

N_PROG SIST II_A7 - Texto de apoio

Site: [EAD Mackenzie](#)

Tema: PROGRAMAÇÃO DE SISTEMAS II {TURMA 03B} 2023/1

Livro: N_PROG SIST II_A7 - Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .

Data: sexta, 5 mai 2023, 01:20

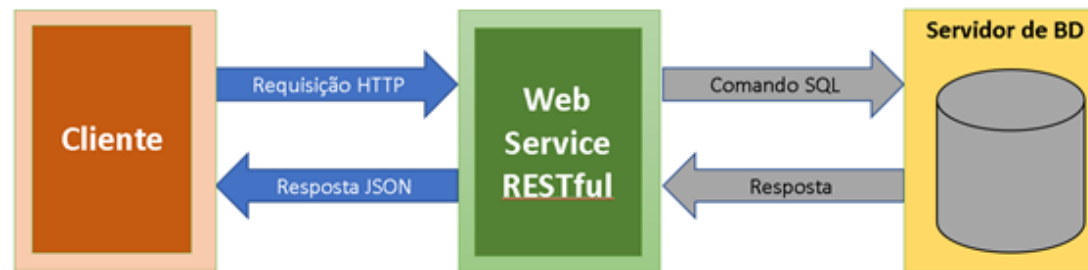
Descrição

Índice

1. WEB SERVICE RESTFUL COM ACESSO A BASE DE DADOS

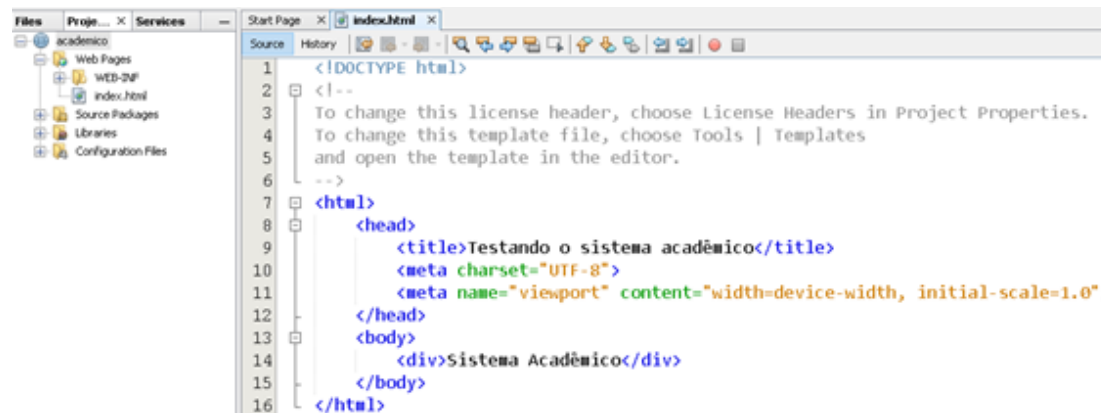
1. WEB SERVICE RESTFUL COM ACESSO A BASE DE DADOS

Na Aula 4, você aprendeu como encapsular as operações de acesso ao banco de dados utilizando os padrões de projeto (Design Pattern) DAO (Data Access Object). Nas Aulas 5 e 6, foram apresentados como testar e implementar um Web Service seguindo o padrão REST. Com esses conhecimentos adquiridos, desenvolveremos um Web Service RESTful com persistência em uma base de dados. A imagem abaixo ilustra o que está sendo proposto. Observe que na figura temos um cliente (browser) que faz requisição a um Web Service RESTful por meio do protocolo HTTP. Em seguida, o Web Service RESTful faz o acesso ao banco de dados, usando um comando SQL, e o Servidor de Banco de Dados devolve a resposta da solicitação ao Web Service RESTful, o qual repassa o resultado da requisição no formato JSON (JavaScript Object Notation) para o cliente.



Fonte: Elaborada pelo autor.

