evita que un consumer/ o un adapter/ acceda directamente a una tecnología concreta. En su lugar, accede a través de una interfaz (el contrato) y una implementación aislada.

```
1  // core/rules/services/user_repository_rule.dart
2
3  abstract class UserRepository {
4    Future<void> createUser(String id, String name);
5    Future<String?> getUserName(String id);
6  }
7
8  // core/use/services/user_repository_impl.dart
9
10 import '../../rules/services/user_repository_rule.dart';
11
12 class InMemoryUserRepository implements UserRepository {
13   final _storage = <String, String>{};
14
15   @override
16   Future<void> createUser(String id, String name) async {
```

```
    _storage[id] = name;
  }

  @override
  Future<String?> getUserName(String id) async {
    return _storage[id];
  }
}

// core/use/services/factory_user_repository.dart

import 'user_repository_impl.dart';
import '../../rules/services/user_repository_rule.dart';

UserRepository createUserRepository() {
  return InMemoryUserRepository();
}
```

***Listing 3**. Repository*

En este ejemplo, la Capa core/rules/ define el contrato UserRepository, y la implementación se desarrolla en use/. Gracias a este patrón, el consumer/ que use este repositorio no necesita saber si los datos están en memoria, en disco o en un servicio externo: solo conoce el contrato.

Adapter

El patrón Adapter permite transformar una estructura en otra sin modificar las clases originales. Su objetivo es hacer compatible una interfaz externa con una estructura interna que no fue diseñada para interactuar directamente. En CMI, este patrón es uno de los más relevantes, ya que se aplica de forma estructural dentro del módulo core/use/.

Los adaptadores en CMI permiten que datos provenientes de orígenes externos (como APIs, bases de datos, archivos o incluso comandos CLI) se conviertan en entidades del dominio, y que estructuras internas puedan prepararse para ser devueltas como respuestas o almacenadas en formatos distintos. Esta separación es obligatoria: **ninguna estructura externa debe ingresar al dominio sin pasar por un adaptador**, y ningún resultado del dominio debe salir directamente sin ser adaptado al formato adecuado.

Ejemplo en CMI:

```
// core/rules/data/user_entity.dart

class User {
```

```
4   final String id;
5   final String name;
6
7   User({required this.id, required this.name});
8 }
9
10 // core/use/adapters/user_adapter.dart
11
12 import '../../rules/data/user_entity.dart';
13
14 class UserAdapter {
15   static User fromMap(Map<String, dynamic> map) {
16     return User(
17       id: map['id'] as String,
18       name: map['name'] as String,
19     );
20   }
21
22   static Map<String, dynamic> toMap(User user) {
23     return {
24       'id': user.id,
25       'name': user.name,
26     };
27   }
28 }
```

***Listing 4**. Adapter*

Este adaptador permite convertir datos externos (por ejemplo, desde un formulario, JSON o base de datos) a la entidad User, y viceversa. Su existencia permite que los consumers/ y los servicios de almacenamiento operen solo sobre estructuras del dominio, y no dependan del formato de origen ni de destino.

Dependency Injection (DI)

El patrón Dependency Injection (DI) consiste en entregar a una clase las dependencias que necesita desde el exterior, en lugar de crearlas internamente. Esto permite reducir el acoplamiento, facilitar las pruebas y aumentar la flexibilidad al cambiar implementaciones sin modificar el consumidor.

En CMI backend, este patrón aparece de forma natural cuando un consumer/ necesita utilizar un service/ o un repository, pero no conoce su implementación concreta. En lugar de instanciar la dependencia directamente, esta le es **inyectada** al momento de su construcción, en función de los contratos definidos en rules/. Este

comportamiento permite cumplir estrictamente con el principio de inversión de dependencias.

Aunque CMI no impone un sistema de inyección de dependencias automatizado, sí estructura el proyecto para que esta práctica se aplique de manera explícita y controlada.

Ejemplo en CMI:

```dart
// core/rules/services/user_repository_rule.dart

abstract class UserRepository {
  Future<void> createUser(String id, String name);
}

// core/use/services/user_repository_impl.dart

import '../../rules/services/user_repository_rule.dart';

class InMemoryUserRepository implements UserRepository {
  final _data = <String, String>{};

  @override
  Future<void> createUser(String id, String name) async {
    _data[id] = name;
  }
}

// core/use/consumers/user_creator_consumer.dart

import '../../rules/services/user_repository_rule.dart';

class UserCreator {
  final UserRepository repository;

  UserCreator({required this.repository});

  Future<void> execute(String id, String name) {
    return repository.createUser(id, name);
  }
}

// core/use/factory/factory_user_creator.dart

import '../services/user_repository_impl.dart';
```

```
37 import '../consumers/user_creator_consumer.dart';
38
39 UserCreator createUserCreator() {
40   return UserCreator(repository: InMemoryUserRepository());
41 }
```

Listing 5. *Dependency Injection (DI)*

En este ejemplo, UserCreator no crea su propio repositorio, sino que lo recibe inyectado. Esto permite cambiar la implementación del repositorio en pruebas, o en diferentes entornos, sin modificar la lógica del consumer/.

Observer

El patrón Observer permite que un objeto notifique a otros cuando su estado cambia, sin conocer sus identidades ni su lógica. Es especialmente útil cuando existen múltiples componentes que deben reaccionar ante un mismo evento o acción sin acoplarse directamente entre sí.

En CMI backend, este patrón aparece de forma natural en proyectos que manejan entradas asincrónicas o basadas en eventos, como colas de mensajes, WebSockets, streams de datos o sistemas de notificación. CMI estructura este comportamiento a través del módulo events/ dentro de la Capa interface/, donde se definen **listeners** que actúan como observadores. Cada uno escucha un tipo de evento y activa un consumer/ correspondiente sin alterar el flujo global del proyecto.

Ejemplo en CMI:

```
1 // interface/events/listeners/user_created_listener.dart
2
3 import '../../../core/use/consumers/user_logger_consumer.
     dart';
4 import '../../../core/use/adapters/user_adapter.dart';
5
6 Future<void> onUserCreatedEvent(Map<String, dynamic> payload
     ) async {
7   final user = UserAdapter.fromMap(payload);
8   final logger = UserLogger();
9
10   await logger.log(user);
11 }
```

Listing 6. *Observer*

En este ejemplo, el listener onUserCreatedEvent reacciona a un evento externo y notifica a un consumer/ del dominio (UserLogger). No transforma datos de negocio, ni valida lógica. Su único rol es observar e informar.

Strategy

El patrón Strategy permite definir una familia de algoritmos o comportamientos intercambiables y encapsularlos por separado, permitiendo que el comportamiento concreto se elija dinámicamente sin modificar la lógica que lo utiliza. Este patrón es ideal cuando se desea mantener el mismo flujo general pero variar su ejecución según el contexto.

En la arquitectura CMI backend, este patrón se manifiesta cuando un consumer/ necesita aplicar una lógica que puede tener múltiples variantes, como diferentes métodos de autenticación, formas de evaluación o políticas de cálculo. En lugar de codificar condicionales dentro de la operación, CMI exige definir **contratos en rules/** y **múltiples implementaciones en use/**, para que la estrategia a utilizar pueda ser inyectada y modificada sin alterar el flujo principal.

Ejemplo en CMI:

```
1  // core/rules/services/discount_strategy_rule.dart
2
3  abstract class DiscountStrategy {
4    double apply(double total);
5  }
6
7  // core/use/services/percentage_discount.dart
8
9  import '../../rules/services/discount_strategy_rule.dart';
10
11 class PercentageDiscount implements DiscountStrategy {
12   @override
13   double apply(double total) => total * 0.9; // 10% de
       descuento
14 }
15
16 // core/use/services/fixed_discount.dart
17
18 import '../../rules/services/discount_strategy_rule.dart';
19
20 class FixedDiscount implements DiscountStrategy {
21   @override
22   double apply(double total) => total - 5.0;
23 }
24
25 // core/use/consumers/cart_consumer.dart
26
27 import '../../rules/services/discount_strategy_rule.dart';
```

```
28
29 class CartConsumer {
30   final DiscountStrategy strategy;
31
32   CartConsumer({required this.strategy});
33
34   double checkout(double total) {
35     return strategy.apply(total);
36   }
37 }
```

***Listing 7**. Strategy*

Este patrón permite que CartConsumer funcione con cualquier estrategia que se le asigne. Si mañana se desea usar una estrategia basada en cupones, basta con crear una nueva clase que implemente DiscountStrategy y no modificar ni una línea del consumer.

Consideraciones al Usar Patrones de Diseño en CMI

Los patrones de diseño no deben entenderse en CMI como herramientas adicionales que el desarrollador puede usar para embellecer o "mejorar" la arquitectura, sino como **comportamientos estructurales que surgen inevitablemente del diseño jerárquico, modular y encapsulado** que la arquitectura impone.

