🏛️ Formation Complète : Architecture Hexagonale

Table des Matières

---

1. 📖 Introduction

L'**Architecture Hexagonale** (également appelée **Ports and Adapters**) est un pattern architectural créé par Alistair Cockburn en 2005. Elle vise à créer des applications faiblement couplées, testables et maintenables.

🎯 Objectifs principaux

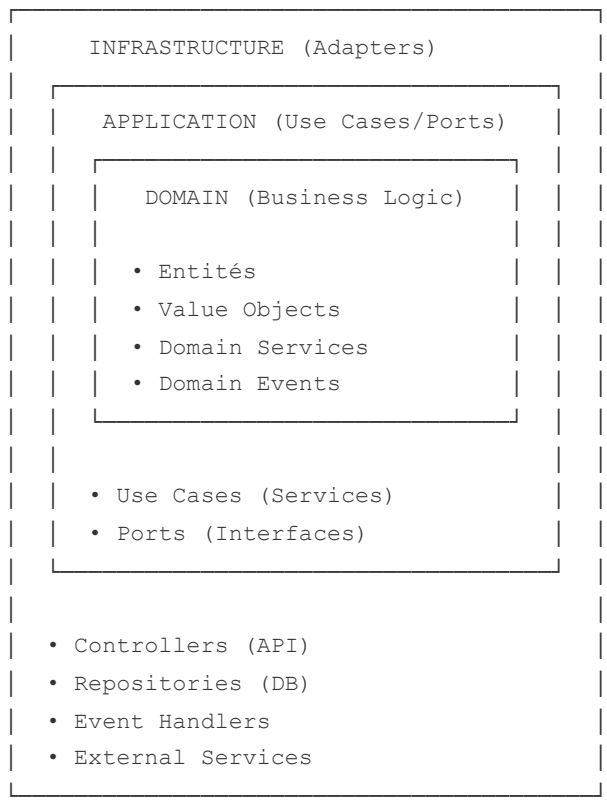
- 🔌 **Indépendance du framework** : La logique métier ne dépend pas des frameworks
- ✅ **Testabilité** : Possibilité de tester la logique métier sans dépendances externes
- 🔄 **Évolutivité** : Facilité de changer les technologies sans impacter le métier
- 📦 **Séparation des responsabilités** : Chaque couche a un rôle bien défini

---

2. 🔷 Définition et Origines

❓ Qu'est-ce que l'Architecture Hexagonale ?

L'architecture hexagonale organise le code en **trois couches principales** :



🤔 Pourquoi "Hexagonale" ?

Le terme "hexagone" est purement symbolique. Il représente que l'application a plusieurs "côtés" (ports) par lesquels elle peut interagir avec le monde extérieur. Le nombre de côtés n'est pas limité à 6.

---

3. 💡 Les Concepts Fondamentaux

3.1 🔌 Les Ports

Un **Port** est une **interface** qui définit un contrat d'interaction.

Deux types de ports :

📩 Ports Entrants ou Port In (Driving Ports)

- ✨ Définissent ce que l'application peut faire (Use Cases)

- 🌐 Appelés par les acteurs externes (UI, API = Application Programming Interface, Tests)

Exemple dans notre code :

```
// Port entrant : définit le cas d'usage
public interface CreateAppointmentRefUseCase {
    String createAppointmentRef(CreateAppointmentRefCommand command)
        throws AppointmentRefAlreadyExistException, FieldNotGivenException;
}
```

📤 Ports Sortants ou Port Out (Driven Ports)

- 🗄️ Définissent ce dont l'application a besoin (Repository, Services externes)
- ⚙️ Implémentés par l'infrastructure
- 🔗 Appelés par le domaine (SPI = Service Provider Interface)

Exemple dans notre code :

```
// Port sortant : définit les besoins de persistance
public interface AppointmentRefRepository {
    Optional<AppointmentRef> findById(String id);
    List<AppointmentRef> findAll();
    void save(AppointmentRef appointmentref);
    void delete(AppointmentRef appointmentref);
}
```

3.2 🔧 Les Adaptateurs

Un **Adaptateur** est une **implémentation** d'un port.