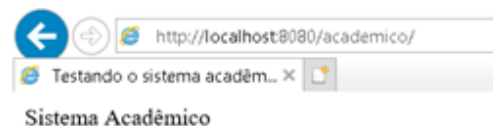
O primeiro passo de nossa implementação é criar um projeto de uma **Web Application**, conforme apresentado na Aula 6. Veja abaixo como deve ficar o projeto:



```
1 <!DOCTYPE html>
2 <!--
3 To change this license header, choose License Headers in Project Properties.
4 To change this template file, choose Tools | Templates
5 and open the template in the editor.
6 -->
7 <html>
8 <head>
9 <title>Testando o sistema acadêmico</title>
10 <meta charset="UTF-8">
11 <meta name="viewport" content="width=device-width, initial-scale=1.0">
12 </head>
13 <body>
14 <div>Sistema Acadêmico</div>
15 </body>
16 </html>
```

Fonte: Elaborada pelo autor.

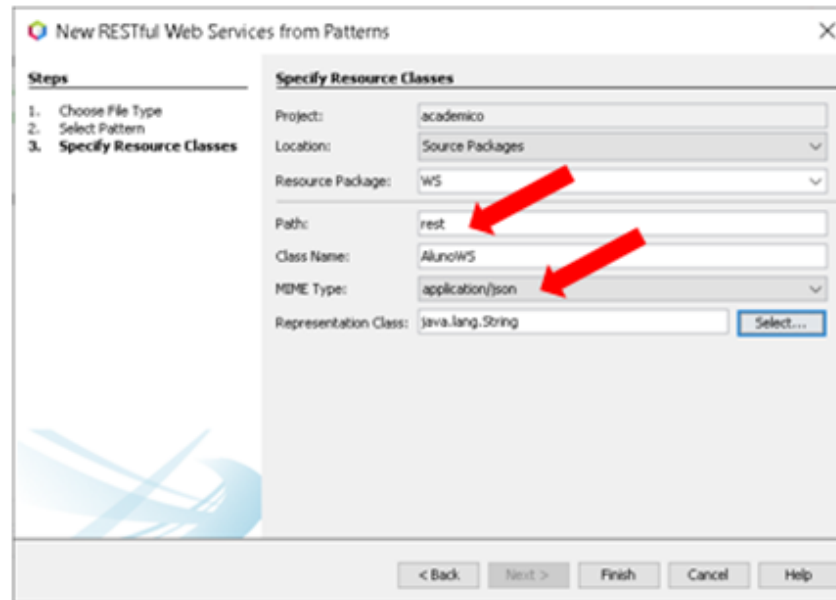
Para testar o projeto **Web Application**, basta clicar com o botão direito do mouse no **projeto academico** e escolher a opção **Deploy** no menu pop-up. Isso é importante para testar o **Servidor GlassFish** e verificar se o projeto foi criado corretamente. Abra um navegador (browser) para teste e digite a seguinte URL: <http://localhost:8080/academico/>. Assim, temos o browser acessando a página HTML **index.html** do projeto **Web Application**, conforme o print abaixo:



Fonte: Elaborada pelo autor.

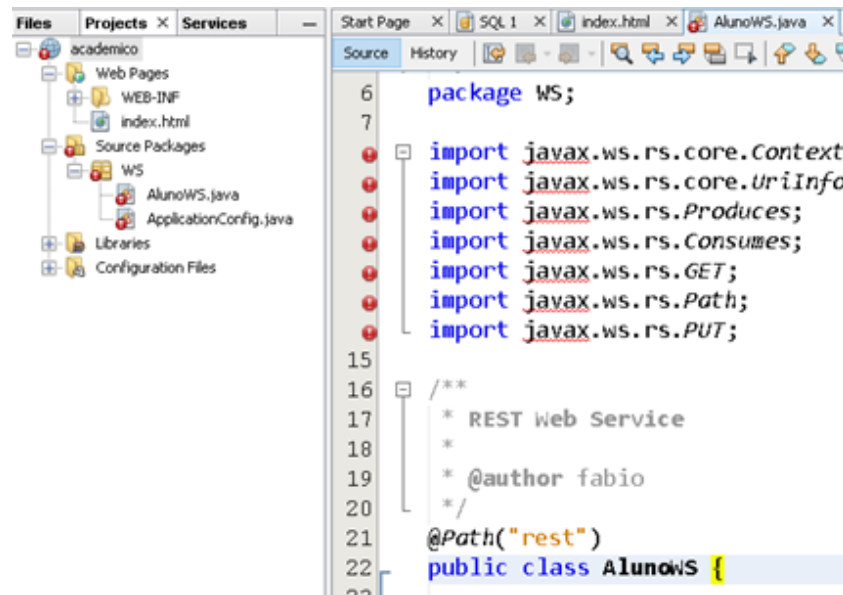
Realizado esse passo, podemos criar no **Web Service RESTful**. Para tanto, clique com o botão direito do mouse no projeto **Web Application academico** e escolha a opção **New -> Web Services RESTful from Patterns....** Preencha as informações nas janelas que aparecem, conforme as orientações da Aula 6. Preste atenção quando for preencher as informações referentes ao Web Service na janela abaixo, principalmente no

