בס"ד

קורס NodeJS תשפה

Asynchronous Syntax

כתיבת קוד אסינכרוני בnodejs מאפשרת לבצע פעולות שונות בצורה מקבילית, תוך ניהול של סדר הפעולות והתלויות ביניהן. ישנן דרכים שונות לכתוב קוד אסינכרוני:

1. Callbacks
2. Promises
3. async / await

כדי להבין לעומק את הניהול האסינכרוני, נפרט על כל אחת מהצורות. הדרך השלישית (**async / await**) היא הקלה, היעילה והשימושית ביותר, ובה נשתמש בד"כ.

Callbacks

ההגדרה של callback היא - פונקציה שנשלחת לפונקציה אחרת כפרמטר. בהקשר של כתיבה אסינכרונית בnodejs, callback היא פונקציה שנשלחת כפרמטר לפונקציה אסינכרונית – פונקציה שנשלחת לביצוע ע"י ספק חיצוני (פעולות I/O בעיקר), כדי שתתבצע לאחר סיום הפונקציה האסינכרונית.

פונקציית הcallback שימושית כשרוצים לחכות לפעולה אסינכרונית שתסתיים לפני ביצוע פעולה אחרת – המתנה לtimer, שימוש בערך מוחזר מהפונקציה, שליטה על סדר בין פעולות שונות וכו'.

לדוגמא, קריאה מקובץ – זו פעולה אסינכרונית שנשלחת לביצוע ע"י מערכת ההפעלה. אם נרצה להשתמש בערך המוחזר מהפונקציה, ז"א בתוכן של הקובץ שנקרא, נוכל לעשות זאת ע"י callback שנעביר שמקבל את התוכן של הקובץ ומשתמש בו:

```
import fs from 'fs';

fs.readFile('file.txt', (err: any, data: Buffer) => {
    console.log(data.toString());
});
```

קטע הקוד כולל קריאה של הקובץ, והדפסה של התוכן שלו לconsole. הcallback שמועבר כפרמטר ירוץ רק לאחר סיום קריאת הקובץ, כשהתוכן זמין.

רוב הפונקציות האסינכרוניות הזמינות לשימוש בnodejs או בחבילות npm השונות, מקבלות פרמטר אופציונלי (אחרון) של callback. הcallback היא פונקציה שמקבלת כפרמטר ראשון תמיד err – משתנה זה יהיה null כשהפונקציה התבצעה בהצלחה, ויכיל שגיאה אחרת. הפרמטר הבא הוא הערך המוחזר של הפונקציה האסינכרונית, והוא אופציונלי ולא יהיה קיים במידה והפונקציה לא מחזירה ערך.

הפרמטר הראשון של הcallback, הerr, חשוב כדי לנהל שגיאות בפונקציות אסינכרוניות. היות והקוד האסינכרוני רץ מחוץ לthread הראשי של הnodejs, ע"י המערכת הפעלה או ספק חיצוני אחר, ללא callback השגיאות ייזרקו ולא ינוהלו בשום מקום (שימוש בtry ו catch לא יתפוס את השגיאה כי הקוד לא רץ בthread אלא מחוצה לו).

קטע הקוד הבא כולל כתיבה לקובץ, עם ניהול שגיאות:

```
import fs from 'fs';

fs.writeFile('file2.txt', 'Some text', (err: any) => {
    if (err) {
        console.log("Failed writing to file:", err)
    }
});
```

בתחילת הcallback קיימת בדיקה אם err קיים, הפרמטר err יהיה null במקרה של הצלחה ואז לא נרצה להדפיס שגיאה. אם err שונה מnull, מודפסת הודעת שגיאה לconsole – בדוגמא זו זה יקרה למשל כשהמשתמש לא מורשה לכתוב לקובץ.

באמצעות callbacks ניתן לשלוט על סדר הפעולות האסינכרוניות, להמתין לפעולות שהתרחשו ולנהל שגיאות. נשים לב שכל פעולה אסינכרונית שנרצה לחכות לה, המשך הקוד ייכתב בתוך callback.

לדוגמא, אם נרצה לכתוב תכנית בה סדר הפעולות הוא קריאה מקובץ של פרמטר, ביצוע קריאת API עם הפרמטר שקראנו וכתיבה של התוצאה שחזרה מהקריאת API לקובץ, הקוד יכיל שלושה callbacks מקוננים:

```
import fs from 'fs';
import superagent, {Response} from 'superagent'

fs.readFile('input.txt', (err: any, data: Buffer) => {
    if (err) {
        console.log("Failed reading input data:", err);
        return;
    }

    const input = data.toString().split(" ");
    const from = input[0];
    const to = input[1];
    const amount = parseInt(input[2]);

    const url =
`https://api.frankfurter.dev/v1/latest?from=${from}&to=${to}`;
    superagent.get(url, (err: any, res: Response) => {
        if (err) {
            console.log(`API request to ${url} has failed: ${err}`);
            return;
        }

        const currency = res.body.rates[to];
        const result = (currency * amount).toString();

        fs.writeFile('output.txt', result, (err: any) => {
            if (err) {
                console.log("Failed writing result to file:", err)
            }
        });
    });
});
```

