




**ALCALDÍA MAYOR  
DE BOGOTÁ D.C.**  
CULTURA RECREACIÓN Y DEPORTE  
Instituto Distrital de las Artes


# GUIA ARQUITECTURA HEXAGONAL



	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 2 de 17


HISTORICO DE CAMBIOS		
Versión	Fecha	Cambios Realizados
01	22/07/2021	Emisión Inicial

<b>Elaboró:</b>  <b>Hernán Mauricio Rincón Bedoya</b> Contratista Tecnologías de la Información  <b>Cristian Camilo Calderón Tapia</b> Contratista Tecnologías de la Información  <b>Steven Hernández Ríos</b> Contratista Tecnologías de la Información  <b>Laura Viviana Cruz Martínez</b> Contratista Tecnologías de la Información	<b>Revisó:</b>    <b>Camila Crespo Murillo</b> Contratista Oficina Asesora de Planeación y Tecnologías de la Información	<b>Aprobó:</b>    <b>Carlos Alfonso Gaitán Sánchez</b> Jefe de la Oficina Asesora de Planeación y Tecnologías de la Información	<b>Avaló:</b>    <b>Carlos Alfonso Gaitán Sánchez</b> Jefe de la Oficina Asesora de Planeación y Tecnologías de la Información
--	---	--	---

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 3 de 17

## Contenido

1. OBJETIVO	4
2. ALCANCE	4
3. GLOSARIO	4
4. APLICACIÓN DE LA ARQUITECTURA HEXAGONAL	4
4.1 DEFICIONES	4
4.2 BENEFICIOS DE USAR ESTA ARQUITECTURA	6
4.3 APLICANDO LA ARQUITECTURA	8
4.3.1 CAPA DE DOMINIO	8
4.3.2 CAPA DE APLICACIÓN	13
4.3.2 CAPA DE INFRAESTRUCTURA	16

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 4 de 17

## 1. OBJETIVO

Definir una arquitectura estándar para los desarrollos de IDARTES, que permita un completo desacople del lenguaje y frameworks utilizados

## 2. ALCANCE

Esta guía permite definir la estructura básica de la arquitectura hexagonal, así como las reglas comunes que se deben utilizar en los desarrollos


## 3. GLOSARIO

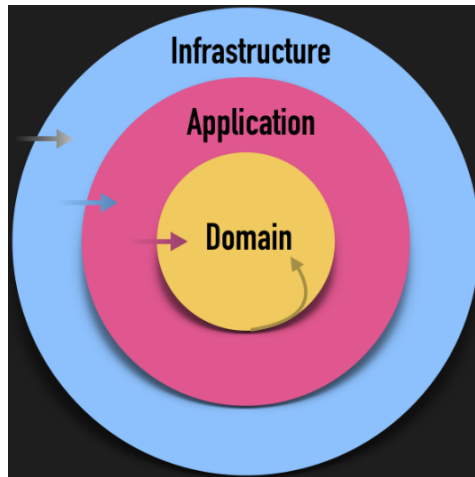
- Dominio: Capa de arquitectura hexagonal más profunda y encargada de la lógica del negocio.
- Aplicación: Capa intermedia de arquitectura hexagonal que contiene los casos de uso.
- Infraestructura: La capa más externa de la arquitectura hexagonal, contiene las implementaciones.
- Int: tipo de dato para almacenar un valor numerico.
- String: tipo de dato para almacenar un valor caracter.
- Domain-Driven Design: provee una estructura de prácticas y terminologías para tomar decisiones de diseño que enfoquen y aceleren el manejo de dominios complejos en los proyectos de software.
- Ubiquitous Language: Es el modelamiento de los objetos de dominios de la aplicación de forma clara.

## 4. APLICACIÓN DE LA ARQUITECTURA HEXAGONAL

### 4.1 DEFICIONES

La arquitectura hexagonal es una arquitectura del software en la que se busca separar el core lógico de la aplicación, dejarlo en el centro totalmente aislado del exterior, del cliente y de otras interacciones.

