Poptato – A Micro-Service Solution

By Idan Asraf (313292781) & Natanel Orenstein (313247512)

Introduction

Poptato is a dedicated, micro-service based, solution for movie theaters to manage and expose their information about movies, shows and orders.

The project was developed for educational, and is not generally intended for commercial use. It is release as open-source under the MIT license, in a dedicated GitHub organization.

We've built the project using NodeJs, leveraging existing high-quality open-source libraries.

The different components in the project use RethinkDB for persistency, and have Docker support for running & testing locally.

The Micro-Services

Discovery Service

This micro-service acts as an abstraction layer, simplifying access to the functional components (micro-services) in the system; by hiding the micro-service architecture implementation.

It acts as a proxy, forwarding requests coming through it to the correct micro-service to handle the request. This is how the proxy is mapping the requests:

HTTP Method	Discovery Request Path	Target Micro-Service
Any (GET, POST, …)	/movies/*	Movie Service
Any (GET, POST, …)	/celebs/*	Movie Service
Any (GET, POST, …)	/oscars/*	Movie Service
Any (GET, POST, …)	/cinemas/*	Show Service
Any (GET, POST, …)	/theaters/*	Show Service
Any (GET, POST, …)	/shows/*	Show Service
Any (GET, POST, …)	/orders/*	Order Service
Any (GET, POST, …)	/customers/*	Order Service

Web-Client Service

This micro-service acts as a simple static-files server, serving the web-client's assets to a browser user.

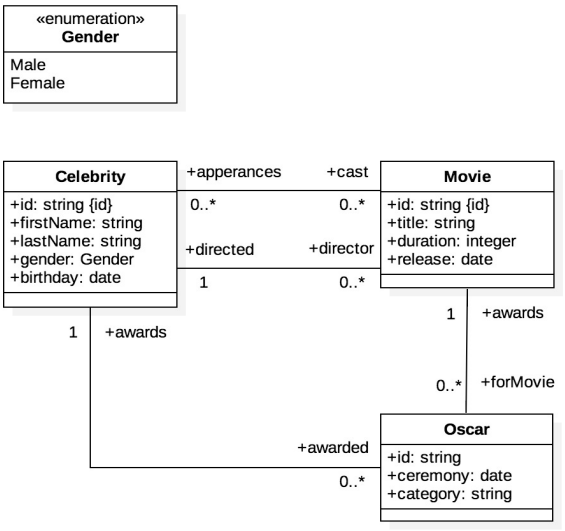
Movie Service

Manages information about movies.

In addition to basic information about the movie itself, the system also manages complex associations of the movie.

This includes cast, director & Oscar awards management.

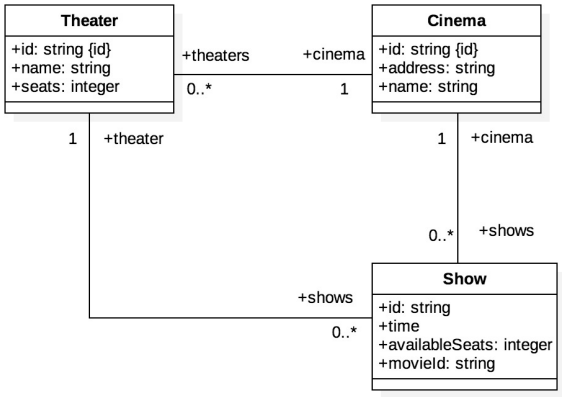
All entities & relation are exposed via secured REST API endpoints, allowing creation and querying of information.



Show Service

Manages information about cinemas, theaters & shows.

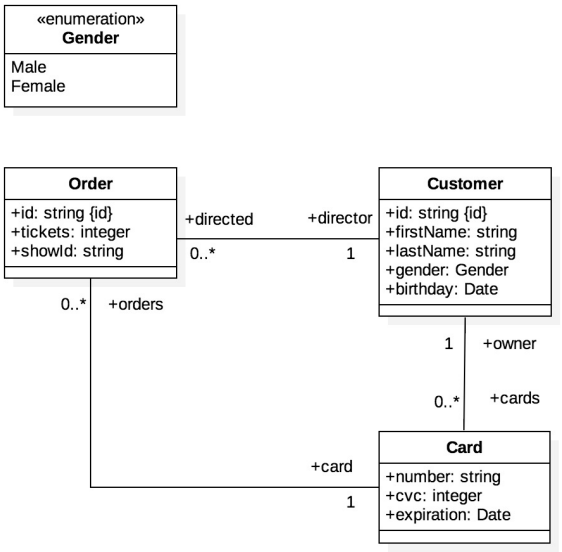
All entities & relation are exposed via secured REST API endpoints, allowing creation and querying of information.



