Sistema Extensível de Gestão de Dados Mestres

Bruno Laitano, Daniel Lee e Pedro Dorneles

Engenharia de Software II

1 Introdução

Com uma arquitetura de integração eficiente, extensível e flexível, desenvolvemos um sistema *Master Data Management* capaz de consolidar, padronizar e garantir a qualidade e a governança de dados mestres provenientes da API pública *REST Countries*, com dezenas de dados geográficos a respeito de 250 países do globo.

Para tanto, idealizamos dois serviços distintos: o *Master Data Management* (MDM) e o *Data Extraction Management* (DEM). O MDM é responsável por armazenar, validar e padronizar os dados mestres. Ele provê uma API *RESTful* própria que permite operações administrativas sobre os registros, possibilitando a criação de novos dados, a recuperação de informações com base em critérios específicos, a atualização de registros existentes e a exclusão de dados obsoletos ou redundantes, mantendo um repositório central confiável e estruturado.

Por outro lado, o DEM tem como função principal executar operações de ETL (Extração, Transformação e Carga) a partir da API *REST Countries*. Esse serviço realiza a extração dos dados geográficos, aplica transformações para padronizar as informações conforme critérios pré-definidos e, em seguida, carrega os dados transformados em um arquivo `.json`. Além disso, disponibiliza APIs *RESTful* próprias que permitem não apenas monitorar as transações realizadas, mas também expondo ao outro serviço a nova coleção de dados baseada na original.

Do ponto de vista arquitetural, utilizamos H2 e JPA para estruturar o nosso projeto. O H2 é um banco de dados relacional capaz de facilitar a configuração e o *reset* do ambiente a cada execução. Já o JPA é uma especificação para mapeamento objeto-relacional, permitindo que entidades Java sejam persistidas e consultadas no banco de dados, sem a necessidade de escrever comando SQL manualmente.

2 *Data Extraction Management*

O serviço extrator é baseado na extração dos dados "crus" (`raw`) da API *REST Countries*, na transformação desses dados em uma nova coleção, com menos atributos, e no registro local dos dados curados (`bronze`). Além disso, um segmento específico do

serviço é dedicado a preparar metadados a respeito das informações coletadas durante a extração.

2.1 ETL: *Extract, Transform, Load*

A leitura da API pública é realizada com `RestTemplate`, que fornece dezenas de operações abrangendo todos os parâmetros de requisição HTTP. Essa estrutura é especialmente utilizada no contexto de uma classe `ExtractService`.

O serviço de extração, além de consumir a API, registra uma cópia local em formato `.json` dos dados geográficos coletados. Isso é realizado por meio de uma operação `saveRaw()`, que nomeia cada arquivo coletado a partir do tempo e da data de extração, garantindo que os registros locais possam ser mapeados em função do momento em que foram extraídos.

Uma vez extraídos, os dados "crus" são transformados com base em atributos previamente selecionados. A transformação se dá sobre duas entidades: `Country` e `Currency`, sendo esta última uma propriedade específica da primeira. Do ponto de vista do banco de dados, ambas estão interrelacionadas por meio de um par chave primária (`PK`)/chave estrangeira (`FK`).

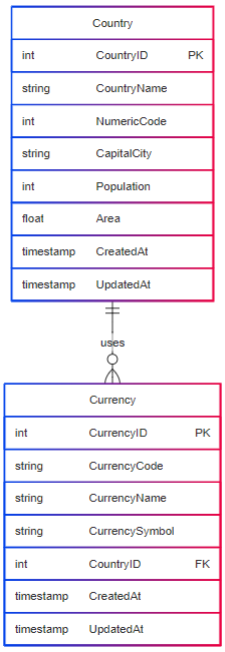

Figura 1: Esquema original na requisição do projeto

Foram adicionados ao esquema de `Country` novos atributos em relação à proposta original. Abaixo, você poderá ver, no contexto da classe/entidade dedicada à figura do país, quais foram as propriedades incluídas no design. Os atributos de `Currency` foram mantidos como o exigido no esquema original. Além disso, atribuímos a cada objeto um identificador único utilizando a classe `UUID`.

```java
public class Country {
    private String id;
    private String commonName;
    private Boolean independent;
    private Boolean unMember;
    private List<Currency> currencies;
    private String capital;
    private String region;
    private String languages;
    private String latlng;
    private String borders;
    private Double area;
    private Long population;
    private String gini;
    private String timezones;
    private String continents;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "
        yyyy-MM-dd'T'HH:mm:ss")
    private Timestamp createdAt;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "
        yyyy-MM-dd'T'HH:mm:ss")
    private Timestamp updatedAt;
}
```

As informações curadas são salvas localmente em um arquivo no formato `.json`, também nomeado a partir do momento do registro, por meio de uma operação `saveBronze()`. Os arquivos curados são classificados como `bronze`. A ideia de nomeá-los de tal forma deve-se ao fato de que, na hipótese de novos refatoramentos das informações a serem coletadas, os dados podem ser transformados de forma cada vez mais precisa (`silver`, `gold`...).

