# Project files

The project can be found at (git link)

# How to start the project

To start the application, enter the following command in the root directory:

docker compose up -d

The -d is an optional flag that starts the container in a 'detached' mode. Without this option, the output runs in your terminal.

The command starts up the "SurveyService", "ResponseService" and a RabbitMQ instance.

If you make any new changes, you can rebuild the images by adding the --build flag at the end. The resulting command looks like this:

docker compose up -d --build

To launch the backend in a 'production mode', you can use the above `docker compose up -d` command.

With this, everything will be run in docker containers in one package, and the Spring Profiles (<https://www.baeldung.com/spring-profiles>) will make sure the environment variables are set properly.

To run everything in a 'development mode', you can individually launch every service using IntelliJ IDEA or just by building and running the project.

Additionally, you will need to install and run a RabbitMQ instance on your computer. This is what I used on my Windows machine: [https://www.rabbitmq.com/install-windows.html#downloads](https://www.rabbitmq.com/install-windows.html%23downloads%20) to run it locally.

If you want to run the RabbitMQ instance in Docker, you will need to adjust the development profile within `main/resources/application-dev.properties` to change the host to the host of the docker container. You have to do this for every related service.

# Cloud deployment

To upload the images to the Azure Container Registry, run the following commands:

1. Login to the ACR using the Azure CLI:

az acr login --name samenaanz.azurecr.io

1. Change image tags to the Container Registry aliasses, and push to Registry.

Survey Service:

docker tag backend-surveys-1 saztestregistry.azurecr.io/backendimages/surveys

docker push samenaanz.azurecr.io/backendimages/surveys

Response Service:

docker tag backend\_responses samenaanz.azurecr.io/backendimages/responses

docker push samenaanz.azurecr.io/backendimages/responses

API Gateway:

docker tag backend\_api-gateway samenaanz.azurecr.io/backendimages/api-gateway

docker push samenaanz.azurecr.io/backendimages/api-gateway

RabbitMQ:

docker tag rabbitmq:3-management samenaanz.azurecr.io/rabbitmq

docker push samenaanz.azurecr.io/rabbitmq

# Front-end

The front-end uses for the following core technologies:  
- SurveyJS (Form library, Creator and Analytics components)  
- Keycloak-Angular with Keycloak-JS  
- Bootstrap

|  |  |  |
| --- | --- | --- |
| Class | Directory | Purpose |
| App.guard.ts | Auth | Authentication guard that runs custom code when accessing a protected page |
| App.init.ts | Auth | Initialize function required to setup keycloak |
| Survey-persistance.service.ts | Services | Wrapper around HTTP client that sends request to the back end server |
| Survey-local-storage.service.ts | Services | Wrapper around local-storage |
| Keycloak-provider.service.ts | Services/Auth | Wrapper around the Keycloak Service provided by Keycloak Angular library |
| Application-roles.ts | Models | Roles in the application |
| Claim-types.ts | Models | Claims present in the access token |
| Survey.ts | Models | Format of a survey |
| App-routing-module.ts | App | Application routes |

The remaining files are the pages in the application. These can be found in the /views folder. Some views use other components that are often re-used. These components are present in the /components folder.

# Security

**Keycloak**

We use Keycloak to secure both the front-end and back-end applications. So far only the front-end is secured using the keycloak-angular and keycloak-js libraries. The keycloak-angular library is a wrapper around the keycloak-js library that makes it easy to integrate Keycloak into an Angular application and interface with the Keycloak instance (authorization server). When a request is made to the back end, the access token is included automatically. At the moment the access token is not validated by the back-end application yet.

The user is always forced to login in order to use the application, as any route is protected by an authentication guard. Some routes require a specific user role to access a route. A user will be redirected to a login page hosted on the authorization server if not logged in. After a successful login, Keycloak will return the user to the front-end application. We get user-specific data based on a user id that is included in the access token. Keep in mind that if the Angular application can't connect to a Keycloak instance, it will return a blank page.

**Authentication Guard**

This is a function that will be called automatically, when a protected route is accessed. It returns a Boolean in order to decide if a user is authorized to access a page. Within this function, you can customize this functionality to fit the application needs.

At the moment, it checks if the user is logged in, and retrieves the required roles to access a route. These routes are then compared with the roles that the user has in order to decide if the user is allowed to access a page. If not, the user will be redirected to the unauthorized page.

When registering routes to correspond to a certain page, you can add the authentication guard to protect it. If you want to require certain roles, you need to set the data property for that route. Please refer to the Setup\_Guide document / source code for examples.

To use the current project with Keycloak, you need to have the correct setup. In order to have the same settings that were used for this application, import the realm-settings.json file into your keycloak realm on creation. After that, you need to create users and assign them a role.

# Recommendations

* CI pipeline to validate code
* Automatic deploy job
* Test if cloud environment works properly under actual conditions, see if it scales etc.
* Link frontend and backend
* Secure backend with Keycloak
* Set up persistent database for production environment
* Write unit tests
* Write end-to-end tests
* Perform load testing to verify if the architecture works
* Further separate backend so the "get surveys" and "create response" functionalities are in their own microservices, since they're expected to have the vast majority of load.