Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/02BE10Strisko

ASINHRONI PRINCIPI U JAVASCRIPT PROGRAMSKOM JEZIKU

ASYNCHRONOUS PRINCIPLES IN JAVASCRIPT PROGRAMMING LANGUAGE

Marko Striško, *Fakultet tehničkih nauka, Novi Sad*

Oblast – RAČUNARSTVO I AUTOMATIKA

Kratak sadržaj – *Tema rada je proučavanje asinhronih principa i definisanih šablona prilikom rada sa asinhro-nošću u JavaScript programskom jeziku. Tokom rada su prikazani različiti principi koji omogućavaju rad sa asinhronim podacima, kako onih osnovnih, tako i onih za čije razumevanje je potrebno čvrsto znanje prethodnih osnovnih principa. Svaki princip sa sobom nosi svoje prednosti i mane, koje su adekvatno objašnjene prime-rima u obliku programskog koda.*

Ključne reči: *Asinhroni principi, JavaScript*

Abstract – *The subject of this paper is the study of asyn-chronous principles and defined patterns when working with asynchronism in the JavaScript programming lan-guage. Different principles that allow working with asynchronous data, both basic and those whose under-standing requires firm knowledge of the previous basic principles, are presented during the work. Each principle carries with it its advantages and disadvantages, which are adequately explained by examples in the form of a program code.*

Keywords: *Asynchronous principles, JavaScript*

1. UVOD

Asinhron (eng. ***asynchronous*** = vremenski neusklađeno, neistovremeno) princip je već duži period prisutan u svetu programiranja. Ustaljena je navika da se kod u najvećem broju slučajeva izvršava sinhrono (eng. ***synchronous*** = istovremen, jednakovremen), međutim postoje scenarija gde je asinhron princip mnogo efikasniji i primenljiviji, koji proističe iz svoje mogućnosti da se u isto vreme izvršava veći broj operacija. Suština problema je u tome što su sinhroni zahtevi potencijalno blokirajući, tj. postoji mogućnost čekanja određeni vremenski period da se operacija izvrši.

2. ASINHRONI I SINHRONI MODELI

U okviru ovog poglavlja, biće ukratko predstavljene glavne razlike između sinhronog i asinhronog modela.

2.1. Sinhroni programski model

Kod sinhronog modela, procesi se izvršavaju jedan za drugim. Kada se pozove izvršavanje određene funkcije, koja izvršava „dugoročni" process, rezulat će biti vraćen onog trenutka kada proces završi svoj posao.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Aleksandar Kupusinac, vanr. prof.

Na ovaj način, sistem koji izvršava navedeni proces biva zaustavljen sve do onog momenta dok se ne vrati krajnji rezultat izvršavanja. Ovo kao posledicu ima slučaj da će sledeći proces ili zadatak započeti tek po završetku prethodnog, što se može videti sa slike 1.

Slika 1. *Sinhrono izvršavanje, jedan tok izvršavanja*

Ukupno vreme izvršavanja biće suma vremena koje je trebalo da se ovi zadaci redom završe, što potencijalno predstavlja problem, u zavisnosti od toga kolika je suma kao krajnja vrednost [1].

2.2. Asinhroni programski model

Asinhroni model omogućava da se više procesa dešavaju u isto vreme. Glavna ideja asinhronosti je omogućavanje da korišćeni *thread* ne stoji blokiran, tj. "besposlen" čekajući da se zadatak izvrši, niti da se sama asinhrona aktivnost dešava u tom trenutno korišćenom *thread-u*. Kada započne izvršavanje izabranog zadatka, sistem će nastaviti svoje izvršavanje potpuno nesmetano, bez blokiranja ili čekanja da se započeti zadatak završi. Kada dođe trenutak uspešnog završetka zadatka, sistem biva obavešten o statusu i dobija pristup rezultatu izvršavanja. Slika 2. prikazuje sam redosled operacija u asinhronom programskom modelu [1].

Slika 2. *Asinhrono izvršavanje zadataka*

2.3. Razlike između asinhronosti i paralelizma.

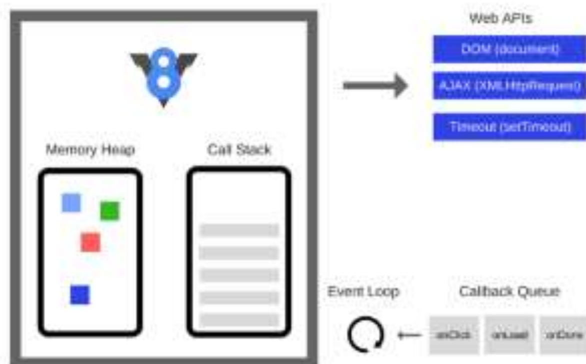
Česta je situacija da se zajedno spominju termini asinhro-nost i paralelno programiranje, ali su oni u osnovi prilično drugačiji. Suštininski gledano, predstavljaju dva načina implementacije konkurentnosti. *JavaScript* bazira svoju implementaciju konkurentnosti na jednom *thread-u.* Dok sa druge strane, ranije napomenuti *thread*-ovi, zajedno sa procesima, predstavljaju glavno „oruđe" prilikom paralel-nog programiranja. Navedena asinhronost, koja se postiže procesom koji se zove ***event loop*** deli posao na manje zadatke i izvršava ih serijski, tj. jedno za drugim, spreča-vajući na taj način paralelni pristup i promenu vredno-stima u deljenoj memoriji. Preklapanje izvršavanja para-lelnih niti i preklapanje asinhronih događaja dešavaju se na totalno drugačijim nivoima. U okruženju koje se iz-vršava upotrebom samo jednog *thread-a*, uopšte nije bitno koji tip operacija se izvršava, jer ništa ne može da prekine taj *thread.* Dok u paralelnim sistemima, gde su u opticaju veliki broj različitih *thread-ova* u istom programu, postoji velika šansa da se dese neplanirana scenarija [3].

3. ASINHRONOST I JAVASCRIPT

Pored nekoliko načina manipulacije izvršavanja asinhronih operacija korišćenjem *JavaScript* jezika, biće prikazana i struktura *JavaScript* okruženja, koja je od primarnog značaja za načine funkcionisanja asinhronih operacija.

3.1. JavaScript okruženje

JavaScript okruženje se sastoji od nekoliko komponenti. Glavna komponenta nosi naziv *JavaScript Engine* koja je najčešće ugrađena u pretraživač i web server, koji omogućava ***runtime compilation*** i nakon toga izvršavanje *JavaScript* koda. Navedena komponenta se sastoji od *Memory Heap-a* i *Call Stack-a*. U preostali deo spadaju *WEB APIs* i *Event Loop* zajedno sa *Callback/Event Queue-om* (slika 3).



Slika 3. *Celokupno JavaScript okruženje*

3.1.1. Call Stack

Call Stack predstavlja linearno struktuiranu memoriju, poseduje veliku brzinu, služi za privremeno čuvanje podataka (prostih primitiva, funkcija) i čuva informacije o tome gde se trenutno nalazimo u kodu. Ova strutkura podataka koristi LIFO (*Last In First Out*) princip. Kad funkcija bude pozvana, tada će biti stavljena na vrh *stack-a*. Kad funkcija vrati rezultat, ista biva izbačena iz *stack-a*.

3.1.2. Event Loop

JavaScript za izvršavanje asinhronih operacija koristi strukturu koja nosi naziv *Event Loop*. Svaki put kad se želi izvršiti neka od asinhronih operacija, prvo što se dešava jeste da se ta operacija dodaje u ***Event Table***. *Event table* je struktura podataka u kojoj je definisano koja funkcija je vezana za koji događaj. Kada se određeni događaj izvrši, funkcija čiji se događaj aktivirao biva poslata u ***Event Queue*** [4]. *Event queue* je takođe struktura podataka, opet se novi podaci dodaju na kraj, ali se samo mogu izbaciti spreda (FIFO). Na ovaj način se čuva tačan redosled ubacivanja u *queue*, da bi se znalo koja funkcija je naredna za obrađivanje. Da bi *event queue* poslao funkciju koja je prva na redu u *call stack*, koristi se prethodno spomenuti *event loop* [4]. *Event Loop* predstavlja proces koji se konstantno izvršava u pozadini i proverava da li je *call stack* prazan. Najlakše ga je zamisliti kao sat, koji kad god otkuca, uradi se provera da li je *call stack* prazan. Ako jeste, onda se izvršava provera stanja u *event queue*. Ako *event queue* sadrži funkcije u sebi, prva na redu se automatski prebacuje u *call stack* [4].

3.2. Callbacks

Asynchronous callbacks ili samo *callbacks* je princip za koji se može slobodno reći da je najčešći način na koji se asinhronost u *JavaScript-u* izražava i upravlja, tj. predstavlja prvobitni i najosnovniji asinhroni patern u *JavaScript* programskom jeziku.