 <p>ALCALDÍA MAYOR DE BOGOTÁ D.C. CULTURA, RECREACIÓN Y DEPORTE Instituto Distrital de las Artes</p>	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 5 de 17



Esta arquitectura está compuesta de 3 capas bien definidas para el desarrollo: Dominio, Aplicación e Infraestructura. A continuación, se definen las 3 capas:

### **Dominio**


En la capa de dominio se definen Conceptos que están en nuestro contexto como por ejemplo (Usuario, Producto, Carrito, etc), y reglas de negocio que vienen determinadas en exclusiva por nosotros (servicios de dominio).

### **Aplicación**

La capa de aplicación es donde viven los casos de uso de nuestra aplicación (registrar usuario, publicar producto, añadir producto al carrito, etc).

### **Infraestructura**

Código que cambia en función de decisiones externas. En esta capa vivirán las implementaciones de las interfaces que definiremos a nivel de dominio. Es decir, nos apoyaremos en el DIP de SOLID para poder desacoplarnos de las dependencias externas.

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 6 de 17</b>

## Puertos y adaptadores

Esta arquitectura también se denomina “Ports and adapters”. De hecho, este término es más acertado ya que tiene connotaciones más cercanas a lo que podríamos pensar al ver cómo queda el código, y no implica un número finito como sí lo hace la palabra hexágono.

Los puertos son las interfaces definidas en la capa de dominio para desacoplarnos de nuestra infraestructura. Ejemplo: UserRepository, ProductoRepository, CarritoRepository.

Los adaptadores (que se definen a nivel de infraestructura) son las implementaciones posibles de esos puertos. Estas implementaciones traducirán esos contratos definidos en la interfaz a la lógica necesaria a ejecutar en base a un determinado proveedor. Ejemplo: OracleUserRepository, MysqlRepository, TestRepository.

## Testeo de código

Otra de las cualidades que presenta esta arquitectura es la facilidad de realizar test, ya que al estar totalmente desacoplada no es necesario realizar pruebas de una funcionalidad recorriendo todo el proceso, si no que solo se probará la función o caso de uso necesario.

En este contexto y siguiendo esta arquitectura se podrá realizar los siguientes tipos de test:


Test unitarios: Capa de Aplicación y Dominio

Test de integración: Capa de Infraestructura

Test de Aceptación: Todas las capas (funcionalidad completa, realizada por un tester o usuario final).

## 4.2 BENEFICIOS DE USAR ESTA ARQUITECTURA

### Independiente del mundo exterior

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 7 de 17

La aplicación puede ser dirigida por cualquier número de controles distintos. Por ejemplo, la lógica de negocio interior puede ser usada a través de una interfaz “Command Line”, otra aplicación o sistema, un humano o un guion automático.

### **Independiente a servicios externos**

Cuando la aplicación es independiente al mundo exterior. Se puede desarrollar el dominio de la aplicación mucho antes de empezar a pensar sobre el tipo de base de datos que se va a utilizar. Definiendo puertos y adaptadores para la base de datos, se está libre de utilizar cualquier tecnología de implementación. Lo que permite utilizar un almacenamiento de información en la memoria en los primeros días, y más tarde tomar la decisión del tipo de base de datos se quiere utilizar cuando se necesite almacenar la información de la aplicación en un almacenamiento persistentemente.

### **Más fácil de testear de manera individual**


Ahora que la aplicación es independiente al mundo exterior, es mucho más fácil de testear de forma individual. Esto significa que, en vez de enviar peticiones en HTTP, o hacia base de datos, simplemente se puede testear cada capa de la aplicación, burlando cualquier dependencia. Esto solo funciona en test rápidos, pero desacopla de manera masiva el código de los detalles de implementación del mundo exterior.

### **Los puertos y adaptadores son reemplazables**

El rol de los puertos y adaptadores es convertir las peticiones y las respuestas a medida que entran del mundo exterior. Este proceso de reconversión permite a la aplicación recibir peticiones y enviar respuestas a cualquier numero de tecnología exterior sin tener que saber nada de ellas. Permite reemplazar un adaptador por una implementación diferente que forme la misma interfaz.

