Programação Lógica

Programação Web em Prolog

Alexsandro Santos Soares
prof.asoares@gmail.com

4 de junho de 2021

Bacharelado em Sistemas de Informação
Faculdade de Computação
Universidade Federal de Uberlândia

Sumário

Sumário i

Sumário ii

Introdução

O projeto bookmarks

O projeto Bookmarks permite ao usuário armazenar seus links prediletos juntamente com uma descrição da página.

Para começar é necessário definir a interface do serviço (API), informando as rotas , os métodos HTTP usados para cada uma e uma breve descrição da funcionalidade.

Além disso as rotas para as páginas do front-end também devem ser descritas juntamente com uma descrição.

Nos próximos slides a definição do site do nosso exemplo.

API

Método	Rota	Descrição
GET	`/api/v1/bookmarks/`	Retorna uma lista com todos os *bookmarks*.
GET	`/api/v1/bookmarks/1`	Retorna o *bookmark* com ID 1 ou erro 404 caso o *bookmark* não seja encontrado.
POST	`/api/v1/bookmarks`	Adiciona um novo *bookmark*. Os dados deverão ser passados no corpo da requisição no formato JSON. Um erro 400 (BAD REQUEST) deve ser retornado caso a URL não tenha sido informada.
PUT	`/api/v1/bookmarks/3`	Atualiza o *bookmark* de ID 3. Os dados deverão ser passados no corpo da requisição no formato JSON.
DELETE	`/api/v1/bookmarks/5`	Apaga o *bookmark* de Id 5.

Front End

Rota	Descrição
`/`	retorna a página de entrada listando todos os *bookmarks*.
`/bookmark/`	retorna a página para o cadastro do *bookmarks*.
`/bookmark/editar/2`	retorna a página para alteração do *bookmark* com ID 2 ou erro 404 caso o *bookmark* não seja encontrado.

Sempre que o usuário fizer alguma operação de alteração ou remoção de algum *bookmark*, ele será redirecionado para a página de entrada.

Usando o CURL para testar a API

Usando o CURL para testar a API

O CURL (*Client URL*)é um programa de linha de comando que nos permite testar e obter dados de uma API.

Podemos usar o CURL para enviar pedidos GET:

- Para obter a lista de todos os bookmarks

```
curl -v http://localhost:8000/api/v1/bookmarks/
```

- Para obter uma tupla específica, por exemplo, aquela com id 1

```
curl -v http://localhost:8000/api/v1/bookmarks/1
```

Usando o CURL para testar a API

Para enviar JSON com POST

- Para obter a lista de todos os bookmarks

```
curl -v -d '{"título":"Exemplo", "url":"https://exemplo.br/"}'
     -H 'Content-Type: application/json'
     http://localhost:8000/api/v1/bookmarks/
```

- Se o existir arquivo com o JSON, por exemplo, `tupla.json`, com o conteúdo

```
{
 "título":"Exemplo",
 "url":"https://exemplo.br/"
}
```

Podemos enviá-lo com

```
curl -v -d @tupla.json -H 'Content-Type: application/json'
      http://localhost:8000/api/v1/bookmarks/
```

Usando o CURL para testar a API

Para enviar JSON com PUT

- Para obter a lista de todos os bookmarks

```
curl -d '{"id": "5", "título":"Exemplo", "url":"https://exemplo.br/"}
     -H 'Content-Type: application/json'
     -X PUT http://localhost:8000/api/v1/bookmarks/
```

- Se o existir arquivo com o JSON, podemos enviá-lo com

```
curl -v -d @tupla.json -H 'Content-Type: application/json'
     -X PUT http://localhost:8000/api/v1/bookmarks/
```

Usando o CURL para testar a API

Para enviar um pedido com DELETE, por exemplo, para remover a tupla com id igual a 5:

```
curl -v -X DELETE http://localhost:8000/api/v1/bookmarks/5
```

Organização de diretórios

Organizando os diretórios do projeto

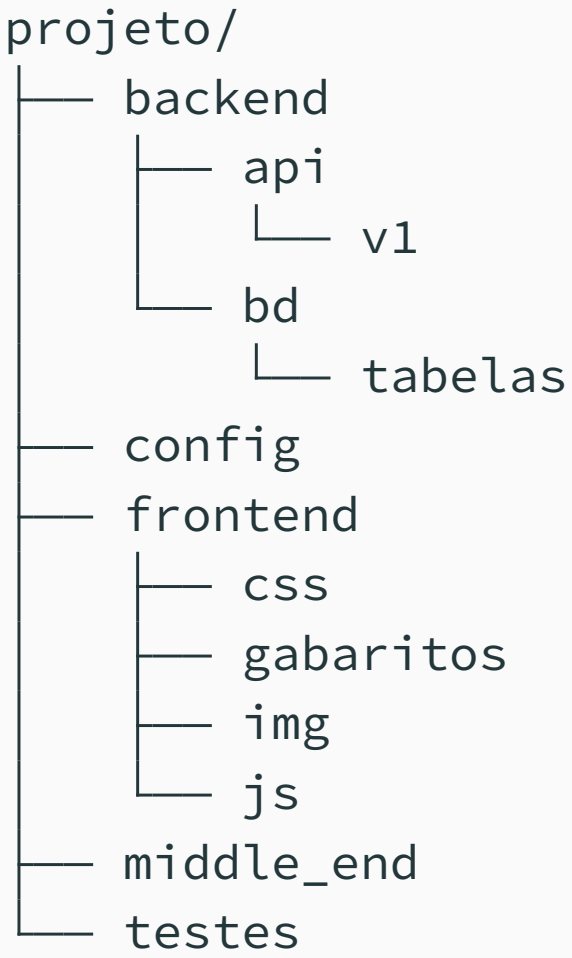
Em qualquer projeto de software uma boa organização da estrutura de diretórios ajuda a encontrar arquivos e também evita uma série de erros futuros.

Adotaremos a seguinte estrutura de diretórios para os nossos projetos:

```
projeto/
├── backend
│   ├── api
│   │   └── v1
│   └── bd
│       └── tabelas
├── config
├── frontend
│   ├── css
│   ├── gabaritos
│   ├── img
│   └── js
├── middle_end
└── testes
```

Essa estrutura será detalhada no próximo slide.

