16.1. Implementando uma chamada na REST API com JavaScript

sexta-feira, 7 de abril de 2023 09:18

- index.html

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8>
  <style>
    button {
      color: #fff;
      background-color: #007bff;
      border: none;
      font-size: 1rem;
      padding: 10px;
    }

    #conteudo {
      margin-top: 10px;
      font-family: Arial;
      font-size: 1rem;
    }
  </style>
</head>
<body>
  <button id="botao" class="btn btn-primary">Fazer requisição</button>
  <div id="conteudo"></div>

  <script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>
  <script src="client.js"></script>
</body>
</html>
```

- client.js

client.js

```
function consultarRestaurantes(){
    $.ajax({
        url: "http://localhost:8080/restaurantes",
        type: "get",

        success: function(response){
            $("#conteudo").text(response);
        }
    });
}

$("#botao").click(consultarRestaurantes);
```

16.2. Testando a requisição na API com JavaScript e entendendo a Same Origin Policy

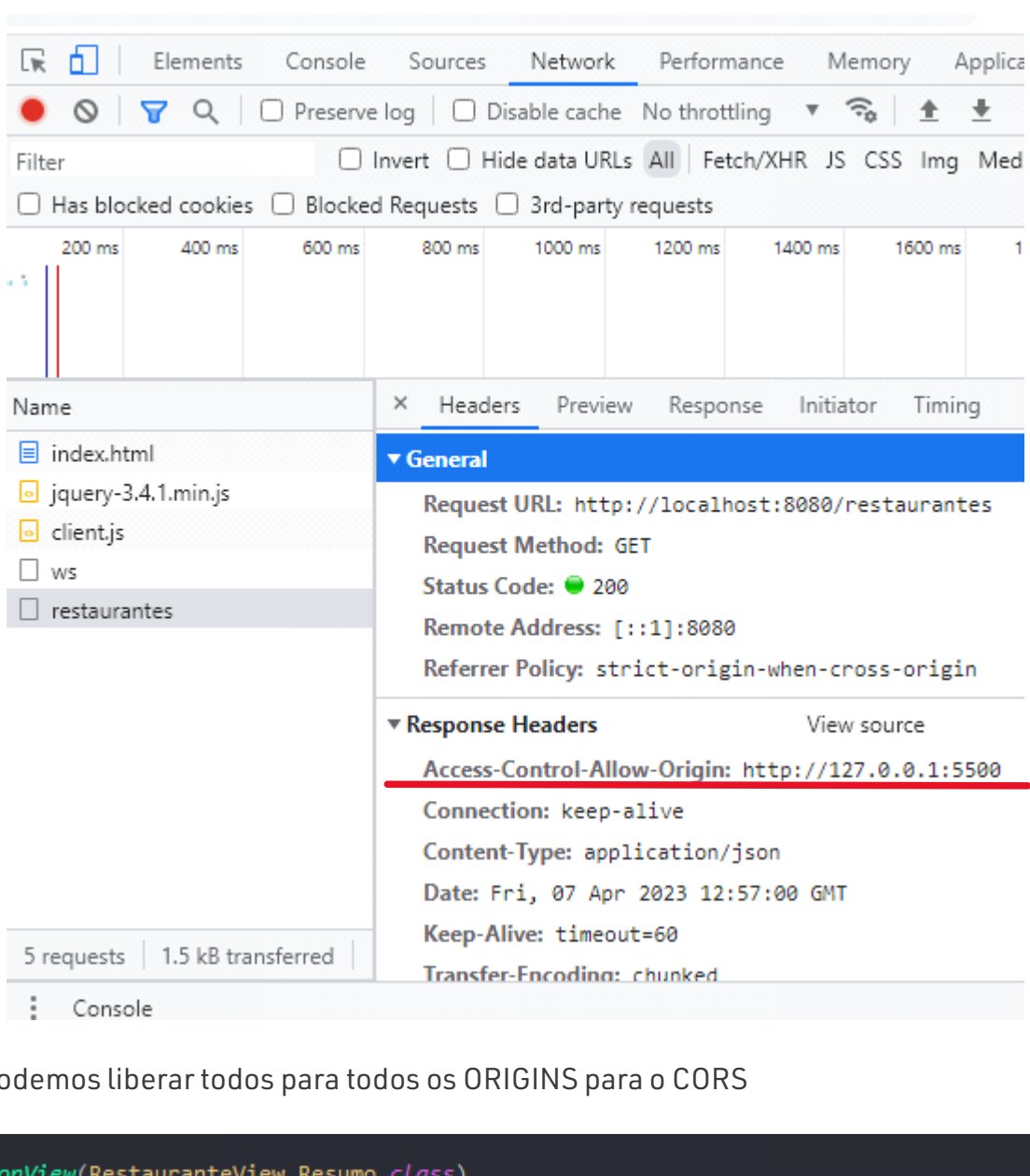
sexta-feira, 7 de abril de 2023 09:37

16.3. Entendendo o funcionamento básico de CORS e habilitando na API

sexta-feira, 7 de abril de 2023 09:49

- Adicionar manualmente no cabeçalho da requisição

