Rust veb razvoj

wrap, Tokio i request

Bastian Gruber

kompjuter biblioteka

MANNING

Rust
veb razvoj

BASTIAN GRUBER

MANNING

kompjuter biblioteka

Izdavač:
kompjuter biblioteka

Obalskih radnika 4a, Beograd

Tel: 011/2520272

e-mail: kombib@gmail.com

internet: www.kombib.rs

Urednik: Mihailo J. Šolajić

Za izdavača, direktor:
Mihailo J. Šolajić

Autor: Bastian Gruber

Prevod: Biljana Tešić

Lektura: Miloš Jevtović

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2023.

Broj knjige: 563

Izdanje: Prvo

ISBN: 978-86-7310-586-4

Rust Web Development
WITH WARP, TOKIO, AND REQWEST

BASTIAN GRUBER

©2023 by Manning Publications Co.
9781617299001

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.738.1:004.652.4
004.438RUST

ГРУБЕР, Бастиан
Rust veb razvoj / Bastian Gruber ; [prevod Biljana Tešić]. - Izd. 1. - Beograd : Kompjuter Biblioteka, 2023 (Zemun : Pekograf). - XXV, 378 str. : ilustr. ; 24 cm. - (Kompjuter biblioteka ; br. knj. 563)

Prevod dela: Rust Web Development. - Tiraž 500. -
O autoru: str. [XVI]. - Registar.

ISBN 978-86-7310-586-4

а) Web презентације -- Дизајн
б) Програмски језик „Rust“

COBISS.SR-ID 111240201

KRATAK SADRŽAJ

SADRŽAJ

DEO 2

POGLAVLJE 6

POGLAVLJE 7

POGLAVLJE 8

DEO 3

POGLAVLJE 9

POGLAVLJE 10

POGLAVLJE 11

PREDGOVOR

Ja sam pragmatičar u srcu. Moj uvod u programiranje inspirisan je susedom iz mog malog rodnog grada, koji je prodavao veb sajtove za preduzeća za (tada) velike sume novca. Mislio sam: ako on može da zaradi prodajući veb sajtove, mogu i ja. Prijatelj i ja smo počeli da izrađujemo veb sajtove za razne kompanije kada sam imao 17 godina. Gledajući iz udobnosti svog doma tu količinu vrednosti koja je „otključana" za kompanije, zaljubio sam se u softversku industriju.

Međutim, programiranje nikada nije bila moja omiljena tema, nikada nešto u šta sam želeo da duboko „zaronim". To je bilo sredstvo za postizanje cilja, nešto što sam morao da radim da bih mogao da isporučim aplikaciju ili veb sajt. Prešao sam sa pisanja PL/I na centralnom računaru na pisanje Java Scripta za aplikacije pregledača, dok sam u međuvremenu izrađivao backend API-e. Jednostavno, volim da programiram za Internet. Ta strast me je dovela do Rusta. Rust jezik i njegov kompajler su mi „čuvali leđa" da bih mogao da se fokusiram na ono što je važno, a to je stvaranje vrednosti za druge.

Knjiga „Rust veb razvoj" je napisana sa tog pragmatičnog stanovišta za softversku industriju stvaranjem vrednosti pomoću najboljih trenutno dostupnih alatki. Ova knjiga pokazuje zašto Rust, čak i ako nije jasan na prvi pogled, savršeno odgovara budućoj generaciji veb aplikacija i API-a. U njoj ne razmatramo samo sintaksu, već obezbeđujemo smernice, „ronimo" duboko i ohrabrujemo vas da sa samopouzdanjem započnete i završite vaš sledeći projekat pomoću Rusta.

Želim da „podignem zavesu" i pogledam „iza kulisa" Rust cratea, samog Rust jezika i veb radnih okvira koje biram. Težiću ka tome da nivo detalja uvek bude pragmatičan – na primer, pokazaću koliko znanja je potrebno da nešto promenite i da razumete rešenje kako biste ga mogli prilagoditi vašem projektu, a ukazaću vam i gde treba da istražujete dalje.

Da citiram jednog od mojih bivših kolega: „Pisanje Rusta je kao varanje!" Nadam se da će vam ova knjiga pomoći da vidite lepotu programiranja za veb pomoću jezika koji vam „čuva leđa" i koji omogućava da neke poslove obavljate brže i bezbednije nego ranije. Čast mi je da vas povedem na ovo „putovanje"!

PRIZNANJA

Prvo, moram da se zahvalim mojoj supruzi Emily, jer je verovala u mene, „gurala“ me napred i nikada nije prestala da veruje da mogu da završim ovu knjigu. Pisanje ove knjige oduzelo nam je mnogo sati od našeg ionako ograničenog vremena i zauvek ću joj biti zahvalan na podršci. Emily, hvala ti što uvek „čuvaš leđa“ našoj porodici i meni. Volim te!

Zatim se zahvaljujem Mikeu Stephensu, jer je omogućio da se ova knjiga realizuje. Prvi razgovori sa njim bili su zaista inspirativni i učinili su da verujem da zaista mogu da napišem knjigu. Njegova mudrost i iskustvo uticali su na ovu knjigu i uticaće na moje pisanje u godinama koje dolaze.

Mom uredniku u Manningu Eleshai Hydei: „Hvala vam na strpljenju, doprinosu, stalnom praćenju e-pošte i neprocenjivim predlozima i instrukcijama tokom ovog ‘putovanja’. Uvek sam se radovao našim susretima“.

Hvala programerima koji su me inspirisali na ovom „putovanju“: Mariano, vaša mudrost i uvidi su me vodili ne samo kroz ovu knjigu, već i kroz dobar deo moje karijere programera. Knute i Blake, naše vreme u smartB-u i diskusije koje su usledile oblikovale su način na koji se ophodim prema čitaocima ove knjige. Simone, naučio si me šta je potrebno da bih postao programer i ozbiljno shvatio svoj zanat. I hvala ti, Paule, što si mi olakšao neke „stvari“, omogućio da „napunim baterije“ i zainteresovao me za naš zanat kroz naše razgovore. Dada, zajedničko učenje sa tobom bio je jedan veliki „kamen-temeljac“ za pisanje ove knjige. I na kraju, ali ne i najmanje važno, Sebastiane i Fernando, vreme koje smo proveli zajedno oblikovalo me je više od bilo čega drugog da budem programer i osoba kakva sam danas.

Dragocenu pomoć da ova knjiga bude što bolja svojim predlozima i sugestijama pružili su mi recenzenti Alain Couniot, Alan Lenton, Andrea Granata, Becker, Bhagvan Kommadi, Bill LeBorgne, Bruno Couriol, Bruno Sonnino, Carlos Cobo, Casey Burnett, Christoph Baker, Christopher Lindblom, Christopher Villanueva, Dane Balia, Daniel Tomás Lares, Darko Bozhinovski, Gábor László Hajba, Grant Lennon, Ian Lovell, JD McCormack, Jeff Smith, Joel Holmes, John D. Lewis, Jon Riddle, JT Marshall, Julien Castelain, Kanak Kshetri, Kent R. Spillner, Krzysztof Hrynczenko, Manzur Mukhitdinov, Marc Roulleau, Oliver Forral, Paul Whittemore, Philip Dexter, Rani Sharim, Raul Murciano, Renato Sinohara, Rodney Weis, Samuel Bosch, Sergiu Raducu Popa, Timothy Robert James Langford, Walt Stoneburner, William E. Wheeler i Xiangbo Mao. Hvala im!

„Rust veb razvoj“ će vam pomoći da pišete veb aplikacije (API, mikrouslugu ili monolit) od početka do kraja. Naučićete sve što vam je potrebno da otvorite API za spoljni svet, da povežete bazu podataka za skladištenje podataka i da testirate i primenite svoje aplikacije.

O OVOJ KNJIZI

Ovo nije knjiga za referencu, već treba da je čitate kao da je radna sveska. Aplikacija koju izrađujemo će „žrtvovati“ svoj dizajn (koji će biti lošiji) kako biste koncepte mogli da naučite u pravo vreme. Potrebno je da pročitate celu knjigu da biste konačno mogli da aplikacije pustite u rad.

Ko bi trebalo da čita ovu knjigu

Ova knjiga je za ljude koji su pročitali prvih šest poglavlja knjige „The Rust Programming Language“ (čiji su autori Steve Klabnik i Carol Nichols - No Starch Press, 2019), a zatim se zapitali šta mogu da urade pomoću Rusta. Takođe je namenjena programerima koji su u prošlosti kreirali veb aplikacije na drugom programskom jeziku i pitaju se da li bi Rust bio dobar izbor za njihov sledeći projekat. I na kraju da napomenem da je „Rust veb razvoj“ odlična knjiga koja će pomoći vama ili novom zaposlenom na radnom mestu koje zahteva da pišete i održavate veb aplikacije u Rustu.

Kako je ova knjiga organizovana: mapa puta

Knjiga „Rust veb razvoj“ ima tri dela, sa ukupno 11 poglavlja i sa jednim dodatkom.

U Delu 1 je objašnjeno zašto i kako treba pisati Rust:

- U Poglavlju 1 razmotreno je u koje okruženje i tim se Rust odlično uklapa i objašnjen je način razmišljanja prilikom odabira Rusta za vaš tim ili sledeći projekat. Izvršeno je poređenje Rust jezika sa drugim jezicima i obezbeđen je kratak pregled njegovog veb eko-sistema.
- U Poglavlju 2 su predstavljeni osnove Rust jezika i znanje koje je potrebno da biste izvršili sve zadatke iz knjige i razumeli predstavljene isečke koda. Takođe su obuhvaćene osnove veb eko-sistema i objašnjene su dodatne alatke koje su potrebne za pisanje asinhronih aplikacija u Rustu.

U Delu 2 je razmotreno kreiranje poslovne logike aplikacije:

- U Poglavlju 3 je postavljena osnova koju ćemo proširiti kasnije u knjizi. Predstavljen je Warp, odnosno veb radni okvir koji koristimo, i način na koji treba odgovoriti na HTTPGET zahteve pomoću JSON-a.
- U Poglavlju 4 su razmotreni HTTP POST, PUT i DELETE zahtevi i način kako se mogu čitati primeri podataka iz memorije. Ovim poglavljem su obuhvaćene i razlike između „tela“ url-form-encoded i JSON.

- U Poglavlju 5 su predstavljeni modularizacija, linting i formatiranje koda. Mi delimo velike delove koda u njihove module i datoteke i koristimo Rust sistem za komentarisanje da označimo bazu koda, da dodamo pravila za linting i da formatiramo kod.
- U Poglavlju 6 analiziramo pokrenutu aplikaciju. Objašnjavamo razliku između evidentiranja i praćenja i prikazujemo različite načine za debagovanje u kodu.
- U Poglavlju 7 se oslobađamo skladišta iz memorije i dodajemo PostgreSQL bazu podataka. Povezujemo se sa bazom podataka na lokalnom hostu i kreiramo skup veza i delimo ga među hendlerima ruta.
- U Poglavlju 8 se povezujemo sa eksternom uslugom, kojoj šaljemo podatke i obrađujemo primljeni odgovor. Razmatramo kako se mogu spojiti asinhrone funkcije i deserijalizovati JSON odgovori.

U Delu 3 osiguravamo da sve bude spremno za izradu finalne verzije koda:

- U Poglavlju 9 razmatrano autentikaciju bez stanja i kako se ona manifestuje u našoj bazi koda. Uvodimo koncept korisnika i dodajemo posrednički program za validaciju tokena.
- U Poglavlju 10 parametrizujemo naše promenljive, kao što su API ključevi i URL-ovi baze podataka, i pripremamo bazu koda za proširenje na različitim arhitekturama i za Docker okruženje.
- U Poglavlju 11 završavamo knjigu testiranjem jedinica i integracije i pokretanjem i isključivanjem modela servera nakon svakog testa.

Dodatak sadrži uputstva za reviziju i pisanje bezbednog koda.

Knjiga se može čitati u delovima. Spremište koda se može koristiti za proveru poglavlja i podešavanja za deo koji trenutno čitate. Aplikacija se izrađuje poglavlje po poglavlje, tako da možete propustiti neke informacije ako preskočite pojedina poglavlja. Međutim, poglavlja se mogu koristiti kao neobavezni referentni vodič.

O kodu

Primeri koda u knjizi „Rust veb razvoj“ su napisani pomoću Rust izdanja iz 2021. godine i testirani su na Linuxu i macOS-u pomoću Intel i Apple čipova.

Ova knjiga sadrži mnogo primera izvornog koda, kako u numerisanim listinzima, tako i u običnom tekstu. U oba slučaja izvorni kod je formatiran u `fontu fiksne širine` da bismo ga odvojili od običnog teksta. Osim toga, **`podebljani font`** se koristi za isticanje koda koji je promenjen u odnosu na prethodne korake u poglavlju – na primer, kada je nova funkcija dodata postojećoj liniji koda. U nekim slučajevima ~~`precrtavanje`~~ se koristi za označavanje koda koji se zamenjuje.

U mnogim slučajevima originalni izvorni kod je ponovo formatiran; dodali smo prelome redova i preradili uvlačenja da bismo se prilagodili prostoru koji je dostupan u knjizi. Osim toga, komentari u izvornom kodu često su uklonjeni iz listinga nakon što je kod opisan u tekstu. Napomene koda se nalaze u mnogim listinzima, čime se naglašavaju važni koncepti.

