Asynchronous Java Script

* **clg** for console.log(object);
* **clo** for console.log('object :', object);
* **ccl** for console.clear(object);
* **cer** for console.error(object);
* **ctr** for console.trace(object);
* **clt** for console.table(object);
* **cin** for console.info(object);
* **cco** for console.count(label);

# Synchronous Vs. Asynchronous Code

const courses = [

{ id: 1, name: 'course1' },

{ id: 2, name: 'course2' },

{ id: 3, name: 'course3' }

];

However in real time application however, instead of using an array in memory we use a database. We will learn how to access database in node.

Let create a folder async-demo

$ mkdir async-demo

$ cd async-demo/

$ npm init –yes

$ npm install nodemon

Create a index.js file

console.log('Before');

//asynchronous or non-blocking function

setTimeout(() => {

console.log('Reading a user from a database');

}, 2000);

console.log('After');

$ $ nodemon index.js

A screen shot of a smart phone

Description automatically generated

# Patterns For Dealing With Asynchronous Code

## Async Pattern

There are three code to deal with asynchronous code

1. callbacks
2. promises
3. Async/await

console.log('Before');

const user = getUser(1);

console.log(user);

console.log('After');

//There are three code to deal with asynchronous code

//callbacks

//promises

//Async/await

function getUser(id) {

//asynchronous or non-blocking function

setTimeout(() => {

console.log('Reading a user from a database');

return { id: id, gitHubuser: 'Uttam' };

}, 2000);

return 1;

}

A picture containing meter, clock

Description automatically generated

## Callbacks

console.log('Before');

//Retrive the user

getUser(1, (user) => {

console.log('User: ', user);

//After retriving the user

//retrive the repositories of the user

gitHubRepository(user.gitHubUsername, (repos) => {

console.log('Repos: ', repos);

for (let val of repos) {

console.log(val);

}

});

});

console.log('After');

function getUser(id, callback) {

//asynchronous or non-blocking function

setTimeout(() => {

console.log('Reading a user from a database');

callback({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

//return 1;

}

//coverting method to async

function gitHubRepository(username, callback) {

setTimeout(() => {

console.log('Call GitHub API...');

callback(['repo1', 'repo2', 'repo3']);

}, 2000);

}

Output

A picture containing drawing, video

Description automatically generated

## Callback Hell

//Asynchronous

console.log('Before');

//Retrive the user

getUser(1, (user) => {

console.log('User: ', user);

//After retriving the user

//retrive the repositories of the user

getRepository(user.gitHubUsername, (repos) => {

//console.log('Repos: ', repos);

for (let repo of repos) {

getCommits(repo, (commits) => {

//CALLBACK HELL

});

}

});

});

console.log('After');

synchronous

//synchronous

console.log('Before');

const user = getUser(1);

const repos = getRepository(user.gitHubUsername);

const commits = getCommits(repos[0]);

console.log('After');

Entire callback methods logic

console.log('Before');

//Retrive the user

getUser(1, (user) => {

console.log('User: ', user);

//After retriving the user

//retrive the repositories of the user

getRepositories(user.gitHubUsername, (repos) => {

console.log('Repos: ', repos);

for (let repo of repos) {

getCommits(repo, (commits) => {

console.log(`${repo}: commits: ${commits}`);

});

}

});

});

console.log('After');

function getUser(id, callback) {

//asynchronous or non-blocking function

setTimeout(() => {

console.log('Reading a user from a database');

callback({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

//return 1;

}

//coverting method to async

function getRepositories(username, callback) {

setTimeout(() => {

console.log('Call GitHub API...');

callback(['repo1', 'repo2', 'repo3']);

}, 2000);

}

//coverting method to async

function getCommits(repo, callback) {

setTimeout(() => {

console.log(`Gets All commits for ${repo}...`);

callback(['commit1', 'commit2', 'commit3']);

}, 2000);

}

## Named Functions to Rescue

console.log('Before');

//Retrive the user

//Named Function

getUser(1, getRepositioriesByUser);

console.log('After');

function getRepositioriesByUser(user) {

console.log('User: ', user);

//After retriving the user

//retrive the repositories of the user

getRepositories(user.gitHubUsername, getRepoCommits);

}

function getRepoCommits(repos) {

console.log('Repos: ', repos);

for (let repo of repos) {

getCommits(repo, displayCommits);

}

}

function displayCommits(commits) {

console.log(`commits: ${commits}`);