Order Service

Manages information about customers, orders and payment methods (credit cards).

All entities & relation are exposed via secured REST API endpoints, allowing creation and querying of information.

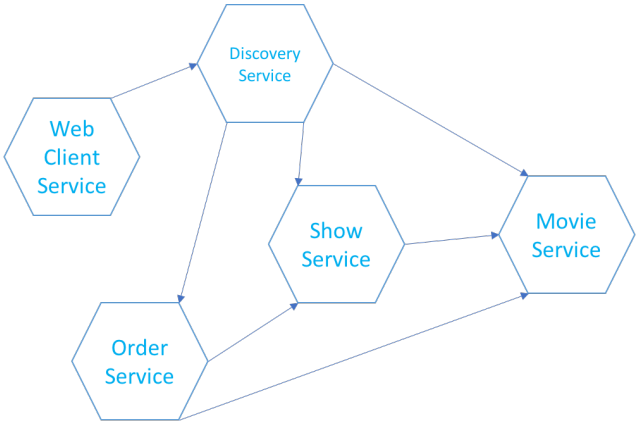


Micro-Service Relations

The micro-services in our application interact with each other, in order to accomplish complex tasks involving functionality from multiple micro-services. This is a part of the micro-service architecture methodology.

The micro-services communicate with the outside world, and with each other using HTTP requests, using the common method of REST API.

Service-to-service communication is secured via HTTPS and support BASIC authentication.



Technology Stack

Services (Server-Side)

For the server-side we've used the following technologies:

- ✓ **NodeJs** open-source, cross-platform JavaScript run-time environment.
- ✓ **ExpressJs** Fast, un-opinionated, minimalist web framework for Node.js.

Database

All our micro-services are using instances of **RethinkDB** which is a NoSQL, open-source database for real-time applications.

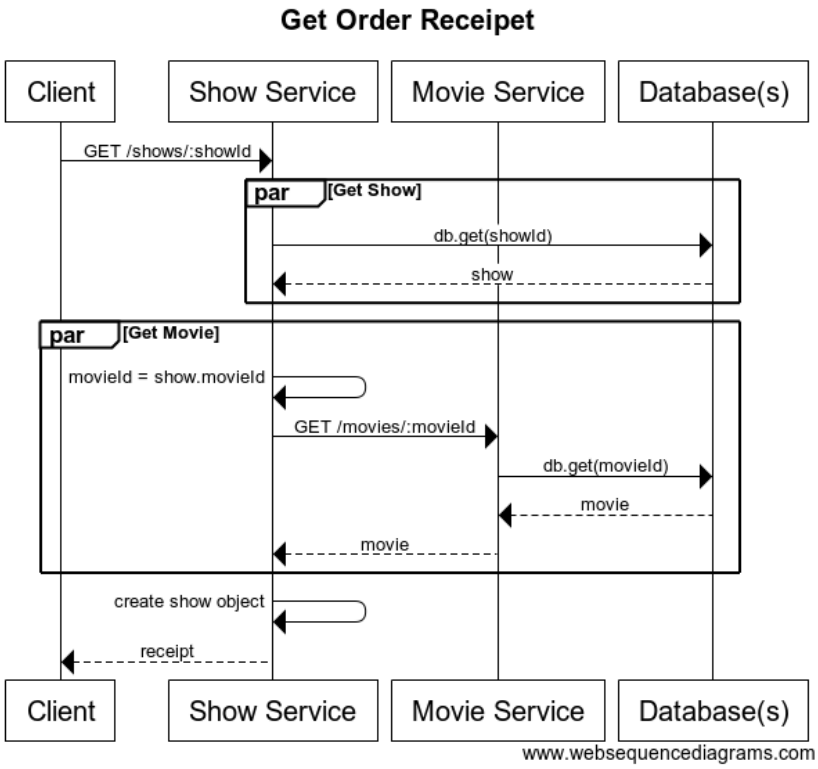
Web Client

Our web client is written in HTML5, JavaScript ES2017 & CSS3.
We're using **Polymer** which is a JavaScript library that helps you create custom reusable HTML elements, and use them to build performant, maintainable apps.

Micro-Service Interaction Examples

Get Show Information

When an API consumer sends a request to get show information, the Show Service requests the movie's information from Movie Service, assembles the response and send it to the consumer.