Estrutura de diretórios do projeto



projeto é o nome de seu projeto.

backend arquivos relacionados ao backend.

- **api/v1** é o lugar para os arquivos da API REST.
- **bd** diretório para os esquemas das tabelas do banco de dados. Os dados das tabelas ficarão em **bd/tabelas**.

config arquivos para configuração de rotas e outras informações.

frontend arquivos relacionados ao frontend, como aqueles responsáveis pela geração das páginas HTML.

- **css** arquivos CSS.
- **gabaritos** gabaritos usados na geração de páginas.
- **img** diretório para as imagens.
- **js** arquivos JavaScript

middle_end arquivos usados tanto no front quanto no backend.

testes arquivos de testes

Arquivos do Bootstrap

Agora que já temos os diretórios, iremos colocar os arquivos nos devidos lugares. O nosso projeto será chamado de `bookmarks`.

Como usaremos o Bootstrap nos exemplos que se seguirão, precisamos copiar os arquivos do Bootstrap para dois diretórios do `frontend`:

```
bookmarks/
└── frontend
    ├── css
    │   └── bootstrap.min.css
    └── js
        └── bootstrap.bundle.min.js
```

Para não ter que lembrar de incluir nas páginas HTML esses arquivos, criaremos um gabarito de nome `bootstrap5`, já visto, mas repetido no próximo slide:

```
bookmarks/frontend/
├── css
│   └── bootstrap.min.css
├── gabaritos
│   └── bootstrap5.pl
└── js
    └── bootstrap.bundle.min.js
```

frontend/gabaritos/bootstrap5.pl

```
% html_requires
:- use_module(library(http/html_head)).

% html, html_post, html_root_attribute
:- use_module(library(http/html_write)).

:- multifile
        user:body//2.

user:body(bootstrap5, Corpo) -->
       html(body([ \html_post(head,
                              [ meta([name(viewport),
                                      content([ 'width=device-width',
                                                'initial-scale=1',
                                                'shrink-to-fit=no'
                                              ])])]),
                   \html_root_attribute(lang,'pt-br'),
                   \html_requires(css('bootstrap.min.css')),

                   Corpo,

                   script([ src('/js/bootstrap.bundle.min.js'),
                            type('text/javascript')], [])
                ])).
```

Caminhos do projeto

Para que o Prolog saiba onde colocamos os arquivos, precisamos configurar um arquivo com abreviações para todos os caminhos que usarmos.

O arquivo com os caminhos será chamado `caminhos.pl` e ficará no diretório raiz de nosso projeto:

```
bookmarks/
├── caminhos.pl
└── frontend
    ├── css
    │   └── bootstrap.min.css
    ├── gabaritos
    │   └── bootstrap5.pl
    ├── img
    └── js
        └── bootstrap.bundle.min.js
```

O arquivo `caminhos.pl` é mostrado nso próximos dois slides.

caminhos.pl

```
:- multifile user:file_search_path/2.

% file_search_path(Apelido, Caminho)
%     Apelido é como será chamado um Caminho absoluto ou
%     relativo no sistema de arquivos

% Diretório principal do servidor: sempre coloque o caminho completo.

user:file_search_path(dir_base, '/home/alex/UFU/programas/aula18').

% Diretório do projeto
user:file_search_path(projeto, dir_base(bookmarks)).

% Diretório de configuração
user:file_search_path(config, projeto(config)).

%% Front-end
user:file_search_path(frontend, projeto(frontend)).

%% Recursos estáticos
user:file_search_path(dir_css, frontend(css)).
user:file_search_path(dir_js,  frontend(js)).
user:file_search_path(dir_img, frontend(img)).
user:file_search_path(dir_webfonts, frontend(webfonts)).

% Gabaritos para estilização
user:file_search_path(gabarito, frontend(gabaritos)).
```

Continuação do arquivo: caminhos.pl

```
%% Backend
user:file_search_path(backend, projeto(backend)).


% Banco de dados
user:file_search_path(bd, backend(bd)).
user:file_search_path(bd_tabs, bd(tabelas)).

% API REST
user:file_search_path(api,  backend(api)).
user:file_search_path(api1, api(v1)).


% Middle-end
user:file_search_path(middle_end, projeto(middle_end)).
```

Banco de dados

Banco de dados

Para o projeto que estamos estudando, precisaremos apenas de uma tabela chamada `bookmark` que possui três atributos:

id a chave primária, um número inteiro positivo.

título o título do *bookmark*, uma string.

url a URL do *bookmark*, uma string.

Entretanto, precisamos que a chave primária para cada tupla nessa tabela seja única, ou seja, nunca se repita.

Temos várias opções resolver isso, mas usaremos uma tabela de nome `chave` que será responsável por nos fornecer um inteiro diferente para cada tabela, toda vez que um for solicitado.

Tabela chave

A tabela `chave` será incluída em todos os bancos de dados que construirmos.

O código para gerar as chaves primárias será mostrado no próximo slide. O arquivo, de nome `chave.pl`, ficará em:

```
bookmarks/
└── backend
    ├── api
    │   └── v1
    └── bd
        ├── chave.pl
        └── tabelas
```

backend/bd/chave.pl

```
:- module(
       chave,
       [ carrega_tab/1,
         pk/2,
         inicia_pk/2 ]
   ).

:- use_module(library(persistency)).

:- persistent
   chave( nome:atom,
          valor:positive_integer ).

:- initialization( at_halt(db_sync(gc(always))) ).

% Informa onde os arquivos de dados serão colocados
carrega_tab(ArqTabela):-
    db_attach(ArqTabela, []).
```

O predicado `carrega_tab/1` deve ser chamado antes de começar a usar tabela, pois é ele que informa onde estão ou serão colocados os dados da tabela.