campo **Path**, com o valor **rest**, e no campo **MIME Type**, ou seja, tipo de formato na comunicação. Neste caso, escolha o formato texto **application/json**.



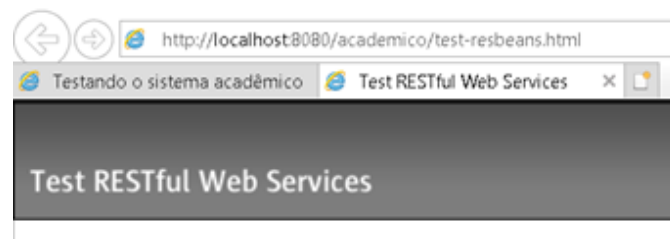
Fonte: Elaborada pelo autor.

No final, teremos um **projeto de um Web Service RESTful**, conforme o print da janela a seguir:



Fonte: Elaborada pelo autor.

Note que a anotação **@Path("rest")** no arquivo **alunosWS.java** está com o mesmo valor definido no campo **Path** da janela anterior. Além disso, o NetBeans acusa um erro no arquivo **alunosWS.java**, indicando que o pacote **javax.ws.rs.core** não foi encontrado no projeto. Esse pacote contém a implementação da especificação **JAX-RS 2.0**. Para resolver esse problema, basta adicionar a biblioteca com a implementação do **JAX-RS 2.0** na pasta **Libraries** do projeto. Após esse ajuste, já podemos testar o Web Service, clicando com o botão direito do mouse em cima do **projeto academico** e selecionando no menu pop-up a opção **Test RESTful Web Services**. Assim, teremos a janela abaixo com a execução do **Web Service RESTful** ainda sem nenhuma funcionalidade. Note que a URL do Web Service é a seguinte: `<http://localhost:8080/academico/test-resbeans.html>`.

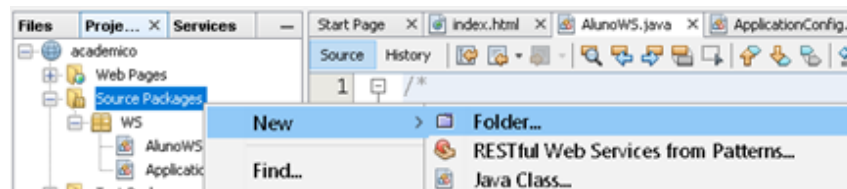


Fonte: Elaborada pelo autor.

Para fazer o acesso ao banco de dados com a tabela **aluno**, precisamos inserir no projeto as classes que desenvolvemos no padrão **DAO** (Data Access Object) da Aula 4. As classes são:

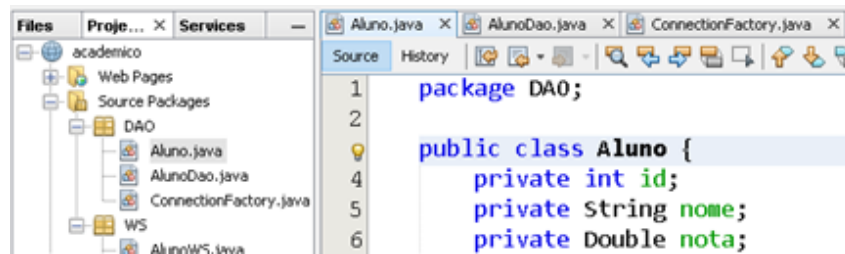
- **Aluno.java**;
- **AlunoDAO.java**; e
- **ConnectionFactory.java**.

