Solução Técnica:

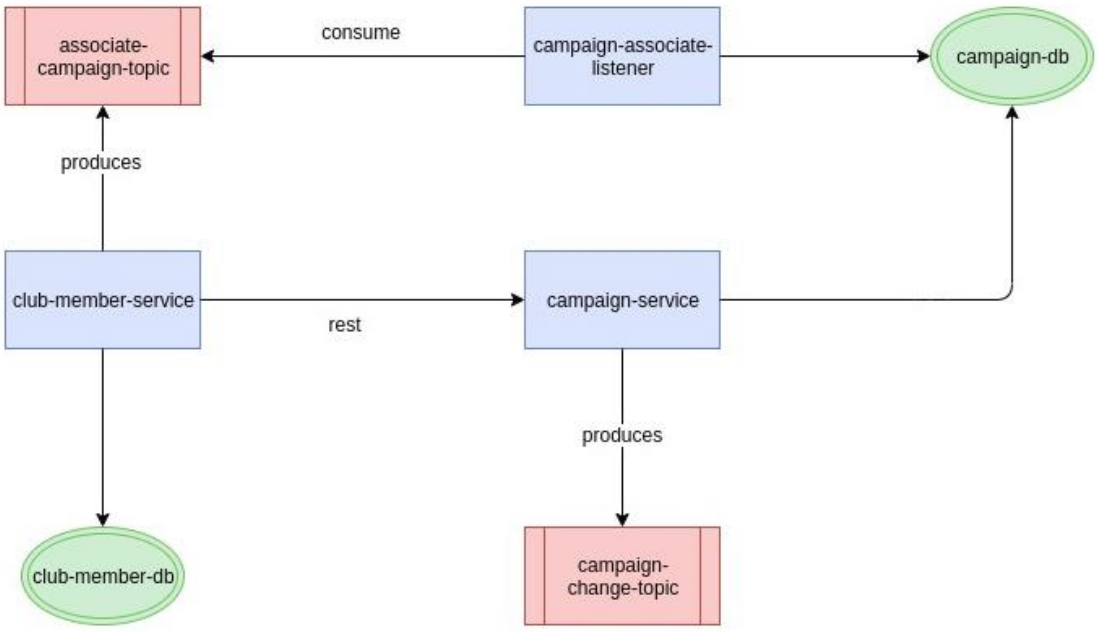
O sistema está dividido em três aplicações, sendo elas campaign-service, club-member-service e campaign-associate-listener. Duas bases de dados estão sendo utilizadas, campaign-db e club-member-db.

A aplicação campaign-service é resposável pelo CRUD de campanhas na base de dados campaign-db. Ela também possui a resposabilidade de produzir mensagens para cada campanha atualizada na base para o tópico campaign-change-topic, avisando assim os sistemas consumidores do tópico sobre a atualização de uma campanha.

Club-member-service é responsável pelo cadastro de usuários na base club-member-db. Ele acessa a aplicação campaign-service via rest (endpoint /campanha com parameter ‘nome_time’) para buscar as campanhas que podem interessar o sócio pelo nome do clube e também para saber se o sócio já possui campanha vinculada(endpoint /campanha com parameter ‘id_socio’). Caso não obtiver resposta, ele retornará apenas os dados do sócio cadastrado. Ele também é responsável por produzir mensagem para o tópico associate-campaign-topic, informando que uma associação entre sócio-torcedor e campanha deve ser realizada.

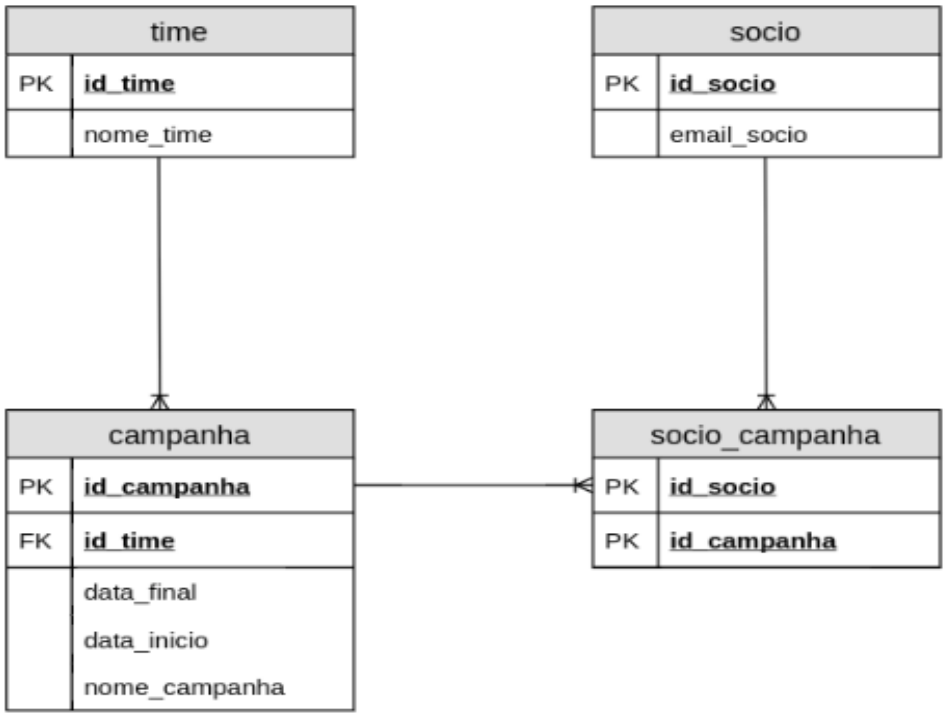
A aplicação campaign-associate-listener, por sua vez, é responsável por consumir o tópico associate-campaign-topic, fazendo assim a associação na base campaign-db entre o sócio torcedor cadastrado, e as campanhas vigentes para o clube do sócio no qual ainda não está associada a ele.