3.2.1. Osnove callback principa

Callback nije ništa drugo do obična *JavaScript* funkcija prosleđena kao argument drugoj funkciji, za koju se očekuje da bude izvršena u nekom trenutku unutar funkcije kojoj je prosleđena. *Callback* funkcije su nasleđene od strane programske paradigme koja se zove funkcionalno progamiranje (eng. ***functional programming***). Glavni razlog postojanja *callback* funkcija je u tome što je *JavaScript* jezik vođen događajima (eng. ***event-driven language***) sa jednim izvršnim *thread-om*. Asinhron *callback* može biti iskorišćen kad se ne želi čekati završetak izvršavanja određene operacije. Ta operacija će na ovaj način postati neblokirajuća, ako se definiše kao *callback* [2].

3.2.2. Nedostaci callback principa

Callback princip zaista predstavlja osnovnu jedinicu asinhronosti u *JavaScriptu*. Ipak, praksa pokazuje da ovaj princip nije dovoljan za evolutivni napredak asinhronog programiranja.

3.2.2.1. Callback hell

Listing 1. prikazuje *setTimeout()* funkciju koja odlaže izvršenje logike za određen vremenski period. Prvi deo programa predstavljaju A i B delovi koda, dok drugi deo predstavlja C. Prvi deo se odmah izvršava i nakon toga dolazi do pauze neodređene dužine. U nekom trenutku u budućnosti, program će nastaviti gde je stao i nastaviti sa drugim delom [5].

```
1.  // A deo koda
2.  setTimeout(function () {
3.     // C deo koda
4.  }, 2000);
5.  // B deo koda
```

Listing 1. *Sinhrono razmišljanje tokom asinhronosti*

Iako je na operativnom nivou mozak predstavljen kao asinhron, ljudi ipak planiraju izvršavanje svojih svakodnevnih zadataka na sekvencijalni, tj. sinhroni način. Stoga, ako se sinhroni mozak dobro mapira na sinhrono izvršenje programske logike, koliko dobro mozak radi kad planira izvršenje asinhronog koda? Ljudsko razmišljanje se odvija korak po korak, ali alati (*callback* funkcije) dostupne u kodu se ne izražavaju po principu korak po korak onog trenutka kad se programski tok prebaci sa sinhronog na asinhrono.

Navedena isključenost je suština samog nedostatka *callback* principa prilikom rada i upravljanja asinhronim operacijama. Onog trenutka kad se predstavi kontinuacija tj. upravljanje tokom programa u obliku *callback* funkcija, omogućili smo razilaženje načina kako možak funkcioniše i način na koji će program funkcionisati [3].

3.2.2.2. Inversion of Control

Zasniva se na ideji da druga strana kojoj je predana *callback* kontinuacija nije funkcija napisana od strane programera koji razvija trenutni sistem, niti pod njegovom kontrolom, već je to funkcionalnost dostavljena od neke druge spoljašnje strane. Ovakva situacija nosi naziv „*inverzija kontrole*“ (eng. ***inversion of control***) gde se uzme deo programa i njegova izvršna kontrola se preda drugoj strani.

Glavni razlog zašto se ne preporučuje upotreba *callback* funkcija je to što ovaj patern ne pruža nikakvu zaštitu od potencijalnih grešaka dok je kontrola na drugoj strani. Sva neophodna mašinerija za takvu vrstu zaštite se mora lično izgraditi.

3.3. Promises

Najveći izazov koji nastaje prilikom rada sa *callback* funkcijama jeste nemogućnost izražavanja asinhronosti na što je moguće više sinhroni način, čime bi se što lakše ispratio tok programa. Rešenje za navedene probleme se pronalazi u obrascu koja nosi naziv ***Promises*** (eng. ***Promise*** = obećanje). Umesto predavanja kontinuacije programa drugoj strani, ovde se očekuje, kao povratni rezultat, mogućnost da se sazna u kom trenutku će zadatak, koji obavlja druga strana, biti završen i onda implementacija može da odluči šta će se dalje raditi [3].

Promise princip predstavlja vrednost koju možemo da iskoristimo u nekom trenutku u budućnosti i u odnosu na *callback* funkcije, pružaju garanciju za buduću vrednost. Jedna od najbitnijih koncepata *Promise* principa predstavlja činjenica da nijedna registrovana strana koja posmatra i čeka razrešenje *Promise* objekta ne može da promeni vrednost rezultata (slučajno ili namerno) nakon razrešenja (*Promise* objekti su nepromenljivi (*eng.* immutable)).

