Programação de Sistemas para Internet

Prof. Diego Cirilo

Aula 15: AJAX

AJAX

- *Asynchronous JavaScript and XML*
- XML (*eXtensible Markup Language*)
- Permite a troca de informações com o servidor sem recarregar a página;
- Ao invés do *browser* fazer a requisição por ação do usuário, o JS é o responsável;
- Pode requisitar novas informações do servidor depois que a página já está carregada;
- Dá *dinamicidade* aos sites.

AJAX

- Pode diminuir a carga do servidor, requisitando apenas os dados necessários;
- Pode permitir uma melhor experiência de usuário;
- Da mesma forma pode deixar o carregamento inicial da página lento;
- Passa a consumir mais recursos no cliente;
- Pode causar problemas de SEO (*Search Engine Optimization*) em páginas públicas;
- Tudo depende de como é implementado.

AJAX

- O AJAX inicia a tendência se *desacoplar* o serviço(*back*) do cliente(*front*);
- O tradicional é o *back* renderizar quase tudo e o AJAX ser utilizado em ações específicas;
- Esse equilíbrio é uma decisão de projeto;
- Há aplicações que não renderizam nada no *back-end*, que se responsabiliza apenas pela lógica/dados;
- O *front* é uma aplicação/projeto separado que se comunica através de uma API;
- Programação Orientada a Serviços (próximo ano).

AJAX

- Hoje se usa mais o JSON no lugar do XML;
- JSON (*JavaScript Object Notation*);
- Formato de texto para troca de dados;
- Usa uma sintaxe equivalente aos objetos JS.

JSON

```
{
    "chave1": "valor",
    "chave2": 34,
    "chave3": ["valor", "valor"],
    "chave4": {
        "chave1": "valor",
        "chave2": 23
     },
     "chave5": [
        {"chave1": "valor1"},
        {"chave1": "valor2"}
     ]
}
```

AJAX

- Sequência:
  - O JS faz a requisição para o servidor;
  - Essa requisição é disparada normalmente por um evento, seja de usuário ou automático;
  - Caso seja um POST, o JS envia os dados para o servidor como JSON;
  - O servidor recebe, processa, e retorna a resposta/dados no formato JSON;
  - O JS processa o JSON para atualizar o DOM com o novo conteúdo.

AJAX com *jQuery*

- O *jQuery* tem uma função `$.ajax`;

```
$.ajax({
  url: "https://api.exemplo.com/dados", // URL do servidor
  type: "POST", // Método HTTP (GET, POST, PUT, DELETE)
  data: JSON.stringify({ nome: "João" }), // Dados enviados
  contentType: "application/json", // Tipo de conteúdo
  success: function(response) { // Quando a requisição for bem-sucedida
    console.log(response);
  },
  error: function(xhr, status, error) { // Em caso de erro
    console.error("Erro:", error);
  }
});
```

AJAX com *jQuery*

- Também existem os *atalhos*;
- O ponto negativo é que não tem a função de `error`;

```
$.get("https://jsonplaceholder.typicode.com/posts/1", function(data) {
  console.log(data);
});

$.post("https://api.exemplo.com/novo", { nome: "Maria" }, function(response) {
  console.log("Usuário criado!");
});
```

AJAX com *jQuery*

- As requisições são *assíncronas*;
- O código continua antes da resposta chegar;
- Devemos executar o que for necessário dentro das funções `success` e `error`;
- O JavaScript permite lidar melhor com essas operações assíncronas:
  - `async/await`
  - `.then()`
- Assunto da próxima disciplina.

Atualizando o DOM com AJAX GET

- A resposta do request GET pode ser acessada na função *callback*;
- Se for um JSON, podemos tratar como um objeto JS comum (parece o dict do Python);
- Usamos as funções de manipulação do DOM para atualizar o conteúdo;

Exemplo

```
$(#meuBotao).click(() => {
  $.get("https://jsonplaceholder.typicode.com/posts/1", function(data) {
    $("#minhaDiv").append(
      `<div>
        <h1>${data.title}</h1>
        <p>${data.body}</p>
       </div>`
    );
  });
});
```

AJAX com HTML renderizado

- A tarefa de renderizar o HTML no *front* usando JS pode ser complicada;
- Especialmente para estruturas maiores;
- Podemos quebrar a consistência, por exemplo, adicionando uma classe no template Django e esquecendo de atualizar no JS;
- Muitas vezes é mais prático continuar renderizando no *back* e enviar no AJAX apenas os blocos HTML prontos;

AJAX com HTML renderizado

- Há uma longa discussão se isso é uma boa prática, pois:
  - O JSON economiza dados;
  - O JSON não amarra os dados à visualização;
  - O JSON também permite o reuso da *view* para outros tipos de cliente;
- No fim é uma decisão de projeto.