Možete preuzeti izvršive isečke koda iz liveBook (onlajn) verzije ove knjige na adresi https://livebook.manning.com/book/rust-web-development. Kompletan kod za primere u knjizi možete preuzeti sa Manning veb sajta na adresi https://www.manning.com/books/rust-web-development i sa GitHuba na adresi https://github.com/Rust-Web-Development/code.

O AUTORU

BASTIAN GRUBER je runtime inženjer u kompaniji „Centrifuge“. Bio je član zvanične grupe „Rust Async Working Group“ i osnovao je grupu „Rust and Tell Berlin Meetup“. Radio je za jednu od najvećih kripto berzi na svetu u corebackendu, koristeći Rust. On je takođe autor nekih stručnih knjiga, koji ima više od 12 godina iskustva i redovno piše o Rustu za „LogRocket“, a njegovi intervjui i govori su prikupljeni na veb sajtu ove knjige (https://rustwebdevelopment.com). Tokom svog rada Bastian je naučio da podučava zainteresovane ljude o složenim konceptima na jednostavan način, a njegovi članci su omiljeni, jer su istovremeno jednostavni i detaljni.

Bastiana možete pronaći na društvenim mrežama pomoću njegovog Twittera @recvonline. Ako želite da stupite u kontakt sa njim, slobodno mu pošaljite e-poštu na adresu foreach@me.com.

DEO 1

UVOD U RUST

U ovom prvom delu knjige postavljamo osnove jezika. Da biste mogli da koristite Rust za veb razvoj, potrebno vam je razumevanje Rust jezika i alatki potrebnih za pisanje asinhronih serverskih aplikacija u Rustu. Prvim delom su obuhvaćene osnove jezika i alatke.

U Poglavlju 1 odgovaramo na pitanje zašto Rust. Pokazujemo kako on može biti efikasniji od drugih jezika, a istovremeno omogućava da pomoću njega lako i bezbedno kreirate aplikacije. Pokazujemo kako da podesite Rust lokalno, kako izgleda lanac alatki, i, što je najvažnije, kako asinhronizovani i veb eko-sistem izgleda u Rustu.

U Poglavlju 2 razmatramo sve osnove koje su potrebne da znate ne samo da biste pratili isečke koda u knjizi, već i da biste lako započeli novi projekat u Rustu.

1 ZAŠTO RUST?

Ovim poglavljem obuhvaćene su sledeće teme:

- alatke koje se isporučuju u paketu sa standardnom Rust instalacijom
- prvi pogled na Rust kompajler i na ono što ga čini jedinstvenim
- šta je potrebno za pisanje veb servisa u Rustu
- koje funkcije podržavaju održavanje Rust aplikacije

Rust je sistemski programski jezik, a umesto interpretiranog jezika, kao što su JavaScript ili Ruby, ima kompajler, kao što imaju Go, C ili Swift. Ne kombinuje aktivni runtime (iz C jezika), ali obezbeđuje jezičku ergonomiju poznatu iz Pythona i Rubya (pogledajte sliku 1.3 za referencu). Sve ovo je moguće zahvaljujući kompajleru koji sprečava greške bilo kojeg tipa i osigurava da ne dođe do problema u memoriji pre nego što pokrenete aplikaciju.

Rust obezbeđuje performanse (nema runtime, niti prikupljanje „smeća“), bezbednost (kompajler osigurava da je sve bezbedno za memoriju, čak i u asinhronim okruženjima) i produktivnost (njegove ugrađene alatke za testiranje, dokumentacija i „menadžer“ paketa čine ga lakim za izradu i održavanje).

Možda ste čuli za Rust, ali nakon što ste pokušali da pratite neke tutorijale, odustali ste od njega. Rust se pojavljuje kao najomiljeniji programski jezik u godišnjim StackOverflow anketama i našao je veliki broj sledbenika u korporacijama kao što su „Facebook“, „Google“, „Apple“ i „Microsoft“. U ovoj knjizi ćete da saznate kako možete da upoznate osnove Rusta i kako da pomoću njega izrađujete i isporučujete stabilne veb usluge.

> **NAPOMENA** Pretpostavljamo da ste napisali nekoliko malih Rust aplikacija i da poznajete opšte koncepte neke veb usluge. Razmotrićemo sve osnovne funkcije Rust jezika i kako se mogu koristiti u ovoj knjizi, ali više u obliku podsetnika (nećemo ulaziti u detalje). Ako ste, na primer, pročitali polovinu knjige „The Rust Programming Language“ autora Stevea Klabnika i Carola Nicholsa (No Starch Press, 2019, https://doc.rust-lang.org/book), nećete imati problema da pratite vežbe predstavljene u ovoj knjizi. Knjigom je obuhvaćen Rust 2021 i kompatibilan je sa starijom verzijom 2018.

Za vas, kao programera, Rust je jedinstvena šansa da proširite svoj horizont. Možda ste frontend programer koji želi da se bavi backend razvojem ili Java programer koji želi da nauči novi jezik. Rust je toliko prilagodljiv da može da proširi vrste sistema koje možete da koristite kada ga naučite. Rust možete koristiti gde god koristite C++ ili C, ali i u situaciji kada biste koristili NodeJS, Java i Ruby. Čak počinje da pronalazi čvrsto uporište u eko-sistemu mašinskog učenja, u kojem Python godinama dominira. Osim toga, Rust je odličan za kompajliranje u WebAssemblyu (https://webassembly.org), a mnoge moderne implementacije blokčejna (Cosmos, Polkadot) su napisane u Rustu.

Što više vremena provodite pišući kod i što više programskih jezika učite, sve više shvatate da su najvažniji koncepti koje naučite i korišćenje programskog jezika koji najbolje odgovara za rešavanje određenih problema. U ovoj knjizi se, dakle, ne zaustavljamo samo na tome da vam pokažemo koje linije Rust koda omogućavaju da kreirate HTTP zahteve, već su opisani uopšteno funkcionisanje veb usluga i osnovni koncepti asinhronog Rusta, tako da možete izabrati Transmission Control Protocol (TCP) apstrakciju koja vam najviše odgovara.

1.1 Uključene „baterije“: Rustove alatke

Rust se isporučuje sa odgovarajućim brojem alatki za lakše pokretanje, održavanje i izradu aplikacija. Na slici 1.1 je pregled najvažnijih alatki koje su vam potrebne da biste započeli pisanje Rust aplikacija.

Možete preuzeti rustup i instalirati Rust, tako što ćete izvršiti komandu na terminalu, koja je prikazana u listingu 1.1. Ovo instaliranje funkcioniše na macOS-u (pomoću komande brew install rustup-init) i na Linuxu. Za instaliranje Rusta sa ažuriranjima na Windowsu, pratite uputstva na Rust veb sajtu (https://www.rust-lang.org/tools/install).

„menadžer“ alatki/verzija	Rust kompajler	formater koda	Linter (program za popravljanje koda)	„menadžer“ paketa	registar paketa
Rustup	Rustc	Rustfmt	Clippy	Cargo	crates.io

Slika 1.1Sve alatke koji su potrebne za pisanje i isporuku Rust aplikacija

Listing 1.1 Instaliranje Rusta

```
$ curl --proto '=https' --tlsv1.2 -sSf https:/ /sh.rustup.rs | sh
```

Alatka komandne linije `curl` je izrađena za prenos podataka pomoću URL-ova. Možete preuzeti udaljene datoteke na vaš računar. Opcija `--proto` omogućava korišćenje protokola, kao što je Hypertext Transfer Protocol Secure (HTTPS), koji mi koristimo. U parametru `--tlsv1.2` koristimo Transport Layer Security (http://mng.bz/o5KM) u verziji1.2. Sledi URL, koji, ako ga otvorimo pomoću pregledača, obezbeđuje shell skript za preuzimanje. Ovaj shell skript se *usmerava* (pomoću operatora |) u alatku komandne linije sh koja ga izvršava.

Ova shell skripta će takođe instalirati alatku rustup i omogućiti da ažurirate Rust i instalirate pomoćne komponente. Ažuriranje Rusta ćete izvršiti jednostavno pokretanjem komande `rustup update`:

```
$ rustup update
info: syncing channel updates for 'stable-aarch64-apple-darwin'
info: syncing channel updates for 'beta-aarch64-apple-darwin'
info: latest update on 2022-04-26,
      rust version 1.61.0-beta.4 (69a6d12e9 2022-04-25)

…

  stable-aarch64-apple-darwin unchanged - rustc 1.60.0
 (7737e0b5c 2022-04-04)
      beta-aarch64-apple-darwin updated -
      rustc 1.61.0-beta.4 (69a6d12e9 2022-04-25)
      (from rustc 1.61.0-beta.3 (2431a974c 2022-04-17))
   nightly-aarch64-apple-darwin updated -
    rustc 1.62.0-nightly (e85edd9a8 2022-04-28)
   (from rustc 1.62.0-nightly (311e2683e 2022-04-18))

info: cleaning up downloads & tmp directories
```

A ako želite da instalirate više komponenata, kao što je linter, koji je pomenut na slici 1.1, koristite i komandu `rustup`.

Listing 1.2 Instaliranje rustfmta

```
$ rustup component add rustfmt
```

Pokretanje ove alatke pomoću `cargo fmta` će uporediti vaš kod sa stilskim vodičem i formatirati kod u skladu sa tim. Morate da navedete fasciklu ili datoteku za koju želite da ga pokrenete. Možete, na primer, da se krećete do osnovne fascikle vašeg projekta i pokrenete `cargo fmt.` (sa tačkom) za sve direktorijume i datoteke.

Nakon što smo izvršili `curl` iz listinga 1.1, ne samo da smo instalirali Rust biblioteku, već i „menadžer" paketa Cargo, pa možemo da kreiramo i pokrećemo Rust projekte. Sada ćemo da kreiramo i izvršimo naš prvi Rust program. U listingu 1.3 prikazano je kako možete da pokrenete Rust aplikaciju. Komanda `cargo run` će izvršiti `rustc`, kompajlirati kod i pokrenuti generisanu binarnu datoteku.

Listing 1.3 Pokretanje našeg novog Rust projekta

```
$ cargo new hello
$ cd hello
$ cargo run

   Compiling hello v0.1.0 (/private/tmp/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.54s
     Running `target/debug/hello`
Hello, world!
```

Naš novi program ispisuje `Hello, World!` u konzolu - uskoro ćete videti zašto. Kada pogledamo u našu fasciklu projekta hello, vidimo datoteke i fascikle navedene u listingu 1.4. Komanda cargo new kreira novu fasciklu sa nazivom koji budemo naveli, a takođe inicijalizuje novu git strukturu za nju.

Listing 1.4 Sadržaj fascikli novog Rust projekta

```
❯ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    └── debug
        ├── build
        ├── deps
        │   ├── ...
        ├── examples
        ├── hello
        ├── hello.d
        └── incremental
            └── ...

9 directories, 28 files
```

Dodajemo zavisnosti nezavisnih proizvođača u datoteku Cargo.toml, koje će biti preuzete dok se izrađuje binarna datoteka.

Glavni fokus je na fascikli src tokom programiranja; vaš kod će se nalaziti u toj fascikli.

Prilikom kreiranja binarne datoteke biće kreirana ciljna fascikla koja sadrži artefakte izrade.

Sama binarna datoteka se nalazi u fascikli debug kada se izvršava cargo run u komandnoj liniji.

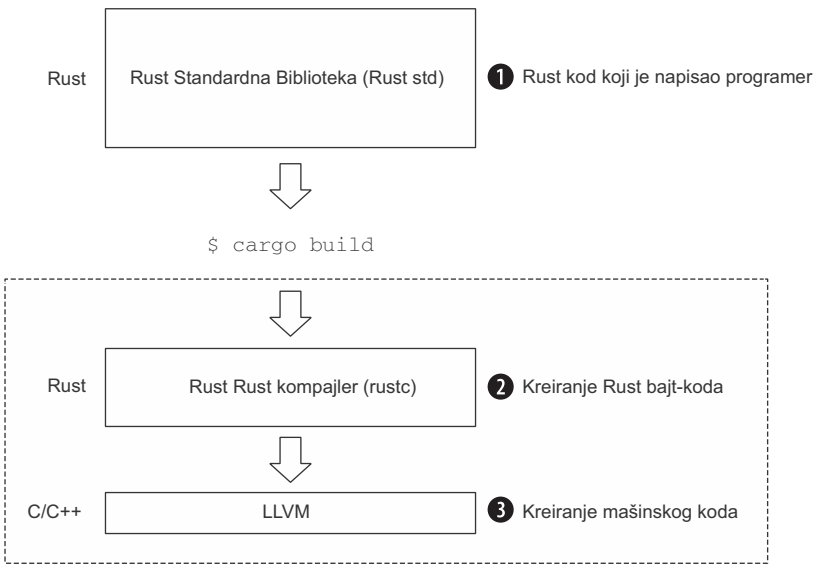
Komande cargo run će izraditi aplikaciju i izvršiti binarnu datoteku koja se nalazi u fascikli ./target/debug. Naš izvorni kod se nalazi u fascikli src/ i, u zavisnosti od tipa aplikacije koju izrađujemo, ima ili datoteku main.rs ili datoteku lib.rs, sa sledećim sadržajem:

Listing 1.5 Automatski generisana datoteka main.rs

```
fn main() {
    println!("Hello, world!");
}
```

U Poglavlju 5 videćete razliku između datoteke lib.rs i datoteke `main.rs` i kada `Cargo` kreira te fascikle. Fascikla `target` sadrži drugu fasciklu koja se zove `debug`, a koja sadrži naš kompajlirani kod generisan pomoću komande `cargo run`. Jednostavna komanda `cargo build` bi imala isti efekat, ali bi samo izradila program, a ne bi ga izvršila.

