JavaScript TUTORIAL

JavaScript Top-level await

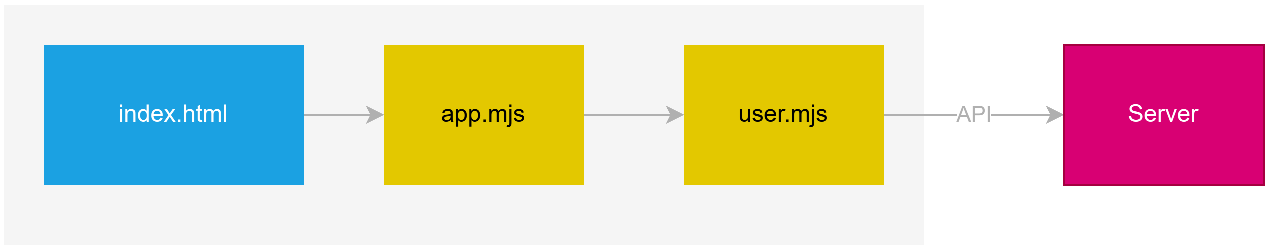
Summary: in this tutorial, you'll learn about the JavaScript top-level await and its use cases.

Introduction to the JavaScript top-level await

ES2020 introduced the top-level await feature, allowing a module to behave like an `async` function.

A module that imports the top-level await module will wait for it to load before evaluating its body.

To better understand the top-level await feature, we'll take an example:



In this example, we'll have three files: `index.html`, `app.mjs`, and `user.mjs`:

- The `index.html` uses the `app.mjs` file.
- The `app.mjs` imports the `user.mjs` file.
- The `user.mjs` fetches the user data in JSON format from an API with the URL endpoint https://jsonplaceholder.typicode.com/users

Here's the index file that uses the `app.mjs` module:

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript Top-Level Await Demo</title>
```

```
</head>

<body>
    <div class="container"></div>
    <script type="module" src="app.mjs"></script>
</body>

</html>
```

The following shows the `user.mjs` file:

```
let users;

(async () => {
  const url = 'https://jsonplaceholder.typicode.com/users';
  const response = await fetch(url);
  users = await response.json();
})();

export { users };
```

The `user.mjs` module uses the fetch API to get the users in JSON format from an API and export it.

Because we can only use the `await` keyword inside an `async` function (before ES2020), we need to wrap the API call inside an immediately invoked async function expression (IIAFE).

The following shows the `app.mjs` module:

```
import { users } from './user.mjs';

function render(users) {
  if (!users) {
    throw 'The user list is not available';
  }

  const list = users
    .map((user) => {
      return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;
    })
    .join('');

  return `<ol>${list}</ol>`;
}

const container = document.querySelector('.container');
try {
  container.innerHTML = render(users);
} catch (e) {
  container.innerHTML = e;
}
```

How it works.

First, import `users` from the `user.mjs` module:

```
import { users } from './user.mjs';
```

Second, create a `render()` function that renders the user list to an ordered list in HTML format:

```
function render(users) {
  if (!users) {
    throw 'The user list is not available.';
  }

  const list = users
    .map((user) => {
      return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;
    })
    .join('');

  return `<ol>${list}</ol>`;
}
```

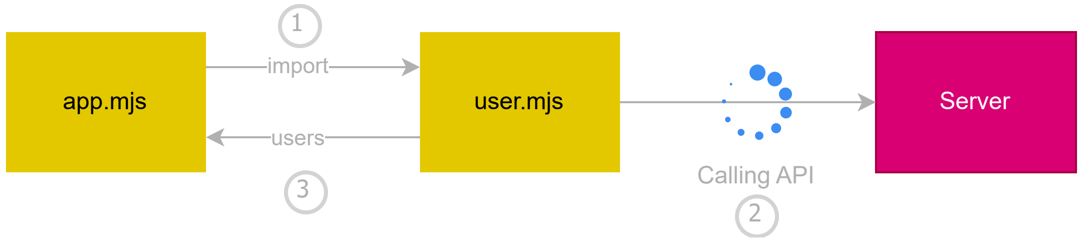
Third, add the user list to the HTML element with the class `.container`:

```
const container = document.querySelector('.container');
try {
  container.innerHTML = render(users);
} catch (e) {
  container.innerHTML = e;
}
```

If you open the `index.html`, you'll see the following message:

```
The user list is not available.
```

The following shows the main flow:



In this flow:

- First, the `app.mjs` imports the `user.mjs` module.
- Second, the `user.mjs` module executes and makes an API call.
- Third, while the second step is still ongoing, the `app.mjs` starts using the `users` data imported from the `user.mjs` module.

Since step 2 has not been completed, the `users` variable was `undefined`. Therefore, you saw the error message on the page.

Workaround

To fix the issue, you can export a `Promise` from the `user.mjs` module and wait for the API call to complete before using its result.

The following shows the new version of the `user.mjs` module:

```
let users;

export default (async () => {
  const url = 'https://jsonplaceholder.typicode.com/users';
  const response = await fetch(url);
  users = await response.json();
})();

export { users };
```

In this new version, the `user.mjs` model exports the `users` and a `Promise` as a default export.

In the `app.mjs` imports the `promise` and `users` from the `user.mjs` file and calls then `then()` method of the `promise` as follows:

```
import promise, { users } from './user.mjs';

function render(users) {
  if (!users) {
    throw 'The user list is not available.';
  }
  let list = users
    .map((user) => {
      return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;
    })
    .join(' ');

  return `<ol>${list}</ol>`;
}

promise.then(() => {
  let container = document.querySelector('.container');
  try {
    container.innerHTML = render(users);
  } catch (error) {
    container.innerHTML = error;
  }
});
```

How it works.

First, import `promise` and `users` from the `user.mjs` module:

```
import promise, { users } from './user.mjs';
```

Second, call the `then()` method of the promise and wait for the API call to complete to use its results:

```
promise.then(() => {
  let container = document.querySelector('.container');
  try {
    container.innerHTML = render(users);
  } catch (error) {
    container.innerHTML = error;
  }
});
```

Now, if you open the `index.html`, you'll see a list of users. However, you need to know the right promise to wait for when using the module.

ES2022 introduced the top-level await module to resolve this issue.

Using the top-level await

First, change the `user.mjs` to the following:

```
const url = 'https://jsonplaceholder.typicode.com/users';
const response = await fetch(url);
let users = await response.json();

export { users };
```

In this module, you can use the `await` keyword without placing a statement inside an `async` function.

Second, import the users from the `user.mjs` module and use it:

```
import { users } from './user.mjs';

function render(users) {
  if (!users) {
    throw 'The user list is not available.';
  }
  let list = users
    .map((user) => {
      return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;
    })
    .join(' ');

  return `<ol>${list}</ol>`;
}

let container = document.querySelector('.container');

try {
  container.innerHTML = render(users);
} catch (error) {
  container.innerHTML = error;
}
```

In this case, the `app.mjs` module will wait for the `user.mjs` module to complete before executing its body.

JavaScript top-level await use cases

When do you use the top-level await? Here are some use cases.

Dynamic dependency pathing

```
const words = await import(`/i18n/${navigator.language}`);
```

In this example, the top-level await allows modules to use runtime values to decide the dependencies, which is useful for the following scenarios:

- Internationalization (i18n)
- Development/production environment splits.

Dependency fallback

In this case, you can use the top-level await to load a module from a server (cdn1). And if it fails, you can load it from a backup server (cdn2):

```
let module;
try {
  module = await import('https://cdn1.com/module');
} catch {
  module = await import('https://cdn2.com/module');
}
```

Summary

- A top-level await module acts like an `async` function.
- When a module imports a top-level await module, it waits for the top-level await module to complete before evaluating its body.