Arquitectura CMI

Note

La arquitectura CMI es la interación por capas, modulos y contenedores y se desarrollo en base al framework de Flutter por consiguiente el contenido estará basado en su implementación en esta tecnología, no se descarta el uso en otras. Si puede aportar a que CMI funcione en más tecnologías sería de mucha ayuda.

Versión: 0.1.0

1. Introducción

CMI es una arquitectura modular y orientada a capas que tiene un enfoque de bajo acoplamiento entre sus diferentes capas, para ello CMi incorpora diferentes principios y patrones de diseño, que permiten proporcionar una forma organizada de estructurar el código de una aplicación de forma que sea mantenible, escalable y testeable.

2. Filisofia

2.1. Modularidad

La arquitectura CMI aplica La modularidad para dividir una capa general en modulos, cada una encargada de una parte específica de la funcionalidad general de la

capa padre. Esto implica que cada módulo tiene una tarea o conjunto de tareas claramente definidas y puede funcionar de manera independiente.

2.2. Separación de Responsabilidades

La arquitectura CMI aplica la separación de responsabilidades lo que implica que cada capa tenga una función o responsabilidad específica. Esto significa que la capa **configs** contiene todas las configuraciones y dependencias de acceso global , la capa **core** contiene las reglas de negocio y la entrada y salida de datos, por ultimo la capa de **ui** se encarga exclusivamente de la interfaz de usuario. Esta separación facilita la comprensión de la aplicación, ya que cada capa tiene un propósito claramente definido. Además, al aislar las responsabilidades, los cambios o mejoras en una capa se pueden realizar con un minimimo o nulo efecto en las demás capas, lo que reduce el riesgo de errores y facilita el mantenimiento, escalabilidad y la testeabilidad.

2.3. Escalabilidad

La arquitectura CMI tiene un enfoque escalable lo cual busca que la aplicación tenga la capacidad de manejar un aumento en la carga de trabajo de manera eficiente, ya sea en términos de volumen de datos, número de usuarios o complejidad de las distintas operaciones. Una aplicación escalable puede crecer y adaptarse a la demanda futura sin experimentar una degradación importante en el rendimiento.

2.4. Flexibilidad

La arquitectura CMI permite la flexibilidad, esta se refiere a la capacidad de la aplicación para adaptarse a cambios en los requisitos, o entornos sin necesidad de

una reescritura completa. Una aplicación flexible es la que puede evolucionar y expandirse con el mínimo impacto en su estructura existente. El objetivo que se busca es la flexibilidad la cual permite la incorporación de nuevas funcionalidades o modificaciones sin interrumpir el funcionamiento de la aplicación.

2.5. Desempeño

La arquitectura CMI busca mejorar el desempeño de la aplicación en términos de tiempo de respuesta, uso de recursos y capacidad para manejar la carga de trabajo. Una aplicación con buen desempeño es la que responde rápidamente a las solicitudes de los usuarios y utiliza los recursos disponibles de manera óptima.

2.6. Mantenibilidad

La arquitectura CMI permite la mantenibilidad de la aplicación lo cual permite modificar para corregir errores, mejorar su funcionalidad o adaptarse a nuevos requisitos. Una aplicación mantenible es la que puede ser mantenida, reparada y mejorada con un esfuerzo y costos mínimos.

2.7. Seguridad

La arquitectura CMI permite que la aplicación tenga la capacidad para protegerse contra accesos no autorizados, ataques y otras amenazas. Una aplicación segura es la que salvaguarda la integridad, confidencialidad y disponibilidad de los datos y servicios, asegurando que solo los usuarios autorizados puedan acceder y modificar la información.

2.8. Tolerancia a Fallos

La arquitectura CMI permite la tolerancia a fallos, la cual permite que la aplicación continue funcionando, total o parcialmente, en presencia de fallos o errores. Una aplicación tolerante a fallos está diseñado para manejar fallos de manera que minimice el impacto en los usuarios y mantenga la disponibilidad del servicio.