No es CMI quien adopta los patrones: es el uso correcto de CMI el que reproduce muchos de estos patrones, incluso sin nombrarlos. Esto significa que, si bien no es obligatorio que el desarrollador los reconozca por su nombre, **sí es obligatorio que respete sus estructuras, roles y límites dentro del proyecto**, ya que son parte natural del funcionamiento de la arquitectura.

Algunas advertencias clave:

- **Los patrones en CMI no son intercambiables**. No se puede reemplazar una implementación del patrón Adapter con lógica directa en un handler, ni sustituir un Repository con acceso directo al origen de datos. Cada patrón representa una función clara dentro de una Capa, un Módulo y un Contenedor definidos.
- **No deben aplicarse de forma decorativa o superflua**. Añadir una estructura con forma de patrón sin que responda a una necesidad real dentro del flujo estructurado puede romper la cohesión del Módulo o sobrecomplicar el mantenimiento.
- **Su implementación debe estar limitada a su Capa correspondiente**. Por ejemplo, el patrón Singleton solo puede usarse en configs/, el Observer en

interface/events/, y el Adapter en core/use/adapters/. Aplicarlos fuera de su contexto natural rompe la jerarquía de responsabilidades.

- **Cada patrón está subordinado al principio de visibilidad controlada**. Ningún patrón puede ser expuesto sin pasar por su enrutador correspondiente, ni puede acceder directamente a elementos internos de otras estructuras sin seguir la ruta formal de importación establecida por la arquitectura.

En resumen, **los patrones no son añadidos a CMI, son consecuencias de aplicarla bien**. Respetar su estructura no es una opción: es una extensión natural de los principios fundamentales de la arquitectura.

Patrones Fundamentales en la Arquitectura CMI

Table 3. *Patrones Fundamentales en la Arquitectura CMI*

Patrón	**Capa/Módulo Recomendado**	**Propósito Principal**
Singleton	Configs	Instancia global controlada
Factory	Core/Use	Crear implementaciones flexibles
Repository	Core/Rules - Core/Use	Abstraer acceso a datos
Adapter	Core/Use (Adapters)	Convertir datos externos
Dependency Injection	UI - Core/Use	Gestionar dependencias
Observer	interface/events/	Reaccionar a eventos externos
Strategy	core/use/consumers/	Lógica intercambiable

Patrones Internos y Compatibilidad con la Arquitectura CMI

En la práctica del desarrollo backend moderno, es común que frameworks, librerías o entornos de trabajo impongan sus propios patrones internos para organizar código, gestionar dependencias o responder a flujos de datos. Estas soluciones pueden ser útiles, pero su incorporación dentro de una arquitectura como CMI **debe realizarse con cuidado y bajo condiciones estrictas**.

CMI no rechaza la incorporación de patrones externos, pero **no cede su estructura a ellos**. La arquitectura define jerarquías inquebrantables, flujos de dependencia controlados y mecanismos formales de encapsulamiento. Cualquier patrón, clase generada, anotación o herramienta externa que pretenda integrarse debe **respetar esa estructura**, no desplazarla ni adaptarla a sus propios fines.

Esta sección no busca evaluar la calidad de esos patrones, sino establecer **cuándo y cómo su uso es compatible con CMI**. Porque en esta arquitectura, **la compatibilidad no depende de lo que se desea hacer, sino de cómo y desde dónde se hace**.

¿Qué son los Patrones Internos?

Se conoce como **patrones internos** a aquellos comportamientos arquitectónicos que surgen desde una herramienta, entorno o librería externa, y que organizan o gobiernan el código de un proyecto según reglas propias, ajenas a la arquitectura definida por el proyecto en sí.

Estos patrones **no son explícitamente creados por el desarrollador**, sino que se introducen al usar frameworks que generan estructuras, aplican convenciones automáticas o imponen formas de interacción entre componentes. Ejemplos comunes incluyen: ciclos de vida gestionados por el framework, anotaciones que generan clases o accesos automáticos, inyecciones preconfiguradas, o estructuras que asumen rutas de acceso predeterminadas.

En muchos entornos modernos de desarrollo backend, estos patrones son promovidos como parte del "estilo del framework", y pueden facilitar la productividad inicial. Sin embargo, su incorporación directa en un proyecto basado en CMI puede provocar conflictos si no se encapsulan adecuadamente, ya que tienden a:

- Romper la **jerarquía de Capas** definida por la arquitectura.
- Introducir **acoplamientos ocultos** entre módulos o estructuras.
- Omitir el uso de **enrutadores**, accediendo directamente a elementos internos.
- Desobedecer los contratos formales definidos en core/rules/.

En otras palabras, **un patrón interno puede parecer útil o funcional, pero si atraviesa la organización estructural impuesta por CMI, deja de ser compatible**.

Por esta razón, todo patrón interno debe ser analizado no solo por su utilidad técnica, sino por su **capacidad de adaptarse sin quebrar la arquitectura**. Si no puede ser encapsulado dentro de su Capa correspondiente, accedido solo por su enrutador, e integrado mediante un contrato, **no debe formar parte del proyecto**.

Compatibilidad de CMI con Patrones Internos

La Arquitectura CMI es **totalmente compatible** con el uso de patrones internos, siempre que se respeten los principios básicos de:

- **Separación de responsabilidades.**
- **Bajo acoplamiento entre elementos.**
- **Alta cohesión dentro de módulos.**
- **Modularidad clara en la estructura.**

En esencia:

CMI organiza el proyecto estructuralmente; los patrones internos organizan el flujo de datos y la interacción entre componentes. (Nota - Compatibilidad de CMI con Patrones Internos)

Ambos enfoques no se contradicen, sino que **se complementan**.

Compatibilidad específica con Patrones Internos

Table 4. *Compatibilidad específica con Patrones Internos*

Patrón Interno	Compatibilidad con CMI	Notas
ORM (Object-Relational Mapping)	Totalmente compatible	Puede utilizarse dentro de use/origins/ si se encapsula y accede solo a través de contratos definidos en rules/.

Continued on next page

Table 4. *Compatibilidad específica con Patrones Internos* (Continued)

Event-driven (Listeners / Pub-Sub)	Totalmente compatible	Debe ubicarse dentro de interface/events/ y activar consumers/. No debe contener lógica de negocio.
IoC Container / Service Locator	Parcialmente compatible	Solo es aceptable si se usa para inyección de dependencias **desde interface/ hacia core/use/**, y nunca como acceso global directo.
CQRS (Command Query Responsibility Segregation)	Compatible	Puede implementarse con consumers/ especializados para comandos y consultas, si se respeta la separación por Módulo.
Clean Architecture	Parcialmente compatible	Algunos de sus principios ya existen en CMI. No debe mezclarse la jerarquía propuesta por Clean con la estructura fija de Capas y Enrutadores en CMI.

Ejemplo Conceptual de Integración

```
interface/
 http/
      routes/
      handlers/
      http.dart

 events/
      listeners/
```

```
     events.dart

 shared/
      middlewares/
      shared.dart

core/
 rules/
     data/
     services/
     consumers/
     rules.dart

 use/
     consumers/
     services/
     adapters/
     origins/
     uses.dart

 core.dart
```

***Listing 8**. Organización externa (CMI)*

Organización interna (por ejemplo, MVC):

- **Listener**: Se define en interface/events/listeners/ y escucha el evento.
- **Consumer**: Se invoca desde el listener y reside en core/use/consumers/.
- **Adapter**: Convierte los datos externos y se encuentra en core/use/adapters/.
- **Entidad de dominio**: Está definida en core/rules/data/.

Así, un flujo de login podría ser:

- onUserCreatedEvent activa RegisterUserConsumer usa UserAdapter.fromMap() instancia UserEntity

Todo este proceso se ejecuta **sin romper la estructura jerárquica ni las responsabilidades de cada Capa**, demostrando cómo se integran patrones internos (como Observer y Adapter) dentro de los límites definidos por la arquitectura CMI.

Estructuras Estándar de la Arquitectura CMI

La **Arquitectura CMI** establece una organización jerárquica basada en tres niveles principales:

Capa → Módulo → Contenedor

Dentro de esta jerarquía, existen las denominadas **estructuras estándar o dominio**, las cuales son de **uso obligatorio** para garantizar la coherencia, escalabilidad y mantenibilidad de cualquier proyecto. Estas estructuras definen cómo debe organizarse el código desde su base, asegurando que todos los desarrolladores trabajen bajo un mismo esquema.

Sin embargo, CMI también contempla la necesidad de adaptarse a proyectos más complejos o específicos. Por ello, junto a las estructuras estándar y dominio, permite la creación de **módulos** y **contenedores adicionales**, siempre que se respeten los principios fundamentales de la arquitectura.

Cuando se requiere una organización más profunda, es posible extender la jerarquía mediante **contenedores recursivos**, manteniendo el orden sin sacrificar flexibilidad.