Para copiar as classes para o projeto, primeiro crie uma pasta (folder) dentro do projeto. Para isso, só clicar com o botão direito do mouse na pasta **Source Packages**. Em meu caso, criei uma pasta com o nome **DAO**, ou seja, um novo **Package** com as classes acima que serão usadas no **Package WS**.



Fonte: Elaborada pelo autor.

Depois que o **Package DAO** for criado, basta copiar as classes para a pasta. Um detalhe importante é que, para que as classes funcionem entre esses **Packages** (**DAO** e **WS**), elas precisam estar declaradas como públicas (**public**), como a classe **Aluno** abaixo:



Fonte: Elaborada pelo autor.

Na classe **WSAluno.java**, implementaremos o método **getAlunos()** que retorna uma lista de alunos no formato **JSON**. No início da implementação do método são usadas algumas anotações do pacote **javax.ws.rs** que definem configurações importantes para o Web Service RESTful:

- **@GET** indica o método **public String getAlunos()** e implementa um método para obter informações sobre um recurso no Web Service;
- **@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)** especifica o tipo **MIME** que o método produzirá como resposta para o cliente; neste caso, o formato escolhido é o JSON;
- **@Path("alunos")** define um caminho para um recurso, além de ser implementada dentro do método.

No corpo do método, temos a chamada do método **public ArrayList <Aluno> getLista()** da classe **AlunoDao**, esse método retorna um **ArrayList** com os objetos da classe **Aluno** (classe **JavaBeans**).

```

@GET
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
@Path("alunos") // anotação para identificar o caminho do recurso (URI)
public String getAlunos() {
    AlunoDao dao = new AlunoDao ();
    List<Aluno> listaAlunos = dao.getLista();
    String listaJSON="";

    for (Aluno aluno : listaAlunos)
        listaJSON+= "ID:" +aluno.getId()+
                    " Nome:" +aluno.getNome()+
                    " Nota:" +aluno.getNota()+"\n";
    return listaJSON;
}

```

Fonte: Elaborada pelo autor.

Para que o teste funcione perfeitamente, precisamos fazer alguns ajustes na implementação, tais como importar o pacote **DAO** e **java.util.List**, conforme abaixo:

```

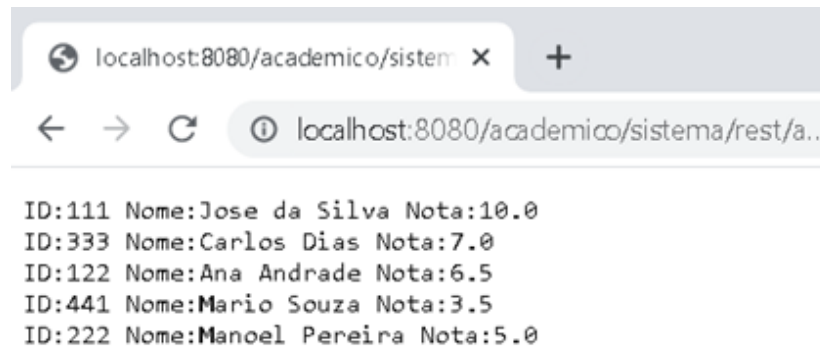
import DAO.*;
import java.util.List;

```

Além disso, podemos alterar a anotação **@javax.ws.rs.ApplicationPath()** na classe **ApplicationConfig**, trocando a String **"webresources"** para a String **"sistema"**. Assim, teríamos a seguinte URI de nosso Web Service RESTful:

- **URI:** <http://localhost:8080/academico/sistema/rest/alunos>.
- **URL:** http://localhost:8080/academico/ :(URL + nome de nosso servidor Web).
- **Caminho e nome do recurso:** sistema/rest/alunos.