2.9. Reutilización

La arquitectura CMI oblica a aplicar la reutilización, la cual hace posible que las distintas partes puedan ser utilizados en múltiples zonas de la aplicación. La reutilización implica crear módulos que sean genérico, contenedores espeficos o genericos, flexibles para ser aplicados en diferentes partes del la aplicación o incluso en diferentes proyectos.

2.10. Portabilidad

La arquitectura CMI permite la portabilidad la cual es la capacidad de la aplicación para ser ejecutada en diferentes entornos o plataformas con un esfuerzo mínimo.

2.11. Documentación

La aplicaciones que utilicen la arquitectura CMI deben tener una documentación clara y completa que describa la arquitectura, diseño y funcionalidad de la aplicación.

2.12. Usabilidad

La arquitectura CMI obliga a que las aplicaciones sean totalmente usables y que los usuarios pueden interactuar con la aplicación y utilizar sus funcionalidades. Un aplicación usable proporciona una experiencia de usuario intuitiva, eficaz y agradable, permitiendo que los usuarios alcancen sus objetivos de manera eficiente.

2.13. Consistencia

La arquitectura CMI es estricto con la consistencia y se refiere a la coherencia en el comportamiento y la presentación de la aplicación a lo largo de todas sus partes. Una aplicación consistente mantiene un comportamiento predecible y homogéneo en todas las interacciones del usuario, lo que facilita la comprensión y su uso.

2.14. Aislación

La arquitectura CMI es estricta con la aislación, la cual es la capacidad de una aplicación para mantener las interacciones y cambios dentro de los límites de una capa, módulo o contenedor. Sin afectar a otros. Un aplicación bien aislada, encapsula la funcionalidad y minimiza las dependencias, lo que permite realizar cambios y pruebas sin riesgo de efectos colaterales.

2.15. Abstracción

La arquitectura CMI aplica la abstracción la cual es el principio que permite que las capas, modulos o contenedores superiores interactúen con niveles inferiores sin conocer sus detalles de implementación. Cada capa, modulo o contenedor proporciona un enrutador, ocultando la complejidad subyacente. Esta abstracción es esencial para crear una aplicación flexible y mantenible, ya que permite cambiar la

implementación de una capa, modulo o contenedor sin que las capas superiores se vean afectadas por completo o en parte. La abstracción también mejora la cohesión interna de las capas, ya que cada una puede centrarse en su responsabilidad específica sin depender directamente de las demás. La capa **core** aplica la abstracción en en su modulo** rule** la cual define la logica de negocios de la aplicación, los contenedores **sources** y **services** siempren son abstractas, y no deben contener codigo que no sea propio del lenguaje mismo.

3. Principios

3.1. Single Responsibility Principle (SRP)

La arquitectura CMI aplica el principio de responsabilidad única la cual establece que cada capa, módulo y contenedor en una aplicación debe tener una única responsabilidad o motivo para cambiar. Esto significa que cada parte debe hacer una sola cosa y hacerlo bien.

3.2. Open/Closed Principle (OCP)

La arquitectura CMI aplica el principio de abierto/cerrado, sugiere que las distintas partes deben estar abiertas para la extensión, pero cerradas para la modificación. Es decir se debe poder agregar una nueva funcionalidad a una parte existente sin cambiar todo su código original.

3.3. Liskov Substitution Principle (LSP)

La arquitectura CMI aplica el principio de sustitución de Liskov la 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.

3.4. 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. Al dividir las interfaces en grupos más pequeños, se garantiza que los usuarios solo tengan que conocer y usar las funcionalidades que realmente necesitan, reduciendo el acoplamiento y mejorando la claridad del código.

3.5. Dependency Inversion Principle (DIP)

La arquitectura CMI aplica el principio de inversión de dependencias lo que establece que las partes de alto nivel no deben depender de las partes 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 modulo **rule**, lo que permite que las dependencias sean invertidas y desacopladas, facilitando la inyección de dependencias y pruebas unitarias.

3.6. Principio de Separación de Intereses (Separation of Concerns)

La arquitectura CMI aplica el principio que implica dividir una aplicación en distintas capas, modulos y contenedores, donde cada una aborda una preocupación específica o una funcionalidad concreta. La separación de intereses mejora la modularidad de la aplicación, permitiendo que los desarrolladores trabajen en partes individuales de la aplicación 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 una sola capa, modulo o contenedor.