3.3.1. Stvaranje Promise objekta

Konstruktor *Promise-a* kao argumente prima dve funkcije. Prva funkcija (***resolve***) biće pozvana kada se asinhrona operacija izvršila uspešno. Njena povratna vrednost biće rezultat asinhrone operacije. Druga funkcija (***rejected***) poziva se kada se asinhrona operacija nije izvršila uspešno. Povratna vrednost *rejected* funkcije sadržaće *error* objekat [4]. Listing 2. prikazuje primer pravljenja *Promise* objekta.

```
1. var p = new Promise(function(resolve, reject) {
2.    if ( /* uslov */ ) {
3.       resolve( /* vrednost */ ); // uspešno razrešen
4.    } else {
5.       reject( /* razlog */ ); // greška, odbijen
6.    }
7. });
```

Listing 2. *Prikaz stvaranja Promise objekta*

Na osnovu navedenog listinga, *Promise* objekti poseduju tri moguća stanja:

- ***Pending***: Predstavlja početno stanje pre nego što operacija započne.
- ***Fulfilled:*** Predstavlja stanje uspešno izvršene operacije
- ***Rejected:*** Operacija se nije uspešno završila, greška je vraćena kao rezultat.

3.3.2. Korišćenje Promise principa

Da bi se iskoristio *Promise* objekat, potrebno je postaviti *handler* na *Promise* korišćenjem *then()* funkcije. Ova funkcija uzima kao parametar funkciju koja će biti prosleđena kao *resolve* vrednost, tj. ona će biti vraćena kao vrednost ako se *Promise* uspešno razreši, dok je drugi parametar funkcija koja će biti pozvana kada *Promise* ne bude uspešno razrešena.

3.3.3. Ulančavanje Promise-a

Promise ne predstavlja isključivo mehanizam za jednostavne pojedinačne korake u obliku *this-then-that*. Naravno, to predstavlja osnovu, ali postoji mogućnost povezivanja više *Promise* objekata zajedno. Svaki poziv *then()* funkcije na *Promise* objektu, stvara novi *Promise* objekat i vraća se kao povratna vrednost, na koji se može dalje ulančavati.

3.3.4. Nedostaci Promise principa

Prilikom ulančavanja *Promise* objekata, javlja se mogućnost da greška koja se javi u bilo kom razrešenju *Promise* objekta bude ignorisana, tj. propuštena. Ako se napravi lanac *Promise* objekata koji nema obradu grešaka uključenu u lanac, bilo koja greška bilo gde u lancu biće propagirana dole do dna lanca.

Promise po definiciji uvek imaju samo jednu uspešno razrešenu vrednost ili samo jedan neuspešni razlog za odbijanje. Onog trenutka kada se napravi *Promise* objekat i kada se registruju uspešna ispunjena ili odbijanja samog objekta, ne postoji način da bi se zaustavio započeti proces. Ova ideja narušava verodostojnost buduće vrednost i ideju o nepromenljivosti na kojoj je originalno *Promise* princip zasnovan [3].

3.4. Async / Await princip

Async/Await princip ili *Async Functions* kako još nosi naziv predstavlja sintaksu za kontrolisanje asinhronog programskog toka u *JavaScript-u*. Inicijalno je započeo u okviru ***C#*** i ***F#*** programskog jezika, nakon čega se pojavljuje kao standard u *EcmaScript2017*, tj. verziji 8. Ovaj princip u potpunost omogućava pisanje asinhronog koda na sinhroni način. Takođe raspolaže sa veoma jasnim i intuitivnim načinom za obradu grešaka, jer u sebe uključuje *try..catch* sintaksu.

```
1. async function getValueWithAsync() {
2.    const value = await this.resolveAfter2Second
   s(20);
3.    console.log(`Async rezultat: ${value}`);
4.    return value;
5. }
```

Listing 3. *Primer upotrebe async / await sintakse*

Obavezna je upotreba *async* ključne reči ako funkcija sadrži *await* ključnu reč, u suprotnom kompajler će prijaviti *SyntaxError* grešku. Svaka *async* funkcija implicitno vraća *Promise* objekat, dok će rezultat razrešenja tog objekta biti ono što je vraćeno iz *async* funkcije.

3.5. Observables

Glavni nedostaci *Promise* principa ugledaju se pre svega u tome da će se kao rezultat uvek vraćati samo jedna vrednost, tj. samo jedna vrednost će biti vraćena nakon uspešnog (ili neuspešnog) razrešenja *Promise* objekta. Još jedan nedostatak predstavlja nemogućnost prekidanja započetog izvršavanja *Promise* objekta. Navedene nedostatke nadoknađuje ***Observables*** princip.