Para testar o Web Service, podemos colocar a URI do recurso diretamente na barra de endereços de um navegador. Em seguida, temos a resposta da operação, ou seja, a lista de alunos é apresentada pelo navegador, conforme a imagem a seguir. No exemplo, utilizei como navegador o **Chrome**.



Fonte: Elaborada pelo autor.

Infelizmente, a resposta retornada pelo método **getAlunos()** não está no formato JSON. Se você revisitar as respostas das requisições realizadas ao site **JSONPlaceholder** na **Aula 5**, poderá constatar que o formato da resposta do método **getAlunos()** não segue a sintaxe do JSON.

O modelo JSON possui uma sintaxe bastante simples para representar uma informação. Em nosso exemplo, queremos representar uma **lista de objetos** da classe **Aluno**, na qual cada **objeto** possui vários **atributos** e **valores**. No formato JSON, cada **atributo** e **valor** é representado por um par **nome/valor**, de forma que o **nome** é o identificador do atributo e é descrito entre aspas seguido por **dois pontos** e pelo **valor do atributo**. Um **objeto** é especificado entre chaves e pode ser composto por múltiplos pares **nome/valor** e, por fim, os objetos da **lista de objetos** são separados por vírgulas. Para o exemplo acima, teríamos a seguinte resposta da requisição no formato JSON:

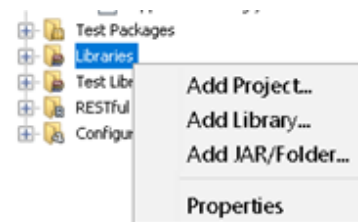
```
[
  {"id":111,"nome":"Jose da Silva","nota":10.0},
  {"id":222,"nome":"Manoel Pereira","nota":5.0},
  {"id":333,"nome":"Carlos Dias","nota":7.0},
  {"id":122,"nome":"Ana Andrade","nota":6.5},
  {"id":441,"nome":"Mario Souza","nota":3.5}
]
```

Fonte: Elaborada pelo autor.

Para converter a lista de objetos da classe **Aluno** para o formato JSON, utilizaremos a biblioteca **GSON** do Google, a qual converte objetos de **classes JavaBeans** em uma representação no formato JSON e vice-versa.

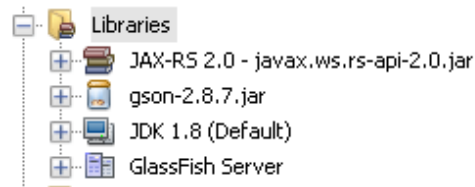
Para adicionar a biblioteca GSON ao projeto do Web Service, precisamos primeiro fazer o download do arquivo **.jar** com a biblioteca GSON no site a seguir: <<https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.7>>.

Em seguida, clique com o botão direito do mouse em **Libraries -> Add JAR/Folder...** Veja abaixo como proceder:



Fonte: Elaborada pelo autor.

Após adicionar o **arquivo .jar** da biblioteca GSON, temos a pasta **Libraries** com as seguintes bibliotecas:



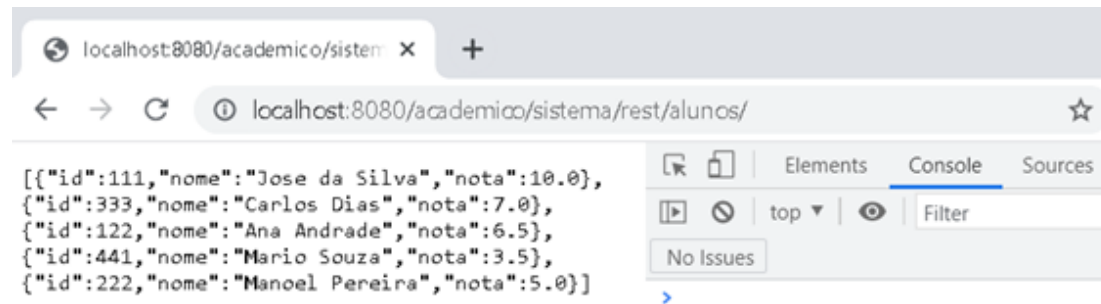
Fonte: Elaborada pelo autor.