Kada izrađujete Rust program, Rust kompajler (Rustc) kreira Rust bajt-kod i prosleđuje ga drugom kompajleru koji se zove LLVM (https://llvm.org) da bi bio kreiran mašinski kod (LLVM koriste, takođe, jezici kao što su Swift i Scala; on pretvara bajt-kod koji je proizveo jezički kompajler u mašinski kod za pokrenuti operativni sistem). To znači da se Rust može kompajlirati na bilo kojem operativnom sistemu koji podržava LLVM. Ceo stek je prikazan na slici 1.2.



Slika 1.2 Nakon instaliranja Rustupa, imate Rust standardnu biblioteku na vašem uređaju koja sadrži Rust kompajler

Još jedna važna datoteka je `Cargo.toml`. Kao što se vidi u listingu 1.6, ona sadrži ukupne informacije o našem projektu i navodi zavisnosti nezavisnih proizvođača ako je potrebno.

> **NAPOMENA** Kada razvijate biblioteke, datoteka Cargo.lock ne treba da bude proveravana u vašem sistemu za upravljanje verzijama (kao što je Git). Međutim, kada kreirate aplikaciju (binarnu), trebalo bi da dodate datoteku u vaš sistem za upravljanje verzijama. Aplikacije (binarne) često zavise od specifičnih verzija eksterne biblioteke, pa drugi programeri sa kojima radite moraju da znaju koje verzije su bezbedne za instaliranje ili na koje treba da se nadgrade. Sa druge strane, biblioteke treba da funkcionišu u najnovijim verzijama korišćenih biblioteka.

Listing 1.6 Sadržaj datoteke Cargo.toml

```
[package]
name = "check"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html
[dependencies]
```

Instaliranje biblioteka nezavisnih proizvođača se izvršava dodavanjem naziva zavisnosti u odeljku `[dependencies]` i pokretanjem komandi `cargo run` ili `cargo build`, pri čemu će biti preuzete biblioteke (tzv. *crates* u zajednici Rust) iz Rust registra paketa crates.io. Aktuelna preuzeta verzija instaliranih paketa pojavljuje se u novoj datoteci Cargo.lock. Ako se datoteka nalazi u osnovnoj fascikli vašeg projekta, Cargo će preuzeti verziju paketa koji su navedeni u datoteci Cargo.lock. To će pomoći programerima koji koriste istu bazu koda da izvrše replikaciju potpuno istog stanja na različitim uređajima.

TOML datoteka

Format datoteke TOML je poput JavaScript Object Notation (JSON) ili YAML formata konfiguracione datoteke. TOML je skraćenica za „Tom's Obvious Minimal Language", pa, kao što sam naziv sugeriše, trebalo bi da olakša čitanje i raščlanjavanje konfiguracija. „Menadžer" paketa cargo koristi ovu datoteku za instaliranje zavisnosti i popunjavanje informacija o projektu.

Da citiram jednog od važnih članova Rusta: „TOML je najmanje zastrašujuća opcija" (https://users.rust-lang.org/t/why-does-cargo-use-toml/3577/4) - to ne znači da je TOML loš, ali uvek postoje ustupci prilikom rukovanja konfiguracionim datotekama.

Poslednja alatka oko našeg „pojasa" je linter Clippy zvaničnog koda. Ova alatka je podrazumevano uključena kada instalirate Rust. Takođe se može instalirati ručno ako koristite starije verzije Rusta.

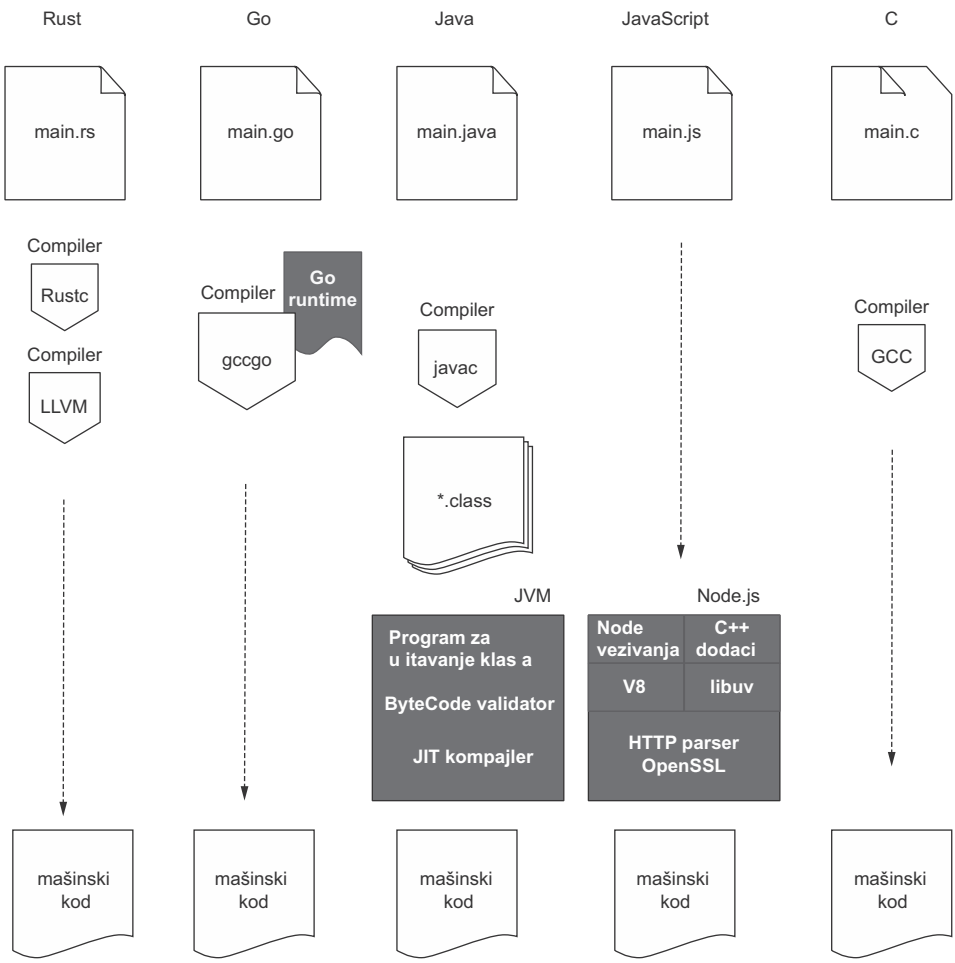
Listing 1.7 Instaliranje Clippya

```
$ rustup component add clippy
```

U Poglavlju 5 predstavljamo korišćenje alatke Clippy i detalje o načinu njene konfiguricije.

1.2 Rust kompajler

Prednost korišćenja Rusta u odnosu na druge jezike je u njegovom kompajleru. Rust se kompajlira u binarni kod bez sakupljanja „smeća“ koje se aktivira tokom izvršavanja, pri čemu Rust postaje brz kao C. Međutim, za razliku od C kompajlera, Rust kompajler obezbeđuje bezbednost memorije tokom kompajliranja. Na slici 1.3 pokazane su razlike između popularnih programskih jezika koji se koriste za programiranje na strani servera i jezika C.



Slika 1.3 Upoređeno kompajliranje mašinskog koda iz izvornog koda u Rustu i u nekoliko drugih jezika

Svaki jezik bira ustupke. Go je najnoviji od demonstriranih jezika i brz je kao jezik C. On koristi runtime za sakupljanje „smeća“ i, stoga, više troši memoriju nego Rust. Go kompajler je brži nego Rustc kompajler. Go je jednostavan i potrebno mu je malo performansi runtimea za prikupljanje “smeća”.

Vidite da Rust ne troši dodatnu memoriju za vreme izvršavanja, a zahvaljujući kompajleru, nudi više jednostavnosti i bezbednosti tokom pisanja koda nego Go ili JavaScript. Java i JavaScript zahtevaju i neku vrstu virtuelne mašine za pokretanje koda. To rezultira velikim smanjenjem performansi.

Jedno od prilagođavanja koje ćete morati da izvršite kada pišete programe u Rustu je da koristite Rust kompajler da biste kreirali stabilne aplikacije. Kada koristite skriptni jezik, ovo je ogromna promena u načinu razmišljanja. Umesto da pokrene aplikaciju za nekoliko sekundi i debaguje dok ne „otkaže“, Rust kompajler će se pobrinuti da sve funkcioniše dobro pre nego što se pokrene. Na primer, upotrebite sledeći isečak koda (koji će biti prikazan kasnije u knjizi, a ovde je odštampan samo za demonstracione svrhe).

Listing 1.8 Proveravanje da li postoji prazan ID

```
match id.is_empty() {
    false => Ok(QuestionId(id.to_string())),
    true => Err(Error::new(ErrorKind::InvalidInput, "No id provided")),
}
```

Ako još uvek ne možete da pročitate ovaj isečak koda, ne brinite, uskoro ćete moći. To je takozvani blok `match`, a kompajler će se pobrinuti da bude obuhvaćen svaki slučaj upotrebe (ako je `id` prazan ili ne). Ako izbrišemo liniju koda pomoću komande `true =>` i pokušamo da kompajliramo naš kod, biće generisana greška.

Listing 1.9 Greška kompajlera zbog nedostajućeg podudaranja šablona

```
error[E0004]: non-exhaustive patterns: `true` not covered
  --> src/main.rs:31:15
   |
31 |         match id.is_empty() {
   |               ^^^^^^^^^^^^^ pattern `true` not covered
   |
   = help: ensure that all possible cases are being handled,
      possibly by adding wildcards or more match arms
   = note: the matched value is of type `bool`
```

Kompajler ističe liniju i tačnu poziciju u našem iskazu, ali i generiše predlog kako da rešimo neki trenutni problem. Dizajniran je da generiše poruke o grešci koje su razumljive i čitljive ljudima, a ne samo da prikazuje internu grešku parsera.

Proveranje da li program funkcioniše pravilno može izgledati zamorno u malim aplikacijama, ali kada budete morali da održavate veće sisteme i dodajete ili uklanjate funkcije, brzo ćete

videti da pisanje Rusta ponekad može da izgleda kao da varate, jer će mnogo problema o kojima ste morali da razmišljate u prošlosti sada biti obuhvaćeno kompajlerom.

Dakle, retko ćete moći odmah da pokrenete upravo napisani Rust kod. Kompajler ćete koristiti u svakodnevnoj rutini i pomoći će vam da razumete gde da poboljšate svoj kod i ukazaće šta ste možda zaboravili da obuhvatite.

Ne možete brzo da pređete na Rust kao što možete, na primer, na JavaScript , čak, na Go. Prvo morate da upoznate skup osnovnih koncepata. Postoji mnogo aspekata Rusta koje treba da naučite da biste postali iskusni Rust programer. Ne morate da znate sve da biste započeli rad u Rustu – možete učiti usput pomoću kompajlera.

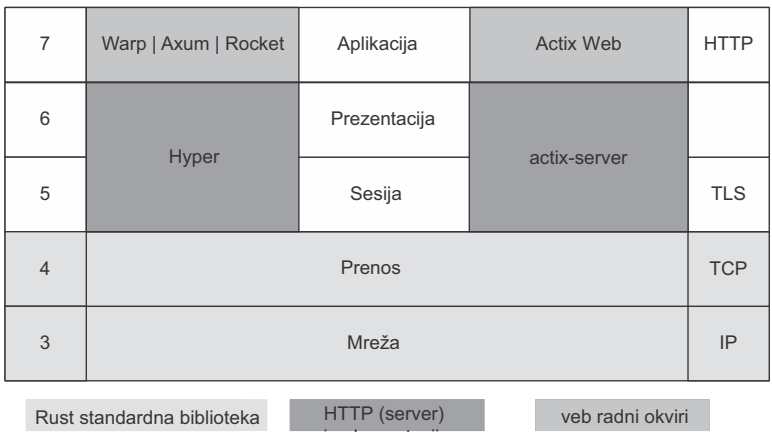
Kada postanete iskusni Rust programer, možete kompajler koristiti u više različitih oblasti – na primer, u razvoju igara, na backend serverima, u mašinskom učenju i možda uskoro u razvoju Linux kernela, o kojem se još uvek diskutuje i koji je u probnoj fazi (https://github.com/Rust-for-Linux).

Ako razvijate aplikacije u većem timu, novoobučeni Rust programeri prvo moraju da nauče da koriste kompajler pre nego što mogu da doprinesu bazi koda, pa će pregledati ogromnu količinu koda i garantovati osnovu kvaliteta koda.

1.3 Rust za veb usluge

Razmotrili smo glavne razloge zbog kojih bismo mogli izabrati Rust u odnosu na druge programske jezike. Pogledajmo kako možemo da počnemo da pišemo veb usluge pomoću njega. Iznenađujuće je da u Rustu nije toliko obuhvaćen HTTP kao, na primer, u jezicima Go ili NodeJS. Pošto je Rust sistemski programski jezik, implementacija HTTP-a i drugih funkcija prepuštena je Rust zajednici.