}

function getUser(id, callback) {

//asynchronous or non-blocking function

setTimeout(() => {

console.log(`Reading a user with userid: ${id} from a database`);

callback({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

//return 1;

}

//coverting method to async

function getRepositories(username, callback) {

setTimeout(() => {

console.log(`Get repositiories for username : ${username}`);

callback(['repo1', 'repo2', 'repo3']);

}, 2000);

}

//coverting method to async

function getCommits(repo, callback) {

setTimeout(() => {

console.log(`Gets All commits for ${repo}...`);

callback(['commit1', 'commit2', 'commit3']);

}, 2000);

}

Output:

A close up of text on a black background

Description automatically generated

## Promises

Promises are extremely powerful which dealing with asynchronous. A promise is an object that holds the eventual result of an asynchronous operations. So when an asynchronous operations completes it can either results in a value or an error. A promise basically promises you that it would give you the result of an asynchronous operations .

This object can be one of the three states.

A close up of a logo

Description automatically generated

Initially let me create a promise object it will be in the pending state.

At this point we can pick up some asynchronous operations when the result is ready , the promise can either be fulfilled/resolved. Which basically means the operations completed successfully. So here we are going to have value.

A close up of a logo

Description automatically generated

Otherwise if something went wrong during the executions of the asynchronous operations, our promise will be in a rejected state. In this case we’re going to have an error. Now let see this in actions

Create a promise.js

//create promise

const p = new Promise((resolve, reject) => {

//kick of the async work

//...

setTimeout(() => {

//resolve(1); //pending => resolve, fulfilled

reject(new Error('Opps! something went wrong !!')); //pending =>rejected

}, 2000);

});

//consume promise

p.then((result) => console.log('Result :', result)).catch((err) =>

console.log('Error: ', err.message)

);

A promise is an object that holds the eventual result of an asynchronous operations. Initially it is in the pending state , when we create a promise . At this point , it kicks off an asynchronous operations . The operation can completes successfully or fail, if it completes successfully we say it resolve or fulfilled. So the state of the promise changes from pending to resolved, which is also called fulfilled. Now if the asynchronous operations fails, the state of the promise will go from pending to rejected. So here we use the reject function to return an error to the consumer of this promise. This is how we create a promise. We call then to get the result and catch to get the error.

A screenshot of a cell phone

Description automatically generated

Now here’s what you need to take away from this lecture : anywhere you have an asynchronous function that takes a callback , you should modify that function to return a promise.

## Replacing Callbacks with Promises

console.log('Before');

//CALLBACKS -nested structure

getUser(1, (user) => {

getRepositories(user.gitHubUsername, (repos) => {

getCommits(repos[0], (commits) => {

console.log(commits);

});

});

});

console.log('After');

function getUser(id) {

return new Promise((resolve, reject) => {

//kick off async work

setTimeout(() => {

console.log(`Reading a user with id: ${id} from a database...`);

resolve({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

});

}

function getRepositories(username) {

return new Promise((resolve, reject) => {

//kick off async work

setTimeout(() => {

console.log('Calling GitHub API - Repository');

resolve(['repo1', 'repo2', 'repo3']);

}, 2000);

});

}

function getCommits(repo) {

return new Promise((resolve, reject) => {

//kick off async work

setTimeout(() => {

console.log('Calling GitHub API - Commits');

resolve(['commit1', 'commit2', 'commit3']);

}, 2000);

});

}

## Consuming Promises

console.log('Before');

//CALLBACKS -nested structure

// getUser(1, (user) => {

// getRepositories(user.gitHubUsername, (repos) => {

// getCommits(repos[0], (commits) => {

// console.log(commits);

// });

// });

// });

//PROMISES - Flat structure

getUser(1)

.then((user) => getRepositories(user.gitHubUsername))

.then((repos) => getCommits(repos[0]))

.then((commits) => console.log('commits: ', commits))

.catch((err) => console.log('Error: ', err.message));

console.log('After');

function getUser(id) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log(`Reading a user with id: ${id} from a database...`);

resolve({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

});

}

function getRepositories(username) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Repository');

resolve(['repo1', 'repo2', 'repo3']);

}, 2000);

});

}

function getCommits(repo) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Commits');

resolve(['commit1', 'commit2', 'commit3']);

}, 2000);

});

}

## Creating Settled Promises

We will see API of promise object in javascript . Sometime you want to create a promise which is already resolved.particulary useful when writing a unit tests. So we want to simulate a scenario where an async operations like calling a webservice completes successfully. In usint test you want to create a promise that is already resolved

Promise has a static method called resolve. This will returen a promise that is already resolved. We cn optionally pass a value 1 or user object.