Dessa forma, podemos reescrever o método **getAlunos()** para converter a lista de objetos da classe **Aluno** para uma **String** no formato JSON. Para tanto, basta passar a lista por parâmetro ao método **gson.toJson(listaAlunos)**.

```
@GET
@Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
@Path("alunos") // anotação para identificar o caminho do recurso (URI)
public String getAlunos() {
    AlunoDao dao = new AlunoDao ();
    List<Aluno> listaAlunos = dao.getList();
    // cria um objeto Gson
    Gson gson = new Gson();
    // gera a lista de alunos no formato JSON e retorna
    return gson.toJson(listaAlunos);
}
```

Fonte: Elaborada pelo autor.

Como visto na **Aula 5**, o método **@GET** pode ser testado usando tanto a barra de endereços de um navegador ou escrevendo uma consulta, utilizando a **API Fetch** do **JavaScript**. Para testar a consulta, acesse no menu do **Chrome Mais Ferramentas -> Ferramentas do desenvolvedor**. Em seguida, abrirá a janela do desenvolvedor. Nela, podemos escrever requisições, utilizando a linguagem **JavaScript**. Lembre-se de que, para escrever as requisições usando a aba console da ferramenta do desenvolvedor do Chrome, é necessário que, na barra de endereço do browser, você esteja acessando a URL de nosso Web Service RESTful. Veja a seguir:



Fonte: Elaborada pelo autor.

Antes de surgir a **API Fetch**, as requisições HTTP eram feitas por meio do objeto **XMLHttpRequest**, o qual fornece funcionalidade ao cliente para transferir dados entre um cliente e um servidor Web.

No exemplo apresentado no próximo print de tela, primeiro é instanciado o objeto **XMLHttpRequest** em uma constante (**const request = new XMLHttpRequest()**). Em seguida, é definido o método (**GET**) e a URL da requisição, para depois declarar dois **call-backs**, **onload** e **onerror**, para as ações de sucesso e erro, respectivamente. Ao final, é executado a requisição com o **request.send()**.

```

> const request = new XMLHttpRequest()

request.open('GET', 'http://localhost:8080/academico/sistema/rest/alunos/')

request.onload = function () {
  console.log(JSON.parse(this.responseText))
}

request.onerror = function () {
  console.log('erro ao executar a requisição')
}

request.send()
< undefined
▼ (5) [{...}, {...}, {...}, {...}, {...}] ⓘ
  ▶ 0: {id: 111, nome: "Jose da Silva", nota: 10}
  ▶ 1: {id: 333, nome: "Carlos Dias", nota: 7}
  ▶ 2: {id: 122, nome: "Ana Andrade", nota: 6.5}
  ▶ 3: {id: 441, nome: "Mario Souza", nota: 3.5}
  ▶ 4: {id: 222, nome: "Manoel Pereira", nota: 5}
    length: 5
  ▶ __proto__: Array(0)
>

```

Fonte: Elaborada pelo autor.

Todo esse processo torna a requisição complexa. Por isso, foi proposta a **API Fetch**, a qual torna a implementação de uma requisição HTTP mais simples.

A **API Fetch** provê ao navegador uma interface para a execução de requisições HTTP por meio de **Promises**. Uma **Promise** (“promessa”) é um objeto usado para o **processamento assíncrono** e representa um valor que pode estar disponível agora, no futuro ou nunca. Veja como fica simples a requisição usando a **API Fetch**:

```
> fetch('http://localhost:8080/academico/sistema/rest/alunos/').
  then(resposta => resposta.json()).
  then(jsonResposta => console.log(jsonResposta));
< ▶ Promise {<pending>}

▼ (5) [{...}, {...}, {...}, {...}, {...}] ⓘ
  ▶ 0: {id: 111, nome: "Jose da Silva", nota: 10}
  ▶ 1: {id: 333, nome: "Carlos Dias", nota: 7}
  ▶ 2: {id: 122, nome: "Ana Andrade", nota: 6.5}
  ▶ 3: {id: 441, nome: "Mario Souza", nota: 3.5}
  ▶ 4: {id: 222, nome: "Manoel Pereira", nota: 5}
    length: 5
    __proto__: Array(0)
>
```

Fonte: Elaborada pelo autor.

