# SADRŽAJ

# UVOD

Ovo je knjiga o korišćenju računara. Računari su danas postali uobičajeni kao i šrafcigeri, ali su poprilično složeniji i nije uvek lako naterati ih da rade ono što vi hoćete.

Ukoliko je zadatak koji imate za računar uobičajen, dobro shvaćen, kao što je prikazivanje e-poruke ili obavljanje posla kalkulatora, možete otvoriti odgovarajuću aplikaciju i baciti se na posao. Ali za jedinstvene ili neodređene zadatke, verovatno ne postoji aplikacija.

Tu u igru ulazi programiranje. *Programiranje* je čin konstruisanja *programa* – skupa preciznih instrukcija koje govore računaru šta da radi. Pošto su računari glupi ali i pedantni, programiranje je u svojoj osnovi naporno i frustrirajuće.

Srećom, ako možete da prevaziđete tu činjenicu, a možda i uživate u krutom razmišljanju u pojmovima s kojima glupe mašine mogu da se izbore, programiranje vam može doneti zadovoljstvo. Ono vam omogućava da u sekundama obavite stvari koje biste ručno radili *beskrajno*. Ono je način da od svog računara napravite alat koji će raditi stvari koje ranije nije mogao da uradi. I odlična je vežba za apstraktno razmišljanje.

Većina programiranja obavlja se pomoću programskih jezika. *Programski jezik* je veštački konstruisan jezik koji se koristi za zadavanje instrukcija računarima. Zanimljivo je da najefikasniji način koji smo pronašli za komuniciranje s računarom uveliko preuzima elemente iz načina na koje komuniciramo

međusobno. Kao i govorni jezici, računarski jezici omogućavaju da se reči i fraze kombinuju na nove načine, zahvaljujući čemu je moguće izraziti nove koncepte.

U jednom periodu, interfejsi zasnovani na jeziku, kao što su komandni odzivnici u BASIC-u i DOS-u iz osamdesetih i devedesetih godina prošlog veka, bili su glavni način interakcije s računarima. Oni su uveliko zamenjeni vizuelnim interfejsima kojima se lakše ovladava, ali nude manje slobode. Računarski jezici su i dalje tu, ako znate gde da ih pronađete. Jedan takav jezik, JavaScript, ugrađen je u svaki moderan čitač veba i stoga je dostupan na gotovo svakom uređaju.

Ova knjiga će pokušati da vas upozna s tim jezikom do mere da sa njim možete da radite korisne i zabavne stvari.

## O programiranju

Pored objašnjavanja JavaScripta, predstaviću vam i osnovne principe programiranja. Kako se ispostavilo, programiranje je teško. Osnovna pravila su jednostavna i jasna, ali programi izgrađeni na tim pravilima obično postaju dovoljno kompleksni da uvode sopstvena pravila i složenost. Moglo bi se reći da gradite svoj lavirint, i mogli biste se izgubiti u njemu.

Dešavaće se da vam čitanje ove knjige deluje užasno frustrirajuće. Ukoliko ste početnik u programiranju, biće mnogo novog materijala koji treba da svarite. Mnogo tog materijala će se onda *kombinovati* na načine koji zahtevaju da pravite dodatne veze.

Na vama je da načinite potreban napor. Kada se budete mučili s praćenjem knjige, nemojte donositi preuranjen sud o sopstvenim sposobnostima. S vama je sve u redu – samo treba da budete uporni. Napravite pauzu, pročitajte ponovo materiju i pobrinite se da pročitate i razumete primere programa i vežbe. Učenje je naporan posao, ali sve što naučite biće vaše i učiniće kasnije učenje lakšim.

> Kada akcija postane neisplativa, prikupljajte informacije;
> Kada informacije postanu neisplative, spavajte.
> – Ursula K. Le Guin, *The Left Hand of Darkness*

Program je više od programa. To je deo teksta koji je napisao programer, to je upravljajuća sila koja tera računar da radi ono što radi, to su podaci u memoriji računara, ali upravljaju akcijama koje se izvode na toj istoj memoriji. Analogije koje pokušavaju da uporede programe sa objektima koji su nam poznati obično nisu adekvatne. Površno odgovarajuća je ona za mašinu – prisutno je mnogo odvojenih delova, a da bi cela stvar radila, moramo da uzmemo u obzir načine na koje su ti delovi povezani i na koje doprinose funkcionisanju celine.

Računar je fizička mašina koja služi kao domaćin za te nematerijalne mašine. Računari sami mogu da obavljaju samo glupo jednostavne stvari. Razlog zbog kog su tako korisni je što te stvari obavljaju neverovatno velikom brzinom. Program može genijalno da kombinuje ogromne brojeve prostih akcija da bi uradio veoma složene stvari.

Program je zgrada misli. Njena izgradnja ne košta ništa, ona nema težinu i lako raste pod našim prstima koji kuckaju.

Ali ako ne vodimo računa, veličina i složenost programa izmaći će kontroli, zbuniće čak i osobu koja ga je napravila. Držanje programa pod kontrolom glavni je problem u programiranju. Kada program radi, to je prelepo. Umetnost programiranja je veština upravljanja složenošću. Odličan program je potčinjen – pojednostavljen je u svojoj složenosti.

Neki programeri veruju da je tu složenost najbolje kontrolisati korišćenjem samo malog skupa dobro shvaćenih tehnika u programima koje pišu. Oni su sastavili stroga pravila („najbolje prakse“) koja propisuju formu koju program treba da ima i pažljivo ostaju unutar svoje komforne zone.

To ne samo da je dosadno, već je i neefikasno. Novi problemi često zahtevaju nova rešenja. Polje programiranja je mlado i još uvek se brzo razvija, i dovoljno je raznovrsno da ima prostora za sasvim različite pristupe. Postoji mnogo užasnih grešaka koje se mogu napraviti u dizajnu programa i ne treba da se ustežete da ih pravite, jer ćete ih tako razumeti. Osećaj za izgled dobrog programa razvija se praksom, ne učenjem liste pravila.

## Zašto je jezik bitan

U vreme nastanka računara nisu postojali programski jezici. Programi su izgledali otprilike ovako:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

To je program za sabiranje brojeva od 1 do 10 i prikazivanje rezultata: 1 + 2 + ... + 10 = 55. Mogao je da se izvede na jednostavnoj, hipotetičkoj mašini. Da bi se programirali rani računari, bilo je neophodno postaviti velike nizove prekidača u odgovarajući položaj ili bušiti rupe u kartonskim trakama i ubacivati ih u računar. Možete zamisliti koliko je ta procedura bila naporna i podložna greškama. Čak je i pisanje jednostavnih programa zahtevalo mnogo sposobnosti i discipline. Oni složeni su bili skoro nezamislivi.

Naravno, ručno upisivanje tih nedokučivih obrazaca bitova (jedinica i nula) davali su programeru duboki osećaj da je moćni čarobnjak. I to sigurno znači nešto kad je u pitanju zadovoljstvo poslom.

Svaki red prethodnog programa sadrži jednu instrukciju. Na srpskom bi se to moglo napisati ovako:

1. Uskladišti broj 0 na memorijsko mesto 0.
2. Uskladišti broj 1 na memorijsko mesto 1.
3. Uskladišti vrednost memorijskog mesta 1 na memorijsko mesto 2.

4. Oduzmi broj 11 od vrednosti na memorijskom mestu 2.
5. Ako je vrednost na memorijskom mestu 2 broj 0, nastavi sa instrukcijom 9.
6. Dodaj vrednost memorijskog mesta 1 memorijskom mestu 0.
7. Dodaj broj 1 vrednosti memorijskog mesta 1.
8. Nastavi sa instrukcijom 3.
9. Prikaži vrednost na memorijskom mestu 0.

Iako je ovo već čitljivije od bućkuriša bitova, još uvek je krajnje nejasno. Korišćenje imena umesto brojeva za instrukcije i memorijska mesta pomaže.

```
 Podesi "total" na 0.
 Podesi "count" na 1.
[petlja]
 Podesi "compare" na "count".
 Oduzmi 11 od "compare".
 Ako je "compare" nula, nastavi od [kraj].
 Dodaj "count" na "total".
 Dodaj 1 na "count".
 Nastavi od [petlja].
[kraj]
 Prikaži "total".
```

Vidite li kako program sada radi? Prva dva reda daju dvama memorijskim mestima početne vrednosti: total će se koristiti da bi se izgradio rezultat izračunavanja, a `count` će pratiti broj koji trenutno geldamo. Redovi koji koriste `compare` verovatno su najčudniji. Program želi da vidi da li je `count` jednako 11 da bi odlučio da li može prestati s radom. Pošto je naša hipotetička mašina krajnje primitivna, ona može da testira samo da li je neki broj nula i da donosi odluku na osnovu toga. Zbog toga koristi memorijsko mesto označeno sa `compare` da bi izračunala vrednost `count` – 11 i donela odluku na osnovu te vrednosti. Naredna dva reda dodaju vrednost `count` rezultatu i povećavaju `count` za 1 svaki put kada program odluči da `count` još uvek nije 11.