Abaixo, está o desenho da arquitetura da aplicação:



Modelagem das bases de dados:

campaign-db:



club-member-db:

Técnologias utilizadas:

O banco de dados utilizado foi o MySql.

As aplicações foram codificadas utilizando a linguagem JAVA, com o framework Springboot. Foi utilizado também JPA com Spring Data para mapeamento objeto-relacional.

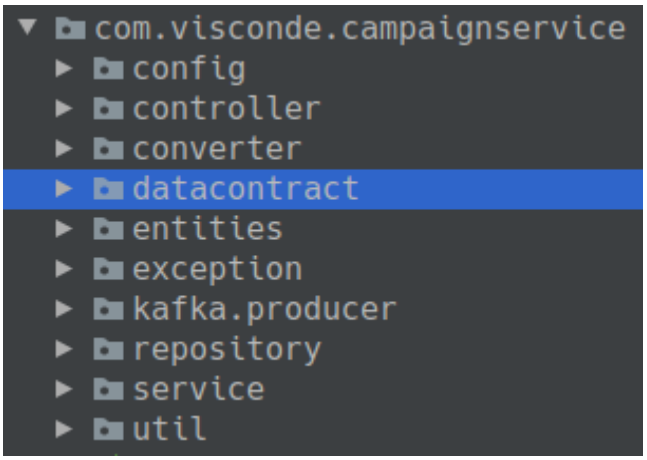
Foram utilizados também os frameworks producer/consumer de mensagens Kafka do Spring, além do lombok para a geração de getters, setters, construtores e builders. Para comunicação REST, foi utilizado o framework declarativo OpenFeign.

Como ferramenta de build da aplicação, foi utilizado o Apache MAVEN.

Para testes unitários, foram utilizados os frameworks do JUNIT para controle e execução dos testes, assim como o MOCKITO para criação de mocks para dependências.

Foi utilizado também o Swagger 2.0 para a documentação dos contratos (chamar host/swagger-ui.html).

Estrutura de packages:



config → Configurações e factories para a aplicação.
controller → Disponibiliza os endpoints REST.
converter → Contém as classes responsáveis para conversão de objetos (Exemplo: objeto que representa uma entidade na base dados para um objeto que representa um JSON).
datacontract → Representa um objeto utilizado como request ou response de uma requisição.
entities → Contém classes que podem representar objetos da base de dados.
Gateway → Classes clients responsáveis por chamadas externas (Interfaces OpenFeign).
exception → Exceptions customizadas para atender regras de negócios e fluxos alternativos.
kafka →Contém classes que possuem resposabilidade de producer/consumer.
services → Contém as interfaces de regra de negócio.
services.imp → Contém as classes com implementação das regras de negócio.
repository → Interfaces que extendem JpaRepository para acesso a base de dados.
util →utilitários

Estrutura do código:

A convenção de nomes de classes/métodos e variáveis seguem a padronização definida pela Oracle (camelCase), exceto para testes unitários. Todas as nomenclaturas da aplicação foram codificadas utilizando a língua inglesa, ficando apenas os atributos de request/response ou atributos relacionados a base de dados (nome de colunas/nome de tabelas) em português.

Todas as dependências são injetadas via construtores. Isso impede que seja injetada uma dependência nova sem que os testes unitários sejam atualizados, causando erro de compilação nas classes de testes.

O foco na criação dos testes unitários foram para as classes services, controllers e converters. Os testes unitário sempre se iniciam com a palavra ‘should’, indicando o que deveria ser feito naquele pedaço de código que está sendo testado. A separação de palavras nos testes unitários é feita pelo caracter ‘_’, para dar mais legibilidade para o teste (exemplo: ‘should_create_a_campaign’). As dependências da classe que está sendo testada são mockadas pelo mockito, porém a injeção é feita de forma manual para haver um erro de compilação caso seja inserida uma nova dependência na classe que está sendo testada.

Para acesso aos contratos das APIs para testes, acessar o Swagger das aplicações na seguinte URL: host/swagger-ui.html.