Architecture Documentation

The Recruiting Application

Name	Email
Adrian Gortzak	**gortzak@kth.se**
Albin Friedner	**friedner@kth.se**
Alexander Nikalayeu	**nikal@kth.se**

1. Introduction

In this document the architecture for the recruit system we have developed is explained. The document will describe the features and properties as well as the decisions behind them. We also explain considerations we have made before the decision of a solution. Non-functional requirements and possible unsolved issues is also described.

2. Functionality View

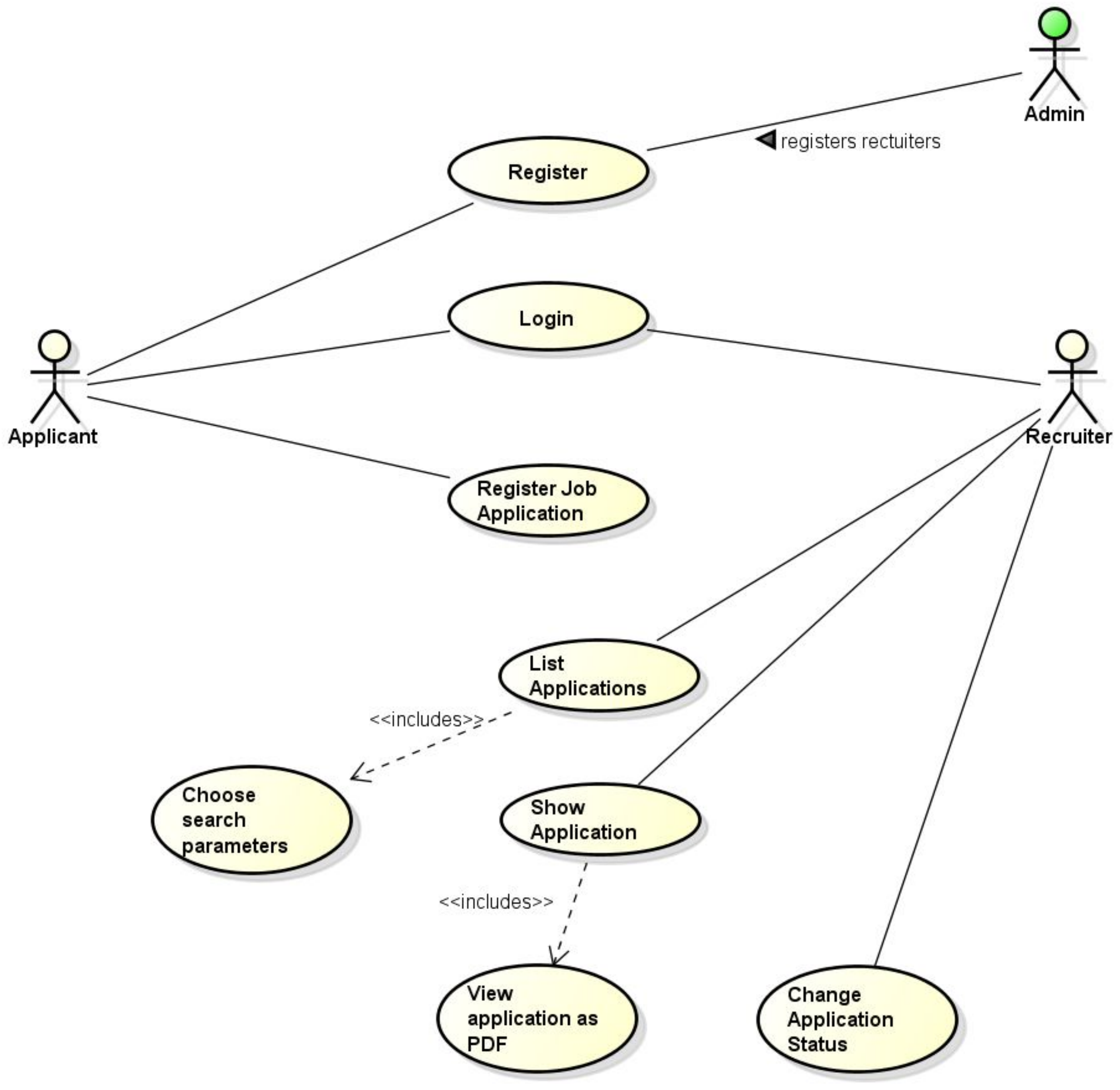


Figure 2.1 Use cases

This use case diagram shows that there are several actors and all they have different functionality. This system has:

- Applicants - they can register themselves, they can login to the system and register an job application by filling a form
- Recruiters - they can login to the system, list job applications made by applicants, they can perform search of job applications with filtering, they can look at an application and change application status, for example from "PENDING" to "ACCEPTED".
- Admins - those who can add new recruiters to the system

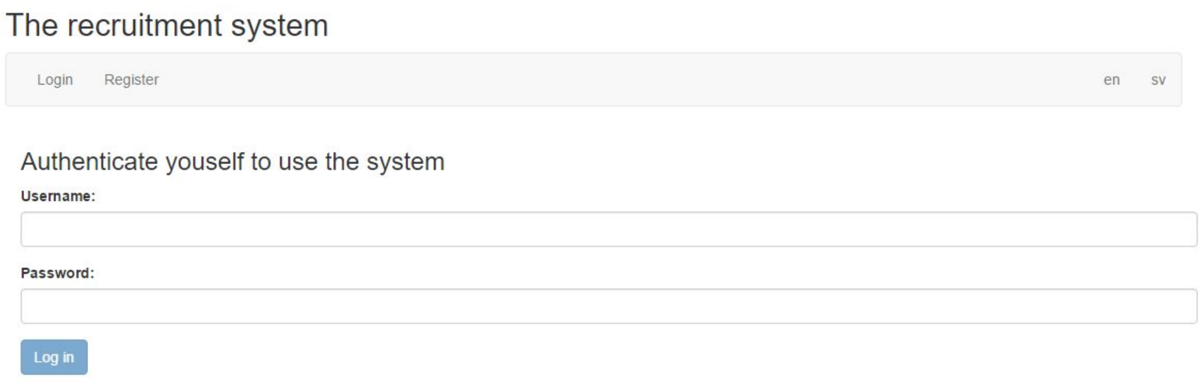
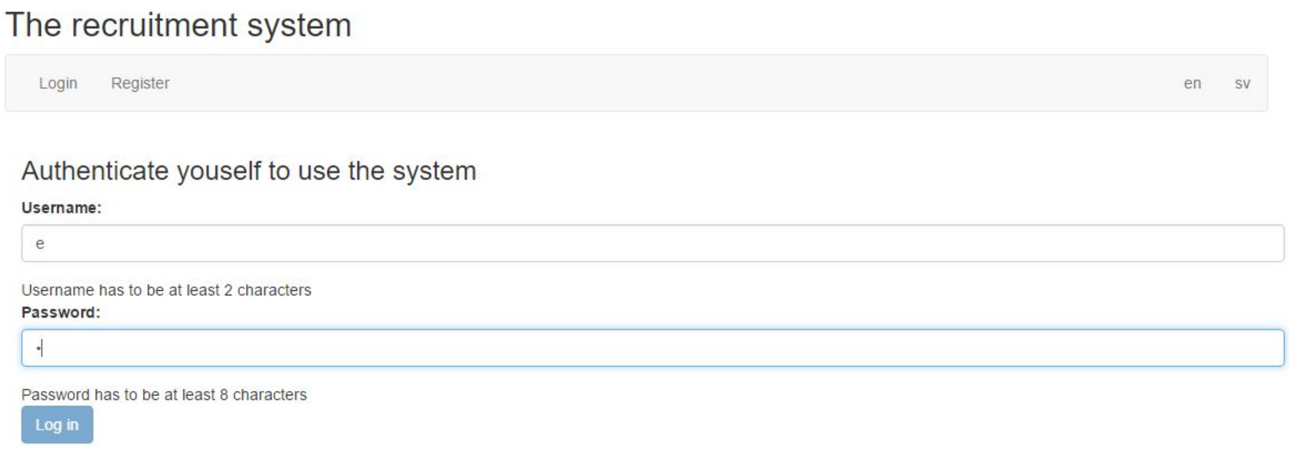


