JavaScript AbortController

Summary: in this tutorial, you'll explore how to use the JavaScript `AbortController` to cancel an ongoing network request.

Introduction to the JavaScript AbortController

`AbortController` is a web API that allows you to abort network requests. It works by linking an `AbortSignal` with network requests. This signal can be used to communicate with network requests to abort them when necessary.

Here are the steps for using an `AbortController` to abort a fetch request:

First, create a `AbortController` object using the `new` keyword:

```
const controller = new AbortController();
```

Second, get the `signal` property of the `AbortController` object. The `signal` property is an instance of the `AbortSignal` object:

```
const signal = controller.signal;
```

This `signal` can be passed to the fetch request, allowing us to control its behavior.

Third, pass a `AbortSignal` object to the fetch method:

```
fetch(url, { signal });
```

Alternatively, you can pass the `signal` to the `Request` object and use it in the `fetch` method:

```
const request = new Request(url, { signal });
fetch(request);
```

Fourth, abort the fetch request if necessary by calling the `abort()` method of the `AbortController` object.

```
 controller.abort();
```

For example, you make a fetch timeout after 2 seconds:

```
setTimeout(() => controller.abort(), 2000);
```

> It's fine if you call abort() after the fetch has already been completed, fetch will ignore it.

Fifth, when you call the abort() method, it notifies the abort signal. You can listen to the abort event using the addEventListener on the signal object:

```
signal.addEventListener('abort', () => {
    console.log(signal.aborted); // true
});
```

Finally, you can react to an aborted fetch:

```
fetch(url, { signal }).then(response => {
    return response.json();
}).then(data => {
    console.log(data);
}).catch(err => {
    if (err.name === 'AbortError') {
       console.log('Fetch aborted');
    }
});
```

Note that the `AbortError` is not a real error therefore we use an `if` statement to handle the `AbortError` specifically.

JavaScript AbortController example

In practice, you use the `AbortController` when dealing with user interactions. For example, on a search page, you can abort a previous request when the user searches for a new search term.

We'll build an app that searches for a term on Wikipedia and cancels the previous search request if the user starts a new one before the last one finishes.

Step 1. Create a new directory for storing the project files.:

```
mkdir wikipedia-search-abortable
```

Step 2. Create the following directories and files within the project directory:

```
wikipedia-search-abortable
├── css
|   └── style.css
├── img
|   ├── spinner.svg
|   └── wikipedia-logo.png
├── index.html
└── js
    └── app.js
```

Step 3. Create HTML structure in the `index.html` file:

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

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Wikipedia Search</title>
        <script src="./js/app.js" defer></script>
```

```
        <link rel="stylesheet" href="./css/style.css">
    </head>

    <body>
        <header>
            <img src="./img/wikipedia-logo.png" alt="wikipedia">
            <h1>Wikipedia Search</h1>
            <form id="search">
                <input type="search" name="term" id="term" placeholder="Enter a search term.
            </form>
        </header>

        <main>
            <div id="searchResult"></div>
            <div id="loading"></div>
            <div id="error"></div>
        </main>

    </body>

</html>
```

Step 4. Modify the `app.js` file with the following code:

```
const form = document.querySelector('#search');
const termInput = document.querySelector('#term');
const error = document.querySelector('#error');
const loading = document.querySelector('#loading');
const resultsContainer = document.querySelector('#searchResult');

let controller = new AbortController();

const search = async (term) => {
  // Abort the previous request
  console.log('Abort the previous request');
  controller.abort();

  // Create a new controller for the new request
  controller = new AbortController();
```

```
  const signal = controller.signal;
  const url = `https://en.wikipedia.org/w/api.php?action=query&list=search&prop=info|extracts

  const response = await fetch(url, { signal });
  const data = await response.json();
  return data.query.search;
};

const resetSearchResult = () => {
  error.innerHTML = '';
  loading.innerHTML = '';
  resultsContainer.innerHTML = '';
};

const handleSubmit = async (e) => {
  e.preventDefault();

  // reset search result
  resetSearchResult();

  // check if term is empty
  const term = termInput.value;
  if (term === '') return;

  try {
    // show the loading
    loading.innerHTML = `<img src='./img/spinner.svg' alt='loading...'>`;

    // make the search
    const results = await search(term);

