Arquitectura Hexagonal

Ejemplo práctico

Luis Bertel

Tabla de contenido

Introducción

Introducción

La arquitectura hexagonal está basada en los principios de Arquitecturas Limpias. Éstas arquitecturas comparten un conjunto de buenas prácticas como lo son:

- Intependencia de los Frameworks.
- Testeable.
- Independencia de la UI.
- Independencia de la persistencia de datos.
- Independencia de entidades externas.

Capas de la arquitectura limpias

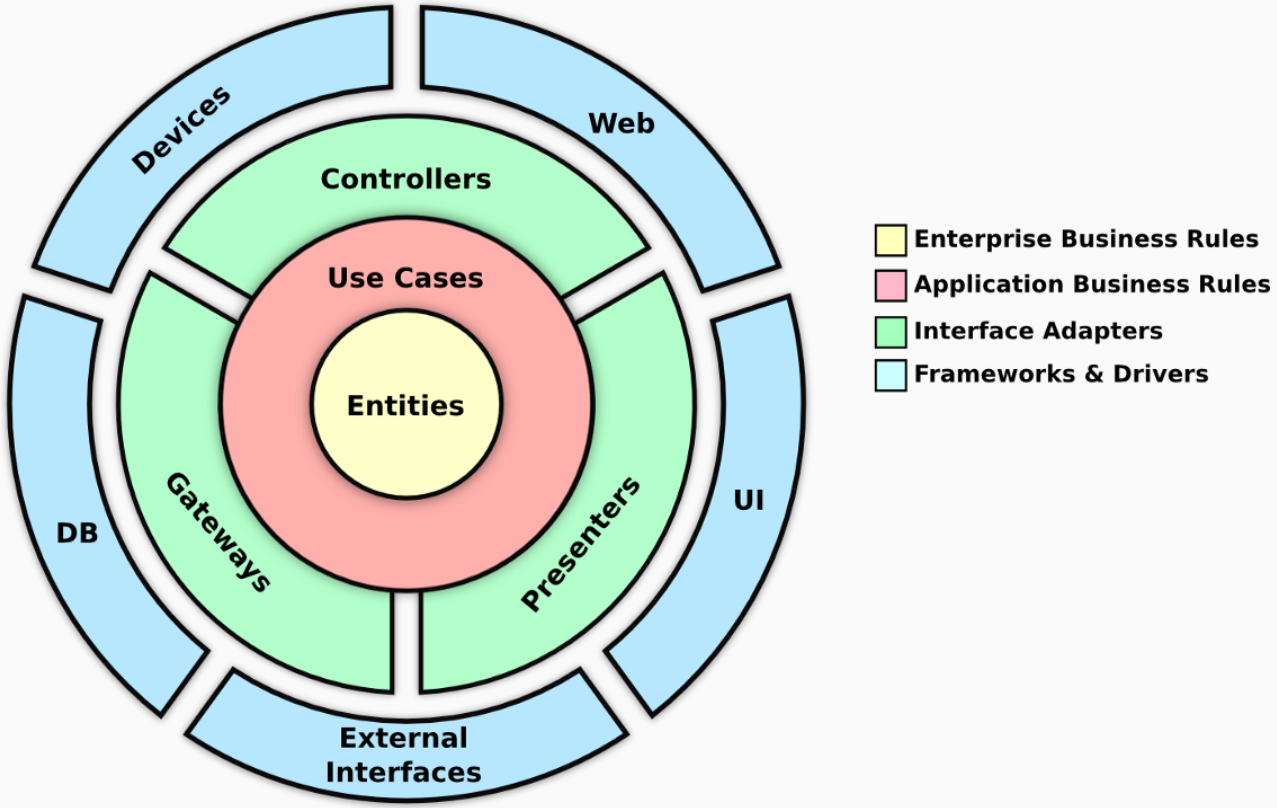


Figura 1: Capas de arquitectura limpia

Capas de la arquitectura hexagonal

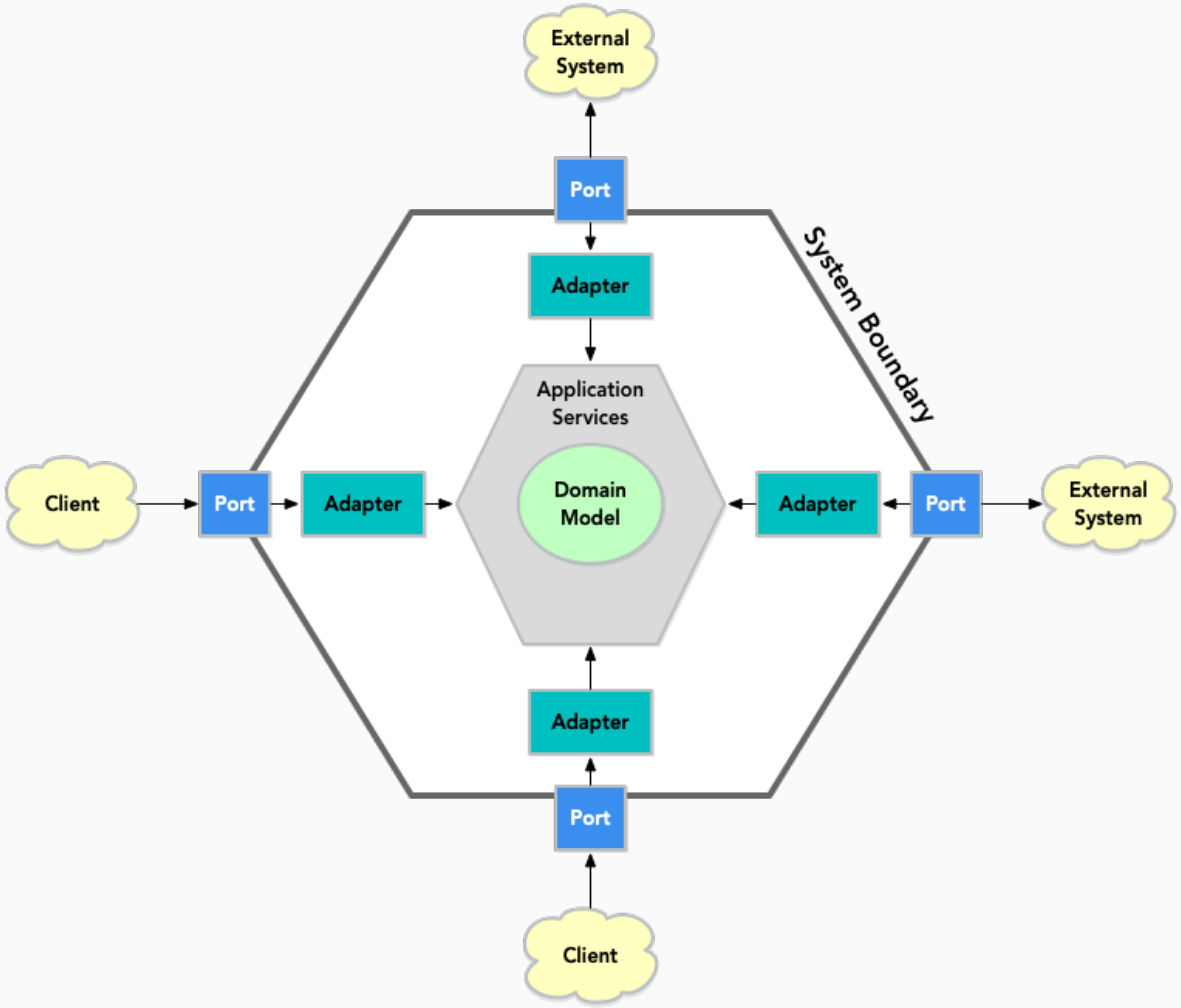
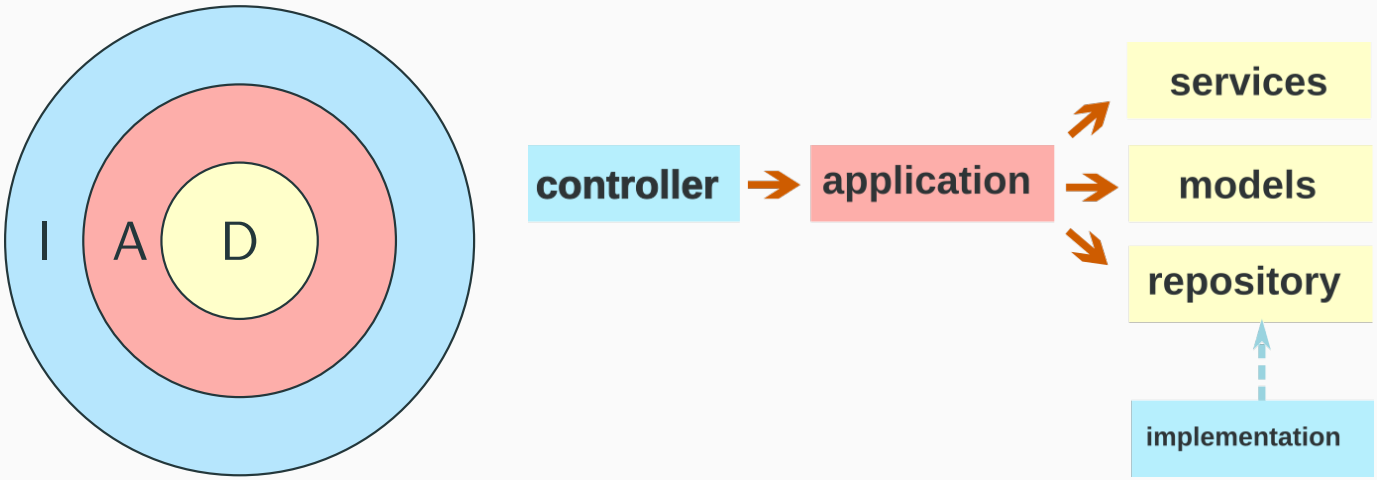


Figura 2: Capas de arquitectura hexagonal

Capas de la arquitectura hexagonal



(a) Vista de capas **(b)** Vista desarrollo

Figura 3: Modelo de capas de arquitectura hexagonal

Caso de estudio

Definición del cado de estudio

Para nuestro ejemplo de estudio se implementará el caso de uso que recupera la información de una capacitación (*training*) dado un identificador de tipo UUID (*Universally Unique Identifier*). Si la capacitación con ese UUID no existe se retornará un valor de error 404 de tipo *not found*. La persistencia se realizará en memoria. El ejercicio se realizará por iteraciones y la primera se enfocará en el flujo necesario para retornar la solicitud de la capacitación. Las demás iteraciones se enfocarán en el implementación utilizando la estructura DDD (*Domain Drive Design*).

Infraestructura

Infraestructura

La capa de infraestructura es el punto de entrada y salida con los demás sistemas. Además se encarga de aislar los elementos como frameworks, librerías, persistencias de vendors externos de las capas de aplicación y dominio.

La capa de infraestructura implementa los adaptadores de la capa de dominio que sean necesarios para cumplir con los requerimientos del caso de uso (capa de aplicación).

Controller

```java
@RestController
@RequestMapping("/api/formacion")
public final class TrainingByIdGetController {

    private final TrainingByIdHandler handler;

    public TrainingByIdGetController(TrainingByIdHandler handler){
        this.handler = handler;
    }

    @GetMapping("/training/{id}")
    public ResponseEntity<TrainingDTO> getTrainingById(@PathVariable UUID id){

        Optional<TrainingDTO> trainingDTO = this.handler.execute(id);

        if (trainingDTO.isPresent()){
            return new ResponseEntity(trainingDTO.get(), HttpStatus.OK);
        } else {
            return new ResponseEntity(new TrainingDTO(""),
                    HttpStatus.NOT_FOUND);
        }
    }
}
```

Aplicación

Aplicación

La capa de aplicación tiene la tarea de ejecutar la lógica de negocio y para ello puede utilizar servicios o repositorios. Los casos de usos pueden ser utilizados por más de un controlador, eventbus o cualquier input del lado de infraestructura, incluso por otros casos de uso.

El caso de uso puede recibir valores primitivos, pero siempre va a retornar y a trabajar con Value Object.

Handler

```java
@Component
public class TrainingByIdHandler {

    private final TrainingDAO dao;

    public TrainingByIdHandler(TrainingDAO dao) {
        this.dao = dao;
    }

    public Optional<TrainingDTO> execute(UUID id){
        return this.dao.getTrainingById(id);
    }
}
```

Dominio

Dominio

En la capa de dominio contiene las clases de almacenamiento y entidades (contienen un UUID que permite identificarse) que modelan nuestro negocio. Las clases del dominio se diseñan para evitar el cambio de estado (no tienen set) y sus datos se asignan al momento de su creación por medio del constructor.

