smava’s Back-end Engineering Homework

Goal

The goal of this homework is to assess the candidate's capability, to provide a well-architected micro-service(s) based solution, given a Monolith Web Application.

Monolith - an old Web Application with all Cross-Domain and Cross-Functional features in one bundle. **Repository of Monolith** can be found here: https://github.com/smava/monolith.

Problem statement

As part of the Monolith, there exists a **Loan Request flow**. Our goal is to replace this flow of a monolith web app with a fleet of micro-service(s).

Refer to **`RegistrationController`** in the above Monolith to understand how the old Loan Request flow works.

In the Monolith Loan Request flow, we capture the objects below in a single JSON payload and save **`User`**, **`LoanApplication`** and **`Customer`** details in the same database.

`User`

```
{
  "id": 1,
  "username": "johnsmith",
  "password": "*******"
  "roles": "ADMIN,USER"
}
```

`LoanApplication`

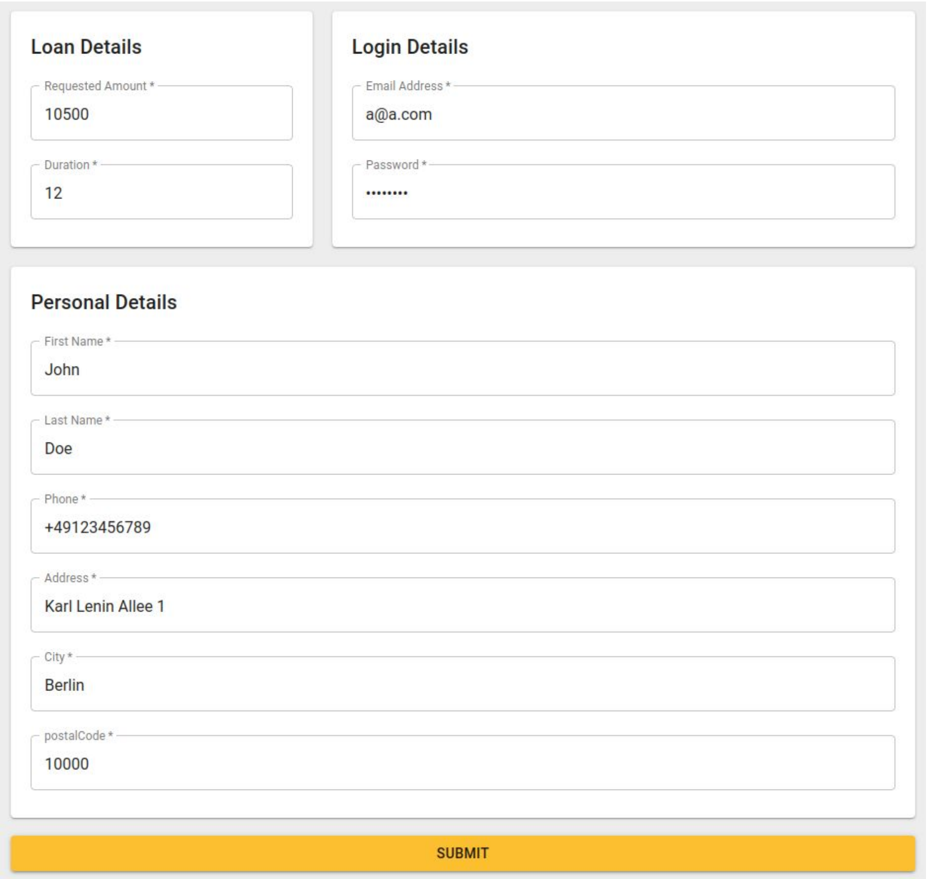
```
{
  "id": 101,
  "customerId": 11,
  "amount": 1000,
  "duration": 12,
  "status": "CREATED"
}
```

`Customer`

```
{
  "id": 11,
  "userId": 1,
  "firstName": "John",
  "lastName": "Smith",
  "email": "johnsmith@example.com",
  "phone": "+49 123 456 78 910"
}
```

New Loan Request flow

To visualize the UI and how the new Loan Request flow would look like, refer the sample UI and the flow below.



Loan Details

Requested Amount * 10500

Duration * 12

Login Details

Email Address * a@a.com

Password * ••••••••

Personal Details

First Name * John

Last Name * Doe

Phone * +49123456789

Address * Karl Lenin Allee 1

City * Berlin

postalCode * 10000

Assume back-end processing starts immediately after the user submits the form above.

