DEVinHouse

Módulo 3 - Projeto Avaliativo 1

SUMÁRIO

1 INTRODUÇÃO

A **LABSky Linhas Aéreas** entrou em operação com uma aeronave para atender a um grupo seleto de clientes que fazem o trecho Florianópolis/SC - Santa Maria/RS diariamente. Você foi escolhido para criar o **back-end** de uma aplicação para gerenciar os passageiros que irão no voo.

A aplicação deve ser uma **API REST** desenvolvida em **Java** com **Spring Boot**, e que atenda aos requisitos elencados neste documento - **principalmente os quesitos de testes** - para controle de quais passageiros confirmaram (realizaram check-in) para o voo e qual assento foi selecionado.

2 ENTREGA

O código desenvolvido deverá ser submetido no **GitHub**, e o link deverá ser disponibilizado junto com o link do **Trello** na tarefa **Módulo 3 - Projeto Avaliativo 1**, presente na semana **5** do **AVA** até o dia **04/06/2023** às **23h55**.

O repositório deverá ser **privado**, com as seguintes pessoas adicionadas:

- Ronyeri Marinho de Souza Almeida - **ronny-souza**
- Operação DEVinHouse - **devinhouse-operacao**

Importante:

1. Não serão aceitos projetos submetidos **após a data limite da atividade**, e, ou **alterados** depois de entregues. Lembre-se de **não modificar** o código no GitHub até receber sua nota e feedback.
2. Não esqueça de **submeter o link no AVA**. Não serão aceitos projetos em que os links não tenham sido submetidos.

3 ROTEIRO DA APLICAÇÃO

Para fins didáticos e para facilitar as configurações, utilize um Banco de Dados embarcado (H2) na aplicação (veja as dicas 1 e 2 de configuração na seção de Dicas).

A empresa também tem um "programa de fidelidade" para os clientes, onde a cada voo o passageiro acumula milhas que podem posteriormente ser trocados por brindes ou descontos na passagem.

O **Passageiro** deve ter os seguintes campos pessoais cadastrados:

- CPF: Tipo String; Identificador do passageiro no sistema
- Nome: Tipo String
- Data de Nascimento: Tipo LocalDate
- Classificação: Enum ou String; Categoria do passageiro no programa de fidelidade
- Milhas: Tipo Integer; Quantidade acumulada de milhas do passageiro

As classificações de um passageiro no programa de fidelidade da empresa podem ser as seguintes: VIP, OURO, PRATA, BRONZE, ASSOCIADO.

Todo cliente da empresa começa na classificação 'Associado' e conforme vai acumulando milhas vai evoluindo até chegar no nível 'VIP'.

A aeronave usada pela companhia possui 10 fileiras de assentos numeradas, com 6 poltronas cada, ordenadas de ‘A’ a ‘F’. Portanto, existem os seguintes assentos:

1A, 1B, 1C, 1D, 1E, 1F,
2A, 2B, 2C, 2D, 2E, 2F,
3A, 3B, 3C, 3D, 3E, 3F,
...
10A, 10B, 10C, 10D, 10E, 10F

Quando o passageiro confirma o voo (realiza o *check-in*), é gerada uma “*Confirmação*” (bilhete de embarque) com os novos campos:

- E-ticket: Tipo String; Identificador da confirmação; Gerado com UUID
- Assento: Tipo String; Assento escolhido pelo passageiro
- Malas Despachadas: Tipo Boolean; Indicador se serão despachadas malas
- Data e Hora da Confirmação: Tipo LocalDateTime; Data e hora

A aplicação deve atender a todos os requisitos elencados abaixo, atendendo também as regras de negócio previstas. Deve-se atentar para se o contrato da API está sendo atendido inclusive em seus detalhes, tais como HTTP status code, nome dos campos de request/response e etc.

Deve ser possível, portanto, consultar todos os passageiros do voo, confirmados com check-in ou não, trazendo os dados do check-in no caso de ter sido realizado.
A confirmação/check-in deve atender todas as regras de negócio previstas, devendo ser indicado o motivo do erro para caso de falha na solicitação.

A aplicação deve contemplar **testes unitários** e prever as boas práticas de programação, respeitando as responsabilidade das classes e camadas/packages criadas.

O código-fonte deve estar disponibilizado no GitHub, usando *feature branchs* para desenvolver cada requisito previsto, ou seja, cada requisito deve ser desenvolvido num branch a parte (exemplo “feature/req02_consulta_passageiros”) e depois mergeado no branch main. O código final deve estar todo “mergeado” no branch “main”. Os branchs de feature devem permanecer no GitHub para conferência (não apagar após o merge).