Evo istog programa u JavaScriptu:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// ! 55
```

Ova verzija daje više poboljšanja. Najbitnije od svega, više nema potrebe da zadajemo način na koji želimo da program skače napred nazad. Konstrukcija `while` se brine za to. Ona nastavlja da izvršava blok (obuhvaćen vitičastim zagradama) ispod nje sve dok je uslov koji joj je dat tačan. Taj uslov je `count` <= 10, što znači „*count* je manje ili jednako 10“. Više ne moramo da pravimo privremenu vrednost i poredimo je sa nulom, što je bio samo nezanimljiv detalj. Deo moći programskih jezika jeste u tome što mogu da se pobrinu za nezanimljive detalje.

Na kraju pograma, kada se konstrukcija `while` završi, operacija `console.log` se koristi za ispisivanje rezultata.

Najzad, evo kako bi program izgledao ako bi nam bile dostupne zgodne operacije `range` i `sum`, od kojih prva pravi kolekciju brojeva u nekom opsegu, a druga izračunava zbir kolekcije brojeva:

```
console.log(sum(range(1, 10)));
// → 55
```

Pouka ove priče je da se isti program može prikazati i naširoko i ukratko, na nejasan i na jasan način. Prva verzija programa je bila ekstremno nejasna, dok je poslednja gotovo ista kao engleski jezik: `log` (evidentiraj) `sum` (zbir) `range` (opsega) brojeva od 1 do 10. (U kasnijim poglavljima ćemo videti kako se definišu operacije kao što su `sum` i `range`.)

Dobar programski jezik pomaže programeru omogućavajući mu da na višem nivou govori o akcijama koje računar treba da izvrši. To pomaže da se izostave detalji, obezbeđuje zgodne gradivne blokove (kao što su `while` i `console.log`), dozvoljava vam da definišete sopstvene gradivne blokove (kao što su `sum` i `range`), i čini lakim uklapanje tih blokova.

## Šta je JavaScript?

JavaScript je predstavljen 1995. kao način da se programi dodaju veb stranama u čitaču Netscape Navigator. Taj jezik je u međuvremenu prihvaćen u svim glavnim grafičkim čitačima veba. On je učinio mogućim moderne veb aplikacije – aplikacije s kojima možete imati direktne interakcije, ne morajući ponovo da učitavate stranu za svaku akciju. JavaScript se koristi i na tradicionalnijim veb prezentacijama kako bi se omogućili razni vidovi interaktivnosti i sposobnosti.

Bitno je napomenuti da JavaScript nema skoro ništa sa programskim jezikom Java. Slično ime je bilo inspirisano marketinškim razlozima, pre nego dobrim rasuđivanjem. Kada je JavaScript uveden, jezik Java je bio obilno reklamiran i sticao je popularnost. Neko je pomislio da je dobra ideja ukačiti se na taj uspeh. I sada smo zaglavili sa ovim imenom.

Nekon njegovog prihvatanja van Netscapea, napisan je standardan dokument koji opisuje način na koji bi jezik JavaScript trebalo da radi da bi razni delovi softvera koji su tvrdili da podržavaju JavaScript zaista govorili o istom jeziku. On je nazvan ECMAScript standard, po organizaciji Ecma International koja je obavila standardizaciju. U praksi, pojmovi ECMAScript i JavaScript mogu se koristiti kao sinonimi – to su dva imena za isti jezik.

Ima onih koji će raći *užasne* stvari o JavaScriptu. Mnoge od tih stvari su tačne. Kada sam prvi put morao da napišem nešto u JavaScriptu, brzo sam počeo da ga prezirem. Prihvatao je gotovo sve što bih upisao, ali je to tumačio na način koji je potpuno različit od onoga što sam ja mislio. Naravno, razlog za to uveliko leži u činjenici da nisam imao pojma šta radim, ali tu postoji i jedan pravi problem: JavaScript je suludo liberalan u onome što

dozvoljava. Zamisao iza takvog dizajna bila je da će on učiniti programiranje na JavaScriptu jednostavnijim za početnike. Zapravo, on gotovo da otežava pronalaženje problema u programima jer vam ih sistem neće pokazati.

Ova fleksibilnost, ipak, ima i svoje prednosti. Ona ostavlja prostor za mnogo tehnika koje su nemoguće u krućim jezicima, a kao što ćete videti (na primer, u poglavlju 10), ona se može upotrebiti i za prevazilaženje nekih nedostataka JavaScripta. Kada sam ispravno naučio jezik i neko vreme radio s njim, naučio sam da *volim* JavaScript.

Postoji nekoliko verzija JavaScripta. ECMAScriptova verzija 3 je bila široko podržana verzija u vreme kada je JavaScript počinjao da dominira, otprilike između 2000. i 2010. Tokom tog vremena, u toku je bio rad na ambicioznoj verziji 4, za koju je planirano više radikalnih poboljšanja i proširenja jezika. Menjanje živog, naširoko korišćenog jezika na tako radikalan način pokazalo se kao politički teško i rad na verziji 4 je obustavljen 2008, vodeći ka znatno manje ambicioznoj verziji 5. Objavljena 2009, verzija 5 je donela samo neka nekontroverzna poboljšanja. Potom je, 2015, izašla verzija 6, krupna izmena koja je uključivala neke od ideja planiranih za verziju 4. Nakon toga, imali smo nove, male dopune svake godine.

Činjenica da se jezik razvija znači da čitači moraju neprekidno da ostanu u koraku s njim, a ako koristite stariji čitač, on možda neće podržavati svaku mogućnost. Dizajneri jezika paze da ne naprave izmene koje bi zaustavile postojeće programe, pa novi čitači i dalje mogu da izvršavaju stare programe. U ovoj knjizi koristim verziju 2017 JavaScripta.

Veb čitači nisu jedine platforme na kojima se koristi JavaScript. Neke baze podataka, kao što su MongoDB i CouchDB, koriste JavaScript kao svoj jezik za izradu skriptova i upita. Nekoliko platformi za desktop i serversko programiranje, od kojih je najvažniji projekat Node.js (tema poglavlja 20), nude okruženje za programiranje na JavaScriptu van čitača.

## Kôd, i šta ćete s njim

*Kôd* je tekst koji čini programe. Većina poglavlja u ovoj knjizi sadrži podosta koda. Verujem da su čitanje koda i pisanje koda nezamenljivi delovi učenja programiranja. Pokušajte da ne prelazite ovlaš preko primera – pažljivo ih pročitajte i shvatite. To je možda sporo i zbunjujuće na početku, ali obećavam da ćete ga brzo savladati. Isto važi i za vežbe. Nemojte pretpostaviti da ih razumete dok ne napišete rešenje koje radi.

Preporučujem da isprobate svoja rešenja za vežbe u pravom JavaScript interpreteru. Na taj način ćete dobiti trenutnu povratnu informaciju o tome da li ono što radite funkcioniše i, nadam se, doći ćete u iskušenje da eksperimentišete i odete i dalje od vežbi.

Najjednostavniji način da pokrenete probni kod iz knjige, i da eksperimentišete s njim, jeste da ga pogledate u internet verziji knjige na lokaciji *https://eloquentjavascript.net* . Tamo možete da pritisnete bilo koji primer koda da biste ga izmenili i pokrenuli, i videli rezultat koji daje. Da biste radili na vežbama, idite na lokaciju *https://eloquentjavascript.net/code*, koja sadrži početni kod za svaku vežbu u programiranju i omogućava da pogledate rešenja.

Ako želite da pokrenete programe definisane u knjizi izvan veb lokacije knjige, moraćete da obratite pažnju. Mnogi programi su samostalni i trebalo bi da rade u bilo kom JavaScript okruženju. Međutim, kod u kasnijim poglavljima je često napisan za određeno okruženje (čitač veba ili Node.js) i može se pokrenuti samo tamo. Pored toga, mnoga poglavlja definišu veće programe, pa delovi koda koji se pojavljuju u njima zavise jedni od drugih ili od spoljnih datoteka. Izolovano okruženje (engl. *sandbox)* na veb lokaciji nudi hiperveze do Zip datoteka koje sadrže sve skriptove i datoteke s podacima potrebne da bi se izvršio kod za dato poglavlje.

## Pregled knjige

Ova knjiga ima tri dela. Prvih 12 poglavlja govore o jeziku JavaScript. Narednih sedam poglavlja govore o veb čitačima i načinu na koji se JavaScript koristi za njihovo programiranje. Najzad, dva poglavlja za temu imaju Node.js, još jedno okruženje za programiranje na JavaScriptu.

U celoj knjizi nalazi se pet *projektnih poglavlja*, koja opisuju veće primere programa da biste okusili pravo programiranje. Redom, radićemo na izgradnji robota za isporuku, programskom jeziku, platformeru, programu za bojenje piksela i dinamičkoj veb prezentaciji.

Jezički deo knjige počinje sa četiri poglavlja koja prikazuju osnovnu strukturu jezika JavaScript. Ona predstavljaju kontrolne strukture (kao što je reč `while` koju ste videli u ovom uvodu), funkcije (pisanje vaših gradivnih blokova) i strukture podataka. Nakon njih, moći ćete da pišete osnovne programe. Potom, u poglavljima 5 i 6, predstavljamo tehnike korišćenja funkcija i objekata za pisanje *apstraktnijeg* koda i za obuzdavanje složenosti.

Nakon prvog projektnog poglavlja, jezički deo knjige nastavlja se poglavljima o radu sa greškama i ispravljanju grešaka, regularnim izrazima (oni su bitan alat za rad sa tekstom), modularnosti (još jedna odbrana od složenosti) i asinhronom programiranju (radu sa događajima koji zahtevaju vreme). Drugo projektno poglavlje završava prvi deo knjige.

Drugi deo, poglavlja 13 do 19, opisuje alate kojima JavaScript u čitaču ima pristup. Naučićete kako da prikazujete stvari na ekranu (poglavlja 14 i 17), kako da reagujete na korisnički unos (poglavlje 15) i kako da komunicirate putem mreže (poglavlje 18). I u ovom delu postoje dva projektna poglavlja.

Nakon toga, poglavlje 20 opisuje Node.js, a poglavlje 21 pravi malu veb prezentaciju koristeći taj alat.

Na kraju, poglavlje 22 opisuje neke okolnosti koje se pojavljuju pri optimizovanju JavaScript programa da bi bili brži.

## Tipografska pravila

U ovoj knjizi, tekst napisan neproporcionalnim fontom predstavljaće elemente programa – to su ponekad kompletni delovi, a ponekad samo ukazuju na deo obližnjeg programa. Programi (a već ste ih videli nekoliko) napisani su na sledeći način:

```
function factorial(n) {
if (n == 0) {
return 1;
} else {
return factorial(n - 1) * n;
}
}
```

Ponekad, da biste videli rezultat koji program daje, očekivani rezultat biće napisan nakon programa, nakon dve kose crte i strelice.

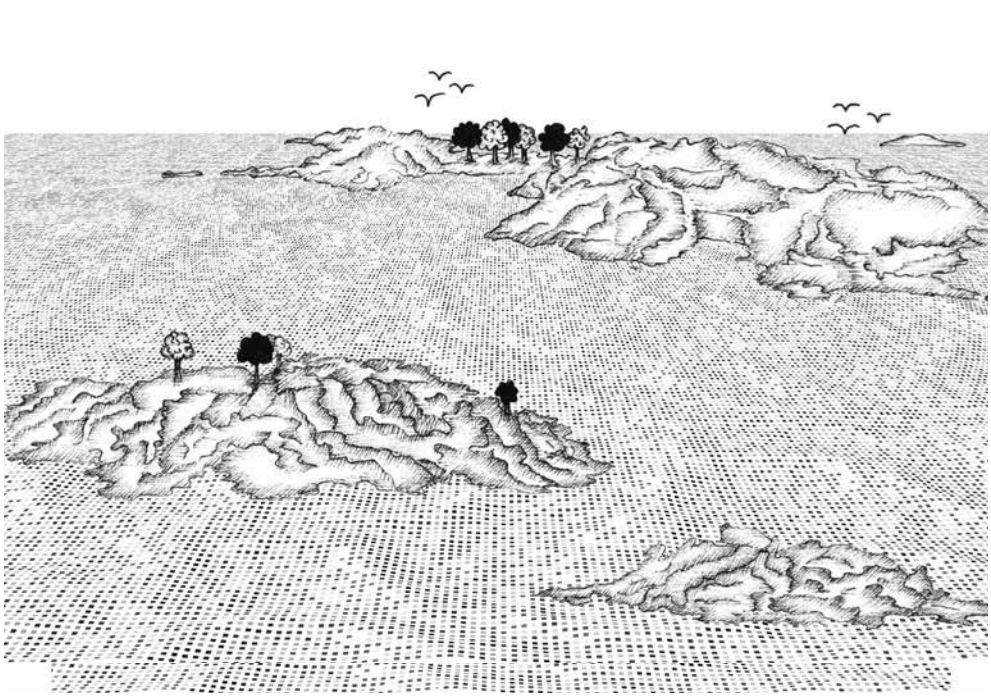
```
console.log(factorial(8));
// → 40320
```

Srećno!

# DEO I

## JEZIK

„Ispod površine mašine, kreće se program. Bez truda, on se širi i skuplja. U velikoj harmoniji, elektroni se rasipaju i ponovo grupišu. Oblici na monitoru samo su talasi na vodi. Suština ostaje ispod, nevidljiva.“
– Majstor Yuan-Ma, *The Book of Programming*



# 1

# VREDNOSTI, TIPOVI I OPERATORI

Unutar računarskog sveta suštinski postoje samo podaci. Čitate podatke, menjate ih, pravite nove podatke. Ono što nije podatak - nije vredno spomena. Svi ti podaci su uskladišteni kao dugačke sekvence bitova i zbog toga su, u osnovi, slični.

*Bitovi* (engl. *bits*) su bilo šta što ima dve vrednosti, obično opisane kao nule i jedinice. U računaru oni poprimaju oblik kao što je visok ili nizak električni naboj, jak ili slab signal ili sjajna ili zamućena tačka na površini CD-a. Bilo koji komad informacije može se svesti na sekvencu nula i jedinica i time predstaviti bitovima.

Na primer, broj 13 možemo izraziti u bitovima. To funkcioniše isto kao decimalni broj, ali umesto deset različitih cifara, imate samo dve, a „težina“ svake se povećava dva puta, zdesna ulevo. Evo bitova koji čine broj 13, sa težinom cifara prikazanom ispod njih:

```
  0   0   0   0  1  1  0  1
128  64  32  16  8  4  2  1
```

Dakle, to je binarni broj 00001101. Vrednost cifara koje nisu nule su 8, 4 i 1, i u zbiru daju 13.

## Vrednosti

Zamislite more bitova – ili čitav okean. Tipičan savremeni računar ima preko 30 milijardi bitova u svojoj radnoj verziji. Čvrsti disk obično ima nekoliko redova veličine više.

Da bismo mogli da radimo sa tolikom količinom bitova, a da se ne izgubimo, moramo ih podeliti u komade koji predstavljaju delove informacija. U okruženju JavaScripta, ti komadi se nazivaju vrednosti (engl. *values*). Iako su sve vrednosti sačinjene od bitova, one igraju različite uloge. Svaka vrednsot ima tip koji određuje njenu ulogu. Neke vrednosti su brojevi, neke su tekst, neke su funkcije itd.

Da biste napravili vrednost, samo treba da pozovete njeno ime. To je zgodno. Ne morate da prikupljate gradivni materijal za vrednosti niti da plaćate za njih. Samo je pozovete, i vuuššš, imate je. Naravno, one ne nastaju ni iz čega. Svaka vrednost mora negde da bude uskladištena i ako želite da koristite ogromnu količinu vrednosti istovremeno, moglo bi vam ponestati memorije. Srećom, to je problem samo ako vam sve one trebaju istovremeno. Čim više ne budete koristili vrednost, ona će iščeznuti i ostaviti za sobom svoje bitove da budu reciklirani kao gradivni materijal za narednu generaciju vrednosti.

Ovo poglavlje predstavlja atomske elemente JavaScript programa, tj. jednostavne tipove vrednosti i operatore koji mogu delovati na takve vrednosti.

## Brojevi

Vrednosti tipa *broj* (engl. *number*), nimalo iznenađujuće, jesu brojčane (numeričke) vrednosti. U JavaScript programu, one se zapisuju ovako:

```
13
```

Upotrebite to u programu i u memoriji računara će nastati obrazac bitova za broj 13.

JavaScript koristi fiksan broj bitova, 64, za skladištenje jedne brojčane vrednosti. Postoji određeni broj obrazaca koje možete napraviti pomoću 64 bita, a to znači da je broj različitih brojeva koje možete predstaviti ograničen. Sa $N$ decimalnih cifara, možete predstaviti $10^N$ brojeva. Slično tome, kada imate 64 binarne cifre, možete predstaviti $2^{64}$ različitih brojeva, što je otprilike 18 triliona (US engl. *quintillion*) (18 sa 18 nula). To je mnogo.

Računarska memorija je nekada bila mnogo manja i ljudi su obično koristili grupe od 8 ili 16 bitova da bi predstavljali brojeve. Bilo je lako slučajno *premašiti* tako male brojeve – završiti s brojem koji se nije uklapao u dati broj bitova. Danas čak i računari koji staju u vaš džep imaju obilje memorije, pa slobodno možete koristiti 64-bitne komade, a o prekoračivanju treba da brinete samo ako radite sa zaista astronomskim brojevima.

Ipak, ne staju svi celi brojevi manji od 18 triliona u JavaScriptov broj. Ti bitovi skladište i negativne brojeve, pa jedan bit označava znak broja. Još je veća stvar to što neceli brojevi takođe moraju biti predstavljeni. Da bi se

to uradilo, neki od bitova se koriste za skladištenje položaja decimalne tačke. Stvarni maksimalan ceo broj koji se može uskladištiti kreće se pre u rangu 9 bilijardi (15 nula) – što je i dalje ugodno ogromno.

Decimalni brojevi (u programskim jezicima) se pišu pomoću tačke.

```
9.81
```

Za veoma velike ili veoma male brojeve, možete koristiti naučnu notaciju tako što ćete dodati *e* (za *eksponent*), i nakon toga napisati eksponent broja.

```
2.998e8
```

To je 2.998 × 108 = 299.800.000.

Izračunavanja sa celim brojevima (engl. *integers*) manjim od gorepomenutih 9 bilijardi, garantovano će uvek biti precizna. Nažalost, izračunavanja sa decimalnim brojevima obično nisu. Kao što π (pi) ne može biti precizno izražen konačnim brojem decimalnih mesta, mnogi brojevi gube preciznost kada je samo 64 bita dostupno za njihovo skladištenje. To je šteta, ali izaziva praktične probleme samo u određenim situacijama. Bitna stvar je da budete svesni tog ograničenja i da decimalne brojeve tretirate kao približne vrednosti, a ne precizne.

### Aritmetika

Glavna stvar koju ćete raditi s brojevima jeste artimetika. Aritmetičke operacije kao što je sabiranje ili množenje uzimaju dve brojčane vrednosti i od njih proizvode nov broj. Evo kako to izgleda u JavaScriptu:

```
100 + 4 * 11
```

Znaci + i * su *operatori*. Prvi predstavlja sabiranje, a drugi množenje. Postavljanje operatora između dve vrednosti primeniće tu operaciju na vrednosti i proizvesti novu vrednost.

Ali da li ovaj primer znači „saberi 4 i 100, i rezultat pomnoži brojem 11“ ili se množenje obavlja pre sabiranja? Kao što ste i pretpostavljali, množenje se dešava prvo. Ipak, kao i u matematici, to možete promeniti obavijajući sabiranje zagradama.