3.7. Principio DRY (Don’t Repeat Yourself)

La arquitectura CMI aplica el principio DRY busca evitar la duplicación de código o lógica. Cada parte de conocimiento o funcionalidad debe tener una representación única y sin duplicados en la aplicación. 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.

3.8. 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 o complejidades innecesarias que puedan complicar el desarrollo o dificultar el entendimiento.

3.9. 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.

3.10. 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 las diferentes capas, módulos o contenedores tienen pocas dependencias entre sí, lo que permite que se modifiquen o evolucionen de manera independiente, reduciendo el impacto de los cambios en la aplicación.

3.11. Principio de Inversión de Control (IoC)

La arquitectura CMI aplica el principio de inversión de control sugiere que el flujo de control en una aplicación debe ser manejado por un gestor de estados o inyección de dependencias, en lugar de la interfaz de usuario lo haga directamente. Esto significa que en lugar de que la interfaz de usuario de la aplicación invoque directamente a las dependencias, el control es entregado a al modulo Logic compartido o al contenedor Logic de cada modulo o contenedor de la capa UI que in-

yecta las dependencias necesarias en tiempo de ejecución. Esto promueve la flexibilidad y permite la creación de aplicaciones más modulares.

3.12. Modularidad

La arquitectura CMI aplica el principio de modularidad que divide una aplicación en capas, modulos y contenedores, independientes y autónomos, donde cada parte encapsula una funcionalidad específica y bien definidas. Esto facilita el desarrollo, la prueba, el mantenimiento y la reutilización de código.

4. Definición de la gestión de la estructura de la aplicación

4.1. Capa

Las “capas” son divisiones generales de una aplicación que separan responsabilidades y agrupan funcionalidades relacionadas. Cada capa tiene un propósito específico y se comunica con otras capas mediante su enrutador, lo que ayuda a organizar el código, mejorar la mantenibilidad y facilitar las pruebas.

4.2. Modulo

Los **módulos** encapsulan un conjunto específico de funcionalidades o responsabilidades. La modularidad se enfoca en dividir una capa en partes autónomas que pueden funcionar de manera independiente pero que interactúan a travez de su capa con el exterior.

4.3. Contenedor

Un contenedor es una entidad que agrupa y encapsula elementos de un modulo. Los contenedores facilitan la organización, reutilización, y gestión de los componentes, mejorando la mantenibilidad y escalabilidad de la aplicación. El ultimo nivel son los contenedores, pueden existir mas contenedores dentro de forma recursiva, estas se consideran sub.niveles pero es mejor evitarlos.

4.4. Enrutador

El enrutador es un archivo que tiene la funciona de gestionar las rutas de los diferentes capas, modulos o contenedores, solo se puede usar un enrutador por cada una y no esta permito usarlo dentro un contenedor generico ya que estos no lo requieren y usan el enrutador de su padre.

5. Gestión de la estructura de la aplicación

5.1. Configs

Es la capa de configuraciones en donde se establecen módulos como Router, theme, constans, inputs, enviroment, notifications, languages, Helpers, Colors y de más configuraciones, las mencionadas son solo un ejemplo, puede ser usada por toda las capas.

Ejemplo-Configs

```
lib/
│
├── main.dart
├── configs/
│   ├── assets
│   ├── env
│   ├── routes/
│   │   └── routes_config.dart
│   ├── types
│   └── configs.dart
├── core
└── ui
```

5.2. Core

Es la capa que agrupa los modulos **rules** y **uses**, son dos módulos importantes para el acceso a las fuentes de datos, las cueales podrian ser locales o remotas (internet).

5.2.1. Rules

Es un módulo abstracto que define las reglas de negocio más no las implementan. El módulo **rules** debe estar compuesto de puramente código propio del lenguaje es decir no debe depender de nada externo al lenguaje mismo, lo contenedores que la componen son:

- **Services**: Clases abstractas que define las reglas para interactuar con una fuente de datos (API, base de datos, etc.).
- **Consumers:** Clases abstracta que define la lógica de negocio de los datos obtenidos de un Service, y la cual permite cambiar entre distintos sources en cualquier momento.

