Aula 10 - Uso de APIs e Tratamento de Erros

aTip Learn - Lógica de Programação com JavaScript

Resumo da Aula

Nesta aula, aprenderemos sobre APIs REST, o que são e como funcionam. Veremos como realizar requisições HTTP com JavaScript para consumir dados de APIs externas. Aprenderemos a processar dados de APIS no formato JSON, exibindo informações recebidas na tela do navegador. Além disso, iremos um pouco mais a fundo no tratamento de erros com `try...catch`, para lidar com falhas de comunicação e validação de dados.

Objetivos

- Entender o que são APIs REST e como funcionam.
- Realizar requisições HTTP com JavaScript utilizando os métodos `GET`, `POST`, `PUT` e `DELETE`.
- Processar dados de APIs no formato JSON.
- Exibir informações recebidas de APIs na tela do navegador.
- Utilizar `try...catch` para tratar erros de comunicação e validação de dados.

Conteúdo

Formulários em Páginas HTML

Formulários são elementos HTML que permitem a coleta de dados de usuários. Eles são compostos por diversos tipos de campos, como campos de texto, caixas de seleção, botões de opção, entre outros. Para criar um formulário em uma página HTML, utilizamos a tag `<form>`, que pode conter diversos campos de entrada de dados, como `<input>`, `<select>`, `<textarea>`, entre outros.

```html
<form id="formularioLogin">
  <div>
    <label for="nome">EMail:</label>
    <input type="password" id="email" name="email">
  </div>
  <div>
    <label for="senha">Senha:</label>
    <input type="password" id="senha" name="senha">
  </div>
  <button type="submit">Enviar</button>
</form>
```

Nesse exemplo temos um formulário com dois campos de entrada de texto, um para o email e outro para a senha. Cada campo possui um elemento `label` associado a ele, que é utilizado para descrever o campo. O botão `Enviar` é do tipo `submit`, que é utilizado para enviar os dados do formulário para um servidor.

Por padrão, quando o botão `submit` é clicado, o formulário é enviado pelo navegador diretamente para o servidor, recarregando a página. Esse comportamento padrão normalmente não é o que desejamos quando estamos trabalhando com JavaScript, pois queremos enviar os dados do formulário para uma API sem recarregar a página. Para evitar esse comportamento padrão, devemos adicionar um *event listener* para o evento `submit` do formulário e então chamar o método `preventDefault()` no evento para cancelar o envio padrão do formulário.

```javascript
const formularioLogin = document.getElementById('formularioLogin');

formularioLogin.addEventListener('submit', function (event) {
  event.preventDefault();
  // código para validar e enviar dados do formulário para uma API
});
```

Toda função utilizada como *event listener* recebe um parâmetro `event`, que é um objeto que contém informações e métodos relacionados ao evento que foi disparado. O método `preventDefault()` é utilizado para cancelar o comportamento padrão do evento, que no caso do evento `submit` é enviar o formulário para o servidor.

Para mais informações sobre o método `preventDefault()`, consulte a documentação oficial da Mozilla.

Acessando Dados do Formulário

Antes de enviar dados para uma API, muitas vezes o primeiro passo é obter esses dados de um formulário preenchido pelo usuário na página HTML. Isso pode ser feito obtendo uma referência para os campos do formulário e acessando a propriedade que armazena o valor do campo. Por exemplo, para obter o valor de um campo de texto, podemos fazer:

```
const campoTexto = document.getElementById('campo-texto');
const valorCampoTexto = campoTexto.value;
```

O método `getElementById()` é utilizado para obter uma referência para um elemento HTML com um determinado `id`. O valor de um campo de texto é armazenado na propriedade `value` do elemento.