Este capítulo presenta las estructuras estándar y dominio obligatorias y explica cómo y cuándo se pueden ampliar mediante estructuras adicionales.

Capas Estándar

Las **capas** son la división principal y más rígida de CMI. Son **fijas**, no pueden ser modificadas, eliminadas ni ampliadas. Toda aplicación bajo CMI debe estructurarse sobre estas tres capas:

Table 5. *Capas Estándar*

Capa	Descripción
configs/	Gestión de las **configuraciones técnicas globales**.
core/	Lógica de negocio y datos.
interface/	Entradas y salidas del sistema: HTTP, CLI, eventos, etc.

Estas capas aseguran la separación de responsabilidades a nivel macro, evitando mezclas entre configuración, lógica de negocio y mecanismos de comunicación con el exterior del backend.

Reglas:

1. Las capas en CMI están **predefinidas** y deben mantenerse constantes en cualquier implementación, garantizando una estructura coherente y universal.
2. Cada capa ocupa un lugar específico dentro del proyecto, formando parte de la **raíz estructural**. Esto asegura que la organización sea predecible y fácil de navegar.
3. La arquitectura está diseñada para operar con un número limitado de capas, evitando la proliferación innecesaria de divisiones que puedan complicar la gestión del proyecto.
4. La interacción entre capas debe ser siempre **controlada y ordenada**, aplicando los principios de **abstracción**, **bajo acoplamiento** y utilizando mecanismos claros de comunicación, como los enrutadores.
5. Cada capa debe gestionar su comunicación con el exterior a través de un **punto de acceso controlado**, asegurando que solo se exponga lo necesario y manteniendo encapsulada su lógica interna.
6. Toda capa debe cumplir una función definida y contener las subdivisiones necesarias (módulos o estructuras equivalentes) que permitan gestionar sus responsabilidades de manera eficiente.

Módulos

Módulos Estándar

Cada capa contiene módulos que organizan sus responsabilidades internas. Los siguientes módulos son **obligatorios** dentro de sus respectivas capas:

En la Capa core/:

Table 6. *Módulos Estándar - Capa core*

Módulo	Descripción

Continued on next page

Table 6. *Módulos Estándar - Capa core* (Continued)

rules/	Encargado de definir las abstracciones de la aplicación . Aquí se responde a la pregunta "*¿Qué debe hacer mi aplicación?*".
use/	Responsable de implementar esas abstracciones. Aquí decides "*¿Cómo lo va a hacer?*".

En la Capa interface/:

Table 7. *Módulos Estándar - Capa interface*

Módulo	**Descripción**
http/	Punto de entrada a través de servidores HTTP (como Shelf, Express, FastAPI, etc.).
cli/	Punto de entrada mediante comandos ejecutables o programas de línea de comandos.
events/	Punto de entrada orientado a eventos, listeners o suscripciones externas.
shared/	Elementos transversales compartidos entre distintos puntos de entrada (como middlewares reutilizables).

Módulos Adicionales

CMI permite la creación de **módulos adicionales** cuando el proyecto lo requiere, por ejemplo, para gestionar funcionalidades específicas que no encajan en los módulos estándar. Estos módulos deben:

- Mantenerse dentro de la jerarquía **Capa → Módulo**.
- Respetar las normas de organización interna (uso de contenedores y enrutadores).
- Ser coherentes con los principios de separación y claridad.

Reglas:

1. Todo módulo debe tener un **propósito específico**, representando una funcionalidad o dominio claramente identificado dentro de la capa en la que se encuentra.
2. Los módulos deben ser **independientes entre sí**, evitando dependencias innecesarias que puedan generar acoplamiento excesivo.
3. La estructura de un módulo debe facilitar la **reutilización**, permitiendo su expansión o modificación sin afectar negativamente al resto de la aplicación.
4. Cada módulo debe gestionar su comunicación con el exterior a través de un **punto de acceso controlado**, asegurando que solo se exponga lo necesario y manteniendo encapsulada su lógica interna.
5. La creación de módulos debe responder a necesidades reales del sistema, evitando divisiones artificiales o genéricas que no aporten valor a la organización del proyecto.

Contenedores

Contenedores de Dominio

CMI contempla el uso de **contenedores de dominio** como parte de su estructura flexible y escalable.

Los contenedores de dominio permiten organizar el código según áreas funcionales o contextos específicos dentro de un módulo. No tienen nombres predefinidos, ya que dependen directamente de la lógica del proyecto.

Ejemplos de contenedores de dominio:

- authentication/
- users/
- tasks/
- notes/
- payments/

Estos contenedores actúan como **carpetas raíz internas**, y dentro de ellos es donde se aplican los **contenedores estándar** obligatorios.

```
interface/
 http/
      authentication/
          routes/
              container/
                  login_route.dart
                  logout_route.dart
              routes_authentication.dart
          handlers/
              container/
                  login_handler.dart
                  logout_handler.dart
              handlers_authentication.dart
          authentication.dart
```

Listing 9. *Ejemplos de contenedores de dominio*

Contenedores Estándar

Los contenedores son obligatorios dentro de los módulos o contenedores de dominio donde aplican. Organizan elementos como servicios, abstracciones, adaptadores, validadores o rutas.

Su propósito es mantener un orden interno estructurado, predecible y coherente con el principio de separación de responsabilidades.

En core/rules/:

El módulo rules/ define las **abstracciones** del sistema. Aquí, los contenedores estándar garantizan que las interfaces, la lógica abstracta y los modelos de datos estén correctamente separados.

Table 8. *Contenedores Estándar - Modulo core/rules*

Contenedor	Propósito
services/	Declaración de interfaces para servicios externos.

Continued on next page

Table 8. *Contenedores Estándar - Modulo core/rules* (Continued)

consumers/	Definición abstracta de la lógica de negocio.
data/	Modelos de datos puros, sin dependencias externas.

En core/use/:

El módulo use/ contiene las **implementaciones** concretas de las abstracciones definidas en rules/. Aquí, los contenedores estándar organizan las implementaciones, la adaptación de datos y el manejo de fuentes externas.

Table 9. *Contenedores Estándar - Modulo core/use*

Contenedor	**Propósito**
services/	Implementaciones concretas de los servicios.
adapters/	Transformación de datos externos a modelos internos y viciversa.
origins/	Manejo de datos crudos (APIs, bases de datos, etc.).
consumers/	Implementación de la lógica de negocio abstracta.

En interface /http/[dominio]/:

Los contenedores de dominio dentro de módulos como 'http/', 'cli/' o 'events/' siguen la misma estructura modular.

A continuación, se muestra un ejemplo de cómo se organiza el dominio 'authentication/' dentro del módulo 'http/', aplicando los contenedores estándar.

Table 10. *Contenedores Estándar - Modulo /http/[dominio]/(example)*

Contenedor	**Propósito**
routes/	Define las rutas y su estructura.
handlers/	Implementa la lógica que responde a las rutas.
middlewares/	Aplica funciones de verificación, seguridad o transformación de las solicitudes.

En ui/shared/:

Table 11. *Contenedores Estándar - Modulo interface/shared*

Contenedor	**Propósito**
middlewares/	Middlewares reutilizables entre rutas HTTP, CLI o eventos.
validators/	Validadores reutilizables de entrada o lógica externa.

Contenedores adicionales

Los **contenedores adicionales** son unidades de organización creadas dentro de un **contenedor estándar** o un **contenedor de dominio**. Su propósito principal es **extender la organización interna y permitir la recursividad estructural de la arquitectura.**

Permiten dividir la lógica, componentes o recursos dentro de un contenedor superior, manteniendo una estructura jerárquica sin romper los principios de modularidad y encapsulamiento.

Tipos de contenedores adicionales. A partir de su propósito y relación con el contenedor superior, los contenedores adicionales se dividen en **dos tipos:**

Contenedor adicional interno:

- Es un contenedor que **solo organiza elementos internos** dentro del contene-

dor estándar o de dominio.

- **No continúa la recursividad estructural**: no contendrá nuevos contenedores adicionales dentro.
- **No expone funcionalidades directamente** hacia otros contenedores o capas externas.
- **No requiere su propio enrutador.** Toda exposición externa se sigue gestionando a través del **enrutador del contenedor superior**.

```
services/
 authentication/
     container/
         email_auth_service.dart
         google_auth_service.dart
     authentication.dart
```

***Listing 10**. Contenedor adicional interno - Ejemplo*

Contenedor adicional expuesto:

- Es un contenedor que **sí continúa la recursividad estructural**: contendrá nuevos contenedores adicionales o subdivisiones internas.
- **Expone funcionalidades directamente** hacia el nivel superior.
- **Requiere su propio enrutador.** Esto asegura una frontera de acceso clara para su estructura interna y sus exposiciones públicas.

```
interface/
 http/
      reports/
          router/
          handlers/
          middlewares/
          exports/          ← contenedor adicional expuesto
              router/
              csv/
              pdf/
              excel/
              exports.dart
```

***Listing 11**. Contenedor adicional expuesto - Ejemplo*

Reglas:

1. **Todo contenedor adicional** requiere un enrutador si continuará organizando más contenedores adicionales en su interior.