- **data:** Define las reglas para el manejo de los datos del dominio, que son representaciones puras de los objetos que forman parte del negocio, esta es la unica capa que puede tener dependencias externas pero solo en casos excepcionales.
- **Enrutador:** Un enrutador (o router en inglés) es un archivo clave en la gestion estructural del módulo. Su principal función es gestionar el acceso a los niveles inferiores de su padre con el exterior.

Ejemplo-Core-Rules

```
lib/
|
├── main.dart
├── configs
├── core/
|   ├── rules/
|   |   ├── services/
|   |   |   ├── authentication/
|   |   |   |   └── authentication_service_rule.dart
|   |   |   ├── socket/
|   |   |   |   └── socket_service_rule.dart
|   |   |   └── sources_rule.dart
|   |   ├── data/
|   |   |   ├── user/
|   |   |   |   └── user_data_rule.dart
|   |   |   ├── message/
|   |   |   |   └── message_data_rule.dart
|   |   |   └── data_rule.dart
|   |   ├── consumers/
|   |   |   ├── authentication/
|   |   |   |   └── authentication_consumer_rule.dart
|   |   |   ├── socket/
|   |   |   |   └── socket_consumer_rule.dart
|   |   |   └── consumers_rule.dart
|   |   └── rules_core.dart
|   ├── use/
|   └── core.dart
```

```
└── ui
```

5.2.2. Uses

los uses siempre dependen y extienden del modulo rules y no se puede crear ninguna excepción a esta regla, lo contenedores que la componen son:

- **Services:** Implementación concreta de un Services Rule, encargada de realizar las operaciones con la fuente de datos, como llamadas HTTP o consultas a la base de datos.
- **Origins:** Representa los datos que vienen de las fuentes externas (como las APIs) o locales, para poder usarse deben para por un adaptador que las convierte en data.
- **Adapters:** Un adaptador que convierte los Origins en data rules y viceversa, un adaptador debe entregar un data totalmente funcional y sin errores.
- **Consumers:** Implementación de un Consumers Rule, que maneja la lógica de negocio aplicando las reglas de negocio sobre los datos obtenidos.
- **enrutador:** Un enrutador (o router en inglés) es un archivo clave en la gestion estructural del módulo. Su principal función es gestionar el acceso a sus contenedores con el exterior.

Ejemplo-Core-Uses

```
lib/
│
├── main.dart
├── configs
├── core
│   ├── rules/
```

```
│   ├── use/
│   │   ├── services/
│   │   │   ├── authentication_1/
│   │   │   │   └── authentication_service_use.dart
│   │   │   ├── authentication_2/
│   │   │   │   └── authentication_service_use.dart
│   │   │   ├── socket/
│   │   │   │   └── socket_service_use.dart
│   │   │   └── sources_use.dart
│   │   ├── origins/
│   │   │   ├── authentication/
│   │   │   │   └── authentication_origin_use.dart
│   │   │   ├── chat/
│   │   │   │   └── chat_origin_use.dart
│   │   │   └── origins_use.dart
│   │   ├── adapters/
│   │   │   ├── authentication/
│   │   │   │   └── authentication_adapter_use.dart
│   │   │   ├── chat/
│   │   │   │   └── chat_adapter_use.dart
│   │   │   └── adapters_use.dart
│   │   ├── consumers/
│   │   │   ├── authentication/
│   │   │   │   └── authentication_consumer_use.dart
│   │   │   ├── socket/
│   │   │   │   └── socket_consumer_use.dart
│   │   │   └── consumers_use.dart
│   │   └── uses_core.dart
│   └── core.dart
└── ui
```

5.3. UI (User Interface)

Es la capa con la que los usuarios interactuan por decirlo de otra forma es la capa visible para el usuario. En flutter se cuenta con un primer widget que normalmente es MaterialApp (app), este puede ser definido con una modulo con un archivo único el cual no requiere un enrutador.