Finalmente, um `DemController` expõe uma nova API ao outro serviço que compõe este projeto, garantindo que os dados transformados também possam ser acessados livremente. Abaixo, indicamos a função do tipo `@GetMapping` responsável por publicizar essas informações.

```java
@GetMapping("")
    public ResponseEntity<String> getCountries() {
        String bronzeData = data.getBronzeData();
        if (bronzeData == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                    .contentType(MediaType.APPLICATION_JSON)
                    .body("{\"error\":\"No bronze data
                        available.\"}");
        }
        return ResponseEntity.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(bronzeData);
```

```
12      }
```

2.2 Metadados

Os metadados a respeito de cada extração são salvos com base em três parâmetros: o nome da fonte (`provider`), a quantidade de elementos salvos (`count`) e a última atualização recebida (`lastUpdate`). Criamos tanto um repositório quanto um controlador para que os metadados possam ser registrados localmente e expostos em um novo *endpoint*.

A título de exemplo, realizada uma extração, é este o tipo de resultado que poderá ser conferido pelo administrador:

```
[
    {
        "provider": "RestCountries API",
        "count": 250,
        "lastUpdate": "2025-05-25_12-19-10"
    }
]
```

3 *Master Data Management*

O módulo MDM foi desenvolvido com o objetivo de gerenciar dados mestres relacionados a países e moedas, proporcionando operações de criação, consulta, atualização e remoção através de uma API *RESTful* implementada com *Spring Boot*. A seguir, são descritas as principais classes que compõem esse módulo.

3.1 `CountryController`

O `CountryController` provê a API *RESTful* para o gerenciamento das entidades de país (`CountryEntity`), cumprindo diretamente os requisitos de criação, consulta, atualização e remoção definidos na especificação do microserviço MDM.

Table 1: Endpoints da classe `CountryController`

Endpoint	Método HTTP	Descrição
`/mdm/countries/all`	GET	Retorna a lista completa de países cadastrados, via `CountryService.getAll()`
`/mdm/countries/id/{id}`	GET	Busca país por identificador UUID, retornando `200 OK` ou `404 Not Found`
`/mdm/countries/name/{name}`	GET	Busca país pelo nome comum (`commonName`), retornando `200 OK` ou `404 Not Found`
`/mdm/countries/capital/{capital}`	GET	Busca país pela capital
`/mdm/countries/region/{region}`	GET	Filtra países por região
`/mdm/countries/borders/{border}`	GET	Retorna todos os países que fazem fronteira com o código ISO especificado
`/mdm/countries/create`	POST	Cria um novo registro de país, recebendo JSON com todos os atributos de `CountryEntity`
`/mdm/countries/update/{id}`	PATCH	Atualiza parcialmente um país existente, através de um `Map<String,Object>` de campos
`/mdm/countries/delete/{id}`	DELETE	Remove o país identificado por `id`

3.1.1 Detalhes de implementação

- **Injeção de dependências**: o controlador recebe via construtor o `CountryService`, responsável pela lógica de negócio, e o `CountryRepository`, que executa operações de persistência;
- **Tratamento de respostas**: métodos que retornam um `Optional` são encapsulados com `ResponseEntity`, garantindo retornos apropriados como `200 OK` ou `404 Not Found`;
- **Consultas personalizadas**: filtros por região e fronteiras são implementados utilizando `Stream API` sobre a lista completa de países.

3.1.2 Exemplo de *payload* para a criação de um novo país

`POST /mdm/countries/create`

```
{
  "id": "b3f5c8e2-9f1e-4a7a-8d8f-2a4e5c6d7b8c",
  "commonName": "Brazil",
  "independent": true,
  "unMember": true,
  "currencies": [
    {
      "id": "e2f1a9d8-3b4c-5e6f-7a8b-9c0d1e2f3a4b",
      "currencyCode": "BRL",
      "name": "Brazilian real",
      "symbol": "R$"
    }
  ],
  "capital": "Brasilia",
```

```
  "region": "Americas",
  "languages": "Portuguese",
  "latlng": "[-15.793889, -47.882778]",
  "borders": ["ARG","BOL","COL","GUF","GUY","PRY","PER","SUR","URY
     ","VEN"],
  "area": 8515767.0,
  "population": 212559409,
  "gini": "53.4",
  "timezones": ["UTC-05:00","UTC-04:00","UTC-03:00"],
  "continents": ["South America"]
}
```

3.1.3 Exemplo de *payload* para atualização parcial

`PATCH /mdm/countries/update/b3f5c8e2-9f1e-4a7a-8d8f-2a4e5c6d7b8c`

```
{
  "population": 213000000,
  "gini": "54.7"
}
```

3.2 `CurrencyController`

O `CurrencyController` fornece a API *RESTful* para o gerenciamento de moedas (`CurrencyEntity`), completando o escopo de domínio financeiro do MDM.