AJAX no Django

- Para o Django o AJAX é uma requisição qualquer;
- A diferença é que ele pode receber/responder JSON, e não apenas HTML;
- `from django.http import JsonResponse`
- O trabalho maior está no JS do *front*.

AJAX no Django

- Podemos separar as *views* do AJAX das comuns;
- Exemplo de *view* de consulta JSON:

```
from django.http import JsonResponse
...
def ajax_get_livro(request, id):
    livro = get_object_or_404(Livro, id=id) # objeto livro
    resultado = {
        "titulo": livro.titulo,
        "autor": livro.autor,
        "ano": livro.ano,
    } # objeto dict
    return JsonResponse(resultado)
```

Serializer

- Permite converter um objeto em um texto;
- Objetos não podem ser enviados pela rede, texto sim;
  - JSON é texto;
- O `JsonResponse` entre outras coisas, faz:
  - Serializa o dado (converte dict em JSON/texto);
  - Define o `Content-Type` como `application/json`;
  - Cria uma resposta HTTP completa.

Serializer

- O `JsonResponse` por padrão só aceita dicionários:
  - Objetos do model não são dicts;
  - `QuerySets` não são dicts;
  - Listas não são dicts;
- Podemos converter campo a campo, como no exemplo anterior;
- Ou usar ferramentas para converter automaticamente.

Conversão automática

- Para um objeto do Model:
  - Função `model_to_dict(obj)`;
  - Retorna todos os campos por padrão;
  - Permite selecionar os campos com atributo `fields`;

```
from django.forms.models import model_to_dict # necessário!
...
def ajax_get_livro(request, id):
    livro = Livro.objects.get(id=id) # objeto livro
    livro_dict = model_to_dict(livro) # dict completo
    # livro_dict = model_to_dict(livro, fields=["nome", "autor"])
    return JsonResponse(livro_dict)
```

Conversão automática

- QuerySets: resultado de uma query `all/filter`;
- Conjunto de objetos;
- Método `.values(campos)`:
  - Converte os objetos no QuerySet em dicts;
  - Se `campos` for vazio, retorna todos;
  - O objeto externo continua sendo um QuerySet;
  - Devemos converter para uma lista `list(obj)`.

Conversão automática

- Exemplo

```
...
def ajax_get_livros(request):
    livros = Livro.objects.all() # queryset
    livros_dict = livros.values() # converte para dicts
    livros_list = list(livros_dict) # converte para lista
    return JsonResponse(livros_list) # ERRO!!!
```

Conversão automática

- O `JsonResponse` também não aceita listas normalmente;
- Podemos forçar com `JsonResponse(livros_list, safe=False)`;
  - Não era considerado seguro;
  - As falhas já foram mitigadas nas versões mais novas do JS (ref.);
- Podemos criar um dict para conter a lista, como é feito com o `context`:
  - `JsonResponse({"livros": livros_list})`

Exemplo completo

```
...
def ajax_get_livro(request, id):
    livro = get_object_or_404(Livro, id=id) # objeto livro
    return JsonResponse(model_to_dict(livro))

def ajax_get_livros(request):
    livros = Livro.objects.all() # queryset
    # livros = Livro.objects.filter(autor__icontains="Mic")
    livros_list = list(livros.values()) # converte de uma vez
    return JsonResponse({"livros": livros_list}) # Funciona!
```

404

- O que acontece se o objeto não for encontrado?
- Podemos usar o `get_object_or_404` e `get_list_or_404`;
- Redireciona a página para uma página de erro;
- Comportamento normal do Django.

404

- Pode ser uma experiência de usuário ruim;
- Não foi necessariamente o usuário que iniciou a requisição;
- O `JsonResponse` aceita definir o `status` da requisição;
- Retornamos um JSON com a informação de erro também;
- O JS trata o problema.

Exemplo

```
...
def ajax_get_livro(request, id):
    try:
        livro = get_object_or_404(Livro, id=id) # objeto livro
        return JsonResponse(model_to_dict(livro))
    except Livro.DoesNotExist:
        dict_do_erro = {
            "erro": "Livro não encontrado",
        }
        return JsonResponse(dict_do_erro, status=404)
```

Retornando HTML

- Como discutido antes, pode simplificar o desenvolvimento;
- Uma *view* normal, só o template que é específico;
  - O problema do 404 continua;
- Templates parciais - *partials*;
- Podemos usar os *partials* para compor a página completa também;
- Garante a consistência.