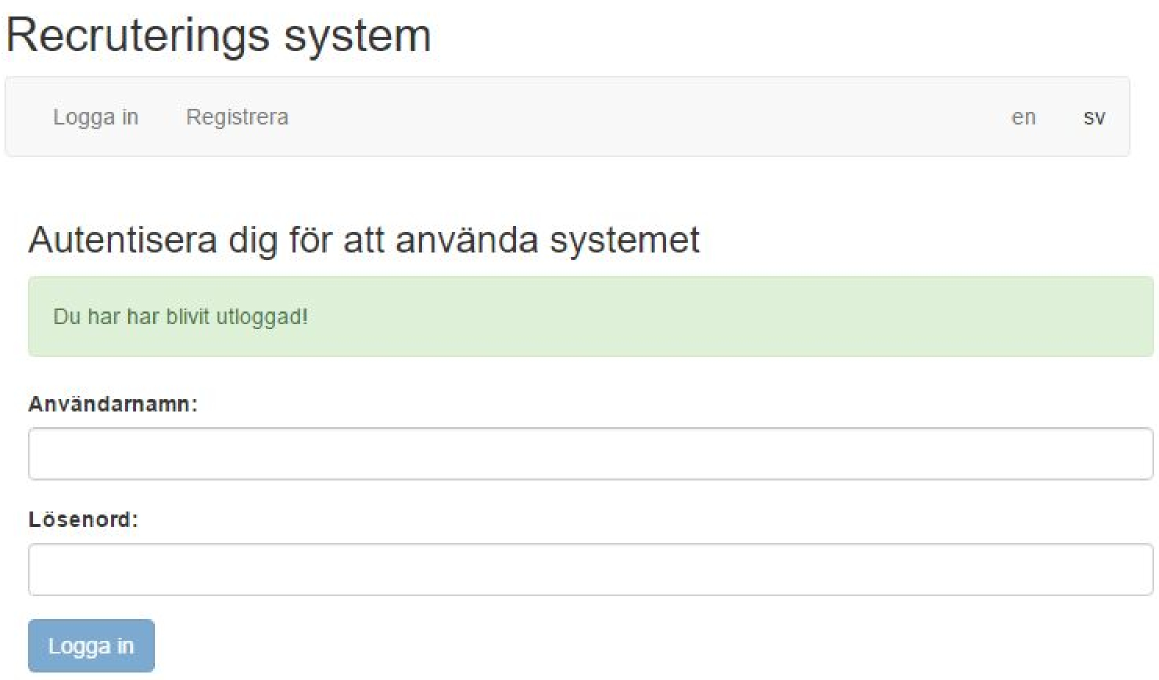
Figure 2.2. Login screen



Username has to be at least 2 characters

Password has to be at least 8 characters

Figure 2.3. Login screen when users entered data does not pass client side validation



Recruterings system

Logga in Registrera en sv

Autentisera dig för att använda systemet

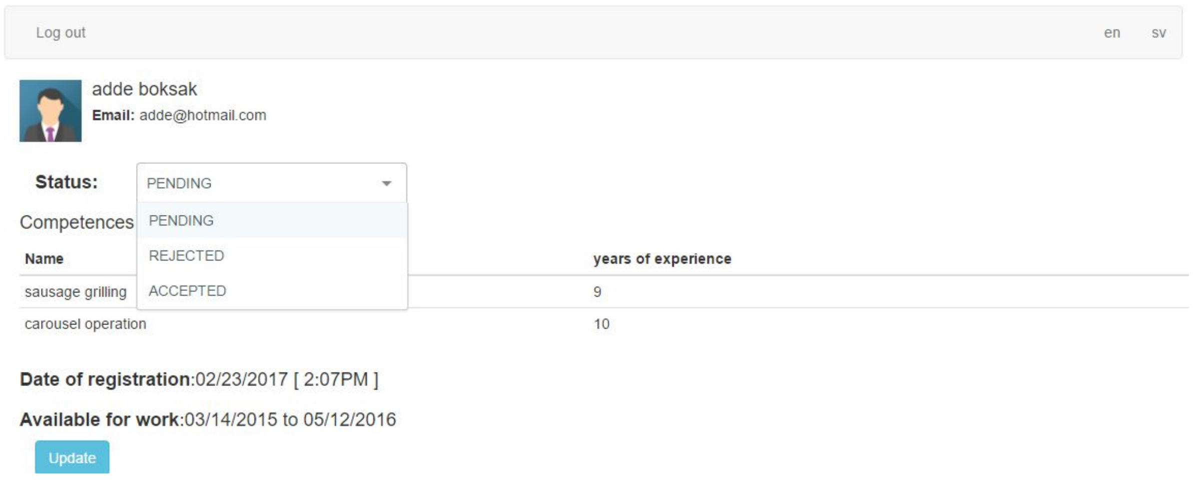
Du har har blivit utloggad!

Användarnamn:

Lösenord:

Logga in

Figure 2.4. Shows login screen after logout and also shows that UI language can be switched



Log out en sv

adde boksak
Email: adde@hotmail.com

Status: PENDING

PENDING
REJECTED
ACCEPTED

Competences

Name	years of experience
sausage grilling	9
carousel operation	10

Date of registration:02/23/2017 [2:07PM]

Available for work:03/14/2015 to 05/12/2016

Update

Figure 2.5. Shows job application from Recruites view

3. Design View

Architecture choice

This application is implemented as Microservices distributed-system. Microservices architecture means separately deployed, independent units and each unit(microservice) has its own objective[1].

Reasons for choosing Microservices architecture pattern are:

- High cohesion and low coupling - all parts of the system are maximum decoupled, every microservice has it own objective and can exist on their own [2]
- High scalability and ease of deployment - since all parts of the system are decoupled each microservice can be deployed in any numbers if needed which makes better use of hardware resources than monotonic application [2]
- Easy to maintain and continue development - each team can independently work on their particular microseviceservice as long as everyone is following agreed public API [2]
- Good unit testability - decoupled parts are simple to test since every test can be targeted for a specific code without any dependencies on other services [2].

Topology choice

There are several topologies of Microservices. In this case we implement API-REST-based topology. API-REST-based topology means that clients request goes through public API and API is talking to fine-grained independent Microservices using REST-based interface. [3] One of the alternative could be a fat client that would have direct access to all Microservices. Since this system intended to be used by different client platforms API-REST based topology suites better.

