POLL - RESTful API

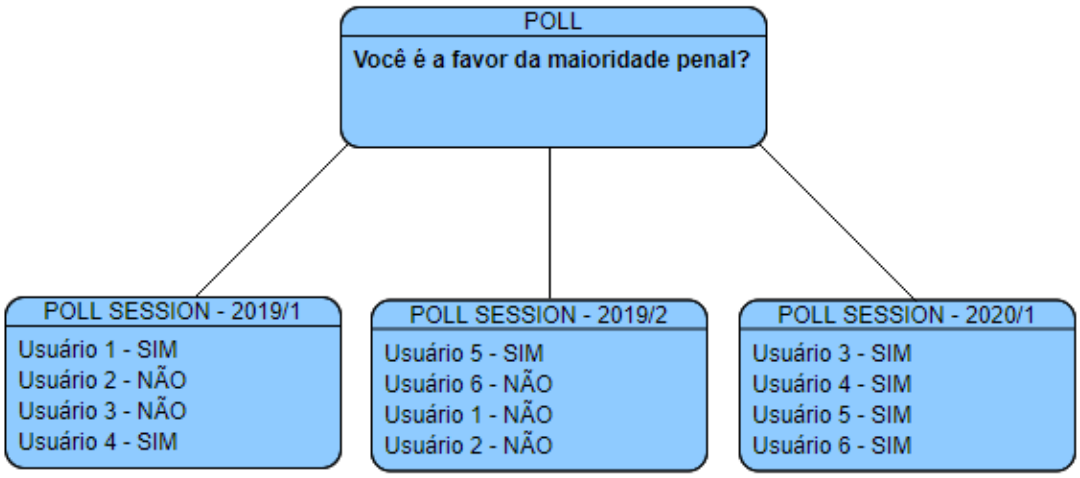
Documentação Informativa

1) INTRODUÇÃO

Este documento tem por objetivo discutir uma solução proposta para implementação de uma API Rest que controla instâncias de votação e que foi publicada/disponibilizada na nuvem.

Abaixo será apresentada uma explicação, debatendo estratégias escolhidas: tanto no âmbito técnico, como no âmbito de modelagem do problema e estratégias adotadas para solução. Ao final, será discutido possíveis evoluções da aplicação, com sugestão de melhorias.

Ponto de atenção: foi considerado que cada sessão de votação é um caso isolado. Ou seja, usuários não podem votar mais de uma vez por sessão. Mas podem existir N sessões de uma *template* de votação.



Desta maneira, teremos resultados de votação para cada sessão de votação. O resultado de uma votação pode ser buscado através do *endpoint*:

```
/api/v1/sessions/{id}/details
```

2) TECNOLOGIAS UTILIZADAS

Para implementação da API Rest, foi utilizado *Spring Boot* para facilitar todo o processo de configuração e publicação da API. Utilizando este recurso, conseguimos focar no negócio, delegando para o *framework Spring* todo trabalho e controle custoso de configuração do projeto.

2.1) PUBLICAÇÃO NA NUVEM E INTEGRAÇÃO CONTÍNUA

A API Rest foi publicada na plataforma de nuvem *Heroku*, estando disponível para utilização via https://poll-restful-api.herokuapp.com/swagger-ui.html.

É importante salientar que foi configurada estratégia de integração contínua, uma prática DevOPs que agiliza o processo de disponibilização/publicação de novo código e garante que o que foi versionado não danificará o que já está implementado e publicado. Foram implementados testes automatizados para garantir integridade do código e assegurar alterações e novas implementações.

2.2) CONTROLE TRANSACIONAL

Como os serviços providos pela API implementada realizam operações em banco de dados, o controle das transações é fator importantíssimo. O controle transacional foi configurado e delegado para os cuidados do *Spring* (através da *annotation @Transactional*).

O *Spring* garante, através de seu controle de transações, que todas as operações estejam sob o conceito **ACID** (Atomicidade, consistência, isolamento, durabilidade). Assim, a consistência dos dados da biblioteca estão garantidos.

2.3) LOGGING COM AOP

É importante que a API forneça informações sobre os recursos que estão sendo acessados e utilizados pelos clientes. Apoiado à programação orientada a aspectos e utilizando o *framework AOP* do *Spring*, alguns comportamentos comuns foram identificados e modelados na classe LoggingHandler. Esta classe possui métodos que interceptam requisições e enviam para o console algumas informações utilizando SLF4J.

2.4) SPRING DATA

O *Spring Data* é um *framework* do ecossistema *Spring* que facilita a utilização da especificação JPA, encapsulando muitos de seus recursos.