Exemplo

- Views:

```
def get_livro(request, id):
    livro = get_object_or_404(Livro, id=id)
    return render(request, "detalhe_livro.html", {"livro": livro})

def ajax_get_livro(request, id):
    livro = get_object_or_404(Livro, id=id)
    return render(request, "partials/_detalhe_livro.html", {"livro": livro})
```

Exemplo

- `partials/_detalhe_livro.html`

```
<div>
    <h1>{{ livro.titulo }}</h1>
    <h2>{{ livro.autor }}</h2>
</div>
```

- `detalhe_livro.html`

```
{% extends "base.html" %}
{% block content %}
...
<div>
    <h1 class="principal">Detalhe do livro</h1>
    {% include "partials/_detalhe_livro.html" %}
</div>
...
{% endblock %}
```

AJAX nos templates Django

- Nos templates base adicionamos o `jQuery`;
- Criamos o `<script>` que fará as requisições;
- Se o `<script>` estiver dentro do template, funções com `url` e `static` podem ser usadas;
- Se estiver em outro arquivo `.js`, não!
- Podemos usar alguns artifícios para passar informações para o `.js`.

AJAX nos templates Django

- A ordem das coisas é muito importante;
- É comum criar um `{% block scripts %}` para adicionar o JavaScript;
- Garante que ele fique sempre no final da página renderizada;

Exemplo

- No `base.html`:

```
...
<head>
...
<script defer src="https://code.jquery.com/jquery-3.7.1.min.js"></script>
</head>
...
<body>
...
  <script>
    window.onload = () => { //garante que os scripts defer foram carregados
      {% block script %}
      {% endblock %}
    }
  </script>
</body>
```

Exemplo

- Ou:

```
...
<body>
...
  <script src="https://code.jquery.com/jquery-3.7.1.min.js"></script>
  <script>
    {% block script %}
    {% endblock %}
  </script>
</body>
...
```

Exemplo

- No template específico:

```
{% block content %}
...
<button id="meuBotao">Clique para carregar livros</button>
...
<div id="minhaDiv">
...
</div>
{% endblock %}
...
{% block script %}
{{ block.super }}
$(#meuBotao).click( function () {
  $.get("{% url 'ajax_get_livros' %}", function(data) {
    $("#minhaDiv").append(
      `<div>
        <h1>${data.titulo}</h1>
        <p>${data.autor}</p>
       </div>`
    );
  });
});
</body>
...
```

Atributos `data` do HTML5

- Representam informações extras não-visuais;
- Podem ser utilizados para passar informações para o JS;
- Muito útil para passar URLs, IDs, etc, para o JS.
- Basta adicionar à *tag*;
- `<tag data-minhainfo="minha info extra" id="meuSeletor">`

Atributos `data` do HTML5

- Para acessar no JS:

```
$("#meuSeletor").data("minhainfo");
// retorna "minha info extra"
```

Exemplo

- No template:

```
<button data-url="{% url 'detalhar-view' coisa.id }" class="btn btn-ajax">Clique</button>
```

- No JS:

```
$(".btn-ajax").click( function () {
  $.get(
    $(this).data("url"), //passou a URL e funciona fora do template!
    function (resposta) {
      console.log(resposta);
    }
});
```

Funções JS

- Podemos organizar nosso código em arquivos `.js` estáticos;
- Criamos funções específicas para cada funcionalidade;
- Dentro dos templates apenas chamamos as funções;
- Centraliza o desenvolvimento;

Exemplo

- `base.html`

```
<head>
...
  <script defer src="https://code.jquery.com/jquery-3.7.1.min.js"></script>
  <script defer src="{% static 'js/script.js' %}"></script>
</head>
<body>
...
  <script>
    window.onload = () => { //garante que os scripts defer foram carregados
      {% block script %}
      {% endblock %}
    }
  </script>
</body>
...
```

Exemplo

- Ou:

```
...
  <script src="https://code.jquery.com/jquery-3.7.1.min.js"></script>
  <script src="{% static 'js/script.js' %}"></script>
  <script>
    {% block script %}
    {% endblock %}
  </script>
</body>
```

Exemplo

- `script.js`

```
function carregarLivrosJson(botao, url) {
  botao.click( function () {
    $.get(url, function(data) {
      $("#minhaDiv").append(
        `<div>
          <h1>${data.titulo}</h1>
          <p>${data.autor}</p>
        </div>`
      );
    });
  });
}
```

