בס"ד

קורס NodeJS תשפה

Asynchronous Flow

בNode.js, קוד JavaScript רץ בסביבה **single-threaded** אך עם **מנגנון אסינכרוני** שמאפשר לטפל בבקשות מרובות בצורה יעילה. אסינכרוניות ב Node.js היא חיונית לעבודה עם קלט/פלט (I/O) כמו קריאות לקבצים, בקשות לרשת, וגישה למסדי נתונים.

setTimeout

הפונקציה setTimeout מאפשרת לתזמן פונקציה לריצה מאוחרת, בעוד x זמן.

הפונקציה אינה חוסמת את ריצת שאר הקוד, אלא בקריאה לפונקציה היא מתזמנת את הפעולה לריצה, וריצת התוכנה לא תסתיים עד שיגיע הזמן והקוד המתוזמן ירוץ, או עד שהtimer יבוטל.

דוגמת קוד:

```
console.log("Start");

setTimeout(() => {
    console.log("Executed after 2 seconds");
}, 2000);

console.log("End");
```

פלט:

```
Start
End
Executed after 2 seconds
```

setInterval

הפונקציה setInterval מאפשרת לתזמן ריצה של פונקציה בinterval – כל x זמן.

כמו הפונקציה setTimeout, גם הפונקציה setInterval אינה חוסמת את ריצת שאר הקוד. ריצת התוכנה לא תסתיים עד שהtimer יבוטל.

הפונקציות setTimeout וsetInterval מחזירות timer. כדי לבטל את התזמון או להפסיק את הריצה החוזרת ניתן לבטל את הtimer באמצעות קריאה לפונקציה clearInterval.

דוגמת קוד:

```
let count = 0;

const interval = setInterval(() => {
    console.log(`Count: ${count}`);
    count++;
    if (count > 5) clearInterval(interval);
}, 1000);
```

פלט:

```
Start
End
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Event Loop

Nodejs רץ על thread אחד. המשמעות היא, שבכל זמן נתון רק פעולה אחת יכולה לרוץ, ללא מקביליות.

כדי להריץ קוד מקבילי ביעילות, nodejs כולל מנגנון פנימי שנקרא event loop. הevent loop מאפשר לnodejs לבצע פעולות I/O (input / output - פעולות של שליחת מידע מהתוכנה החוצה או קבלת מידע מבחוץ לתוכנה. הפעולות החיצוניות יכולות להיות כתיבה או קריאה מהמערכת קבצים, קריאות API ועוד.) מבלי לחסום את הריצה, בעזרת שליחת הפעולות האלו לביצוע ע"י מערכת ההפעלה.

היות ומערכות הפעלה הן בד"כ multi-threaded, הן יכולות להריץ את הפעולות האלו במקביל ברקע מבלי להפריע לריצת התוכנה. כשביצוע פעולה מסתיים, מערכת ההפעלה מודיעה לnodejs שהפעולה הסתיימה וnodejs יודע לקבל את התוצאה שלה ולהריץ את הקוד הבא שתלוי באותה פעולה, אם יש כזה.

הסבר המנגנון

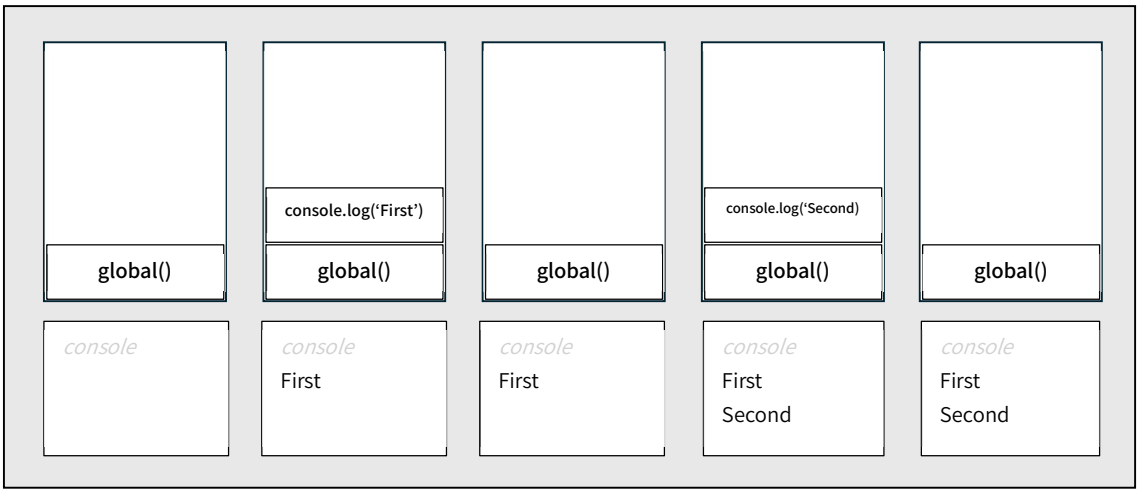
הרכיב הראשי שכלול בnodejs ומיועד להרצת קוד סינכרוני, הוא הcall stack. הstack כולל את הפונקציות שרצות בnodejs בעצמו ולא ע"י המערכת הפעלה.

הפונקציה הראשונה, אם אפשר לקרוא לה כך, שנכנסת לstack היא הglobal scope – הקובץ שאותו מריצים. כל פונקציה שנקראת בו נוספת לstack, ולאחר ההרצה שלה מוסרת ממנו.

לדוגמא בקטע הקוד הבא:

```
console.log(‘First’);
console.log(‘Second’);
console.log('Third’);
```

הglobal scope נכנס לstack, אחריו הקריאה הראשונה לconsole.log. אחרי הרצת הפונקציה, היא מוסרת מהstack והקריאה השניה לconsole.log נכנסת למחסנית, וכן הלאה:



כדי להריץ קוד אסינכרוני, הevent loop כולל בנוסף על הstack של nodejs כמה queues. פעולות אסינכרוניות שמחכות לריצה נכנסות לqueues בהתאם לסוג שלהן, וההרצה שלהן מתבצעת בזמן המתאים.

כדי לפשט את ההבנה נתעלם כרגע מהqueues השונים ונניח שכל הפעולות האסינכרוניות רצות על queue אחד.

בתהליך הרצת הקוד, כל פונקציה להרצה נכנסת לstack או לqueue – פעולות סינכרוניות נכנסות לstack, ופעולות אסינכרוניות לqueue.לפעולות הנמצאות בstack יש עדיפות על פני הפעולות הנמצאות בqueue. זה אומר שכל עוד קיימות פעולות בstack, הevent loop לא יתחיל לעבוד ולא ירוצו פעולות מהqueue.

תהליך ההרצה יתחיל תמיד בstack, כל פונקציה אסינכרונית תיכנס לqueue, וכאשר כל הפעולות הסינכרוניות ירוצו הevent loop יתחיל למשוך פעולות מהqueue ולהריץ אותן. כל פעולה שנמשכת מהqueue נכנסת לstack, וההרצה שלה מתנהלת כמו פונקציה סינכרונית בstack – אם היא קוראת לפונקציות סינכרוניות הן תיכנסנה לstack, ואם היא קוראת לפונקציות אסינכרוניות הן תיכנסנה לqueue. כך, הevent loop ירוץ עד סיום ביצוע כל הפעולות ועצירת התוכנה.

לדוגמא בקטע קוד הבא:

```
import {readFile} from 'node:fs';

console.log("Start");
setTimeout(() => {
    console.log("Timer call");
}, 2000);

readFile("file.txt", (txt) => {
    console.log("Reading file ended", txt);
});
console.log("End");
```

סדר הפעולות:

1. הglobal scope נכנס לstack.
2. console.log("Start") נכנס לstack, מתבצע ונמחק מהstack.
3. setTimeout נכנס לqueue.
4. readFile נכנס לqueue.
5. console.log("End") נכנס לstack, מתבצע ונמחק מהstack.
6. הglobal scope נמחק מהstack.

כעת הפעולות מהqueue מתחילות להתבצע:

1. readFile נשלח לביצוע ע"י מערכת ההפעלה.
2. מערכת ההפעלה מעדכנת שהקובץ נקרא בהצלחה, console.log("Reading file ended") נכנס למחסנית, מבוצע ונמחק מהמחסנית.
3. לאחר 2 שניות (2000 ms) הtimer נקרא, console.log("Timer call") נכנס למחסנית, מבוצע ונמחק מהמחסנית.

בשלב הנוכחי הstack ריק וגם הqueue ריק. ריצת הקוד מסתיימת.

פלט:

```
Start
End
Reading file ended Hello
Timer call
```

נשים לב שכאשר פונקציית setTimeout מזומנת בקוד, היא נכנסת לqueue כי היא פעולה אסינכרונית. לכן, אם הגיע זמן הריצה אך הstack לא ריק, הפעולה לא תבוצע בזמן המיועד אלא רק כשהstack יתרוקן.

לדוגמא בקטע קוד הבא:

```
console.log("Start");
setTimeout(() => {
    console.log("Timer call");
}, 1000);

for (let i = 0; i < 10000; i++) {}
console.log("End");
```

לאחר ריצת הlog הsetTimeout נכנס לqueue. בstack עדיין רץ הglobal scope – מתחילה לרוץ הלולאה. ריצת הלולאה לוקחת יותר משניה, ולכן הקוד שבתוך הsetTimeout ירוץ רק לאחר סיום הלולאה והלוג שאחריה, גם אם הזמן של הריצה כבר הגיע קודם.

לסיכום:

- קוד סינכרוני בnodejs רץ בstack.
- קוד אסינכרוני נכנס לqueue.
- כשהstack ריק, הevent loop שולף מהqueue פעולות אסינכרוניות שצריכות לרוץ, ומעביר אותן לריצה בstack.

סדר הפעולות בevent loop

ריצת הevent loop מורכבת מכמה שלבים שחוזרים על עצמם. בכל שלב רצות פעולות שונות שנכנסות מראש לqueues שונים. הqueues השונים כוללים queue לtimers (setTimeout, setInterval), queue לפעולות I/O ועוד.

סדר הפעולות נקבע ע"י הevent loop ותלוי בסדר ריצת השלבים השונים. ניתן לקרוא על כך עוד בקישור הבא:

https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick