# 1. Introduction

This document presents details architecture and design of a microservice, which will be used by multiple applications for login and authentication functionality.

It should contain sequence diagram and use cases accordingly. The microservice auth/oath 2 or any other solution should be stateless and should be used by many applications.

# 2. Core Architecture & Design

## 2.1 Key Architecture Principles & Guidelines

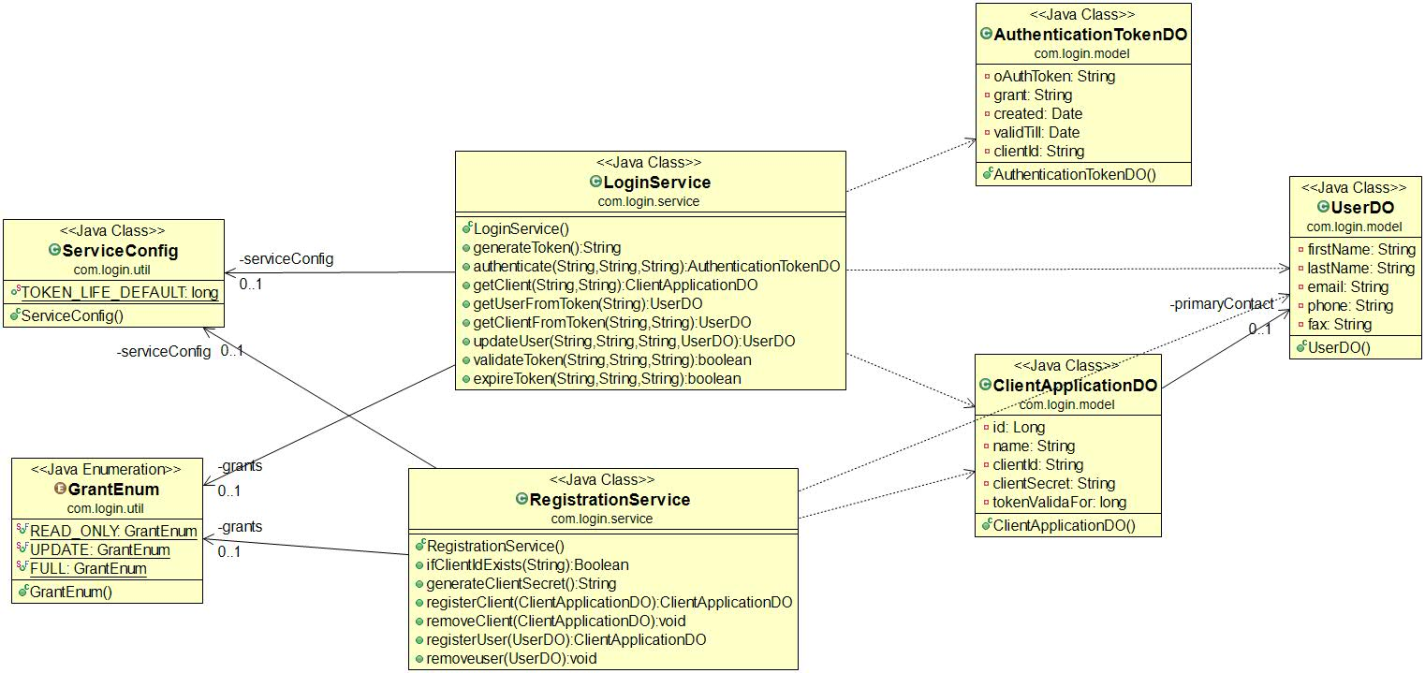
|  |  |
| --- | --- |
| SL# | Description |
| 1 | The service will be built following microservice architecture style. It will be a self-contained service without any dependency (run-time or design-time) on another microservice(s). |
| 2 | The service will support REST APIs for oAuth 2.0 styled authentication. |
| 3 | The service will support REST APIs for registration of the applications. |
| 4 | The service will use implicit authorization grant. This will enable consumer applications to receive the oAuth token directly. This will reduce number of service calls and hence will help to achieve better performance on both sides i.e. LoginService as well as consumer applications. |
| 5 | The initial version of the service will use an in-memory database for persistence of various information entities. |