Exemplo

- No template

```
...
{% block script %}
  carregarLivrosJson($("#meuBotao"), {% url 'ajax_get_livros' %})
{% endblock %}
```

Recebendo HTML renderizado

- Caso o Django responda HTML, o JS fica bem simplificado;

```
$(#meuBotao).click( function () {
  $.get("{% url 'ajax_get_livros' %}", function(bloco_html) {
    $("#minhaDiv").html(bloco_html);
  });
});
```

POST

- Via AJAX, não é necessário um `<form>` para fazer o POST;
- Por questões de segurança, o Django exige o token CSRF;
- É necessário escrever uma função JS para pegar o token CSRF mesmo sem `form`;
- Ou usar o `<form>` com o `{% csrf_token %}` normalmente.

Exemplo de JS sem Form

```
// pega o CSRF token nos cookies
function getCSRFToken() {
    return document.cookie.split('; ')
        .find(row => row.startsWith('csrftoken='))
        ?.split('=')[1];
}

$("#botaoLike").click( function() {
  $.ajax(
    url: "{% url 'ajax_like_livro' %}", // só funciona dentro do template
    data: { //dados
      csrfmiddlewaretoken: getCSRFToken(),
      id_livro: 6
    },
    success: function (resposta) {//callback de sucesso
      alert(resposta.mensagem)
      $("#numLikes").text(resposta.likes);
    },
    error: function (xhr, status, error) => {//callback de erro
      alert(data.status);
    }
  );
});
```

POST

- Na *view* acessamos os dados recebidos normalmente;

```
def ajax_like_livro(request):
    if request.method == "POST":
        id_livro = request.POST.get("id_livro")
        livro = Livro.objects.get(id=id_livro)
        if livro:
            livro.likes = livro.likes + 1
            livro.save()
            return JsonResponse({"mensagem": "Like registrado!", "likes": livro.likes }, status=201)
        else:
            return JsonResponse({"mensagem": "Livro não encontrado!"}, status=404)
```

Forms

- Na prática quase sempre forms são usados para entrada de dados pelo usuário;
- Devemos aproveitar as funcionalidades do Django Forms;
- Acessamos os campos do form no JavaScript;
  - Objeto `FormData`;
    - Aceita arquivos;
    - Já pega o CSRF;
- Temos que impedir que o *submit* do form atualize a página.

Exemplo

- Há duas possibilidades para o form:
  - Ter sido carregado junto com a página no início:
    - Só precisamos ler e enviar os dados (POST);
  - Ser carregado também por AJAX:
    - Precisamos de duas ações;
    - Uma pra pedir o form (GET);
    - Outra pra enviar os dados (POST).

Exemplo form já carregado

```
$("#livroForm").submit( function (evento) {
  evento.preventDefault(); // evita a submissão "normal" do form
  $.ajax({
    url: {% url 'ajax_criar_livro' %},
    method: "POST",
    data: new FormData($("#livroForm")[0]), //cria o objeto FormData
    success: function (resposta) {
      $("#livroForm")[0].reset(); //limpa o form pra permitir nova submissao
      alert(resposta.mensagem);
    },
    error: function (xhr, status, error) {
      $("#livroForm").html(error); //substitui com o form do django (traz os erros)
      alert(resposta.mensagem);
    }
  });
});
```

Carregando form

```
$("#botaoNovoForm").click( function () {
  $.ajax({
    url:{% url 'ajax_criar_livro' %},
    method: "GET",
    success: function (resposta) {
      $(#divForm).append(resposta.form);
    },
    error: function (xhr, status, error) {
      alert("Erro ao carregar dados");
    }
  })
})
```

- O resto é igual ao anterior.

No Django

- View

```
def ajax_criar_livro(request):
    if request.method == 'POST':
        form = LivroForm(request.POST)
        if form.is_valid():
            form.save()
            return JsonResponse({"mensagem": "Livro criado com sucesso"}, status=201)
        else:
            status = 400
    else:
        status = 200
        form = LivroForm()
    return render(request, "partials/_criar_livro_form.html", {"form": form}, status=status)
```

Autenticação e Autorização

- As mesmas regras de autorização e autenticação valem para as *views* AJAX;
- O AJAX também recebe/envia as informações de usuário e permissões pelo request;
- É importante proteger as views AJAX, porém os decorators não funcionam bem com AJAX;
  - É possível verificar `request.user.is_authenticated` na *view* e retornar o status 401 *Unauthorized*;
  - E verificar o `request.user.has_perms("app.permissao")` da mesma forma, mas com erro 403 *Forbidden*;