No exemplo, é utilizado o método **fetch()**, tendo como parâmetro a URI do recurso do Web Service. A resposta retornada pelo método **fetch()** é um objeto **Response (resposta)**, ou seja, uma **Promise**. Se for retornada corretamente, então é executado o método **json()** dentro do método **then()**. Esse método permite recuperar a informação retornada (**jsonResposta**) pela requisição no formato JSON e, em seguida, na próxima chamada do método **then()**, é possível escrever o resultado retornado (**jsonResposta**) no console da Ferramenta do Desenvolvedor.

Com isso, finalizamos a implementação do método **getAlunos()**, o qual devolve uma lista de alunos e os testes de acesso ao recurso disponibilizados pelo Web Service RESTful com a utilização da **API Fetch** do **JavaScript**. Agora, desenvolveremos um outro método que devolverá os dados de um determinado aluno. O método **getAluno(@PathParam("id") int id)** receberá, por parâmetro, o **id** do aluno que terá suas informações recuperadas. Veja a seguir a implementação do método **getAluno()**:


```

@GET
@Produces( javax.ws.rs.core.MediaType.APPLICATION_JSON)
@Path("alunos/{id}")
public String getAluno(@PathParam("id") int id) {
    AlunoDao dao = new AlunoDao();
    Aluno aluno = dao.consulta(id);
    if( aluno != null ){
        // cria um objeto Gson
        Gson gson = new Gson();
        // converte o objeto aluno no formato JSON e retorna
        return gson.toJson(aluno);
    }
    else
        // sinaliza que o aluno não foi encontrado
        throw new WebApplicationException(Response.Status.NOT_FOUND);
}

```

Fonte: Elaborada pelo autor.

A implementação do método começa com a anotação **@GET** para indicar que ele responderá a requisições do **HTTP @GET**, seguido da anotação **@Path** para definir um caminho específico para esse recurso.

Note que estamos utilizando **@GET** em dois métodos diferentes da classe **AlunoWS**. Entretanto, cada um deles atende a um **path único**. Se tentarmos mapear um único conjunto (método HTTP + path) para mais de um método, teremos um erro em tempo de execução quando o Web Service for colocado em execução. Assim, considerando o recurso **Alunos**, temos os seguintes métodos GET implementados no Web Service:

Método HTTP	Identificação do recurso	Utilização
GET	/sistema/rest/alunos	Recupera os dados de todos os alunos.
GET	/sistema/rest/alunos/id	Recupera os dados de um determinado aluno.

Para diferenciar o **path** dos métodos **getAlunos()** e **getAluno()**, utilizamos o **@Path("alunos/{id}")** para o método **getAluno()**, juntamente com a anotação **@PathParam("id")** que referencia a variável **"id"**, associando seu valor ao parâmetro **int id**.

Para recuperar os dados de um aluno, o método **getAluno()** utiliza o método **consulta(int id)** da classe **AlunoDao**, e o método **consulta(int id)** retorna um objeto da classe **Aluno** quando o **id** do aluno constar na base de dados. Em seguida, o objeto é convertido para o formato JSON e retornado ao cliente. Caso nenhum aluno seja selecionado, é lançada uma exceção do tipo **WebApplicationException**. Essa é uma exceção especial, definida pela **JAX-RS 2.0**, que tem como propósito mapear o erro ocorrido em um código de status **HTTP**. No construtor da exceção é informado o código de status do erro igual a **404 (Response.Status.NOT_FOUND)**, utilizado quando um recurso não é encontrado.