Validação de Dados do Formulário

Combinando os conceitos anteriores temos as ferramentas necessárias para validar os dados de um formulário no momento do envio. Podemos obter os valores dos campos do formulário, validar esses valores e exibir mensagens de erro para o usuário caso os dados sejam inválidos. No exemplo do nosso `formularioLogin`, podemos validar se os campos de email e senha estão preenchidos antes de enviar o formulário:

```
function validarCampoObrigatorio(nomeCampo) {
  const campo = document.getElementById(nomeCampo);
  if (campo.value === '') {
    alert(`Por favor, preencha o campo ${nomeCampo}.`);
    return false;
  }
  return true;
}

function validarFormulario() {
  if (validarCampoObrigatorio('email') === false) {
    return false;
  }
  if (validarCampoObrigatorio('senha') === false) {
    return false;
  }

  return true;
}

function loginSubmitHandler(event) {
  event.preventDefault();
  if (validarFormulario()) {
    // enviar dados do formulário para uma API
  }
}

const formularioLogin = document.getElementById('formularioLogin');
formularioLogin.addEventListener('submit', loginSubmitHandler);
```

Envio de Dados do Formulário

Após validarmos os dados do formulário, podemos enviar esses dados para uma API utilizando a função `fetch()`. Na última aula vimos como obter dados de uma API com essa função, agora veremos como enviar dados para uma API utilizando o método `POST`.

```
async function enviarDadosFormulario() {
  const dadosFormulario = {
    email: document.getElementById('email').value,
    senha: document.getElementById('senha').value,
  };

  const dadosJSON = JSON.stringify(dadosFormulario);

  const url = 'https://api.com/login';

  const resposta = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: dadosJSON,
  });

  if (resposta.ok) {
    alert('Login realizado com sucesso!');
  } else {
    alert('Erro ao realizar login.');
  }
}
```

Nesse exemplo, criamos um objeto `dadosFormulario` com os valores dos campos de email e senha do formulário. Em seguida convertemos esse objeto em uma string no formato JSON utilizando o método `JSON.stringify()`. Armazenamos a URL da nossa API na variável `url` e então utilizamos a função `fetch()` para enviar os dados do formulário para a API. Diferente da aula passada, nessa chamada para a função `fetch()` passamos um segundo argumento com um objeto de configuração para a requisição, esse objeto especifica o método da requisição (`method`), os cabeçalhos da requisição (`headers`) e o corpo da requisição (`body`).

Utilizamos o método `POST` para enviar os dados do formulário para a API, nos cabeçalhos da requisição adicionamos um cabeçalho `Content-Type` com o valor `application/json` para indicar que o corpo da requisição está no formato JSON. Por fim atribuimos a string JSON ao corpo da requisição.

APIs REST

As APIs REST são um estilo de arquitetura de APIs que segue um conjunto de regras e convenções para comunicação entre sistemas na Internet. Essas APIs são baseadas no protocolo HTTP, que é o protocolo utilizado para comunicação entre navegadores Web e servidores, e se utilizam das ferramentas desse protocolo para realizar operações de leitura, escrita, atualização e exclusão de dados.

Métodos HTTP Uma das principais características das requisições feitas para APIs REST é o uso dos métodos HTTP para indicar a operação que deve ser realizada. Os métodos mais comuns são:

- `GET`: Utilizado para obter dados de um recurso.
- `POST`: Utilizado para criar um novo recurso.
- `PUT`: Utilizado para atualizar um recurso existente.
- `DELETE`: Utilizado para excluir um recurso.

Por exemplo, se quisermos criar um novo usuário em uma API REST, podemos enviar uma requisição `POST` para a URL da API com os dados do novo usuário no corpo da requisição. Se quisermos obter informações de um usuário específico, podemos enviar uma requisição `GET` para a URL da API com o ID do usuário na URL, e assim por diante.

