

Arquitetura Hexagonal

Ports & Adapters Pattern

Projeto: arq-hexagonal

Stack: Spring Boot + Gradle Multi-module

Padrões: Hexagonal + Clean + DDD + CQRS

O que é Arquitetura Hexagonal?

Criada por: Alistair Cockburn (2005)

Objetivo: Isolar a lógica de negócio de detalhes técnicos

Também conhecida como:

- Ports and Adapters (nome alternativo)

Este projeto: Hexagonal com influências de Clean e DDD

Problema que Resolve

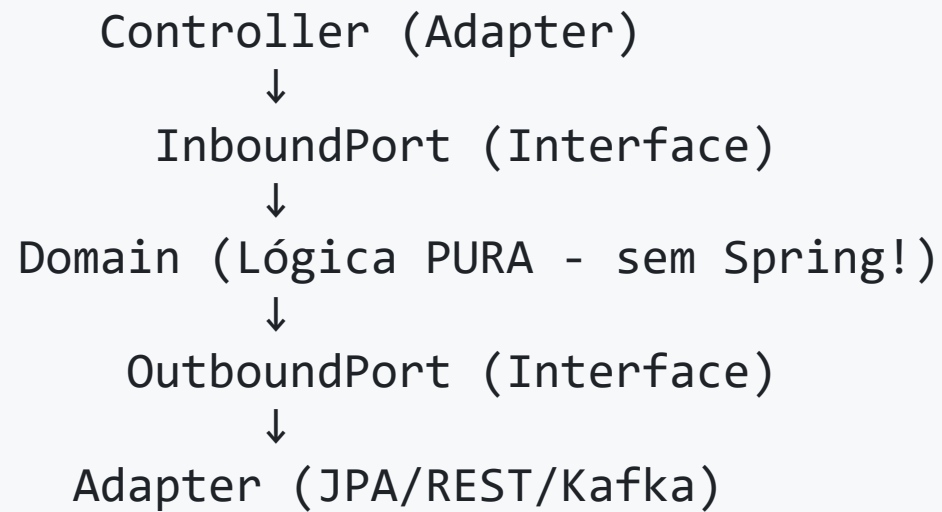
Arquitetura Tradicional (Acoplada)

```
Controller → Service → Repository → Database
    ↓           ↓           ↓
  Spring   @Service    JPA
```

Lógica de negócio ACOPLADA aos frameworks

Problema: Trocar framework = reescrever tudo

✓ Arquitetura Hexagonal (Desacoplada)



Benefício: Trocar framework = só trocar adapter

Estrutura do Projeto

3 Módulos Gradle

```
application/    ← Configuração + Driving Adapters
  ↓ conhece
infrastructure/ ← Driven Adapters (JPA, etc.)
  ↓ conhece
domain/         ← Lógica de Negócio PURA
```

Módulo 1: DOMAIN (Núcleo)

O coração do sistema - 100% puro

```
domain/
├── model/           # Entidades (Usuario)
├── valueobject/     # Value Objects (Email, CPF)
├── usecase/         # Lógica de negócio
├── ports/
│   ├── in/         # InboundPort (entrada)
│   └── out/         # OutboundPort (saída)
└── exception/      # Exceções de domínio
```

Características:

- ❌ SEM Spring, SEM JPA, SEM frameworks
- ✅ Apenas Java puro
- ✅ Protegido por build.gradle

Módulo 2: INFRASTRUCTURE

Adaptadores de Saída (Driven)

```
infrastructure/  
├── persistence/  
│   ├── entity/           # UsuarioEntity (JPA)  
│   ├── repository/       # Spring Data JPA  
│   └── adapter/          # Implementa OutboundPort
```

O que faz:

- Implementa `OutboundPort` usando JPA
- Converte `Value Objects` → `Strings` (banco)
- Tecnologias: Spring Data JPA, H2