Na API implementada, nenhum código SQL precisou ser escrito (de maneira explícita). Essa prática aumenta produtividade, acelerando o processo de trabalho em aplicações que utilizam banco de dados.

2.5) TESTES AUTOMATIZADOS

A escrita de testes é fator crucial para desenvolvimento de qualquer aplicação. Foram implementados testes de unidade e testes para checar a comunicação da aplicação com o banco de dados (*PostgreSQL*).

Cada regra de negócio da aplicação foi testada. Como foi adotada estratégia de total isolamento das regras, se tornou tarefa bem mais branda (e organizada) escrever os testes de unidade.

2.6) CONTROLE DE ERROS

Erros precisam ser controlados e - obviamente - transmitidos para os clientes. Foram definidas duas principais exceções: ResourceNotFoundException e BusinessException.

A primeira, ResourceNotFoundException, trata dos erros quando algum recurso não é encontrado. Foi definido que esta exceção retornará um HTTP *code* 400 (*Bad Request*). Abaixo dela, foram especializadas quatro exceções específicas. A segunda. trata dos erros de negócio (regras). Abaixo dela, foram especializadas sete exceções.

Já a BusinessException, trata dos erros de negócio (regras). Para ela foi definido como retorno o HTTP *code* 404 (*Not Found*). Abaixo dela, foram especializadas sete exceções.

Para a implementação dos testes automatizados foi utilizado *JUnit* e *Mockito*.

2.7) REGRAS DE NEGÓCIO SEPARADAS

Para ter melhor organização da aplicação e aumento da coesão, foi utilizada estratégia de separação de cada regra de negócio. Assim, conseguimos isolar nosso código, além de reduzir a quantidade de código na camada de *Service*.

Um outro ponto positivo é que esta estratégia facilita a escrita dos testes de unidade.

2.8) SWAGGER

Foi utilizada a ferramenta *Swagger* para documentar e auxiliar os clientes na utilização da API. A configuração pode ser conferida na classe SwaggerConfiguration.

Conforme informado acima, o *Swagger* da API pode ser acessado via: https://poll-restful-api.herokuapp.com/swagger-ui.html.

2.9) VERSIONAMENTO

É importante controlar as versões da biblioteca. S*oftwares* mudam. E com as demandas atuais, mudam com mais velocidade. Visando organização e garantia para os clientes a cada disponibilização de nova versão, a API apresentada foi versionada e é atendida via */api/v1*.

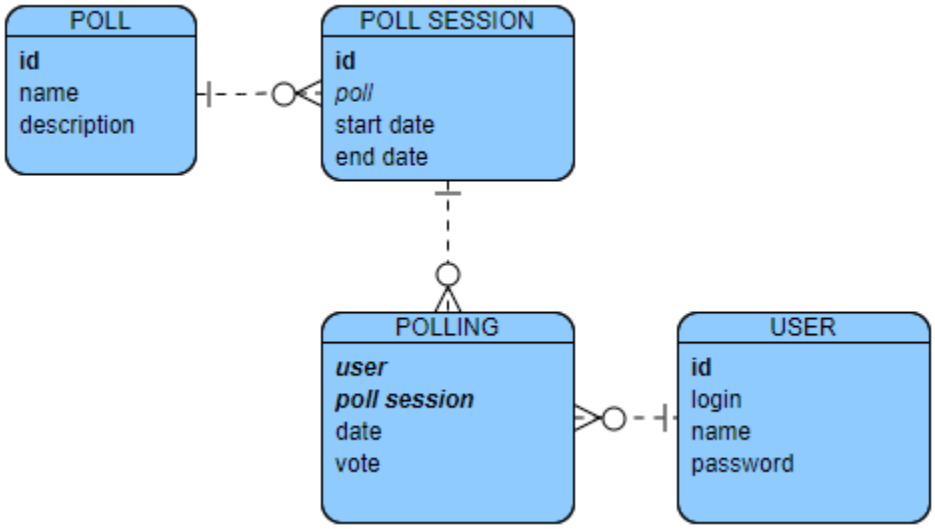
2.10) LOMBOK

A biblioteca *Lombok*, escrita em *Java*, foi utilizada como auxílio para implementação da API, aumento produtividade, velocidade e redução do código.