Continuação do arquivo: backend/bd/chave.pl

```
 % Sempre que precisar de uma chave primária, chame
 % pk(NomeDaTabela, Chave), o segundo argumento é
 % a chave primária que será criada por esse predicado.

pk(Nome, Valor):- !,
    atom_concat(pk, Nome, Mutex),
    with_mutex(Mutex,
                (
                    ( chave(Nome, ValorAtual) ->
                      ValorAntigo = ValorAtual;
                      ValorAntigo = 0 ),
                    Valor is ValorAntigo + 1,
                    retractall_chave(Nome,_), % remove o valor antigo
                    assert_chave(Nome, Valor)) ). % atualiza com o novo


% Talvez você queira um valor inicial diferente de 1

inicia_pk(Nome, ValorInicial):- !,
    atom_concat(pk, Nome, Mutex),
    with_mutex(Mutex,
                (  chave(Nome, _)
                -> true % Não inicializa caso a chave já exista
                ;  assert_chave(Nome, ValorInicial) )).
```

Tabela bookmark

Agora a tabela `bookmark` que possui três atributos:

id a chave primária, um número inteiro positivo.

título o título do *bookmark*, uma string.

url a URL do *bookmark*, uma string.

será implementada no arquivo `bookmark.pl`, visto na sequência, e colocada em:

```
bookmarks/
└── backend
    ├── api
    │   └── v1
    └── bd
        ├── chave.pl
        ├── bookmark.pl
        └── tabelas
```

backend/bd/bookmark.pl

```
:- module( bookmark,
           [ carrega_tab/1,
             bookmark/3,
             insere/3, remove/1, atualiza/3 ]).
:- use_module(library(persistency)).

:- use_module(chave, []).

:- persistent
   bookmark( id:positive_integer, % chave primária
             título:string,
             url:string ).

:- initialization( at_halt(db_sync(gc(always))) ).

carrega_tab(ArqTabela):- db_attach(ArqTabela, []).

insere(Id, Título, URL):-
    chave:pk(bookmark, Id), % obtenha a chave primária
    with_mutex(bookmark,
               assert_bookmark(Id, Título, URL)).

remove(Id):- with_mutex(bookmark,
                        retractall_bookmark(Id, _Título, _URL)).

atualiza(Id, Título, URL):-   % pode falhar se o Id não existir
    with_mutex(bookmark,
               ( retract_bookmark(Id, _TítAntigo, _URLAntiga),
                 assert_bookmark(Id, Título, URL)) ).
```

Configuração do banco de dados

As duas tabelas que usaremos no projeto já estão prontas, mas elas precisam procedimentos adicionais para funcionarem.

Num projeto maior, o número de tabelas crescerá. Para organizar o processo, colocaremos todas as configurações que precisarmos em um arquivo `banco_de_dados.pl` que ficará no diretório `config`:

```
bookmarks/
├── backend
│   ├── api
│   │   └── v1
│   └── bd
│       ├── bookmark.pl
│       ├── chave.pl
│       └── tabelas
├── caminhos.pl
└── config
    └── banco_de_dados.pl
```

O arquivo é mostrado a seguir.

config/banco_de_dados.pl

```
% Banco de dados
% Coloque aqui todas as tabelas do banco.

tabela(chave).      % tabela sempre usada
tabela(bookmark).

% Não mexa daqui em diante

:- initialization( carrega_tabelas ).

carrega_tabelas():-
    findall(Tab, tabela(Tab), Tabs),
    maplist(carrega_tab,Tabs).

carrega_tab(Tabela):-
    use_module(bd(Tabela),[]),
    atomic_list_concat(['tbl_', Tabela, '.pl'], ArqTab),
    expand_file_search_path(bd_tabs(ArqTab), CaminhoTab),
    Tabela:carrega_tab(CaminhoTab).
```

Nesse arquivo serão colocados os nomes das tabelas usadas e localizadas em `backend/bd`. O módulo então inicializará automaticamente todas essas tabelas.

Por exemplo, a tabela `bookmark` será inicializada para manipular os dados em `backend/bd/tabelas/tbl_bookmark.pl`.

Configuração do servidor e das rotas

Configuração do servidor e das rotas

Continuando no diretório `config`, temos mais dois arquivos:

`servidor.pl`: responsável pelo lançamento do servidor e configuração da porta que ele ouvirá.

`rotas.pl`: conterá todas as rotas atendidas pelo servidor.

Com as novas adições, o diretório `config` do projeto ficará assim:

```
bookmarks/
├── caminhos.pl
└── config
    ├── banco_de_dados.pl
    ├── rotas.pl
    └── servidor.pl
```

Na seguir será apresentado o arquivo `servidor.pl` e na sequência, `rotas.pl`.

config/servidor.pl

```
% http_server
:- use_module(library(http/thread_httpd)).

% http_dispatch
:- use_module(library(http/http_dispatch)).

%  http_set_session_options
:- use_module(library(http/http_session)).

servidor(Porta) :-
    % As sessões serão criadas quando forem explicitamente solicitadas
    http_set_session_options([ create(noauto) ]),
    http_server(http_dispatch, [port(Porta)]).
```

Aqui introduzimos uma chamada para o predicado `http_set_session_options`, da biblioteca `http/http_session` que controla as sessões do servidor.

Esse conceito será discutido em aulas posteriores.

O arquivo contendo todas as rotas e tratadores respectivos está nos próximos slides.

config/rotas.pl

```
% http_handler, http_reply_file
:- use_module(library(http/http_dispatch)).

% http:location
:- use_module(library(http/http_path)).

:- ensure_loaded(caminhos).

/**********************************
 *                                *
 *      Rotas do Servidor Web     *
 *                                *
 **********************************/

:- multifile http:location/3.
:- dynamic   http:location/3.

%% http:location(Apelido, Rota, Opções)
%      Apelido é como será chamada uma Rota do servidor.
%      Os apelidos css, icons e js já estão definidos na
%      biblioteca http/http_server_files, os demais precisam
%      ser definidos.

http:location(img, root(img), []).
http:location(api, root(api), []).
http:location(api1, api(v1), []).
http:location(webfonts, root(webfonts), []).
```

Continuação de config/rotas.pl

```
/*************************
 *                       *
 *      Tratadores       *
 *                       *
 *************************/

% Recursos estáticos
:- http_handler( css(.),
                 serve_files_in_directory(dir_css), [prefix]).
:- http_handler( img(.),
                 serve_files_in_directory(dir_img), [prefix]).
:- http_handler( js(.),
                 serve_files_in_directory(dir_js),  [prefix]).
:- http_handler( webfonts(.),
                 serve_files_in_directory(dir_webfonts), [prefix]).

% Rotas do Frontend

%% A página inicial
:- http_handler( root(.), entrada,   []).

%% A página de cadastro de novos bookmarks
:- http_handler( root(bookmark), cadastro, []).

%% A página para edição de um bookmark existente
:- http_handler( root(bookmark/editar/Id), editar(Id), []).
```

Continuação de config/rotas.pl

```
% Rotas da API
:- http_handler( api1(bookmarks/Id), bookmarks(Método, Id),
                 [ method(Método),
                   methods([ get, post, put, delete ]) ]).
```

Note que no slide anterior definimos todas as rotas para os recursos estáticos e também para o *frontend*. Em particular, note a última rota definida lá:

```
:- http_handler( root(bookmark/editar/Id), editar(Id), []).
```

Sempre que o usuário acessar, por exemplo, a rota `bookmark/editar/1`, o pedido será encaminhado para o tratador `editar(1)`, já com o identificador da tupla na tabela `bookmark`.