Módulo 3: APPLICATION

Configuração + Adaptadores de Entrada (Driving)

```
application/
├── rest/           # Controllers REST
├── service/
│   ├── command/   # Write (passa por Domain)
│   └── query/      # Read (pode fazer bypass)
├── config/        # @Configuration manual
└── exception/     # Handler global
```

O que faz:

- REST Controllers (entrada)
- CQRS (Command/Query)
- Configura beans do Domain manualmente

Portas (Ports)

Interfaces que definem contratos

InboundPort (Entrada)

```
public interface UsuarioInboundPort {  
    Usuario criarUsuario(String nome, Email email, CPF cpf);  
}
```

Quem implementa: Domain (UseCase)

Quem chama: Application (CommandService)

OutboundPort (Saída)

```
public interface UsuarioOutboundPort {  
    Usuario salvar(Usuario usuario);  
    Optional<Usuario> buscarPorId(Long id);  
}
```

Quem implementa: Infrastructure (Adapter)

Quem chama: Domain (UseCase)

Adaptadores (Adapters)

Driving Adapters (Primários)

Iniciam ações → Ficam no APPLICATION

```
@RestController
public class UsuarioController {
    private final UsuarioInboundPort inboundPort;

    @PostMapping
    public Response criar(@RequestBody Request req) {
        return commandService.criar(req);
    }
}
```

Driven Adapters (Secundários)

São chamados → Ficam na INFRASTRUCTURE


```
@Component
public class UsuarioRepositoryAdapter
    implements UsuarioOutboundPort {


    private final UsuarioJpaRepository jpaRepo;

    public Usuario salvar(Usuario usuario) {
        // Converte Domain → JPA
        // Salva no banco
    }
}
```

Decisões Arquiteturais Importantes

1 Use Cases no Domain

 Application/usecase/
UsuarioUseCaseImpl

 Domain/usecase/
UsuarioUseCaseImpl (puro)

Por quê?

"Orquestração de negócio É PARTE DO NEGÓCIO"

Para manter puro, Application cria `@Bean` manualmente

2 Nomenclatura Genérica

✗ `UsuarioUseCase`
(acoplado)

✓ `UsuarioInboundPort`
(genérico)

✗ `UsuarioRepository`
(indica BD)

✓ `UsuarioOutboundPort`
(pode ser BD, REST, Kafka)

Por quê?

"Portas devem ser agnósticas de tecnologia"

3 Domain 100% Puro

```
// ❌ PROIBIDO no Domain
@Service
@Autowired
@Entity
import org.springframework.*

// ✅ PERMITIDO no Domain
public class UsuarioUseCaseImpl { }
import java.util.*
```

Proteção: Build gradle bloqueia frameworks!

4 Controllers no Application

✗ Infrastructure/rest/
UserController

✓ Application/rest/
UserController

Por quê?

"Controllers são Driving Adapters (entrada)"

Infrastructure só tem Driven Adapters (saída)

5 CQRS (Separação Command/Query)

CQRS = Separar Commands de Queries

Commands (Write) → Domain

```
POST → CommandService → Domain (UseCase)
```

Queries (Read) → Domain

```
GET → QueryService → Domain ou Infrastructure
```

6 Bypass (Otimização Adicional)

Bypass = Pular Domain em queries simples

```
GET → QueryService → Infrastructure (pula Domain)
```

Vantagem: Performance (queries sem lógica)

Value Objects

Características:

- ✓ Imutáveis (final, sem setters)
- ✓ Auto-validáveis (validação no construtor)
- ✓ Igualdade por valor (equals)
- ✓ Rico em comportamento
- ✓ Substitui primitivos com regras

Value Objects

Objetos imutáveis definidos pelo valor

Email

```
Email email = Email.of("joao@test.com");  
email.getDomain();           // "test.com"  
email.isFromDomain("test.com"); // true
```

CPF

```
CPF cpf = CPF.of("123.456.789-09");  
cpf.getFormatted(); // "123.456.789-09"  
cpf.getMasked();    // "***.***. 789-09"
```