Formato JSON Outra característica comum das APIs REST é o uso do formato JSON para representar os dados que são enviados e recebidos pelas APIs. O JSON é um formato de texto simples que é fácil de ler e escrever, e é amplamente utilizado para representar dados estruturados em APIs REST. Seu formato lembra muito a estrutura de objetos em JavaScript, o que facilita a conversão entre os dois formatos. Veja por exemplo o JSON que representa um usuário:

```json
{
  "id": 1,
  "nome": "João",
  "email": "joao@acme.com",
  "idade": 30,
  "habilitado": true
}
```

Uma das principais diferenças entre um objeto do JavaScript e um objeto JSON é que as chaves de um objeto JSON devem ser envolvidas por aspas duplas, enquanto em um objeto JavaScript as aspas são opcionais. Já os valores de um objeto JSON também são um pouco mais limitados que os valores de objetos JavaScript, em um objeto JSON podemos ter valores dos tipos string, número, booleano, objeto, array e `null`.

Em JavaScript podemos utilizar os métodos `JSON.parse()` e `JSON.stringify()` para converter entre objetos JavaScript e strings JSON. O método `JSON.parse()` é utilizado para converter uma string JSON em um objeto JavaScript, e o método `JSON.stringify()` é utilizado para converter um objeto JavaScript em uma string JSON.

Recursos em APIs REST

Os recursos em APIs REST são os objetos que representam os dados que são manipulados pela API. Cada recurso é identificado por uma URL única, que é utilizada para acessar, criar, atualizar e excluir o recurso. Por exemplo, se quisermos acessar um recurso de usuários em uma API REST, podemos utilizar a URL `https://api.com/usuarios` para acessar a lista de usuários, e a URL `https://api.com/usuarios/1` para acessar o usuário com ID 1. Ambas as requisições são feitas utilizando o método `GET`, mas enquanto a primeira retorna uma lista de usuários, a segunda retorna um único usuário.

Exemplo de Requisição GET

Vamos ver um exemplo de como fazer uma requisição `GET` para uma API REST que retorna uma lista de usuários. Nesse exemplo, utilizaremos a API JSONPlaceholder, que é uma API de teste que retorna dados fictícios em formato JSON.

```javascript
async function obterUsuarios() {
  const url = 'https://jsonplaceholder.typicode.com/users';

  const resposta = await fetch(url);

  if (resposta.ok) {
    const usuarios = await resposta.json();
    console.log(usuarios);
  } else {
    console.error('Erro ao obter usuários.');
  }
}
```

Nesse exemplo não utilizamos um segundo argumento na chamada para a função `fetch()`, dessa forma a função `fetch()` utiliza por padrão o método `GET`. Também não precisamos especificar um cabeçalho ou um corpo da requisição, pois não estamos enviando dados para a API.

Após obter a resposta da API, verificamos se a resposta foi bem sucedida utilizando a propriedade `ok` do objeto `Response`. A propriedade `ok` é um booleano que indica se a resposta da API foi bem sucedida ou não. Se a resposta foi bem sucedida, utilizamos o método `json()` do objeto `Response` para converter a resposta em um objeto JavaScript. O método `json()` é uma função assíncrona que retorna uma *promise* que é resolvida com o corpo da resposta convertido em um objeto JavaScript, não sendo necessário utilizar o método `JSON.parse()` nesse caso.

Códigos de Status HTTP

No exemplo anterior utilizamos o campo booleano `ok` para verificar se a resposta foi bem sucedida ou não. Esse campo é uma forma simplificada de verificar o código de status HTTP da resposta, que é uma informação mais detalhada sobre o resultado da requisição. Os códigos de status HTTP são números de três dígitos que indicam o resultado da requisição, sendo divididos em cinco classes:

- `1xx`: Informacional
- `2xx`: Sucesso
- `3xx`: Redirecionamento
- `4xx`: Erro do Cliente
- `5xx`: Erro do Servidor

Alguns dos códigos de status mais comuns são:

- `200 OK`: Indica que a requisição foi bem sucedida.
- `201 Created`: Indica que um novo recurso foi criado.
- `400 Bad Request`: Indica que a requisição possui um erro.
- `401 Unauthorized`: Indica que o usuário não está autenticado.
- `403 Forbidden`: Indica que o usuário não tem permissão para acessar o recurso.
- `404 Not Found`: Indica que o recurso não foi encontrado.
- `500 Internal Server Error`: Indica um erro interno no servidor.