```
(100 + 4) * 11
```

Za oduzimanje se koristi operator -, a deljenje se može obaviti operatorom /.

Kada se operatori javljaju zajedno bez zagrada, redosled kojim se primenjuju određen je *prioritetom* operatora. Prethodni primer je pokazao da množenje ide pre sabiranja. Operator / ima isti prioritet kao *. Isto važi i za + i -. Kada je više operatora sa istim prioritetom jedan uz drugi, na primer, `1 - 2 + 1`, oni se primenjuju sleva udesno: `(1 - 2) + 1`.

Ta pravila prioriteta nisu nešto o čemu treba da brinete. Kada ste u nedoumici, samo dodajte zagrade.

Postoji još jedan aritmetički operator koji možda nećete odmah prepoznati. Znak `%` se koristi za predstavljanje *ostatka*. `X % Y` jeste ostatak deljena broja `X` brojem `Y`. Na primer, `314 % 100` daje rezultat `14`, a `144 % 12` daje 0. Prioritet operatora za ostatak isti je kao za množenje i deljenje. Ovaj operator često se naziva i *modulo*.

### *Posebni brojevi*

U JavaScriptu postoje tri posebne vrednosti koje se smatraju brojevima, ali se ne ponašaju kao obični brojevi.

Prve dve su su `Infinity` i `-Infinity` i one predstavljaju pozitivnu i negativnu beskonačnost. `Infinity – 1` je i dalje `Infinity`, itd. Ipak, nemojte biti previše poverljivi prema izračunavanju zasnovanom na beskonačnosti. Ono nije matematički ispravno i brzo će dovesti do narednog posebnog broja: `NaN`.

`NaN` predstavlja „nije broj“, iako tip te vrednosti *jeste* broj. Taj rezultat ćete dobiti, na primer, kada pokušate da izračunate `0 / 0` (deljenje nule nulom), `Infinity – Infinity`, ili bilo koju od više numeričkih operacija koje ne daju smislen rezultat.

## Znakovni nizovi

Naredni osnovni tip podataka jeste *znakovni niz* (engl. *string*). Znakovni nizovi se koriste za predstavljanje teksta. Oni se pišu postavljanjem njihovog sadržaja između navodnika.

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

Možete koristiti polunavodnike, navodnike ili obrnute polunavodnike da biste označili znakovne nizove, samo je bitno da znak za navođenje na početku i na kraju znakovnog bude isti.

Gotovo sve se može postaviti između navodnika, a JavaScript će od toga napraviti vrednost znakovnog niza. Međutim, neki znaci su teži. Možete zamisliti koliko bi teško bilo postaviti navodnike između navodnika. Znak za *novi red* (engl. *newline*) koji dobijete kada pritisnete E N T E R, može se dodati bez korišćenja izlazne sekvence za znakove samo ako znakovni niz obuhvatite obrnutim polunavodnicima (`).

Da biste takve znakove uključili u znakovni niz, koristićete narednu notaciju: kad god se u tekstu nalazi obrnuta kosa crta (\), ona označava da znak nakon nje ima posebno značenje. To je *izlazna sekvenca* za znak. Ako pre navodnika postavite obrnutu kosu crtu, nećete završiti znakovni niz, već će navodnik biti njegov deo. Kada se nakon obrnute kose crte pojavi znak `n`, on se tumači kao nov red. Slično tome, `t` nakon obrnute kose crte predstavlja znak za tabulator. Pogledajte naredni znakovni niz:

```
"Ovo je prvi red\nA ovo je drugi"
```

Tekst koji on sadrži je ovaj:

```
Ovo je prvi red
A ovo je drugi
```

Naravno, postoje situacije kada ćete želeti da obrnuta kosa crta u znakovnom nizu bude samo obrnuta kosa crta, a ne poseban kod. Ukoliko dve obrnute kose crte slede jedna za drugom, one će se sažeti i ostaće samo jedna u rezultujućoj vrednosti znakovnog niza. Evo kako možete prikazati znakovni niz `"Znak za nov red zapisuje se kao "\n"."`:

```
"Znak za nov red zapisuje se kao \"\\n\"."
```

I znakovni nizovi moraju da budu oblikovani kao nizovi bitova da bi mogli da postoje u računaru. Način na koji JavaScript to radi zasnovan je na standardu *Unicode*. Taj standard dodeljuje broj svakom znaku koji bi vam ikada mogao zatrebati, uključujući i znakove iz grčkog jezika, arapskog, japanskog, jermenskog itd. Ukoliko imamo broj za svaki znak, znakovni niz može biti opisan kao sekvenca brojeva.

I to je ono što JavaScript radi. Ali tu postoji komplikacija: JavaScript za predstavljanje koristi 16 bitova po znakovnom elementu, pa može da opiše $2^{16}$ različitih znakova. Međutim, Unicode definiše više znakova od toga – trenutno je to oko dvaput više. Znači, neki znakovi, kao što su mnogi emotikioni, zauzimaju do dve „znakovne pozicije“ u JavaScriptovim znakovnim nizovima. Vratićemo se tome u odeljku „Znakovni nizovi i kodovi znakova“, na strani 87.

Znakovni nizovi se ne mogu deliti, množiti ili oduzimati, ali operator + *može* se koristiti na njima. On ne samo da ih sabira, već ih *nadovezuje* (engl. *concatenate*) – lepi dva znakovna niza jedan za drugi. Naredni red će proizvesti znakovni niz `"concatenate"`:

```
"con" + "cat" + "e" + "nate"
```

Postoji više funkcija (*metoda*) koje se mogu koristiti da bi se na vrednostima tipa znakovni niz izvele druge operacije. O njima ću više reći u odeljku „Metode“ na strani 60.

Znakovni nizovi napisani sa polunavodnicima ili navodnicima ponašaju se isto – razlika je samo u tome koji tip navodnika želite da upotrebite unutar samog znakovnog niza. Znakovni nizovi postavljeni u obrnute polunavodnike obično se nazivaju *šablonski literali* (engl. *template literals*) i mogu da urade još neke trikove. Osim što mogu da se prostiru u više redova, u njih možete ugraditi druge tipove vrednosti.

```
`pola od 100 je ${100 / 2}`
```

Kada u šablonskom literalu nešto napišete unutar `${}`, rezultat toga će biti izračunat, pretvoren u znakovni niz i dodat na to mesto. Gornji primer daje `pola od 100 je 50`.

## Unarni operatori

Nisu svi operatori simboli. Neki se pišu kao reči. Jedan primer je operator `typeof`, koji vraća tip vrednosti koju ste mu prosledili.

```
console.log(typeof 4.5)
// → number console.log(typeof "x")
// → string
```

U ovom primeru koda koristimo `console.log` da bismo naznačili da želimo da vidimo rezultat neke procene. Više o tome u narednom poglavlju.

Ostali prikazani operatori radili su sa dve vrednosti, ali `typeof` uzima samo jednu. Operatori koji koriste dve vrednosti nazivaju se *binarni* operatori, dok se oni koji uzimaju jednu nazivaju *unarni* operatori. Operator minus može se koristiti i kao binaran i kao unaran operator.

```
console.log(- (10 – 2))
// → -8
```

## Bulove vrednosti

Često je korisno imati tip vrednosti koji pravi razliku između dve mogućnosti, kao što su „da“ i „ne“ ili „uključeno“ i „isključeno“. Za tu namenu, JavaScript ima tip *Boolean* (Bulove, logičke vrednosti), koji ima samo dve vrednosti, tačno – *true* i netačno – *false*.

### *Poređenje*

Evo jednog načina da se proizvedu Bulove vrednosti:

```
console.log(3 > 2)
// → true console.log(3 < 2)
// → false
```

Znakovi `>` i `<` su tradicionalni simboli za „veće od“ i „manje od“. Oni su binarni operatori. Njihova primena za rezultat ima Bulovu vrednost koja označava da li je izraz tačan u datom slučaju.

I znakovni nizovi se mogu porediti na isti način.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

Način na koji su znakovni nizovi poređani otprilike je alfabetski, ali nije baš onakav kakav biste videli u rečniku: velika slova su uvek „manja“ nego mala slova, pa je `"Z" < "a"`, a nealfabetski znakovi (!, – itd) su takođe uključeni u redosled. Kada poredi znakovne nizove, JavaScript prolazi kroz znakove sleva udesno, poredeći Unicode kodove jedan po jedan.

Drugi slični znakovi su `>=` (veće ili jednako), `<=` (manje ili jednako), `==` (jednako) i `!=` (različito).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

Postoji samo jedna vrednost u JavaScriptu koja nije jednaka sama sebi, a to je `NaN` („nije broj“ – engl. *„not a number“*).

```
console.log(NaN == NaN)
// → false
```

`NaN` bi trebalo da označava da nema rezultata u računanju koje nema smisla i taj rezultat, nije jednak ni jednom *drugom* rezultatu bilo kog izračunavanja koje nema smisla.

### Logički operatori

Postoje neke operacije koje se mogu primenjivati na same Bulove vrednosti. JavaScript podržava tri logička operatora: *i*, *ili* i *ne*. Oni se mogu koristiti za „rasuđivanje“ o Bulovim vrednostima.

Operator `&&` predstavlja logičko *i*. To je binarni operator i njegov rezultat je *true* (tačno) samo ako su obe date vrednosti *true*.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

Operator `||` označava logičko *ili*. On daje rezultat *true* ako je bilo je koja dodeljena vrednost *true*.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

*Ne* se piše kao znak uzvika (`!`). To je unarni operator koji obrće vrednost koja mu je data — `!true` daje `false`, a `!false` daje `true`.

Pri mešanju Bulovih operatora sa aritimetičkim i drugim operatorima, nije uvek očigledno kada su vam potrebne zagrade. U praksi je obično dovoljno znati da, od operatora koje ste dosad videli, `||` ima najniži prioritet, sledi `&&`, pa operatori poređenja (`>`, `==` itd) i onda ostali. Taj redosled je izabran tako da vam je, u tipičnim izrazima kao što je naredni, potrebno što je moguće manje zagrada:

```
1 + 1 == 2 && 10 * 10 > 50
```

Poslednji logički operator o kojem ću govoriti nije unarni, ni binarni, već je *ternarni*, i radi sa tri vrednosti. Piše se sa znakom pitanja i dvotačkom, ovako:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Naziva se *uslovni* (engl. *conditional*) operator (ili prosto *ternarni* operator, jer je jedini takav operator u jeziku). Vrednost levo od upitnika „bira“ koja će od druge dve vrednosti biti rezultat. Kada je vrednost levo od upitnika *true*, ona za rezultat bira srednju vrednost, a kada je *false*, rezultat je desna vrednost.

## Prazne vrednosti

Postoje dve posebne vrednosti, zapisuju se kao `null` i `undefined`, koje se koriste za označavanje nedostatka *smislene* vrednosti. One same za sebe jesu vrednosti, ali ne sadrže nikakve informacije.

Mnoge operacije u jeziku koje ne proizvode smislenu vrednost (kasnije ćete videti neke) za rezultat imaju `undefined` prosto zato što moraju da daju *neku* vrednost.

Razlika u značenju između `undefined` i `null` je slučajnost JavaScriptovog dizajna, i u većini slučajeva nije bitna. U situacijama kada treba da se bavite tim vrednostima, preporučujem da ih tretirate kao vrednosti koje uglavnom mogu zamenjivati jedna drugu.

## Automatska konverzija tipa

U uvodu sam pomenuo da JavaScript daje sve od sebe da bi prihvatio gotovo svaki program koji mu date, čak i programe koji rade čudne stvari. To je fino prikazano u narednim izrazima:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

Kada je operator primenjen na „pogrešan“ tip vrednosti, JavaScript će tiho pretvoriti tu vrednost u tip koji mu treba, koristeći skup pravila koja često nisu ono što biste očekivali. To se naziva *konverzija tipa* (engl. *type coercion*). Vrednost `null` u prvom izrazu postaje `0`, a `"5"` u drugom izrazu postaje `5` (umesto znakovnog niza postaje broj). U trećem izrazu, `+` pokušava da nadoveže znakovne nizove pre nego da sabere brojeve, pa je `1` pretvoreno `"1"` (umesto broja postaje znakovni niz).

Kada se u broj pretvara nešto što ne odgovara broju na očigledan način (kao što je `"five"` ili `undefined`), dobićete vrednost `NaN`. Dalje aritmetičke operacije nad vrednošću `NaN` i dalje za rezultat imaju `NaN`, pa ako se nađete u nekoj od tih neočekivanih situacija, potražite nenamerne konverzije tipova.

Kada poredite vrednosti istog tipa koristeći `==`, rezultat je lako predvideti: treba da dobijete *true* kada su obe vrednosti iste, osim u slučaju `NaN`. Međutim, kada se tipovi razlikuju, JavaScript koristi složen i zbunjujuć skup pravila da bi utvrdio šta da radi. U većini slučajeva, prosto pokušava da konvertuje jednu od vrednosti u tip druge vrednosti. Međutim, kada se `null` ili `undefined` javljaju na bilo kojoj strani operatora, rezultat će biti *true* samo ako je i na drugoj strani operatora vrednost `null` ili `undefined`.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

Takvo ponašanje je često korisno. Kada hoćete da testirate da li je neka vrednost prava vrednost, a ne `null` ili `undefined`, možete da je uporedite sa `null` koristeći operator `==` (ili `!=`).

Ali šta ako hoćete da proverite da li se nešto odnosi baš na vrednost `false`? Izrazi kao što su `0 == false` i `"" == false` su *true*. Kada *ne* želite da dođe do automatske konverzije tipa, postoje dva dodatna operatora: `===` i `!==`. Prvi proverava da li je vrednost *precizno* jednaka drugoj, a drugi proverava da li nije precizno jednaka. Tako `"" === false` ima rezultat *false* kao što se i očekuje.

Preporučujem da troznačne operatore za poređenje koristite u odbrambene svrhe, da biste sprečili neočekivane konverzije tipa da vas sapletu. Međutim, kada ste sigurni da su tipovi sa obe strane isti, nema problema s korišćenjem kraćih operatora.

### *Skraćeno izračunavanje logičkih operatora*

Logički operatori `&&` i `||` na čudan način izlaze na kraj s vrednostima različitog tipa. Pretvoriće vrednost sa njihove leve strane u Bulov tip da bi odlučili šta da rade, ali zavisno od operatora i rezultata te konverzije, oni će vratiti ili *originalnu* levu vrednost ili desnu vrednost.

Operator `||`, na primer, vraća vrednost koja mu stoji na levoj strani kada se ona može pretvoriti u *true*, a inače vraća vrednost na desnoj strani. To ima očekivano dejstvo kada su vrednosti Bulove, a radi nešto analogno za vrednosti drugih tipova.

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

Tu funkcionalnost možemo koristiti kao način da se vratimo na unapred zadatu vrednost. Ako imate vrednost koja bi mogla biti prazna, možete nakon nje postaviti `||` sa zamenskom vrednošću. Ukoliko početna vrednost može biti konvertovana u *false*, umesto nje ćete dobiti zamensku vrednost. Pravila

za konvertovanje znakovnih nizova i brojeva u Bulove vrednosti kažu da se `0`, `NaN` i prazan znakovni niz (`""`) računaju kao `false`, a sve ostale vrednosti se računaju kao `true`. Znači `0 || -1` daje rezultat `-1`, a `"" || "!?"` daje rezultat `"!?"`.

Operator `&&` radi slično, ali obrnuto. Kada je vrednost na njegovoj levoj strani nešto što se pretvara u *false*, on vraća tu vrednost, a inače vraća vrednost koja mu stoji desno.

Drugo bitno svojstvo ova dva operatora jeste da se deo koji im stoji desno procenjuje samo kada je to neophodno. U slučaju `true || X`, ma šta da je `X` – čak i ako je to deo programa koji radi nešto *užasno* – rezultat će biti *true*, i `X` neće biti procenjivano. Isto važi i za `false && X`, što je *false* i zanemaruje `X`. To se naziva *skraćeno izračunavanje* (engl. *short-circuit evaluation*).

Uslovni operator funkcioniše na sličan način. Između druge i treće vrednosti, procenjuje se samo ona koja je izabrana.

## Rezime

U ovom poglavlju smo razmotrili četiri tipa JavaScriptovih vrednosti: brojeve, znakovne nizove, Bulove vrednosti i nedefinisane vrednosti.

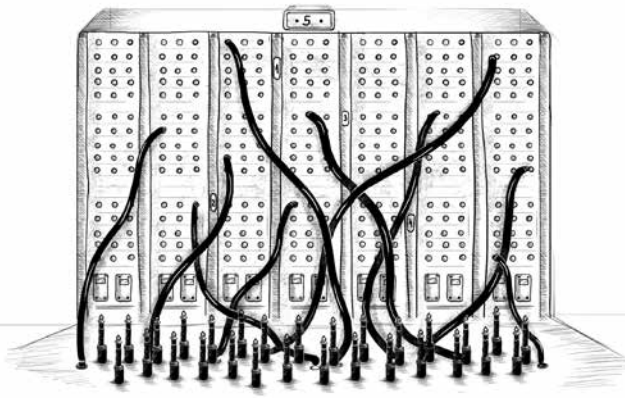
Takve vrednosti se prave upisivanjem njihovog imena (`true`, `null`) ili vrednosti (`13`, `"abc"`). Možete kombinovati i transformisati vrednosti pomoću operatora. Videli smo binarne operatore za aritmetiku (`+`, `-`, `*`, `/` i `%`), operatore za nadovezivanje znakovnih nizova (`+`), poređenje (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) i logičke operatore (`&&`, `||`), kao i nekoliko unarnih operatora (`-` za negaciju broja, `!` za logičku negaciju i `typeof` za pronalaženje tipa vrednosti) i jedan ternarni operator (`?:`) za biranje jedne od dve vrednosti na osnovu treće vrednosti. Time ste dobili dovoljno informacija da biste JavaScript koristili kao džepni kalkulator, ali ne i za nešto više od toga. Naredno poglavlje počinje da povezuje ove izraze u osnovne programe.

# DEO II

## ČITAČ VEBA

„San u osnovi veba jeste san o zajedničkom informacionom prostoru u kojem komuniciramo razmenjivanjem informacija. Njegova univerzalnost je neophodna: činjenica da hiperveza može da pokaže bilo šta, bilo to lično, lokalno ili globalno, bilo to samo nacrt ili savršeno uglancano."

– Tim Berners-Lee,
*The World Wide Web: A very short personal history*



# 13

# JAVASCRIPT I ČITAČ VEBA

Naredna poglavlja ove knjige govoriće o čitačima veba. Bez njih ne bi bilo ni JavaScripta. A čak i kada bi ga bilo, niko na njega ne bi obraćao pažnju.

Veb tehnologija je bila decentralizovana od samog početka, ne samo tehnički već i po načinu na koji se razvijala. Razni proizvođači čitača dodavali su nove funkcionalnosti, *ad hoc* i ponekad slabo promišljeno, što su onda, ponekad, prihvatali ostali – i na kraju je bilo zapisano kao standard.

To je istovremeno i blagoslov i prokletstvo. S jedne strane, podsticajno je da ne postoji jedna glavna strana koja upravlja sistemom, već ga unapređuju razne strane koje rade u labavoj saradnji (ili, povremeno, u otvorenom neprijateljstvu). S druge strane, opasan način na koji se veb razvio znači da rezultujući sistem nije baš najsjajniji primer unutrašnje doslednosti. Neki njegovi delovi su dozlaboga zbunjujući i loše zamišljeni.

## Mreže i internet

Računarske mreže postoje otprilike od pedesetih godina prošlog veka. Ako postavite kablove između dva ili više računara, i dozvolite im da jedni drugima šalju podatke preko tih kablova, možete raditi razne vrste divnih stvari.

A ako povezivanje dve mašine u istoj zgradi omogućava da radimo divne stvari, povezivanje mašina širom planete trebalo bi da bude još bolje.

Tehnologija za početak implementiranja ove vizije razvijena je osamdesetih godina prošlog veka, a dobijena mreža nazvana je *internet*. Ispunila je očekivanja.

Računar može da koristi tu mrežu da bi ispaljivao bitove ka drugom računaru. Da bi iz tog ispaljivanja bitova izronila ikakva efikasna komunikacija, računari na oba kraja moraju znati šta ti bitovi predstavljaju. Značenje bilo koje date sekvence bitova u potpunosti zavisi od vrste stvari koju ona pokušava da izrazi i od upotrebljenog mehanizma kodiranja.

*Mrežni protokol* (engl. *network protocol*) opisuje stil komunikacije na mreži. Postoje protokoli za slanje elektronske pošte, za pozivanje e-pošte, za razmenjivanje datoteka, čak i za upravljanje računarima koji su inficirani zloćudnim softverom.

Na primer, *Hypertext Transfer Protocol* (HTTP) protokol je za preuzimanje imenovanih resursa ili izvora (komada informacija, kao što su veb strane ili slike). On zadaje da strana koja šalje zahtev treba da počne ovakvim redom, imenujući izvor i verziju protokola koji pokušava da koristi:

```
GET /index.html HTTP/1.1
```

Postoji još mnogo pravila u vezi sa načinom na koji zahtevalac može da uključi još informacija u zahtev i načinom na koji druga strana, koja vraća izvor, pakuje svoj sadržaj. HTTP ćemo detaljnije razmatrati u poglavlju 18.

Većina protokola je izgrađena povrh drugih protokola. HTTP tretira mrežu kao uređaj nalik potoku u koji možete staviti bitove tako da oni stignu na ispravno odredište ispravnim redosledom. Kao što smo videli u poglavlju 11, i samo obezbeđivanje toga je krajnje težak problem.

*Protokol za upravljanje prenosom* (engl. *Transmission Control Protocol*, *TCP*) protokol je koji se bavi tim problemom. „Govore“ ga svi uređaji povezani u internet i većina komunikacije na internetu izgrađena je na njemu.

TCP konekcija radi na sledeći način: jedan računar mora da čeka, ili *osluškuje*, dok drugi računari ne počnu da govore s njim. Da bi na jednoj mašini mogao da osluškuje različite vrste komunikacije u isto vreme, svakom osluškivaču dodeljen je broj (koji se naziva *port*). Većina protokola unapred zadaje port koji treba da se koristi. Na primer, kada želimo da pošaljemo e-poruku koristeći protokol SMTP, mašina kroz koju je šaljemo treba da osluškuje na portu 25.

Drugi računar tada može da uspostavi konekciju sa ciljnom mašinom koristeći odgovarajući broj porta. Ukoliko se može doći do ciljne mašine i ona osluškuje taj port, konekcija je uspešno napravljena. Računar osluškivač naziva se *server*, a računar koji se povezuje sa njim naziva se *klijent* (engl. *client*).

Takva konekcija funkcioniše kao dvosmerna cev kroz koju mogu da teku bitovi – mašine na oba kraja mogu da ubacuju podatke u nju. Kada se bitovi uspešno prenesu, mašina na drugom kraju može da ih pročita. To je zgodan model. Moglo bi se reći da TCP omogućava apstrahovanje mreže.

## Veb

Globalna računarska mreža, *World Wide Web* (koji ne treba mešati sa internetom kao celinom) skup je protokola i formata koji nam omogućavaju da posećujemo veb strane preko čitača veba (engl. *browser*). „Mreža“ (engl. *web*) u imenu odnosi se na činjenicu da se takve strane mogu lako povezati jedna s drugom i tako napraviti ogromnu mrežu kroz koju korisnici mogu da se kreću.

Da biste postali deo veba, potrebno je samo da povežete mašinu sa internetom i kažete joj da osluškuje na portu 80 sa HTTP protokolom, tako da drugi računari mogu od nje da traže dokumenta.

Svaki dokument na vebu imenovan je *jedinstvenim identifikatorom resursa* (engl. *Uniform Resource Locator, URL*), koji izgleda otprilike ovako:

```
http://eloquentjavascript.net/13_browser.html
|      |                      |              |
protokol       server             putanja
```

Prvi deo nam govori da ovaj URL koristi HTTP protokol (a ne, na primer, šifrovani, bezbedni HTTP, koji bi bio *https://* ). Sledi deo koji označava od kog servera tražimo dokument. Poslednji je niz sa putanjom koji označava određeni dokument (ili *resurs*) za koji smo zainteresovani.

Mašine povezane na internet dobijaju *IP adresu*, a to je broj koji se može koristiti za slanje poruka toj mašini, i ona izgleda ovako: `149.210.142.219`, ili ovako: `2001:4860:4860::8888`. Međutim, liste manje-više nasumičnih brojeva teško se pamte i nezgodne su za upisivanje, pa umesto njih možete registrovati ime domena za određenu adresu ili skup adresa. Ja sam registrovao ime domena *eloquentjavascript.net* koja pokazuje IP adresu mašine kojom ja upravljam i to ime mogu da koristim da bih isporučivao veb strane.

Ako taj URL upišete u adresno polje svog čitača veba, čitač će pokušati da pozove i prikaže dokument na tom URL-u. Vaš čitač prvo mora da otkrije na koju se adresu odnosi *eloquentjavascript.net*. Potom, koristeći HTTP protokol, on se povezuje sa serverom na toj adresi i traži resurs */13_browser.html*. Ako sve prođe kako treba, server šalje dokument koji vaš čitač prikazuje na ekranu.

## HTML

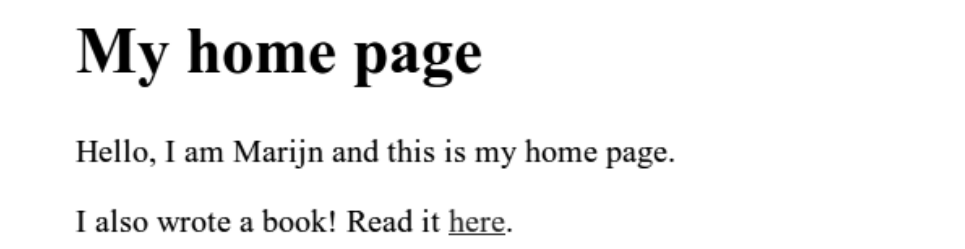
HTML, što je skraćenica za *Hypertext Markup Language* – jezik za označavanje hiperteksta, format je dokumenata koji se koristi za veb strane. HTML dokument sadrži tekst, kao i *oznake* (engl. *tags*) koje daju strukturu tekstu opisujući stvari kao što su hiperveze, pasusi i naslovi.

Kratak HTML dokument mogao bi izgledati ovako:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
```

```
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

Takav dokument bi u čitaču izgledao ovako:




Oznake, postavljene u uglaste zagrade (< i >, znake za *manje od* i *veće od*), pružaju informacije o strukturi dokumenta. Preostali tekst je običan tekst.

Dokument počinje sa `<!doctype html>`, što čitaču govori da stranu tumači kao *savremeni* HTML, za razliku od raznih dijalekata koji su bili u upotrebi u prošlosti.

HTML dokumenti imaju zaglavlje i telo. Zaglavlje sadrži informacije *o* dokumentu, a telo sadrži sâm dokument. U ovom slučaju, zaglavlje deklariše da je naslov ovog dokumenta „My home page“ i da on koristi kodiranje UTF-8, što je način da se Unicode tekst kodira kao binarni podaci. Telo dokumenta sadrži naslov (`<h1>`, od „heading 1“, tj. „naslov 1“; `<h2>` do `<h6>` daju podnaslove) i dva pasusa (`<p>`).

Oznake mogu imati nekoliko oblika. Element (kao što je telo, pasus ili hiperveza) počinje *početnom oznakom* (engl. *opening tag*) kao što je `<p>`, a završava se *završnom oznakom* (engl. *closing tag*) kao što je `</p>`. Neke početne oznake, kao što je oznaka za hipervezu (`<a>`), sadrže dodatne informacije u vidu parova `ime="vrednost"`. One se nazivaju *atributi*. U ovom slučaju, odredište hiperveze je naznačeno sa `href="http://eloquentjavascript.net"`, gde `href` predstavlja „hypertext reference“ – referencu hiperteksta.

Neke vrste oznaka ne obuhvataju ništa drugo, pa ne moraju da imaju završni deo. Oznaka za metapodatke, `<meta charset="utf-8">` primer je takve oznake.

Pošto uglaste zagrade u u HTML-u imaju posebno značenje, da bi mogle da se koriste u tekstu, mora se uvesti još jedan oblik specijalne notacije. Obična otvorena uglasta zagrada („manje od“) piše se kao `&lt;`, a zatvorena uglasta zagrada („veće od“) piše se kao `&gt;`. U HTML-u, znak ampersend (&) nakon kog sledi ime ili kôd znaka i tačka i zarez (;) naziva se *entitet* i biće zamenjen znakom koji kodira.

To je analogno načinu na koji se obrnute kose crte koriste u JavaScriptovim znakovnim nizovima. Pošto ovaj mehanizam i znaku ampersend daje specijalno značenje, taj znak u tekstu morate pisati kao `&amp;`. U vrednostima atributa, koje su postavljene u navodnike, `&quot;` se može koristiti da bi se ubacio sâm navodnik.

Raščlanjivanje HTML-a ima veliku toleranciju za greške. Kada nedostaju oznake koje bi trebalo da postoje, čitač veba ih rekonstruiše. Način na koji se to radi je standardizovan, i možete se uzdati da će svi savremeni čitači to činiti na isti način.

Naredni dokument će biti tretiran potpuno isto kao i onaj prethodni:

```
<!doctype html>
<meta charset=utf-8>

<title>My home page</title>
<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentjavascript.net>here</a>.
```

Oznake `<html>`, `<head>` i `<body>` potpuno su nestale. Čitač zna da `<meta>` i `<title>` pripadaju zaglavlju, a da `<h1>` znači da je počelo telo. Štaviše, nisam izričito zatvorio pasuse, pošto će ih otvaranje novog pasusa ili završavanje dokumenta implicitno zatvoriti. Navodnici oko vrednosti atributa takođe su nestali.

U ovoj knjizi će iz primera obično biti izostavljene oznake `<html>`, `<head>` i `<body>` da bi bili kraći i manje zakrčeni. Ali zatvaraću oznake i postavljati navodnike oko atributa.

Pored toga, obično ću izostavljati deklaraciju `doctype` i `charset`. Time vas ne ohrabrujem da ih izostavljate iz HTML dokumenata. Čitači često rade smešne stvari kada ih zaboravite. Smatrajte da `doctype` i `charset` metapodaci uvek postoje u primerima, čak i kada se ne vide u tekstu.

## HTML i JavaScript

U kontekstu ove knjige, najbitnija HTML oznaka je `<script>`. Ona nam omogućava da u dokument uključimo JavaScript.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

Takav skript će biti izvršen čim čitač, pri čitanju HTML-a, naiđe na njegovu oznaku `<script>`. Kada se ova strana otvori, na njoj će iskočiti okvir za dijalog – funkcija `alert` podseća na `prompt` po tome što otvara mali prozor, ali ona samo prikazuje poruku, ne tražeći unos.

Postavljanje velikih programa direktno u HTML dokument često je nepraktično. Oznaka `<script>` može da dobije atribut `src` da bi uzela datoteku skripta (tekstualnu datoteku koja sadrži JavaScript program) sa nekog URL-a.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

Datoteka *code/hello.js* koja je ovde dodata sadrži isti program – `alert("hello!")`. Kada HTML strana referencira druge URL-ove kao deo te strane – na primer, datoteku slike ili skript – čitači veba će ih odmah pozvati i uključiti ih u stranu.

Oznaka za skript uvek mora biti završena sa `</script>`, čak i ako se odnosi na datoteku skripta i ne sadrži nikakav kôd. Ako to zaboravite, ostatak strane će biti protumačen kao deo skripta.

U čitač možete učitati ES module (pročitajte „ECMAScript moduli“ na strani 167) tako što ćete oznaci za skript dati atribut `type="module"`. Takvi moduli mogu zavisiti od drugih modula tako što će u deklaracijama `import`, kao imena modula koristiti URL-ove koji su relativni u donosu na njih.

Neki atributi mogu da sadrže i JavaScript program. Oznaka `<button>`, prikazana u nastavku (koja prikazuje dugme) ima atribut `onclick`. Vrednost tog atributa biće pokrenuta svaki put kada se dugme pritisne.