- app: Primer widget que ejecuta flutter.
- **Layouts:** Es un modulo que contiene una variedad de layouts, y contiene varias views y pages.Las pages se pueden subdividr en views y las views se pueden subdividir una vez mas en PartViews. tambien dentro de un layout pueden exister otro tipo de contenedores como un contenedor que tenga todos los componentes exclusivos de ese layout o la logica de esta misma, como tambien las pages o views tambien pueden tenerlos.
- **Pages:** Es una parte de la interfaz de usuario que representa una pantalla completa o una vista de un layout. Cada Page generalmente contiene un conjunto de views que definen cómo se ve y cómo interactúa el usuario con esa pantalla.
- **Shared:** Es un módulo que contiene todas los componentes y demás que se usen de forma global en la UI.
- **Enrutador:** Un enrutador (o router en inglés) es un archivo clave en la gestion estructural del módulo. Su principal función es gestionar el acceso a sus contenedores con el exterior.

Ejemplo-UI

```
lib/
|
├── main.dart
├── configs
├── core
└── ui/
|   ├── app/
|   |   └── app.dart
|   ├── layouts/
|   |   ├── authentication/
|   |   |   ├── dialogs
|   |   |   ├── logic
```

```
|   |   |   ├── components/
|   |   |   |   ├── container/
|   |   |   |   |   ├── text_field_component_authentication.dart
|   |   |   |   |   ├── google_button_component_authentication.dart
|   |   |   |   |   └── components_authentication.dart
|   |   |   |   ├── views/
|   |   |   |   |   ├── login/
|   |   |   |   |   |   ├── logic
|   |   |   |   |   |   └── login_view_authentication.dart
|   |   |   |   |   ├── register/
|   |   |   |   |   |   ├── logic
|   |   |   |   |   |   └── register_view_authentication.dart
|   |   |   |   |   └── views_authentication.dart
|   |   |   |   └── authentication_layout.dart
|   |   |   ├── home/
|   |   |   |   ├── dialogs
|   |   |   |   ├── logic
|   |   |   |   ├── pages/
|   |   |   |   |   ├── chat/
|   |   |   |   |   |   ├── logic
|   |   |   |   |   |   └── chat_page_home.dart
|   |   |   |   |   └── pages_home.dart
|   |   |   |   ├── components/
|   |   |   |   |   ├── container/
|   |   |   |   |   |   ├── message_component_home.dart
|   |   |   |   |   |   ├── mini_user_component_home.dart
|   |   |   |   |   └── components_home.dart
|   |   |   |   ├── views/
|   |   |   |   |   ├── users/
|   |   |   |   |   |   ├── logic
|   |   |   |   |   |   └── users_view_home.dart
|   |   |   |   |   └── views_home.dart
|   |   |   |   └── home_layout.dart
|   |   |   └── layout_ui.dart
|   |   ├── pages/
|   |   |   ├── containers
|   |   |   └── pages_ui.dart
|   |   ├── shared/
|   |   |   ├── components/
|   |   |   |   ├── container
|   |   |   |   └── components_shared.dart
|   |   |   ├── dialogs/
```

```
|   |   |   ├── container
|   |   |   └── dialogs_shared.dart
|   |   ├── logic/
|   |   |   ├── container
|   |   |   └── logic_shared.dart
|   |   └── shared_ui.dart
|   └── ui.dart
```

6. Convenciones de Nomenclatura

La arquitectura CMI aplica estrictamente el uso correcto de convenciones de nomenclatura. Son reglas fundamentales para tener un código correcto, en este caso nos permiten dar un correcto nombre a los archivos, variables, funciones, clases, y una serie de elementos.

El uso de las convenciones de nomenclatura no solo es un objetivo sin razón, se convierte en una base importante el cual nos permite tener una mayor legibilidad y simplicidad en el mantenimiento.

La forma de realizar la nomenclatura es independiente del tipo de nomenclatura que use, debe de cumplirse las pautas mencionadas aqui.

la forma en que se nombran todo en CMI, el orden va de acuerdo al nivel en el que se encuentre, es de decir de menor a mayor nivel. Evite tener nombres demasiado largos e innecesarios.