```java
@JsonView(RestauranteView.Resumo.class)
@GetMapping
public ResponseEntity<List<RestauranteDTO>> listar() {
    final List<RestauranteDTO> restauranteDTOS = rAssembler.toListDTO(restauranteService.listar());

    return ResponseEntity.ok()
            .header(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, ...headerValues: "http://127.0.0.1:5500")
            .body(restauranteDTOS);
}
```



ou podemos liberar todos para todos os ORIGINS para o CORS

```java
@JsonView(RestauranteView.Resumo.class)
@GetMapping
public ResponseEntity<List<RestauranteDTO>> listar() {
    final List<RestauranteDTO> restauranteDTOS = rAssembler.toListDTO(restauranteSer

    return ResponseEntity.ok()
            .header(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN, ...headerValues: "*")
            .body(restauranteDTOS);
}
```

cliente

```
index.html   client.js
client.js > consultarRestaurantes > success
1   function consultarRestaurantes(){
2       $.ajax({
3           url: "http://localhost:8080/restaurantes",
4           type: "get",
5
6           success: function(response){
7               $("#conteudo").text(JSON.stringify(response));
8           }
9       });
10  }
11
12  $("#botao").click(consultarRestaurantes);
```

16.4. Habilitando CORS em controladores e métodos com @CrossOrigin

sexta-feira, 7 de abril de 2023 10:04

- Habilitar o CORS usando o suporte nativo do Spring usando @CrossOrigin
  - Anotação de nível de Método e Classe
  - Pode passar um origin ou vários entre {}

```
@CrossOrigin(origins = "http://127.0.0.1:5500")
@JsonView(RestauranteView.Resumo.class)
@GetMapping
public List<RestauranteDTO> listar() {
    final List<Restaurante> restaurantes = restauranteService.listar();

    return rAssembler.toListDTO(restaurantes);
}

@JsonView(RestauranteView.ResumoApenasNome.class)
@GetMapping(params = "projecao=apenas-nome")
public List<RestauranteDTO> listarApenasNome() {

    return listar();
}
```

Resposta

Request URL: http://localhost:8080/restaurantes
Request Method: GET
Status Code: 403
Referrer Policy: strict-origin-when-cross-origin

A diferença é que o servidor fazia a busca no banco de dados e processava a resposta, mas agora o próprio Spring não deixa fazer a requisição no controlador, pois o Origin não é o mesmo, ele faz a validação e envia uma resposta com o código 403 Forbidden a qual indica que o servidor recebeu a requisição e foi capaz de identificar o autor, porém não autorizou a emissão de um resposta.