Imagine the front-end applications orchestrate the Loan Request flow in the way explained below:

1. Call your new micro-service(s) through protected Gateway micro-service obtaining Access Token using provided user credentials.
   There are 2 users predefined:

John	Jack

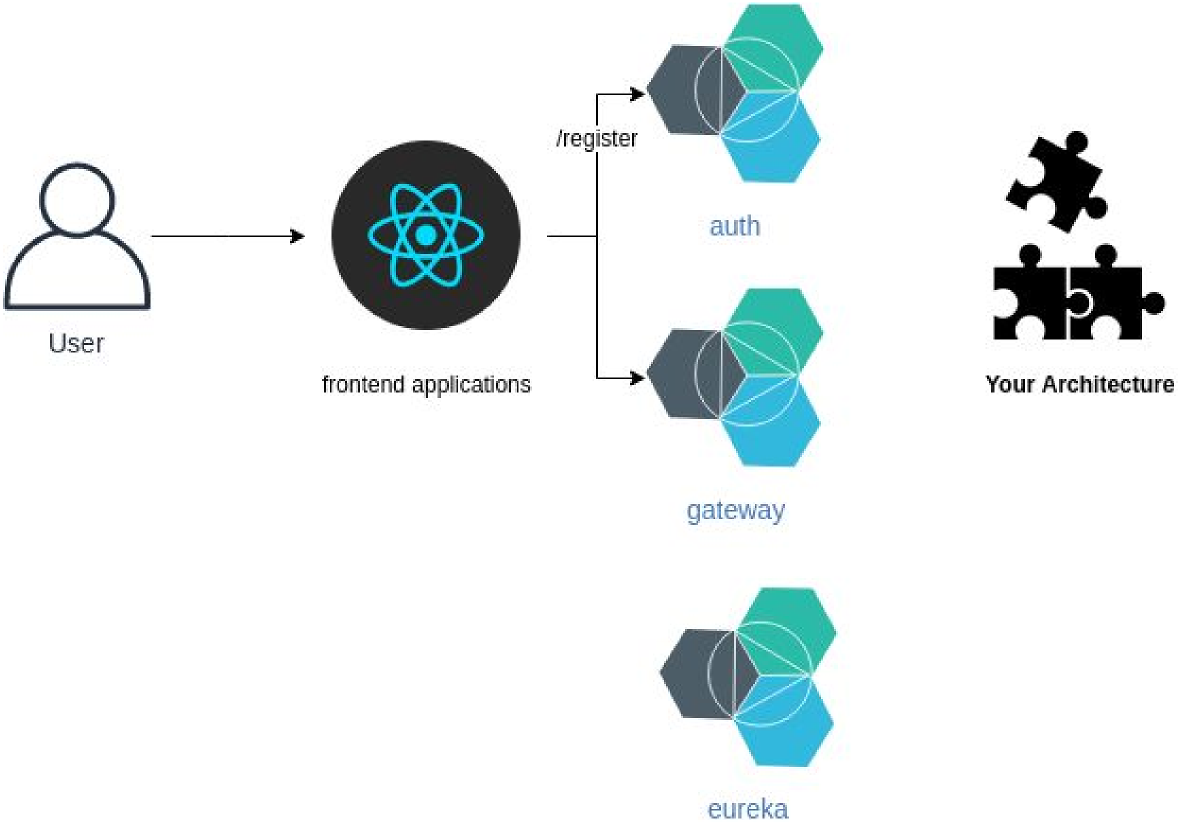
{
  "id": "1",
  "username": "john",
  "password": "john"
}
	{
	  "id": "2",
	  "username": "jack",
	  "password": "jack"
	}

> A new user can be registered using the POST: `/register` API in auth micro-service.

The default client details for OAuth are:

```
client_id: client
client_secret: secret
```

2. Save `LoanApplication` and `Customer` details in our system for processing.
    a. `userId` obtained in step #1 is passed along with `Customer` payload - to create a new `Customer`.
    b. `customerId` obtained in step #2.a is passed along with `LoanApplication` payload - to create new `LoanApplication`.



Challenge

The challenge is to **save** the loan request data for processing in a fleet of micro-services you should develop.

During loan request processing we fetch applications from different banks and update the status of `LoanApplication` with `[CREATED, DENIED, APPLIED]`.

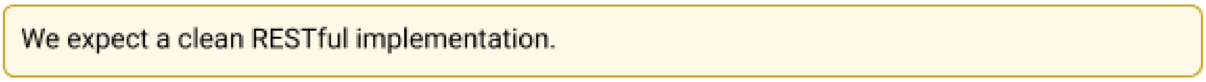
We expect you to **create 2 micro-services** with best practices in mind and **provide 4 REST APIs** mentioned in the "Required API endpoints" section.

For the sake of simplicity let's just name them `customer-service` and `loan-application-service`.

Required API endpoints

The following 4 endpoints are expected to be implemented.

Please do not change the signature of API. All the API signatures should be accessible through the API Gateway.



We expect a clean RESTful implementation.