Técnica semelhante é usada para tratar as rotas da API, como visto no trecho acima.

Carregamento dos arquivos do projeto

Por fim, precisamos carregar em Prolog todos os arquivos que serão usados no projeto.

Para isso há o arquivo `carrega_website.pl`, localizado no diretório `config`:

```
bookmarks/
├── backend
├── caminhos.pl
├── config
│   ├── banco_de_dados.pl
│   ├── carrega_website.pl
│   ├── rotas.pl
│   └── servidor.pl
├── frontend
├── middle_end
└── testes
```

Esse arquivo é mostrado no próximo slide. Ele está configurado para caregar o servidor na porta 8000, se quiser outra porta é só mudar.

config/carrega_website

```
% Configuração do servidor

% Carrega o servidor e as rotas
:- load_files([ servidor,
                rotas
              ],
              [ silent(true),
                if(not_loaded) ]).

% Inicializa o servidor para ouvir a porta 8000
:- initialization( servidor(8000) ).

% Carrega arquivos do frontend
:- load_files([ gabarito(bootstrap5),  % gabarito usando Bootstrap 5
                gabarito(boot5rest),   % Bootstrap 5 com API REST
                frontend(entrada),
                frontend(bookmark)
              ],
              [ silent(true),
                if(not_loaded) ]).


% Carrega arquivos do backend
:- load_files([ api1(bookmarks) % API REST
              ],
              [ silent(true),
                if(not_loaded) ]).
```

Executando o servidor

Centralizaremos todas as inicializações em um único arquivo de nome `executar.pl` que ficará no diretório raiz do projeto:

```
bookmarks/
├── backend
├── caminhos.pl
├── executar.pl
├── config
│   ├── banco_de_dados.pl
│   ├── carrega_website.pl
│   ├── rotas.pl
│   └── servidor.pl
├── frontend
├── middle_end
└── testes
```

Assim, para iniciar o servidor com todas as dependências, basta digitar no diretório raiz do projeto:

```
swipl executar.pl
```

bookmark/executar.pl

```
% http_handler
:- use_module(library(http/http_dispatch)).

% http:location está aqui
:- use_module(library(http/http_path)).

% serve_files_in_directory
:- use_module(library(http/http_server_files)).

% Para as ações de logging
:- use_module(library(http/http_log)).

% Para usar JSON
:- use_module(library(http/http_json)).

/* Aumenta a lista de tipos aceitos pelo servidor */
:- multifile http_json/1.

http_json:json_type('application/x-javascript').
http_json:json_type('text/javascript').
http_json:json_type('text/x-javascript').
http_json:json_type('text/x-json').

:- load_files([ caminhos,   % arquivo contendo os caminhos dos diretórios
                config(banco_de_dados),
                config(carrega_website)
              ],
              [ silent(true),
                if(not_loaded) ]).
```

Considerando as modificações feitas, o nosso projeto estará assim:

```
bookmarks/
├── backend
│   ├── api/v1
│   └── bd
│       ├── tabelas
│       ├── bookmark.pl
│       └── chave.pl
├── config
│   ├── banco_de_dados.pl
│   ├── carrega_website.pl
│   ├── rotas.pl
│   └── servidor.pl
├── frontend
│   ├── css
│   │   └── bootstrap.min.css
│   ├── gabaritos
│   │   └── bootstrap5.pl
│   ├── img
│   └── js
│       └── bootstrap.bundle.min.js
├── middle_end
├── testes
├── caminhos.pl
└── executar.pl
```

Implementando a API REST

Implementando a API do projeto

Vamos relembrar os rotas que precisam ser programadas para a API:

Método	Rota	Descrição
GET	`/api/v1/bookmarks/`	Retorna uma lista com todos os *bookmarks*.
GET	`/api/v1/bookmarks/1`	Retorna o *bookmark* com ID 1 ou erro 404 caso o *bookmark* não seja encontrado.
POST	`/api/v1/bookmarks`	Adiciona um novo *bookmark*. Os dados deverão ser passados no corpo da requisição no formato JSON. Um erro 400 (BAD REQUEST) deve ser retornado caso a URL não tenha sido informada.
PUT	`/api/v1/bookmarks/3`	Atualiza o *bookmark* de ID 3. Os dados deverão ser passados no corpo da requisição no formato JSON.
DELETE	`/api/v1/bookmarks/5`	Apaga o *bookmark* de Id 5.

Todas os operações com a API envolverão o envio ou o recebimento de um arquivo JSON. Além disso, os navegadores só compreendem diretamente os métodos GET e POST e não PUT e DELETE. Para resolver esses dois problemas usaremos JavaScript.

Tratador de eventos para uma API REST

Para lidar com os detalhes do envio e recebimento de arquivos JSON, usaremos o arquivo JavaScript `rest.js` que ficará no diretório `frontend/js/`.

Os detalhes exatos da programação desse arquivo não são importantes. O que importa é saber como usá-lo em nosso projeto.

O arquivo `rest.js` possui duas funções básicas:

- `enviarDados` envia os dados de um formulário, que ela transforma no formato JSON, usando um dos métodos: POST ou PUT.
- `remover` essa função deve ser chamada sempre que desejar apagar uma tupla em uma tabela. Ela envia um pedido com verbo DELETE.

No próximo slide vamos detalhar os argumentos que essas funções necessitam.