Microservices in the system

(Recommended to look at deployment diagram while reading this)

Edge service

Edge service is gateway for this system. That is were all client request goes through. Reasons to use a gateway and not let users get to microservices directly are to load balance requests, to use gateway for security (not in this system), client has to know only one address and one port - not all services, structure of the system is not revealed. The system is using Spring Cloud Netflix projects to implement this. Netflix Zuul as gateway since it is a reliable easy to use gateway library that also implements load balancer and circuit breaker design pattern so we do not have to do it manually. [5]

Edge service also the service that provides web-client to a web browser users. Explanation can be found in Edge service as web-client provider section.

Configuration Service

Configuration Service is one of the core components of this system. This service provides configuration for all other services and keys for all shared resources. Reason to use Configuration service is that this system uses a database and in this implementation only one physical DB. Every DB have a location and credentials and so on. If DB would be moved - then we would have to change information on all services and also restart all of them. To prevent that configuration service will provide all configuration and services will ask it for configuration information. Also, services will be able to update their configuration on run-time. Also, no need to rely on every service to keep sensitive information safe - only one service has to be secured to hold sensitive information save.

Configuration service also is secured with basic security with so sensitive information can be given only to authorized services.[5]

Eureka Discovery Service

Eureka is an discovery service developed by Netflix and used by services to find each other. The idea is that to make services independent from ip-addresses and ports and give them all a name.

Problem is that when new service is added or some service got scaled(deployed on several nodes), all services has to get some kind of reference to be able to use newly deployed service. Since this is a distributed system - ip-addresses and ports are the references. But it would be a lot of work, if not impossible, to keep track of all services and their addresses and give list of services to other services manually.

That is why when a new service is being deployed it will register itself on discovery service under a certain name. Then all services inside the system will be able to call this service by name, not ip-address and port. If several instances of same service is up them load balancer can easily find one that is best suited - less load and not down - to perform the task. [5]

Redis Service

Redis will be used as message broker. Idea is to use Redis in future stage for HTTP POST and PUT requests - so that no "writing" request will be lost due to a down service.[5] This will be implemented in the future.

Authentication Service

Authentication service provides authentication and also serves as JWT(Json Web Token) provider. This service will use registration-service to get user credentials, perform check, create and return JWT if proper credentials are given.[6] Also this service has an embedded database that contains credentials for other services so other services in the system can get special authorization tokens. More on security can be read in [security section](#).

Registration Service

Registration service will perform registration of new users. This service has a REST API that accepts HTTP POST requests from a form on the client side. The form input is validated by the service and persisted in a database. The service is also going to assign the role 'Applicant' to each new user that registers. Later on Redis will be configured. Service is implemented with Spring Boot and Spring Cloud.

Also registration service is responsible for providing information about registered users to other services. Endpoints that provides information about users are not routed by the gateway and also open only for authorized clients inside the system network.

The structure of the register services can be seen in picture 3.1

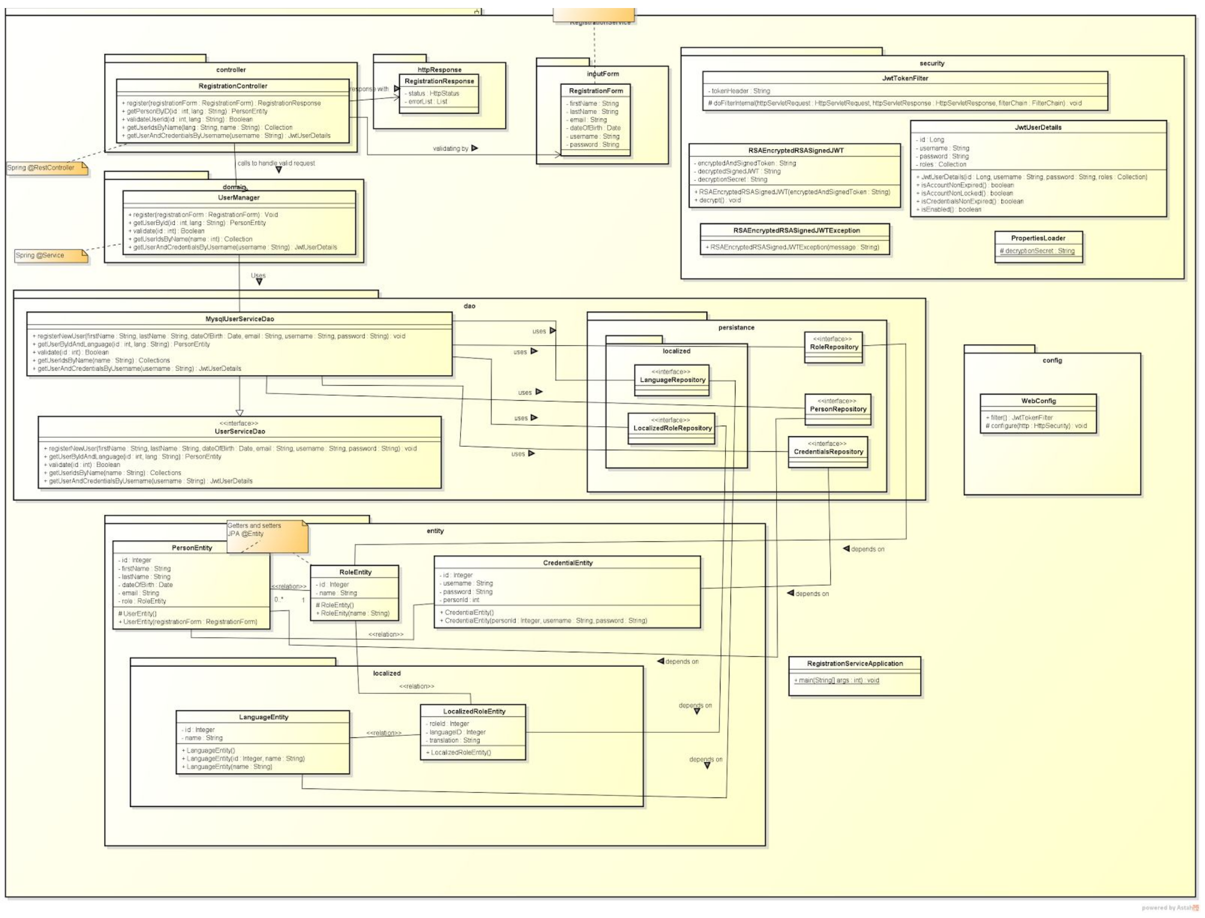


Figure 3.1. Register service architecture

JobApplication Service

Job application service will handle all interactions with the job applications. It has a RESTapi and is used to create new applications, updating application statuses by a recruiter after being looked over, retrieving a single application or a list of applications in a more page-like form, filtering applications by parameters and storing everything consistently. The user information is completely detached for the application service to remove redundant code. Instead the user service is called with the user identifier at the retrieval of job applications. The service is implemented with Spring Boot and Spring Cloud.