Autenticação e Autorização

- O JavaScript deve tratar os erros;
- Agir de acordo com o código;
  - Informar o erro em uma mensagem/alert/modal;
  - O redirecionamento não é interessante no caso;

Exemplo

```
def view_protegida(request):
    if not request.user.is_authenticated:
        return JsonResponse("mensagem": "Usuário não autenticado!", status=401)
    if not request.user.has_perm("app.permissao"):
        return JsonResponse("mensagem": "Usuário não autorizado!", status=403)

    # resto da view
```

Exemplo

```
$.ajax({
  url: "url/view_protegida",
  method: "GET",
  success: (resposta) => {
    ...
  },
  error: (xhr, status, error) => {
    const resposta = xhr.responseJSON;
    alert(resposta.mensagem);
  }
})
```

Mensagens Django

- O *framework* de mensagens Django não funciona diretamente com AJAX;
- É necessário implementar a funcionalidade;
  - Uma *view* que retorna apenas as mensagens;
  - Uma função JS que requisita as mensagens ao *back-end* e apresenta na tela;
- Podemos usar a mesma lógica de *partials* para manter a consistência;
- Obviamente há outras formas de fazer isso.

Mensagens Django

- View de mensagens AJAX:

```
from django.contrib.messages import get_messages
...
def ajax_mensagens(request):
    messages = get_messages(request)
    return render(request, 'partials/_messages.html', {'messages': messages})
```

Mensagens Django

- Template parcial (usando Bootstrap Alerts):

```
{% if messages %}
  {% for message in messages %}
  <div class="alert alert-{{message.tags}} alert-dismissible fade show" role="alert">
    {{message}}
    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>
  </div>
  {% endfor %}
{% endif %}
```

Mensagens Django

- Template base:

```
...
<div id="div-mensagens">
{% include 'partials/_messages.html' %}
</div>
```

Mensagens Django

- JS:

```
function buscarMensagens() {
  $.get("url das mensagens",
    (resposta) => {
      $("#div-mensagens").html(resposta);
    }
  );
}
```

- Essa função deve ser chamada sempre que quisermos atualizar as mensagens depois de um request AJAX;

Exemplo

```
$.ajax({
  url: "ler_livro/2",
  method: "GET",
  success: (resposta) => {
    ...
    buscarMensagens(); //atualiza as mensagens
  },
  error: (xhr, status, error) => {
    ...
    buscarMensagens(); //atualiza as mensagens
  }
});
```

Modais

- Uma das formas de exibir informação dinâmica é com o uso de *modais*;
- Janelas pop-up que sobrepõem o conteúdo da página;
- Bootstrap
- Usam os atributos `data-bs-toggle="modal"` e `data-bs-target="#id-do-modal"` no botão que abre o modal;
- O Bootstrap5 não usa mais jQuery.

Modais

- Para controlar o modal usando JS Vanilla:

```
const meuModal = new bootstrap.Modal(document.getElementById('id-do-modal'));
meuModal.show();
```

Latência

- A ordem que as coisas acontecem importa muito no AJAX;
- As requisições podem demorar a responder;
- A página pode carregar de forma estranha;
- A experiência de usuário fica comprometida.

Latência

- Nem sempre dá pra perceber durante o desenvolvimento;
- Quando é feito o *deploy* nada mais funciona direito;
- Verifique sempre o console do navegador para ver os erros de JS;
- Teste adicionando um `time.sleep(tempo_em_segundos)` na view;
  - Precisa importar `import time` antes.
- Não esqueça de remover isso antes de fazer o commit!!!

CRUD completo com AJAX

- Exemplo completo com tudo isso:
  - Github

AJAX sem jQuery

- You might not need jQuery
- Usamos a função `fetch()` do JS *Vanilla*;
- A manipulação do DOM é como vimos na aula anterior;
- Usaremos na próxima disciplina.

AJAX sem jQuery

```
async function lerLivro(){
  try {
    const response = await fetch('url', {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json'
      },
    });

    const data = await response.json();

    const minhaDiv = document.querysetSelector("#minhaDiv");
    minhaDiv.innerHTML = `
      <h1>${data.titulo}</h1>
      <h2>${data.autor}</h2>
    `
  } catch (erro) {
    alert(erro)
  }
}

lerLivro();
```

Referências

- https://api.jquery.com
- https://docs.djangoproject.com/en/5.1/topics/serialization/

Dúvidas?