async & await

THIAGO DELGADO PINTO

versão: 2022.07.05

aviso

esse conteúdo necessita de
boa compreensão sobre Promessas

revise o conteúdo anterior

introdução

`async` e **`await`** foram introduzidos no ES2017

visam **simplificar** o uso de **promessas**

torna código **assíncrono** parecido com **síncrono**

introdução

substitui o uso de explícito de **`Promise`** na função pela declaração **`async`**

troca uso de **`Promise.then`** por **`await`**

troca uso de **`Promise.catch`** por bloco **`try/catch`**

async function

Quando declaramos	ES cria cria objeto da classe
`function`	`Function`
`async function`	`AsyncFunction`

objetos da classe AsyncFunction

sempre retornam uma Promise

exemplo

```
async function x() {
   return 10;
}
```

equivale a

```
function x() {
   return new Promise( ( resolve, reject ) => {
      return resolve( 10 );
   } );
}
```

ou seja

a **implementação** da função é **automaticamente encapsulada** em uma **`Promessa`**

um **valor retornado** significa *cumprimento*

uma **exceção** significa *rejeição*

outro exemplo

```
async function x() {
  if ( Math.random() > 0.5 ) { return 10; }
  throw new Error( 'Ops' );
}
```

equivale a

```
function x() {
  return new Promise( ( resolve, reject ) => {
    if ( Math.random() > 0.5 ) { return resolve( 10 ); }
    return reject( new Error( 'Ops' ) );
  } );
}
```

uso do exemplo anterior – sem `await`

```
async function x() {
  if ( Math.random() > 0.5 ) { return 10; }
  throw new Error( 'Ops' );
}

x()
  .then( valor => console.log( valor ) )
  .catch( razao => console.error( razao ) );
```

await

é uma **facilidade sintática** do interpretador

ele **simula** código síncrono

mas **internamente** ele **gera código assíncrono**

`await` – exemplo

```
const valor = await x();
console.log( valor );
```

equivale a

```
x().then( ( valor ) => {
  console.log( valor );
} );
```

`await` – comportamento

código da linha igual ou posterior ao **`await`** é analisado pelo interpretador e ele monta uma expressão **`then`** contendo o código **dentro** de seu **`callback`**.

havendo outro **`await`**, outro **`then`** é criado

e assim por diante

`await` – outro exemplo

```
const valorX = await x();
const valorY = await y();
console.log( valorX + valorY );
```

equivale a

```
x().then( ( valorX ) => {
  y().then( ( valorY ) => {
    console.log( valorX + valorY );
  } );
} );
```

`await` – mais um exemplo

```
const valorX = await x();
const valorY = await y( valorX );
console.log( await z( valorY ) );
```

equivale a

```
x().then( ( valorX ) => {
  y( valorX ).then( ( valorY ) => {
    z( valorY ).then( ( tmpZ ) =>
    console.log( tmpZ );
  } );
} );
```

`await` – restrição no EcmaScript

a palavra await só pode ser usada dentro de uma async function

encapsulando

```
async function consultarFrutas() {
  if ( Math.random() > 0.5 ) {
    return [ 'Maçã', 'Laranja', 'Uva' ];
  }
  throw new Error( 'Sem frutas boas hoje.' );
}

( async () => {
  try {
    console.log( Frutas: ', await consultarFrutas() );
  } catch ( err ) {
    console.log( err.message );
  }
} )();
```

exemplo com fetch

```
async function consultarProdutos() {
  const response = await fetch( 'http://localhost/produtos' );
  if ( response.status >= 400 ) {
    throw new Error( 'Erro ' + response.status );
  }
  return await response.json(); // Pode melhorar...
}

( async () => {
  try {
    console.log( 'Produtos: ', await consultarProdutos() );
  } catch ( err ) {
    console.log( err.message );
  }
} )();
```

exemplo com fetch – com melhoria no retorno

```
async function consultarProdutos() {
  const response = await fetch( 'http://localhost/produtos' );
  if ( response.status >= 400 ) {
    throw new Error( 'Erro ' + response.status );
  }
  return response.json(); // Não é precisa await em return (!)
}

( async () => {
  try {
    console.log( 'Produtos: ', await consultarProdutos() );
  } catch ( err ) {
    console.log( err.message );
  }
} )();
```

exercício 1

Crie uma função assíncrona com ***async*** que retorne o número 100 após 3 segundos. Use ***setTimeout*** na solução. Invoque a função criada com ***await*** e imprima o número retornado.

exercício 2

Crie um servidor RESTFul simulado (com Json-Server) que contenha recursos em JSON em um arquivo **recursos.json**.

Nele, crie o recurso "**contatos**", que contenha objetos de contato com **id**, **nome** e **telefone**.

Então, crie uma função assíncrona que use *fetch* para consultar os contatos. Em uma outra função, criada por você, mostre os contatos recebidos em uma tabela cujas linhas sejam criadas dinamicamente.

Obs.: Consulte material de Promessas, em caso de dúvidas.

exercício 3

Com base no exercício anterior, crie um botão Remover que remova o contato selecionado na tabela através do envio de um DELETE para o servidor. Para isso, crie uma função assíncrona *removerContato*, que receba o id a ser removido. Faça também a remoção da linha da tabela correspondente, porém fora da função criada.

referências

MDN. *Async Function*. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction

MDN. *Funções Assíncronas*. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/funcoes_assincronas

fim

2022.07.05 – Adiciona slide 19. Faz pequenas melhorias de formatação.
2020.11.02 – Versão Inicial.