The structure of the Job application services can be seen in picture 3.2

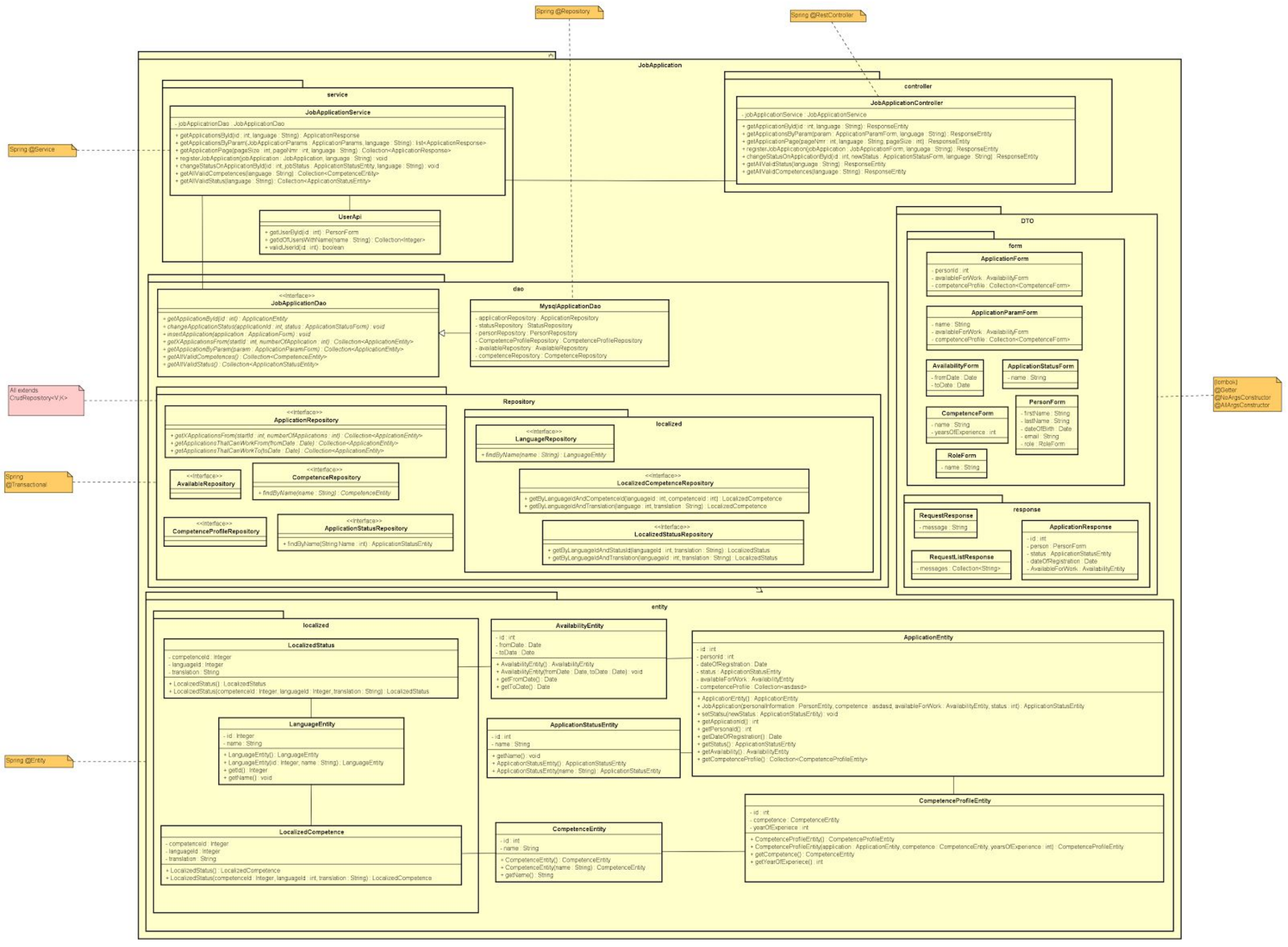


Figure 3.2. Job application service architecture

Design choices

Client-side load balancing

This system is using client-side balancing. That means that all client services in the system (services that contact other services) are keeping track of other service instances in the system. One alternative would be a centralized load-balancing where one service is a load balancer and other services would go through it to get to other services. That could create bottlenecks. Client-side load balancing solves that problem. [4]

This system is using Ribbon as load balancer. It is also used by Zuul Gateway by default.

Redis for POST-requests

This system is using Redis message broker for any "write"(HTTP POST, PUT) requests. That is used to make sure that no requests that are changing state are lost due to a down service. This feature is intended to be implemented in the future.

Edge service as web-client provider

In this system Edge service provides web-client to clients that makes HTTP GET request to default “/” endpoint. That means that there is no web server that provides HTML, CSS, JS and so on - the Edge service does it.

The main reason is that all requests made by web-client will be handled by an Edge service. This system is build to be scalable - several Edge nodes could be running. If there would be a web-service that would provides web-clients then there would be a problem of defining which edge-service should client call. Since services can be dynamically power up and do down no hardcoded values could be in the web-client. That means that a load balancer would have to be put up in front to handle all the request and forward to edge service which would forward to a service inside the system. This creates need for more nodes, more complexity and also creates a single point of failure - the first load balancer.

The way system works right now is that edge service returns web-client with reference to it self so web-client knows which edge service to request. If several edge-services are running then web-clients will automatically request different edge-services and load will be balanced.

Alternative could be to use DNS to handle situation instead of single front load balancer but it is still adding complexity to the system without any bigger advantages.

4. Security View

Security issues considered

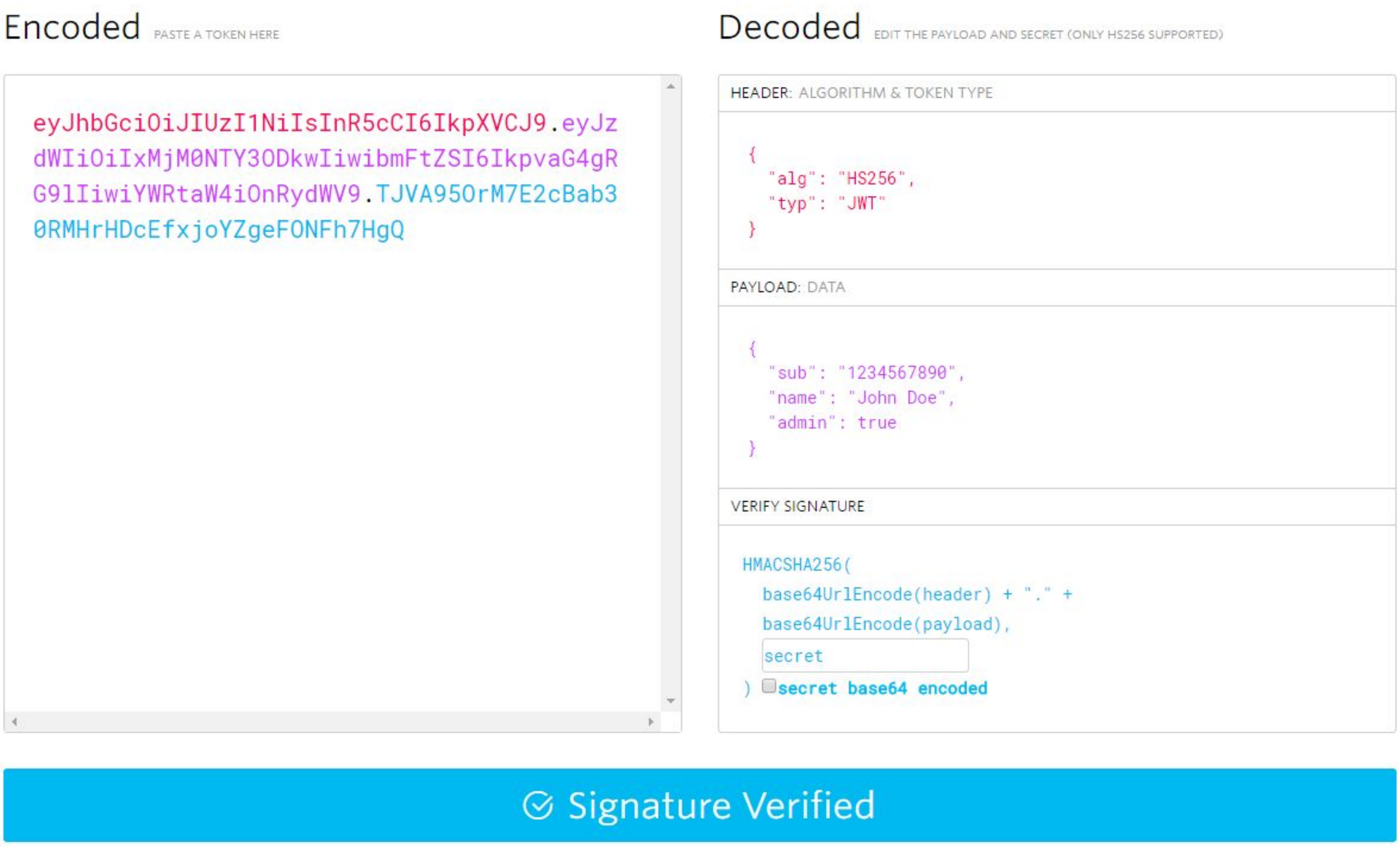
- Authentication and authorization with JWT
- Authorization on each service
- Accessing services without gateway
- Encrypting all client-traffic
- Access to config files
- Access to credentials files