Deux types d'adaptateurs :

🎨 Adaptateurs Entrants (Driving Adapters)

- 📨 Reçoivent les requêtes externes
- 📞 Appellent les ports entrants

Exemple : Controller REST

```
@Path("/appointmentRef")
public class AppointmentRefController {
    private final CreateAppointmentRefUseCase createAppointmentRefUseCase;

    @POST
    @Path("/")
    public RestResponse<IdDto<String>> createAppointmentRef(
        CreateAppointmentRefCommand command) {
        String id = createAppointmentRefUseCase.createAppointmentRef(command);
        return RestResponse.ResponseBuilder.ok(new IdDto<>(id, "success")).build();
```

```
    }
}
```

🔌 Adaptateurs Sortants (Driven Adapters)

- ✓ Implémentent les ports sortants
- 🌍 Interfacent avec les technologies externes

Exemple : Repository JPA

```
@ApplicationScoped
public class AppointmentRefJpaRepository implements AppointmentRefRepository {
    private final AppointmentRefPanacheRepository panacheRepository;

    @Override
    public void save(AppointmentRef appointmentRef) {
        AppointmentRefEntity entity = mapper.toEntity(appointmentRef);
        panacheRepository.getEntityManager().merge(entity);
    }
}
```

---

4. 📂 Structure du Projet

Notre projet est organisé selon l'architecture hexagonale :

```
src/main/java/ci/orange/archi/hexago/
├── domain/                          # COUCHE DOMAINE
│   ├── common/                      # Utilitaires domaine partagés
│   │   ├── base/
│   │   │   ├── BaseDomain.java     # Classe de base pour entités
│   │   │   └── DomainOperation*.java
│   │   ├── event/                   # Événements du domaine
│   │   │   ├── DomainEvent.java
│   │   │   └── DomainEventPublisher.java
│   │   ├── exception/               # Exceptions métier
│   │   ├── pagination/              # Logique de pagination
│   │   └── search/                  # Logique de recherche
│   │
│   └── core/                        # Logique métier spécifique
│       └── appointmentref/
│           ├── AppointmentRef.java           # ENTITÉ
│           ├── dto/
│           │   ├── CreateAppointmentRefCommand.java  # DTO
│           │   └── GetAppointmentRefDto.java
│           ├── exception/
│           │   └── AppointmentRefNotFoundException.java
│           ├── event/
│           │   └── AppointmentRefCreatedEvent.java
│           ├── repository/
```

```
│           │   └── AppointmentRefRepository.java     # PORT SORTANT
│           ├── use_case/
│           │   └── CreateAppointmentRefUseCase.java  # PORT ENTRANT
│           └── service/
│               └── CreateAppointmentRefService.java  # IMPLÉMENTATION
│
└── infrastructure/                  # COUCHE INFRASTRUCTURE
    ├── common/                      # Utilitaires infrastructure
    │   ├── contract/                # DTOs de réponse API
    │   └── search/                  # Parsers de requêtes
    │
    └── adapter/                     # ADAPTATEURS
        └── appointmentref/
            ├── controller/
            │   └── AppointmentRefController.java    # ADAPTATEUR ENTRANT (REST)
            ├── entity/
            │   ├── AppointmentRefEntity.java        # ENTITÉ JPA
            │   └── AppointmentRefMapper.java        # MAPPER
            ├── repository/
            │   ├── AppointmentRefJpaRepository.java # ADAPTATEUR SORTANT (JPA)
            │   └── AppointmentRefPanacheRepository.java
            ├── event/
            │   └── AppointmentRefCreatedEventHandler.java
            └── AppointmentRefAppConfig.java         # CONFIGURATION (DI)
```

---

5. 💎 Le Domaine (Domain)

Le **Domaine** est le cœur ❤️ de l'application. Il contient toute la logique métier et est **complètement indépendant** des frameworks et technologies.

5.1 🏷️ Les Entités (Entities)

Les entités représentent les **objets métier** avec une identité unique.

Exemple : AppointmentRef

```
@Data
@Accessors(fluent = true)  // Permet des accesseurs fluides : obj.id()
public class AppointmentRef extends BaseDomain {
    private String id;
    private String href;
    private String description;
    private String baseType;
    private String schemaLocation;
    private String type;
    private String referredType;
}
```

Caractéristiques : - 🚫 Pas d'annotations JPA (`@Entity`, `@Table`, etc.) - ⚡ Utilise Lombok pour réduire le boilerplate - 🧬 Hérite de `BaseDomain` pour les fonctionnalités communes

5.2 📦 La Classe BaseDomain

```
@Data
@Accessors(fluent = true)
public abstract class BaseDomain<T> {
    private final List<DomainEvent> domainEvents = new ArrayList<>();
    protected T user;
    protected Date createdAt;
    protected Date updatedAt;

    public void addDomainEvent(DomainEvent event) {
        domainEvents.add(event);
    }
}
```

Fonctionnalités : - 📣 Gestion des **événements du domaine** - 📅 Métadonnées communes (dates, utilisateur) - 🔄 Pattern **Event Sourcing** simplifié

5.3 📋 Les Value Objects (DTOs)

Les DTOs transportent les données sans identité propre.

