

JavaScript



A classe **FormData**



A classe `FormData` do JavaScript é uma interface útil para construir um conjunto de pares chave/valor representando os campos de um formulário e seus valores, que pode ser facilmente enviada usando a interface `XMLHttpRequest` ou a interface `fetch`.

Função e Aplicação



Função Principal

A principal função da `FormData` é simplificar a coleta e a formatação dos dados de um formulário HTML para que possam ser enviados via requisições HTTP. Isso é particularmente útil para submissões assíncronas (AJAX), onde você deseja enviar os dados do formulário sem recarregar a página.

Aplicação

1. **Coleta de Dados do Formulário:** A `FormData` coleta automaticamente todos os campos do formulário (inputs, selects, textareas) e seus valores, incluindo arquivos.
2. **Envio de Dados:** É comumente usada em conjunto com `XMLHttpRequest` ou `fetch` para enviar dados para o servidor sem recarregar a página.

Métodos e Propriedades



A classe `FormData` fornece vários métodos úteis:

- **`append(name, value)`**: Adiciona um novo valor a uma chave existente ou cria uma nova chave se ela não existir.
- **`delete(name)`**: Remove todos os valores associados a uma chave.
- **`get(name)`**: Retorna o primeiro valor associado a uma chave.
- **`getAll(name)`**: Retorna todos os valores associados a uma chave.
- **`has(name)`**: Verifica se existe uma chave.
- **`set(name, value)`**: Define um valor para uma chave, substituindo os valores existentes.
- **`entries()`**: Retorna um iterador com todos os pares chave/valor.
- **`keys()`**: Retorna um iterador com todas as chaves.
- **`values()`**: Retorna um iterador com todos os valores.

Métodos e Propriedades



A classe `FormData` fornece vários métodos úteis:

- **`append(name, value)`**: Adiciona um novo valor a uma chave existente ou cria uma nova chave se ela não existir.
- **`delete(name)`**: Remove todos os valores associados a uma chave.
- **`get(name)`**: Retorna o primeiro valor associado a uma chave.
- **`getAll(name)`**: Retorna todos os valores associados a uma chave.
- **`has(name)`**: Verifica se existe uma chave.
- **`set(name, value)`**: Define um valor para uma chave, substituindo os valores existentes.
- **`entries()`**: Retorna um iterador com todos os pares chave/valor.
- **`keys()`**: Retorna um iterador com todas as chaves.
- **`values()`**: Retorna um iterador com todos os valores.

Métodos e Propriedades

JS

```
1  const formData = new FormData();
2  formData.append('username', 'johnDoe');
3  formData.append('email', 'john@example.com');
4
5  // Verificar se a chave existe
6  console.log(formData.has('username')); // true
7
8  // Obter valores
9  console.log(formData.get('username')); // johnDoe
10
11 // Substituir valor
12 formData.set('username', 'janeDoe');
13 console.log(formData.get('username')); // janeDoe
14
15 // Apagar uma chave
16 formData.delete('email');
17 console.log(formData.has('email')); // false
```

Conclusão

JS

- A classe `FormData` é uma ferramenta poderosa e flexível para coletar e enviar dados de formulário em aplicações web modernas. Com métodos convenientes para manipulação de dados e suporte nativo para arquivos, ela simplifica muitas tarefas comuns no desenvolvimento web, tornando o envio de formulários assíncronos mais eficiente e direto.

Função e Aplicação

JS

```
1  <form id="myForm">
2    <input type="text" name="username" value="johnDoe">
3    <input type="email" name="email" value="john@example.com">
4    <input type="file" name="profilePic">
5    <button type="submit">Submit</button>
6  </form>
7
8  <script>
9    const form = document.getElementById('myForm');
10   const formData = new FormData(form);
11
12   // Log para verificar o conteúdo
13   for (let [key, value] of formData.entries()) {
14     console.log(`${key}: ${value}`);
15   }
16 </script>
```


Fetch



O `fetch` é uma função moderna e nativa do JavaScript usada para fazer requisições HTTP (como GET, POST, PUT, DELETE) de forma assíncrona. Ela é baseada em Promises, o que facilita o trabalho com operações assíncronas de rede, como carregar dados de um servidor ou enviar dados para ele.