Pregled Rust standardne biblioteke i biblioteke nezavisnih proizvođača OSI modela



Slika 1.4 Rustom je obuhvaćen TCP, ali HTTP i veće implementacije veb radnog okvira su prepuštene zajednici

Na slici 1.4 je prikazan tipičan tehnički stek veb usluge i u kojoj meri Rust nudi pomoć. Dva donja sloja (TCP/IP) su obuhvaćena Rust stekom. Rust standardna biblioteka implementira TCP i možemo otvoriti TCP (ili User Datagram Protocol - UDP) socket i „osluškivati“ dolazne poruke.

Međutim, ne postoji implementacija HTTP-a. Stoga, ako želite da napišete čisti HTTP server, morate ga ili implementirati „od nule“ ili koristiti neku od biblioteka nezavisnih proizvođača (kao što je biblioteka hyper, koju koristi curl „iza kulisa“).

Kada koristite veb radni okvir, izbor je već automatski izvršen umesto vas.Veb radni okvir ActixWeb koristi, na primer, svoju implementaciju HTTP servera `actix-server`. Kada koristite Warp, Axum ili Rocket, treba da znate da svi oni koriste Hyper kao veb server (otvaraju socket, čekaju i raščlanjavaju HTTP poruke).

Na slici 1.4 možemo videti da je TCP uključen u Rust standardnu biblioteku, ali sve ostalo što je navedeno je omogućila zajednica. Kako to izgleda u kodu? Uzmimo primer iz jezika Go. U sledećem listingu je prikazan primer HTTP GET zahteva u jeziku Go.

Listing 1.10 Kratak primer kako da pokrenete HTTP GET zahtev u jeziku Go

```
package main

import (
    "fmt"
    "net/http"
)

func hello(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "hello\n")
}


func main() {

    http.HandleFunc("/hello", hello)

    http.ListenAndServe(":8090", nil)
}
```

Vidimo da nam Go obezbeđuje HTTP paket za slanje zahteva. Rustu nedostaje HTTP deo i implementira samo TCP protokol. Možemo koristiti Rust za kreiranje TCP servera, kao što se vidi u sledećem listingu. Međutim, ne možemo ga odmah koristiti za odgovaranje pomoću formalnih HTTP zahteva.

Listing 1.11 Primer TCP servera napisanog u Rustu

```
use std::net::{TcpListener, TcpStream};

fn handle_client(stream: TcpStream) {
    // do something
}

fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:80")?;
```

```
    for stream in listener.incoming() {
        handle_client(stream?);
    }
    Ok(())
}
```

Stoga, zajednica treba da implementira HTTP. Srećom, već postoji mnogo implementacija. A kada kasnije izaberete veb radni okvir, ne morate više da brinete o tom delu steka.

Još jedan veliki „kamen-temeljac“ veb usluge je asinhrono programiranje, koje omogućava da obrađujemo više zahteva odjednom i skraćuje vreme čekanja na odgovore sa servera.

Kada veb server primi zahtev, mora se izvršiti neki posao (pristupanje bazi podataka, pisanje datoteka itd). Ako bi drugi zahtev stigao pre nego što je prvi posao završen, trebalo bi da sačeka da se prvi posao završi. Zamislite da milioni zahteva pristižu skoro u isto vreme.

Dakle, potreban nam je način da posao nekako smestimo u pozadinu i nastavimo da prihvatamo zahteve na serveru. Sada „u igru“ ulazi asinhrono programiranje. Drugi radni okviri i jezici (kao što su NodeJS i Go) to rade do izvesne mere automatski u pozadini. U Rustu moramo da razumemo gradivne blokove asinhronog programiranja malo detaljnije da bismo znali koje radne okvire da izaberemo.

Programski jezik (ili eko-sistem oko njega) mora da ponudi sledeće koncepte da bismo mogli da kreiramo aplikacije koje mogu da obrađuju posao asinhrono:

- *sintaksa* - označava deo koda kao asinhroni
- *tip* - složeniji tip koji može zadržati stanje asinhronog toka
- *raspored rada niti (runtime)* - služi za upravljanje programskim nitima ili drugim metodima za stavljanje posla u pozadinu i za napredak tog posla
- apstrakcije kernela - koriste asinhrone Kernel metode u pozadini

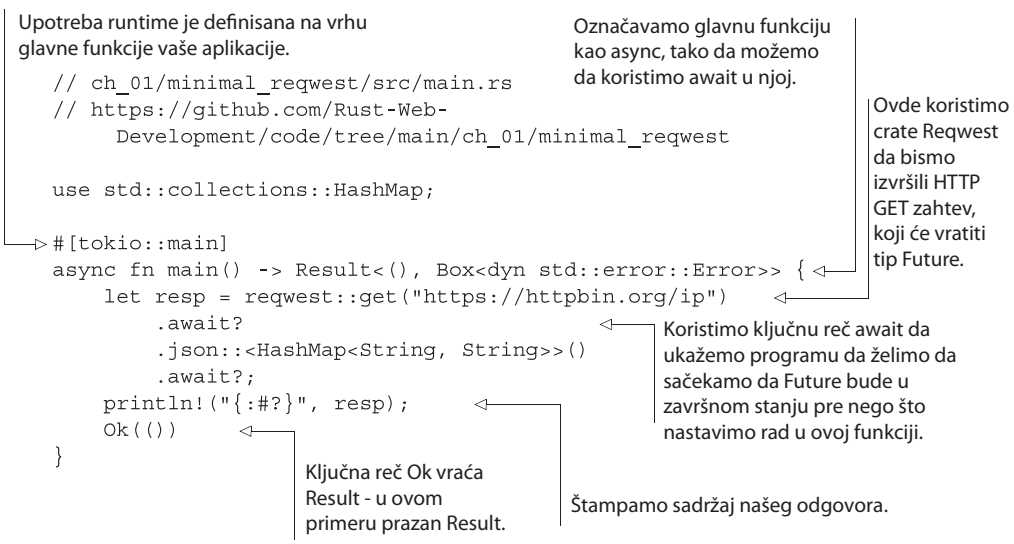
Asinhroni runtimesi u Rustu

Runtime nije isto što i Java runtime ili Go sakupljanje „smeća“. Tokom kompajliranja runtime će biti kompajliran u „statički“ kod. Svaka biblioteka ili radni okvir koji omogućavaju neki oblik asinhronog koda će izabrati runtime za izradu programa. Posao runtimea je da izabere svoj način rukovanja programskim nitima i upravljanja radom (zadacima) „iza kulisa“.

Zbog toga je moguće da na kraju imate više runtimea. Na primer, ako izaberete veb radni okvir koji diktira tokio runtime i ima pomoćnu biblioteku za izvršavanje asinhronih HTTP zahteva, koja je nadgrađena na drugi runtime, onda u osnovi imate dva runtimea kompajlirana u vašoj binarnoj datoteci. Da li će doći do neželjenih efekata zavisi od dizajna vaše aplikacije.

Kako asinhroni runtimesi izgledaju u Rustu? Hajde da pogledamo u listingu 1.12 kako asinhrono preuzimanje veb sajta izgleda u Rustu. Ovaj kod možete naći u GitHub spremištu ove knjige (https://github.com/Rust-Web-Development/code). Ovde ne ulazimo u detalje – oni će biti razmotreni u Poglavlju 2 i drugim poglavljima, ali sledeći isečak koda bi trebalo da vam pomogne da shvatite kako asinhroni kod izgleda u Rustu. Da bi ovaj isečak koda funkcionisao, potrebno je da dodate eksterni crate Reqwest vašem projektu (pomoću datoteke Cargo.toml).

Listing 1.12 Slanje HTTP GET zahteva asinhrono u Rustu




```
// ch_01/minimal_reqwest/src/main.rs
// https://github.com/Rust-Web-
     Development/code/tree/main/ch_01/minimal_reqwest

use std::collections::HashMap;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let resp = reqwest::get("https://httpbin.org/ip")
        .await?
        .json::<HashMap<String, String>>()
        .await?;
    println!("{:#?}", resp);
    Ok(())
}
```

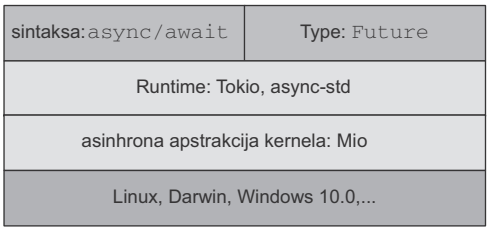
Označavamo glavnu funkciju koristeći #[tokio::main]. Tokio je asinhroni runtime (ili raspored niti) koji ovde koristimo. Sam Tokio koristi (pomoću drugog cratea koji se zove Mio) asinhroni Kernel aplikacijski programski interfejs (API) iz operativnog sistema koji smo ranije naveli.

Možemo označiti funkciju kao `async` pomoću standardne Rust sintakse i čekati (`await`) future (http://mng.bz/5mv1). *Future* u Rustu je trejt koji se može primeniti na tipove. Ovaj trejt diktira da implementacija mora da ima tip `Output` (koji ukazuje šta future vraća kada se obrada završi) i funkciju koja se zove `poll` (koju runtime može pozvati da izvršava zadatke u futureu). Kada koristite veb radne okvire, možda nikada nećete „taknuti“ stvarnu implementaciju futurea, ali je važno da razumete osnovni koncept, tako da vam poruke kompajlera i implementacije radnog okvira budu razumljive.

Za razliku od drugih jezika, u Rustu posao u Futuresu počinje tek kada se preda runtimeu i aktivno pokrene. Asihrona funkcija vraća tip `Future`, a pozivalac funkcije je odgovoran za prosleđivanje tog futurea runtimeu koji će nešto uraditi pomoću njega.

To znači da programer treba da doda `.await` u pozivu funkcije, pri čemu ukazuje da runtime treba da bude izvršen. Postoje i drugi načini za pokretanje Futuresa (na primer, pomoću tokio makroa `join!`: https://docs.rs/tokio/latest/tokio/macro.join.html), koje ćemo koristiti u Poglavlju 8.

Na slici 1.5 pokazano je šta Rust može da obezbedi u asinhronom programiranju. U Poglavlju 2 će biti više reči o ovoj temi. Rust obezbeđuje sintaksu i tip u svojoj standardnoj biblioteci, a implementacija apstrakcije runtimea i kernela je prepuštena zajednici.



Slika 1.5 Rust obezbeđuje sintaksu i tip za asinhrono programiranje, ali apstrakcije runtimea i kernela se implementiraju izvan osnovnog jezika

O odabiru runtimea se, u osnovi, brine veb radni okvir, koji ćete kasnije da izaberete**.** On diktira runtime od kojeg zavisi.

U listingu 1.13 je prikazana minimalna radna veb aplikacija sa veb radnim okvirom Warp, koji je naš kasniji izbor za ovu knjigu. Ovaj veb radni okvir je nadgrađen na tokio runtime, pa takođe moramo da dodamo tokio našem projektu (pomoću datoteke Cargo.toml, kao što je prikazano u listingu 1.14).

Listing 1.13 Minimalni radni HTTP server u Rustu sa Warpom

```
// ch_01/minimal_warp/src/main.rs
// https://github.com/Rust-Web-Development/code/tree/main/ch_01/minimal-warp

use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

Listing 1.14 Datoteka Cargo.toml za minimalni Warp primer