Enviar Dados

`enviarDados(evento, callback)` envia os dados de um formulário, no formato JSON, usando um dos métodos: POST ou PUT. Se o método não for informado em uma entrada com nome _método, a função assume como default que o método é POST. Os argumentos dessa função são:

- evento que é um evento qualquer gerado pelo navegador, por exemplo, ao se clicar em um botão.
- callback é a função de resposta. Ela é responsável por tratar a resposta que chegar do servidor. Essa função deve ser implementada separadamente em outro arquivo JavaScript, pois pode depender do que se deseja fazer ao receber uma resposta em uma determinada página.

A técnica usada aqui para enviar dados JSON via POST ou PUT pressupõe a existência de um elemento HTML escondido no formulário informando o método de envio:

```
<input type="hidden" name="_método" value="PUT">
```

Remover Dados

A função `remover(url, callback)` deve ser chamada sempre que desejar apagar uma tupla em uma tabela.

Os argumentos dessa função são:

- url, a ponta da api que está sendo chamada. Por exemplo: se a URL for `/api/v1/bookmarks/5`, significa que se deseja apagar a tupla com o identificador 5 da tabela `bookmark`.
- callback, que possui o mesmo uso da função anterior.

Tratador de eventos para a tabela bookmark

Usando as funções disponibilizadas no módulo `rest.js` podemos agora implementar as funções específicas para lidar com as transações REST relativas à tabela `bookmark`.

Essas funções ficarão no arquivo `/frontend/js/bookmark.js` e são elas:

- redirecionaResposta(evento, rotaRedireção) que chama a função `enviarDados` do módulo `rest.js` para enviar os dados de um formulário usando POST ou PUT. Ao receber uma resposta afirmativa do servidor ela redireciona o cliente para a rota indicada por rotaRedireção. Se um erro ocorrer, uma página de erro será mostrada.

- apagar(evento, rotaRedireção) que solicita ao servidor para apagar a tupla indicada por uma URL. Ela chama a função `remover` de `rest.js` e ao receber uma resposta afirmativa, também redireciona para a rota informada em rotaRedireção. Se um erro ocorrer, uma página de erro será mostrada.

frontend/js/bookmark.js

```
/* Esse módulo assume que rest.js tenha sido corregado.
 */

/**
 *  Função auxiliar que imprime para o console do navegador
 *  a resposta recebida do servidor e depois redireciona
 *  para a rota dada.
 *
 * @param {Object} resp - corpo da resposta devolvida pelo servidor
 * @param {URL}    rota - a rota a ser seguida
 */
function redireciona(resposta, rota){
  console.log(resposta);       /* escreve a resposta para o console */
  window.location.href = rota; /* redireciona para a rota dada */
}

function redirecionaResposta(evento, rotaRedireção) {
  enviarDados(evento, resposta => redireciona(resposta, rotaRedireção));
}

function apagar(evento, rotaRedireção) {
  evento.preventDefault();
  const elemento = evento.currentTarget;
  const url = elemento.href;

  console.log('delete url = ', url);   /* escreve a url para o console */
  remover( url,  resposta => redireciona(resposta, rotaRedireção));
}
```

Gabarito para o Bootstrap com API REST

Agora que temos os arquivos para lidar no lado do cliente com a API REST, precisamos incluir esses arquivos em todas as páginas HTML geradas.

como essa é uma atividade rotineira em praticamente todas as páginas do projeto, vamos escrever um gabarito de nome `boot5rest` para incluir tanto os arquivos do Bootstrap quanto o arquivo `rest.js` nas páginas HTML.

Esse gabarito ficará no arquivo `frontend/gabaritos/bootrest.pl` e, com todas essas novas adições, o subdiretório `frontend` ficará assim:

```
bookmarks/frontend/
├── css
│   └── bootstrap.min.css
├── gabaritos
│   ├── boot5rest.pl
│   └── bootstrap5.pl
├── img
└── js
    ├── bookmark.js
    ├── bootstrap.bundle.min.js
    └── rest.js
```

frontend/gabaritos/boot5rest.pl

```
% html_requires
:- use_module(library(http/html_head)).

% html, html_post, html_root_attribute
:- use_module(library(http/html_write)).

:- multifile
        user:body//2.

user:body(boot5rest, Corpo) -->
       html(body([ \html_post(head,
                              [ meta([name(viewport),
                                      content([ 'width=device-width',
                                                'initial-scale=1',
                                                'shrink-to-fit=no'
                                              ])])]),
                   \html_root_attribute(lang,'pt-br'),
                   \html_requires(css('bootstrap.min.css')),
                   \html_requires(js('rest.js')),

                   Corpo,

                   script([ src('/js/bootstrap.bundle.min.js'),
                            type('text/javascript')], [])
                 ])).
```

Tratador para o rota bookmarks da API

Tratador para o rota bookmarks da API

Podemos agora iniciar a escrita do tratador para todas as rotas da API que envolvam a tabela `bookmark`.

O arquivo para essa rota ficará em `backend/api/v1/bookmarks.pl`. Vamos relembrar o que indicamos no arquivo `config/rotas.pl`

```
:- http_handler( api1(bookmarks/Id), bookmarks(Método, Id),
                 [ method(Método),
                   methods([ get, post, put, delete ]) ]).
```

Ou seja, qualquer solicitação que envolva a rota `bookmarks/` será tratada pelo predicado `bookmarks` que recebe, além do pedido, o método no qual o pedido foi feito e o identificador `Id`, se houver. Caso a rota seja acessada sem um identificador, a variável `Id` é unificada com o átomo vazio `''`.