### **Separación de las diferentes tasas de cambio**

Las capas más exteriores que típicamente tienen más cambios. Por ejemplo, la interfaz de usuario, gestionando peticiones o trabajando con servicios externos que normalmente evoluciona más rápido que las reglas de negocio de la aplicación. Esta separación le permite rápidamente iterar en

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 8 de 17</b>

las capas más externas sin tocar las capas más interiores que se han de mantener consistentes. La capa más interna no tiene conocimiento de las más exteriores y estos cambios pueden darse sin dañar el código.

### **Alto mantenimiento**

El mantenimiento es la ausencia de deuda técnica. Cambios en un área de la aplicación no afectan a otras. Añadir características no requiere un largo cambio de bases de código. Añadiendo nuevas maneras de interactuar con la aplicación requiere pocos cambios. El testeo es relativamente fácil.

## **4.3 APLICANDO LA ARQUITECTURA**

### **4.3.1 CAPA DE DOMINIO**

Esta capa contendrá:

- Objeto de Dominio
- Value Objects
- Excepciones de Dominio
- Repositorio


Razones para usar Value Objects

Como una buena práctica se debe construir un objeto de dominio que esté compuesto por Value Objects.

Al revisar este listado de atributos de un dominio

- IDs
- Email
- Nombre
- Contraseña
- Edad



	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 9 de 17

- Nivel de permisos
- Cantidad
- Precio

Los posibles valores que tendrán son un subconjunto dentro de alguno de los tipos primitivos. Es decir, "Cantidad" por ejemplo tendrá la restricción de que, debe ser tipo Int, sólo puede adquirir valores mayores o igual a 1. En el caso del Email, será un String, pero deberá cumplir unas ciertas reglas para considerarse un email válido.

Al modelar en el ámbito de dominio, habrá cierta lógica al respecto relacionada con estos conceptos. Por cohesión, desde el momento en el que tenemos una clase específica para modelar estos conceptos, se convierte en el lugar perfecto para albergar estos métodos. Por ejemplo, en Edad podríamos modelar el método para determinar si es mayor de edad, en Email podríamos tener un método para extraer el proveedor de email, etc.

Aporta semántica de dominio al código. No es lo mismo tener una firma tal que `User::register(string, string, int, int)`, que una tal que `User::register(UserId, UserEmail, Age, AccessLevel)`. Esto es algo crucial en términos de Domain-Driven Design (DDD) derivado del concepto de Ubiquitous Language.

Por definición, una instancia de un ValueObject será inmutable. Con lo cual, nos previene de posibles bugs por temas de cambio de estado. Algo más crítico aún si estamos en un sistema con una alta concurrencia.

#### Excepciones de Dominio


Las excepciones de dominio serán lanzadas en el momento de asignar un valor a un Value Object y que este valor no cumpla las validaciones del objeto.

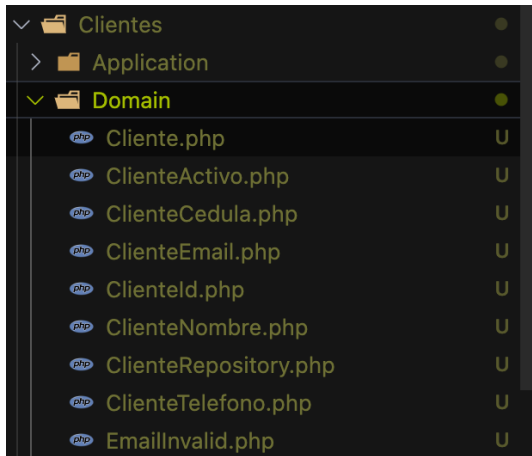
#### Repositorio

Es una interfaz donde se declaran las acciones que se podrán realizar con los objetos de dominio.

#### Ejemplo:

Tenemos una aplicación para una firma de abogados, en esta aplicación se identifica uno de los objetos de dominio que es Cliente.

 ALCALDÍA MAYOR DE BOGOTÁ D.C. CULTURA, RECREACIÓN Y DEPORTE Instituto Distrital de las Artes	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS  COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 10 de 17</b>



En la imagen se identifica la entidad Cliente, que tiene unos atributos, Id, Nombre, Cedula, Email, Telefono, Activo.