1. Create `LoanApplication`

`POST /api/loanapplications`	
`Request`	`Response`

`{`
`"customerId": 11,`
`"amount": 1000,`
`"duration": 12`
`}`
	`{`
	`"id": 101`
	`}`

2. Retrieve `LoanApplications` by `CustomerId`

`GET /api/loanapplications?customerId=11`	
`Request`	`Response`

	`{`
	`"customer": {`
	`"id": 11,`
	`"firstName": "John",`

```
    "lastName": "Smith"
  },
  "loans": [
    {
      "id": 101,
      "amount": 1000,
      "duration": 12,
      "status": "APPLIED"
    },
    {
      "id": 102,
      "amount": 2000,
      "duration": 24,
      "status": "DENIED"
    }
  ]
}
```

3. Create Customer

POST /api/customers

Request

```
{
  "userId": 1,
  "firstName": "John",
  "lastName": "Smith",
  "email": "johnsmith@example.com",
  "phone": "+49 123 456 78 910"
}
```

Response

```
{
  "id": 11
}
```

4. Retrieve Customer

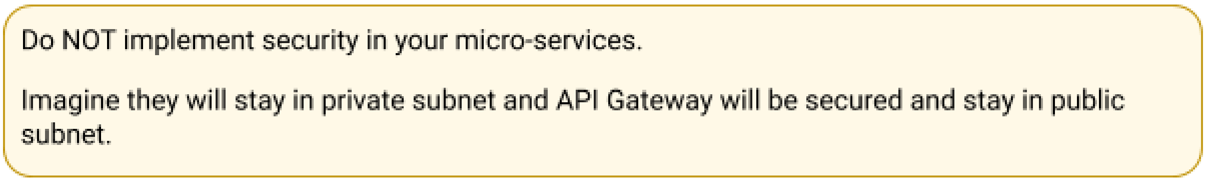
GET /api/customers/11

Request

Response

```
{
  "id": 11,
  "userId": 1,
  "firstName": "John",
  "lastName": "Smith",
  "email": "johnsmith@example.com",
  "phone": "+49 123 456 78 910"
}
```

Given



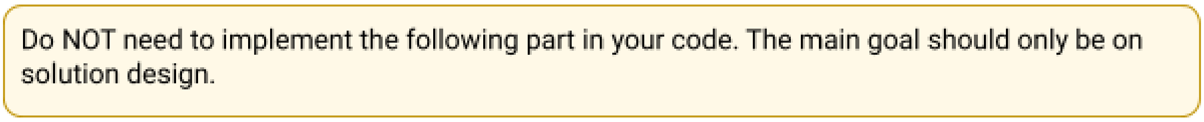
Do NOT implement security in your micro-services.

Imagine they will stay in private subnet and API Gateway will be secured and stay in public subnet.

We have the micro-services below ready, so that you can concentrate on the solution right away:

1. **Auth** microservice is an authorization service.
   User data is stored in Auth micro-service's database (i.e. `username` and `password`).
2. **Eureka** microservice is a service discovery service.
3. **Gateway** microservice is an API Gateway to those micro-services you will develop.

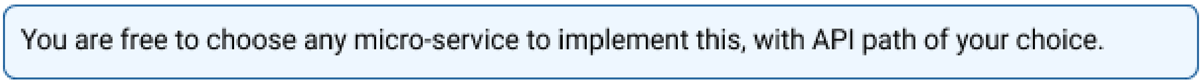
Additional Requirement



Do NOT need to implement the following part in your code. The main goal should only be on solution design.

Assume that the micro-services you provide will be used to create a search functionality of `Customers` based on `LoanApplication` `Amount` and `Status`.

Create a **Story.md** file in your repository directly under root folder with description on how to design such an API.



You are free to choose any micro-service to implement this, with API path of your choice.

<table>
<tr><th colspan="2">GET /search?minAmount=1000&maxAmount=10000&status=APPLIED</th></tr>
<tr><th>Request</th><th>Response</th></tr>
<tr><td></td><td><pre>
[
  {
    "id": 11,
    "userId": 1,
    "firstName": "John",
    "lastName": "Smith",
    "email": "johnsmith@example.com",
    "phone": "+49 123 456 78 910"
  },
  {
    "id": 12,
    "userId": 2,
    "firstName": "Mehmed",
</pre></td></tr>
</table>

```
    "lastName": "Demir",
    "email": "mehmeddemir@example.com",
    "phone": "+49 109 876 54 321"
  }
]
```

Expectations

Must Have

1. Should provide an appropriate and clear set of instructions for **Build, Startup, Usage and Testing** process of the services.
2. New micro-services should be created adhering to the best practices and Micro-service Design Patterns.
3. The new services should be registered to Eureka/Discovery
4. The APIs should be accessible through secured API Gateway (i.e. a registered user's OAuth Token should be able to authorize a client to access your APIs).
5. Should implement docker based solution by extending existing `docker-compose.yml` file.
6. Should use database and a data access layer (preferably JPA) in the program to exhibit experience in them.
    a. Should **not** use in-memory Datasource. You can optionally choose Postgres datasource which is part of `docker-compose.yml`. If you prefer some other database you could use it with docker.
7. Micro-services should be based on Spring and Spring Boot.
8. Micro-services should be production-ready in terms of logging and monitoring.
    a. Logger configuration
    b. Log statements
    c. Include monitoring/metrics publishers
9. Write at least one Unit and one Integration Test for each of the micro-services.
10. Final version of code changes should be merged to Master branch.

Nice to Have

1. Making use of Messaging Queue System.
2. Making use of Distributed caching solution.
3. Can have Swagger-UI for the new micro-services.
4. JUnit-5 can be used.
5. Distributed Tracing for Logs.
6. API tests as part of integration-tests.