4 REQUISITOS DA APLICAÇÃO

A aplicação que deverá ser realizada **individualmente** deve contemplar os seguintes requisitos:

1. Carga inicial de dados:
A aplicação deve ser inicializada com os dados iniciais dos passageiros conforme listagem abaixo:

CPF	Nome	Data de Nasc.	Classificação	Milhas
000.000.000-00	Rachel Green	11/01/1969	VIP	100
111.111.111-11	Phoebe Buffay	30/07/1963	OURO	100
222.222.222-22	Ross Geller	02/11/1966	PRATA	100
333.333.333-33	Monica Geller	15/06/1964	OURO	100
444.444.444-44	Chandler Bing	19/08/1969	OURO	100
555.555.555-55	Joey Tribbiani	25/07/1967	BRONZE	100
666.666.666-66	Mike Hannigan	06/04/1969	VIP	100
777.777.777-77	Gunther Tyler	28/05/1962	ASSOCIADO	100
888.888.888-88	Janice Wheeler	07/08/1961	BRONZE	75
999.999.999-99	Richard Burke	29/01/1945	BRONZE	50

Para a carga inicial de dados, escrever um script SQL de inserção (*data.sql*) e adicioná-lo na aplicação (dentro da pasta resources - ver dica 3 na seção de Dicas).

2. Consulta de Dados completos de Passageiros:
Deve retornar todos os passageiros registrados com os dados disponíveis.
Requisição: HTTP GET → /api/passageiros
Resposta: HTTP 200-OK com lista de passageiros contendo os campos: *cpf, nome, dataNascimento, classificação, milhas, eticket, assento, dataHoraConfirmacao*. Os campos que não estiverem preenchidos podem ser nulos ou omitidos no response body. No response os campos devem ter a nomenclatura indicada neste requisito.

Exemplo de response:

```
[
        {
        "cpf": 000.000.000-00,
        "nome": "Rachel Green",
        "dataNascimento": "11/01/1969",
        "classificacao": "VIP",
        "milhas": 100,
        "eticket": null,
        "assento": null,
        "dataHoraConfirmacao": null
        },
        {
        "cpf": 111.111.111-11,
        "nome": "Phoebe Buffay",
        "dataNascimento": "30/07/1963",
        "classificacao": "OURO",
        "milhas": 100,
        "eticket": null,
        "assento": null,
        "dataHoraConfirmacao": null
        },
 ...
]
```

3. Consulta de Passageiro por CPF:

Deve retornar o passageiro registrado com o CPF informado.

Requisição: HTTP GET → /api/passageiros/{cpf}

Resposta:

- Sucesso: HTTP 200-OK com dados do passageiro contendo os campos: *cpf, nome, dataNascimento, classificação, milhas*. No response os campos devem ter a nomenclatura indicada neste requisito.
- Registro não encontrado: HTTP 404 - NOT FOUND com mensagem de erro.

Exemplo de response (sucesso):

```
{
        "cpf": 000.000.000-00,
        "nome": "Rachel Green",
        "dataNascimento": "11/01/1969",
        "classificacao": "VIP",
        "milhas": 100
}
```

4. Consulta de Assentos:
Deve retornar todos os assentos da aeronave.
Requisição: HTTP GET → /api/assentos
Resposta: HTTP 200 - OK com lista de assentos (lista de Strings).

Exemplo de response:

```
[
        "1A",
        "1B",
        "1C",
        "1D",
        "1E",
       ...
        "10D",
        "10E",
        "10F"
]
```

5. Realizar confirmação (check-in):
Deve registrar os dados de confirmação do voo (check-in), reservando o assento para o passageiro e gerando o e-ticket.
Devem ser atendidas as seguintes Regras de Negócio:

1. Se não houver passageiro registrado com o CPF informado na requisição, deve retornar mensagem de erro e HTTP Status Code 404 - Not Found.
2. Se o assento informado na requisição não existir, deve retornar mensagem de erro e HTTP Status Code 404 - Not Found.
3. Se o assento indicado já estiver sido reservado para outro passageiro, deve retornar mensagem de erro informativa e com HTTP Status Code 409 - Conflict.
4. As fileiras "5" e "6" da aeronave são consideradas "fileiras de emergência" (5A, 5B, 5C, 5D ... 6D, 6E, 6F) por estarem próximo à asa do avião, e os ocupantes destas fileiras devem estar aptos a realizar os procedimentos de emergência. Desta forma, se o passageiro for menor de idade (idade menor que 18 anos), ele não poderá escolher assentos destas duas fileiras. Portanto, caso o passageiro seja menor de idade e tenha escolhido uma das fileiras de emergência, deve retornar mensagem de erro informativa e HTTP Status Code 400 - Bad Request.
5. Como as fileiras "5" e "6" da aeronave são consideradas "fileiras de emergência" estas tem layout diferenciado e não comportam bagagens pessoais trazidas pelos passageiros, os quais, se escolherem alguma das poltronas destas fileiras, devem obrigatoriamente despachar suas malas. Caso o passageiro tenha escolhido uma das fileiras de emergência e não tenha despachado as malas, deve retornar mensagem de erro informativa e HTTP Status Code 400 - Bad Request.