Value Objects em Ação


Sem Value Objects

```
String email = "invalido"; // Aceita qualquer coisa!  
usuario.setEmail(email);
```

Com Value Objects

```
Email email = Email.of("invalido"); // ✗ Exceção!  
Email email = Email.of("joao@test.com"); // ✓ Validado  
  
usuario.setEmail(email); // Sempre válido!  
email.getDomain(); // "test.com"
```

6 Value Objects nas Ports

```
//  Type-safe e validado cedo
public interface UsuarioInboundPort {
    Usuario criarUsuario(String nome,
                        Email email,    // Value Object
                        CPF cpf);      // Value Object
}
```

Benefícios:

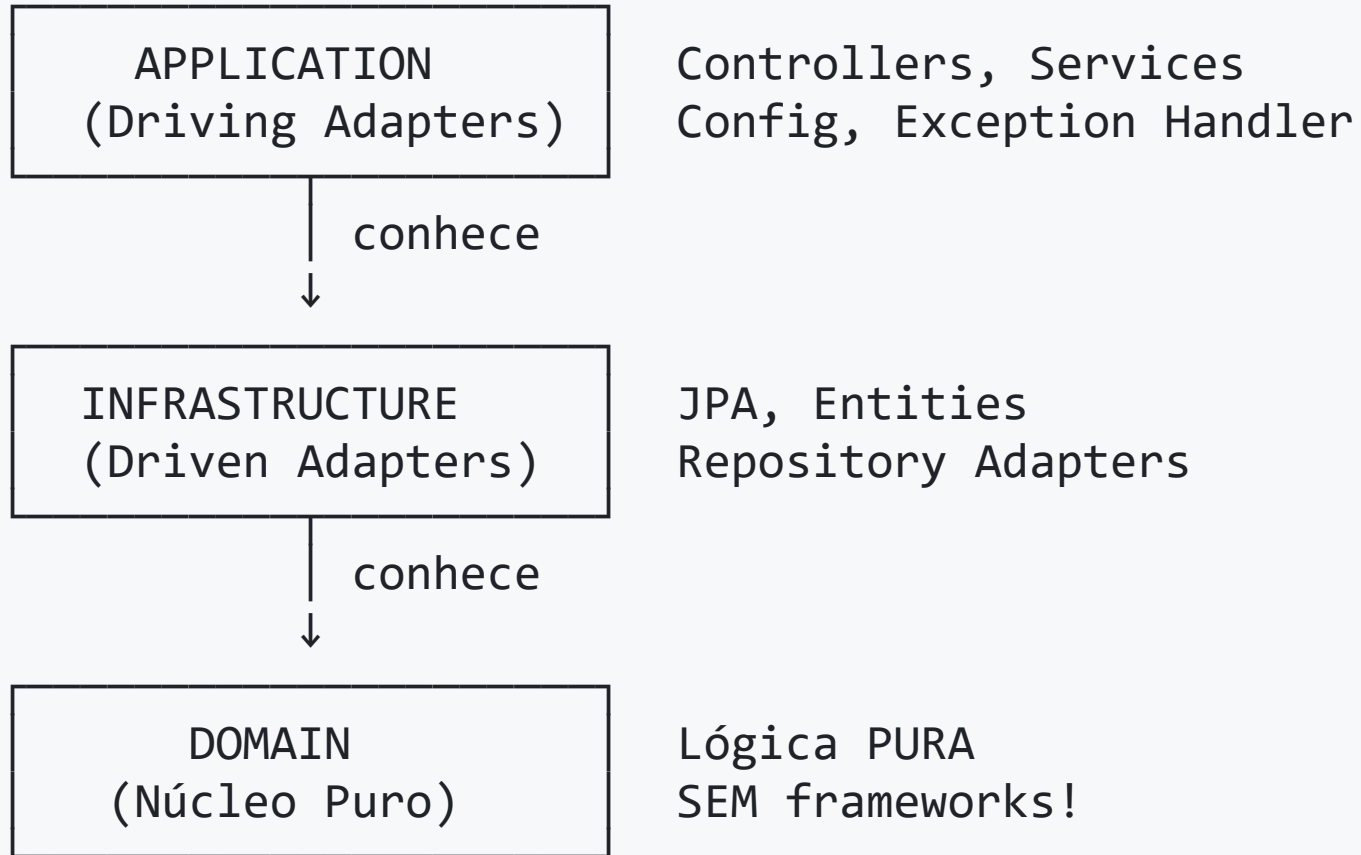
- Type safety (compilador ajuda)
- Validação no Application (fail fast)
- Port autodocumentado