Command (pour la création/modification) :

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CreateAppointmentRefCommand {
    protected String href;
    protected String description;
    // ...

    public AppointmentRef toDomain() {
        return new AppointmentRef()
            .href(href)
            .description(description)
            .baseType(baseType);
    }

    public Map<String, Object> toMap() {
        Map<String, Object> map = new HashMap<>();
        map.put("href", href);
        map.put("description", description);
        return map;
    }
}
```

Query DTO (pour la lecture) :

```
@Data
public class GetAppointmentRefDto {
    private String id;
    private String href;
    private String description;

    public GetAppointmentRefDto(AppointmentRef appointmentRef) {
        this.id = appointmentRef.id();
        this.href = appointmentRef.href();
        this.description = appointmentRef.description();
    }
}
```

5.4 📣 Les Événements du Domaine

Les événements capturent ce qui s'est passé dans le domaine.

```
public class AppointmentRefCreatedEvent extends GenericDomainEvent {
    public AppointmentRefCreatedEvent(String id, Map<String, Object> data) {
        super("AppointmentRefCreated", id, data);
    }
}
```

Utilisation : - 📊 Suivi des changements - 💬 Communication entre bounded contexts - 🔍 Audit et traçabilité

5.5 ⚠️ Les Exceptions du Domaine

```
public class AppointmentRefNotFoundException extends NotFoundException {
    public AppointmentRefNotFoundException(String field, String value) {
        super("AppointmentRef", field, value);
    }
}
```

Hiérarchie cohérente : - 🌳 Héritent de `BaseCustomException` - 📝 Messages standardisés - 🎯 Gestion centralisée

---

6. ⚙️ L'Application (Use Cases)

La couche application orchestre les cas d'usage métier.

6.1 🚪 Les Ports Entrants (Use Cases Interface)

```
public interface CreateAppointmentRefUseCase {
    String createAppointmentRef(CreateAppointmentRefCommand command)
```

```
        throws AppointmentRefAlreadyExistException, FieldNotGivenException;
}
```

Caractéristiques : - 🎯 Interface simple et claire - ⚠️ Déclare les exceptions métier - 🔒 Indépendante de l'implémentation

6.2 🛠️ Les Services (Use Cases Implementation)

```
public class CreateAppointmentRefService implements CreateAppointmentRefUseCase {
    private final AppointmentRefRepository appointmentRefRepository;
    private final ServiceOrderItemRepository serviceOrderItemRepository;

    public CreateAppointmentRefService(
        AppointmentRefRepository appointmentRefRepository,
        ServiceOrderItemRepository serviceOrderItemRepository) {
        this.appointmentRefRepository = appointmentRefRepository;
        this.serviceOrderItemRepository = serviceOrderItemRepository;
    }

    @Override
    public String createAppointmentRef(CreateAppointmentRefCommand command)
        throws FieldNotGivenException {
        final String id = Utilities.uuid();

        return DomainOperationExecutor.executeWithEventHandling(
            command,
            command.toMap(),
            cmd -> {
                // 1. Validation des champs requis
                Map<String, Object> requiredFields = new HashMap<>();
                requiredFields.put("href", cmd.getHref());
                requiredFields.put("description", cmd.getDescription());
                Validator.requireNonNull(requiredFields);

                // 2. Conversion Command -> Domain
                AppointmentRef appointmentRef = cmd.toDomain();

                // 3. Logique métier (si nécessaire)
                // ...

                // 4. Persistance via le port sortant
                appointmentRef.id(id);
                appointmentRefRepository.save(appointmentRef);

                return appointmentRef.id();
            },
            (cmd, cmdMap) -> new AppointmentRefCreatedEvent(id, cmdMap),
            (errorCode, errorMessage) -> (cmd, cmdMap)
                -> new AppointmentRefCreationFailedEvent(errorCode, errorMessage, cmdMap)
        );
    }
}
```

Points clés : 1. 🔪 **Injection de dépendances** : Les repositories sont injectés (Dependency Inversion) 2. ✓ **Validation** : Utilise `Validator.requireNonNull()` 3. 🔄 **Transformation** : Command → Domain 4. 📣 **Publication d'événements** : Via

`DomainOperationExecutor` 5. 🚫 **Pas de dépendance technique** : Uniquement des interfaces

6.3 🗄️ Les Ports Sortants (Repository Interface)

```
public interface AppointmentRefRepository {
    Optional<AppointmentRef> findById(String id);
    List<AppointmentRef> findByHref(String href);
    List<AppointmentRef> findAll();

    PaginatedData<GetAppointmentRefDto> findAppointmentRefByCriteria(
        CriteriaWrapper criteria,int page, int size
    ) throws FieldNotGivenException;

    void save(AppointmentRef appointmentref);
    void delete(AppointmentRef appointmentref);
}
```

Caractéristiques : - 🏠 Interface dans le **domaine** - 💼 Manipulation d'objets **métier** (pas d'entités JPA) - 🤷 Abstraction de la persistance

---

7. 🏗️ L'Infrastructure (Adapters)

L'infrastructure contient toutes les implémentations techniques.

7.1 📩 Les Adaptateurs Entrants

🌐 Le Controller REST