Authentication on each service, JWT

Since the system is decoupled in independent services we have to handle security on each service to some extend. Also since we have REST services we want to have stateless authentication to keep having stateless system. There are several ways but most popular are OAuth2 and/or JWT.

OAuth has different flows.[8] Mostly problem for us is that OAuth2 usually stores tokens and other information in either its own database or in memory. User will authenticate themselves, get at token, use that token to request resource service, the resource service will ask authentication service if token is valid and can also return some more information about the user if required. This is a good secure way, but problem is that authentication service becomes single point of failure - even logged in users will be trouble - and also it kind of creates state in stateless system.

Maybe being authentication service it is not that bad to be single point of failure since if some part of security system is down, then something is really wrong and then system should not continue working, but we wanted to create something less fragile but still secure.


Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz
dWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gR
G9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab3
0RMHrHDcEfxjoYZgeFONFh7HgQ
```

Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) ☐secret base64 encoded
```

Signature Verified

Figure 3.1. Simple unencrypted Json Web Token example from jwt.io [9]

That is where JWT comes in. JWT has three parts - header, payload and signature (see figure 3.1). Payload contains any information that is needed about a user and signature makes sure the not one has been manipulating the payload. With JWT token containing all information need and having signature that could be checked by each client there would be no reason to constantly as authentication service for validation. It looked good, but there still was recommendations to use a reference token and value token for JWT. Reference token is just a big string that will travel outside system network and user will store it. Then some proxy will convert reference to value token when user request comes to the system. Value token contains information about user. Again, good idea but some service will again be overloaded since it has to convert reference token to value tokens.Gateway could do it but then it would be too much for it to do and it has to store key-value pairs.

We decided to instead to encrypt JWT with RSA256 2048 key and send value outside system network. We also sign JWT with different RSA256 2048 key and send the token over HTTPS. That way we create secure tokens that could be safely travel with encryption that takes time to break. This still invites the danger that someone will eventually breaks the encryption, but the same way someone could break into the network where system runs. Also if someone breaks the encryption they will learn structure of the payload, but they will have to brake signature key to be able to sign the payload. That will take a lot of time too - at least today. And since systems should changes their encryption key from time to time, by the time anyone brakes RSA encryption key and RSA signing key, system will already have new keys.

Performance downside of our approach is that tokens gets very big - around 1060 characters (see figure 3.2) . That means big headers and more traffic. Also machine that will decrypt and encrypt tokens has to be powerful to handle many requests.

```
eyJjdHkiOiJKV1QiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiYWxnIjoiUlNBLU9BRVAi
fQ.7Ewta22Me0aoQFIw5Ue4SGEmyjqkP8CLMsWdjuVtKid-hI3wBUhif9pkIeE_X_w6S
kcMikbvmV5wG6RrteYIrTrc1A65Ze3BqLGoiJIjGo3A3wHa0IoXkMsYttCsy4mK21M85
2NO5iA-we0-LuILobyhvrm8FmYe3k32mwtd9Sb0pJZlKNTf2sjIvBQP3x8qf1A_Hg_Zu
cddTzggfKRNDDQpXPOJsc-QaEZruSbrosYQcnnnkRMicr_JZz4uenh58EvZdzcmM4H3x
XUhJW7x4Zek5qLg1A1b3-oOHiBcrXXIay9Cj-Vm9xcA3aOIOrE0BH7zRJy8ULbuNecAP
xN92g.Lw_cwk7KKimDxlabswcOUw.xjeG0C0YodKyB9ySimdv9GGB_vnsMXmNfAg-U0N
6IggeKy5fuX8WRBxIdfckk858sQo0odNhvRL7yGFWctRDHLsWbuA0FOqFLkEyGqnNBnT
em0prwciuWWc-JOxT_f5hQ4mOmTcU4gG8sWTwbnro8rWMnnRa6sB1Xx-fwUCjCLf32vs
iuUBswGrLt0oN7c0w5q8E8597qAPmSi9IJQFW7jL0uOWZXe1IIs8mA7K5A53IV6MvYjy
v-hT66j4hoR3r2yAI_PW5naU30l9dz6K3kVNkgiSXhcrYBAcy3rdvwlicF8ftjin5DBf
a7BQ9pk2Cw3I1wwKhgMAYKkdpRSrJz8rdD_14JUa3V19GTNxjkMMnmELYeTVEwCNS-sp
GyucRgg_IF_6P5Y8Z0uM94f5hb2KVDRrKYQLm9T_nw56ppr2cAlWJCbdsc0Q9TDm6w0c
9MohjifPG5EPr4xgHK-YRKrGPWMnfMmIWRb-HHXV3BIH6ZOPbzk4-lGVjeIBWZUSge8v
hrBpWfiMYMAMQIdTfbJgSSIhqdAEt9SeR993rSoxnJ473icER20u2gEiCvL_CWthpr9m
SkpY0EBHEKF2yGg.F8rYlQ8RSsu1l7ss_xVhjg
```

Figure 3.2. Example of user JWT token in this system encrypted with RSA and signed by RSA

JWT tokens contain some information about the user. Of course no password. Username and granted authorities can be found i payload of a token. That is why token is so heavily encrypted and signed.

Authorization on each service

Different services provide different information to different users. Some clients will be applicants, some are recruiters and some are other services. Each business login service have a web security configuration and a custom filter to allow only certain client to use service and its certain endpoints.

Most endpoints require an authorization - client has to have a granted authority. JWT contains granted authority. When a request with JWT hits a service, it is handled by custom filter that decrypts and validates JWT. A token is valid if it was possible to decrypt it, signature is correct and it has not expired. If token is valid then information in it is added to security context of the service. After that the request goes a controller that might have a granted authority requirement (see figure 3.3). @PreAuthorize is used to set role constaint.

```
@PreAuthorize ("hasRole('RECRUITER')")
@PutMapping (value =  "/{language}/jobapplications/status/{id}",
consumes = MediaType. APPLICATION_JSON_VALUE)
public ResponseEnt ity changeStatusOnApplicationById(...) { ... }
```

Figure 3.3 Changing status of a job application is available only for user with role RECRUITER

Some endpoint are open for everyone without any authorization - for example login and registration endpoints.