```
[package]
name = "minimal-warp"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
tokio = { version = "1.2", features = ["full"] }
warp = "0.3"
```

Sintaksa bi vam sada mogla izgledati nepoznato, ali možete videti da je sve apstrahovano iza našeg radnog okvira, a jedini znak runtimea je u liniji iznad naše glavne funkcije. Više reči o radnom okviru koji smo izabrali i kako/zašto ga biramo biće u Poglavlju 2.

Možda se pitate, ako Rust nema standardni runtime za rukovanje asinhronim kodom, niti ima dodat HTTP u standardnu biblioteku, zašto bi imalo smisla pisati veb servise pomoću njega. Rust se svodi na jezičke funkcije i zajednicu.

Možete tvrditi da standardne implementacije HTTP-a i runtime ne čine Rust prilagodljivijim za buduće potrebe, jer zajednica uvek može da se uključi i poboljša nešto ili ponudi različita rešenja za različite probleme. Bezbednost tipa, brzina i ispravnost jezika igraju ključnu ulogu u okruženju u kojem se bavite asinhronim radom u vašoj aplikaciji i upravljate ogromnom količinom saobraćaja. Brz i bezbedan jezik će se dugoročno isplatiti.

1.4 Održavanje Rust aplikacija

Rust će vam biti od pomoći i u drugim temama. Na primer, isporučuje se sa ugrađenom dokumentacijom. Kod ugrađen u vaše komentare će takođe biti testiran, tako da možete biti sigurni da nikada nećete imati zastarele primere koda. „Menadžer“ paketa Cargo ima komandu za generisanje dokumentacije iz vaših komentara koda, koja se može pretraživati lokalno i koja će podrazumevano biti izrađena prilikom izvoza biblioteke u registar paketa crates.io. Kod koji je ugrađen u vašu dokumentaciju za kod neće se pojaviti samo u unapred generisanim HTML dokumentima, već će biti testiran, tako da možete biti sigurni da nikada nečete imati zastarele primere koda.

Osim dokumentovanja, modularizacija vaše baze koda pomaže pri grupisanju delova koda ili izdvajanju koda za višekratnu upotrebu u zaseban crate. Rust čini modularizaciju prilično lakom, tako što koristi odeljak dependency u datoteci Cargo.toml za uključivanje lokalnih biblioteka iz zvaničnog registra crates.io ili iz bilo koje druge lokacije po vašoj želji.

Rust podrazumevano takođe podržava i testiranje. Nisu vam potrebni dodatni crate ili druge pomoćne alatke za kreiranje i pokretanje testova. Sve ove ugrađene i standardizovane funkcije uklanjaju mnoga trvenja i rasprave u vašem timu, pa se možete fokusirati na pisanje i implementaciju koda, umesto da uvek tražite novu alatku za pisanje dokumentacije ili testova.

Ako vam zatreba pomoć kasnije, a nemate profesionalca u svom timu, možete koristiti desetine Discord kanala, Reddit foruma i StackOverflow tagova za traženje smernica. Na primer, pomoć za runtime tokio i veb radni okvir warp možete pronaći na tokio Discord serveru (https://discord.com/invite/tokio), koji ima kanal za svaku alatku, pa možete da zatražite pomoć ili pročitate komentare drugih ljudi kako biste saznali više informacija o alatkama koje su vam pri ruci.

Rezime

- Rust je sistemski programski jezik koji proizvodi binarne datoteke.
- Rust se isporučuje sa preciznim kompajlerom koji sadrži korisne poruke o greškama, tako da možete lako uočiti greške i poboljšanja.
- Alatke Rusta se isporučuju sa samom Rust instalacijom ili postoje zvanične preporuke, tako da ćete uštedeti vreme, jer nećete stalno istraživati, diskutovati i učiti o novim alatkama.
- Morate da izaberete runtime kada pišete asinhroni kod (kako izabrati runtime biće razmotreno u Poglavlju 2), pošto Rust ne sadrži runtime kao Go ili NodeJS.
- Veb radni okviri su nadgrađeni na runtimeu, tako da vaš izbor runtimea zavisi od radnog okvira koji ćete da odaberete kasnije.
- Brzina, bezbednost i ispravnost Rusta će vam biti od pomoći u velikoj meri kada održavate i male i velike veb usluge i baze kodova.
- Dokumentacija i testiranje su ugrađeni u sam jezik, što ga čini još stabilnijim pri održavanju vašeg koda.

2

POSTAVLJANJE OSNOVE

Ovim poglavljem obuhvaćene su sledeće teme:

- upoznavanje Rust tipova
- razumevanje Rust sistema vlasništva
- implementacija prilagođenog ponašanja na vaše tipove
- razumevanje gradivnih blokova asinhronog eko-sistema
- odabir biblioteka nezavisnih proizvođača za kreiranje veb usluga pomoću Rusta
- podešavanje osnovnog radnog veb servisa pomoću Rusta

U prvom poglavlju ste videli prilično dobar pregled funkcija koje se isporučuju sa Rustom i alatke koje treba da dodate da biste mogli da kreirate veb usluge pomoću Rusta. U ovom poglavlju ćemo elaborirati te teme. U prvom delu će biti detaljno objašnjeno kako da koristite jezik za kreiranje vaših tipova i funkcija, a u drugom delu ćemo dodati veb server, tako da možete da pošaljete vaš prvi odgovor korisniku.

Kao što je ranije pomenuto, najbolje bi bilo da pročitate poglavlja iz knjige „The Rust Programming Language" (https://doc.rust-lang.org/book/). U ovom poglavlju ćete učiti o konceptima koji su potrebni za dovršavanje ove knjige, tako da biste mogli da pročitate ovo poglavlje bez ikakvog prethodnog znanja. Međutim, ponovo preporučujem da bar pregledate prvih šest poglavlja knjige „The Rust Programming Language" da biste imali odgovarajuću osnovu za sam jezik.

U ovoj knjizi ćemo kreirati primer Q&A veb usluge, u kojoj korisnici mogu postavljati pitanja i odgovarati na njih. Kreiraćemo Representational State Transfer (REST API) i imaćemo pokrenutu, primenjenu i testiranu uslugu do kraja knjige. Sačuvaćemo nova pitanja, ažuriraćemo ih ili brisati i moći ćemo da objavljujemo odgovore na ta pitanja. Kasnije u knjizi ćete shvatiti kako da se autentifikujete za ovu veb uslugu i kako da ispravno izvršite testiranje.

U celoj knjizi se fokusiramo na Rust aspekte veb usluge. Trebalo bi da možete da razumete ovu knjigu, čak i ako izaberete drugi veb radni okvir, bazu podataka ili drugi crate koji vam mogu biti od koristi. Cilj je pokazati jedan način implementacije.

Važno je da imate na umu da je Rust dvostruk. Ne morate da znate detalje nižeg nivoa operativnih sistema. Međutim, profitiraćete ako znate nešto više o unutrašnjem radu računara. Ne bi trebalo samo da naučite drugu sintaksu, već da poboljšate svoje opšte znanje o sistemima i veb uslugama.

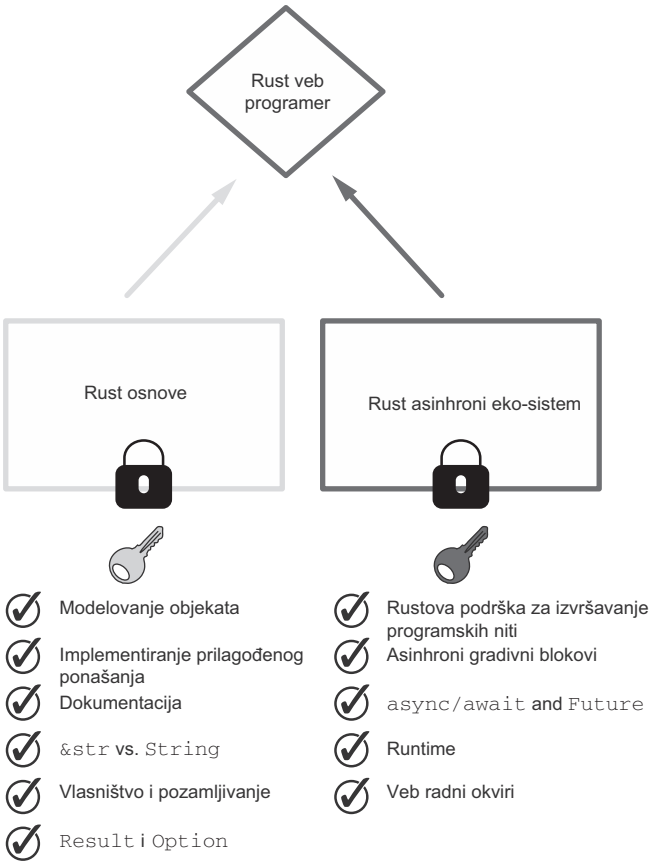
U ovom poglavlju želimo da postavimo osnovu. Na slici 2.1 prikazane su teme sledećih odeljaka. Kada razumete oblasti Rusta na koje ćete nailaziti više puta, možete odvojiti vreme da ih detaljno naučite. Isto važi i za veb uslugu. Ako ikada naiđete na probleme performansi ili niste zadovoljni svojim izborom radnog okvira, znaćete kako da u Rust eko-sistemu izaberete cratese koji više pogoduju vašim potrebama.

Da biste izradu primera iz ove knjige učinili vrednom vežbom, trebalo bi da sada naučite ove osnove, tako da možete da profitirate od njih za mnogo godina. To takođe znači da je ovo poslednje poglavlje knjige u kojem će biti više reči o osnovama, nego o kodu, ali ćemo ubrzati tempo u sledećem poglavlju.

Počinjemo da implementiramo našu veb uslugu pomoću structova i na kraju poglavlja imaćemo osnovni veb server koji funkcioniše, a usput treba da objasnimo koncepte. U sledećim poglavljima ćemo ubrzati tempo, pri čemu ćemo pretpostaviti da znate osnove Rust jezika i eko-sistema.

2.1 Rust pravilnik

Rust je složen jezik, ali ne morate da znate sve detalje na početku, pa čak ni tokom većeg dela projekta. Kompajler i druge alatke (na primer, Clippy, koji ćemo predstaviti u Poglavlju 5) mnogo će pomoći da kod bude čist i lep. Zbog toga, nećemo razmotriti svaki aspekt Rust jezika, ali ćete sami moći da istražitete teme kada naiđete na njih.



Slika 2.1 Plan „puta“ u ovom poglavlju za „otključavanje“ vaših mogućnosti da postanete Rust veb programer

Da biste samouvereno radili nešto u Rustu, morate da naučite sledeće veštine:

- kako potražiti tipove i ponašanja u zvaničnoj Rust dokumentaciji docs.rs
- brzo ponovno pregledanje grešaka ili problema sa kojima se suočavate
- kako funkcioniše Rust sistem vlasništva
- šta su makroi, kako ih uočiti i koristiti
- kreiranje svojih tipova pomoću structova i implementiranje ponašanja pomoću `impla`
- kako implementirati trejtove i makroe na postojeće tipove
- funkcionalni Rust sa opcijama (Options) i rezultatima (Results)

U ovom poglavlju ćemo razmotriti osnove svih ovih veština, a u narednim poglavljima ćemo vežbati i „zaroniti” u detalje. Važno je znati da čak i složeni problemi ili izazovi sa kojima se suočavate usput mogu biti rešeni pomoću prethodno navedenih veština. Potrebni su samo iskustvo i odgovarajući način razmišljanja da biste prevazišli te probleme.

Pošto je Rust striktno tipizirani jezik, potrebno je da uložite malo više truda na početku vašeg programa. Brzo preuzimanje JSON datoteke sa druge krajnje tačke ili modelovanje jednostavnog programa može potrajati malo duže ako ne poznajete postojeće tipove i ako ne znate da rukujete nepoznatim vrednostima koje prvo morate da ispitate.

2.1.1 Modelujte svoje resurse pomoću structova

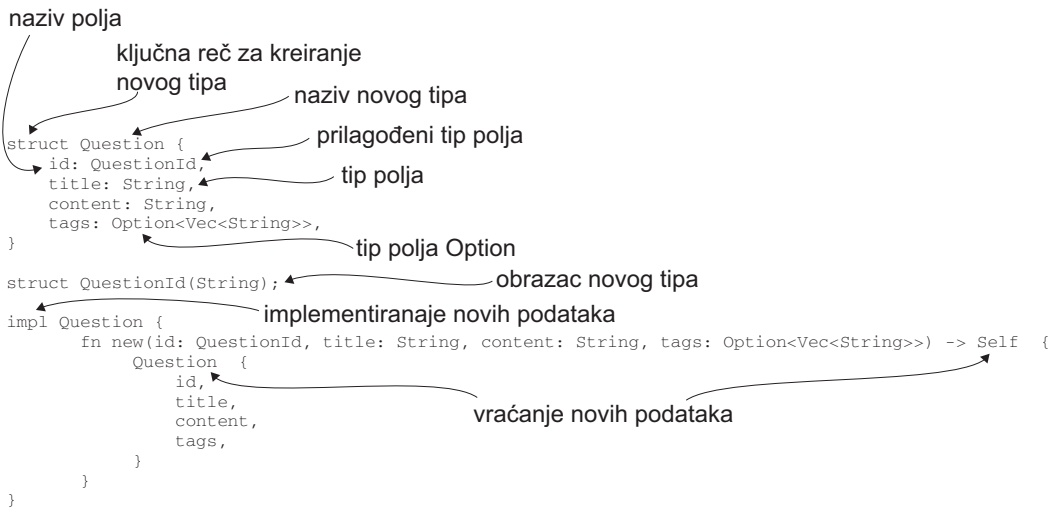
Želimo da kreiramo RESTful API, pa obezbeđujemo rute za kreiranje, čitanje, ažuriranje i brisanje (CRUD) resursa. Stoga je prvi korak da razmislimo koje modele ili tipove koristimo u našoj veb usluzi.

Mudro je da prvo mapiramo naše najmanje radne aplikacije. To uključuje prilagođene tipove podataka koje želite da implementirate i njihovo ponašanje (metode). Za našu aplikaciju potrebni su:

- korisnici
- pitanja
- odgovori

Korisnici mogu da se registruju i prijave na sistem i da postavljaju i pregledaju pitanja i odgovore na ta pitanja. Na korisnike ćemo se fokusirati kasnije u ovoj knjizi, kada bude reči o autentikaciji i autorizaciji aplikacije. Za sada, implementiramo svaku rutu bez provere lozinki ili korisničkih ID-ova.

Na slici 2.2 je prikazano šta je potrebno za kreiranje i implementaciju tipova u Rustu. Prvo ćemo implementirati tip `Question` i razjasniti sve dileme na koje nailazimo i tipove koji su nam potrebni na tom putu. Svoj tip kreirate korišćenjem ključne reči `struct`, a dodavanjem funkcije u njoj koristite ono što je poznato kao blok `impl` da biste tipu dodali ponašanje.



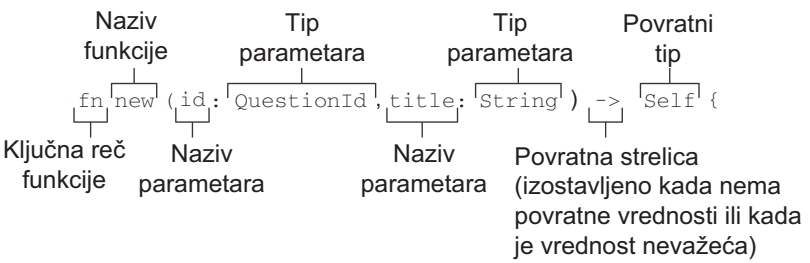
Slika 2.2 Prilagođeni tipovi se mogu kreirati pomoću ključne reči struct, a dodavanje prilagođenih metoda se izvršava pomoću bloka `impl`.

Prvo ćemo kreirati naša pitanja i odgovore i razmotriti osnovni „životni ciklus" kreiranja prilagođenog tipa u Rustu.

Listing 2.1 Kreiranje i implementiranje tipa Question