Vamos discutir o arquivo `backend/api/v1/bookmarks.pl` nos próximos slides.

backend/api/v1/bookmarks.pl

Abaixo está a parte do arquivo que lida com rotas feitas com GET.

```
/* http_parameters   */
:- use_module(library(http/http_parameters)).
/* http_reply        */
:- use_module(library(http/http_header)).
/* reply_json_dict   */
:- use_module(library(http/http_json)).

:- use_module(bd(bookmark), []).

/*
   GET api/v1/bookmarks/
   Retorna uma lista com todos os bookmarks.
*/
bookmarks(get, '', _Pedido):- !,
    envia_tabela.

/*
   GET api/v1/bookmarks/Id
   Retorna o `bookmark` com Id 1 ou erro 404 caso o `bookmark` não
   seja encontrado.
*/
bookmarks(get, AtomId, _Pedido):-
    atom_number(AtomId, Id), % o identificador aparece na rota como um átomo,
    !,                       % converta-o para um número inteiro.
    envia_tupla(Id).
```

backend/api/v1/bookmarks.pl

Abaixo está a parte do arquivo que lida com rotas feitas com POST.

```
/*
   POST api/v1/bookmarks
   Adiciona um novo bookmark.
   Os dados são passados no corpo do pedido usando o formato JSON.

   Um erro 400 (BAD REQUEST) deve ser retornado caso a URL não tenha sido
   informada.
   */

bookmarks(post, _, Pedido):-
    http_read_json_dict(Pedido, Dados), % lê o JSON enviado com o Pedido
    !,
    insere_tupla(Dados).
```

Note que o `Id`, se informado, é descartado, pois trata-se de uma inserção na tabela e, assim, um novo identificador será gerado.

backend/api/v1/bookmarks.pl

Abaixo está a parte do arquivo que lida com rotas feitas com PUT.

```
/*
  PUT api/v1/bookmarks/Id
  Atualiza o bookmark com o  Id dado.
  Os dados são passados no corpo do pedido usando o formato JSON.
*/

bookmarks(put, AtomId, Pedido):-
    atom_number(AtomId, Id),
    http_read_json_dict(Pedido, Dados),  % lê o JSON enviado com o Pedido
    !,
    atualiza_tupla(Dados, Id).
```

backend/api/v1/bookmarks.pl

Abaixo está a parte do arquivo que lida com rotas usando o verbo DELETE.

```
/*
   DELETE api/v1/bookmarks/Id
   Apaga o bookmark com o Id informado
*/

bookmarks(delete, AtomId, _Pedido):-
    atom_number(AtomId, Id),
    !,
    bookmark:remove(Id),
    throw(http_reply(no_content)). % Responde usando o código 204 No Content

 /* Se algo ocorrer de errado, a resposta de método não
    permitido será retornada.
 */

 bookmarks(Método, Id, _Pedido) :-
    % responde com o código 405 Method Not Allowed
    throw(http_reply(method_not_allowed(Método, Id))).
```

Os códigos de resposta do servidor podem ser encontrados na documentação do predicado `http_status_reply`.

backend/api/v1/bookmarks.pl

A parte final do arquivo contém os predicados para inserir, atualizar ou remover tuplas da tabela bookmark:

```
insere_tupla( _{ título:Título, url:URL}):-
    % Validar URL antes de inserir
    bookmark:insere(Id, Título, URL)
    -> envia_tupla(Id)
    ;  throw(http_reply(bad_request('URL ausente'))).

atualiza_tupla( _{ título:Título, url:URL}, Id):-
       bookmark:atualiza(Id, Título, URL)
    -> envia_tupla(Id)
    ;  throw(http_reply(not_found(Id))).


envia_tupla(Id):-
       bookmark:bookmark(Id, Título, URL)
    -> reply_json_dict( _{id:Id, título:Título, url:URL} )
    ;  throw(http_reply(not_found(Id))).


envia_tabela :-
    findall( _{id:Id, título:Título, url:URL},
             bookmark:bookmark(Id,Título,URL),
             Tuplas ),
    reply_json_dict(Tuplas).  % envia o JSON para o solicitante
```

Testes da API

Com a API terminada, podemos testá-la usando o cURL, conforme já visto.

Vamos criar alguns arquivos JSON e os colocaremos no diretório `testes`, dentro do subdiretório `bookmarks`, que é novo e deve ser criado.

Arquivo `tupla1.json`:

```
{
   "título": "Exemplo 1",
   "url":    "https://exemplo1.br/"
}
```

Arquivo `tupla2.json`:

```
{
   "título": "Exemplo 2",
   "url":    "https://exemplo2.br/"
}
```

Arquivo `tupla3.json`:

```
{
   "título": "Exemplo 3",
   "url":    "https://exemplo3.br/"
}
```

Arquivos para testes da API

Arquivo `tupla4.json`:

```
{
   "título": "UFU",
   "url":    "http://www.ufu.br"
}
```

Arquivo `tupla5.json`:

```
{
   "título": "Exemplo errado 5",
   "url":    "www.exemplo5.br"
}
```

Arquivo `tupla5atual.json`:

```
{
   "título": "Exemplo 5",
   "url":    "https://exemplo5.br/"
}
```

Resumo dos diretórios

Os diretórios `backend` e `testes` ficarão assim:

```
bookmarks/
├── backend
│   ├── api
│   │   └── v1
│   │       └── bookmarks.pl
│   └── bd
│       ├── tabelas
│       ├── bookmark.pl
│       └── chave.pl
├── testes
│   └── bookmarks
│       ├── tupla1.json
│       ├── tupla2.json
│       ├── tupla3.json
│       ├── tupla4.json
│       ├── tupla5atual.json
│       └── tupla5.json
├── caminhos.pl
└── executar.pl
```

Testagem

Para testar a API, abra dois terminais:

1. no primeiro vá para o diretório raiz do projeto e digite

```
swipl executar
```

2. no segundo, vá para o diretório `testes/bookmarks`

```
cd testes/bookmarks/
```

   e siga os experimentos a seguir.

Para inserir um novo bookmark via POST:

```
curl -v -d @tupla1.json -H 'Content-Type: application/json' \
    http://localhost:8000/api/v1/bookmarks/

* Connection #0 to host localhost left intact
{"id":1, "título":"Exemplo 1", "url":"https://exemplo1.br/"}
```

A resposta indica que uma nova tupla foi criada.