Fluxo Completo

1. HTTP POST /api/usuarios
↓
2. Controller (Application)
↓
3. CommandService converte String → Value Objects
↓
4. Chama InboundPort com Value Objects
↓
5. UseCase (Domain) - lógica de negócio
↓
6. Chama OutboundPort
↓
7. Adapter (Infrastructure) - converte → JPA
↓
8. Database



Diagrama Visual



Padrão 1: CQRS

Command Query Responsibility Segregation

Command (Write)

```
POST /usuarios
  ↓
CommandService ← Orquestra write operations
  ↓
Domain (UseCase) ← Validação, regras
  ↓
Infrastructure
```

⚡ Padrão 2: Bypass (Otimização)

Queries simples pulam Domain

Query com Bypass

```
GET /usuarios  
  ↓  
QueryService  
  ↓  
Infrastructure ← BYPASS Domain (sem lógica)
```

CQRS + Bypass = Dois padrões combinados

Proteções Implementadas

1. Build Protection

```
// domain/build.gradle
if (dependency.contains('springframework')) {
    throw Exception("Domain não pode ter Spring!")
}
```

2. Testes Arquiteturais

```
@Test
void domainNaoDeveImportarSpring() {
    // Verifica automaticamente
    // Falha se encontrar Spring no Domain
}
```

Benefícios Conquistados

1. Independência

- Domain não conhece frameworks
- Pode mudar Spring → outro framework
- Domain portátil

2. Testabilidade

```
// Testes puros, sem Spring
Usuario u = new UsuarioUseCaseImpl(mockPort);
assertNotNull(u.criar(...));
```

3. Flexibilidade

```
// Múltiplas implementações da mesma porta  
class UsuarioJpaAdapter implements UsuarioOutboundPort { }  
class UsuarioMongoAdapter implements UsuarioOutboundPort { }  
class UsuarioRestAdapter implements UsuarioOutboundPort { }
```

Troca sem afetar Domain!

4. Clareza

Responsabilidades bem definidas:

- Domain: Lógica de negócio
- Infrastructure: Tecnologias (BD, APIs)
- Application: Configuração e entrada

Padrões Implementados

- ✓ Arquitetura Hexagonal (Ports & Adapters)
- ✓ Clean Architecture (Domain puro)
- ✓ DDD (Value Objects, Entidades)
- ✓ CQRS (Command/Query Separation)
- ✓ Dependency Inversion
- ✓ Domain Purity (protegido)

Benefícios do Projeto

Type Safety

```
// Impossível confundir  
Usuario criar(String nome, Email email, CPF cpf);  
criar(nome, cpf, email); // ✗ Erro de compilação!
```


Validação Cedo (Fail Fast)

```
try {  
    Email email = Email.of("inválido");  
} catch (Exception e) {  
    // Falha AQUI, antes de chamar Domain  
    return badRequest(e);  
}
```

Testabilidade

```
// Teste puro, sem Spring
@Test
void teste() {
    UsuarioOutboundPort mock = mock(...);
    UsuarioUseCaseImpl useCase = new UsuarioUseCaseImpl(mock);

    Usuario result = useCase.criarUsuario(...);

    assertNotNull(result);
}
// Rápido e simples!
```

Flexibilidade

```
// Trocar implementação
@Bean
public UsuarioOutboundPort outboundPort() {
    // return new UsuarioJpaAdapter();    // Produção
    // return new UsuarioMongoAdapter();  // Outra tech
    return new UsuarioMockAdapter();     // Testes
}
```

Domain não muda!