El dominio utiliza puertos que permiten definir contratos de métodos que son implementados por la capa de infraestructura. Utilizando éste enfoque el dominio especifica el contrato e infraestructura los implementas. Un ejemplo contreto son los repositorio, el dominio define los datos que necesita para operar y la infraestructura realizar su implementación cumpliendo con el contrato ya sea utilizando una base de datos, eventbus, archivos planos, etc.

Data Transfer Object - DTO

```
1   public class TrainingDTO {
2
3       private final String name;
4
5       public TrainingDTO(String name) {
6           this.name = name;
7       }
8
9       public String getName() {
10          return name;
11      }
12  }
```

Data Access Object - DAO

```
1  public interface TrainingDAO {
2
3      List<TrainingDTO> getAllTraining();
4      Optional<TrainingDTO> getTrainingById(UUID id);
5
6  }
```

Infraestructura - Implementación Repository

```java
@Repository
public class InMemoryTrainingDAO implements TrainingDAO {

    @Override
    public List<TrainingDTO> getAllTraining() {

        List<TrainingDTO> dtos = new ArrayList<>();

        dtos.add(new TrainingDTO("Basic Java"));
        dtos.add(new TrainingDTO("Hexagonal Architecture"));
        dtos.add(new TrainingDTO("The importance of water in navigation"));
        dtos.add(new TrainingDTO("The super me in existential dilemmas"));

        return dtos;
    }

    @Override
    public Optional<TrainingDTO> getTrainingById(UUID id) {
   // esto hay que analizarlo mejor
        return Optional.ofNullable(
            new TrainingDTO("The super me existential dilemmas"));
    }
}
```

Docker

Comandos de docker

Descarga de una imagen

```
> docker pull ubuntu
```

Listado de imagenes descargadas

```
> docker images
```

REPOSITORY	TAG	IMAGE ID	CREAT
maestro-server	v1.0	a7df52488e71	7 day
redis	6.0-alpine3.11	0b4e853530a5	7 day
rabbitmq	3.8-management-alpine	a6ce06daac72	8 day
mysql/mysql-server	latest	671b2c453d03	8 day
formacion-server	v1.0	d4f8885f07ee	8 day
gateway-server	v1.0	3f0b5eba91c9	8 day
eureka-server	v1.0	8e2215c019d9	8 day
redis	<none>	360360313017	2 wee
rabbitmq	<none>	e64c67658b55	2 wee
mysql/mysql-server	<none>	716286be47c6	2 wee
openjdk	11	f5de33dc9079	3 wee
parrotstream/centos-openjdk	latest	59dbb5986f22	19 mo

Comandos de docker

Listado de imágenes ejecutandose

```
> docker ps -a
```

Listado de imágenes ejecutandose -otra forma-

```
> docker ps --format "table {{.ID}}\t{{.Names}}"
```

```
CONTAINER ID        NAMES
784b92cf40bc        eureka-server
6a110c9f8e9e        gateway-server
3d5f53436e9a        maestro-server
5f4bdc9e48ec        rabbitmq-server
e30d1d74928d        redis-server
```

Comandos de docker

Detener una imagen

```
> docker stop redis-server
```

Eliminar una imagen -debe estar detenida-

```
> docker rm redis-server
```

Correr una imagen

```
> docker run -p 6379:6379 --name redis-server --restart always
       -d redis:6.0-alpine3.11
b519b42a60f7973de76dc8395af2e9b84753d21b0043e9c9779f76c78dcb7dec
```

Anatomia de un Dockerfile

```
1  FROM openjdk:11
2
3  MAINTAINER "Luis Bertel" lbertel@gmail.com
4
5  RUN mkdir /app
6  ADD ./build/libs/formacion-server-1.0.jar /app/formacion-server-1.0.jar
7  ENTRYPOINT ["java","-jar","/app/formacion-server-1.0.jar"]
```

Ejecutar el Dockerfile

```
> docker build -t formacion-server .
```

Se debe estar ubicado en el mismo directorio donde esta el Dockerfile y no olvide el punto al final del comando.

Pendientes

Pendientes

1. Domain Drive Design
    1.1 Implementación de Value Object
    1.2 Aggregate y aggregate root
    1.3 Command para los casos de usos
2. Test
    2.1 Unitarios
    2.2 Aceptación
    2.3 Integración

No olviden los principios de los patrones SOLID y STUPID.