Testagem

Vamos inserir mais algumas tuplas com POST

```
curl -v -d @tupla2.json -H 'Content-Type: application/json' \
        http://localhost:8000/api/v1/bookmarks/

curl -v -d @tupla3.json -H 'Content-Type: application/json' \
        http://localhost:8000/api/v1/bookmarks/

curl -v -d @tupla4.json -H 'Content-Type: application/json' \
        http://localhost:8000/api/v1/bookmarks/

curl -v -d @tupla5.json -H 'Content-Type: application/json' \
        http://localhost:8000/api/v1/bookmarks/
```

Testagem

Vamos pedir a tabela `bookmark` inteira com GET:

```
curl -v http://localhost:8000/api/v1/bookmarks/
```

A resposta será:

```
[
  {"id":1, "título":"Exemplo 1", "url":"https://exemplo1.br/"},
  {"id":2, "título":"Exemplo 2", "url":"https://exemplo2.br/"},
  {"id":3, "título":"Exemplo 3", "url":"https://exemplo3.br/"},
  {"id":4, "título":"UFU",        "url":"http://www.ufu.br"},
  {"id":5, "título":"Exemplo errado 5", "url":"www.exemplo5.br"}
]
```

Testagem

Vamos agora atualizar a tupla 5 com novos valores usando PUT:

```
curl -v -d @tupla5atual.json -H 'Content-Type: application/json' \
     -X PUT http://localhost:8000/api/v1/bookmarks/5

{"id":5, "título":"Exemplo 5", "url":"https://exemplo5.br/"}
```

Note que a tupla com identificador 5 foi atualizada.

Vamos apagar o registro de número 4 usando DELETE:

```
curl -v -X DELETE http://localhost:8000/api/v1/bookmarks/4
```

Testagem

Para terminar, vamos pedir a tabela `bookmark` atualiza com todas as modificações feitas:

```
curl -v http://localhost:8000/api/v1/bookmarks/
```

A resposta será:

```
[
  {"id":1, "título":"Exemplo 1", "url":"https://exemplo1.br/"},
  {"id":2, "título":"Exemplo 2", "url":"https://exemplo2.br/"},
  {"id":3, "título":"Exemplo 3", "url":"https://exemplo3.br/"},
  {"id":5, "título":"Exemplo 5", "url":"https://exemplo5.br/"}
]
```

Que está exatamente como esperávamos.

Com os resultados dos testes aumenta a nossa confiança que a API está funcionando corretamente.

Frontend

Implementando o frontend

Agora podemos implementar as rotas do frontend, que repetimos abaixo:

Rota	Descrição
`/`	retorna a página de entrada listando todos os *bookmarks*.
`/bookmark/`	retorna a página para o cadastro do *bookmarks*.
`/bookmark/editar/2`	retorna a página para alteração do *bookmark* com ID 2 ou erro 404 caso o *bookmark* não seja encontrado.

Sempre que o usuário fizer alguma operação de alteração ou remoção de algum *bookmark*, ele será redirecionado para a página de entrada.

Recordando as rotas do frontend e seus tratadores respectivos, que colocamos em `config/rotas.pl`, temos:

```
%% A página inicial
:- http_handler( root(.), entrada,   []).

%% A página de cadastro de novos bookmarks
:- http_handler( root(bookmark), cadastro, []).

%% A página para edição de um bookmark existente
:- http_handler( root(bookmark/editar/Id), editar(Id), []).
```

Página inicial

O tratador para a página inicial será colocado em `frontend/entrada.pl`:

```
/* html//1, reply_html_page  */
:- use_module(library(http/html_write)).
/* html_requires  */
:- use_module(library(http/html_head)).

:- ensure_loaded(gabarito(boot5rest)).

:- use_module(bd(bookmark), []).    % não importa nada implicitamente

entrada(_):-
    reply_html_page(
        boot5rest,
        [ title('Bookmarks')],
        [ div(class(container),
              [ \html_requires(css('entrada.css')),
                \html_requires(js('bookmark.js')),
                \título_da_página('Meus bookmarks'),
                \tabela_de_bookmarks
              ]) ]).

título_da_página(Título) -->
    html( div(class('text-center align-items-center w-100'),
    h1('display-3', Título))).
```

Note a inclusão do gabarito `boot5rest`. Também note a maneira como incluímos a tabela `bookmark`. Ambos serão necessários nesta página.

Continuação de frontend/entrada.pl

```
tabela_de_bookmarks -->
    html(div(class('container-fluid py-3'),
             [ \cabeça_da_tabela('Bookmarks', '/bookmark'),
               \tabela
             ]
            )).


cabeça_da_tabela(Título,Link) -->
    html( div(class('d-flex'),
              [ div(class('me-auto p-2'), h2(b(Título))),
                div(class('p-2'),
                    a([ href(Link), class('btn btn-primary')],
                      'Novo'))
              ]) ).


tabela -->
    html(div(class('table-responsive-md'),
             table(class('table table-striped w-100'),
                   [ \cabeçalho,
                     tbody(\corpo_tabela)
                   ]))).
```

Continuação de frontend/entrada.pl

```
cabeçalho -->
    html(thead(tr([ th([scope(col)], '#'),
                    th([scope(col)], 'Título'),
                    th([scope(col)], 'URL'),
                    th([scope(col)], 'Ações')
                  ]))).

corpo_tabela -->
    { findall( tr([th(scope(row), Id), td(Título), td(Link), td(Ações)]),
               linha(Id, Título, Link, Ações),
               Linhas ) },
    html(Linhas).

linha(Id, Título, Link, Ações):-
    bookmark:bookmark(Id, Título, URL),
    ações(Id,Ações),
    Link = a([href(URL)], URL).

ações(Id, Campo):-
    Campo = [ a([ class('text-success'), title('Alterar'),
                  href('/bookmark/editar/~w' - Id),
                  'data-toggle'(tooltip)],
                [ \lápis ]),
              a([ class('text-danger ms-1'), title('Excluir'),
                  href('/api/v1/bookmarks/~w' - Id),
                  onClick("apagar( event, '/' )"),
                  'data-toggle'(tooltip)],
                [ \lixeira ])
            ].
```

Continuação de frontend/entrada.pl

Neste e no próximo slide estão os ícones do Bootstrap que usei para marcar a edição (lápis) e a remoção (a lixeira) de uma tupla.

Esses e outros ícones são próprios do Bootstrap e não necessitam de nenhuma instalação adicional.

```
% Ícones do Bootstrap 5

lápis -->
    html(svg([ xmlns('http://www.w3.org/2000/svg'),
               width(16),
               height(16),
               fill(currentColor),
               class('bi bi-pencil'),
               viewBox('0 0 16 16')
             ],
             path(d(['M12.146.146a.5.5 0 0 1 .708 0l3 3a.5.5 0 0 1 0 .708l-10 10a.5.5 0 0',
             ' 1-.168.11l-5 2a.5.5 0 0 1-.65-.65l2-5a.5.5 0 0 1 .11-.168l10-10zM11.207 2.5',
             ' 13.5 4.793 14.793 3.5 12.5 1.207 11.207 2.5zm1.586 3L10.5 3.207 4',
             ' 9.707V10h.5a.5.5 0 0 1 .5.5v.5h.5a.5.5 0 0 1 .5.5v.5h.293l6.5-6.5zm-9.761',
             ' 5.175-.106.106-1.528 3.821 3.821-1.528.106-.106A.5.5 0 0 1 5',
             ' 12.5V12h-.5a.5.5 0 0 1-.5-.5V11h-.5a.5.5 0 0 1-.468-.325z']), [])))).
```

Continuação de frontend/entrada.pl

```
lixeira -->
    html(svg([ xmlns('http://www.w3.org/2000/svg'),
               width(16),
               height(16),
               fill(currentColor),
               class('bi bi-trash'),
               viewBox('0 0 16 16')
             ],
             [ path(d(['M5.5 5.5A.5.5 0 0 1 6 6v6a.5.5 0 0 1-1 0V6a.5.5 0 0 1',
                       ' .5-.5zm2.5 0a.5.5 0 0 1 .5.5v6a.5.5 0 0 1-1 0V6a.5.5',
                       ' 0 0 1 .5-.5zm3 .5a.5.5 0 0 0-1 0v6a.5.5 0 0 0 1 0V6z']), []),
               path(['fill-rule'(evenodd),
                     d(['M14.5 3a1 1 0 0 1-1 1H13v9a2 2 0 0 1-2 2H5a2 2 0 0',
                        ' 1-2-2V4h-.5a1 1 0 0 1-1-1V2a1 1 0 0 1 1-1H6a1 1 0 0 1',
                        ' 1-1h2a1 1 0 0 1 1 1h3.5a1 1 0 0 1 1 1v1zM4.118 4 4',
                        ' 4.059V13a1 1 0 0 0 1 1h6a1 1 0 0 0 1-1V4.059L11.882',
                        ' 4H4.118zM2.5 3V2h11v1h-11z'])], [])])).
```

Páginas de cadastro e edição

A página para cadastro de novos bookmarks será colocada em `frontend/bookmark.pl`.

Por economia de espaço nos slides, também a página de edição será colocada nesse mesmo arquivo.

```
/* html//1, reply_html_page  */
:- use_module(library(http/html_write)).
/* html_requires  */
:- use_module(library(http/html_head)).

:- ensure_loaded(gabarito(boot5rest)).


/* Página de cadastro de bookmark */
cadastro(_Pedido):-
    reply_html_page(
        boot5rest,
        [ title('Bookmarks')],
        [ div(class(container),
              [ \html_requires(js('bookmark.js')),
                h1('Meus bookmarks'),
                \form_bookmark
              ]) ]).
```

Continuação de frontend/bookmark.pl

```
form_bookmark -->
    html(form([ id('bookmark-form'),
                onsubmit("redirecionaResposta( event, '/' )"),
                action('/api/v1/bookmarks/') ],
              [ \método_de_envio('POST'),
                \campo(título, 'Título', text),
                \campo(url, 'URL', url),
                \enviar_ou_cancelar('/')
              ])).

enviar_ou_cancelar(RotaDeRetorno) -->
    html(div([ class('btn-group'), role(group), 'aria-label'('Enviar ou cancelar')],
             [ button([ type(submit),
                        class('btn btn-outline-primary')], 'Enviar'),
               a([ href(RotaDeRetorno),
                   class('ms-3 btn btn-outline-danger')], 'Cancelar')
            ])).

campo(Nome, Rótulo, Tipo) -->
    html(div(class('mb-3'),
             [ label([ for(Nome), class('form-label') ], Rótulo),
               input([ type(Tipo), class('form-control'),
                       id(Nome), name(Nome)])
             ] )).
```

Continuação de frontend/bookmark.pl

```
/* Página para edição (alteração) de um bookmark  */

editar(AtomId, _Pedido):-
    atom_number(AtomId, Id),
    ( bookmark:bookmark(Id, Título, URL)
    ->
    reply_html_page(
        boot5rest,
        [ title('Bookmarks')],
        [ div(class(container),
              [ \html_requires(js('bookmark.js')),
                h1('Meus bookmarks'),
                \form_bookmark(Id, Título, URL)
              ]) ])
    ; throw(http_reply(not_found(Id)))
    ).


form_bookmark(Id, Título, URL) -->
    html(form([ id('bookmark-form'),
                onsubmit("redirecionaResposta( event, '/' )"),
                action('/api/v1/bookmarks/~w' - Id) ],
              [ \método_de_envio('PUT'),    % informa o método de envio
                \campo_não_editável(id, 'Id', text, Id),
                \campo(título, 'Título', text, Título),
                \campo(url,    'URL',    url,  URL),
                \enviar_ou_cancelar('/')
              ])).
```

Continuação de frontend/bookmark.pl

```
campo(Nome, Rótulo, Tipo, Valor) -->
    html(div(class('mb-3'),
             [ label([ for(Nome), class('form-label')], Rótulo),
               input([ type(Tipo), class('form-control'),
                       id(Nome), name(Nome), value(Valor)])
             ] )).

campo_não_editável(Nome, Rótulo, Tipo, Valor) -->
    html(div(class('mb-3 w-25'),
             [ label([ for(Nome), class('form-label')], Rótulo),
               input([ type(Tipo), class('form-control'),
                       id(Nome),
                       % name(Nome),%  não é para enviar o id
                       value(Valor),
                       readonly ])
             ] )).

método_de_envio(Método) -->
    html(input([type(hidden), name('_método'), value(Método)])).
```

Testando o frontend

Agora que já programamos todas as rotas do site, podemos testar o site diretamente de um navegador.

Para isso, vá para o diretório raiz do projeto e digite

```
swipl executar.pl
```

Depois, vá ao navegador e digite a URL `http://localhost:8000`.

Você deverá ver as todas as tuplas da tabela exibidas na página inicial.

Para saber mais

Para saber mais

- SWI-Prolog HTTP support. Disponível em

  `https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/http.html%27)`

Referências bibliográficas

Referências bibliográficas

- Anne Ogborn. *Tutorial - Creating Web Applications in SWI Prolog*. Disponível em `https://github.com/Anniepoo/swiplwebtut/blob/master/web.adoc`