Accessing service without gateway

A simple way to make sure that services are only accessible through gateway is to have all services running same network and give all services, except gateway, local ip-address. Also all services are secured by basic authentication which means outside user has to know username and password to the service itself.

Encrypting all client-traffic

To encrypt traffic to gateway system is using SSL. System have self-signed certificate with RSA256 2048 key exchange and AES_128_GCM cipher. That way all traffic is encrypted.

Access to config files

Access to config files is restricted and only config-service has credentials. Credentials are not saved in repository but distributed between developers. In future, information in config-files can be encrypted. Traffic between config-service and configuration information itself is encrypted using SSL.

Access to credentials files

There are several files holding sensitive information like secret to private keys. Those files will never made it to git repository. Those files are distributed between developers.

Logging

Information logging

Each service has different needs to store different information. Each team decides which information will be logged. General rules are to log incoming request and preferably different stages so a flow of execution can be seen for debugging. One very important rule is that no sensitive information should be in logging files. For example: no passwords and username+ROLE information should be stored in logs.

Error logging

Should be logged at the place the error occur. Also preferable with programmes own error message to understand error from this particular system view - java exception are very general, adding custom exception explanation can make debugging much faster.

5. Data View

Structure

There are two data sources for this project.
The first one, used at runtime, is a mysql server. It is accessible from anywhere on the internet by username, password and the non standard port. The reason for this is that we want to work with the same test data. Because it's easier to replicate a bug working with the same version of the program program and the exact same data.

The second database is an embedded h2 server meant for testing so we don't change the real data during a test.

Every service that need's a database has an mysql database connected to that container. At this time there are two databases, one for the user information and one for the applications. We thought that the user service should take care of everything that concerns the users and therefore has a db structure with only user information (seen in pic 5.1). The application service will require some user information but will ask for this information by the user services RESTapi. The application service therefore only contain application information and its structure can be seen in picture 5.2

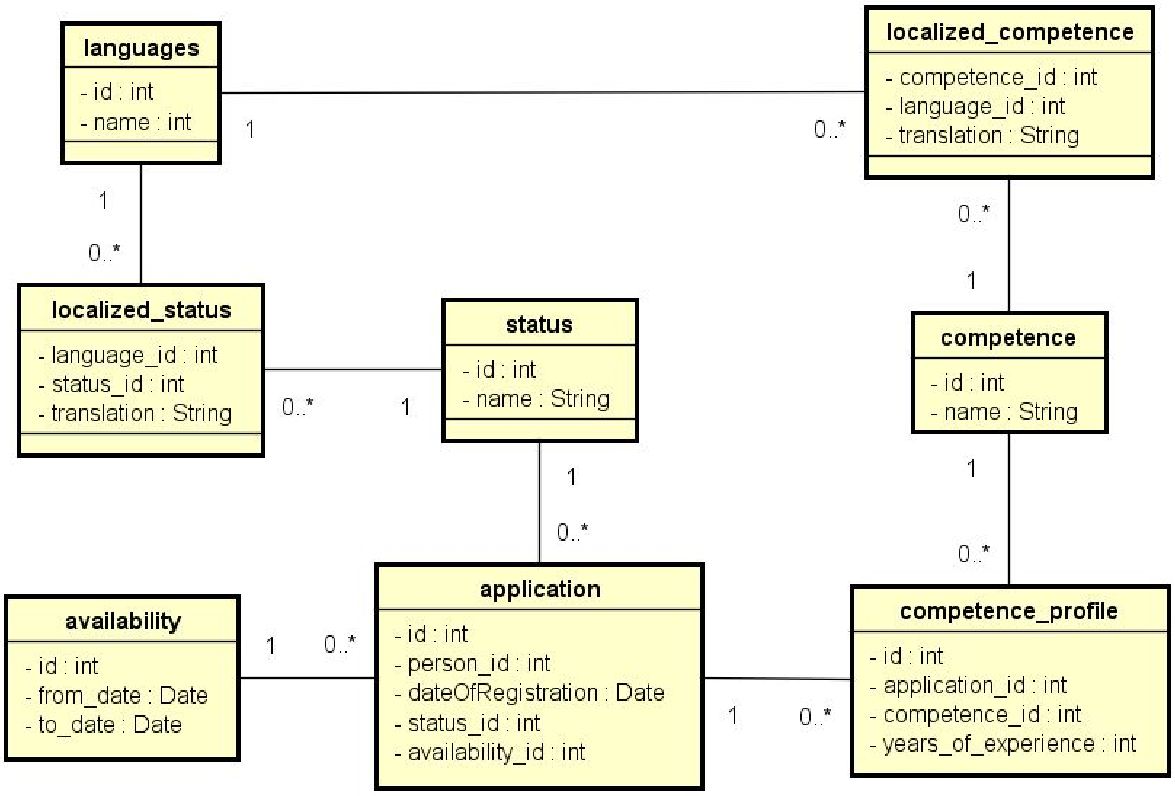


Figure 5.1. Database for JobApplication service

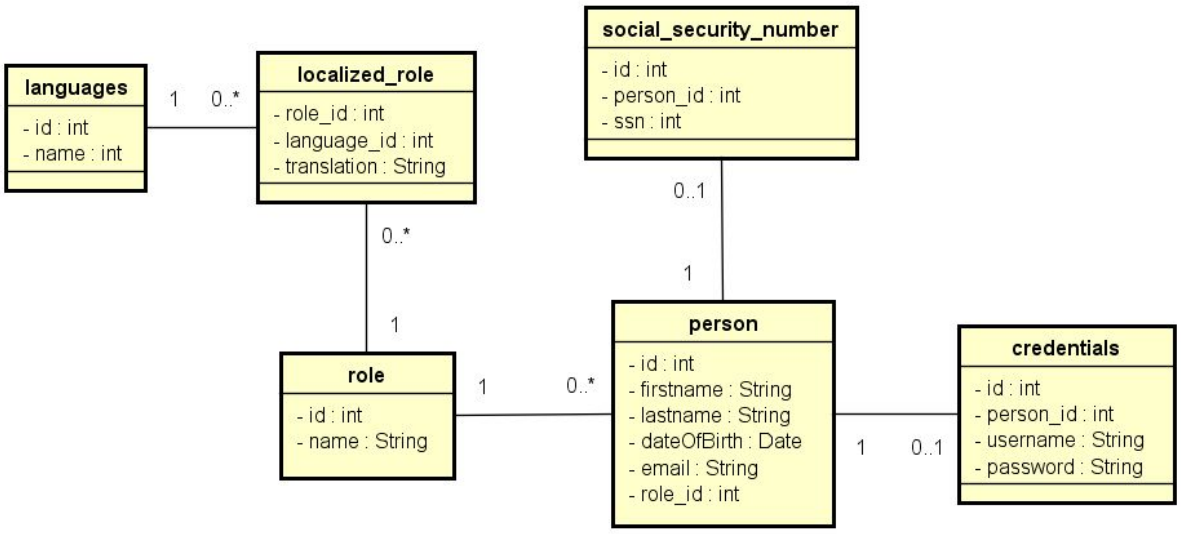


Figure 5.2. Database for registration service

Old SSN to new dateOfBirth

In old system SSN were entered by applicants and in the new system date of birth will be used instead. New database will be redesigned but old data cannot be lost. Solution is to create a new db table where persons id is a foreign key to the person and SSN store in SSN column.