```
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}
struct QuestionId(String);

impl Question {
    fn new(
        id: QuestionId,
        title: String,
        content: String,
        tags: Option<Vec<String>>
     ) -> Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}
```

Kreiramo naš tip `Question`, kao što je prikazano u listingu 2.1, a zatim koristimo `id` da bismo kasnije razlikovali različita pitanja (za sada kreiramo ID ručno, a na automatski generisani ID preći ćemo kasnije u knjizi). Svako pitanje ima `title` i stvarno pitanje u `contentu`. Takođe koristimo `tags` za grupisanje određenih pitanja. Objasnićemo šta `Option` znači u sledećem odeljku 2.12. Funkcije izgledaju veoma slične funkcijama u drugim programskim jezicima. Na slici 2.3 su prikazani potpis funkcije u Rustu i značenje svakog elementa.



Slika 2.3 Detaljan potpis funkcije Rust prilikom vraćanja vrednosti

Uvek morate da izvršite ove korake da biste kreirali prilagođene podatke u Rustu:

1. Kreirajte novi struct, koristeći `structQuestion {…}`.
2. Dodajte polja sa njihovim tipovima u structu.

3. Rust nema podrazumevano naziv za konstruktor, pa je najbolja praksa da koristite metod koji se zove `new`.
4. Vi dodajete ponašanje svom prilagođenom tipu pomoću bloka `impl`.
5. Vratite `Self` ili `Question` da biste instancirali novi objekat ovog tipa.

Takođe koristimo *obrazac New Type* (http://mng.bz/o5Zr), u kojem ne koristimo jednostavno string za naš ID pitanja, već enkapsuliramo string u structu koji se zove `QuestionId`, umesto da samo prosledimo string. Kada posedujemo prilagođeni tip, prenosi se idea koju kompajler može da primeni automatski.

Za sada se kreiranje prilagođenih tipova čini nepotrebnim, ali u većim aplikacijama oni daju parametrima više značenja. Prilagođene tipove možete zamisliti kao ID-ove. Možete imati funkcije koje prihvataju ID `Question` unutar aplikacije (kao što je naša) i koje takođe rukuju ID-ovima `Answer`. Enkapsuliranje primitivnih tipova u structove daje značenje tim tipovima, a kasnije i fleksibilnost u njihovom instanciranju.

2.1.2 Razumevanje opcija (Options)

Opcije su važne u Rustu. One omogućavaju da se uverimo da nemamo `null` tamo gde bismo očekivali vrednost. Pomoću enuma `Option` uvek možemo da proverimo da li je zadata vrednost prisutna ili nije i možemo da obradimo slučaj kada vrednost nije prisutna. Kada nam je pri ruci enum `Option`, kompajler će se pobrinuti da uvek obradimo svaki slučaj (`Some` ili `None`). To takođe omogućava da deklarišemo polja koja nisu obavezna, tako da ne moramo, kada kreiramo novo pitanje, da navedemo listu tagova, ali možemo da je navedemo ako želimo.

Kasnije, kada budete koristili eksterne API-e i budete želeli da primate podatke, od velike pomoći će biti ako označite određena polja kao opciona. Pošto je Rust striktno tipiziran, kad god očekujete da će polje za tip biti postavljeno, a nije postavljeno, program će prikazati grešku. Podrazumevano, sva polja struct su obavezna kada su navedena. Zbog toga, morate ručno da proverite da li su polja koja nisu postavljena označena kao `Option<Type>`.

> **Rust Playground**
>
> Važan deo prilikom učenja Rusta je korišćenje dostupnih alatki za brzo testiranje ideja. Veb sajt Rust Playground (https://play.rust-lang.org/) obezbeđuje Rust kompajler i najčešće korišćene cratese za brzo izvršavanje iteracije nad manjim programima. Stoga, ne morate uvek da kreirate lokalni Rust projekat da biste se bavili određenom temom.

Uobičajeni način provere da li se u `Optionu` krije vrednost je korišćenje ključne reči `match`. U listingu 2.2, koji možete kopirati i „nalepiti" i pokrenuti u prethodno opisanom Rust Playgroundu (ili kliknite na linkhttp://mng.bz/neZg), prikazano je kako možete da primenite blok `match` na opcionu vrednost.

Obrazac match u Rustu

Programeri početnici često posmatraju match kao na alternativnu ključnu reč za `switch`. Međutim, obrazac `match` u Rustu je mnogo moćniji. Poglavlje 18 knjige „The Rust Programming Language" sadrži više detalja o matchu (http://mng.bz/AVlp).

Obrazac `match` takođe omogućava destrukturiranje structova (http://mng.bz/49na), enumova (http://mng.bz/Qnww) i još mnogo čega. Ovaj moćni mehanizam čini vaš kod čitljivijim i koristi Rustov moćan sistem tipova da izrazi više značenja u vašoj bazi koda koja kompajler može da primeni.

Listing 2.2 Upotreba ključne reči match za korišćenje vrednosti Option

```
fn main() {
    struct Book {
        title: String,
        isbn: Option<String>,
    }

    let book = Book {
        title: "Great book".to_string(),
        isbn: Some(String::from("1-123-456"))
    };

    match book.isbn {
        Some(i) => println!(
            "The ISBN of the book: {} is: {}",
            book.title,
            i
        ),
        None => println!("We don't know the ISBN of the book"),
    }
}
```

Standardna biblioteka takođe obezbeđuje veliki broj metoda i trejtova koje možete koristiti za vrednost `Option`. Na primer, metod `book.isbn.is_some` će vratiti `true` ako vrednost postoji, a `false` ako vrednost ne postoji.

2.1.3 Korišćenje dokumentacije za rešavanje grešaka

Ovo jednostavno podešavanje će omogućiti da razumete osnovno Rust ponašanje i mnoge funkcije, pa ćemo sada da pokušamo da kreiramo novo pitanje u našem programu pomoću konstruktora koji smo prethodno implementirali u structu `Question` (da biste otklonili greške u Rust Playgroundu, koristite linkhttp://mng.bz/yaNG). Ovaj kod neće kompajlirati i generisati neke greške (listing 2.4), koje ćemo zajedno otkloniti odmah nakon toga.

Listing 2.3 Kreiranje i štampanje primera pitanja

```
// ch02/src/main.rs

struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

struct QuestionId(String);

impl Question {
    fn new(
        id: QuestionId,
        title: String,
        content: String,
        tags: Option<Vec<String>>
     ) -> Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}

fn main() {
    let question = Question::new(
        "1",
        "First Question",
        "Content of question",
        ["faq"]
    );
    println!("{}", question);
}
```

Dodajte ovaj isečak koda na kraj datoteke main.rs i pokrenite ga. Obratite pažnju da koristimo duple dvotačke (::) da bismo pozvali metod `new` u `Questionu`. Rust ima dva načina primene funkcija na tipove:

- korišćenje pridruženih funkcija
- korišćenje metoda

Pridružene funkcije ne prihvataju `&self` kao parametar i pozivaju se pomoću duple dvotačke (::). One su, otprilike, ekvivalentne „statičkim" funkcijama u drugim programskim jezicima. Uprkos svom nazivu („pridružene"), funkcije nisu povezane sa određenom instancom. *Metodi* prihvataju `&self` i pozivaju se jednostavno pomoću tačke (**.**). Na slici 2.4 prikazana je razlika u načinu implementacije i pozivanja svake od ove dve opcije.

```
impl Question {
   fn new(id: QuestionId, title: String, …) -> Self {
       Question {
           id,
           title,
           content,
           tags,
       }
    }

    fn update_title(&self, new_title: String) -> Self {
        Question::new(self.id, new_title, self.content, self.tags)
    }
}
```

```
let q = Question::new(QuestionId("1".to_string()), "title".to_string(), …);
```

```
q.update_title("better_title".to_string());
```

Slika 2.4 Pridružena funkcija (`new()` na vrhu) ne prihvata parametar `&self` i poziva se pomoću duple dvotačke (`::`). Metod (`update_title` na dnu) prihvata parameter `&self` i poziva se pomoću tačke (`.`). Pozivanje funkcije unutar bloka `impl` moguće je pomoću naziva bloka (u našem primeru pomoću naziva bloka `Question::new(...)`).

Možemo pokrenuti našu aplikaciju pomoću komande `cargorun` na terminalu. To će, međutim, vratiti dugačku listu grešaka. Te greške neće nestati, čak i ako pišete Rust u dužem vremenskom periodu. Kompajler je veoma precizan, pa morate početi da se navikavate na ovo mnoštvo crvenih grešaka iz kompajlera.

Rust želi da proizvede bezbedan i ispravan kod i stoga bira šta dozvoljava da se kompajlira, pa ćete tako naučiti kako da pišete kod i biće prikazane odlične poruke o grešci, tako da možete da vidite gde grešite.

Listing 2.4 Poruke o grešci nakon pokušaja kompajliranja našeg koda do sada

```
error[E0308]: arguments to this function are incorrect
  --> src/main.rs:27:20
   |
27 |     let question = Question::new(
   |                    ^^^^^^^^^^^^^
28 |         "1",
   |         --- expected struct `QuestionId`, found `&str`
29 |         "First Question",
   |         ---------------- expected struct `String`, found `&str`
30 |         "Content of question",
   |         --------------------- expected struct `String`, found `&str`
31 |         ["faq"],
   |         ------- expected enum `Option`, found array `[&str; 1]`
   |
   = note: expected enum `Option<Vec<String>>`
              found array `[&str; 1]`
note: associated function defined here
  --> src/main.rs:11:8
   |
11 |     fn new(
   |        ^^^
12 |         id: QuestionId,
   |         --------------
```

Rust kompajler nam pokazuje tačno gde je i u čemu je problem.

Stavljanje teksta između dvostrukih navodnika ne označava String, već &str.

Umesto niza, kompajler očekuje enum Option za naše tagove.

```
13 |         title: String,
   |         -------------
14 |         content: String,
   |         ---------------
15 |         tags: Option<Vec<String>>,
   |         ------------------------
help: try using a conversion method
   |
29 |         "First Question".to_string(),
   |                          ++++++++++++
help: try using a conversion method
   |
30 |         "Content of question".to_string(),
   |                               ++++++++++++

error[E0277]: `Question` doesn't implement `std::fmt::Display`
  --> src/main.rs:33:20
   |
33 |     println!("{}", question);
   |                    ^^^^^^^^ `Question` cannot
       be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Question`
   = note: in format strings you may be able to use `{:?}` (or {:#?}
           for pretty-print) instead
   = note: this error originates in the macro `$crate::format_args_nl`
           (in Nightly builds, run with -Z macro-backtrace for more info)

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: aborting due to 5 previous errors; 1 warning emitted
```

Ne možemo da štampamo pitanje u konzoli.

Koristićete ove greške da biste naučili više o Rust jeziku i njegovim funkcijama, tako da budete spremni za izradu stabilne veb aplikacije. Videćete da neke greške imaju još dva problema, dok su druge greške samo ponovljena greška koju smo načinili i nadamo se da će nestati ispravljanjem prethodnih grešaka.

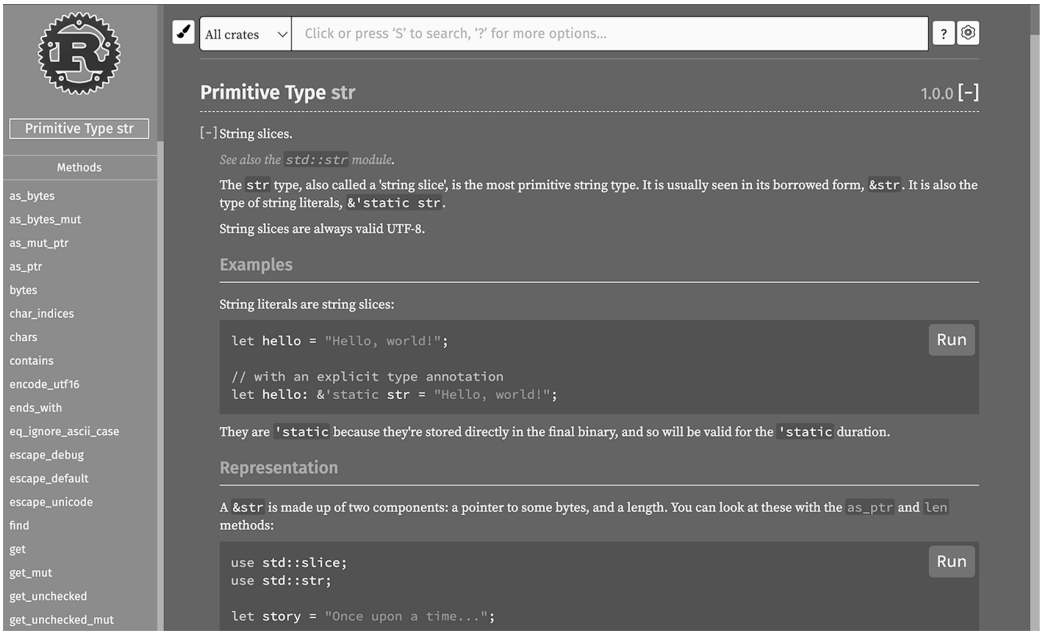
Najbolje je da uvek počnete da rešavate prvu grešku koju vidite, jer bi njeno prenebregavanje moglo da bude razlog zašto se pojavljuju i kasnije greške. Dakle, hajde da vidimo kako bismo mogli da rešimo prvi problem.

Listing 2.5 Naša prva greška kompajlera

