Keep Your Promises in TypeScript using async/await

TypeScript is designed for development of large applications and transpiles to JavaScript. As TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs. TypeScript may be used to develop JavaScript applications for both client-side and server-side execution.

The `await` keyword is syntactical shorthand for indicating that a piece of code should asynchronously wait on some other piece of code. It is a hint that you can use to mark methods as task-based asynchronous methods.

Let's see how `async` / `await` works and how we can use it in TypeScript.

Getting Started

If you're using VS Code, it includes TypeScript by default. If you didn't install TypeScript with VS Code, you can download it here.

You can also install TypeScript using this command:

```
npm install -g typescript
```

Let’s create a new folder named `typescript`. Inside, create a JSON filed named `tsconfig.json`. This file will be used by the TypeScript compiler to compile our code.

Also, create a new folder named `src` inside the `typescript` folder.

Simplify Async Callback Functions using Async/Await

Lets see how we can write a `Promise` and use it in `async await`. This method helps simplify the code inside functions like `setTimeout`.

Create a new file inside `src` folder called `index.ts`. We’ll first write a function called `start` that takes a `callback` and calls it using the `setTimeout`function.

```
const start = callback => {
  setTimeout(() => {
    callback('Hello');
    setTimeout(() => {
      callback('And Welcome');
      setTimeout(() => {
        callback('To Async Await Using TypeScript');
      }, 1000);
    }, 1000);
  }, 1000);
};

start(text => console.log(text));
```

We can use this function to run a `callback` that logs some text to the console.

To compile this code, run:

```
$ tsc src/index.ts
```

This will create a new file named `index.js`. You can run this file in the console using `node`.

```
$ node src/index.js
```

You will now notice that the code runs exactly as we want it to. But we can make things much simpler using `async await`.

All we need to do to use `async await` is to create a `Promise` based delay function.

```
const wait = (ms) => new Promise(res => setTimeout(res, ms));
```

This function takes a number of milliseconds and returns a `Promise` that gets resolved using `setTimeout` after the given number of milliseconds.

Now create an `async` function called `startAsync`. This function is actually quite similar to the `start` function that we had written before. Inside this function, we will use the `await` to pause the execution of the code until the `Promise` is resolved and call the `callback` passing in the time.

```
const startAsync = async callback => {
  await wait(1000);
  callback('Hello');
  await wait(1000);
  callback('And Welcome');
  await wait(1000);
  callback('To Async Await Using TypeScript');
};

startAsync(text => console.log(text));
```

Running this code using `node`, we can see that it still behaves the same way, but our code is much simpler.

Promises in TypeScript

We begin by creating a simple promise like this:

```
const one = new Promise<string>((resolve, reject) => {});
```

In this `Promise`, I have used the promise constructor to take in `string` as the generic type for the Promise's `resolve` value. The promise constructor takes an `executor`callback which the compiler will call by the runtime with these two arguments:

- `resolve` — This is a function that is used to resolve the promise.
- `reject` — This is a function that is used to reject the promise.

So, our promise can either be resolved, or rejected. The `resolve` part is taken care of by `.then`, and the `reject` part is taken care of by `.catch`.

```
one.then(value => {
  console.log('resolved', value);
});
one.catch(error => {
  console.log('rejected', error);
});
```

If we resolve a `Promise`, `then` callbacks are executed. Else, it means that the `Promise` is rejected and the `catch` callbacks are executed.

Promise resolutions are easy to write:

```
resolve('Hello')
```

Promise rejections on the other hand, should only be done in exceptional cases. It is considered as a bad practice to `reject` a promise with a raw string. Always use the error constructor `new Error` when rejecting a promise.

```
reject(new Error('failed'));
```

Promise Chains

The `then` function actually creates a new `Promise` that is `distinct` from the `Promise`that the `then` function belongs to. To verify this, create a new variable called `two` that calls the `then` function of `one`.

```
const one = new Promise<string>((resolve, reject) => {
  resolve('Hello');
});
const two = one.then(value => {});
console.log(one === two);
```

Running this code will print out a `false`, verify that `one`‘s `then` function creates a new `Promise` that is distinct from `one`. `two` also has its own `then`and `catch` callbacks. Replace the `console.log` statement with this:

```
two.then(value => {
  console.log('Hi', value);
});
two.catch(error => {
  console.log('Oops', value);
});
```

If you return a value inside `two`, that value will become the resolved value of the second `Promise`. Re-write `two` like this:

```
const two = one.then(value => {
  return 'Hey';
});
```

Running this code, will give you a new output that has the string `Hey` in it. If we are now returning anything inside `two`, TypeScript will replace the previous `Hey` with an `undefined`.