6.1. Archivos

Son ejemplos de como se implementan los nombres de los archivos

Forma correcta de nombrar los archivos

```
namefile_padre(...).dart
message_component_home.dart (correcto)// Es un archivo componente
que solo pertenece al layout home
message_component_components_home.dart (incorrecto) // Es un
archivo componente que solo pertenece al layout home pero
contiene el nombre de ¨components¨ esta por demas, porque ya se
sobre entiende al poner ¨component¨
message_component.dart (incorrecto) // Es un archivo componente
perdido.
message.dart (incorrecto) // Es un archivo desconocido.
```

Enrutador: En caso de los archivos de tipo enrutador su nombre es el mismo que el de su padre.

Forma correcta de nombrar los archivos enrutador

```
configs/
  configs.dart (correcto) // Esta es la forma correcta de nombra
a los enrutadores
...views/
  views_authentication.dart (correcto) // Esta es la forma
correcta de nombra a los enrutadores, pero en esta caso en un
enrutador que pertenece a las views de authentication que es un
layout
```

6.2. Clases

- Ser descriptivo y conciso: Utiliza nombres descriptivos que reflejen claramente la responsabilidad y la función de la clase. Evita nombres demasiado genéricos o ambiguos que no proporcionen una indicación

clara de lo que hace la clase. Mantén los nombres de las clases lo más cortos posible sin sacrificar la claridad.

- Utilizar sustantivos que representen entidades: El nombre de la clase debe representar una entidad o un concepto en el dominio del problema que estás abordando. Utiliza sustantivos que sean significativos y relevantes para el contexto de la aplicación.
- Evitar abreviaturas y acrónimos oscuros: Aunque a veces es tentador abreviar o acortar nombres de clases largos, trata de evitarlo en la medida de lo posible. Opta por la claridad y la legibilidad en lugar de la concisión extrema. Si es necesario usar una abreviatura, asegúrate de que sea ampliamente reconocida y entendida en el contexto del problema.

Forma correcta de nombrar las clases

```
class ElevatedButtonComponentAuthentication extends
StatelessWidget {} // Este es un componente que pertenece
solamente al layout de authentication.
class CustomInputComponentShared extends StatelessWidget {} //
Este es un componente de uso gloabal por que pertenece al
contenedor components que es parte de los shared
```

6.3. Funciones

- Ser descriptivo y conciso: Utiliza nombres descriptivos que reflejen claramente el propósito y la funcionalidad de la función. Evita nombres genéricos o ambiguos que no indiquen claramente lo que hace la función. Mantén los nombres de las funciones lo más cortos posible sin sacrificar la claridad.

- Utilizar verbos para denotar acciones: Usa verbos descriptivos para indicar qué hace la función. Esto ayuda a que el nombre de la función sea más expresivo y a que los desarrolladores puedan entender rápidamente qué hace la función sin necesidad de revisar su implementación.
- Seguir un estilo coherente: Mantén la coherencia en la nomenclatura de las funciones en todo tu código.
- Ser consistente con el dominio del problema: Utiliza términos y conceptos del dominio del problema en los nombres de las funciones siempre que sea posible. Esto ayuda a que el código refleje más fielmente el dominio del problema y facilita la comunicación entre los miembros del equipo y los interesados.

Forma correcta de nombrar las funciones

```
// Esta es una función la forma de nombrarlas es la misma que el
de las clase
GoRouter routerConfig(WidgetRef ref) => GoRouter(
      initialLocation: LoadingPageShared.link,
      routes: _routes,
    );
```

6.4. Metodos

- Usar verbos para denotar acciones: Los nombres de los métodos deben reflejar la acción que realizan. Usa verbos descriptivos para indicar qué hace el método.
- Ser específico y conciso: Intenta hacer que el nombre del método sea lo más específico y conciso posible, describiendo claramente su propósito

y función sin necesidad de comentarios adicionales. Evita nombres demasiado largos o ambiguos.

- Utilizar nombres autoexplicativos: Los métodos deben ser autoexplicativos y no requerir comentarios adicionales para entender su propósito.
- Seguir un estilo coherente: Mantén la coherencia en la nomenclatura de los métodos en todo tu código.