    // show the search result
    resultsContainer.innerHTML = renderSearchResult(results);
  } catch (err) {
    // show the error
    error.innerHTML = `Something went wrong.`;
    console.error(err);
  } finally {
    // hide the loading
    loading.innerHTML = '';
```

```
  }
};

const renderSearchResult = (results) => {
  return results
    .map(({ title, snippet, pageid }) => {
      return `<article>
                <a href="https://en.wikipedia.org/?curid=${pageid}">
                    <h2>${title}</h2>
                </a>
                <div class="summary">${snippet}...</div>
            </article>`;
    })
    .join('');
};

form.addEventListener('submit', handleSubmit);
```

How it works.

First, select the DOM elements using the `querySelector` method:

```
const form = document.querySelector('#search');
const termInput = document.querySelector('#term');
const error = document.querySelector('#error');
const loading = document.querySelector('#loading');
const resultsContainer = document.querySelector('#searchResult');
```

Second, create an `AbortController` for aborting a fetch request:

```
let controller = new AbortController();
```

Third, define a search function that calls the Wikipedia API based on an input search term:

```
const search = async (term) => {
  // Abort the previous request
  console.log('Abort the previous request');
```

```
  controller.abort();

  // Create a new controller for the new request
  controller = new AbortController();
  const signal = controller.signal;
  const url = `https://en.wikipedia.org/w/api.php?action=query&list=search&prop=info|extracts
  const response = await fetch(url, { signal });
  const data = await response.json();
  return data.query.search;
};
```

In the `search()` function, we call the `abort()` method of the `AbortController` object to cancel the previous search request if the user starts a new one:

```
controller.abort();
```

Before making a new request, we create a new `AbortController` object and pass its `AbortSignal` to the `fetch()` method:

```
controller = new AbortController();
const signal = controller.signal;
const url = `https://en.wikipedia.org/w/api.php?action=query&list=search&prop=info|extracts&
const response = await fetch(url, { signal });
```

Step 5. Define a function to reset the error, loading, and search result:

```
const resetSearchResult = () => {
  error.innerHTML = '';
  loading.innerHTML = '';
  resultsContainer.innerHTML = '';
};
```

Step 6. Define a `handleSubmit()` function that executes the search when the form is submitted:

```
const handleSubmit = async (e) => {
  e.preventDefault();

  // reset search result
  resetSearchResult();

  // check if term is empty
  const term = termInput.value;
  if (term === '') return;

  try {
    // show the loading
    loading.innerHTML = `<img src='./img/spinner.svg' alt='loading...'>`;

    // make the search
    const results = await search(term);

    // show the search result
    resultsContainer.innerHTML = renderSearchResult(results);
  } catch (err) {
    // show the error
    error.innerHTML = `Something went wrong.`;
    console.error(err);
  } finally {
    // hide the loading
    loading.innerHTML = '';
  }
};
```

How it works.

First, prevent the form to submit to the server:

```
e.preventDefault();
```

Second, call the `resetSearchResult` function to reset the search result:

```
resetSearchResult();
```

Third, return immediately if the search term is blank:

```
const term = termInput.value;
if (term === '') return;
```

Fourth, show the loading progress indicator before making a search request:

```
loading.innerHTML = `<img src='./img/spinner.svg' alt='loading...'>`;
```

Fifth, make a search request:

```
const results = await search(term);
```

Sixth, display the search result:

```
resultsContainer.innerHTML = renderSearchResult(results);
```

Seventh, display a user-friendly user message and log the error detail in the console:

```
 } catch (err) {
    // show the error
    error.innerHTML = `Something went wrong.`;
    console.error(err);
 }
```

Eighth, hide the loading indicator regardless of whether an error occurs or not:

```
} finally {
  // hide the loading
  loading.innerHTML = '';
}
```

Ninth, define a function `renderSearchResult` that renders the search result:

```
const renderSearchResult = (results) => {
  return results
    .map(({ title, snippet, pageid }) => {
      return `<article>
                <a href="https://en.wikipedia.org/?curid=${pageid}">
                    <h2>${title}</h2>
                </a>
                <div class="summary">${snippet}...</div>
            </article>`;
    })
    .join('');
};
```

Finally, register the `handleSubmit` function as a submit event handler of the form:

```
form.addEventListener('submit', handleSubmit);
```

Download the project source code

Download the project source code

Summary

- Use `AbortController` to abort a web request to prevent unnecessary network requests and handle user interactions efficiently.