```
<button onclick="alert('Boom!');">NE PRITISKAJ</button>
```

Primetićete da sam upotrebio polunavodnike za znakovni niz u atributu `onclick` jer se navodnici već koriste za navođenje celog atributa. Mogao sam da upotrebim i `&quot;`.

## U izolovanom okruženju

Pokretanje programa preuzetih sa interneta može biti opasno. Ne znate mnogo o ljudima koji stoje iza većine lokacija koje posećujete i ne mora značiti da vam svi žele dobro. Pokretanje programa ljudi koji vam ne žele dobro način je da svoj računar zarazite virusima, da vam podaci budu ukradeni i nalozi hakovani.

Pa ipak, privlačnost veba je u tome što možete da ga pregledate ne morajući da verujete svim stranama koje posetite. Zbog toga čitači oštro ograničavaju stvari koje JavaScript program može da uradi: on ne može da pregleda datoteke na vašem računaru niti da menja išta što nije povezano sa veb stranom u koju je ugrađen.

Ovakvo izolovanje programskog okruženja je poput ostavljanja programa da se „igra“ u kutiji s peskom (engl. *sandbox*). Međutim, ovu konkretnu vrstu kutije s peskom treba da zamislite kao da je pokrivena kavezom od debelih gvozdenih šipki, tako da programi koji se u njoj igraju ne mogu da izađu.

Teži deo postavljanja u izolaciju jeste obezbeđivanje programima dovoljno prostora da budu korisni, a u isto vreme sprečiti ih da urade išta opasno. Mnogo korisnih funkcionalnosti, kao što je komunikacija s drugim serverima ili čitanje sadržaja sa klipborda, takođe se može upotrebiti za problematične stvari koje narušavaju privatnost.

Svako malo, neko smisli nov način da zaobiđe ograničenja čitača i uradi nešto štetno, od otkrivanja nebitne lične informacije, do preuzimanja cele mašine na kojoj čitač radi. Programeri čitača reaguju popravljajući rupu i sve je opet kako treba – sve dok se ne otkrije sledeći problem i, nadamo se, objavi javnosti, umesto da ga u tajnosti eksploatiše neka vladina agencija ili mafija.

## Kompatibilnost i ratovi čitača

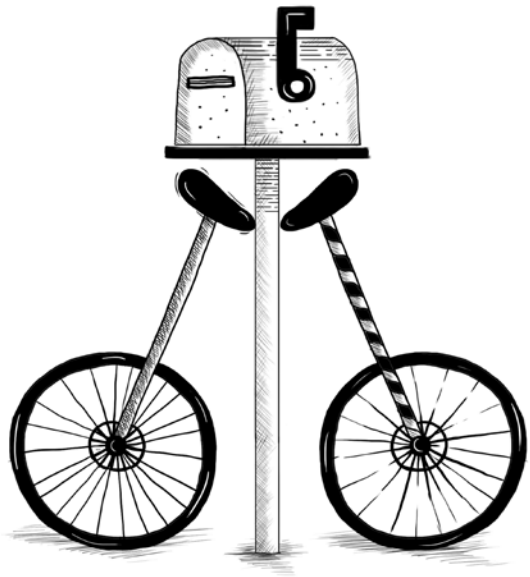
U ranim fazama veba, čitač Mosaic dominirao je tržištem. Nakon nekoliko godina, prevagu je odneo Netscape, kojeg je potom, zauzvrat, uveliko premašio Microsoftov Internet Explorer. Kad god je samo jedan čitač bio dominantan, njegov proizvođač uzimao je za pravo da jednostrano izmišlja nove mogućnosti za veb. Pošto je većina korisnika koristila najpopularniji čitač, veb lokacije bi prosto počele da koriste te mogućnosti – ko mari za ostale čitače.

To je bilo mračno doba kompatibilnosti, često nazivano *ratovi čitača veba*. Veb programeri nisu dobili jedan ujedinjeni veb, već dve ili tri nekompatibilne platforme. Da bi stvari bile još gore, čitači koji su se koristili oko 2003. bili su puni programskih grešaka, a, naravno, greške su se razlikovale za svaki čitač. Život nije bio lak za ljude koji su čekali veb strane.

Mozilla Firefox, neprofitni izdanak Netscapea, ugrozio je položaj Internet Explorera krajem prve decenije ovog veka. Pošto Microsoft u to vreme nije bio naročito zainteresovan da ostane konkurentan, Firefox je mu je oduzeo veliki deo tržišta. Otprilike u isto vreme, Google je predstavio svoj čitač Chrome, a Appleov čitač Safari stekao je popularnost, dovevši do situacije u kojoj su postojala četiri velika igrača, umesto jednog.

Novi igrači su imali ozbiljnije stavove o standardima i bolje prakse u projektovanju, dajući nam manje nekompatibilnosti i manje programskih grešaka. Microsoft, videvši da se njegov udeo u tržištu smanjuje, dozvao se pameti i prihvatio te stavove u svom čitaču Edge, koji zamenjuje Internet Explorer. Ako danas počinjete da učite veb programiranje, smatrajte se srećnim. Najnovije verzije glavnih čitača ponašaju se prilično ujednačeno i imaju relativno malo grešaka.

„Ako imate znanje, podelite ga sa drugima.“
– Margaret Fuller



# 21

# PROJEKAT: VEB LOKACIJA ZA RAZMENU VEŠTINA

Razmena veština (engl. *skill-sharing meeting*) događaj je na kojem se okupljaju ljudi sa zajedničkim interesovanjima i daju male, neformalne prezentacije o stvarima koje znaju. Na baštovanskoj razmeni veština, neko bi mogao objasniti kako se gaji celer. A vi biste mogli svratiti u programersku grupu za razmenu veština i ispričati ljudima o Node.js-u.

Takva okupljanja, koja se često nazivaju i *korisničke grupe* (engl. *users group*) kada se bave računarima – odličan su način da se prošire horizonti, upozna sa novim dostignućima, ili da se prosto upoznaju ljudi sa sličnim interesovanjima. Mnogi veći gradovi imaju JavaScript okupljanja. Ona su obično besplatna, a ja sam, posetivši neka, otkrio da su ljudi na njima druželjubivi i otvoreni.

U poslednjem projektnom poglavlju, naš cilj je da napravimo veb lokaciju za upravljanje predavanjima održanim tokom sastanka za razmenu veština. Zamislite malu grupu ljudi koji se redovno sastaju u kancelariji nekog od članova da bi razgovarali o vožnji monocikla. Prethodni organizator sastanaka se preselio u drugi grad i niko se nije javio da preuzme njegovo mesto. Želimo sistem koji će omogućiti učesnicima da daju predloge za predavanja i međusobno raspravljaju o predavanjima, bez glavnog organizatora.

Ceo kôd za projekat možete preuzeti sa veb lokacije za ovu knjigu, *https://eloquentjavascript.net/code/skillsharing.zip*.

## Dizajn

Ovaj projekat ima *serverski* (engl. *server*) deo, napisan za Node.js, i *klijentski* (engl. *client*) deo, napisan za čitač veba. Server čuva sistemske podatke i obezbeđuje ih za klijenta. On isporučuje i datoteke koje implementiraju sistem na klijentskoj strani.

Server čuva spisak predavanja predloženih za naredni sastanak, a klijent prikazuje tu listu. Svako predavanje ima ime predavača, naslov, sažetak i niz komentara povezanih s njim. Klijent omogućava korisnicima da predlažu nova predavanja (dodaje ih listi), da brišu predavanja i komentarišu postojeća. Kad god korisnik napravi takvu izmenu, klijent pravi HTTP zahtev kojim obaveštava server o tome.


Aplikacija će biti podešena tako da daje *živi* prikaz tekućih predloženih predavanja i njihove komentare. Kad god neko, negde, pošalje novo predavanje ili doda komentar, svi ljudi kojima je strana otvorena u čitaču trebalo bi odmah da vide izmenu. To predstavlja mali izazov – ne postoji način da veb server otvori konekciju sa klijentom, niti postoji dobar način da se zna koji klijenti trenutno posmatraju datu veb lokaciju.

Uobičajeno rešenje za taj problem naziva se *dugačko anketiranje* (engl. *long polling*), i ono je jedan od razloga za Nodeov dizajn.

## Dugačko anketiranje

Da bismo mogli odmah da obavestimo klijenta da se nešto promenilo, potrebna nam je veza s tim klijentom. Pošto čitači veba tradicionalno ne prihvataju konekcije, a klijenti su često iza rutera koji bi svejedno blokirali takve konekcije, nije praktično podesiti da server inicira tu konekciju.

Možemo urediti da klijent otvori konekciju i održava je tako da je server može koristiti za slanje informacija kada je to potrebno.

Međutim, HTTP zahtev dozvoljava samo jednostavan tok informacija: klijent šalje zahtev, server vraća jedan odgovor i to je to. Postoji tehnologija nazvana *WebSockets*, podržana u savremenim čitačima, koja omogućava da se otvore konekcije za proizvoljnu razmenu podataka. Međutim, nije jednostavno ispravno je koristiti.

U ovom poglavlju ćemo koristiti jednostavniju tehniku – dugačko anketiranje – u kojoj klijenti neprekidno traže od servera nove informacije koristeći obične HTTP servere, a server odugovlači sa svojim odgovorom kada nema ništa da prijavi. Sve dok se klijent brine da neprekidno ima otvoren zahtev anketiranja, on će primati informacije od servera ubrzo nakon što postanu dostupne. Na primer, ako Fatma u svom čitaču ima otvorenu aplikaciju za razmenu veština, taj čitač će postaviti zahtev za ažurne informacije i čekaće odgovor na taj zahtev. Kada Iman pošalje predavanje o ekstremnom spustu monociklom, server će primetiti da Fatma čeka ažurne informacije i poslaće odgovor koji sadrži novo predavanje za njen zahtev na čekanju. Fatmin čitač će primiti podatke i ažurirati ekran da prikaže predavanje.

Da bi se sprečilo isticanje konekcije (njeno prekidanje zbog neaktivnosti), tehnika dugačkog anketiranja obično zadaje maksimalno vreme za svaki zahtev, nakon kog će server svejedno odgovoriti, čak i ako nema ničega o čemu bi izvestio, a nakon toga će klijent započeti nov zahtev. Periodično restartovanje zahteva čini ovu tehniku i izdržljivijom, omogućavajući klijentima da se oporave od privremenih prekida konekcije ili problema sa serverom.

Zauzet server koji koristi dugačko anketiranje mogao bi imati hiljade zahteva na čekanju i otvorenih TCP konekcija. Node, koji olakšava upravljanje velikim brojem konekcija, bez pravljenja posebne niti izvršavanja za svaku, dobar je izbor za takav sistem.

## HTTP interfejs

Pre nego što počnemo da dizajniramo server ili klijenta, razmislimo o tački gde se oni dodiruju: HTTP interfejsu preko kog će komunicirati. Koristićemo JSON kao format tela zahteva i odgovora. Kao i za server datoteka iz poglavlja 20, trudićemo se da pametno koristimo HTTP metode i zaglavlja. Interfejs se vrti oko putanje `/talks`. Putanje koje ne počinju sa `/talks` koristiće se za isporučivanje statičnih datoteka – HTML i JavaScript koda za sistem na klijentskoj strani.

Zahtev `GET` za `/talks` vraća JSON dokument poput ovoga:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]}
```