Table 2: Endpoints da classe `CurrencyController`

Endpoint	Método HTTP	Descrição
`/mdm/currencies/all`	GET	Lista todas as moedas disponíveis no repositório
`/mdm/currencies/id/{id}`	GET	Retorna moeda por UUID, com `200 OK` ou `404 Not Found`
`/mdm/currencies/code/{code}`	GET	Busca moeda pelo código ISO (e.g. `"USD"`), retornando um `Optional<CurrencyEntity>`
`/mdm/currencies/create`	POST	Cria uma nova moeda, recebendo JSON com atributos de `CurrencyEntity`
`/mdm/currencies/update/{id}`	PATCH	Atualiza parcialmente uma moeda existente
`/mdm/currencies/delete/{id}`	DELETE	Remove a moeda pelo identificador

3.2.1 Detalhes de implementação

- **Consultas**: busca por código de moeda (`getByCode`) retornando `Optional`, permitindo tratamento seguro de resultados;
- **Responsabilidade**: validação de formatos (ISO 4217) e regras de negócio são realizadas na camada de serviço (`CurrencyService`);
- **Padrão *RESTful***: todos os métodos seguem boas práticas REST, garantindo consistência com demais serviços.

3.2.2 Exemplo de *payload* para criação de uma nova *currency*

`POST /mdm/currencies/create`

```
{
    "currencyCode": "FCR",
    "currencyName": "Fictional currency",
    "currencySymbol": "Fcr"
}
```

4 Integração entre DEM e MDM

A forma como definimos a integração entre ambos os serviços implementados envolve dois segmentos criados no contexto de MDM. Em `CountryService`, um método com a anotação `@PostConstruct`, sendo executado logo após a conclusão da injeção de dependências, é responsável por permitir o acesso de MDM aos dados transformados (`bronze`) em DEM. O método foi chamado de `syncWithDem()`.

A razão pela qual optamos por permitir ao MDM acessar os dados transformados é que, dessa forma, as operações `CRUD` poderiam ser realizadas sobre essas informações, facilitando os testes realizados e garantindo a integridade das operações. Essas operações também são implementadas no contexto da classe `CountryService`.

A classe `IntegrationService` foi criada justamente com o intuito de implementar o método `fetchCountriesFromDem()`, que consome o *endpoint* exposto pelo serviço extrator, por meio do qual a coleção de dados transformados pode ser acessada.

```
public List<CountryEntity> fetchCountriesFromDem() {
        int maxAttempts = 10;
        int delayMs = 3000;
        for (int attempt = 1; attempt <= maxAttempts; attempt
           ++) {
            try {
                String url = "http://dem:8080/dem/countries";
                CountryEntity[] countriesArray = restTemplate.
                   getForObject(url, CountryEntity[].class);
                return Arrays.stream(countriesArray).collect(
                   Collectors.toList());
            } catch (ResourceAccessException e) {
                System.err.println("Attempt " + attempt + "
                   failed: " + e.getMessage());
                if (attempt == maxAttempts) {
                    System.err.println("Could not connect to
                       DEM after " + maxAttempts + " attempts.
                       ");
                    return List.of();
                }
                try {
```

```
16                      Thread.sleep(delayMs);
17                  } catch (InterruptedException ie) {
18                      Thread.currentThread().interrupt();
19                      return List.of();
20                  }
21              }
22          }
23          return List.of();
24      }
```

5 Uso da plataforma *Docker*

O uso do *Docker* neste projeto proporcionou um ambiente padronizado para a execução dos serviços, facilitando tanto o desenvolvimento quanto o *deploy*. Tanto o *Data Extraction Management* quanto o *Master Data Management* possuem os seus próprios arquivos `Dockerfile`, que definem todas as instruções necessárias para construir a imagem de cada serviço.

Além disso, a utilização de um arquivo `docker-compose` permite orquestrar de forma otimizada a execução conjunta dos serviços. Com um único arquivo de configuração, é possível definir como os *containers* do DEM e do MDM devem ser construídos, conectados e inicializados. Dessa forma, a implantação do sistema é mais simples, bastando um único comando para subir toda a infraestrutura necessária para o funcionamento do sistema.

```
1  services:
2      dem:
3        build:
4          context: ./dem
5          dockerfile: dockerfile
6        volumes:
7          - ./files:/dem/files
8        ports:
9          - 8081:8080
10
11     mdm:
12       build:
13         context: ./mdm
14         dockerfile: dockerfile
15       depends_on:
16         - dem
17       ports:
18         - 8082:8082
```