Observable princip (eng. *Observable* = vidljiv, opažen) zasnovan je na *Observer* paternu. Ovaj patern funkcioniše tako što postoji *subjekat*, koji sadrži listu *objekata* koji zavise od njega. Prilikom bilo koje promene stanja, *subjekat* poziva *posmatrače* (eng. ***observer*** = posmatrač) koji automatski obaveštavaju zavisne *objekte* o promeni stanja *subjekta*. *Observables* princip predstavljen je kao deo *EcmaScript-a 2016*. Ovaj princip potiče iz **RxJS** (*Reactive Extensions Java Script*) biblioteke, koja predstavlja *Observables* princip kao novi *push* sistem za *JavaScript* [6].

Observables se može uporediti sa funkcijama. Sličnost se zasniva na tome da pozivanje *subscribe()* kod *Observables* principa analogno je pozivanju funkcije. Razlike između funkcije i *Observables* principa leži u tome da funkcije mogu da vrate jednu i samo jednu vrednost. Dok to ne važi za *Observables*, koji mogu da vrate koliko god vrednosti je potrebno [6].

3.5.1. Struktura Observables-a

Postoje četiri glavna aspekta koja se nalaze unutar *Observables* instanci, mada su neki od aspekata vezani za druge tipove, kao što su *Observer* i *Subscription*. Spomenuta četiri aspekta su:

- Stvaranje *Observable* instance.
- Pretplaćivanje (*subscribe*) na *Observable* instance.
- Izvršavanje *Observable* instance.
- Odlaganje *Observable* instance.

4. ZONE.JS

Ranije je spomenuto da je neophodno okruženje da bi se *JavaScript* mogao izvršavati, bilo unutar pretraživača ili unutar *Node.js runtime-a*. Navedena okruženja direktno su odgovorna za zakazivanje izvršavanja *JavaScript* operacija. Problem koji se javlja je efikasno praćenje izvršavanja svih zadataka, kako sinhronih, tako i asinhronih. *Zone.js* upravo pruža takvu mogućnost, stvarajući kontekst, tj. celinu u kome se izvršavaju asinhrone operacije, nad kojom stvaraoc takvog konteksta ima mogućnost da posmatra i kontroliše izvršavanje koda unutar navedene zone [7].

5. ZAKLJUČAK

JavaScript nesumnjivo predstavlja jedan od najpopularnijih jezika u programerskoj delatnosti. *JavaScript* se ne koristi više samo kao klijentski deo, već je sposoban i za rad nad serverskim delom, što omogućava stvaranje *full-stack* aplikacije potpuno pod kontrolom *JavaScript-a.* Međutim, ispostavlja se da njegova prisutna ekscentričnost čini potpuno ovladavanje svim principima koje poseduje kao izuzetno težak podvig. Jedan od glavnih principa prilikom rada sa *JavaScript-om* leži u njegovoj sposobnosti u izvršavanju više različitih zadatak u isto vreme, upotrebom asinhronosti. Upravo u tome leži sama ekscentričnost samog jezika, koja ovim omogućava izuzetne perfomanse. Iako je opšte poznato da se *JavaScript* može koristiti bez razumevanja, to isto razumevanje se često jako teško postiže.

6. LITERATURA

[1] https://eloquentjavascript.net/11_async.html (pristupljeno u junu 2018.)

[2] K. Gallaba, A. Mesbah and I. Beschastnikh, “*Don’t call us, we’ll call you: Characterizing Callbacks in JavaScript*”, Canada 2015.

[3] Kyle Simpson, *“ You Don’t Know JS: Async & Performance.”*, USA 2015.

[4] Daniel Parker, “*JavaScript with Promises*”, USA 2015

[5] Keheliya Gallaba, “*Characterizing and refactoring asynchronous JavaScript callbacks*”, University of Moratuwa, 2011

[6] Mark Clow, “*Observers, Reactive Programming and RxJS*”, California 2018.

[7] https://docs.google.com/document/d/-1F5Ug0jcrm031vhSMJEOgp1l-Is-Vf0UCNDY-LsQtAIY (pristupljeno u avgustu 2018.)

Kratka biografija:

Marko Striško, rođen je u Beogradu 1993. god. Osnovnu i srednju školu završio u Staroj Pazovi. Diplomirao 2016. god. na Fakultetu tehničkih nauka, smer računarstvo i automatika, na kojem iste godine upisuje master studije.