```
2 usages
41  @CrossOrigin
42  @RestController
43  @RequestMapping("/restaurantes")
44  public class RestauranteController {
45
46      @Autowired
```

Para liberar todos os CROS origins, basta adicionar o asterisco * no valor de origin. Porém, por padrão, somente @CrossOrigin já libera todos as origens.

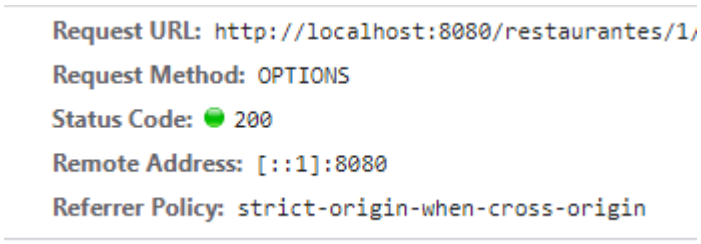
```
2 usages
@CrossOrigin(origins = "*")
@RestController
@RequestMapping("/restaurantes")
public class RestauranteController {
```

16.5. Entendendo o funcionamento de preflight do CORS

sexta-feira, 7 de abril de 2023 11:08

uma requisição preflight de CORS é uma requisição de CORS que verifica se o protocolo CORS é entendido e se o servidor aguarda o método e cabeçalhos('headers') especificados.

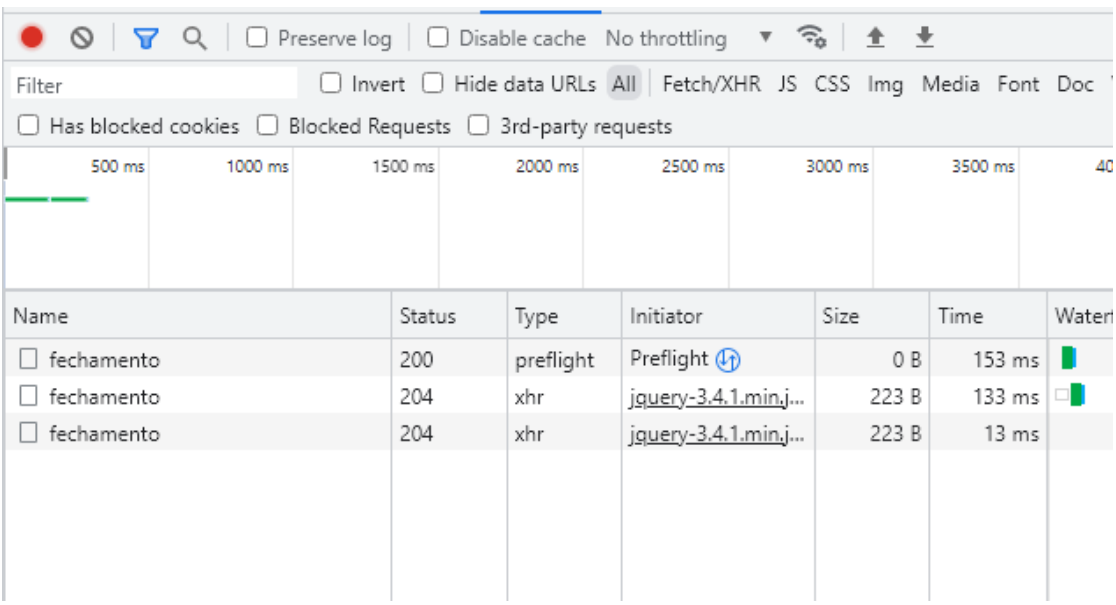
Ou seja, basicamente o cliente (nesse exemplo a aplicação javascript) pergunta ao servidor se este permitiria uma requisição PUT, antes de enviá-la.



Logo, o cliente faz 2 requisições para o servidor afim de verificar a requisição 'complexa' por ser um put.

Requisição

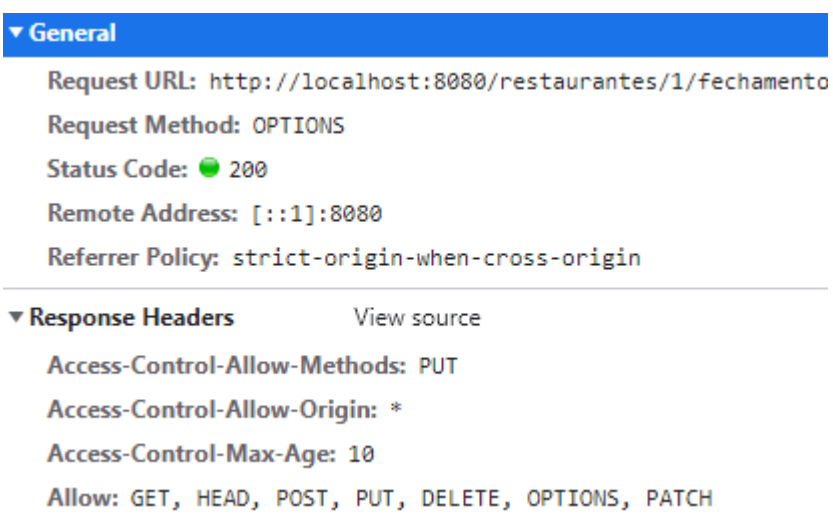
```
function fecharRestaurante() {
  $.ajax({
    url: "http://api.algafood.local:8080/restaurantes/1/fechamento",
    type: "put",

