# fourtwall-challenge Home

## Purpose

The purpose of this document is to describe and explain implementation to the code / system design challenge for Fourtwall https://popshop.atlassian.net/wiki/external/Y2ZlNDViODdkYmY4NDg5OWE0MjgwNjdiMTFjMTZjYjg . Based on the solution implementation and this document the Fourtwall team will conduct a “Yay or Nay” decision to either move forward with the process for Full Stack JVM Developer. The challenge itself is testing the candidate across the board from architecture & coding through project organization.

## Core Requirements

The requirements list is not long but the way it’s written they are open ended and a candidate has multiple, spots where he can stomp on the quicksand and get drowned. One needs to decided where he wants to spend time and what will bring the most benefits long term. Is it focusing on domain and slow on start.

### List

- ☑ An internal endpoint in which they (i.e., the cinema owners) can update show times and prices for their movie catalog ☑ FT-11: REST Endpoints GOTOWE ☑ FT-10: Domain Implementation GOTOWE ☑ FT-5: Prepare Domain GOTOWE
- ☑ An endpoint in which their customers (i.e., moviegoers) can fetch movie times ☑ FT-11: REST Endpoints GOTOWE
- ☑ An endpoint in which their customers (i.e., moviegoers) can fetch details about one of their movies (e.g., name, description, release date, rating, IMDb rating, and runtime). Even though there's a limited offering, please use the OMDb APIs (detailed below) to demonstrate how to communicate across APIs. ☑ FT-9: Integrate with OMDB GOTOWE ☑ FT-13: Endpoint Returning movie details from OMDB GOTOWE
- ☑ An endpoint in which their customers (i.e., moviegoers) can leave a review rating (from 1-5 stars) about a particular movie ☑ FT-11: REST Endpoints GOTOWE ☑ FT-10: Domain Implementation GOTOWE
- ☑ And add anything else that you think will be useful for the user... ☑ FT-10: Domain Implementation GOTOWE
- ☑ Creating a persistence layer to save results for specific actions (e.g., via SQL/NoSQL/etc.) ☑ FT-8: Create Repositories GOTOWE ☑ FT-2: Prepare Infrastructure GOTOWE ☑ FT-4: Populate main tables GOTOWE ☑ FT-3: Set up entity model GOTOWE
- ☑ Present API documentation leveraging OpenAPI/Swagger 2.0 standard: ☑ FT-12: General infrastructure GOTOWE ☑ FT-11: REST Endpoints GOTOWE

## Approach, Organization - Focus

As it has been mentioned earlier one “can’t have the cake and eat the cake”, the candidate immediately needs to make the stressful decision; features over base or the other way around. The overall goal is to have a long running project with potential to expand with the business expansion to continue to bring value. At the beginning only one developer is working on the project (full stack developer) with idea that in the future the team will grow or the project will be passed to the next developer depending on the priorities.

Features make money, tech debt spends money. As the listed core requirements is the minimal viable product agreed by the stake holders to bring value at this stage. The decision has been made to focus on the list and infrastructure, everything above is a benefit.

### Organization

#### Project

Project is being run through Jira and each code changes should be linked to a Jira ticket even if it’s initially  a one liner. The usage brings structure and forces the parties to create a plan – might not be estimated. Even though at the beginning the story / task / bug definition might not be of the best quality it will create a habit in the parties and force them to think how to improve the collaboration.

- high adoption
- clarity
- GitHub Actions
- cheaper for small scale than GitLab with on pair functionality

As this is the startup phase of the project the decision has been made to abandon GitFlow and push directly to master. This is to avoid unnecessary slowdowns and avoid accepting own MRs - no other developer has been assigned to the project. At this stage it's important to have split the changes in to manageable commits. After the start up phase 2 eye principal will be a mandatory thing where a MR will need to be approved and accepted by a team.

## Architecture & Stack

### Today

At the start up of the project it's hard to predict to what size it will grow and to what direction it will shift. For business it's important to balance the cost & revenue ratio, for the tech geeks the paradise is to go wild - cloud native, reactive microservices, large machines, managed services without worrying about the costs which spin like the gas meter this days.

Today we have a core list of requirements and a promise that the project will be a long living profitable activity. We should start simple and solid, expand and turn back when needed.