Transactions

Transactions in this system are handled by Spring Framework [11]. We use declarative transaction management in form of the @Transactional annotation. The annotation can be used on both classes, interfaces and methods. In our case we annotate the repositories with the @Transactional annotation (see figure 5.3). This will make all the methods in the interface make use of transactions when making a database call. When using the annotation without attributes the transaction will be used both for read and write operations. The propagation of the transaction is set to required as default which means that if it already exists a current transaction, a newly created transaction will join that transaction. If not, a new transaction will be created.

The transaction will be started just before a method is called. The transaction aspect will call a transaction manager that is going to decide whether to start a new transaction or join an existing one. If a new transaction is created, the transaction manager will create a new entity manager and bind it to a thread and a datasource. The datasource and transaction manager will be auto configured by Spring Boot. When the method call is made, the call will go to a proxy that retrieves the entity manager from the thread that executes the database operation. If an unchecked exception (RuntimeException) is thrown the transaction will do a rollback. If no exceptions has been made the transaction will do a commit. This is handled by the transaction aspect after the method call. The transaction aspect works like an interceptor for the method call and will therefore “go around” the actual call by doing some logic both before and after the actual call. As for transactions in general the transaction is atomic, which means that the transaction will either be completed (when committed) or have no effect at all (when roll-back).

```
@Transactional
public interface CompetenceProfileRepository extends
CrudRepository<CompetenceProfileEntity, Integer> {
}
```
Figure 5.3 Spring Repository

6. Non-Functional View

This part includes information about non-functional requirements that are not mentioned in other parts of the documentation. For security see security section, for packaging see implementation view and so on.

Other considered non-functional requirements

- Scaling
- Availability
- Reliability

Scaling

Independent services that are packaged in a Docker container are easy to scale. Just start a new container.

Availability

Since we have independent services that can be horizontally scaled something has to handle which request should go to proper instance. Otherwise no there is no way to use scaling if all requests will go to the same service which eventually will not be able to handle all requests. That is why a load balancer is used. Load balancer will keep track on load on each service and choose one a right one to send request to. Also load balancer keeps track of which servers are down and makes sure that no request goes to a down service. Since this system used Zuul as gateway a load balancer comes in the same package and automatically is used by the proxy.

Also all services will in future release have a load balancer to perform client-side load balancing.

Reliability

Services can go down. That happens. To make sure that no write (POST/PUT) request gets lost due to down service we send all POST/PUT traffic will be send using Redis message broker. As long as message broker is up no write request will be lost.

7. Deployment View

Since the architecture of this project is Microservices, every service can run on separate hardware. Different nodes communicate by network.

Each service runs on Tomcat application server configured by Spring Boot. Each service has its own configuration that is delivered by configuration service - configuration service has its own configuration.

All services except configuration service register themselves at Discovery service and after that each services can contact each other by name instead of ip-addresses and ports.

Auth-service, register-service and jobapplication-service has their own RESTapi that clients can send requests to. Services can also be clients, for example auth-service asks register-service for detailed information about a user with given username. Edge service also kind of has an RESTapi but it just forwards requests.

The structure of the Microservices can be seen in picture 7.1

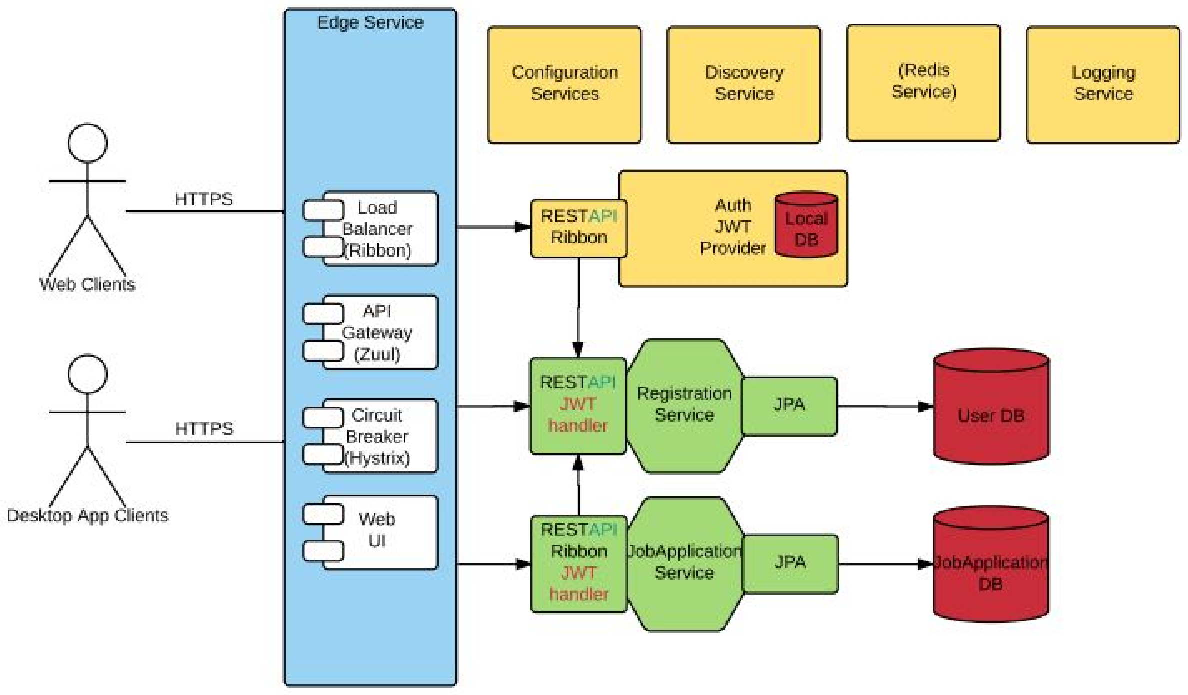


Figure 7.1. Microservices deployment diagram

Explanation to the diagram:

- Blue containers - has connection to outside world
- Green containers - business logic
- Yellow containers - services that handle non-functional requirements, connected to all nodes except DB (and Client)
- Red containers - databases
- White components - important components inside a service

8. Implementation View

Instruction to build and run the system:

The system can be deployed in two different ways. one more suitable for developers and one for the end user. We decided to do this because the first one creates smaller containers but requires the user to install dependencies on their own system and also download the code manually. The end user should not be concern on how to install this requirements on their system or how to retrieve the source code and this build will take care of all dependencies inside the container and also download all the necessary code itself.

Development

Information

Running Environment

Development is supported on different platforms. Both Unix and Windows OS can be used to develop this system. Development environment supports Docker Toolbox and newer Docker where VM is not required.

To run the application all that is needed is Docker machine to run containers. All needed component already exists in docker containers like openjdk 8.

To run locally without Docker, at least Java 8 has to be installed with JCE package.[10] If any encryption/decryption errors encountering, there is a great chance that JCE package is missing.

Building, deployment and running

Services are build by Maven and packaged as Jar-files. Jar-files are putted in Docker images. To automate the process Maven Docker plugin is used to build Jars and build Docker images. Maven Docker Plugin also deploys image to given Docker machine.

