ASYNC CONSTRUCTS IN JAVASCRIPT

Promises and Async-await

Prashanth Puranik
Web Developer and Trainer

Copyright Notice

Drawback of Callback

Drawback of Callback

- Serial Async operations result in unreadable code
  - Called the Christmas Tree or Callback Hell
- Callback function is essentially handed over to 3rd party APIs
  - Security risk
- Synchronizing multiple asynchronous operations is not easy
  - Carrying out concurrent async operations, and working with results once all of them complete
  - Carrying out serial async operations, usually where one utilizes the result of the previous async operations

Promises

What is a Promise?

- Promise is an in-built class in ES2015
- Acts as mediator between code that requires result of an async operation and code that performs an async operation
- Has constructs and an API that simplifies synchronization of multiple async tasks
- Flow of code is clearer
- More secure than using callbacks as your code is not passed to a 3rd party API
  - Rather communication is via promises

Promise – An analogy

- You ask me a question and I do not know the answer
- I give you a **promise** to give an answer. The promise is *returned* **immediately**, the answer is not.
- I *resolve* my promise at a later point in time with a *value* (the answer)
- If I fail to find an answer I *reject* the promise with a *reason for failure*

In other words...

A Promise object represents the future result of an asynchronous operation.

A promise can be in one of three states - pending, resolved or rejected. A promise is settled if it is either resolved or rejected.

Promisifying a function

- Steps to convert an async function that uses callbacks to one that uses promises
  - Remove the callback argument
  - Create and return a Promise object
    - Pass a function to the Promise constructor and do the async operation within
    - The function passed is invoked by the constructor
    - It is passed 2 functions as arguments – say, *resolve* and *reject*
      - Call resolve, passing result of async operation once the operation completes
      - Call reject, with reason for failure (an Error object), in case of failure

Example

```
function promisifiedAPI( a /*, b, ... */ ) { // no callbacks in arguments
        return new Promise(( resolve, reject ) => {
                if( /* condition for errors */ ) {
                        return reject( new Error( 'reason for error' ) );
                }

                doSomethingAsync( result => resolve( result ) );
        });
}
```

Using a method that returns a Promise

- The Promise API works like the try…catch construct, but for asynchronous code
- The resolve function passed to then are called when the promise is resolved
- The return value of a resolve handler is passed to the next resolve handler in sequence
- In case the promise rejects, or one of the resolve handlers throws an error, the nearest reject handler (set using catch()) is called
- then() and catch() are non-blocking functions

```
sumAsync( 1, 2 )
        .then( resolvedValue => { … ; return X; } )
        .then( X => { … } )
        …
        .catch( error => { … } );
        .then( previousValue => { … } )
```

Note: The then() method can be passed a reject handler as 2nd argument, but using it is not a good practice

async...await

Async..await

- A construct that simplifies use of promises for performing serial async operations
- A function marked `async` always returns a promise. If it specifies a non-promise a return value, the value is wrapped in a promise that resolves (with the value)
- If it throws an error, the Promise rejects with the error as reason

```
async function foo() {
      return 1;
}
foo().then( console.log );
```

```
async function foo() {
      throw new Error('ff');
}
foo().catch( err => console.log( err.message ) );
```

await

- The `await` keyword can be used only within a function marked `async`
- It expects a promise after it. The async function "**pauses**" when await is called. It "**resumes**" only after the promise resolves/rejects
- The JavaScript runtime continues execution of other statements while the async function is paused
- If promise resolves, the resolved value is the value of the await expression
- If promise rejects, an error is thrown
- Tip: Surround await calls with a try..catch block.

await

```
(async function() {
	try {
		let result1 = await sumAsync( 1, 2 );
		let result2 = await sumAsync( result1, 3 );
		let result3 = await sumAsync( result2, 4 );
		console.log( result3 );
	} catch( error ) {
		console.error( error );
	}
}());
```

await (error thrown)

```
(async function() {
        try {
                let result1 = await sumAsync( 1, 2 );
                let result2 = await sumAsync( result1, 'hello' );
                let result3 = await sumAsync( result2, 4 );
                console.log( result3 );
        } catch( error ) {
                console.error( error );
        }
}());
```

THANK YOU!

Questions are welcome

More you can explore...

When a then() handler returns a Promise…

- When a then() handler returns a Promise object, the next then() handler receives the resolved value of the Promise object (or next catch receives the error on rejection)

```
sumAsync( 1, 2 )
    .then( result1 => { … ; return sumAsync( result1, 3 ); } )
    .then( result2 => { … } )
    …
    .catch( error => { … } );
```

- This makes it easy to perform serial async operations

Promise.all

- Promise.all() allows multiple async operations to be executed concurrently

```
Promise.all( [ sumAsync( 1, 2 ), sumAsync( 3, 4 ), sumAsync( 5, 6 ) ] )
        .then(function( values ) {
                console.log( values ); // logs [ 3, 7, 11 ]
        });
```

Copyright Notice