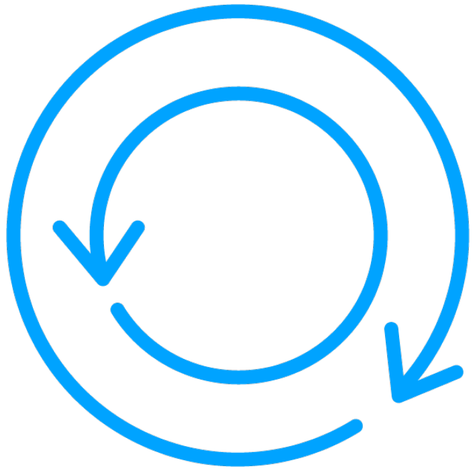
Asynchronous Best Practices

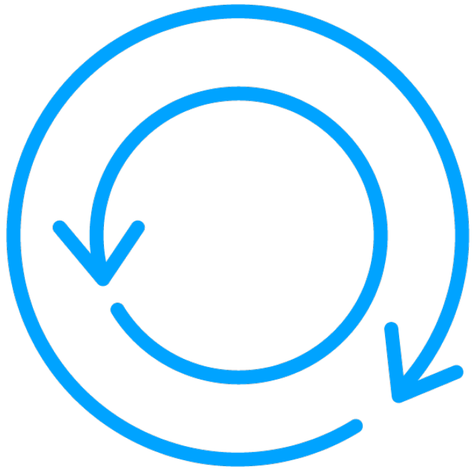
Dan Wellman
Senior Front-end Developer

Why Should You Follow Asynchronous Best Practices?



- Consistent, readable, maintainable, and extensible
- Robust in terms of error handling and performance
- Type-safe and able to leverage TypeScript's powerful development features like IntelliSense and code completion

Use Async/Await



- Syntactic sugar on top of native promises
- Write asynchronous code that looks synchronous

```
function getData() {
  let data1, data2;

  fetch('http://some-web-api.com').then(response1 => {
    data1 = response1.json();

    fetch('http://other-web-api.com').then(response2 => {
      data2 = response2.json();
      // do something with data1 and data2...
    });
  });
}
```

In typical asynchronous code, which makes multiple requests, response handlers using `then` can become deeply nested, making the code harder to understand

```
async function getData() {
  const response1 = await fetch('http://some-web-api.com');
  const data1 = await response1.json();

  const response2 = await fetch('http://other-web-api.com');
  const data2 = await response2.json();

  // do something with data1 and data2...
}
```

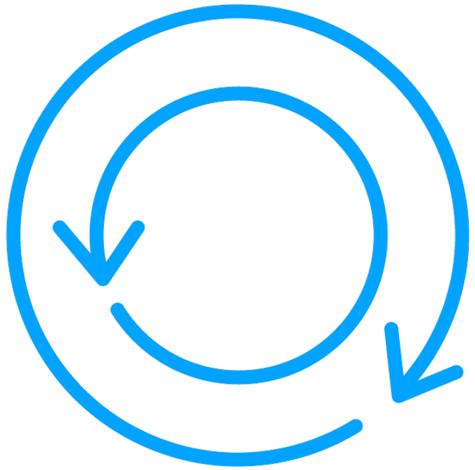
This code is much more linear, easier to read, and a little more compact

Only `async` functions can use `await`

`await` makes the function pause while the fetch completes, so the code looks synchronous

Always Handle Errors

Handling Asynchronous Errors



- Asynchronous code is often used to make HTTP requests to a server
- There are many reasons an HTTP request can fail
- Promises have the `catch` method to catch any errors

```
function getData() {
  let data1, data2;

  fetch('http://some-web-api.com').then(response1 => {
    data1 = response1.json();

    fetch('http://other-web-api.com').then(response2 => {
      data2 = response2.json();
      // do something with data1 and data2...
    });
  });
}
```

If either of the `fetch` calls fail, an unhandled error will be seen in the browser, which may cause the application to fail

```
fetch('http://some-web-api.com').then(response1 => {
  data1 = response1.json();

  fetch('http://other-web-api.com').then(response2 => {
    data2 = response2.json();
    // do something with data1 and data2...
  });
}).catch(error => {
  // handle the error...
});
```

You should add a `catch` handler after the `then` method of the first `fetch`

The arrow function passed to the `catch` method receives the error as a parameter and prevents the error from being unhandled

However, an error in the nested second `fetch` call will not be caught by the `catch` on the outer `fetch`

```
fetch('http://some-web-api.com').then(response1 => {
  data1 = response1.json();

  fetch('http://other-web-api.com').then(response2 => {
    data2 = response2.json();
    // do something with data1 and data2...
  }).catch(error => {
    // handle the error
  });
}).catch(error => {
  // handle the error...
});
```

You should also add a `catch` handler to the nested `fetch` method

You can easily identify the source of any error

Always add a `catch` method whenever you use a `then` method.