```
@ApplicationScoped
@Transactional(rollbackOn = {BaseCustomException.class})
@Path("/appointmentRef")
@Produces(MediaType.APPLICATION_JSON)
public class AppointmentRefController {
    private final CreateAppointmentRefUseCase createAppointmentRefUseCase;
    private final UpdateAppointmentRefUseCase updateAppointmentRefUseCase;
    // ...

    public AppointmentRefController(
        CreateAppointmentRefUseCase createAppointmentRefUseCase,
        UpdateAppointmentRefUseCase updateAppointmentRefUseCase,
        // ...
    ) {
        this.createAppointmentRefUseCase = createAppointmentRefUseCase;
        this.updateAppointmentRefUseCase = updateAppointmentRefUseCase;
        // ...
    }

    @POST
```

```
    @Path("/")
    public RestResponse<IdDto<String>> createAppointmentRef(
        CreateAppointmentRefCommand command
    ) throws AppointmentRefAlreadyExistException, FieldNotGivenException {
        String id = createAppointmentRefUseCase.createAppointmentRef(command);
        return RestResponse.ResponseBuilder.ok(
            new IdDto<>(id, "success")
        ).build();
    }
}
```

Rôle du Controller : - 🌐 Gère les requêtes HTTP - ➡️ Délègue au use case - 🔄 Transforme les réponses métier en JSON - ⚠️ Gère les exceptions via `@Transactional`

7.2 📤 Les Adaptateurs Sortants

🗃️ L'Entité JPA

```
@Data
@Entity
@Table(name="appointment_ref")
public class AppointmentRefEntity implements Serializable {
    @Id
    @Column(name="id", nullable=false, length=255)
    private String id;

    @Column(name="href", length=500)
    private String href;

    @Column(name="description", length=500)
    private String description;

    // ...
}
```

Caractéristiques : - 🏷️ Annotations JPA (@Entity, @Table, @Column) - 📍 Située dans `infrastructure/adapter` - ✂️ **Séparée** de l'entité domaine

🔄 Le Mapper

```
@Mapper(componentModel = "cdi")
public interface AppointmentRefMapper {
    // Domain -> Entity
    AppointmentRefEntity toEntity(AppointmentRef domain);

    // Entity -> Domain
    AppointmentRef toDomain(AppointmentRefEntity entity);
    Optional<AppointmentRef> toDomainOptional(AppointmentRefEntity entity);
    List<AppointmentRef> toDomainList(List<AppointmentRefEntity> entities);

    // Entity -> DTO
    GetAppointmentRefDto toDto(AppointmentRefEntity entity);
```

```
    List<GetAppointmentRefDto> toDtos(List<AppointmentRefEntity> entities);
}
```

Utilise MapStruct : - 🤖 Génération automatique du code de mapping - ⚡ Performance optimale - 🔒 Type-safe

💾 Le Repository JPA

```
@ApplicationScoped
public class AppointmentRefJpaRepository implements AppointmentRefRepository {
    private final AppointmentRefPanacheRepository panacheRepository;
    private final AppointmentRefMapper mapper;

    @Override
    public Optional<AppointmentRef> findById(String id) {
        AppointmentRefEntity entity = panacheRepository
            .find("id = :id", Map.of("id", id))
            .firstResult();
        return entity != null
            ? mapper.toDomainOptional(entity)
            : Optional.empty();
    }

    @Override
    public void save(AppointmentRef appointmentRef) {
        AppointmentRefEntity entity = mapper.toEntity(appointmentRef);
        AppointmentRefEntity entitySaved = panacheRepository
            .getEntityManager()
            .merge(entity);
        appointmentRef.id(entitySaved.getId());
    }
}
```

Responsabilités : 1. ✅ Implémente le port `AppointmentRefRepository` 2. 🏛️ Utilise Panache pour JPA 3. 🔄 Convertit Entity ↔ Domain via Mapper 4. 💾 Gère la persistance

Le Repository Panache

```
@ApplicationScoped
public class AppointmentRefPanacheRepository
    implements PanacheRepositoryBase<AppointmentRefEntity, String> {
}
```

Simple wrapper autour de Panache pour bénéficier de ses fonctionnalités.

7.3 ⚙️ La Configuration (Dependency Injection)

```
public class AppointmentRefAppConfig {
    @Inject
    Instance<AppointmentRefRepository> appointmentRefRepository;
    @Inject
    Instance<ServiceOrderItemRepository> serviceOrderItemRepository;
```