2. **Todo contenedor adicional** requiere un enrutador si necesita exponer directamente funcionalidades o recursos a otros contenedores o capas.

3. Un **contenedor adicional interno** no requiere un enrutador si solo organiza elementos y su acceso sigue pasando por el enrutador del contenedor superior.

4. Un **contenedor adicional expuesto** debe declarar su propio enrutador incluso si depende parcialmente del enrutador del contenedor superior.

5. Un **contenedor adicional** nunca debe exponer directamente sin pasar por un enrutador (propio o del contenedor superior).

6. Un **contenedor adicional interno** solo puede existir si está anidado en un contenedor con enrutador que gestione sus exposiciones.

7. Si un **contenedor adicional interno** necesita crecer en el futuro, deberá migrar su estructura a un contenedor adicional expuesto creando su propio enrutador.

Gestión de la estructura de la aplicación

La **gestión de la estructura** en la Arquitectura CMI no consiste solo en seguir un conjunto de reglas predefinidas, sino en aplicar esas reglas con criterio, entendiendo que cada decisión estructural impacta directamente en la claridad, mantenibilidad y escalabilidad del proyecto.

En el capítulo anterior, se definieron las **estructuras de CMI**: las capas, módulos y contenedores que forman la base inmutable de CMI. Sin embargo, conocer estas estructuras no es suficiente. La verdadera fortaleza de CMI se demuestra cuando sabes **cómo implementar** esa jerarquía en un proyecto real, cómo expandirla cuando sea necesario y cómo mantener el orden a medida que el sistema crece.

Este capítulo te guiará paso a paso en la aplicación práctica de la estructura CMI, desde sus elementos más básicos hasta las variantes flexibles como módulos adicionales, contenedores de dominio y recursivos. Además, abordaremos buenas prácticas, criterios de expansión y errores comunes que debes evitar para no comprometer la integridad de tu arquitectura.

El objetivo es que no solo sigas un esquema, sino que **comprendas el porqué de cada decisión estructural** y construyas proyectos preparados para evolucionar sin caer en el desorden.

Aplicación Correcta de la Estructura CMI

En CMI, la estructura **no depende del tamaño del proyecto**. Ya sea una aplicación pequeña o un sistema complejo, siempre se debe aplicar la **estructura completa** desde el inicio. Esto se debe a que CMI está diseñada para proyectos que están destinados a crecer, y la mejor forma de garantizar un crecimiento ordenado es establecer una base sólida y coherente desde el primer día.

La Regla Principal: La Forma Siempre es Completa

No importa si hoy tienes solo un servicio o un par de vistas. La jerarquía de CMI —con sus capas, módulos, contenedores de dominio y contenedores estándar— debe estar presente desde el inicio.

Esto evita que en el futuro tengas que realizar reestructuraciones dolorosas cuando el proyecto escale, ya que todo estará listo para soportar nuevas funcionalidades de manera ordenada.

Ejemplo General de Estructura Inicial

A continuación, se presenta cómo debería lucir cualquier proyecto que implemente CMI correctamente desde el inicio, incluso si es un proyecto "pequeño":

```
lib/
 configs/                                # Enrutador principal de la capa configs/
   environments/
       env_config.dart                   # óConfiguracin por entorno (dev, prod, etc.)
       environment_keys.dart             # Claves de entorno (API, tokens, endpoints)
       environments.dart

   constants/
       app_constants.dart                # Constantes globales como timeout, maxSize, etc.
       constants.dart

   loggers/
       logger_singleton.dart             # Sistema de registros del proyecto
       loggers.dart

   configs.dart

 core/
```

```
19      rules/
20          services/
21          consumers/
22          data/
23          rules.dart
24
25      use/
26          services/
27          adapters/
28          origins/
29          consumers/
30          uses.dart
31
32      core.dart
33
34  interface/
35      http/
36          routes/
37          handlers/
38          http.dart
39
40      events/
41          listeners/
42          events.dart
43
44      cli/
45          commands/
46          cli.dart
47
48      shared/
49          middlewares/
50          shared.dart
51
52      interface.dart
53
54  main.dart                                   # Punto de entrada
      del backend
```

***Listing 12**. Ejemplo General de Estructura Inicial*

Jerarquía en Acción

La fortaleza de la Arquitectura CMI radica en su jerarquía bien definida. Cada proyecto, sin importar su complejidad, debe construirse siguiendo este flujo lógico:

> Capa → Módulo → Contenedor de Dominio (En caso Especiales) → Contenedores Estándar → Contenedores Internos (si aplica) (Jerarquía en Acción - Flujo)

A continuación, veremos cómo aplicar cada nivel de esta jerarquía, entendiendo el propósito de cada elemento y su correcta implementación.

Capas Estándar: La Base Inamovible

Toda aplicación CMI comienza con las **tres capas estándar**:

```
lib/
 configs/
 core/
 interface/
```

***Listing 13**. Capas Estándar: La Base Inamovible*

Estas capas representan la separación fundamental:

- **configs/**: Centraliza las configuraciones técnicas globales.
- **core/**: Contiene la lógica del dominio, las reglas de negocio y las implementaciones funcionales.
- **interface/**: Gestiona los puntos de entrada del sistema (HTTP, CLI, eventos, etc.) y canaliza las solicitudes hacia core/.

> Regla: Las capas son fijas, obligatorias y nunca deben ser modificadas, eliminadas o duplicadas. (Capas Estándar - Regla)

Módulos Estándar y Módulos Adicionales

Dentro de cada capa, defines los **módulos estándar** que organizan las responsabilidades principales.

Ejemplo en core/:

```
core/
 rules/
 use/
```

***Listing 14**. Módulos Estándar y Módulos Adicionales - Ejemplo en core/*

Ejemplo en interface/:

```
interface/
 http/
 events/
 cli/
```

Listing 15. *Módulos Estándar y Módulos Adicionales - Ejemplo en interface/*

Si el dominio del proyecto lo requiere, puedes crear **módulos adicionales**. Por ejemplo, en configs/ podrías añadir un módulo environment/ para gestionar configuraciones de entornos específicos.

> Criterio: Los módulos adicionales deben responder a necesidades claras, nunca ser creados sin propósito definido. (Módulos Estándar y Módulos Adicionales - Regla)

Contenedores de Dominio: Organizando por Funcionalidad

En módulos como http/, events/ o cli/, el siguiente paso es crear **contenedores de dominio** que agrupen elementos relacionados a una funcionalidad específica del sistema, como autenticación, usuarios, productos, entre otros.

Ejemplo en layouts/:

```
interface/
 http/
       authentication/
       users/
```

Listing 16. *Contenedores de Dominio: Organizando por Funcionalidad - Ejemplo en interface/*

Cada contenedor de dominio establece un contexto funcional claro dentro del proyecto backend, evitando mezclar rutas, controladores o manipuladores que pertenecen a áreas diferentes del sistema.

Esto facilita el mantenimiento, el escalado y la comprensión modular del código.

Contenedores Estándar: La Organización Interna Obligatoria

Dentro de cada contenedor de dominio, se aplican los **contenedores estándar** definidos por CMI.

Ejemplo:

```
interface/
 http/
```

```
      authentication/
          routes/
          handlers/
          middlewares/
          adapters/
          authentication.dart
```

Listing 17. *Contenedores Estándar: La Organización Interna Obligatoria - Ejemplo en un layout*

> Regla: Los contenedores estándar son obligatorios donde corresponda, aunque inicialmente tengan poco contenido. (Contenedores Estándar: La Organización Interna Obligatoria - Regla)

Contenedores Internos y Recursividad

Cuando la cantidad de elementos dentro de un contenedor estándar crece, puedes crear **contenedores internos** para mejorar la organización.

Ejemplo:

```
handlers/
 users/
     Container/
        get_user_handler.dart
        create_user_handler.dart
      users.dart
```

Listing 18. *Contenedores Internos y Recursividad - Ejemplo*

Puedes utilizar contenedores con nombre o genéricos (container/), según el caso.

> Criterio: La recursividad debe aplicarse solo cuando aporte claridad, evitando sobreestructurar innecesariamente. (Contenedores Internos y Recursividad - Criterio)

Enrutadores: Controlando el Acceso

Cada capa, módulo y contenedor en CMI debe contar con su **enrutador**, el cual actúa como único punto de acceso autorizado hacia el exterior de su contexto estructural.

En proyectos backend, los enrutadores permiten controlar qué contratos, funciones, servicios o estructuras están disponibles para otras Capas, Módulos o contenedores.