    success: function(response) {
      alert("Restaurante foi fechado!");
    }
  });
}

$("#botao").click(fecharRestaurante);
```



primeiro a requisição preflight para verificar o cross origin aceito pelo servidor, depois outra requisição onde não houve preflight porque definimos o tempo que o browser pode armazenar o cache do preflight

```
2 usages
@CrossOrigin(maxAge = 10)
@RestController
@RequestMapping("/restaurantes")
public class RestauranteController {
```

Resposta da requisição preflight



No cabeçalho da resposta vinda do servidor, deve constar o tipo de método, as origens aceitas, o máximo de tempo em segundos que o browser guarda o cache da resposta preflight e os verbos aceitos.

As condições de uma requisição simples podem ser vistas na documentação oficial

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Simple_requests

também podemos adicionar headers específicos para serem aceitos pelo CORS do servidor

```
2 usages
@CrossOrigin(maxAge = 1800,allowedHeaders = "*")
@RestController
@RequestMapping("/restaurantes")
public class RestauranteController {
```

6.6. Habilitando CORS globalmente no projeto da API

sexta-feira, 7 de abril de 2023 14:44

- Podemos habilitar globalmente na aplicação o CORS para que não precisemos definir em cada método ou em cada classe controladora

```java
package com.algaworks.algafood.core.web;

import ...

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping( pathPattern: "/restaurantes/**")
                .allowedMethods("GET","PUT");
    }
}
```

O método registry.addMapping() é usado para definir as URLs que serão tratadas pelas configurações CORS. Ele permite que você especifique os endpoints do seu controlador que serão afetados pelas políticas CORS. Você pode usar allowedMethods(), allowedHeaders(), exposedHeaders(), allowCredentials(), maxAge() e allowedOrigins() para configurar as políticas de CORS para esses endpoints.

Por outro lado, o método allowedOrigins() é usado para especificar os domínios que têm permissão para acessar os recursos da sua aplicação. Ele é usado em conjunto com registry.addMapping() para definir as configurações CORS para os endpoints especificados.

Em resumo, registry.addMapping() é usado para especificar os endpoints que terão suas configurações CORS definidas, enquanto allowedOrigins() é usado para permitir que domínios específicos acessem esses endpoints. Ambos os métodos são importantes para garantir que sua aplicação Spring MVC seja segura e tenha uma política de CORS bem definida.

O método PUT de um endpoint adicionado em registry.addMapping() pode ser bloqueado pelo CORS por padrão, porque ele é considerado um método de solicitação "não seguro" para a maioria dos navegadores. Por outro lado, o método GET não é bloqueado porque é considerado um método de solicitação "seguro".

O CORS é um mecanismo de segurança do navegador que impede que um site mal-intencionado acesse recursos de outros domínios sem permissão. Quando um navegador faz uma solicitação para um recurso em outro domínio, ele faz uma verificação prévia de CORS para garantir que a resposta da solicitação tenha permissão para ser compartilhada com o domínio atual.

O método PUT é considerado "não seguro" porque ele pode alterar o estado do servidor, o que pode levar a problemas de segurança se o recurso for acessado por um site mal-intencionado. Portanto, por padrão, o CORS bloqueia solicitações PUT de outros domínios, a menos que o servidor tenha explicitamente permitido o acesso usando as políticas de CORS apropriadas.

Por outro lado, o método GET é considerado "seguro" porque ele só recupera informações do servidor, sem alterar nenhum estado. Por esse motivo, o CORS geralmente não bloqueia solicitações GET de outros domínios.

No entanto, é importante notar que o CORS é uma medida de segurança baseada em navegador e pode ser desativado ou contornado em outros contextos. Portanto, é importante garantir que as políticas de segurança de sua aplicação sejam adequadas para proteger contra ameaças potenciais.