Novo predavanje se pravi postavljanjem zahteva `PUT` URL-u kao što je `/talks/Unituning`, gde je deo nakon druge kose crte naslov predavanja. Telo zahteva `PUT` treba da sadrži JSON objekat koji ima svojstva `presenter` (predavač) i `summary` (rezime).

Pošto naslovi predavanja mogu da sadrže razmake i druge znakove koji se obično ne javljaju u URL-u, znakovni nizovi naslova moraju biti šifrovani funkcijom `encodeURIComponent` pri izgradnji takvog URL-a.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

Zahtev da se napravi predavanje o stajanju u mestu na uniciklu (engl. *idling*) mogao bi izgledati ovako:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
"summary": "Standing still on a unicycle"}
```

Takvi URL-ovi podržavaju i zahteve `GET` za preuzimanje JSON-ove predstave predavanja i zahteve `DELETE` za brisanje predavanja.

Komentari se predavanju dodaju pomoću zahteva `POST` za URL kao što je `/talks/Unituning/comments`, sa JSON telom koje ima svojstva `author` (autor) i `message` (poruka).

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
"message": "Will you talk about raising a cycle?"}
```

Da bi podržali dugačko anketiranje, zahtevi `GET` za `/talks` mogu uključivati dodatna zaglavlja koja javljaju serveru da odloži odgovor ako nema dostupnih novih informacija. Mi ćemo koristiti par zaglavlja koja su obično namenjena upravljanju keširanjem: `ETag` i `If-None-Match`.

Serveri mogu u odgovor da uključe zaglavlje `ETag` (skraćeno od „*entity tag*“ – oznaka entiteta). Njegova vrednost je znakovni niz koji označava tekuću verziju resursa. Klijenti, kada kasnije ponovo budu zahtevali taj resurs, mogu napraviti *uslovljeni zahtev* (engl. *conditional request*) uključujući zaglavlje `If-None-Match` čija vrednost sadrži taj isti znakovni niz. Ako se resurs nije promenio, server će odgovoriti statusnim kodom 304, što znači „nije izmenjen“, govoreći klijentu da je verzija koju ima u kešu i dalje aktuelna. Kada se oznaka ne poklapa, server će odgovoriti kao i obično.

Nešto takvo nam je potrebno da bi klijent mogao serveru da kaže koju verziju liste predavanja ima, i da server odgovara samo kada je ta lista izmenjena. Ali umesto trenutnog vraćanja odgovora 304, server bi mogao odugovlačiti sa odgovorom i vratiti ga samo kada je nešto novo dostupno ili kada je

zadata količina vremena istekla. Da bismo napravili razliku između zahteva dugačkog anketiranja i običnih uslovljenih zahteva, daćemo im još jedno zaglavlje, `Prefer: wait=90`, koje serveru govori da je klijent voljan da čeka do 90 sekundi na odgovor.

Server će čuvati broj verzije koji će ažurirati svaki put kada se predavanja promene i koristiće je kao vrednost `ETag`. Klijenti prave zahteve poput ovoga da bi bili obavešteni kada se predavanja promene:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(prolazi vreme)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

Protokol koji je ovde opisan ne obavlja nikakvu kontrolu pristupa. Svi mogu da komentarišu, menjaju predavanja, pa čak i da ih brišu. (Pošto je internet pun huligana, postavljanje takvog sistema na mrežu, bez dodatne zaštite, verovatno se ne bi dobro završilo.)

## Server

Počnimo sa izgradnjom serverske strane programa. Kôd u ovom odeljku pokreće Node.js.

### *Usmeravanje*

Naš server će koristiti `createServer` za pokretanje HTTP servera. U funkciji koja obrađuje nov zahtev, moramo da napravimo razliku između raznih vrsta zahteva (kao što je određeno metodom i putanjom) koje podržavamo. To se može uraditi dugim lancem naredbi `if`, ali postoji i elegantniji način.

*Ruter* ili usmerivač (engl. *router*) komponenta je koja pomaže u otpremanju zahteva do funkcije koja će ga obraditi. Na primer, ruteru možete reći da se datom funkcijom mogu obraditi zahtevi `PUT` sa putanjom koja se poklapa sa regularnim izrazom `/^\/talks\/([^\/]+)$/` (`/talks/` nakon čega sledi naslov predavanja). Pored toga, to će pomoći u izdvajanju smislenih delova putanje (u ovom slučaju, naslova predavanja), obavijenih zagradama u regularnom izrazu, i njihovom prosleđivanju do funkcije za obradu.

Postoji više dobrih paketa rutera na NPM-u, ali mi ćemo ga ovde napisati sami da bismo prikazali princip.

Ovo je `router.js`, koji ćemo kasnije zahtevati (koristeći `require`) od našeg serverskog modula:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
    }
    return null;
  }
};
```

Modul izvozi klasu `Router`. Objekat rutera dozvoljava registrovanje novih obrađivača metodom `add` i može da razreši zahteve svojom metodom `resolve`.

Ova druga će vratiti odgovor kada obrađivač bude pronađen, a inače će vratiti `null`. Ona isprobava rute jednu po jednu (redom kojim su definisane) sve dok ne pronađe odgovarajuću.

Funkcije obrađivača pozivaju se sa vrednošću `context` (koja će u našem slučaju biti instanca servera), sa znakovnim nizovima za pronalaženje grupa koje su definisane u njihovom regularnom izrazu i sa objektom zahteva. Znakovni nizovi moraju biti dekodirani za URL jer sirovi URL može da sadrži kodove u stilu `%20`.

### *Isporučivanje datoteka*

Kada se zahtev ne poklapa ni sa jednim tipom zahteva definisanim u ruteru, server mora da ga tumači kao zahtev za datoteku iz direktorijuma `public`. Mogli bismo koristiti server datoteka definisan u poglavlju 20 za isporučivanje takvih datoteka, ali nije nam ni potrebna ni poželjna podrška za zahteve `PUT` i `DELETE` za datoteke, a voleli bismo naprednije mogućnosti kao što je podrška za keširanje. Zbog toga ćemo koristiti pouzdani, provereni server statičnih datoteka sa NPM-a.

Ja sam se odlučio za `ecstatic`. To nije jedini takav server na NPM-u, ali dobro radi i odgovara našim potrebama. Paket `ecstatic` izvozi funkciju koja se može pozvati sa konfiguracionim objektom da bi nastala funkcija za obradu zahteva. Koristićemo opciju `root` da bismo serveru rekli gde da traži datoteke. Funkcija obrađivača prihvata parametre `request` i `response` i može se proslediti direktno do funkcije `createServer` da bi nastao server koji isporučuje *samo* datoteke. Međutim, prvo želimo da proverimo ima li zahteva koje treba da obradimo posebno, pa ga obavijamo drugom funkcijom.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

Ovde se za odgovore koristi slična konvencija kao za server datoteka iz prethodnog poglavlja – obrađivači vraćaju obećanja koja se razrešuju kao objekti koji opisuju odgovor. Ona obavija server objektom koji čuva i njegovo stanje.

### *Predavanja kao resursi*

Predložena predavanja sačuvana su u svojstvu `talks` servera, objektu čija su imena svojstava naslovi predavanja. Ona će biti izložena kao HTTP resursi u okviru `/talks/[title]`, pa ruteru treba da dodamo obrađivače koji implementiraju razne metode koje klijenti mogu koristiti da bi radili sa njima.

Obrađivač za zahteve koji dobavljaju (`GET`) jedno predavanje mora da potraži predavanje i da odgovori ili JSON podacima predavanja ili odgovorom koji prijavljuje grešku 404.