כפי שניתן לראות, ככל וכמות הקוד גדלה, כך רמת הסיבוכיות של הכתיבה בcallbacks מתגברת, ונוחות הכתיבה וההבנה יורדת משמעותית. הקוד נהיה מסורבל ולא קריא. לכן, כדי לנהל קוד אסינכרוני בצורה נוחה ויעילה יותר, החל מגרסת ES6 של javascript נוסף מנגנון שתומך בpromises.

Promises

Promise הוא אובייקט מיוחד בjavascript שמגדיר משימה אסינכרונית שמסתיימת בהצלחה או כישלון ומחזירה ערך (או שגיאה במקרה של כשלון). הpromise משמש כplaceholder לערך המוחזר מהפונקציה, שעדיין לא זמין כי הפעולה היא אסינכרונית, אך יהיה זמין בעתיד.

הpromise יכול להיות בשלושה מצבים אפשריים:

1. **Pending** – המצב ההתחלתי, בו הפעולה האסינכרונית נשלחה לביצוע ועדיין רצה.
2. **Fulfilled** – הפעולה הסתיימה בהצלחה, והpromise קיבל resolve עם הערך המוחזר.
3. **Rejected** – הפעולה נכשלה, והpromise קיבל reject עם השגיאה.

נשים לב שבevent loop, ישנו queue מיוחד לביצוע פעולות של promises שהסתיימו, כך שהם מקבלים עדיפות על פני המשימות שבqueue הרגיל.

יצירת promise

האובייקט של promise בconstructor מקבל פונקציה, שמקבלת שני פרמטרים – resolve ו reject. שני הפרמטרים הם פונקציות בפני עצמם, שייקראו כאשר הpromise יצליח או ייכשל.

לדוגמא:

```
const simplePromise = new Promise((resolve, reject) => {
    const success = true;
    if (success) {
        resolve("Operation was successful!");
    } else {
        reject("Something went wrong.");
    }
});
```

בקטע הקוד הpromise מייצג פונקציה פשוטה שמכילה פעולה סינכרונית – מסיימת את הpromise מייד, עם הצלחה או כישלון כתלות במשתנה success.

כמובן שבד"כ הpromise יכיל פעולה אסינכרונית. לדוגמא, קטע הקוד הבא מכיל promise שמייצג פעולה אסינכרונית של קריאה מקובץ:

```
import fs from 'fs';

const readFilePromise = new Promise((resolve, reject) => {
    fs.readFile('file.txt', 'utf8', (err, content: String) => {
        if (err) {
            reject(err);
        }

        resolve(content);
    });
});
```

כשהpromise יזומן הוא יהיה במצב pending, כשהקריאה מהקובץ תסתיים בהצלחה הפונקציה resolve תיקרא, ותגרום לpromise לעבור למצב fulfilled. אם תתרחש שגיאה בזמן הקריאה מהקובץ, הפונקציה reject תיקרא ותגרום לpromise לעבור למצב rejected.

שימוש בpromises

השימוש בpromises הוא באמצעות זימון של הפונקציות **then**, **catch** ו**finally** עליהם. שלוש הפונקציות מקבלות כפרמטר פונקציה שמשמשת לניהול המצבים השונים:

- הפונקציה שמועברת ל**then** תיקרא במצב שהpromise הצליח. היא מקבלת פרמטר שמכיל את הערך המוחזר של הpromise (הערך שמועבר כפרמטר לפונקציה resolve)
- הפונקציה שמועברת ל**catch** תיקרא במצב שהpromise נכשל. היא מקבלת פרמטר שמכיל את השגיאה (הערך שמועבר כפרמטר לפונקציה reject)
- הפונקציה שמועברת ל**finally** תיקרא בכל מקרה אחרי הפונקציות הקודמות, במקרה של הצלחה או כישלון.

לדוגמא שימוש בpromise שיצרנו קודם:

```
readFilePromise.then((data) => {
    console.log("DATA:", data);
}).catch((err) => {
    console.log("ERROR:", err);
}).finally(() => {
    console.log("The promise has completed");
});
```

כמובן שזימון הפונקציות then, catch וfinally אינו חובה וניתן לזמן חלק מהן או בכלל לא.

למעשה, פונקציות אסינכרוניות רבות – המובנות בnodejs או קיימות בחבילות npm השונות, קיימות בצורה של promise כך שלא צריך ליצור promise ידנית אלא ניתן להשתמש בפונקציה ישירות ולהפעיל עליה את then, catch וfinally.