```
    @Produces
    @ApplicationScoped
    CreateAppointmentRefUseCase createAppointmentRefUseCase() {
        return new CreateAppointmentRefService(
            appointmentRefRepository.get(),
            serviceOrderItemRepository.get()
        );
    }
}
```

Rôle : - 🔌 **Wire** les dépendances - 🏭 Instancie les services avec leurs dépendances - ☕ Utilise CDI (Quarkus)

---

8. 🔁 Les Flux de Données

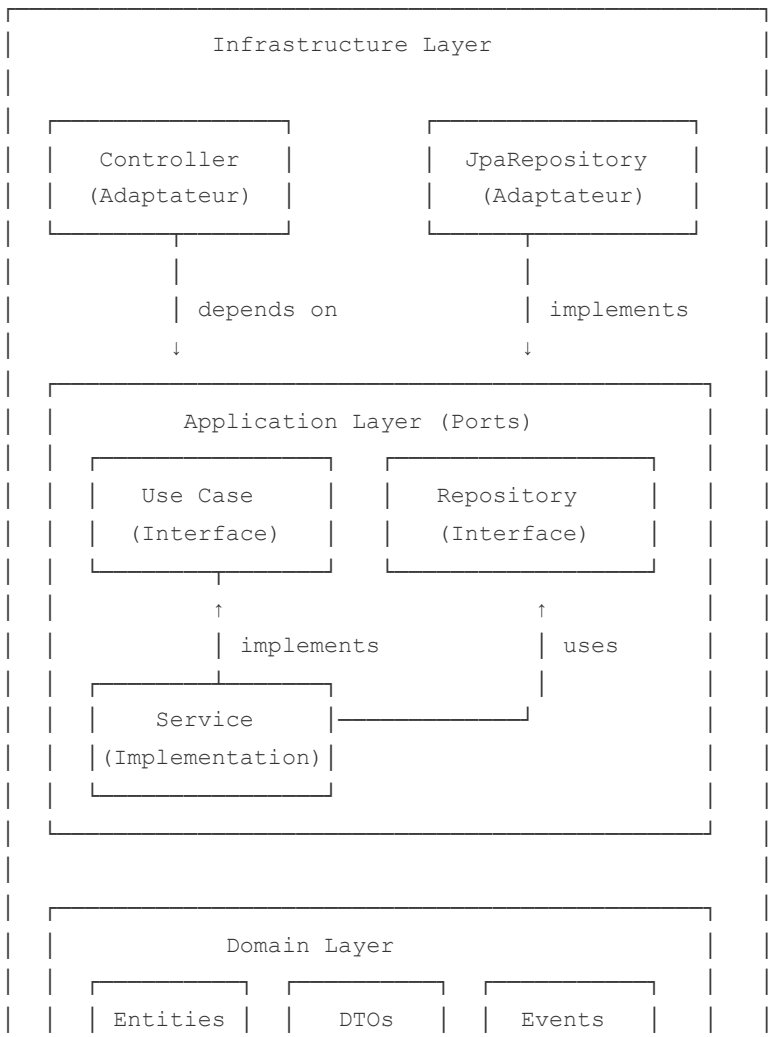
8.1 ✏️ Flux de Création (POST)

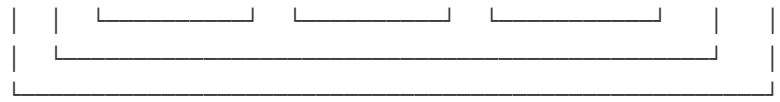
```
1. Client HTTP
   ↓ POST /appointmentRef + CreateAppointmentRefCommand

2. AppointmentRefController (Adaptateur Entrant)
   ↓ Reçoit la requête
   ↓ Appelle le Use Case

3. CreateAppointmentRefUseCase (Port Entrant)
   ↓ Interface

4. CreateAppointmentRefService (Implémentation Use Case)
   ↓ Valide les données
   ↓ Convertit Command → Domain
   ↓ Appelle Repository.save()

5. AppointmentRefRepository (Port Sortant)
   ↓ Interface

6. AppointmentRefJpaRepository (Adaptateur Sortant)
   ↓ Convertit Domain → Entity (via Mapper)
   ↓ Persiste en base (via Panache)

7. Base de données PostgreSQL
   ↓ Données sauvegardées

8. Retour : ID généré
   ↓ Entity → Domain (via Mapper)
   ↓ Domain → Service
   ↓ Service → Controller
   ↓ Controller → Response JSON

9. Client reçoit : { "id": "uuid", "message": "success" }
```

8.2 📖 Flux de Lecture (GET)

```
1. Client HTTP
   ↓ GET /appointmentRef/{id}

2. AppointmentRefController
   ↓ Extrait l'ID du path
   ↓ Appelle getAppointmentRefByIdUseCase

3. GetAppointmentRefByIdService
   ↓ Appelle repository.findById(id)

4. AppointmentRefJpaRepository
   ↓ Requête JPA findById
   ↓ Entity → Domain (via Mapper)

5. Retour : AppointmentRef (Domain)
   ↓ Service → Controller
   ↓ Domain → GetAppointmentRefByIdDto
   ↓ DTO → Response JSON