Vamos explorar `fetch` em detalhes:

Sintaxe Básica



A sintaxe básica do `fetch` é a seguinte:

```
fetch(url, options)
```

- **`url`**: O endpoint da requisição, ou seja, o endereço para onde a requisição será enviada.
- **`options`** (opcional): Um objeto que contém configurações adicionais para a requisição, como método HTTP, cabeçalhos, corpo da requisição, etc.

Exemplo de POST com fetch



- Para enviar dados para um servidor usando o método POST, você precisa incluir algumas opções adicionais, como o método HTTP e o corpo da requisição:

```
fetch('https://api.example.com/data', {  
  method: 'POST',  
  headers: { 'Content-Type': 'application/json' },  
  body: JSON.stringify({key1: 'value1',key2: 'value2'})  
}).then(response => response.json()).then(data =>  
  console.log(data)).catch(error => console.error('Erro:', error));
```

Explicação do Exemplo:

A yellow square with the letters 'JS' in black, representing JavaScript.

`fetch('https://api.example.com/data', { ... }):`

- Faz uma requisição HTTP POST para o URL fornecido com as opções especificadas.

`method: 'POST':`

- Define o método HTTP como POST.

`headers: { 'Content-Type': 'application/json' }:`

- Define os cabeçalhos da requisição. O `Content-Type` é definido como `application/json` para informar ao servidor que os dados no corpo da requisição estão no formato JSON.



Explicação do Exemplo:

`body: JSON.stringify({ ... }):`

- Define o corpo da requisição, que é convertido para uma string JSON usando `JSON.stringify`.

`.then(response => response.json()):`

- Converte a resposta para JSON.

`.then(data => console.log(data)):`

- Manipula os dados JSON da resposta.

`.catch(error => console.error('Erro:', error)):`

- Captura qualquer erro que tenha ocorrido durante o processo de requisição ou parsing do JSON.

Propriedades e Métodos do Objeto Response

JS

O objeto `Response` possui várias propriedades e métodos úteis:

- `response.ok`: Booleano que indica se a requisição foi bem-sucedida (status no intervalo 200-299).
- `response.status`: O código de status HTTP da resposta.
- `response.statusText`: O texto associado ao status HTTP.

Propriedades e Métodos do Objeto Response

JS

- **response.json()**: Retorna uma Promise que resolve com os dados JSON.
- **response.text()**: Retorna uma Promise que resolve com os dados como uma string de texto.
- **response.blob()**: Retorna uma Promise que resolve com os dados como um **Blob** (útil para arquivos).
- **response.arrayBuffer()**: Retorna uma Promise que resolve com os dados como um **ArrayBuffer**.
- **response.headers**: Um objeto **Headers** associado à resposta.

Conclusão



- A função `fetch` é uma ferramenta poderosa para fazer requisições HTTP no JavaScript moderno. Sua utilização baseada em Promises permite lidar com operações assíncronas de forma clara e eficiente. Entender como configurar e usar `fetch` corretamente é essencial para qualquer desenvolvedor web que precise interagir com APIs e servidores.

Promises em JavaScript



- **O que são Promises?**
- Promises são objetos que representam a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante. Elas facilitam o trabalho com operações assíncronas, permitindo que você escreva código que espera pela conclusão dessas operações sem bloquear o fluxo do programa.

Promises em JavaScript



Estados de uma Promise

Uma Promise pode estar em um dos três estados:

1. **Pending (Pendente)**: Inicialmente, a Promise está neste estado, aguardando ser resolvida ou rejeitada.
2. **Fulfilled (Concluída)**: A operação assíncrona foi concluída com sucesso e a Promise tem um valor.
3. **Rejected (Rejeitada)**: A operação assíncrona falhou e a Promise tem um motivo de falha (erro).