6. Quando é feito o registro da confirmação (check-in), deve ser gerado um valor de e-ticket usando o UUID (ver dica e na seção de Dicas).
7. No momento da confirmação, também deve ser calculado o novo saldo de milhas do passageiro, de acordo com a classificação/categoria dele:
    a. VIP → + 100 pontos
    b. OURO → + 80 pontos
    c. PRATA → + 50 pontos
    d. BRONZE → + 30 pontos
    e. ASSOCIADO → + 10 pontos
8. Deve ser gerado log (no console) com a frase: "*Confirmação feita pelo passageiro de CPF xxx.xxx.xxx-xx com e-ticket xxxxxxxxxx*".

Requisição: HTTP POST → /api/passageiros/confirmacao
Campos da requisição: *cpf, assento, malasDespachadas*. Os campos devem ter a nomenclatura indicada. Todos os campos da Request são obrigatórios!

Exemplo de Request:

```
{
        "cpf": 222.222.222-22,
        "assento": "5B",
        "malasDespachadas": true
}
```

Resposta: HTTP 200-OK com dados da confirmação contendo os campos: *eticket, dataHoraConfirmacao*. No response os campos devem ter a nomenclatura indicada neste requisito.

Exemplo de response:

```
{
        "eticket": "89cce4b6-d315-4382-9652-500c61cde768",
        "dataHoraConfirmacao": "25/05/2023 15:24:12"
}
```

5 CRITÉRIOS DE AVALIAÇÃO

A tabela abaixo apresenta os critérios que serão avaliados durante a correção do projeto. O mesmo possui variação de nota de 0 (zero) a 10 (dez) como nota mínima e máxima, e possui peso de **40% sobre a avaliação do módulo**.

Serão **desconsiderados e atribuída a nota 0 (zero)** os projetos que apresentarem plágio de soluções encontradas na internet ou de outros colegas. Lembre-se: Você está livre para utilizar outras soluções como base, mas **não é permitida** a cópia.

Nº	Critério de Avaliação	0	0,5	1
1	Aluno implementou a carga inicial de dados (Item 1)	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
2	Aluno implementou consulta de Passageiros e Confirmações (Item 2), conforme especificação do requisito.	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
3	Aluno implementou consulta de Passageiros por CPF (Item 3), conforme especificação do requisito.	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
4	Aluno implementou consulta de Assentos (Item 4), conforme especificação do requisito.	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
5	Aluno implementou Confirmação (check-in) de voo (Item 5) com validação de campos obrigatórios, e validação de passageiro ou assento não cadastrado (Regra 1 e Regra 2).	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
6	Aluno implementou Confirmação (check-in) de voo (Item 5), verificando se assento já alocado (Regra 3); e em caso de fileira de emergência validar se passageiro maior de idade e se despachou malas (Regra 4 e Regra 5).	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
7	Aluno implementou Confirmação (check-in) de voo (Item 5) com sucesso	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.

	gerando o valor de e-ticket (Item 6), atualizando o valor de milhas do passageiro (Regra 7) e logando no console (Regra 8) a mensagem descrito no requisito.			
8	Aluno implementou testes unitários da camada de serviço (Service), com cobertura de testes adequada e cobrindo as regras de negócio previstas; e também na camada de Controller (com pelo menos um teste por endpoint existente)	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
9	Aluno aplicou conceitos de clean code e boas práticas de código, respeitando as responsabilidade das camadas e implementando um código de fácil manutenção.	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.
10	Aluno usou feature branchs para desenvolver cada requisito, mergeando o código final no branch main do GitHub, bem como usou mensagens de commit adequadas.	Aluno não implementou.	Aluno implementou parcialmente.	Aluno implementou corretamente.