לדוגמא קריאת API באמצעות החבילה superagent, הפונקציה get קיימת גם עם ערך החזרה של promise, כך שניתן לזמן אותה עם then וcatch:

```
import superagent from 'superagent';

superagent
    .get('url')
    .then((data) => {
        // Some operations ...
    })
    .catch((err) => {
        // Error handling ...
    });
```

Promises chaining

אחד מהיתרונות הגדולים של השימוש בpromises, הוא היכולת לשרשר כמה promises יחד כך שפעולות אסינכרוניות שונות ירוצו אחת אחרי השנייה.

צורת הכתיבה היא שפונקציית then מחזירה promise בעצמה, כך שניתן לקרוא שוב לthen כדי לנהל את התוצאה של הpromise השני.

לדוגמא, קטע הקוד הבא מכיל את אותה לוגיקה של קריאה מקובץ, שליחת תוכן הקובץ כפרמטר לקריאת API וכתיבת הresult של הקריאה לקובץ נוסף:

```
import {promises} from 'fs';
import superagent, {Response} from 'superagent'

let to: string;
let amount: number;
promises.readFile('input.txt', 'utf8')
    .then((data: String) => {
        const input = data.split(" ");
        const from = input[0];
        to = input[1];
        amount = parseInt(input[2]);

        const url =
`https://api.frankfurter.dev/v1/latest?from=${from}&to=${to}`;
        return superagent.get(url);
    }).then((res: Response) => {
        const currency = res.body.rates[to];
        const result = (currency * amount).toString();

        return promises.writeFile('output.txt', result)
    }).then(() => {
        console.log("Output was written successfully!");
    }).catch((err: any) => {
        console.log("Error in one of the promises:", err);
    });
```

כפי שניתן לראות בדוגמא, promises chaining מאפשר כתיבת קוד מורכב שכולל פעולות רבות בצורה קריאה ונוחה.

כמו כן, ניתן להוסיף קריאה לפונקציה catch בכל שלב בchaining כדי לתפוס שגיאות בשלבים שונים, להוסיף finally היכן שנדרש, לכלול רצף של promises בפונקציה ולהשתמש בה ברצף אחר ועוד.

Async / Await

כדי לפשט את השימוש בpromises, בגרסאות מתקדמות של js נוספו הkeywords - **async** ו**await**.

- **async** – מגדירה פונקציה כאסינכרונית וגורמת לערך המוחזר ממנה להיות תמיד promise.
- **await** – מחכה לסיום promise שרץ בתוך פונקציה שמסומנת בasync.

שימוש בkeywords האלו מאפשר לכתוב קוד אסינכרוני כחלק אינטגרלי מהקוד של המערכת, כך שקל יותר לקרוא ולתחזק אותו.

פונקציה שמסומנת בasync ולא מחזירה promise, הערך המוחזר נעטף באופן אוטומטי בpromise והופך להיות promise. כך, ניתן לזמן את הפונקציות then ו catch על הפונקציה.

כל פונקציית promise ניתנת לזימון עם await, כך שהקוד לא ימשיך לרוץ עד שהpromise יושלם. ניתן לקבל את הערך המוחזר מהpromise כמו כל ערך המוחזר מפונקציה, מבלי צורך להשתמש בcallback או בthen.

לדוגמא בקטע הקוד הבא:

```
import { promises } from "fs";

async function readFileAsync(filePath: string): Promise<void> {
    const content = await promises.readFile(filePath, 'utf8');
    console.log("File content:", content);
}

readFileAsync("file.txt");
```

הפונקציה readFileAsync מסומנת כasync, כך שניתן לזמן בתוכה פונקציות אסינכרוניות עם await. הפונקציה readFile מחזירה promise, ולכן ניתן לזמן אותה עם await. כך, כשהפונקציה readFileAsync נקראת הpromise של readFileAsync מתחיל לפעול, וריצת הפונקציה מושהית עד לסיום קריאת הקובץ. כמובן, הevent loop של nodejs ממשיך לפעול ברקע ולבצע פעולות נוספות.

כאשר קריאת הקובץ מסתיימת, התוכן שלו נכתב למשתנה content, ריצת הפונקציה ממשיכה והתוכן נכתב לconsole.

הפונקציה readFileAsync מחזירה promise, לכן אם נרצה להשתמש בה בפונקציות אחרות ניתן לזמן אותה עם await ולהבטיח שהקובץ נקרא לפני ביצוע פעולות אחרות.