#### Host

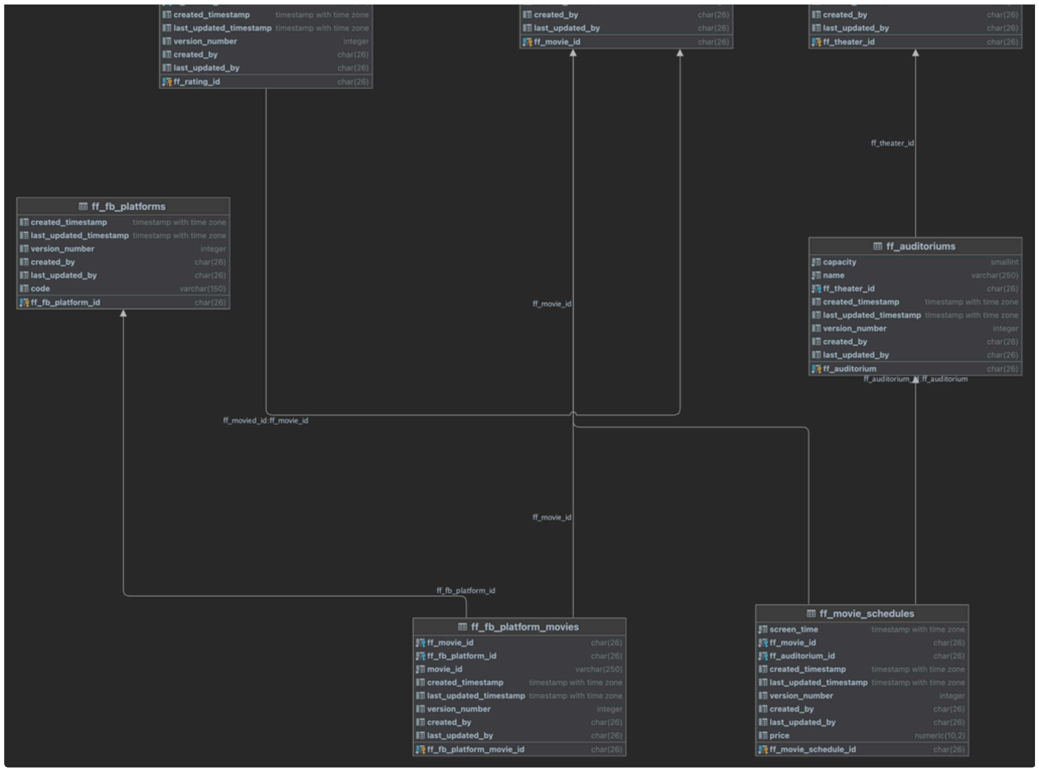
Due to a startup nature and lack of discounts for cloud providers a decision has been made to utilize the on premise machines available in the organization. Today we are not in the cloud but we want to be ready to be there – if needed.

#### Persistence

For the persistence layer PostgreSQL has been selected due to:

- high adoption
- active maintenance and growth
- flexibility
  - relational
  - ACID compliant
  - schema-less - jsonb (of reasonable size; TOAST)
  - available as managed services (RDS)
- business is often able to create reports on it's own

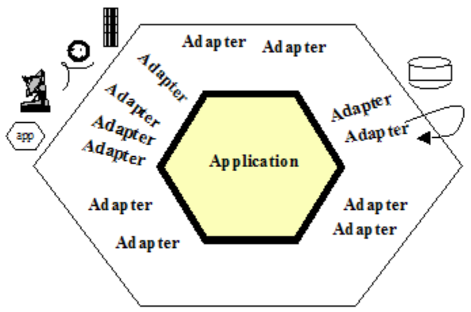
##### Schema



FF Theater Schema

## Application

The main requirement for the application layer is adaptability, ease of adding additional components and possibility to break out into smaller components (microservices or lambdas) to maintain availability and performance. As a start a single application has been created with a pseudo-hexagonal architecture. The architecture secures the domain as it would be a holy grail not allowing the outer world to pollute it, at the same time driving the outer world growth. It gives more comfort when components need to be replaced as the core logic shouldn't be affected by the change. It allows for multiple domains and easier break down to smaller components.