```
    --> src/main.rs:27:20
     |
27   |     let question = Question::new(
     |                    ^^^^^^^^^^^^^
28   |         "1",
     |         --- expected struct `QuestionId`, found `&str`
```

Ova greška nam pokazuje dva problema. Prvo, moramo da prosledimo naš prilagođeni tip `QuestionId`, a ne `&str`. I posmatrajući našu definiciju structa, videćemo da moramo enkapsulirati `String`, a ne `&str`.

To nam daje priliku da otvorimo dokumentaciju `&str` (https://doc.rust-lang.org/std/primitive.str.html) i vidimo šta možemo da uradimo da bismo rešili problem. Prvo otvaranje Rust dokumentacije može biti zastrašujuće, ali ne bojte se - samo je potrebno vreme da se naviknete na nju (pogledajte sliku 2.5).



Slika 2.5 Iako je Rust dokumentacija veoma kompleksna, kroz nju se možemo brzo kretati, mada sadrži mnoštvo informacija.

Dokumentacija sadrži glavni prozor i bočnu traku sa leve strane. Glavni prozor obično predstavlja tip koji koristite, a bočna traka sadrži detalje implementacije, metode i trejtove koji se primenjuju na ovom tipu. Važno je da razmotrimo:

- metode
- implementacije trejtova
- implementacije automatskih trejtova
- opšte implementacije

Pretražujte dokumentaciju lokalno i oflajn

Ako ste u vozu, avionu ili samo želite da imate Rust dokumentaciju lokalno, možete instalirati komponentu docs pomoću rustupa:

```
$ rustup component add rust-docs
```

Nakon toga, možete otvoriti dokumentaciju iz standardne biblioteke u vašem podrazumevanom pregledaču pomoću sledeće komande:

```
$ rustup doc --std
```

Takođe možete da generišete dokumentaciju iz vaše baze kodova, koja uključuje i sve Cargo zavisnosti, pomoću sledeće komande:

```
$ cargo doc --open
```

Ona će takođe uključiti strukture i funkcije koje ste definisali, čak i ako niste eksplicitno kreirali dokumentaciju za to.

Pre nego što krenemo dalje i izaberemo metod da bismo transformisali naš `&str` u `String`, moramo da razlučimo u čemu je razlika između ova dva tipa i zašto Rust njima rukuje drugačije.

2.1.4 Rukovanje Stringovima u Rustu

Glavna razlika između `Stringa` (http://mng.bz/M0w7) i `&stra` (https://doc.rust-lang.org/std/primitive.str.html) u Rustu je u činjenici da je `String` promenljive veličine. *String je* kolekcija bajtova, koja je implementirana kao vektor. Možemo pogledati definiciju u izvornom kodu.

Listing 2.6 Definicija Stringa u standardnoj biblioteci

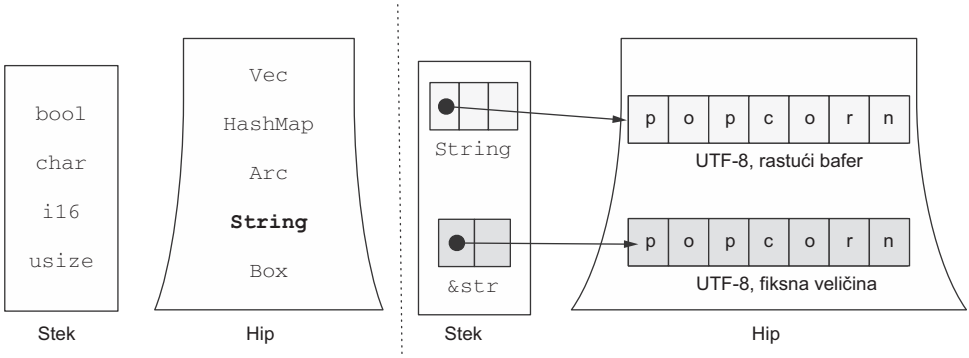
```
// Source: https://doc.rust-lang.org/src/alloc/string.rs.html#294-296

pub struct String {
    vec: Vec<u8>,
}
```

Stringovi se kreiraju pomoću funkcije `String::from("popcorn");` i mogu se izmeniti nakon što su kreirani. Vidimo da se ispod `Stringa` nalazi vektor, pa znači da ga možemo ukloniti i umetnuti vrednosti `u8` u ovaj vektor po želji.

`&str` (literal stringa) je reprezentacija vrednosti `u8` (teksta), koju ne možemo da menjamo. Možete ga videti kao prozor fiksne veličine za neki osnovni niz znakova. Objasnićemo Rustov koncept vlasništva u sledećem odeljku, ali sada je važno razumeti da, ako imamo `String`, „posedujemo" ovaj deo memorije i možemo ga modifikovati.

Kada koristimo `&str`, bavimo se pokazivačem na prostor u memoriji i dozvoljeno nam je da čitamo, ali ne i da menjamo sadržaj memorije.To čini korišćenje `&stra` efikasnijim za memoriju. Opšte pravilo: ako kreirate funkcije, koristite `&str` kao tip parametra kada samo želite da pročitate `String`. Ako želite da ga „posedujete" i modifikujete, koristite `String`.

Kao što vidite na slici 2.6, oba teksta se nalaze u hipu (dinamičkoj memoriji), ali imaju drugačiji pokazivač dodeljen u steku. Ne morate detaljno da razumete koncept hipa i steka, ali, da biste bolje razumeli greške kompajlera u budućnosti, biće vam od koristi da upoznate koncept. U sledećem izdvojenom tekstu (poređenje steka i hipa) objašnjeni su glavni koncepti.



Slika 2.6 Primitivni tipovi se čuvaju u steku u Rustu, dok se složeniji tipovi čuvaju u hipu; `String` i `&str` ukazuju na složenije tipove podataka (kolekciju UTF-8 vrednosti). Dok `&str` ima podebljani pokazivač (adresu i polje dužine) koji pokazuje adresu u hipu, `String` pokazivač ima ne samo adresu i polje dužine, već i polje kapaciteta.

Poređenje steka i hipa

Operativni sistemi dodeljuju memoriju za sav rad koji izvršavaju pomoću promenljivih i funkcija. Operativni sistem mora da čuva i poziva funkcije i da obrađuje i ponovo koristi podatke. Postoje dva koncepta za ovaj rad: stek i hip.

Stek obično kontroliše program, a svaka programska nit ima svoj stek koji čuva jednostavne instrukcije ili promenljive. U osnovi, sve što ima fiksnu veličinu može da bude sačuvano u steku.

Hip je jedinstven (iako može biti više hipova) i rad u njemu je skuplji. Podaci nemaju fiksnu veličinu i mogu biti podeljeni na više blokova, tako da čitanje može malo duže potrajati.

Šta je str?

`&str` bez znaka ampersenda je samo `str`, tj. stvarni tip podataka koji koristimo. Međutim, `str` je nepromenljiva sekvenca UTF-8 bajtova koja nema fiksnu dužinu, pa možemo njime samo da rukujemo iza pokazivača, pošto je njegova veličina nepoznata (pogledajte odličan StackOverflow odgovor na pitanje „Šta je str?" na linku http://mng.bz/aP9z).

Ili da citiramo rečenicu iz Rust dokumentacije: „Tip `str`, koji se takođe naziva ‚string slice', predstavlja najprimitivniji tip stringa. Obično ga možete videti u pozajmljenoj formi `&str`" (https://doc.rust-lang.org/std/primitive.str.html). U odeljku 2.1.5 su razmotreni detalji o pozajmljivanju u Rustu.

Kratak rezime:

- Ako treba da „posedujete" i modifikujete tekst, kreirajte `String`.
- Koristite `&str` kada vam je potreban samo prikaz osnovnog teksta.

- Kada kreirate nove tipove podataka pomoću structa, obično kreirajte tipove polja `String`.
- Kada koristite tekst u funkciji, obično koristite `&str`.

2.1.5 Premeštanje, pozajmljivanje i vlasništvo

Jednostavno poređenje `Stringa` i `&stra` je još ozbiljnije i dotiče jedan od Rustovih glavnih koncepata: vlasništvo. Jednostavno rečeno, Rust želi da bezbedno upravlja memorijom bez sakupljača „smeća" ili bez mnogo opreza programera.

Svaki kompjuterski program koristi memoriju, a posao sakupljača „smeća" je da čisti i osigura da ni jedna promenljiva ne može da ukaže na praznu vrednost ili programer mora da razmišlja o prikupljanju „smeća". Rust ne bira ni jedno, ni drugo i uvodi potpuno novi koncept.

Sledeće listinge kodova (2.7–2.9) je najbolje pokrenuti u Rust Playgroundu:http://mng.bz/gRml. Možete isprobati različite kombinacije i videti da li možete sami da ispravite greške.

Listing 2.7 Dodeljivanje &str vrednosti

```
fn main() {
    let x = "hello";
    let y = x;

    println!("{}", x);
}
```

Kada pokrenemo ovaj program, vidimo da se `hello` štampa u konzoli. Sada pokušajmo da dodelimo `String` vrednosti.

Listing 2.8 Dodeljivanje String vrednosti

```
fn main() {
    let x = String::from("hello");
    let y = x;