Ejemplo:

```
core/
 rules/
       services/
       consumers/
       data/
       rules.dart

interface/
 http/
       authentication/
           routes/
           handlers/
           middlewares/
           authentication.dart
```

***Listing 19**. Enrutadores: Controlando el Acceso - Ejemplo*

Los enrutadores refuerzan la abstracción y protegen la integridad de la estructura, evitando dependencias directas hacia carpetas internas.

Reglas:

- Todo enrutador (ya sea de capa, módulo o contenedor) es el único punto de acceso autorizado para exponer elementos internos hacia otras partes de la aplicación.
- Los enrutadores garantizan el control de acceso, el encapsulamiento y la comunicación ordenada entre estructuras, evitando accesos directos o dependencias indebidas.
- Los enrutadores deben respetar la jerarquía de CMI: capas → módulos → contenedores. Los contenedores genéricos no requieren enrutador, ya que usan el de su padre.
- La existencia de un enrutador solo se justifica si la estructura necesita exponer elementos fuera de su contexto; si todo su contenido es interno, no se requiere.
- Mantener el enrutador actualizado al agregar, eliminar o modificar los elementos que expone es esencial para preservar la coherencia y estabilidad del proyecto.

Criterios para Expandir la Estructura

La **Arquitectura CMI** está diseñada para ofrecer un equilibrio entre una estructura normativa sólida y la capacidad de adaptarse a las necesidades reales de cada proyecto. Aunque las capas, módulos y contenedores estándar son obligatorios,

existen situaciones donde es necesario **expandir la estructura** para mantener el orden a medida que el sistema crece.

Sin embargo, esta expansión debe realizarse siguiendo **criterios claros**. Expandir no significa crear carpetas por costumbre, sino tomar decisiones conscientes que respondan a problemas reales de organización y escalabilidad.

Creación de Módulos Adicionales

La estructura estándar de CMI está diseñada para cubrir casi todas las necesidades de un proyecto bien organizado. Por eso, la creación de **módulos adicionales** debe ser algo **poco frecuente** y siempre justificado.

Cada capa tiene un nivel distinto de flexibilidad para expandirse. Aquí te explicamos **cómo y cuándo** hacerlo correctamente.

Configs: La Capa Más Flexible. Esta capa está dedicada exclusivamente a almacenar **parámetros técnicos estáticos** que afectan al sistema completo y no están acoplados a una sola capa.

Su contenido no debe incluir lógica, validación, acceso a datos ni ningún recurso perteneciente a core/ o interface/. Los elementos en configs/ deben ser constantes, sin lógica interna, y **compartibles entre capas** de forma segura.

Por ejemplo, en configs/ pueden almacenarse configuraciones de entorno (environment_config.dart), constantes globales (app_constants.dart), claves (security_keys.dart), o configuraciones de registro (logger_config.dart).

En cambio, elementos como estructuras de entrada (comandos, rutas HTTP, listeners) deben residir en interface/, ya que están acoplados directamente al punto de entrada del sistema.

Esta capa **no debe usarse como un espacio genérico para cosas globales**, sino únicamente para aquellas que sean **transversales, sin lógica, sin estado y sin acoplamiento a una capa específica.**

En configs/ es normal crear módulos adicionales. Esta capa está pensada para crecer según las necesidades del proyecto.

Puedes añadir módulos como:

- **security/** Para gestionar certificados, claves o permisos compartidos.
- **environments/** Para configuraciones según entorno (dev, prod, staging).
- **loggers/** Para definir y centralizar el sistema de registros del backend.
- **Cualquier configuración global** que requiera orden adicional sin lógica activa.

Aquí expandir es parte natural de su función.

Core: La Capa Más Rígida. En core/, la regla es simple:

Regla: Si puedes resolverlo con rules/ y use/, no necesitas más módulos.
(Core: La Capa Más Rígida - Regla)

No importa si trabajas con APIs, servicios del dispositivo, sensores o almacenamiento local. Todo debe pasar por el flujo estándar de **services(origins adapters) consumers**.

Solo en casos **extremos**, como un motor autónomo de IA o procesamiento especializado que no encaje en este flujo, podrías crear un módulo adicional.

Interface: Usa Bien lo Estándar . La capa interface/ ya es flexible por diseño. Puntos de entrada como solicitudes HTTP, comandos CLI o eventos del sistema **se organizan perfectamente** en:

- **http/** Para entradas por red (REST, WebSockets, RPC, etc.).
- **cli/** Para comandos ejecutados desde terminal o scripts.
- **events/** Para integrar listeners que reaccionan a eventos del entorno o del sistema.
- **shared/** Para middlewares y transformadores comunes entre módulos de entrada.

Crear un módulo adicional en interface/ debe ser algo **muy excepcional**.

La clave está en **aprovechar correctamente los módulos estándar**, usando su enrutador y contenedores sin romper la jerarquía ni el flujo controlado hacia core/.

Uso de Contenedores Internos y Recursividad

La recursividad en contenedores es una herramienta poderosa para organizar grandes volúmenes de código dentro de los contenedores estándar, especialmente en services/, adapters/ o handlers/.

¿Cuándo aplicar recursividad?

- Cuando un contenedor estándar empieza a manejar demasiados elementos de diferentes tipos o propósitos.
- Para separar por categorías funcionales, por ejemplo:

```
handlers/
 users/
 sessions/
 auth/
```

***Listing 20**. Uso de Contenedores Internos y Recursividad - Ejemplo*

- Cuando necesitas agrupar elementos específicos sin justificar una categoría, puedes usar un contenedor genérico:

```
services/
 container/
```

Listing 21. *Uso de Contenedores Internos y Recursividad - Ejemplo Contenedor Generico*

Errores a evitar:

- Crear niveles innecesarios que compliquen la navegación del proyecto.
- Recursividad excesiva que dificulte localizar archivos.
- No usar enrutadores en contenedores con nombre que exponen elementos.

Uso Correcto de Contenedores Genéricos (container/)

El contenedor container/ es útil cuando necesitas agrupar elementos pero no hay una categoría definida o cuando el conjunto es muy específico del contexto.

Criterios para usar container/:

- Úsalo solo dentro de contenedores estándar o internos.
- No abuses de él para evitar pensar en una organización adecuada.
- Recuerda que **no requiere enrutador propio**, ya que depende del contenedor padre.

Ejemplo de Expansión Bien Aplicada

Supongamos que tu módulo shared/ en la capa interface/ empieza a manejar muchos tipos de middlewares reutilizables y validadores generales para diferentes entradas:

```
interface/
 shared/
      middlewares/
      validators/
          container/
              auth_validators.dart
              input_validators.dart
          validators_shared.dart      # Manejador úcomn para validaciones
      loggers/
```

Listing 22. *Ejemplo de Expansión Bien Aplicada*

Aquí se han creado dos **contenedores adicionales** (validators/ y loggers/), organizando elementos reutilizables que **no encajan en los contenedores estándar existentes**, sin romper la jerarquía ni mezclar responsabilidades.

Ejemplo Visual de Estructura CMI

A continuación, se presenta un ejemplo completo de cómo debe organizarse un proyecto siguiendo la **Arquitectura CMI (versión backend)**. Este ejemplo refleja la correcta aplicación de:

- Las **tres capas estándar**.
- Los **módulos estándar**.
- Los **contenedores de dominio** donde corresponda.
- Los **contenedores estándar** obligatorios.
- La presencia de **enrutadores** en cada nivel clave.
- Un uso básico de contenedores internos para organización adicional.

Este esquema es válido tanto para proyectos pequeños como para proyectos en crecimiento, ya que CMI siempre establece la estructura completa desde el inicio.

Estructura General del Proyecto CMI:

```
lib/
 main.dart
   # Punto de entrada de la óaplicacin backend.
   # Inicializa servicios y configura el servidor principal.

 configs/
     assets/
         container/
             images_assets.dart
         assets.dart

     theme/
             theme.dart

     configs.dart

 core/
     core.dart                    # Enrutador principal de
   la capa Core.
```

```
    rules/
        data/
            user/
                container/
                    user_data.dart
                user.dart
            data.dart
        services/
            authentication/
                container/
                    auth_service.dart
                authentication.dart
            services.dart
        consumers/
            authentication/
                container/
                    auth_consumer.dart
                authentication.dart
            consumers.dart
        rules.dart
    use/
         services/
            authentication/
                container/
                    email_auth_service.dart
                    google_auth_service.dart
                    apple_auth_service.dart
                authentication.dart
            services.dart
        adapters/
            authentication/
                container/
                    email_auth_adapter.dart
                    google_auth_adapter.dart
                    apple_auth_adapter.dart
                authentication.dart
            adapters.dart
        consumers/
            authentication/
                container/
                    auth_consumer.dart
                authentication.dart
```

```
            consumers.dart
        origins/
            authentication/
                container/
                    email_auth_origin.dart
                    google_auth_origin.dart
                    apple_auth_origin.dart
                authentication.dart
            origins.dart
         uses.dart

interface/
      http/
          authentication/
              routes/
                  container/
                      login_route.dart
                      logout_route.dart
                  routes_authentication.dart

              handlers/
                  container/
                      login_handler.dart
                      logout_handler.dart
                  handlers_authentication.dart

              middlewares/
                  container/
                      auth_guard.dart
                  middlewares_authentication.dart

              authentication.dart

          http.dart

      cli/
          users/
              commands/
                  container/
                      create_user_command.dart
                      delete_user_command.dart
                  commands_users.dart
```

```
            users.dart

        cli.dart

    events/
        authentication/
            listeners/
                container/
                    on_user_registered_listener.dart
                listeners_authentication.dart

            authentication.dart

        events.dart

    shared/
        middlewares/
            container/
                logger_middleware.dart
            middlewares_shared.dart

        shared.dart
```

***Listing 23**. Estructura General del Proyecto CMI*

Convenciones de Nomenclatura

Una parte fundamental de la **Arquitectura CMI** es la consistencia en la manera de nombrar y estructurar cada elemento del proyecto. El respeto por estas normas garantiza claridad, facilita el trabajo en equipo y evita errores o confusiones en proyectos de cualquier tamaño.