```java
package com.algaworks.algafood.core.web;

import ...

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping( pathPattern: "/**")
                .allowedMethods("*");
    }
}
```

CORS habilitado para todos os endpoints da aplicação e todos os métodos HTTP

16.7. Desafio: implementando uma requisição GET com JavaScript

sexta-feira, 7 de abril de 2023 18:52

JS client.js > ...

```js
function consultar() {
    $.ajax({
        url: "http://localhost:8080/formapagamentos",
        type: "get",

        success: function(response) {
            preencherTabela(response);
        }
    });
}

function preencherTabela(formasPagamento) {
    $("#tabela tbody tr").remove();

    $.each(formasPagamento, function(i, formaPagamento) {
        var linha = $("<tr>");

        linha.append(
            $("<td>").text(formaPagamento.id),
            $("<td>").text(formaPagamento.descricao)
        );

        linha.appendTo("#tabela");
    });
}


$("#btn-consultar").click(consultar);
```

```html
<div class="container">
    <div class="row">
        <button id="btn-consultar" class="btn btn-primary">Consultar</button>
    </div>

    <div class="row">
        <table id="tabela" class="table table-striped table-hover">
            <thead class="thead-dark">
            <tr>
                <th>Código</th>
                <th>Descrição</th>
            </tr>
            </thead>
            <tbody>
            <tr>
                <td colspan="2">Nenhuma forma de pagamento encontrada</td>
            </tr>
            </tbody>
        </table>
    </div>
</div>
```

16.10. Implementando um client da REST API com Java e Spring (RestTemplate)

sexta-feira, 7 de abril de 2023 18:54

```java
@AllArgsConstructor
public class RestauranteClient {
    1 usage
    private static final String RESOURCE_PATH = "/restaurantes";
    1 usage
    private String url;
    1 usage
    private RestTemplate restTemplate;

    1 usage
    public List<RestauranteResumoDto> listar(){

        URI restauranteUri = URI.create(url + RESOURCE_PATH);
        final RestauranteResumoDto[] restauranteResumoDtos =
                restTemplate.getForObject(restauranteUri, RestauranteResumoDto[].class);

        assert restauranteResumoDtos != null;
        return Arrays.asList(restauranteResumoDtos);
    }
}
```

```java
public class ListagemRestaurantesMain {
    public static void main(String[] args) {

        final RestauranteClient restauranteClient =
                new RestauranteClient( url: "http://localhost:8080", new RestTemplate());

        final List<RestauranteResumoDto> restaurantes = restauranteClient.listar();

        restaurantes.forEach(System.out::println);
    }
}
```

16.11. Tratando respostas com código de erro no client Java

sábado, 8 de abril de 2023 10:11

https://app.algaworks.com/aulas/2104/tratando-respostas-com-codigo-de-erro-no-client-java?pagina=0