6 PLANO DE PROJETO

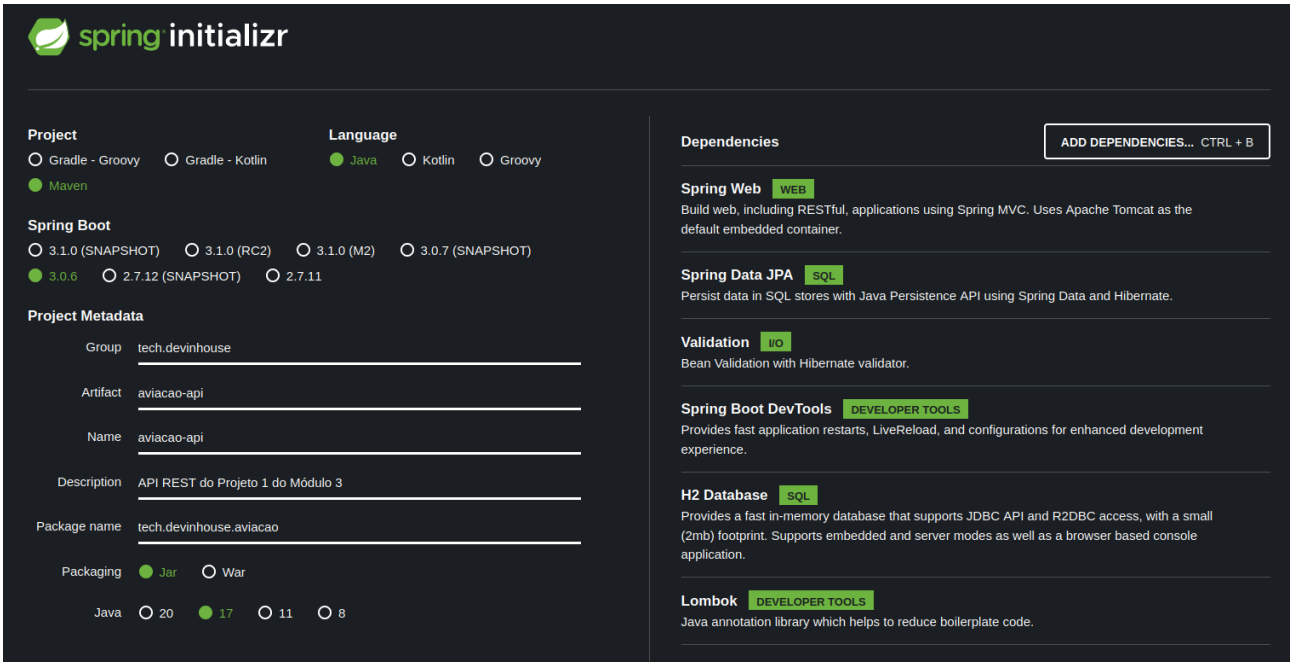
Ao construir a aplicação proposta, o aluno estará colocando em prática os aprendizados em:

- **REST API**: Implementação de serviços REST para atender requisições HTTP;
- **SQL**: Construção de comandos SQL para povoamento de base de dados relacional;
- **Spring Boot**: Utilização do framework de desenvolvimento;
- **Testes Unitários**: Implementação de testes unitários para garantir o funcionamento adequado do código-fonte desenvolvido;
- **Clean Code e boas práticas**: aplicação de boas práticas de implementação e organização de código-fonte no projeto.

7 DICAS

7.1 Configuração inicial da aplicação Spring Boot:

Sugere-se o uso da criação do projeto com o Spring Initializr com as dependências vistas em aulas, conforme exemplo sugerido a seguir:



Um exemplo de como ficaria a seção de dependências no “pom.xml” completo ficaria conforme abaixo:

```
...
  <dependencies>
        <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-devtools</artifactId>
              <scope>runtime</scope>
              <optional>true</optional>
        </dependency>
        <dependency>
```

```
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-ui</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.modelmapper</groupId>
            <artifactId>modelmapper</artifactId>
            <version>3.0.0</version>
        </dependency>
    </dependencies>
...
```

7.2 Configuração do banco de dados embarcado H2:

Quando é adicionada a dependência do banco de dados H2 no "pom.xml", automaticamente a aplicação passa a usar este banco de dados de forma embarcada.

Atenção: a cada re-deploy da aplicação o Banco de Dados também é reiniciado e os dados armazenados são perdidos.

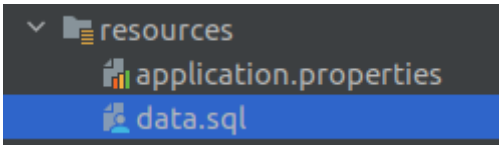
7.3 Configuração de arquivo de carga de dados inicial:

O Spring Boot executa automaticamente o arquivo "*data.sql*" com comandos SQL (DML) que for adicionada dentro da pasta "*resources*". Portanto, basta escrever os comandos SQL e gravá-los neste arquivo para inserção dos dados iniciais.
Nas últimas versões do Spring Boot, é necessário adicionar algumas configurações no "*application.properties*" para esta execução dos comandos SQL funcionar na inicialização:

Arquivo application.properties:

```
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

Estrutura da pasta “resources”:



7.4 Utilização do UUID:

Para gerar um identificador único, não é necessário adicionar nenhuma outra biblioteca como dependência. Basta importar a classe java.util.UUID.

Exemplo de geração:

```
String eticket = UUID.randomUUID().toString();
```