Requirements

- Docker - to run
- Maven - to build
- Git - to develop

Instructions to deploy services

1. Have docker installed on the machine
2. Open terminal and go to shell_scripts directory
3.
    - For Docker Toolbox/Windows without Hyper V
        a. In build-services-docker-toolbox.sh enter your specific DOCKER_HOST and DOCKER_CERT_PATH
        b. Run the script
    - For MacOS/Linux:
        a. Run build-service-docker-unix.sh
4. Run start-config-service.sh - it has to be started before other services
5. Run start-other-services.sh

Production

Information

Running Environment

To run the application all that is needed is Docker machine to run the containers and being able to unzip .zip or tar.gz

Building, deployment and running

The build will already be compiled and stored in folders with the name connected to the service. To deploy the services we just execute the deploy script. This will take the user to a menu with three step by step guides; on setting up all services, just restarting one/all or updating the system. At the full installation option the host will also receive a "recruitment system" shortcut command for easier maintenance.

On the topic of creating ssl keys for the application the user have the options to provide their own keys, create keys with their own information or just make a quick install with no information provided.

If the system has been installed and the message "all services are now running without any errors" is shown, the user can navigate themselves to a web browser of their own choice and go to **{runninghost}:8080**. Running host can be localhost or ip-address of your Docker machine. There they will be provided with a login form.

Standard credentials for a recruiter

- **Username** : Recruiter
- **Password** : 7a9d89as79d8as7d

If you need to see the logs of a service, you can do so by running:

docker logs name_of_the_service

or restart one service by running:

docker restart name_of_the_service

Requirement

- **Docker**

Releases

A release is a version of the application that could be shipped to the client. It should be production ready and well documented. Every release should follow the following structure, making all information about how to install and install script valid.

The releases is mainly docker based for a easy deployment and controlled environment but there is nothing stopping a client from running all services directly on the host. There is no documentation on how the client would do this because running them directly would introduce many more possible problems that aren’t connected to our product. So they are free to try to set it up on their own.

The installation and deployment of a release follows the **production deployment** described above.

Every service should have their own version in the docker image. This is important for maintenance and bug fixes later.

Folder structure

The following folder structure shows how the client would receive the application after checking that the files haven’t been altered with by comparing hash codes and then unzipping it. The structure is important for the install scripts, because they only go through the structure and start the services. By doing it this way we do not need to change any script if we would introduce a new micro service or remove a micro service.

The structure should be divided into documentation, run/deploy scripts and the micro services in jar format with a connected dockerfile that is used to create the docker image. The actual source is provided in the theRecruitmentSystem folder. When handing over from one team to another the git history will be removed by deleting the git folder to prevent any retrieval of old keys/password that might been used.

A visualisation of the structure can be seen in picture 8.1.

- folder root
  - **docs**/
  - **scrips**/
  - **services**/
  - **theRecruitmentSystem**/

- ■ .git/

Figure 8.1 folder structure of a release

Docs

All releases should have the latest documentation directly in the folder's root under doc/ and should include:

- javadoc
- UML classdiagram
- database structure UML
- database start structure & data file for an easy import

for all the services. It's the developer's responsibility that new changes are documented.

The README.txt file should contain information about how the client can go from validating the package signature to having a running system and about how a regular backup could be made and how a recovery could be executed.

Archiving releases

Every release should be documented in the (DockerReleses/Versions/releases.md) file. It should contain the offical version number of the application, the name of the different file options that are given to the clients and two different checksums in the form of md5 and sha256 as well as the date of the release.

An example of a release can be seen below:

Version	File	checksum_md5	checksum_sha256	release date
v1.0	TheRecruitmentSystem_v.1.0.tar.gz	e03ede35c89bd856ecec74c2f2f5f8f9	f67a6f0ee384b51badb1073078f6ead98e0949e6b983a0652a4ad06b4effab38	2017-03-14

Figure 8.2 example of a release

Tools, frameworks and libraries

- Spring Framework with Spring Boot, Spring Cloud, Spring Security and more Spring projects - main development framework
- Project Lombok - runtime generator for constructors, getters, setters and toString methods
- Nimbus Jose JWT - for Json Web Tokens
- Minidev Json Smart - for Json manipulation
- Maven - for dependency managing, test running, building docker images
- JUnit 4 / JUnit 5 - unit testing framework

- Spring Test - spring integration testing framework
- Mockito - framework for mock tests in Java
- Netflix Eureka - discovery service so that services can find each other
- Netflix Hystrix - implementation circuit breaker design pattern that handles situation when service is unavailable
- Netflix Ribbon - client side load balancer that knows which servers are up and available
- Redis - used as message broker between services to make sure that no messages disappear due to down server
- Docker - packaging and deployment tool
- AngularJS - client side model
- Bootstrap - web client layout

9. Problems

Data consistency in Microservices

Most important in this architecture is that services are independent. Developers can easily work on changes and depend on others except public API. Services can be horizontally scaled easily. Sounds good, but when it comes to question of database and data consistency there is a problem. How can teams work independently when database is used by everyone? Solution is simple - every service has it's own database. Good, but if all services have their own databases, how can data be consistent?

There are several solution. One is to use event-driven method where a writing to database is an event and when that happens services will get a notification that it is time to update their data.

Due to time constraint and resources limitation in this course project we implemented different solution. We have only one database and different services has a limited access to database tables using database build-in authentication and authorisation.

Distribution of sensitive data between developers

Sensitive information like password and keys is distributed between developer without using open public channels like version control system. Developers get information using USB or could use LassPass secure notes. The problem lies in people themselves. How to handles situations where developers are leaving the team but still has access to sensitive information? One way is to use an account for each developer and using that account each developer has access to a credentials service. One example is Cloud Foundry.

Outdated docker start image Java:8

Outdated docker start image Java:8 will be changed to openjdk image, "This image is officially deprecated in favor of the openjdk image, and will receive no further updates after 2016-12-31 (Dec 31, 2016). Please adjust your usage accordingly." [7]

10. References

[1] Richards, M. (2015) '*Microservices Architecture Pattern, Pattern Description*', in Scherer, H. (ed.) Software Architecture Patterns Understanding Common Architecture Patterns and When to Use them. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, pp. 27.

[2] Richards, M. (2015) *'Microservices Architecture Pattern, Pattern Analysis'*, in Scherer, H. (ed.) Software Architecture Patterns Understanding Common Architecture Patterns and When to Use them. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, pp. 34–35.

[3] Richards, M. (2015) 'Microservices Architecture Pattern, Pattern Topologies', in Scherer, H. (ed.) Software Architecture Patterns Understanding Common Architecture Patterns and When to Use them. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, pp. 29–32.

[4] Li, R., Oliver, K. and Rajagopalan, R. (2015) *Baker street: Avoiding bottlenecks with a client-side load Balancer for Microservices.* (Accessed: 9 February 2017).

[5] NewCircle Training (2016) Building Microservices with spring cloud. (Accessed: 9 February 2017).

[6] Syer, D. (2015) Spring and angularJS: A secure single Page Application. (Accessed: 9 February 2017).

[7] Java's official docker repository

[8] API Gateway OAuth 2.0 Authentication Flows

[9] JWT.io (Accessed: 9 march 2017)

[10] Java Cryptography Extension (JCE) Unlimited Strength for Java 8 (Accessed: 9 march 2017)

[11] Introduction to Spring Framework transaction management
(Access: 9 march 2017)