Esta sección define las **reglas precisas** para nombrar **capas**, **módulos**, **contenedores**, **archivos** y, en especial, los **enrutadores**, asegurando que toda la estructura siga un estándar comprensible y mantenible.

Principios Generales

- **Contexto primero:** Todo nombre debe reflejar su lugar dentro de la arquitectura. La ruta + nombre del archivo deben ser suficientes para entender su propósito.
- **Evita nombres genéricos:** Nunca uses nombres como utils.dart, data.dart o router.dart sin especificar contexto.

- **Claridad sobre brevedad:** Prefiere nombres descriptivos aunque sean un poco más largos, antes que abreviaciones o nombres ambiguos.
- **Consistencia total:** Aplica las mismas reglas en todo el proyecto, sin excepciones.

Nomenclatura de Capas

Las capas tienen nombres **predefinidos y únicos**:

Table 12. *Nomenclatura de Capas*

Capa	Nombre de Carpeta
Configs	configs/
Core	core/
Interface	interface/

Estas carpetas no deben renombrarse, ya que forman la base estructural de CMI.

Nomenclatura de Módulos

Módulos por Defecto = Nombre Fijo y Estándar

En CMI, ciertos módulos ya están **predefinidos** por la arquitectura. Estos deben mantener su nombre **exacto y sin modificaciones**, porque representan funciones universales dentro del esquema de CMI.

Lista de Módulos por Defecto en CMI:

Table 13. *Módulos por Defecto = Nombre Fijo y Estándar*

Capa	Módulo por Defecto
Core	rules/, use/

Continued on next page

Table 13. *Módulos por Defecto = Nombre Fijo y Estándar* (Continued)

interface	http/, cli/, events/, shared/
Configs	Depende de la configuración (loggers/, environments/, validation/, etc.)

Reglas:

1. Mantener el nombre fijo: Los módulos como rules/, use/, shared/, http/, entre otros, deben conservar su nombre exacto.
2. Evitar redundancias en nombres de archivos: No repitas el nombre del módulo dentro de los archivos que ya están contextualizados por la ruta.
3. Enrutadores de módulos: Usa la convención [módulo]_[capa].dart para nombrar los enrutadores de estos módulos.
4. Contenedores: Si creas contenedores dentro de estos módulos, aplica las normas generales de nomenclatura, siempre priorizando la claridad y el contexto.

Nomenclatura de Contenedores

Los contenedores deben reflejar el tipo de elementos que agrupan:

Table 14. *Nomenclatura de Contenedores*

Tipo de Contenedor	**Nombre Sugerido**
Servicios	services/
Adaptadores	adapters/
Orígenes	origins/
Consumidores	consumers/

Continued on next page

Table 14. *Nomenclatura de Contenedores* (Continued)

Validadores	validators/
Comandos CLI	commands/
Escuchas/Eventos	listeners/
Middlewares	middlewares/
Rutas	routes/
Manejadores	handlers/
Componentes Técnicos Compartidos	shared/

Para subcontenedores específicos, añade descripciones claras:

```
services/container/         // éGenrico
services/authentication/    // íEspecfico por dominio
handlers/container/         // éGenrico
routes/users/               // íEspecfico por contexto
   funcional
```

Listing 24. *Nomenclatura de Contenedores*

Nomenclatura de Archivos

La estructura general para nombrar archivos es:

```
[nombre_base]_[tipo]_[contexto].dart
```

Listing 25. *Nomenclatura de Archivos-Estructura general*

Ejemplos Correctos:

***Table 15**. Nomenclatura de Archivos*

Elemento	**Nombre Correcto**
Middleware	auth_guard_middleware.dart
Handler HTTP	login_authentication_handler.dart
Ruta	register_route_authentication.dart
Adapter	google_auth_adapter.dart
Origin	user_api_origin.dart
Command CLI	delete_user_command_cli.dart
Event Listener	on_user_registered_listener.dart
Service Rule	task_service_rule.dart

Normas Específicas para Enrutadores

Los **enrutadores** son casos especiales dentro de la nomenclatura.
Reglas:

1. **El nombre del enrutador** debe ser el de la capa, módulo o contenedor que expone**:**
   (a) configs/configs.dart
   (b) core/rules/rules.dart
   (c) interface/http/authentication/authentication.dart
   (d) core/use/services/authentication/authentication.dart
2. **No uses nombres genéricos** como **router.dart** o **routes.dart.**
3. Si hay riesgo de confusión, añade el contexto superior:

   (a) handlers_authentication.dart
   (b) listeners_authentication.dart
   (c) commands_users.dart

4. Contenedores genéricos no deben tener enrutador propio: Su contenido debe ser expuesto desde el enrutador del contenedor padre.

Manejo de Nombres Repetidos

Si te encuentras con posibles conflictos de nombres:

- **Agrega el contexto funcional:**
  - En lugar de repetir data.dart, usa: user_data_rule.dart y message_data_rule.dart.
- **Evita abreviaturas oscuras o genéricas.**
- Siempre prioriza la legibilidad y la identificación rápida del propósito del archivo.

Clases

La correcta elección de nombres en las clases es fundamental para mantener la claridad, coherencia y mantenibilidad dentro de la Arquitectura CMI.

Reglas Generales:

1. **Descriptivos y Precisos:** Utiliza nombres que reflejen claramente la **responsabilidad** y el **propósito** de la clase. Evita términos genéricos como Manager, Handler o Data sin contexto adicional.
2. **Uso de Sustantivos Claros:** Las clases deben representar **entidades**, **conceptos** o **roles** dentro del dominio de la aplicación. Ejemplos correctos:
   (a) UserConsumerRule
   (b) AuthServiceUse
   (c) LoggerMiddleware
   (d) EmailCommandCli
3. **Consistencia con la Estructura CMI** Los nombres deben facilitar la comprensión inmediata de **dónde** y **para qué** se usa cada clase:
   (a) En rules/: usar sufijos como Rule para indicar abstracción.
   (b) En use/: usar sufijos como Use para señalar implementación.

(c) En adapters/: terminar en AdapterUse.

(d) En origins/: terminar en OriginUse.

(e) En middlewares/: terminar en Middleware.

(f) En cli/commands/: terminar en CommandCli.

(g) En events/listeners/: terminar en Listener.

4. **Evitar Abreviaturas Ambiguas:** No utilices siglas o abreviaturas que no sean universalmente reconocidas. Prefiere siempre la **claridad** sobre la brevedad excesiva.

   (a) Ejemplo incorrecto: UsrCnsmrR

   (b) Ejemplo correcto: UserConsumerRule

Convenciones de Sufijos en Clases:
Cada clase debe terminar con un sufijo que indique su propósito, según su ubicación en la estructura CMI.

Table 16. *Convenciones de Sufijos en Clases*

Ubicación	**Ejemplo de Clase**	**Sufijo**
core/rules/services/	AuthServiceRule	Rule
core/use/services/	AuthServiceUse	Use
core/use/adapters/	AuthAdapterUse	AdapterUse
core/use/origins/	UserApiOriginUse	OriginUse
interface/http/middlewares	AuthGuardMiddleware	Middleware
interface/cli/commands/	DeleteUserCommandCli	CommandCli
interface/events/listeners/	OnUserRegisteredListener	Listener

Continued on next page

Table 16. *Convenciones de Sufijos en Clases* (Continued)

configs/loggers/	LoggerConfig	Config
configs/environments/	EnvironmentsConfig	Config

Ejemplos Aplicados:

```
// Middleware para óautenticacin
class AuthGuardMiddleware {}

// Enrutador de comandos para usuario
class DeleteUserCommandCli {}

// Escuchador de evento
class OnUserRegisteredListener {}

// óAbstraccin de servicio
abstract class AuthServiceRule {}

// óImplementacin de servicio
class AuthServiceUse implements AuthServiceRule {}

// Adaptador bidireccional
class TaskAdapterUse {}

// Origen externo
class UserApiOriginUse {}
```

Listing 26. *Forma correcta de nombrar las clases*

Errores Comunes:

- Nombres vagos como Processor, Helper, Service sin contexto.
- Mezclar idiomas o estilos (ej: TareaService en un código en inglés).
- Usar nombres largos por describir **todo**. Busca el equilibrio.