6. Client reçoit : { "id": "...", "href": "...", ... }
```

8.3 🔄 Dépendances et Inversion de Contrôle





⭐ Principe clé : Les dépendances pointent VERS LE DOMAINE

---

9. 💻 Exemples Pratiques

🔍 Exemple 1 : Recherche par Critères

```
// 1. Controller reçoit les critères
@POST
@Path("/by-criteria")
public RestResponse<PaginatedData<GetAppointmentRefDto>>
getAppointmentRefByCriteria(SearchCriteria searchCriteria) {
    PaginatedData<GetAppointmentRefDto> result =
        getAppointmentRefByCriteriaUseCase.getAppointmentRefByCriteria(searchCriteria);
    return RestResponse.ResponseBuilder.ok(result).build();
}

// 2. Service traite la logique
public class GetAppointmentRefByCriteriaService
    implements GetAppointmentRefByCriteriaUseCase {

    @Override
    public PaginatedData<GetAppointmentRefDto> getAppointmentRefByCriteria(
        SearchCriteria searchCriteria
    ) throws InvalidOperatorException, FieldNotGivenException {

        CriteriaWrapper criteria = conditionUtils.buildCriteria(
            searchCriteria.getConditions(),
            JpaFieldEnums.AppointmentRef.class
        );

        return appointmentRefRepository.findAppointmentRefByCriteria(
            criteria,
            searchCriteria.getPage(),
            searchCriteria.getSize()
        );
    }
}

// 3. Repository exécute la requête
@Override
public PaginatedData<GetAppointmentRefDto> findAppointmentRefByCriteria(
    CriteriaWrapper criteria,
    int page,
    int size
) {
    List<AppointmentRefEntity> entities = panacheRepository
```

```
        .find(criteria.criteria(), criteria.parameters())
        .page(page, size)
        .list();

    long count = panacheRepository.count(criteria.criteria(), criteria.parameters());

    return PaginatedData.of(mapper.toDtos(entities), page, size, count);
}
```

📣 Exemple 2 : Gestion des Événements

```
// Publication d'événement lors de la création
return DomainOperationExecutor.executeWithEventHandling(
    command,
    command.toMap(),
    cmd -> {
        // Logique métier
        appointmentRefRepository.save(appointmentRef);
        return appointmentRef.id();
    },
    // Événement de succès
    (cmd, cmdMap) -> new AppointmentRefCreatedEvent(id, cmdMap),
    // Événement d'échec
    (errorCode, errorMessage) -> (cmd, cmdMap)
        -> new AppointmentRefCreationFailedEvent(errorCode, errorMessage, cmdMap)
);

// Handler d'événement
@ApplicationScoped
public class AppointmentRefCreatedEventHandler {

    public void handle(@Observes AppointmentRefCreatedEvent event) {
        log.info("AppointmentRef créé : {}", event.getEntityId());
        // Logique additionnelle : notification, analytics, etc.
    }
}
```

---

10. ✅ Bonnes Pratiques

10.1 ✂️ Séparation Domain / Infrastructure

✅ **CORRECT** :

```
// domain/core/appointmentref/AppointmentRef.java
@Data
@Accessors(fluent = true)
public class AppointmentRef extends BaseDomain {
    private String id;
```

```
    private String href;
    // Pas d'annotations JPA !
}

// infrastructure/adapter/appointmentref/entity/AppointmentRefEntity.java
@Entity
@Table(name="appointment_ref")
public class AppointmentRefEntity {
    @Id
    private String id;
    @Column(name="href")
    private String href;
}
```

❌ **INCORRECT** :

```
// Mélange domaine et infrastructure
@Entity  // ❌ Annotation JPA dans le domaine
@Table(name="appointment_ref")
public class AppointmentRef {
    @Id  // ❌
    private String id;
}
```

10.2 ➡️ Dépendances unidirectionnelles

✅ **CORRECT** :

```
Infrastructure → Application → Domain
```

❌ **INCORRECT** :

```
Domain → Infrastructure  (🚫 Jamais !)
```

10.3 🎯 Use Cases atomiques

✅ **CORRECT** : Un use case = une action métier

```
CreateAppointmentRefUseCase
UpdateAppointmentRefUseCase
DeleteAppointmentRefUseCase
GetAppointmentRefByIdUseCase
```

❌ **INCORRECT** : Use case "fourre-tout"

