Go back

Using async/await in ExpressJS middlewares

Feb 15, 2018 | 12:10 PM

If you are not living in a cave for the past year you'll probably know the **async/await** keywords are one of the most interesting additions on ES7. It merges the benefits of a sequential syntax with the power of asynchronous programming.

In this post we will cover how we must use correctly async functions as express middleware.

async/await

async/await is an extremely useful notation. There are plenty of good articles explaining them and how to use it and, IMO, there is an extremely useful visual explanation in 7 secs: **Async/Await in JavaScript, 7 seconds**.

Simply compare the syntax evolution from callbacks, passing through the use of promises until async/await (extracted from **Asynchronous JavaScript: From Callback Hell to Async and Await**):

Copy

```
// Verifying a user using callbacks
const verifyUser = function (username, password, callback) {
  dataBase.verifyUser(username, password, (error, userInfo) => {
    if (error) {
      callback(error);
    } else {
      dataBase.getRoles(username, (error, roles) => {
        if (error) {
          callback(error);
        } else {
          dataBase.logAccess(username, error => {
```

```
            if (error) {
              callback(error);
            } else {
              callback(null, userInfo, roles);
            }
          });
        }
      });
    }
  });
};
```

Copy

```
// Verifying a user with promises
const verifyUser = function (username, password) {
  database
    .verifyUser(username, password)
    .then(userInfo => dataBase.getRoles(userInfo))
    .then(rolesInfo => dataBase.logAccess(rolesInfo))
    .then(finalResult => {
      //do whatever the 'callback' would do
    })
    .catch(err => {
      //do whatever the error handler needs
    });
};

// Here you must implement `database.verifyUser`, `database.getRoles`
// and `database.logAccess` as promises, i.e:
const getRoles = new (function (userInfo) {
  return new Promise((resolve, reject) => {
    database
      .connect()
      .then(connection => connection.query("get roles sql"))
      .then(result => resolve(result))
      .catch(reject);
  });
})();
```

Copy

```
// Verifying a user with async/await
const verifyUser = async function (username, password) {
  try {
    const userInfo = await dataBase.verifyUser(username, password);
```

```
    const rolesInfo = await dataBase.getRoles(userInfo);
    const logStatus = await dataBase.logAccess(userInfo);
    return userInfo;
  } catch (e) {
    //handle errors as needed
  }
};

// Here we use the same `database.verifyUser`, `database.getRoles`
// and `database.logAccess` implementation based on promises
```

As you can see the async/await notation is more clear, in the sense it visually looks like a sequential set of imperative sentences, but with the powerful of JS asynchronous programming.

Notes on async/await

When you use async/await you are responsible to handle errors at the point you desire. In the previous example we could also write:

Copy

```
// Verifying a user with async/await
const verifyUser = async function (username, password) {
  const userInfo = await dataBase.verifyUser(username, password);
  const rolesInfo = await dataBase.getRoles(userInfo);
  const logStatus = await dataBase.logAccess(userInfo);
  return userInfo;
};
```

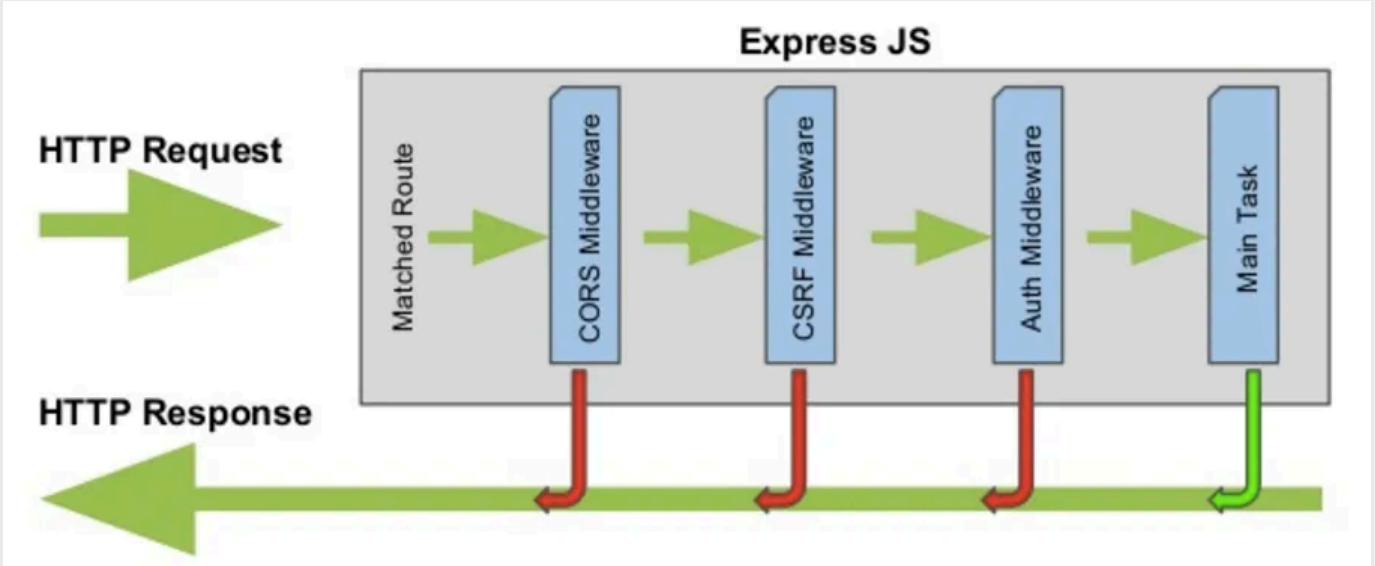
The issue is if **verifyUser** fails at some point the function will thrown an exception that should be catch by caller function (there is nothing new, this is the same catch we can use for promises.):

Copy

```
async function run() {
  try {
    const userInfo = await verifyUser();
    // Do something with the info
  } catch (error) {
    // Do whatever
  }
}
```

Middlewares

Express is one of the most famous and used NodeJS frameworks. Among other things it adds the concept of middleware. Given a HTTP request (also the response) we can imagine a pipeline to traverse, where on each step a task is made: check request is authenticated, parse body and "inject" as an extra param in the request, check params are right, do some business logic, etc.



In express, a middleware is nothing more than a callback function that receives three params: **`function middleware (request, response, next) {}`**

- **`request`** : Reference to the object representing the HTTP request. We use it to get any data associated to the request: body, url, headers, etc.
- **`response`** : Reference to the object representing the HTTP response. We need it to write a response: response code, body, headers, etc.
- **`next`** : Callback we need to execute if we want to continue the pipeline of middlewares.

Copy

```
const express = require("express");

const app = express();

app.get("/hello", (req, res, next) => {
  response.status(200).end("This is a not async/await middleware");
});
```

How to use async/await functions as middlewares

Simply remember to handle async/await errors. So **never to this**:

Copy

```
// NEVER DO THIS !!!
app.get("/hello", async (req, res, next) => {
  // Some code here
});
```

Because if for some reason the code inside the async function fails it will throw the error to the caller function (which is expressjs) and it will never be handled.

The right way would be as:

Copy

```
// DO THIS !!!
app.get("/hello", async (req, res, next) => {
  try {
    // Do something
    next();
  } catch (error) {
    next(error);
  }
});
```

Inside the middleware we make some actions and if things goes fine we invoke the next middleware or catch the error and invoke the next middleware with the error, this way expressjs will detect and handle the error.

Applying some DRY

One thing we can do to avoid repeating the try/catch code on each async middleware is write once in a high order function.

Copy

```
const asyncHandler = fn => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next);
```

The **asyncHandler** receives a function and returns a function with three input params (oh wait!!! that's like a middleware function). This new function is responsible to executes the original function passing the three params and catching any error.

Now we can rewrite our asynchronous middlewares like:

Copy

```
app.get(
  "/hello",
  asyncHandler((req, res, next) => {
    // Some code here. Any error will be catch and pass to expressjs
  })
);
```

Conclusions

My advice is: embrace **async/await**. It is very powerful notation one step beyond promises. Simply remember do not believe in magic and handle errors (the same way like with promises and callbacks) and remember to apply this too when working with expressjs.

#async #await #nodejs #express

Share this post on:

Back to Top

Previous Post
Express API with autogenerated OpenAPI doc through Swagger