Vale destacar que o *Design Pattern* Builder foi utilizado apoiado ao *Lombok*, através da *annotation @Builder.*

3) MODELAGEM

A aplicação propõe a seguinte solução: existem quatro entidades para controlar a votação como um todo. São elas: **POLL**, **POLL-SESSION**, **POLLING** e **USER**. Cada uma delas será discutida abaixo. Serão apresentados exemplos para auxiliar:



3.1) POLL

A entidade **POLL** trata-se de um *template* de uma votação. Neste domínio será registrado o **nome** a **descrição** da pauta. O **nome** é um registro único da pauta, enquanto que a **descrição** trata-se do tópico que será votado pelos usuários.

```
POLL {
  "name": "Votação Maioridade Penal",
  "description": "Você é a favor da redução da maioridade penal?"
}
```

Comentário: perceba que nenhuma data para execução da votação foi informado. Datas de execução e duração da votação são controlados pela entidade **POLL-SESSION**.

3.2) POLL SESSION

A entidade **POLL-SESSION** trata-se de uma representação de uma sessão para uma votação (**POLL**). A **POLL-SESSION**, quando aberta, precisa apontar para uma **POLL**.

Ela funciona como uma instância da votação. Nesta entidade são armazenadas as **datas de início** e **fim da sessão**. A sessão de votação é iniciada no momento em que é criada. Já a **data de encerramento** depende da duração (que pode ser escolhida pelo cliente). Caso o cliente não escolha uma duração específica, 60 segundos será considerado.

```
POLL SESSION {
    "poll": POLL
    "startDate": "10/04/2019"
    "endDate":  "10/06/2019"
}
```

Comentário: a sessão de votação, enquanto aberta, poderá receber votos dos associados (usuários). Os votos são controlados pela entidade **POLLING**.

Vale também frisar que os resultados são em cima das sessões. Os resultados da votação da sessão só podem ser mostrados caso a sessão esteja encerrada. Caso algum cliente faça uma requisição para visualização de resultados para uma sessão ainda aberta, a exceção PollSessionRunningException será lançada.

3.3) POLLING

A entidade **POLLING** é a representação de um voto em si. Um usuário cadastrado no sistema poderá votar - uma, e somente uma, vez - em qualquer sessão (**POLL-SESSION**) aberta de votação (**POLL**).

Importante frisar que o voto é controlado pelo *Enum* VoteOption. Perceba que, como as opções de voto são apenas sim/não, poderíamos mapear o voto como um tipo *booleano*. O grande problema desta abordagem é que perderíamos escalabilidade, visto que novas opções de voto podem ser criadas no futuro. Da maneira como foi implementado, a aplicação estaria pronta para receber novos opções de voto.

```
POLLING {
    "pollSession": POLL SESSION
    "user": USER,
    "vote": "YES"
}
```

3.4) USER

A entidade **USER** é a representação de um associado. Todos os associados podem votar em qualquer sessão (**POLL-SESSION**) aberta.

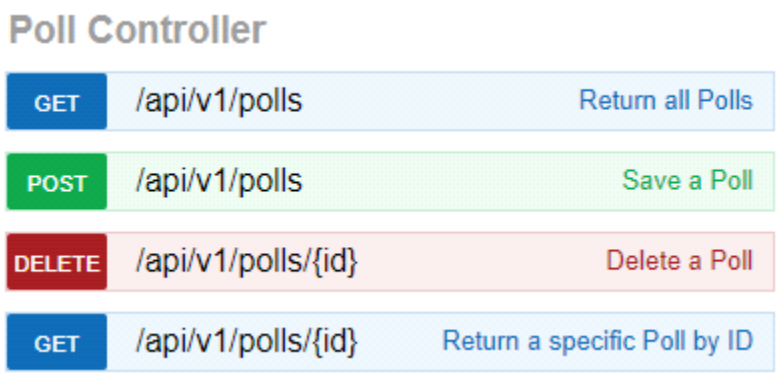
Para testes, foram adicionados 10 associados *default*. Vale frisar que existe um *endpoint* para criação de um associado na aplicação. Em um mundo real de produção, provavelmente não seria necessário a disponibilização da possibilidade de inclusão de usuário pelos clientes externos. Para fins de testes, esta opção foi considerada na API implementada.

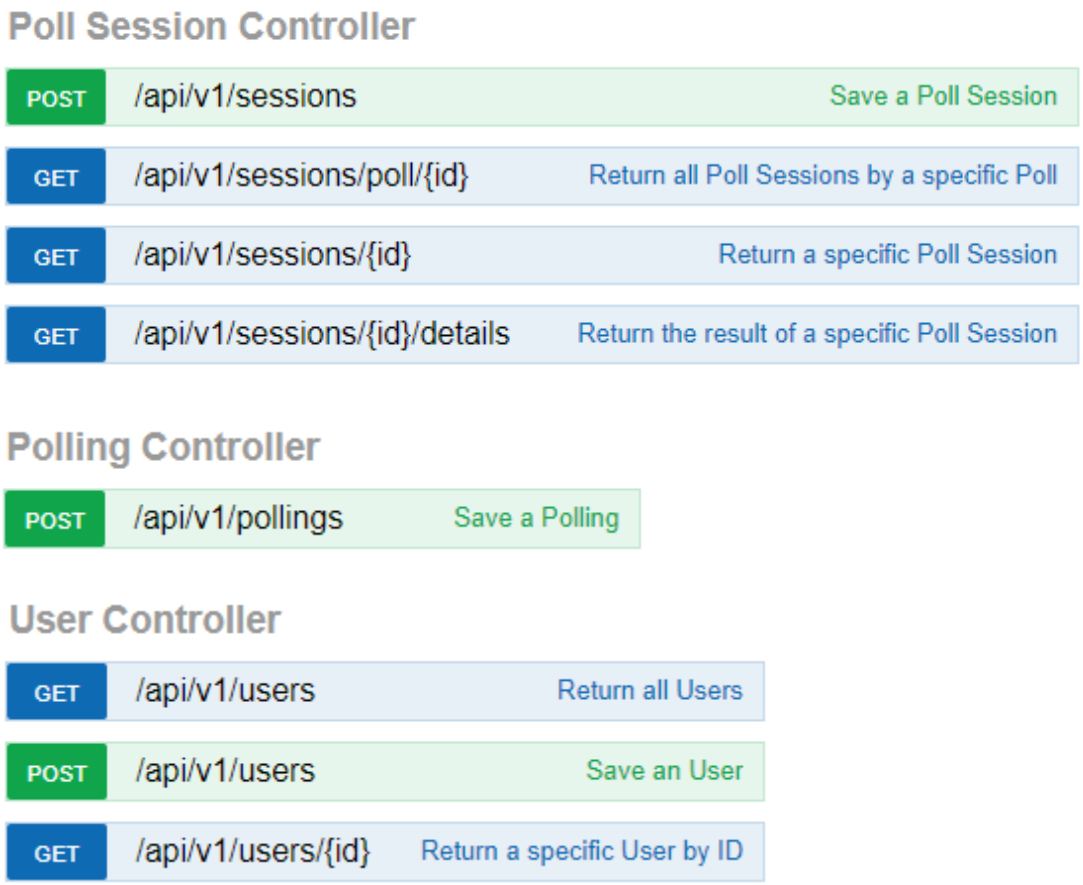
```
USER {
    "name": "Paul McCartney",
    "login": "paulmc"
    "password" : "blackbird123"
}
```

Comentário: *DTOs* foram implementados para expor as informações para os clientes externos. Não é uma boa prática expor as entidades como saída da API. Utilizando *DTOs*, conseguimos omitir o **password** dos usuários para os clientes externos, por exemplo. Com esta estratégia, podemos definir quais dados serão expostos, preservando os que forem desnecessários/sigilosos para apresentação.

4) REQUISIÇÕES

Abaixo serão mostrados cada um dos *endpoints* da API. Esta informação está presente na documentação *Swagger* da API. De qualquer maneira, se faz útil aparecer também neste documento:





5) CONCLUSÃO

Após finalização do desenvolvimento da API, alguns pontos suscetíveis a melhoria. Eles seguem abaixo:

- SEGURANÇA: para o teste técnico, foi pedido que a segurança da API fosse desconsiderada. Uma boa evolução seria a implantação de segurança utilizando JWT, por exemplo. Além de ter fácil integração com o *Spring*, trata-se de um padrão extremamente seguro e leve (muito por conta da utilização de JSON).

- HATEOAS: principalmente para o caso de a API evoluir e novos serviços/*endpoints* forem desenvolvidos. Com esta prática, são retornados também *links* (e não apenas os recursos). Desta maneira, fornecemos maiores informações para os clientes, facilitando o acesso e navegação entre os recursos.

- SPRING WEBFLUX: desde da versão 5 do *Spring*, o *framework* fornece possibilidade de trabalhar com programação assíncrona e não-bloqueante. Desta maneira, preparamos nossa API pra responder às requições de maneira orientada a eventos.

- Validação DTOs de Request: a API valida os dados enviados pelo Request dos clientes, mas deveria apresentar melhor o retorno. Uma boa prática seria implementar uma classe para encapsular as mensagens e retornar ao cliente.