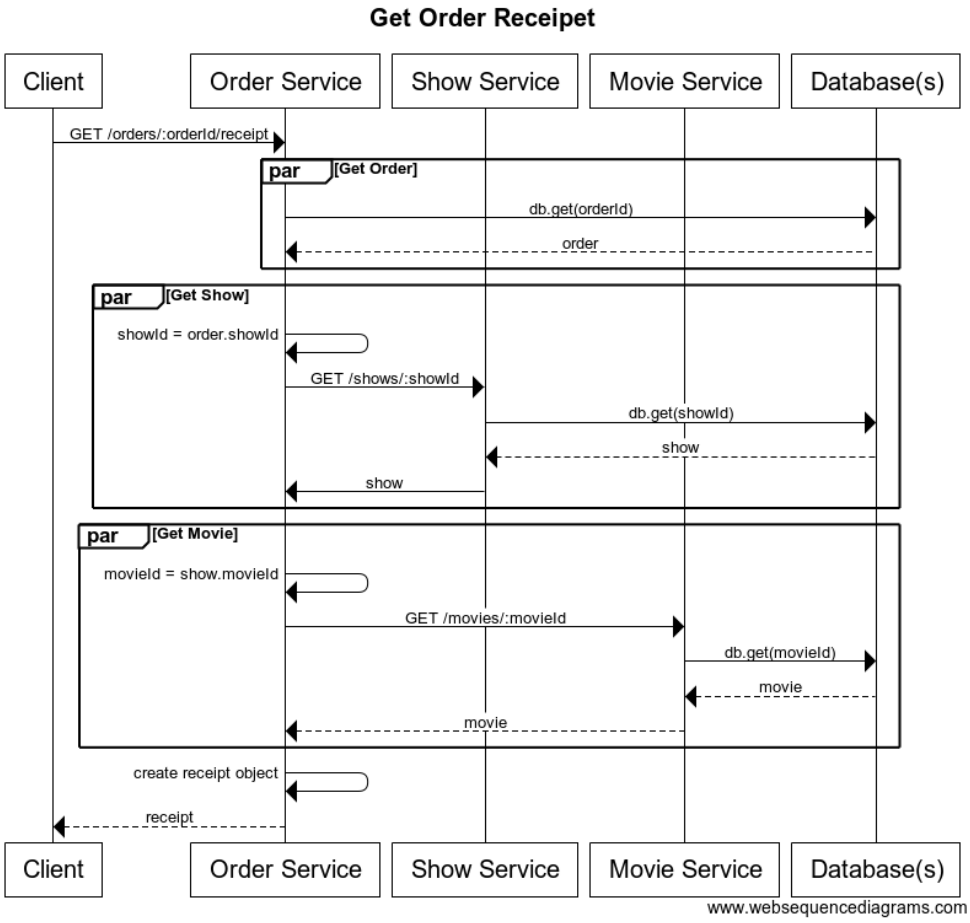


Get Order Receipt

When an API client requests a receipt for an order, the Order Service requests the following information from other services:

1. Requests the show information from the Show Service, based on the id in order.
2. Requests the movie information from the Movie Service based on the id in show.

Then, order service assembles the receipt object before sending the response to the client.



Testing

To test and verify our solution we've used the common "testing pyramid" approach.

Each component (micro-service) in our solution have both unit-tests & API-tests.
In addition, we have an integration test suite, which validate the integration between the different components and verify the main flows in the solution.

Non-functional testing, focusing on performance testing, is being measured and handled by Docker.

Tool Stack

The testing technology stack we've chose consists of the following tools:

- ✓ **Mohca** as test runner for all functional tests.
- ✓ **Chai** as assertion library, using BDD testing style.
- ✓ **Sinon** as mock library.
- ✓ **Supertest** for HTTP mocking & REST API testing.
- ✓ **Docker** for running component locally & non-functional testing.
- ✓ **Docker-Compose** for running components together & integration testing.

Difficulties

During the development of our micro-service based solution we've encountered two major issues, both related to running locally and testing the solution.

Running Full Solution Locally

Our solution consists of multiple stand-alone components, each being an HTTP server. Running the services together and making them play nicely while keeping the environment clean has proved to be difficult.

Our solution to this problem is using Docker for wrapping each service in a container for running the application locally on an isolated environment (container). We also used Docker-Compose to create container networks for combining each service with its own database container, and creating a network of all the services.

Synchronizing Database(s) Population

Populating the database(s) with mock data is being done asynchrony, and we have no meaning of synchronizing between them before running integration tests.

Our solution to this problem, after few hours of research and trying was to omit a signal from each process and awaiting all signals to arrive before executing the integration tests. This was not easy has each service is running in its own Linux container.