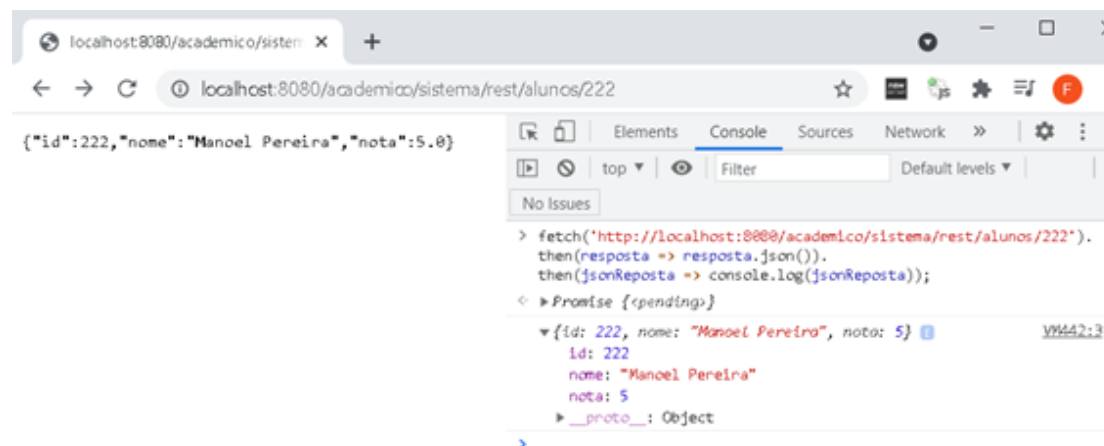
No trecho de código a seguir é apresentada a implementação do método **consulta(int id)** da classe **AlunoDao**. Note que o código do método é similar à implementação do método **getList()** da classe **AlunoDao**.

```
public Aluno consulta(int id){
    String sql = "SELECT * FROM aluno WHERE id_aluno = ?";
    Aluno aluno = null;
    try {
        PreparedStatement stmt = this.connection.prepareStatement(sql);
        stmt.setLong(1,id);
        ResultSet rs = stmt.executeQuery();
        // o metodo next() retorna true se tiver um elemento no ResultSet
        if ( rs.next() ){
            // preenche o objeto aluno com as informações do ResultSet
            aluno = new Aluno();
            aluno.setId(rs.getInt( "id_aluno" ));
            aluno.setNome(rs.getString("nome"));
            aluno.setNota(rs.getDouble("nota"));
        }
        rs.close();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return aluno;
}
```

Fonte: Elaborada pelo autor.

No método **consulta(int id)**, utilizamos o método **executeQuery()** que retorna um **ResultSet**, o qual, quando retornado pela execução da query, terá sempre seu cursor posicionado antes do primeiro registro. Por isso, é necessário utilizar o método **next()** que retorna um booleano, indicando se existe um próximo registro e, se existir, posiciona o cursor do **resultSet** nesse registro.

O teste do método **getAluno(@PathParam("id") int id)** pode ser realizado acessando a URI do recurso diretamente na barra de endereço do browser ou utilizando a **API Fetch**, conforme apresentado abaixo:



Fonte: Elaborada pelo autor.

O método **getAluno()** lança uma exceção com status de erro igual **404**, caso o **id** do aluno não seja encontrado. Por conta disso, devemos fazer o tratamento dessas falhas. Veja abaixo o tratamento de erro, utilizando a **API Fetch**:

```
> fetch('http://localhost:8080/academico/sistema/rest/alunos/221').
  then(resposta => {
    // verifica se a conexao falhou
    if (!resposta.ok){
      // gera um erro que caira no catch abaixo
      throw Error(resposta.status);
    }
    return resposta.json();
  }).
  then(jsonReposta => console.log(jsonReposta)). // ok
  catch(erro => console.log("erro não encontrado id do aluno")); // falha
< ▶ Promise {<pending>}
```

```
✖ ▶ GET http://localhost:8080/academico/sistema/rest/alunos/221 404 (Not Found)
    erro não encontrado id do aluno
```

Fonte: Elaborada pelo autor.

No exemplo acima, foi passado ao Web Service o **id=221**, que não consta na tabela **aluno**. Assim, o Web Service devolverá o erro **404** que é tratado como script pelo método **catch()** da **API Fetch**. O método **catch()** é executado quando falha a requisição.

Na próxima aula, implementaremos os outros métodos (Adicionar, Atualizar e Remover) de nosso Web Service.