If you return a `Promise`, the resolution of this `two` determined by the fate of this new `Promise`.

If the new `Promise` resolves, then `two` will also resolve by taking the new `Promise`'s resolved value as its own. And if the new `Promise` gets rejected, then `two` will also get rejected with the same `Error`.

We can also throw `Error` inside a `then` callback:

```
const two = one.then(value => {
  throw new Error("OH OH!");
});
```

Also, make sure that you are not using any undeclared variables inside a Promise, as it will cause the promise to be rejected.

A very important concept in chained `Promises` is the **Propagation of Rejection.**

Inside `index.ts` file, create a Promise Chain as shown below:

```
new Promise<boolean>((res, rej) => {
  res(true);
})
  .then(res => {
    console.log(res);
    return false;
  })
  .then(res => {
    console.log(res);
    return true;
  })
    .then(res => {
    console.log(res);
  })
  .catch(error => {
    console.log('ERROR:', error.message);
});
```

Run this code in your console, and you will get the output as `true`, `false`, and `true`.

A rejection at any point inside a Promise Chain will result in all `then`functions to be ignored and the execution will directly go to nearest `catch`handler. To show this, add an undeclared variable inside any of the `then`functions and run the code again.

Asynchronous Functions with async await

Using `async await` lets us use `Promises` in a reliable and safe way. This method prevents chances of any programming errors.

Writing asynchronous functions is really easy. Just write a function and add the `async`keyword to it like this:

```
async function gilad() {
  return 'Gilad';
}
// or
const gilad = async () => {
  return 'Gilad';
}
// or
class Gil {
  async gilad() {
    return 'Gilad';
  }
}
```

An `async` function always returns a `Promise`. The `Promise` resolves to value that is returned by the function. In the async function below, we are returned an `undefined`value.

```
async function gilad() {
}
gilad().then(value => {
  console.log(value);
});
```

Lets also write a couple of `Promises` that we can use inside the `async`function.

- Create a variable that is not a `Promise`.

```
const one = 'One';
```

- Create a `Promise` with a `resolve`.

```
const two = new Promise(resolve => resolve('Two'));
```

- Create a `Promise` with a `reject`.

```
const three = new Promise((resolve, reject) => reject(new
Error('Three')));
```

Asynchronous functions can use the `await` operator in their bodies.
The `await` operator can be attached to any variable. If that variable is not a `Promise`, the value returned for the `await` operator is the same as the variable.

But if the variable is a `Promise`, then the execution of the function is paused untill it is clear whether the `Promise` is going to be resolved or rejected.

If the `Promise` resolves, the value of the `await` operator is the resolved value of `Promise`, and if the variable is a promise that gets rejected, the `await`operator throws an error in the body of the `async` function which we can catch with `try/catch` constructs.

```
async function gilad() {
  const four = await one;
  console.log({ one: four });
  const five = await two;
  console.log({ two: five });
  try {
    const six = await three;
    console.log('This will not get called at all');
  }
  catch(e) {
    console.log({ three: e.message});
  }
}
gilad();
```

Running this code, and you will see that everything works as it should:

- The not a `Promise` variable `one` resolves to itself.
- The `Promise` that will resolve returns its final resolved value.
- The `Promise` that gets rejected, interacts with `try/catch` as expected.

If we add a `setTimeout` to our `async` function, the execution at the `await`operator pauses till we know if it should `resolve` or `reject`.

```
async function gilad() {
  await new Promise(resolve => setTimeout(resolve, 5000));
  console.log('Done!);
}
gilad();
```

In the above code snippet, I am giving the `async` function 5 seconds before it consoles out some string.

`async/await` allows you to write asynchronous code based on `Promises`, in a manner that allows you to reuse your synchronous code writing skills.

Parallel and Serial Execution of Promises

Asynchronous code allows our app to do multiple things in parallel. This is really helpful to us when we want our to make multiple network requests.