## 2.2 Service Contracts / Interfaces

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function | HTTP Method | REST Endpoint URL | Input | Remarks |
| Login | POST (recommended)  and  GET | /login/authorize | Input  client\_id, client\_secret, redirect\_url  Output  Grant object JSON with following attributes-  access\_token,  grant\_level | LoginService will validate client id and secret.  If success I.e.e registered application, display login form where user can enter her id, password and choose grant level she wants to provide to the client app.  LoginService will authenticate user w.r.t entered credentials.  Once authenticate, the service will generate oAuth2 token and redirect (either to url give in input param, or if absent default redirect url for the client app.  LoginService will use following Grant Levels-  1. READ\_ONLY (user is granting permission to consumer app to get her details in read only mode)  2. UPDATE (consumer app will be able to update user details by using update method)  3. FULL (consumer app is given full access to user profile) |
| Logout | POST (recommended)  GET | /login/invalidate | Input  client\_id, client\_secret,  access\_token | Consumer applications should call this when logged in user uses logout option on the respective app.  The service will validate if the token was generated for the client identified by provided details, and invalidate the token |
| Registration of consumer/ client applications | POST | /login/register/client | Input  ClientApplication object JSON with following attributes-  application\_name,  description,  client\_id,  primary\_contact\_user,  effective\_from\_date,  effective\_to\_date,  grant\_levels  Output  ClientApplication object JSON with following attributes filled up-  application\_id,  client\_id,  client\_secret | This method will be used for registration of consumer appications, which can use LoginService for their user authentication  The service will validate the client-id to be unique (mustnot be used by another application). It will generate an unique client\_secret.  The consumer application must use the client id and seceret thus generated during registration duirng successive authentication of users.  primary\_contact\_user – this will be user object json |
| Get User Details | POST (recommendd)  or  GET | /login/users/{token} | Input  access\_token  client\_id  client\_secret  Output  User (JSON object) | Validate the client\_id, secrete and oAuth2 token  Also validate the GRANT\_LEVEL provided by the user to this consumer application (client\_id)  If found valid, return user details as JSON object in response |
| Update User Details | POST | /login/users | Input  access\_token  client\_id  client\_secret  User | Validate the client\_id, secrete and oAuth2 token  Also validate the GRANT\_LEVEL provided by the user to this consumer application (client\_id)  If ok, update user details |
| Registration of users | POST | /login/register/user | Input  User object JSON  Output  User object JSON | This method will be used for registration of users |

## 2.3 Class diagrams for core components

The class diagrams provided below shows the key classes for the proposed LoginService.



Apart from the methods corresponding to REST APIs listed in the section above, following special methods are included in the design-

1. LoginService.generateToken() - this will generate an unique oAuth2.0 token id. It is recommended to use following format for the token: <servicename><10 digit random number><version><expirationtime>

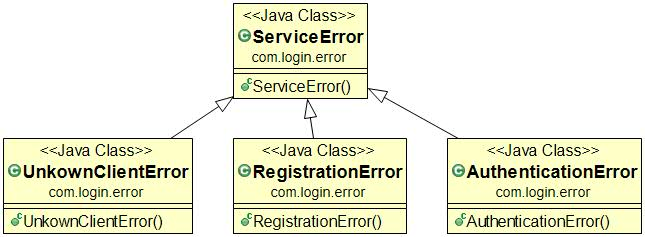
E.g. login2453627891v1.01493821061384

2. LoginService.validateToken() - this will be invoked by any potential consumer or resource server to validate if the token is still valid for the logged in user. The application should manage fine-grained authentication based on coarse-grained access granted by the user (to the app during login).