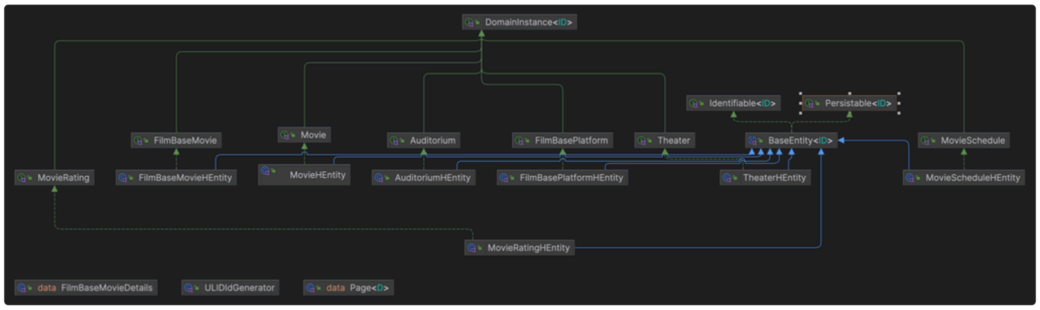
Hexagonal architecture

current  moment the structure is defined through packages with the attention to not mix concerns. From this stage it's an easy step to break down the packages into modules.

### Application Stack

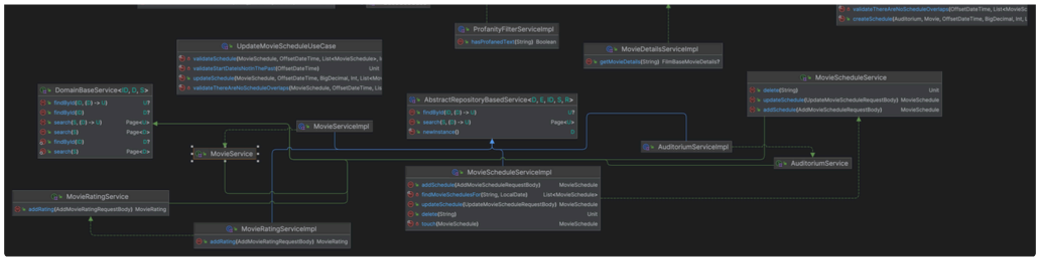
- Spring Boot 3.4
  - Spring Logging - logging capabilities
  - Spring Open API integration - exposure of OpenAPI documentation with Swagger
  - Spring Security - authentication (JWT)
  - Spring WebFlux - WebClient and non blocking streams
  - Spring AOP - needed to enrich proxies
  - Spring Data JPA - hibernate integration with Spring
  - Spring Actuator - exposes observability
- Kotlin 1.9.25 on JDK 21 - business forced decision 🙂
- Hibernate 6.4 - ORM
- HikariCP 4.0.3 - connection pool
- ulid-creator 5.2.2 - Universally Unique Lexicographically Sortable Identifier for instance identification (primary keys)
- liquibase 4.24.0 - database change management
- micrometer-prometheus-registry - exposed for monitoring
- datafaker - component allowing for easy generation of fake data
- resilience4j 2.1.0 - fault tolerant library - current usage Circuit Breaker for OMDB
- testcontainers - postgres for integration tests 1.20.4
- profanity-filter - component used to identify bad language in ratings (additional feature 🙂 )

### Domain Layer

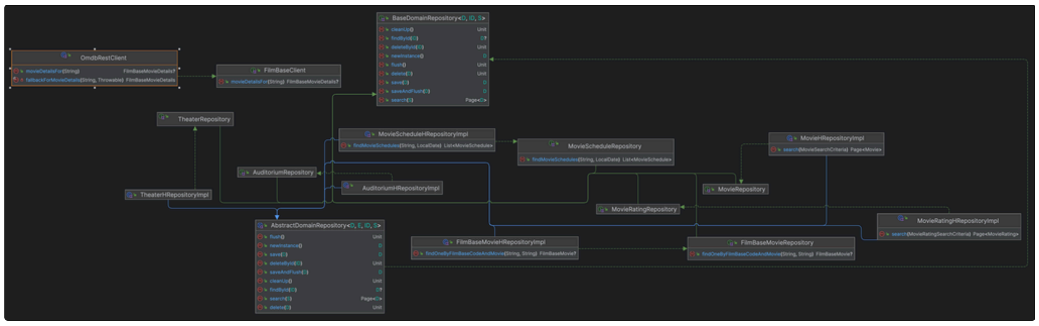


Domain Objects with Hibernate Entity Adapters

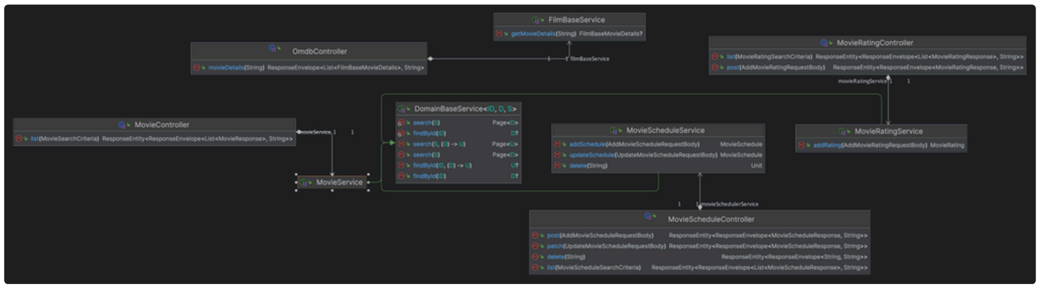
Domain Objects with Hibernate Entity Adapters (properties)