```
AppointmentRefUseCase {
    create()
    update()
```

```
    delete()
    findAll()
    // ❌ Trop de responsabilités
}
```

10.4 📋 DTOs pour les frontières

✅ **CORRECT** :

```
// Controller reçoit un Command
@POST
public RestResponse<IdDto> create(CreateAppointmentRefCommand command) {
    // ...
}

// Controller retourne un DTO
@GET
public RestResponse<GetAppointmentRefDto> getById(String id) {
    AppointmentRef domain = useCase.getById(id);
    return new GetAppointmentRefDto(domain);  // Conversion explicite
}
```

❌ **INCORRECT** :

```
// Exposition directe de l'entité domaine
@GET
public AppointmentRef getById(String id) {  // ❌
    return useCase.getById(id);
}
```

10.5 ✓ Validation dans le domaine

✅ **CORRECT** :

```
public String createAppointmentRef(CreateAppointmentRefCommand command) {
    Map<String, Object> requiredFields = Map.of(
        "href", command.getHref(),
        "description", command.getDescription()
    );
    Validator.requireNonNull(requiredFields);  // ✅ Validation métier
    // ...
}
```

10.6 🔄 Mappers dédiés

✅ **CORRECT** :

```
@Mapper(componentModel = "cdi")
public interface AppointmentRefMapper {
    AppointmentRefEntity toEntity(AppointmentRef domain);
    AppointmentRef toDomain(AppointmentRefEntity entity);
}
```

---

11. 🚫 Anti-Patterns à Éviter

❌ 1. Anemic Domain Model

Problème : Entités sans logique, tout dans les services.

```
// ❌ Entité anémique
public class AppointmentRef {
    private String id;
    private String href;
    // Seulement des getters/setters, aucune logique
}

// ❌ Toute la logique dans le service
public class AppointmentRefService {
    public void validate(AppointmentRef ref) {
        if (ref.getHref() == null || ref.getHref().isEmpty()) {
            throw new Exception("Invalid href");
        }
    }
}
```

✅ **Solution** : Logique métier dans le domaine

```
public class AppointmentRef extends BaseDomain {
    private String href;

    public void updateHref(String newHref) {
        if (newHref == null || newHref.isEmpty()) {
            throw new InvalidHrefException("Href cannot be empty");
        }
        this.href = newHref;
        addDomainEvent(new HrefUpdatedEvent(this.id, newHref));
    }
}
```

❌ 2. Dépendance du Domain vers Infrastructure

```
// ❌ Dans le domain
import ci.orange.archi.hexago.infrastructure.adapter.SomeClass;

public class AppointmentRef {
    private SomeClass infraComponent;  // ❌ Violation !
}
```

✅ **Solution** : Injection d'interface (Port)

```
// ✅ Interface dans le domain
public interface NotificationService {
    void notify(String message);
}

// ✅ Implem dans infrastructure
public class EmailNotificationService implements NotificationService {
    // ...
}
```

❌ 3. Repositories qui retournent des Entities JPA

```
// ❌ Repository qui expose des entités JPA
public interface AppointmentRefRepository {
    AppointmentRefEntity findById(String id);  // ❌
}
```

✅ **Solution** : Retourner des objets domaine

```
public interface AppointmentRefRepository {
    Optional<AppointmentRef> findById(String id);  // ✅
}
```

❌ 4. Controllers avec logique métier

```
// ❌ Logique métier dans le controller
@POST
public Response create(CreateCommand cmd) {
    if (cmd.getHref() == null) {  // ❌ Validation métier
        return Response.status(400).build();
    }

    AppointmentRef ref = new AppointmentRef();
    ref.setHref(cmd.getHref());  // ❌ Construction manuelle
    repository.save(ref);  // ❌ Accès direct au repository

    return Response.ok().build();
}
```

✅ **Solution** : Déléguer au use case

```
@POST
public Response create(CreateAppointmentRefCommand cmd) {
    String id = createUseCase.createAppointmentRef(cmd);  // ✅
    return Response.ok(new IdDto(id)).build();
}
```

❌ 5. Entités Domain = Entités JPA

```
// ❌ Une seule classe pour les deux
@Entity  // JPA
@Table(name = "appointment_ref")
@Data
public class AppointmentRef {  // Utilisé partout
    @Id
    private String id;
    // ...
}
```

✅ **Solution** : Séparation claire