3. LoginService.expireToken() - this will expire the token.

4. RegistrationService.ifClientIdExists() - This method can be used during client application registration process to verify that the client id is unique and does not already exists in LoginService

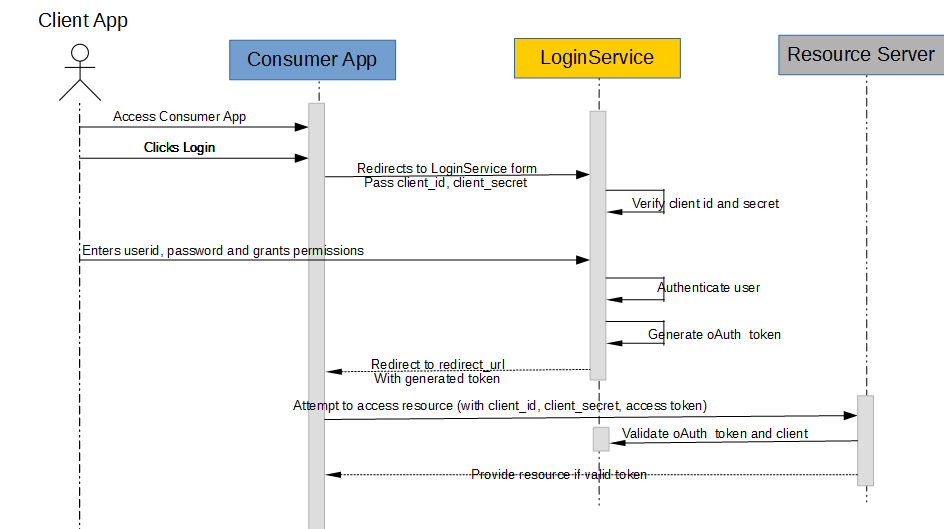
The service must return appropriate error responses to the consumer applications. The error classes for the service are shown below.



# 3. Details Design

## 3.1 Sequence Diagram for Scenarios

The following UML sequence diagram illustrates the key sequence of operations of how a consumer application will invoke LoginService for authenticating an user.



## 3.2 Technology Recommendations

This sections lists technologies recommended for implementation of the microservice. These are chosen based on following principles -

1. Implementation based on open-source free Java components

2. No cloud platform specific components, so that it can be deployed on all major platforms without lock-in

3. Easy containerization (Docker considerations detailed in next section)

4. Serverless architecture

5. No dependency on cloud-platform specific components (e.g. AWS database) Rapid scalability (scaling up and down using native features across major cloud platforms)

The technologies recommended are -

1. Spring Boot – the microservice can be implemented using latets version of Spring boot (fr both web forms and REST APIs)

2. Spring ORM to abstract database CRUD operations

3. MySQL (primary choice), or PostgreSQL (secondary, as many cloud platforms like pivotal has native support). In either case, the database should be hosted separately and bind to Spring boot based microservice, so that both layers can be scaled/descaled independently.

4. Use of gradle as preferred build mechanism , as it will enable achieving flexible CI/CD chain in future

## 3.2 Containerization Recommendations

This sections lists some considerations for containerization of the LoginService with docker.

Running the microservice in complete isolation bundled as docker containers can enable high flexibility and efficiency of operations.

Following are high steps to follow to create Docker container basedon the Spring implementation as described in previous section-

1. Setup a dockerized database instance (MySQL or PostGreSQL as intended), there are images available from docker store

2. Dockerize the spring based microservice. The guide at <https://spring.io/guides/gs/spring-boot-docker/> should provide starting point-

2a) Start with a standard image which already includes Java 8 installed

2b) Define volumes as necessary e.g. /tmp

2c) Add deployable jar file e.g. LoginService.jar

2d) Declare ports open on the container; stanard 8080 can be used for both web forms and REST APIs

3. Attach/link docker container for database to the microserice container.

4. The database connection url must be configurable in the Spring Boot microservice. The IP of the vm can be identified by “docker ps” command and used in the configuration param for microservice spring

6. For scaling/descaling, any docker container management orchestrator can be used. Kubernetes is the most popular and recommended choice.