בשימוש בsyntax של async / await, ניהול השגיאות נעשה באופן דומה לניהול שגיאות בקוד סינכרוני – באמצעות try וcatch. לדוגמא:

```
import { promises } from "fs";

async function readFileAsync(filePath: string): Promise<void> {
    try {
        const content = await promises.readFile(filePath, 'utf8');
        console.log("File content:", content);
    } catch (err) {
        console.log("Failed reading file:", err);
    }
}

readFileAsync("file.txt");
```

באופן זה, הכתיבה באמצעות async / await היא נוחה, קריאה ומהירה. ניהול הקוד האסינכרוני הופך לפשוט כמו ניהול קוד סינכרוני וניתן לשלב ביניהם בצורה זורמת ופשוטה.

לדוגמא, קטע הקוד הבא מכיל את אותה לוגיקה של קריאה מקובץ, שליחת תוכן הקובץ כפרמטר לקריאת API וכתיבת הresult של הקריאה לקובץ נוסף:

```
import {promises} from 'fs';
import superagent from 'superagent'

async function apiWithFiles(): Promise<void> {
    try {
        const data = await promises.readFile('input.txt', 'utf8');
        const input = data.split(" ");
        const from = input[0];
        const to = input[1];
        const amount = parseInt(input[2]);

        const url =
`https://api.frankfurter.dev/v1/latest?from=${from}&to=${to}`;
        const res = await superagent.get(url);

        const currency = res.body.rates[to];
        const result = (currency * amount).toString();

        await promises.writeFile('output.txt', result)
        console.log("Output was written successfully!");
    } catch (err) {
        console.log("Error in one of the promises:", err);
    }
}

apiWithFiles();
```

לסיכום:

ניתן לכתוב קוד אסינכרוני בnodejs בדרכים שונות שהתפתחו לאורך הזמן. הsyntax המתקדם ביותר הוא של async / await שמאפשר לכתוב קוד אסינכרוני באופן אינטגרלי עם הקוד הרגיל.

בכתיבת קוד אסינכרוני בnodejs נשתמש תמיד בasync / await תוך הבנת הpromises והתהליך האסינכרוני שמתבצע באמצעות הevent loop.

פונקציות Promise נוספות

המחלקה Promise כוללת פונקציות סטטיות שעוזרות לנהל promises. נפרט כמה מהן:

Promise.all

הפונקציה מקבלת מערך של promises ומחזירה promise חדש שמסתיים רק כאשר כל הpromises מסתיימים בהצלחה. אם אחד מהpromises נכשל, הpromise ייכשל מיידית.

לדוגמא בקטע הקוד הבא, הפונקציה תסתיים רק לאחר שהכתיבה לשני הקבצים תסתיים:

```
import {writeFile} from 'fs/promises';

async function writeMultipleFiles() {
    const promise1 = writeFile("file1.txt", "content1");
    const promise2 = writeFile("file2.txt", "content2");

    await Promise.all([promise1, promise2]);
}
```

נשים לב שהזימונים לפונקציה writeFile הם ללא await, ז"א יצרנו promises בלי לחכות לסיום שלהם, כדי לחכות לשניהם יחד בעזרת promise.all.

אם הכתיבה לאחד הקבצים תיכשל, הפונקציה תיכשל מיד בלי לחכות לכתיבה של הקובץ השני.

הפונקציה מחזירה promise שערך הresolve שלו הוא מערך של התוצאות של הpromises השונים לפי הסדר שלהם.

פעולה זאת שימושית לביצוע פעולות שונות בו זמנית.

Promise.allSettled

הפונקציה מקבלת מערך promises ומחזירה promise שמחכה לכל הpromises, אך בשונה מpromise.all הפונקציה לא חוזרת מיידית במקרה של כישלון של אחד הpromises, אלא מחכה לסיום של כולם בהצלחה או בכישלון.

הפונקציה מחזירה promise שערך הresolve שלו הוא מערך של התוצאות של הpromises השונים לפי הסדר שלהם – לכל אחד ערך ההצלחה או הכישלון (השגיאה).

Promise.race

הפונקציה מקבלת מערך promises ומחזירה promise שמסתיים ברגע שאחד הpromises במערך מסתיים בהצלחה או כישלון.

לדוגמא בקטע הקוד הבא, הפונקציה תסתיים כאשר אחד מהtimeouts יגיע:

```
import { setTimeout } from 'timers/promises';

async function timersRace(time1: number, time2: number) {
    const promise1 = setTimeout(time1);
    const promise2 = setTimeout(time2);

    await Promise.race([promise1, promise2]);
}
```

הפונקציה שימושית בין היתר כדי להגדיר timeout לפעולות שעלולות להיות ארוכות.

פונקציות נוספות שכדאי להכיר: Promise.try(), Promise.reject(), Promise.resolve(), Promise.any() .

ניתן לקרוא עליהן פה:

https://nodejs.org/en/learn/asynchronous-work/discover-promises-in-nodejs#advanced-promise-methods