Se modela cada uno de estos atributos como una clase Value Object

```


final class ClienteEmail{
    private $value;

    public function __construct(string $value){
        $this->setEmail($value);
    }

    private function setEmail(string $value){
        if (!filter_var($value, FILTER_VALIDATE_EMAIL)) {
            throw new EmailInvalid();
        }
        $this->value = $value;
    }

    public function value(): string{
        return $this->value;
    }
}

```

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 11 de 17</b>

Al encapsular la lógica en un objeto, centralizamos la validación de este objeto y no dejamos código regado por toda la aplicación.

```


use src\proffense\Cientes\Domain\ClienteId;
use src\proffense\Cientes\Domain\ClienteCedula;
use src\proffense\Cientes\Domain\ClienteNombre;
use src\proffense\Cientes\Domain\ClienteEmail;
use src\proffense\Cientes\Domain\ClienteTelefono;
use src\proffense\Cientes\Domain\ClienteActivo;

final class Cliente{
    private $id;
    private $cedula;
    private $name;
    private $email;
    private $phone;
    private $activo;

    public function __construct(
        ClienteId $id,
        ClienteCedula $cedula,
        ClienteNombre $name,
        ClienteEmail $mail,
        ClienteTelefono $phone,
        ClienteActivo $activo
    ){
        $this->id = $id;
        $this->cedula = $cedula;
        $this->name = $name;
        $this->email = $mail;
        $this->phone = $phone;
        $this->activo = $activo;
    }

    public function id():ClienteId{
        return $this->id;
    }
}

```

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 12 de 17

```

public function cedula():ClienteCedula{
    return $this->cedula;
}

public function name():ClienteNombre{
    return $this->name;
}

public function email():ClienteEmail{
    return $this->email;
}

public function phone():ClienteTelefono{
    return $this->phone;
}

public function activo():ClienteActivo{
    return $this->activo;
}

public static function create(ClienteId $id,ClienteCedula $cedula,ClienteNombre
$name,ClienteEmail $mail,ClienteTelefono $phone,ClienteActivo $activo): self{
    $cliente =new self($id,$cedula,$name,$mail,$phone,$activo);
    return $cliente;
}
}

```

El modelo cliente ahora está compuesto por una serie de Value Objects.

El repositorio como se mencionó es una interfaz con las posibles acciones que se podrán realizar sobre el objeto de dominio


```

interface ClienteInterface{

    public function find(int $id): ?Cliente;

    public function findByCriteria():?Cliente;
}

```

 <p>ALCALDÍA MAYOR DE BOGOTÁ D.C. CULTURA, RECREACIÓN Y DEPORTE Instituto Distrital de las Artes</p>	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 13 de 17

```
public function save(Cliente $cliente): void;

public function delete(int $clientId): void;

public function update(Cliente $cliente): void;
}
```

### 4.3.2 CAPA DE APLICACIÓN

En esta capa están todos los casos de uso que puede tener el objeto de dominio. Para el ejemplo existirán 5 casos de uso inicialmente:


- Find
- FindByCriteria
- Save
- Delete
- Update

Para implementar cada uno de los casos se debe crear una clase que en el constructor recibirá un objeto del tipo repositorio el cual fue definido en la capa de Dominio, a esto le llamamos una inyección de dependencia.

Por ejemplo para el caso usó Save, con el cual se creará un nuevo cliente, el código sería el siguiente:

```
use src\proffense\Cientes\Domain\Cliente;
use src\proffense\Cientes\Domain\ClienteActivo;
use src\proffense\Cientes\Domain\ClienteCedula;
use src\proffense\Cientes\Domain\ClienteEmail;
use src\proffense\Cientes\Domain\ClienteId;
use src\proffense\Cientes\Domain\ClienteNombre;
use src\proffense\Cientes\Domain\ClienteInterface;
use src\proffense\Cientes\Domain\ClienteTelefono;

final class CreateClienteUseCase{
```

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 14 de 17

```
private $repository;

public function __construct(ClienteInterface $clienteRepository){
    $this->repository = $clienteRepository;
}

public function __invoke(ClienteId $id,
    ClienteCedula $cedula,
    ClienteNombre $name,
    ClienteEmail $mail,
    ClienteTelefono $phone,
    ClienteActivo $activo)
{
    $cliente = Cliente::create($id,$cedula,$name,$mail,$phone,$activo);
    $this->repository->save($cliente);
}
}
```

En el constructor se inyecta la dependencia, y se cuenta con el método \_\_invoke para que se ejecute la acción.