Estrutura de Arquivos

```
arq-hexagonal/  
├── domain/  
│   ├── model/Usuario.java  
│   ├── valueobject/Email.java, CPF.java  
│   ├── usecase/UsuarioUseCaseImpl.java  
│   └── ports/  
│       ├── in/UsuarioInboundPort.java  
│       └── out/UsuarioOutboundPort.java  
├── infrastructure/  
│   └── persistence/  
│       ├── entity/UsuarioEntity.java  
│       ├── repository/UsuarioJpaRepository.java  
│       └── adapter/UsuarioRepositoryAdapter.java  
└── application/  
    ├── rest/UsuarioController.java  
    ├── service/command/UsuarioCommandService.java  
    ├── service/query/UsuarioQueryService.java  
    └── config/UseCaseConfiguration.java
```

Verificação de Pureza

```
# Domain não deve ter Spring
grep -r "org.springframework" domain/src/main/java/
# (vazio = sucesso ✅)

# Testes arquiteturais
./gradlew :domain:test --tests ArchitectureTest

# Build
./gradlew build
```



Como Executar

```
# Compilar
./gradlew build

# Executar
./gradlew :application:bootRun

# Testar API
curl -X POST http://localhost:8080/api/usuarios \
  -H "Content-Type: application/json" \
  -d '{"nome":"João","email":"joao@test.com"}'
```

Documentação Completa

Arquivo	Conteúdo
README.md	Visão geral
ARCHITECTURE.md	Diagramas detalhados
CQRS.md	CQRS explicado
VALUE-OBJECTS.md	Guia de Value Objects
DOMAIN-PURITY.md	Como manter puro
PORTS-NOMENCLATURE.md	Nomenclatura
BEST-PRACTICES-SUMMARY.md	Todas decisões
GUIA-RAPIDO.md	Referência rápida

Checklist de Qualidade

- ✓ Domain 100% puro (sem frameworks)
- ✓ Nomenclatura genérica (InboundPort/OutboundPort)
- ✓ Use Cases no Domain
- ✓ Controllers no Application
- ✓ Value Objects implementados
- ✓ CQRS com bypass
- ✓ Testes completos
- ✓ Build protegido
- ✓ Documentação extensa

Princípios Aplicados

SOLID

- SRP: Cada classe tem uma responsabilidade
- OCP: Aberto para extensão, fechado para modificação
- LSP: Substituição de Liskov (ports)
- ISP: Interfaces segregadas
- DIP: Inversão de dependência (Domain no centro)

Lições Aprendidas

1. Orquestração É Negócio

Use Cases ficam no Domain, não no Application

2. Portas São Genéricas

InboundPort/OutboundPort, não UseCase/Repository

3. Application Adapta

Application converte tipos externos → Domain

4. Domain É Puro

Sem Spring, sem JPA, sem frameworks!

5. Value Objects São Poderosos

Encapsulam validação e comportamento

6. CQRS Otimiza

Write via Domain, Read pode fazer bypass

Resultado Final

Arquitetura:

- Profissional ✓
- Type-safe ✓
- Testável ✓
- Flexível ✓
- Documentada ✓

Padrões:

- Hexagonal ✓
- Clean ✓
- DDD ✓
- CQRS ✓

Referências

- Alistair Cockburn - Hexagonal Architecture (2005)
- Robert C. Martin - Clean Architecture (2012)
- Eric Evans - Domain-Driven Design (2003)
- Greg Young - CQRS Pattern (2010)

Perguntas?

Repositório: github.com/seu-usuario/arq-hexagonal

Documentação: README.md e arquivos .md no projeto

Contato: seu@email.com

Obrigado! 🎉

Arquitetura Hexagonal
com Domain Puro, Value Objects e CQRS

