Async Iterators in JavaScript

Code For Real
@codeforreal

JavaScript async iterators are a powerful feature introduced in ECMAScript 2018 (ES9) that allow you to iterate over asynchronous data sources like **streams**, **files**, or **API** responses in a more convenient and readable way. Async iterators are particularly useful when working with asynchronous operations, as they simplify the process of consuming asynchronous data.

To understand async iterators, let's break down the concept with detailed code examples:

Code For Real
@codeforreal

Save

Creating an Async Iterable:

To create an async iterable, you need an object that implements the async iterable protocol. This means it should have an [Symbol.asyncIterator] method that returns an async iterator. Each time you call next() on this iterator, it returns a promise that resolves to an object with two properties: value and done.

Here's a simple example using an async generator function:

```
async function* asyncDataGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const asyncIterable = asyncDataGenerator();
const asyncIterator =
asyncIterable[Symbol.asyncIterator]();
```

Code For Real
@codeforreal

Save

Consuming an *Async* Iterable:

To consume data from an async iterable, you can use for-await-of or manually iterate using the next() method with await. Let's see both approaches:

Using for-await-of (Preferred):

```
async function consumeAsyncIterable() {
  for await (const value of asyncIterable) {
    console.log(value);
  }
}

consumeAsyncIterable();
```

Code For Real
@codeforreal

Save

Using next() with await:

```
async function consumeAsyncIterable() {
  const asyncIterator = asyncIterable[Symbol.asyncIterator]();
  let result = await asyncIterator.next();

  while (!result.done) {
    console.log(result.value);
    result = await asyncIterator.next();
  }
}

consumeAsyncIterable();
```

Code For Real
@codeforreal

Save

Real-world Example with an *Asynchronous* Data Source:

Let's simulate a more practical scenario where you fetch data from an API using async iterators. In this example, we'll use the fetch API to retrieve data asynchronously.

```
async function fetchUserData() {
  const response = await
fetch('https://jsonplaceholder.typicode.com/users');
  const data = await response.json();

  async function* asyncDataGenerator() {
    for (const user of data) {
      yield user;
    }
  }

  for await (const user of asyncDataGenerator()) {
    console.log(user.name);
  }
}

fetchUserData();
```

Code For Real
@codeforreal

Save

In this example, we fetch user data from an API and create an async iterable from it. Then, we use for-await-of to iterate through the user data and log each user's name.

Async iterators simplify the handling of asynchronous data streams and can be particularly useful when dealing with large datasets or data from real-world asynchronous sources like **APIs** and **streams**. They make the code more readable and maintainable by abstracting away the complexities of managing asynchronous operations.

Code For Real
@codeforreal

Save