Funciones

En la Arquitectura CMI, la nomenclatura de las funciones debe ser clara, precisa y coherente con las acciones que representan. Una función correctamente

nombrada facilita la lectura del código, mejora la comprensión del dominio de la aplicación y asegura la mantenibilidad a largo plazo.

A continuación, se establecen las reglas que deben seguirse para definir nombres de funciones dentro de cualquier proyecto basado en CMI.

Reglas Generales:

1. **Ser descriptivo y conciso:** El nombre de una función debe reflejar claramente su propósito. Se deben evitar términos genéricos o ambiguos que no indiquen con exactitud la operación que realiza. Es preferible utilizar nombres directos que comuniquen la acción sin necesidad de revisar la implementación de la función.
   (a) **Ejemplo correcto:**
      i. createNote()
      ii. deleteTask()
   (b) **Ejemplo incorrecto:**
      i. handleData()
      ii. processInfo()
2. **Utilizar verbos que indiquen acción:** Toda función debe comenzar con un **verbo** que denote la acción principal que ejecuta. Esto aplica a cualquier contexto, ya sea en Consumers, Services, Adapters o Logics. Los verbos deben ser claros y relacionados con el dominio de la aplicación.
   (a) **Ejemplos:**
      i. fetchNotes()
      ii. saveUserProfile()
      iii. toggleDarkMode()
3. **Mantener coherencia en el estilo:** Se debe utilizar el estilo **camelCase** para todas las funciones. Es obligatorio mantener un estilo uniforme en todo el proyecto, evitando mezclar idiomas o estilos de nomenclatura.
   (a) **Ejemplo correcto:** updateUserSettings()
   (b) **Ejemplo incorrecto:**
      i. Update_user_settings()
      ii. actualizarUsuario()
4. **Reflejar el dominio del problema:** Los nombres de las funciones deben incorporar términos propios del **dominio** en el que se está trabajando. Esto permite que el código sea más expresivo y entendible dentro del contexto de la aplicación.

   (a) **Ejemplo correcto:**
      i. archiveNote() (si la acción es específica de un sistema de notas)
   (b) **Ejemplo incorrecto:**
      i. setFlag() (demasiado genérico)

5. **Evitar verbos ambiguos:** Se deben evitar verbos como handle, process, manage, salvo que describan una acción concreta y necesaria dentro del dominio. Siempre que sea posible, optar por verbos más específicos.
   (a) **Ejemplo correcto:** assignTaskToUser()
   (b) **Ejemplo incorrecto:** handleTask()

```dart
// core/use/adapters/validators/container/email_validator.dart
bool isValidEmail(String email) {
  return RegExp(r'^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$').hasMatch(email);
}

// core/use/adapters/helpers/container/date_format_helper.dart
String formatDateDDMMYYYY(DateTime date) {
  return '${date.day.toString().padLeft(2, '0')}/'
         '${date.month.toString().padLeft(2, '0')}/'
         '${date.year}';
}

// core/use/adapters/helpers/container/string_case_helper.dart
String capitalize(String text) {
  if (text.isEmpty) return '';
  return text[0].toUpperCase() + text.substring(1).toLowerCase();
}

// core/use/adapters/helpers/container/path_generator.dart
String generateNoteDetailPath(String noteId) {
  return '/notes/details/$noteId';
}
```

***Listing 27**. Forma correcta de nombrar las funciones*

Metodos

En la Arquitectura CMI, los **métodos** son el principal medio para definir el comportamiento dentro de las clases, ya sea en Consumers, Services, Adapters, Logics o componentes UI. Por ello, su nomenclatura debe ser precisa, coherente y alineada con la responsabilidad de la clase donde se definen.

Reglas Generales:

1. **Usar verbos que indiquen acción:** Todo método debe comenzar con un **verbo** claro que describa la operación que realiza, reflejando la intención de manera directa.

2. **Evitar redundancia con el nombre de la clase:** Si la clase ya define el contexto o dominio (como NoteConsumerUse), no es necesario repetir ese contexto en el nombre del método.

   (a) **Ejemplo correcto:** create()

   (b) **Ejemplo incorrecto:** createNote()

3. **Ser específico y conciso:** El método debe describir claramente su propósito sin ser excesivamente largo ni ambiguo. Evitar términos genéricos como handle(), process(), o doWork().

4. **Métodos que gestionan eventos en UI:** Es recomendable usar el prefijo **on** para métodos que respondan a eventos, especialmente en Logic o Componentes:

   (a) onSubmitPressed()

   (b) onNoteSelected()

5. **Métodos privados claros:** Los métodos privados deben comenzar con guion bajo (_) y ser igual de descriptivos que los públicos:

   (a) **Ejemplo correcto:** _validateInput()

   (b) **Ejemplo incorrecto:** _doSomething()

6. **Consistencia en el estilo:**

   (a) Utilizar siempre **camelCase**.

   (b) Mantener coherencia en todo el proyecto.

   (c) Evitar mezclar idiomas.

```dart
// core/use/consumers/notes/container/note_consumer.dart
class NoteConsumerUse implements NoteConsumerRule {
  void create(String content) {
    // Crear una nueva nota
  }

  List<String> fetchAll() {
    // Obtener todas las notas
    return [];
  }

  void delete(String id) {
    // Eliminar nota por ID
  }
}

//   core/use/services/authentication/container/
   authentication_service.dart
class AuthenticationServiceUse implements
   AuthenticationServiceRule {
  Future<void> login(String username, String password) async
    {
    // Validar credenciales y generar token
  }

  Future<void> logout() async {
    // Invalidar ósesin
  }
}

// core/use/adapters/tasks/container/task_adapter.dart
class TaskAdapterUse {
  TaskDataRule fromJson(Map<String, dynamic> json) {
    // Convertir JSON a entidad del dominio
    return TaskDataRule(
      id: json['id'],
      title: json['title'],
    );
  }

  Map<String, dynamic> toJson(TaskDataRule task) {
    // Convertir entidad del dominio a JSON
```

```
40      return {
41        'id': task.id,
42        'title': task.title,
43      };
44    }
45  }
```

***Listing 28**. Forma correcta de nombrar los metodos*

Variables

Las variables son elementos fundamentales en cualquier proyecto. En la Arquitectura CMI, el uso correcto de nombres para variables garantiza un código **claro**, **legible** y fácil de mantener. Una variable bien nombrada evita confusiones y facilita la comprensión tanto del contexto como de la intención del código.

A continuación, se establecen las recomendaciones obligatorias para la nomenclatura de variables en proyectos basados en CMI.

Reglas Generales:

1. **Utilizar nombres descriptivos y específicos:** Toda variable debe reflejar claramente el propósito de su contenido. El nombre debe indicar **qué representa** o **qué almacena**, evitando ambigüedades.
   (a) **Ejemplo correcto:** noteList, isDarkModeEnabled, userEmail
   (b) **Ejemplo incorrecto:** data, value, temp
2. **Evitar nombres genéricos o innecesariamente cortos:** Solo en contextos muy limitados, como iteraciones simples, es aceptable usar nombres como i, j. En cualquier otro caso, los nombres deben ser significativos.
   (a) **Ejemplo correcto:** `for (var index = 0; index < notes.length; index++) { }`
   (b) **Ejemplo incorrecto:** `var x = notes.length;`
3. **Seguir el estilo camelCase:** Todas las variables deben definirse utilizando **camelCase**, sin excepciones.
   (a) **Ejemplo correcto:** selectedTaskId
   (b) **Ejemplo incorrecto:** Selected_task_id
4. **Prefijos para variables booleanas:** Las variables de tipo booleano deben comenzar con palabras que indiquen una condición o estado.
   (a) **Ejemplo correcto:** isActive, hasPermission, canEdit

(b) **Ejemplo incorrecto:** active, permission, editFlag

5. **Consistencia con el dominio del problema** Utiliza términos que pertenezcan al contexto de la aplicación. Esto facilita que el código sea más intuitivo para el equipo.