Para mais informações sobre códigos de status HTTP, consulte a documentação oficial da Mozilla.

Tratamento de Erros em Requisições

Quando verificamos a propriedade `ok` ou a propriedade `status` de um objeto `Response`, isso indica que a comunicação com o servidor utilizando o protocolo HTTP foi bem sucedida, ou seja, foi possível enviar uma requisição e receber uma resposta à essa requisição.

Entretanto, uma requisição pode falhar sem que seja possível receber uma resposta, isso pode ocorrer por diversos motivos, como problemas de conexão com a Internet, problemas no servidor, ou erros de programação. Para lidar com esses erros, devemos utilizar um bloco `try...catch` para capturar exceções que ocorrem durante a execução da função `fetch()`.

```
async function obterUsuarios() {
  const url = 'https://jsonplaceholder.typicode.com/users';

  try {
    const resposta = await fetch(url);

    if (resposta.ok) {
      const usuarios = await resposta.json();
      console.log(usuarios);
    } else {
      console.error('Resposta de erro ao obter usuários.');
```

```
    }
  } catch (erro) {
    console.error('Erro de conexão ao obter usuários:', erro);
  }
}
```

No exemplo acima, utilizamos um bloco `try...catch` para envolver a chamada para a função `fetch()`. Se ocorrer um erro durante a execução da função `fetch()`, o bloco `catch` será executado, e o erro será armazenado na variável `erro`. Podemos então exibir uma mensagem de erro para o usuário, ou realizar alguma outra ação para lidar com o erro.

Validação das Respostas da API

Além de verificar se a requisição foi bem sucedida utilizando a propriedade `ok` e tratar possíveis erros de conexão, é também importante validar o conteúdo da resposta da API. Por exemplo, se esperamos receber um objeto com um determinado formato, podemos verificar se a resposta possui esse formato antes de utilizá-la.

```
async function obterUsuarios() {
  const url = 'https://jsonplaceholder.typicode.com/users';

  try {
    const resposta = await fetch(url);

    if (resposta.ok) {
      const usuarios = await resposta.json();

      if (Array.isArray(usuarios)) {
        console.log(usuarios);
      } else {
        console.error('Resposta inválida ao obter usuários.');
      }
    } else {
      console.error('Resposta de erro ao obter usuários.');
    }
  } catch (erro) {
    console.error('Erro de conexão ao obter usuários:', erro);
  }
}
```

Nesse exemplo, apenas validamos se o conteúdo da resposta é um array. Além disso seria possível validar cada item do array individualmente para assegurar que cada item possui o tipo e a estrutura esperada.

```
function validarUsuario(usuario) {
  if (typeof usuario !== 'object' || usuario === null) {
    return false;
  }
  if (typeof usuario.id !== 'number') {
    return false;
  }
  if (typeof usuario.nome !== 'string') {
    return false;
  }
  if (typeof usuario.email !== 'string') {
    return false;
  }
  if (typeof usuario.idade !== 'number') {
```

```
    return false;
  }
  if (typeof usuario.habilitado !== 'boolean') {
    return false;
  }

  return true;
}
```

No exemplo acima, criamos a função `validarUsuario` que recebe um argumento `usuario` e então utiliza o operador `typeof` para verificar se o argumento é um objeto e se ele possui cada uma das propriedades esperadas com seu respectivo tipo de dados. Se o usuário passar em todas as validações, a função retorna `true`, caso contrário retorna `false`. Esse tipo de validação é importante para assegurar que os dados recebidos de uma API estão no formato esperado, e dessa forma evitar erros em funções que utilizam esses dados posteriormente.