Consumindo uma Promise

JS

1. Para consumir o resultado de uma Promise, você usa os métodos `.then()`, `.catch()` e `.finally()`.

```
1  const minhaPromise = new Promise((resolve, reject) => {
2    // Simulando uma operação assíncrona com setTimeout
3    setTimeout(() => {
4      const sucesso = true; // Alterne para false para simular uma falha
5      if (sucesso) {
6        resolve("Operação bem-sucedida!");
7      } else {
8        reject("Operação falhou.");
9      }
10   }, 1000);
11 });
12 minhaPromise
13   .then((resultado) => {
14     console.log(resultado); // "Operação bem-sucedida!"
15   })
16   .catch((erro) => {
17     console.error(erro); // "Operação falhou."
18   })
19   .finally(() => {
20     console.log("Operação concluída."); // Executado sempre, independentemente do sucesso ou falha
21   });
```

Promessa do Professor

JS

```
1  function getCoffeePromise() {
2      return new Promise((resolve, reject) => {
3          // Rejeitar a promessa para simular que o professor não pagou o café
4          reject(new Error("O professor não pagou o café!"));
5      });
6  }
7
8  getCoffeePromise()
9      .then(() => {
10         // Esta parte nunca será executada
11         console.log("O café foi pago!");
12     })
13     .catch((error) => {
14         // Contar a piada quando cair no catch
15         console.log("Por que o professor sempre promete café e nunca paga?");
16         setTimeout(() => {
17             console.log("Porque ele sempre deixa a conta 'pendente'! 😏");
18         }, 2000);
19     });
```

async e await em JavaScript

JS

1. O que são async e await?

`async` e `await` são palavras-chave que facilitam o trabalho com Promises, permitindo que você escreva código assíncrono de maneira mais síncrona e legível. `async` é usado para declarar uma função assíncrona, enquanto `await` é usado para esperar a resolução de uma Promise dentro de uma função `async`.

Funções async

JS

Uma função `async` sempre retorna uma Promise. Dentro de uma função `async`, você pode usar `await` para esperar a resolução de uma Promise.

```
1  async function minhaFuncao( ) {  
2    |    return "Hello, world!";  
3  }  
4  // Chamando a função  
5  minhaFuncao( ).then( (resultado) => console.log(resultado) ); // "Hello, world!"
```

Usando await

JS

`await` só pode ser usado dentro de funções `async` e faz com que a execução da função assíncrona pause até que a Promise seja resolvida ou rejeitada.

Exemplo

JS

```
1  async function minhaFuncaoAssincrona() {
2      try {
3          const resultado = await minhaPromise;
4          console.log(resultado); // "Operação bem-sucedida!"
5      } catch (erro) {
6          console.error(erro); // "Operação falhou."
7      } finally {
8          // Executado sempre, independentemente do sucesso ou falh
9          console.log("Operação concluída.");
10     }
11 }
12
13 // Chamando a função
14 minhaFuncaoAssincrona();
```


async/await



Usando `async/await`, o código se torna mais limpo e legível, pois parece mais com código síncrono:

```
1  async function minhaFuncaoAssincrona() {
2      try {
3          const resultado = await minhaPromise;
4          console.log(resultado);
5
6          const outroResultado = await outraPromise;
7          console.log(outroResultado);
8
9          const maisUmResultado = await maisUmaPromise;
10         console.log(maisUmResultado);
11     } catch (erro) {
12         console.error(erro);
13     }
14 }
15 // Chamando a função
16 minhaFuncaoAssincrona();
```

Conclusão

JS

- **Promises:** Representam uma operação assíncrona e permitem encadear ações a serem tomadas quando a operação é concluída ou falha. Utilizam métodos `.then()`, `.catch()` e `.finally()`.
- **async/await:** Tornam o código assíncrono mais legível e fácil de escrever, permitindo esperar pela resolução de Promises de uma maneira que parece síncrona. Funções `async` retornam Promises e dentro delas, você pode usar `await` para esperar pela resolução de outra Promise.

Dominar Promises e `async/await` é fundamental para escrever código JavaScript moderno e eficiente, especialmente quando se trata de operações que envolvem interações com APIs, bancos de dados, ou outras operações de rede.