Forma correcta de nombrar los métodos

```
import 'package:intl/intl.dart';
class FormatterHelperConfig {
  /// [ymd] changes the format of the datetime to ymd format.
  static DateFormat ymd = DateFormat('yyyy/MM/dd');

  /// [ym] changes the format of the datetime to ym format.
  static DateFormat ym = DateFormat('yyyy/MM');

  /// [kma] changes the format of the datetime to kma format.
  static DateFormat kma = DateFormat('kk:mm\ta');

  ///[completTime] is a function that corrects a time.
  static String completTime({
    required int hour,
    required int minute,
    String divisor = ':',
    String spacer = ' ',
  }) {
    if (hour < 10 && minute < 10) {
      return '0$hour$spacer{divisor}${spacer}0$minute';
    } else if (hour < 10 && minute > 9) {
      return '0$hour$spacer$divisor$spacer$minute';
    } else if (hour > 9 && minute < 10) {
      return '$hour$spacer$divisor${spacer}0$minute';
    } else {
      return '$hour$spacer$divisor$spacer$minute';
    }
  }
```

```
  static String number(double number,
      {int decimals = 0, required String locale}) {
    final formatterNumber = NumberFormat.compactCurrency(
            decimalDigits: decimals, symbol: '', locale: locale)
        .format(number);
    return formatterNumber;
  }
}
```

Forma incorrecta de nombrar los métodos

```
import 'package:intl/intl.dart';
class FormatterHelperConfig {
  /// [ymd] changes the format of the datetime to ymd format.
  static DateFormat ymd = DateFormat('yyyy/MM/dd');

  /// [ym] changes the format of the datetime to ym format.
  static DateFormat ym = DateFormat('yyyy/MM');

  /// [kma] changes the format of the datetime to kma format.
  static DateFormat kma = DateFormat('kk:mm\ta');

  ///[completTime] is a function that corrects a time.
  static String FormatterCompletTime({
    required int hour,
    required int minute,
    String divisor = ':',
    String spacer = ' ',
  }) {
    if (hour < 10 && minute < 10) {
      return '0$hour$spacer{divisor}${spacer}0$minute';
    } else if (hour < 10 && minute > 9) {
      return '0$hour$spacer$divisor$spacer$minute';
    } else if (hour > 9 && minute < 10) {
      return '$hour$spacer$divisor${spacer}0$minute';
    } else {
      return '$hour$spacer$divisor$spacer$minute';
    }
```

```
  }

  static String FormatterHelperNumber(double number,
      {int decimals = 0, required String locale}) {
    final formatterNumber = NumberFormat.compactCurrency(
            decimalDigits: decimals, symbol: '', locale: locale)
        .format(number);
    return formatterNumber;
  }
}
```

6.5. Variables

Siga todas estas recomendaciones para el correcto manejo de la nomenclatura de las variables.

- Usar nombres descriptivos: Elige nombres que sean descriptivos y claros sobre el propósito y el contenido de la variable. Esto facilita la comprensión del código para ti y para otros desarrolladores que puedan leerlo en el futuro.
- Evitar nombres genéricos: Evita nombres genéricos como `x`, `y`, `temp`, etc. Utiliza nombres que reflejen el propósito específico de la variable en el contexto en el que se utiliza.
- Utilizar nombres autoexplicativos: Los nombres de variables deben ser autoexplicativos y no requerir comentarios adicionales para entender su significado. Esto ayuda a mejorar la legibilidad del código.

Forma correcta de nombrar las variables

```
import 'package:intl/intl.dart';
class FormatterHelperConfig {
```

```
  /// [ymd] changes the format of the datetime to ymd format.
  static DateFormat ymd = DateFormat('yyyy/MM/dd');

  /// [ym] changes the format of the datetime to ym format.
  static DateFormat ym = DateFormat('yyyy/MM');

  /// [kma] changes the format of the datetime to kma format.
  static DateFormat kma = DateFormat('kk:mm\ta');

  ///[completTime] is a function that corrects a time.
  ...
}
```