Y de esta forma con cada uno de los casos de usos que se presenten.


```
use src\proffense\Cientes\Domain\ClienteId;
use src\proffense\Cientes\Domain\ClienteRepository;

final class DeleteClienteUseCase{

    private $repository;

    public function __construct(ClienteRepository $repository){
        $this->repository = $repository;
    }

    public function __invoke(ClienteId $Clienteid){
        $this->repository->delete($Clienteid);
    }
}
```

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 15 de 17</b>

```

use src\proffense\Cientes\Domain\Cliente;
use src\proffense\Cientes\Domain\Clienteld;
use src\proffense\Cientes\Domain\ClienteRepository;

final class GetClienteByIDUseCase{

    private $repository;

    public function __construct(ClienteRepository $repository){
        $this->repository = $repository;
    }

    public function __invoke(Clienteld $clienteID):cliente{
        $cliente = $this->repository->find($clienteID);
        return $cliente;
    }
}

```

```

use src\proffense\Cientes\Domain\Cliente;
use src\proffense\Cientes\Domain\ClienteRepository;


final class UpdateClienteUseCase{

    private $repository;

    public function __construct(ClienteRepository $repository){
        $this->repository = $repository;
    }

    public function __invoke(Cliente $cliente){
        $this->repository->update($cliente);
    }
}

```

	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	<b>Código: GTIC-G-04</b>
		<b>Fecha: 22/07/2021</b>
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	<b>Versión: 01</b>
		<b>Página 16 de 17</b>

#### 4.3.2 CAPA DE INFRAESTRUCTURA

En esta capa se creará una implementación concreta de la aplicación, es decir el requerimiento es que se pueda persistir los datos del cliente en una base de datos, o en un archivo plano, es en esta capa donde se deberá realizar la adaptación.

Por ejemplo, para nuestro caso se utilizará Eloquent como ORM.

La implementación quedaría así:

```

use App\Client;
use src\proffense\Cientes\Domain\Clienteld;
use src\proffense\Cientes\Domain\Cliente;
use src\proffense\Cientes\Domain\ClienteInterface;

final class EloquentClienteRepository implements ClienteInterface{

    private $Cliente;


    public function find(Clienteld $id): ?Cliente
    {
        $this->Cliente = Client::where("id","=", $id)->get();
        return $this->Cliente;
    }

    public function findByCriteria(): ?Cliente
    {
        return null;
    }

    public function save(Cliente $cliente): void
    {
        Client::create([
            'id' => $cliente->id,
            'nombre' => $cliente->name,
            'cedula' => $cliente->cedula,
            'email' => $cliente->email,

```



	<b>GESTIÓN DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES - TIC</b>	Código: GTIC-G-04
		Fecha: 22/07/2021
	<b>GUIA ARQUITECTURA HEXAGONAL</b>	Versión: 01
		Página 17 de 17

```

        'telefono' => $cliente->telefono,
        'activo' => $cliente->activo,
    ]
);
}

public function delete(ClienteId $clienteId):void
{
    $this->Cliente = Client::where("id","=", $clienteId)->get();
    $this->Cliente->delete();
}

public function update(Cliente $cliente): void
{
    $this->Cliente = Client::where("id","=", $cliente->id)->get();
    $this->Cliente->name = $cliente->name;
    $this->Cliente->cedula = $cliente->cedula;
    $this->Cliente->email = $cliente->email;
    $this->Cliente->telefono = $cliente->telefono;
    $this->Cliente->activo = $cliente->activo;
    $this->Cliente->update();
}
}

```

Como se puede observar, la infraestructura se implementa la interfaz del dominio, y es en esta capa donde usamos las herramientas propias del framework en este caso el ORM Eloquent; que por el ejemplo llamamos Client.