```
// Domain
public class AppointmentRef extends BaseDomain { }

// Infrastructure
@Entity
public class AppointmentRefEntity { }

// Mapper entre les deux
@Mapper
public interface AppointmentRefMapper { }
```

---

12. 📝 Exercices Pratiques

📚 Exercice 1 : Créer une nouvelle entité

🎯 **Objectif** : Créer l'entité `Customer` avec architecture hexagonale.

📝 **Étapes** : 1. Créer `Customer.java` dans `domain/core/customer/` 2. Créer `CreateCustomerCommand` dans `domain/core/customer/dto/` 3. Créer `CustomerRepository` (interface) dans `domain/core/customer/repository/` 4. Créer `CreateCustomerUseCase` dans `domain/core/customer/use_case/` 5. Créer `CreateCustomerService` dans `domain/core/customer/service/` 6. Créer `CustomerEntity` dans `infrastructure/adapter/customer/entity/` 7. Créer `CustomerMapper` 8. Créer `CustomerJpaRepository` 9. Créer `CustomerController` 10. Créer `CustomerAppConfig`

🛠️ Exercice 2 : Ajouter une règle métier

📌 **Contexte** : Un `AppointmentRef` ne peut être supprimé que si son `type` est "DRAFT".

📝 **À faire** : 1. Ajouter une méthode `canBeDeleted()` dans `AppointmentRef` 2. Lever une exception métier si la suppression est interdite 3. Créer `AppointmentRefCannotBeDeletedException` 4. Modifier `DeleteAppointmentRefService` pour utiliser cette règle

💡 **Solution** :

```
// 1. Dans AppointmentRef.java
public boolean canBeDeleted() {
    return "DRAFT".equals(this.type);
}

// 2. Exception
public class AppointmentRefCannotBeDeletedException
    extends BaseCustomException {
    public AppointmentRefCannotBeDeletedException(String type) {
        super("Cannot delete AppointmentRef with type: " + type);
    }
}

// 3. Dans DeleteAppointmentRefService
@Override
public void deleteAppointmentRef(String id) {
    AppointmentRef ref = repository.findById(id)
        .orElseThrow(() -> new AppointmentRefNotFoundException("id", id));

    if (!ref.canBeDeleted()) {  // ✅ Règle métier
        throw new AppointmentRefCannotBeDeletedException(ref.type());
    }

    repository.delete(ref);
}
```

📧 Exercice 3 : Ajouter un adaptateur sortant

🎯 **Objectif** : Ajouter un service d'envoi d'email lors de la création.

📝 **À faire** : 1. Créer `EmailService` (interface) dans `domain/common/` 2. Créer `SmtpEmailService` dans `infrastructure/adapter/email/` 3. Injecter et utiliser dans `CreateAppointmentRefService`

💡 **Solution** :

```
// 1. Port sortant (domain/common/)
public interface EmailService {
    void sendCreationNotification(String email, String entityId);
}

// 2. Adaptateur (infrastructure)
@ApplicationScoped
public class SmtpEmailService implements EmailService {
    @Override
```

```
    public void sendCreationNotification(String email, String entityId) {
        // Implémentation SMTP
        log.info("Sending email to {} for entity {}", email, entityId);
    }
}

// 3. Utilisation dans le service
public class CreateAppointmentRefService {
    private final AppointmentRefRepository repository;
    private final EmailService emailService;  // Nouveau

    @Override
    public String createAppointmentRef(CreateAppointmentRefCommand cmd) {
        // ...
        repository.save(appointmentRef);

        emailService.sendCreationNotification(
            cmd.getNotificationEmail(),
            appointmentRef.id()
        );

        return appointmentRef.id();
    }
}
```

🎉 Conclusion

Vous avez maintenant une compréhension complète de l'**Architecture Hexagonale** :

📌 Points clés à retenir

1. ✂️ **Séparation stricte** : Domain ≠ Infrastructure
2. 🔌 **Ports et Adaptateurs** : Interfaces pour tout ce qui est externe
3. 🔄 **Inversion de dépendances** : Tout pointe vers le domaine
4. ✅ **Testabilité** : Le domaine est testable sans infrastructure
5. 🚀 **Évolutivité** : Changer de framework/DB sans toucher au métier

🌟 Avantages

✅ **Code maintenable** : Séparation claire des responsabilités
✅ **Testable** : Tests unitaires du domaine sans mocks complexes
✅ **Évolutif** : Ajout facile de nouveaux adaptateurs
✅ **Indépendant** : Pas de couplage au framework
✅ **Compréhensible** : Structure claire et uniforme

🚀 Prochaines étapes

1. 💪 Pratiquer avec les exercices
2. 🔍 Lire le code du projet en détail
3. ✏️ Créer vos propres entités en suivant les patterns
4. 📚 Explorer d'autres patterns : CQRS, Event Sourcing, DDD

📚 Ressources additionnelles

- 🌐 **Blog d'Alistair Cockburn** : https://alistair.cockburn.us/hexagonal-architecture/
- 💻 **Netflix Tech Blog** : Articles sur l'architecture hexagonale à grande échelle
- 📖 **Domain-Driven Design** (Eric Evans) : Livre de référence sur le DDD

Bonne maîtrise de l'Architecture Hexagonale ! 🎯