Let’s create a new file inside `src` called `hero.ts`. Write the following code inside it:

Delete everything in `index.ts` and import the `async` function `getHero`.

```
import {getHero} from './hero';
```

The `getHero` function simply takes the `hero` and returns a `Promise` to the details which are resolved asynchronously.

Create another asynchronous function in `index.ts`. This function will have an array named `handles` containing the name of a couple of `heroes`.

```
async function gilad() {
  const handles = [
    'superman',
    'batman',
    'flash'
  ];
}

gilad();
```

To get the details of each of these `handles` is a very simple process. We just simply loop through the handles with a `for-of` loop. Inside the `async`function `gilad()`, write:

```
for (const handle of handles) {
  const item = await getHero(handle);
  console.log(`
Name: ${item.name}
Alias: ${item.alias}
  `);
}
```

Run this code in the terminal. Doing this serial sequence of events is something that is way easier with `async await`.

But there are times, when you may want your app to run more than one operations at a time, and wait for them all to resolve. To do this, we can use `Promise.all`, which is a native function that takes an array of `Promises` and returns a new `Promise` that resolves with an array of resolved values for each of the promise. Delete the `for-of` loop inside the `async` function and write:

```
const all = handles.map(getHero);
const combine = Promise.all(all);
const details = await combine;
for (const item of details) {
  console.log(`
Name: ${item.name}
Alias: ${item.alias}
  `);
}
```

Start off by running all the calls to `getHero` in parallel. At this point, we have an array of `Promises` that will `resolve` independently. With the simple `Promise`, we can `await` it by giving a single array of resolved values. We then simply loop over the elements of the array and log it out.

Run this code in the terminal. Unlike the serial execution method, we get all the values at the same time.

Another method worth mentioning is `Promises.race`. `Promises.race` is a function that takes an array of `Promises` and returns a new `Promise`. This `Promise`‘s value is equal to that of the first `Promise` that resolves or rejects. Inside the async function `gilad`, delete the `for-of` loop and write:

```
const resolvedPromise = Promise.race(all);
const item = await resolvedPromise;
console.log(`
  Name: ${item.name}
  Alias: ${item.alias}
`);
```

Running this code will give you the first hero’s details as output.

Asynchronous Iteration using for-await-of

The `for-await-of` syntax shares some similarities with `for-of` iteration. The main difference between these two syntaxes is that `for-await-of`automatically awaits any `Promises` generated by this iterator. `for-await-of`essentially allows you to use `async await` in a generator function.

Create a simple generator function in the `index.ts` file. This function will return numbers one by one till 10.

This generator function can be used inside a standard synchronous JavaScript `for-of` loop.

What if I want to get the next number from a back-end service?

Create a new file named `external.ts` that has the following code in it:

```
export function external(num: number) {
  return new Promise<number>(res => {
    setTimeout(() => res(num + 1), 1000);
  });
}
```

Import this file inside `index.ts` file:

```
import {external} from './external';
```

Now inside the generator function, replace the `index` statement with this:

```
import {external} from './external';
  function* numbers() {
  let index = 1;
  while (true) {
    yield index;
    index = external(index);
    if (index > 10) {
      break;
    }
  }
}
```

You will now get a type mismatch on `index`. This is because we have a `Promise` to a `number` and we would really like to have the resolved `number` to make our decision for the loop termination.

This can be done with `async await` like this:

```
import {external} from './external';

async function* numbers() {
  let index = 1;
  while(true) {
    yield index;
    index = await external(index);
    if (index > 10) {
      break;
    }
  }
}
```

`for-await-of` requires a runtime polyfill called the `asyncIterator` to work correctly.

```
(Symbol as any).asyncIterator =
  (Symbol as any).asyncIterator
  || Symbol.for("Symbol.asyncIterator");
```

This `numbers` function is an async generator and returns an async iterator. We can use `for-await-of` loops with async iterators. Re-write the `gilad` function as an async function like this:

```
async function gilad() {
  for await (const num of numbers()) {
    console.log(num);
  }
}
gilad();
```

Run the code in your terminal. You can see that the `for-await-of` loop works as expected.

Conclusion

The `async` keyword tells the JavaScript compiler to treat the function differently. The compiler pauses whenever it reaches the `await` keyword within the same function. It assumes that the expression after `await` is returning a `Promise` and waits until the `Promise` is resolved or rejected before moving further.