Similary sometime you want to create a promise that is already rejected. So instead of calling resolve method call reject method. And here you pass an error object. So when our promise is rejected we need to call catch to get the error.

Create a file promise-api.js

// const p = Promise.resolve({ id: 1 });

// p.then((result) => console.log('result', result));

const p = Promise.reject(new Error('reason for rejections...'));

p.catch((err) => console.log('Error', err));

//p.catch((err) => console.log('Error', err.message));

output

A screenshot of a computer

Description automatically generated

## Running Parallel Promises

Sometime you want to run few asynchronous operations in parallel, and when they all complete, you want to do something after. For example you may call different api’s like Facebook api and twitter api, and when the result of both these asynchronous operations are ready then you want something return to the client.

const p1 = new Promise((resolve) => {

setTimeout(() => {

console.log('Calling async operation: Facebook API');

resolve(1);

}, 2000);

});

const p2 = new Promise((resolve) => {

setTimeout(() => {

console.log('Calling async operation: Twitter API');

resolve(2);

}, 2000);

});

//Want both to run parallel and when the result of both these asynchronous

//operations are ready then you want something return to the client

Promise.all([p1, p2]).then((result) => console.log(result));

A black sign with white text

Description automatically generated

Output: Return an array of object

If one of the operations has error. The promise is rejected.

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling async operation: Facebook API');

reject(new Error('Error in calling Facebook API...'));

}, 2000);

});

const p2 = new Promise((resolve) => {

setTimeout(() => {

console.log('Calling async operation: Twitter API');

resolve(2);

}, 2000);

});

//Want both to run parallel and when the result of both these asynchronous

//operations are ready then you want something return to the client

Promise.all([p1, p2])

.then((result) => console.log(result))

.catch((err) => console.log('Error: ', err.message));

Output

A close up of a sign

Description automatically generated

Now let say you want the promise to complete as soon as one of the async operation completes

const p1 = new Promise((resolve) => {

setTimeout(() => {

console.log('Calling async operation: Facebook API');

resolve(1);

//reject(new Error('Error in calling Facebook API...'));

}, 2000);

});

const p2 = new Promise((resolve) => {

setTimeout(() => {

console.log('Calling async operation: Twitter API');

resolve(2);

}, 2000);

});

//Want both to run parallel and when the result of both these asynchronous

//operations are ready then you want something return to the client

Promise.race([p1, p2])

.then((result) => console.log(result))

.catch((err) => console.log(err.message));

Output

A close up of a sign

Description automatically generated

## Async and Await

Async and await helps you to write asynchronous code like synchronous code.

When there is no error

//Async and await approch

async function displayCommits() {

try {

const user = await getUser(1);

const repos = await getRepositories(user.gitHubUsername);

const commits = await getCommits(repos[0]);

console.log('commits', commits);

} catch (err) {

console.log('Error: ', err.message);

}

}

displayCommits();

console.log('After');

function getUser(id) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log(`Reading a user with id: ${id} from a database...`);

resolve({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

});

}

function getRepositories(username) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Repository');

resolve(['repo1', 'repo2', 'repo3']);

//reject(new Error('Could not get the repositories!!!'));

}, 2000);

});

}

function getCommits(repo) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Commits');

resolve(['commit1', 'commit2', 'commit3']);

}, 2000);

});

}

A screenshot of a cell phone

Description automatically generated

When there is error from and async operation

//Async and await approch

async function displayCommits() {

try {

const user = await getUser(1);

const repos = await getRepositories(user.gitHubUsername);

const commits = await getCommits(repos[0]);

console.log('commits', commits);

} catch (err) {

console.log('Error: ', err.message);

}

}

displayCommits();

console.log('After');

function getUser(id) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log(`Reading a user with id: ${id} from a database...`);

resolve({ id: id, gitHubUsername: 'Uttam' });

}, 2000);

});

}

function getRepositories(username) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Repository');

//resolve(['repo1', 'repo2', 'repo3']);

reject(new Error('Could not get the repositories!!!'));

}, 2000);

});

}

function getCommits(repo) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Calling GitHub API - Commits');

resolve(['commit1', 'commit2', 'commit3']);

}, 2000);

});

}

Output

A screenshot of a cell phone

Description automatically generated