

RadioHub - Documentation Technique

Auteurs :

- Théo Garcès

1. Introduction

Ce document présente l'architecture logicielle du projet RadioHub, un agrégateur de web-radios conçu selon les principes SOLID et utilisant des Design Patterns classiques. L'objectif est de fournir une solution modulaire, extensible et maintenable.

L'application permet :

- De gérer un catalogue de radios.
- De s'abonner à des flux (Stations).
- De diffuser des émissions avec des notifications en temps réel (simulées).
- D'ajouter de nouveaux types de contenus sans modifier le cœur du système.

2. Diagramme de Classes (UML Détailé)

Voici la structure des classes principales et leurs interactions.

```
classDiagram
    %% --- FAÇADE & SERVICES ---
    class RadioHubFacade {
        -mainCatalogue: Catalogue
        -auditeurs: Map~string, Auditeur~
        +createRadio(name, themeName)
        +subscribeAuditeur(auditeur, radio)
        +emitEmission(radio, type, content)
        +listenToRadio(auditeur, radio)
    }

    class ThemeRegistry {
        -instance: ThemeRegistry
        -themes: Map~string, IThemeStrategy~
        +getInstance()$ 
        +registerTheme(theme)
        +getTheme(name)
    }

    %% --- FACTORY ---
    class EmissionFactory {
        +createEmission(type, content, details)$ IEmission
    }

    %% --- OBSERVABLE (Radio) ---
```

```
class ISubject {
    <>
    +attach(observer)
    +detach(observer)
    +notify(data)
}

class Radio {
    -name: string
    -observers: IOobserver[]
    -theme: IThemeStrategy
    +notify(emission)
}

%% --- OBSERVER (Auditeur) ---
class IOobserver {
    <>
    +update(data)
}

class Auditeur {
    -name: string
    +update(data)
}

%% --- STRATEGY (Themes) ---
class IThemeStrategy {
    <>
    +getName()
    +formatEmission(emission)
}

class MusicTheme {
    +formatEmission(emission)
}
class SportTheme {
    +formatEmission(emission)
}

%% --- ITERATOR & COMPOSITE ---
class ICatalogueItem {
    <>
    +getName()
}

class IIIterator~T~ {
    <>
    +hasNext()
    +next()
}

class Catalogue {
    -items: ICatalogueItem[]
    +add(item)
```

```

        +createIterator()
    }

    class CatalogueIterator {
        -items: ICatalogueItem[]
        -position: number
        +hasNext()
        +next()
    }

%% --- RELATIONS ---
RadioHubFacade --> Catalogue : possède
RadioHubFacade ..> ThemeRegistry : utilise (Singleton)
RadioHubFacade ..> EmissionFactory : utilise (Creation)

Catalogue ..|> ICatalogueItem
Radio ..|> ICatalogueItem
Catalogue --> "*" ICatalogueItem : contient
Catalogue ..> CatalogueIterator : crée

Radio ..|> ISubject
Radio --> "1" IThemeStrategy : utilise
Radio o-- "*" IOobserver : notifie

Auditeur ..|> IOobserver

MusicTheme ..|> IThemeStrategy
SportTheme ..|> IThemeStrategy

```

3. Analyse Architecturale & Design Patterns

3.1. Façade (CU_9)

- **Problème :** Le système interne est complexe (gestionnaires de thèmes, itérateurs, factories). Exposer cette complexité au contrôleur Web rendrait le code difficile à maintenir.
- **Solution :** La classe **RadioHubFacade** centralise toutes les opérations. Le serveur Express n'appelle que cette classe.
- **Avantage :** Découplage total entre la couche présentation (API/Web) et la couche métier.

3.2. Observer (CU_1 & CU_2)

- **Problème :** Comment informer les auditeurs qu'une émission commence sans que la Radio ne connaisse les détails spécifiques de l'auditeur ?
- **Solution :** Pattern Pub/Sub. La **Radio** (Subject) maintient une liste d'abonnement. L'**Auditeur** (Observer) implémente une méthode **update**.
- **Avantage :** On peut ajouter des milliers d'auditeurs sans modifier la classe Radio.

3.3. Strategy & Registry (CU_5)

- **Problème :** Comment gérer différents formats d'affichage (Musique vs Sport) et permettre l'ajout de futurs thèmes (Météo, Bourse) sans violer le principe Open/Closed ?
- **Solution :** Une interface `IThemeStrategy`. Chaque thème est une classe (`MusicTheme`, `SportTheme`). Un `ThemeRegistry` (Singleton) les recense.
- **Avantage :** Pour ajouter un thème "Cinéma", il suffit de créer la classe et de l'enregistrer. Aucune modification de `Radio.ts` n'est requise.

3.4. Iterator (CU_3)

- **Problème :** Parcourir le catalogue sans exposer sa structure de stockage interne (Array, Set, Map).
- **Solution :** L'interface `IIterator` standardise le parcours (`hasNext`, `next`).
- **Avantage :** Si demain le catalogue devient un arbre complexe ou une base de données distante, le code client (qui liste les radios) restera identique.

3.5. Factory (CU_6)

- **Problème :** La création d'objets JSON pour les émissions est répétitive et dépend du type.
- **Solution :** `EmissionFactory.createEmission(type, ...)` gère l'instanciation et le formatage des données.
- **Avantage :** Centralisation de la logique de création. Si le format JSON change, on ne modifie qu'un seul fichier.

4. API & Commandes

L'application expose une API REST simplifiée sur le port 3000.

Méthode	Route	Description	Exemple Body
GET	<code>/radios</code>	Liste le catalogue complet	-
POST	<code>/subscribe</code>	Abonne un auditeur	<code>{"auditeur": "Theo", "radio": "Skyrock"}</code>
POST	<code>/unsubscribe</code>	Désabonne un auditeur	<code>{"auditeur": "Theo", "radio": "Skyrock"}</code>
POST	<code>/message</code>	Envoi un message à l'animateur (CU_7)	<code>{"auditeur": "Theo", "radio": "Skyrock", "message": "Salut"}</code>
POST	<code>/listen</code>	Simule l'écoute du flux audio (CU_8)	<code>{"auditeur": "Theo", "radio": "Skyrock"}</code>
POST	<code>/emit</code>	Diffuse une émission	<code>{"radio": "Skyrock", "type": "Musique", "content": "Hit", "details": {...}}</code>
POST	<code>/radios</code>	Crée une radio (Admin)	<code>{"name": "Fun", "theme": "Musique"}</code>

5. Réflexion Critique & Limitations

Ce qui fonctionne bien

- L'architecture est très découpée. Ajouter une fonctionnalité métier se fait souvent sans toucher au serveur Web.
- Le principe Open/Closed est respecté pour les thèmes.

Limitations actuelles

- **Pas de persistance** : Les données sont en mémoire (RAM). Au redémarrage serveur, tout est perdu.
- **Synchronisme** : Les notifications sont envoyées de manière synchrone (bloquante). Si 100 000 auditeurs sont abonnés, le serveur figera pendant la notification. Il faudrait utiliser des files d'attente (RabbitMQ/Redis) pour passer à l'échelle.
- **Gestion des erreurs** : La gestion des erreurs est basique.

Améliorations possibles

- Implémenter le pattern **Composite** plus profondément pour gérer des sous-catégories de radios (Dossier Rock -> Radios Rock).
- Ajouter une vraie persistance (SQLite ou MongoDB).

4. Réflexion Critique

4.0 Limitations actuelles

- Le système ne gère pas encore la suppression dynamique de thèmes au runtime.
- L'authentification des auditeurs est simulée (pas de login/mot de passe).