Catching asynchronous errors makes your code more robust and resilient.

```
async function getData() {
  try {
    const response1 = await fetch('http://some-web-api.com');
    const data1 = await response1.json();

    const response2 = await fetch('http://other-web-api.com');
    const data2 = await response2.json();

    // do something with data1 and data2...
  } catch (error) {
    // handle the error
  }
}
```

You should use a single `try`/`catch` block when using `async` and `await`

Any error from any awaited statement will be caught by the `catch` clause

However, an error on the first `fetch` here would prevent the second `fetch` being made

```
const response1 = await fetch('http://some-web-api.com').catch(error => {
  // handle the error ...
});
```

You can add a `catch` to `fetch` as it returns a promise

Now, any statements after `fetch` will continue to execute

If all your awaited calls have `catch` methods, the outer `try`/`catch` becomes redundant

Use `Promise.all` and `Promise.race`

Promise.all/Promise.race

- Static methods on `Promise`
- Easy to use
- Used for concurrent, asynchronous requests

```
function getData() {
  let data1, data2;

  fetch('http://some-web-api.com').then(response1 => {
    data1 = response1.json();

    fetch('http://other-web-api.com').then(response2 => {
      data2 = response2.json();
      // do something with data1 and data2...
    });
  });
}
```

Use `Promise.all` when making multiple asynchronous requests and needing to wait for all to complete before proceeding

```
function getData() {
  const req1 = fetch('http://some-web-api.com');
  const req2 = fetch('http://other-web-api.com');

  Promise.all([req1, req2]).then([data1, data2] => {
    // do something with data1 and data2...
  }).catch(error => {
    // handle the error
  });
}
```

`Promise.all` **can avoid nested** `then` **handlers**

Pass multiple promises to `Promise.all`

More efficient to make requests in parallel

```
function getData() {
  const req1 = fetch('http://some-web-api.com');
  const req2 = fetch('http://other-web-api.com');

  Promise.all([req1, req2]).then([data1, data2] => {
    // do something with data1 and data2...
  }).catch(error => {
    // handle the error
  });
}
```

`Promise.all` **returns a promise which resolves when all promises passed to it are resolved**

The `then` **attached to** `Promise.all` **receives an array containing all values from promises passed to it**

```
function getData() {
  const req1 = fetch('http://some-web-api.com');
  const req2 = fetch('http://other-web-api.com');

  Promise.all([req1, req2]).then([data1, data2] => {
    // do something with data1 and data2...
  }).catch(error => {
    // handle the error
  });
}
```

If any promise passed to `Promise.all` rejects, the `catch` handler attached to it is invoked

Handle errors in a centralized way

Use `Promise.all` when you make multiple requests and can handle the results of them all together.

```
function getData() {
  const req1 = fetch('http://some-web-api.com');
  const req2 = fetch('http://other-web-api.com');

  Promise.race([req1, req2]).then(data => {
    // do something with data...
  }).catch(error => {
    // handle the error
  });
}
```

`Promise.race` **takes multiple promises and resolves when the first promise resolves**

```
function getData() {
  const req1 = fetch('http://some-web-api.com'); // returns a number
  const req2 = fetch('http://other-web-api.com'); // returns a string

  Promise.race([req1, req2]).then((data: number | string) => {
    // do something with data...
  }).catch(error => {
    // handle the error
  });
}
```

Always add a union type for the types returned with each of the promises

Promise.all/Promise.race

- Designed to use when working with multiple promises concurrently
- Use, when possible, for readability, maintainability, and performance

Use Loading States and Timeouts

Loading States

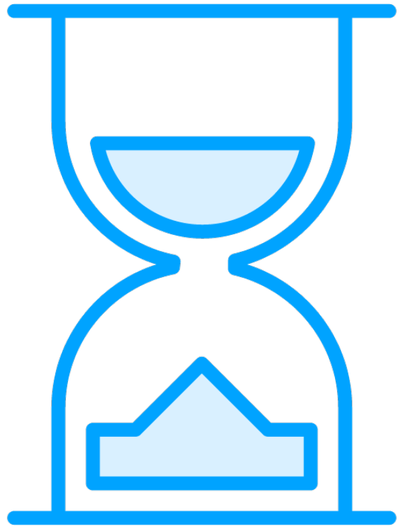
- Important visual feedback that something is happening
- Determinate—you know how long it will take and can report progress to the user
- Indeterminate—you don't know how long it will take, just spin until done

Loading States

- Users want responsive apps
- Loading states reassure the user
- If the app does nothing at all, the user may think it has crashed and will leave

Loading states are a key part of user-centric development.

Timeouts



- A way to abort a request if it is taking too long
- A loading state that continues forever will also frustrate users
- Replace loading state after a short time with a message that the request failed

`Promise.race` can be used to implement timeouts.

```
function timeout(ms: number): Promise<string> {
  return new Promise((_, reject) => {
    setTimeout(() => reject(new Error(`Timeout after ${ms}`)), ms);
  });
}
```

Return a promise which is rejected after a short delay

If the request takes too long, the timeout promise will reject and trigger the catch

Always Clean up Subscriptions

RXJS

- Popular framework for working with asynchronous code
- Commonly used with Angular/React/etc.
- Observe values and react when they change
- Respond to a stream of events

RXJS

- Add subscriptions using Observable's `subscribe` method
- Register observers to be notified when the value changes
- Powerful, but can create memory leaks if not handled properly

Whenever you call the `subscribe` method on an Observable, you should always call the `unsubscribe` method.

Handling Subscriptions

- You can store the subscription to unsubscribe it later
- Angular gives you `takeUntilDestroyed` to handle subscription cleanup

RXJS

- Observables have a method called `pipe`, which you can use to transform or operate on observable values

The performance of your applications can suffer if you don't unsubscribe.