Forma incorrecta de nombrar las variables

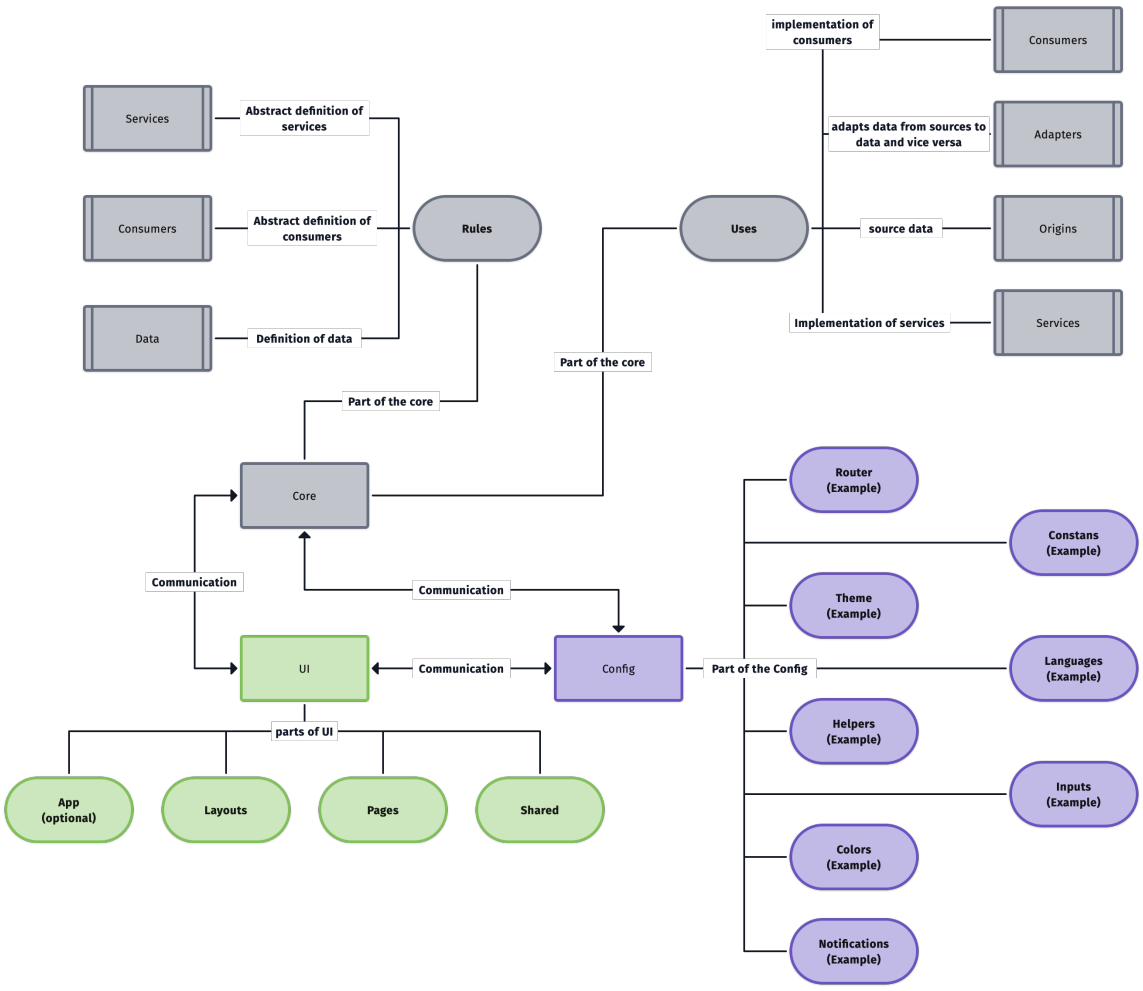
```
import 'package:intl/intl.dart';
class FormatterHelperConfig {
  /// [ymd] changes the format of the datetime to ymd format.
  static DateFormat FormatterYmd = DateFormat('yyyy/MM/dd');

  /// [ym] changes the format of the datetime to ym format.
  static DateFormat FormatterHelperYm = DateFormat('yyyy/MM');

  /// [kma] changes the format of the datetime to kma format.
  static DateFormat FormatterHelperConfigKma =
DateFormat('kk:mm\ta');

  ///[completTime] is a function that corrects a time.
  ...
}
```

Anexos



Licencia

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.