Domain Services with Application Implementations

### Outbound Ports


Outbound Ports with Adapters

### Inbound Ports


Inbound Ports with services usage

### Api

The api has been grouped into two:

- `/public/api/{versionNumber}` - public api available for any user, with the version increment to not break external applications
- `/api/` - internal api that is being secured by JWT

**Currently there's only one hardcode user with the given credentials:**

- username: `cinemaadmin`
- password: `password`

configuration to the OMDB API url and key → **Mea, culpa, I know that you have written to not include the key in the repository but in the it has been done - just a human here.**

#### Testing

The testing emphasizes has been put on having fully coverage of unit tests on the domain logic, put emphasis on testing the adapters with integration tests based on test containers and test accounts for endpoints. Application layer should be tested with fake implementation to verify orchestration behaviours.

#### Summary

Set up of the application structure took more time than anticipated, some of the challenges have been related to knowledge gaps in Kotlin as the candidate comes from pure Java world. But once those have been wiped off, the future developers land with a base where building new functionality fallbacks to extending and composing from existing components. At the current moment the domain model looks anemic but with the growth of requirements that will change and more core logic will be available there.

### Business

All of the core requirements have been implemented at the best of will and understanding of them. With them additional business rules have been implemented:

- User ratings with comments and rating user
- Profanity Filtering for user ratings - validation preventing adding comments that include bad language
- Circuit breaker for the OMDB integration - more of a technical aspect but has impact on the business
- The persistence and domain model is ready to support multiple cinemas and auditoriums in those cinemas when the business successfully grows 🙂

## Tomorrow - when business blends with technology

### Business

The current state of business logic is just a start and one can imagine new requirements:

1. expand further on the concept of multiple cinemas with multiple auditoriums
   a. support more internal users
2. besides the schedules the cinema could reserve and eventually sale tickets for events
   a. Order business process
   b. Payment gateway integration
3. notifications - movie lovers would be interested in box offices and old movie surprises
4. with the grow of the business the cinema could move into the film base direction as a known brand
5. become a post box office video service
6. expose movie trailers
7. and more !

to be made. The scale will change, it will be harder to maintain availability and performance and the architecture will need to be broken down to scale up with the cost of deployment complexity and component dependency.

#### How the above business changes would influence architecture ?

1. Proper user details implementation allowing authentication & authorization with potential to SSO (hype)
    a. Redundancy the application would need to be scaled
2. Inclusion of business process engine, esb integration
3. SMTP provider integrations, queuing  of messages (RabbitMQ)
4. Movie lowers search movies on different criteria which are relational and indicate some preferences for suggesting next movie for example - Neo4j would be helpful here
5. Becoming a VOD service includes streaming and here a switch to reactive programing could happen
6. Storage changes - cheap efficient storage to host posters and trailers (S3)

#### Must have changes

- Proper CI / CD - today deployment is based on manual docker image builds which is not sustainable.
- Break down the project into multiple modules - even though today the boundaries are maintained at somepoint when the project grows someone will have the temptation to cross them. Having multi-module project with correctly driven dependencies will avoid it
- Proper user handling, today it’s a single hard coded user
- Caching, for example the OMDB results could be cached to not need to hit a restricted endpoint with low availability. The data there is not critical

#### Potential issues with the decisions currently made

- Transactional database
    - the usage of ACID transactional database creates the side-effect of transaction. If one starts do depend on it and decides to lean towards non transactional databases like Mongo there will be changes in flow.
    - When splitting the application into smaller services one needs to think about distributed commits
        - 2 phase
        - Saga pattern
- After working with hibernate through interfaces one needs to be aware of LazyLoadingException :)

## How to run it

The application has been dockerized, the main directory at GitHub is holding the docker-compose.yaml which one just needs to docker-compose up.

Credentials for the authentication endpoint:

- username: `cinemaadmin`
- password: `password`

fourtwall-challenge