   (a) **Ejemplo correcto:** archivedNotesCount

   (b) **Ejemplo incorrecto:** counter (si se refiere al número de notas archivadas)

```
1 // core/use/services/authentication/container/
     authentication_service.dart
2 class AuthenticationServiceUse implements
     AuthenticationServiceRule {
3   // Token actual del usuario autenticado (mantenido en
     memoria)
4   String _userToken = '';
5
6   // Estado de óautenticacin actual
7   bool get isLoggedIn => _userToken.isNotEmpty;
8
9   Future<void> login(String username, String password) async
      {
10    // ...ólgica de óautenticacin
11    _userToken = 'tokenEjemplo123';
12  }
13
14  Future<void> logout() async {
15    _userToken = '';
16  }
17
18  String getToken() => _userToken;
19 }
20
21 // core/use/consumers/notes/container/note_consumer.dart
22 class NoteConsumerUse implements NoteConsumerRule {
23  final List<NoteDataRule> _noteCache = [];
24
25  void create(String content) {
26    final note = NoteDataRule(id: 'x1', content: content);
27    _noteCache.add(note);
28  }
29
```

```
30   List<NoteDataRule> fetchAll() => _noteCache;
31
32   void delete(String id) {
33     _noteCache.removeWhere((note) => note.id == id);
34   }
35
36   bool get hasNotes => _noteCache.isNotEmpty;
37 }
```

***Listing 29**. Forma correcta de nombrar las variables*

Buenas Prácticas en CMI

La **Arquitectura CMI** no solo proporciona una estructura clara y principios sólidos, sino que también promueve un conjunto de **buenas prácticas** que aseguran que el desarrollo sea limpio, sostenible y eficiente.

Adoptar estas prácticas desde el inicio de un proyecto garantiza que el código sea más fácil de mantener, escalar y comprender, especialmente en entornos colaborativos o en aplicaciones de larga duración.

Organización del Proyecto

- **Respeta la estructura de Capas, Módulos y Contenedores**: Mantén siempre la jerarquía definida por CMI. Evita crear carpetas o archivos fuera de su contexto correspondiente.
- **Evita la sobrecarga de contenedores**: Aunque es posible anidar contenedores, limita la profundidad para no complicar la navegación del proyecto.
- **Centraliza las configuraciones**: Todo lo relacionado con constantes, rutas, temas, idiomas, etc., debe estar en la capa configs. No dupliques configuraciones en otras estructuras.

Manejo de Dependencias

- **Aplica siempre Inyección de Dependencias (DI)**: Nunca instancies directamente servicios o consumidores dentro de los puntos de entrada del sistema (como handlers, comandos o listeners). Toda lógica debe atravesar los contratos definidos en rules/, permitiendo que la implementación real sea proporcionada por el sistema.
- **Usa abstracciones en lugar de implementaciones concretas**: Trabaja siempre con las interfaces declaradas en core/rules/ y evita acoplarte directamente a clases de use/. Esto permite intercambiar implementaciones sin romper la estructura general ni afectar capas superiores.

Nomenclatura y Legibilidad

- **Sigue estrictamente las convenciones de nomenclatura**: Mantén nombres claros, descriptivos y evita abreviaciones innecesarias. Un buen nombre reduce la necesidad de comentarios adicionales.

- **Documenta las clases y métodos complejos**: Aunque CMI promueve la claridad por estructura, siempre documenta aquellas partes que puedan ser confusas o críticas.

Reutilización y Componentización

- **Crea utilidades reutilizables en shared/:** Todo middleware, validador o herramienta que pueda ser usada en más de un módulo debe ubicarse en interface/shared/ para evitar duplicación de lógica o fragmentación del código.

- **No abuses de los componentes genéricos**: Si un recurso es muy específico de un módulo o contenedor de dominio, debe quedarse dentro del contenedor de dominio correspondiente.

Manejo de Errores

- **Implementa manejo de errores centralizado**: Define estrategias globales para capturar excepciones, especialmente en la capa core/use/services. La UI solo debe mostrar mensajes amigables, nunca manejar la lógica de errores directamente.

Testing

- **Escribe pruebas unitarias para rules y uses**: La abstracción de CMI facilita el testing. Asegúrate de cubrir la lógica de negocio y las integraciones.

Control de Versiones y Colaboración

- **Organiza ramas siguiendo la estructura de CMI**: Cuando trabajes en equipo, estructura las ramas Git por módulo o funcionalidad siguiendo la lógica de la arquitectura.

- **Utiliza Pull Requests con revisiones enfocadas en respetar la arquitectura**: Asegúrate que cada aporte siga las buenas prácticas y no rompa la coherencia estructural.

Optimización y Desempeño

- **Minimiza la sobrecarga inicial del servidor:** Organiza correctamente los handlers y listeners en módulos separados para facilitar su gestión e inicio controlado.
- Carga perezosa (Lazy Loading): Implementa estrategias para cargar adaptadores, orígenes o consumidores solo cuando sean necesarios, especialmente en servicios extensos o con múltiples integraciones.

Compatibilidad Tecnológica de la Arquitectura CMI

La Arquitectura CMI fue concebida inicialmente como una solución estructural para organizar proyectos complejos de forma clara, coherente y escalable.

Aunque su aplicación práctica comenzó en entornos donde se valoraba la modularidad, el tipado estricto y la mantenibilidad, CMI nunca estuvo atada a un lenguaje, tecnología o framework específico.

Debido a su enfoque en la separación jerárquica, el control de exposición y la reutilización estructurada, sus principios pueden adaptarse con facilidad a distintos entornos backend, siempre que se respete la organización en capas, módulos, contenedores y enrutadores.

Este capítulo presenta una guía de compatibilidad de CMI con distintos lenguajes de programación, frameworks y entornos backend que pueden adoptar su estructura sin perder flexibilidad ni escalabilidad.

Principios que permiten la adaptabilidad de CMI

CMI es adaptable a cualquier entorno que permita cumplir con sus principios fundamentales:

- **Modularidad:** División clara en capas, módulos y contenedores.
- **Separación de responsabilidades:** Cada elemento debe cumplir una función específica.
- **Bajo acoplamiento:** Los elementos deben estar conectados de manera mínima y controlada.
- **Alta cohesión:** Los elementos relacionados deben mantenerse agrupados lógicamente.
- **Control de accesos:** A través de mecanismos como enrutadores o puntos de exportación bien definidos.

Siempre que un lenguaje o framework permita aplicar estos principios, CMI puede ser implementada de forma efectiva.

Compatibilidad con Lenguajes de Programación

Table 17. *Compatibilidad con Lenguajes de Programación*

Lenguaje	**Compatibilidad**	**Notas**
Dart (Shelf)	Totalmente compatible	Base recomendada para estructura modular en CMI.
Go (Fiber, Echo)	Totalmente compatible	Gran afinidad con CMI por modularización por paquetes.
Node.js (Express)	Compatible	Requiere estructura disciplinada y manual de carpetas.
Deno (Fresh)	Compatible	Similar a Node.js con mejor separación por archivo.
Rust (Actix, Rocket)	Compatible	Permite organizar por capas y control de acceso.
Python (FastAPI)	Compatible	Estructura modular flexible y tipado claro.
Java (Spring Boot)	Compatible con ajustes	Requiere adaptar la estructura por anotaciones.

Compatibilidad con Frameworks y Librerías

***Table 18**. Compatibilidad con Frameworks y Librerías*

Framework / Librería	Compatibilidad	Notas
Shelf (Dart)	Totalmente compatible	Base recomendada para estructura modular en CMI.
Fiber (Go)	Totalmente compatible	Permite modularizar según paquetes, ideal para CMI.
Express (Node.js)	Compatible	Requiere estructura manual y disciplinada para mantener la jerarquía.
FastAPI (Python)	Compatible	Modularidad flexible con control explícito de rutas y contratos.
Rocket (Rust)	Compatible	Requiere separación clara de controladores y módulos.
Spring Boot (Java)	Compatible con ajustes	Necesita adaptación para cumplir la jerarquía CMI, especialmente en inyección.
Deno (Fresh)	Compatible	Estructura basada en archivos, pero admite organización modular controlada.

Notas sobre Aplicación Multiplataforma

CMI está diseñada para ser completamente multiplataforma.

Puede ser aplicada en:

- Servidores backend modulares
- APIs RESTful o GraphQL
- Microservicios organizados por dominio, responsabilidad y escalabilidad
- Plataformas cloud o serverless (Dart, Go, Node.js, Deno, etc.)

Siempre que se mantenga la organización y modularidad exigida, la arquitectura sigue funcionando de forma efectiva.

Conclusión

La **Arquitectura CMI** establece un marco sólido y normativo para el desarrollo de proyectos de software, especialmente en entornos donde la organización, escalabilidad y mantenibilidad son esenciales. A través de sus pilares —**capas**, **módulos**, **contenedores** y **enrutadores**—, CMI garantiza que cada proyecto crezca de forma ordenada, evitando los problemas comunes de arquitecturas flexibles o mal definidas.

Aplicar CMI no solo implica seguir una estructura, sino adoptar una filosofía de trabajo disciplinada que prioriza la claridad, el control y la preparación para el futuro.

Este documento ha presentado las bases teóricas, normativas y prácticas necesarias para implementar la arquitectura de manera efectiva.