    println!("{}", x);
}
```

Pojaviće se sledeća greška:

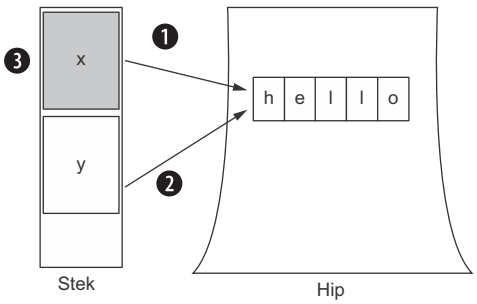
```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:5:20
  |
2 |     let x = String::from("hello");
  |         - move occurs because `x` has type `String`,
  |           which does not implement the `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |
5 |     println!("{}", x);
  |                    ^ value borrowed here after move
```

Zašto se javlja ova greška? U listingu 2.7 kreiramo novu promenljivu tipa `&str`: referenca (`&`) u string sliceu (`stru`), https://doc.rust-lang.or/std/primitive.str.html) pomoću vrednosti `hello`. Ako ovu promenljivu dodelimo novoj promenljivoj (x = y), kreiramo novi pokazivač koji ukazuje na istu adresu u memoriji. Sada imamo dva pokazivača koji ukazuju na istu osnovnu vrednost.

Kao što je prikazano na slici 2.6, ne možemo promeniti string slice nakon što je kreiran; stoga je on nepromenljiv. Sada možemo da štampamo obe promenljive, koje su važeće i ukazaće na osnovnu memoriju koja sadrži reprezentaciju reči *hello*.

Okolnosti se menjaju kada se bavimo stvarnim stringom, umesto referencom za jedno vlasništvo. U listingu 2.8 kreiran je složeni tip `String`. Rust kompajler sada primenjuje princip jednog vlasništva. Kada ponovo dodelimo `String` pomoću promenljive x = x kao ranije, prenosimo vlasništvo sa promenljive `x` na `y`.

Prenosom vlasništva vrednosti sa `x` na `y` `x` izlazi iz opsega i Rust ga interno označava kao `uninit` (https://doc.rust-lang.org/nomicon/drop-flags.html). Na slici 2.7 je prikazan ovaj koncept. Ako pokušamo da odštampamo promenljivu `x`, ona više ne postoji, pošto je vlasništvo preneto na `y`, a `x` nema vrednost.



1 `let x = String::from("hello");`

2 `let y = x;`

3 `//mark x as uninitialized`

Slika 2.7 Kada ponovo dodeljuje složene tipove novoj promenljivoj, Rust kopira informacije o pokazivaču i dodeljuje vlasništvo novoj promenljivoj. Stara promenljiva više nije potrebna i ne primenjuje se.

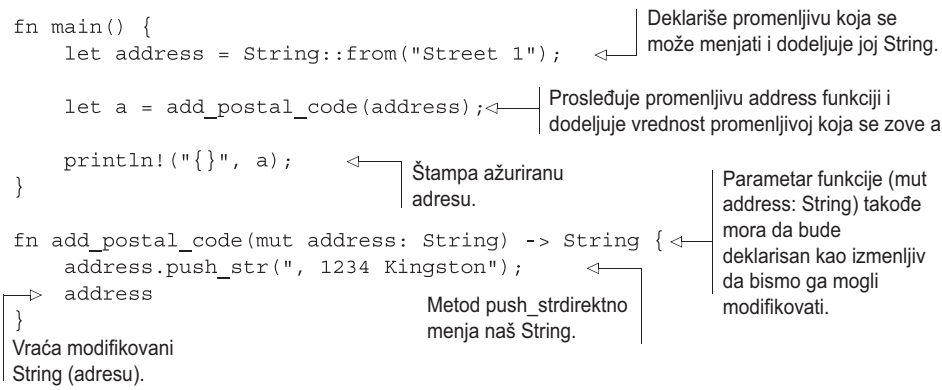
Hajde da istražimo još jednu oblast u kojoj moramo da razumemo Rustov princip vlasništva - rad u funkcijama. Prosleđivanjem promenljive funkciji predaje se funkciji vlasništvo nad osnovnim podacima. Postoje različiti načini da to uradimo u Rustu:

- premeštanjem vlasništva u funkciju i vraćanjem nove promenljive iz funkcije
- prosleđivanjem reference promenljive da bi se zadržalo vlasništvo

U listingu 2.9 je prikazan primer modifikacije objekta `String` u funkciji. Promenljivu referencu (da bismo je mogli modifikovati) prosleđujemo `Stringu` u funkciji. Funkciji je sada omogućeno da pristupi osnovnim podacima i da ih modifikuje. Kada se funkcija završi, vraćamo vlasništvo unutar funkcije `main` i možemo da štampamo `address`.

Možete se „igrati" različitim opcijama korišćenjem ovog Playground linka: http://mng.bz/epBz.

Listing 2.9 Prenošenje vlasništa funkciji



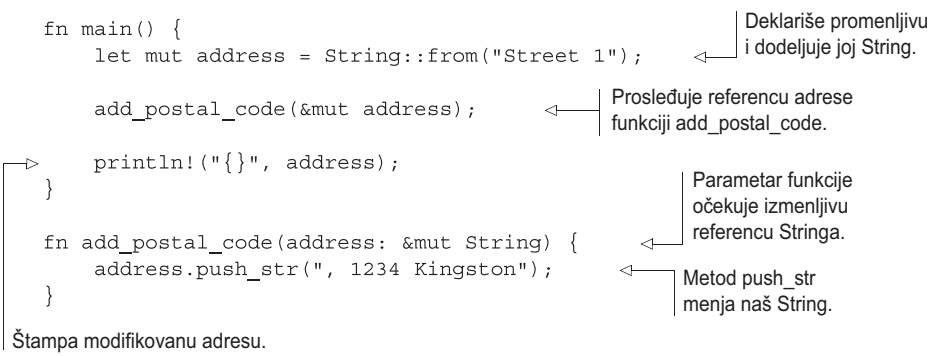
Razmotrimo ovaj primer detaljno. Prvo, nove promenljive služe podrazumevano samo za čitanje, a ako želimo da ih promenimo („mutiramo"), moramo da ključnoj reči `mut` dodamo ključnu reč `let` kada kreiramo novu promenljivu. Zatim, pozivamo funkciju `add_postal_code`, koja će dodati poštanski broj objektu `String` koji smo upravo kreirali.

Prosledivanjem adrese funkciji `add_postal_code` prenosimo vlasništvo na tu funkciju. Kada pokušamo da odštampamo `address` posle te linije koda, pojaviće se greška, kao u listingu 2.8. Funkcija `add_postal_code` očekuje izmenljivi objekat `String` (sa ključnom rečju `mut` u parametru) i dodaje mu nove znakove pomoću funkcije `.push_str`. Zatim, funkcija `.push_str` vraća ažurirani `String` koji dodeljujemo promenljivoj koja se zove `a`.

Umesto da smišljamo nove nazive za ovu novu promenljivu, možemo da koristimo potpuno isti naziv kao i ranije `address`. To se zove *zaklanjanje promenljivih* (variable shadowing) (http://mng.bz/p6ZG) i funkcija je Rusta, tako da ne morate stalno da pronalazite nove nazive za promenljive koje želite da modifikujete.

U listingu 2.10 je prikazan malo drugačiji pristup, na koji ćete možda češće nailaziti u Rust bazama koda. Umesto da prosledimo vrednost promenljive `address` i izgubimo vlasništvo, mi prosleđujemo referencu. Stoga, zadržavamo vlasništvo i pozajmljujemo ga funkciji kada joj je potrebno.

Listing 2.10 Prosleđivanje reference

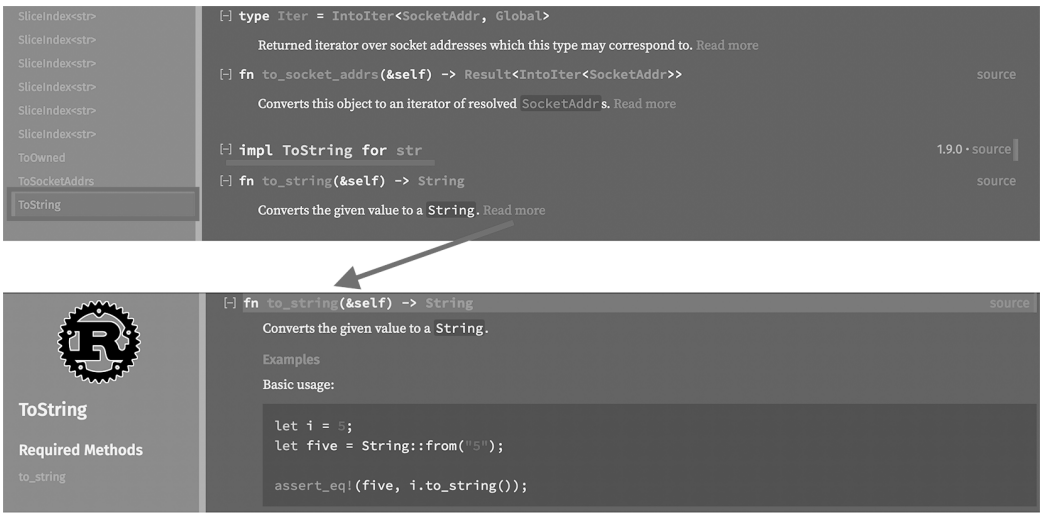


Funkcija `add_postal_code` pozajmljuje vlasništvo onoliko dugo koliko traje „telo" funkcije. Stoga, promenljiva `address` ne izlazi iz opsega (kao ranije) pre nego što pokušamo da je odštampamo.

Sumirali smo naše upoređivanje `Stringa` i `&stra` i principe vlasništva u Rustu. Jedna jednostavna greška otkrila je mnogo unutrašnje dinamike jezika. Sada znate kako da ispravite našu početnu grešku (očekujući `QuestionId`, umesto tipa `&str`) u listingu 2.5.

2.1.6 Korišćenje i implementiranje trejtova

Kompajler nam je ukazao da očekuje tip `QuestoinId`, umesto tipa `&str`. Otvorili smo dokumentaciju za tip`&str` (https://doc.rust-lang.org/std/primitive.str.html) da bismo saznali kako možemo da ga transformišemo u vrednost tipa `String`. Kada skrolujemo nadole, vidimo implementaciju trejta koji se zove `ToString`. Moramo da kliknemo na [+] pored `ToStringa` da bismo videli više detalja (pogledajte sliku 2.8). Kliknimo na Read More da bismo prešli na definiciju funkcije `to_string`, koju možemo da koristimo kada tip implementira trejt `ToString` (što čini `&str`).



Slika 2.8 Bočna traka sadrži svaki metod koji je dostupan određenom tipu, a pronalaženje detalja implementacije ponekad zahteva malo istraživanja.

Trejtovi

Za implementaciju zajedničkog ponašanja Rust ima koncept trejtova. Možete koristiti taj koncept da kreirate trejtove kojima je potrebno više od jednog tipa u vašoj aplikaciji. Takođe možete standardizovati ponašanje pomoću trejtova. Na primer, kada pretvarate jedan tip u drugi, možete koristiti trejt (baš kao trejt ToString u standardnoj biblioteci).

Dodatne

Još jedna prednost trejtova je da možete upotrebiti tipove u drugačijem kontekstu. Rust programi mogu biti generički, pa prihvataju svaki tip koji se ponaša na određeni način.

Razmislite o restoranu koji prima sve kućne ljubimce koji mogu da stanu ispod stola i piju vodu. Vaše funkcije u Rust programu se mogu ponašati na isti način. One vraćaju, na primer, tipove sa određenom karakteristikom. Sve dok vaš tip implementira ove karakteristike, one se mogu vratiti.

Na primer, brzo ćete implementirati trejtove u Rustu kada želite da odštampate prilagođene structove u konzoli (pomoću makroa derive, koji piše celu ručnu implementaciju trejta tokom kompajliranja).

Čini se da možemo da koristimo `to_string` da bismo konvertovali naš `&str` u `String`. Metod prihvata `&self`, pa možemo da ga pozovemo pomoću tačke u bilo kojem `&stru` koji definišemo, a zatim vraća `String`. To bi trebalo da nam pomogne da rešimo mnoštvo naših grešaka. Osim toga, pokušavamo da enkapsuliramo `id` u `QuestionId`, pošto smo ID na taj način definisali u našem structu.

Listing 2.11 Pretvaranje &stra u String

```
// ch_02/src/main.rs

struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

struct QuestionId(String);

impl Question {
    fn new(
       id: QuestionId,
       title: String,
       content: String,
       tags: Option<Vec<String>>
     ) -> Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}
```

```
fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        ["faq".to_string()],
    );
    println!("{}", question);
}
```

Izvršavanje komande `cargo run` u komandnoj liniji pokazuje određeni napredak. Imamo samo dve greške. Pogledajmo ponovo prvu grešku.

Listing 2.12 Greška pri vraćanju niza, umesto vektora

```
error[E0308]: mismatched types
  --> src/main.rs:25:9
   |
25 |         ["faq".to_string()],
   |         ^^^^^^^^^^^^^^^^^^^
               expected enum `Option`,
               found array of 1 element
   |
   = note: expected enum `Option<Vec<String>>`
              found array `[String; 1]`
```

Ovo takođe liči na dve greške u jednoj. Kompajler je očekivao enum `Option` sa vektorom `Vec`, ali je, umesto toga, pronašao niz. Ponovo koristimo dokumentaciju i tražimo `Option` u docs.rsu (https://doc.rust-lang.org/std/option/index.html).

Na osnovu navedenih primera vidimo da moramo da enkapsuliramo naše tagove u `Some`. A umesto niza, zahtevali smo u structu `Question` vektor sa stringovima. U Rustu vektor i niz nisu isto, a ako želite niz kao u drugim programskim jezicima, verovatno ćete tražiti `Vec` u Rustu.

Dokumentacija je takođe od pomoći i ovde, jer nam pokazuje dva načina zakreiranje vektora u Rustu. Možemo koristiti `Vec::new`, a zatim `.push` za umetanje elemenata, odnosno možemo koristiti makro koji se zove `vec!`. Koristimo ovaj primer da bismo ažurirali naš kod u skladu sa tim.

Listing 2.13 Enkapsuliranje i kreiranje vektora

```
fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{}", question);
}
```

Kada ponovo pokrenemo program, biće prikazana samo još jedna greška.

Listing 2.14 Poslednja greška sa kojom se suočavamo je nedostajuća implementacija trejta

```
error[E0277]: `Question` doesn't implement `std::fmt::Display`
  --> src/main.rs:27:20
   |
27 |     println!("{}", question);
   |                     ^^^^^^^^ `Question` cannot
          be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Question`
   = note: in format strings you may be able to use `{:?}` (or {:#?}
           for pretty-print) instead
   = note: required by `std::fmt::Display::fmt`
   = note: this error originates in a macro
      (in Nightly builds, run with -Z macro-backtrace for more info)

error: aborting due to previous error
```

Izgleda da nam nedostaje implementacija trejta koji se zove `std::Display`. U Rustu odštampajte promenljive pomoću makroa `println!` i dodajte vitičaste zagrade (`{}`) za svaku promenljivu koju želite da odštampate:

```
println!("{}", variable_name);
```

Biće pozvan metod `fmt` iz implementacije trejta `Display`: http://mng.bz/O6wn. Greška takođe predlaže korišćenje zagrada `{:?}`, umesto uobičajenih vitičastih zagrada `{}`. U sledećem izdvojenom tekstu „Poređenje Displaya i Debuga" objašnjena je razlika.

Poređenje Displaya i Debuga

Trejt `Display` je implementiran na sve primitivne tipove u Rustu (http://mng.bz/YKZN). On ukazuje da treba obezbediti implementaciju koja prikazuje podatke u obliku čitljivom za ljude. Lako je za brojeve i stringove, ali šta da radimo sa vektorima? Sve može da se nalazi unutar vektora, jer je `Vec<T>` generički kontejner za tipove podataka. Za ove slučajeve upotrebe (tj. za upotrebu složene strukture podataka poput vektora) Rust standardna biblioteka koristi trejt `Debug`.

Razlika za programera u korišćenju trejta Display i trejta Debug je sledeća: prilikom rukovanja stringovima i brojevima štampanje nečega se vrši pomoću dve vitičaste zagrade `{}`:`println!("{}", 3)`, a kad koristite složenije strukture podataka, kao što su structovi ili JSON vrednosti, koristite znakove `{:?}`, koji, umesto trejta `Display`, pozivaju trejt `Debug`: `println!("{:?}", question)`.

Trejt `Debug` možete da izvedete pomoću makroa `derive` i da stavite na vrh vaših structova. Stoga, ne morate da pišete implementacije prilagođenih trejtova i još uvek možete da odštampate svoje strukture podataka:

```
#[derive(Debug)]
struct Question {
    title: String,
    …
}
```

Takođe možete lepo da odštampate nešto, tako što ćete dodati `#: println!("{:#?}", question)`. To će omogućiti višelinijsko predstavljanje strukture podataka, umesto jednog dugačkog `Stringa`.

Baš kao ranije trejt `ToString`, trejt `Display` je još jedan standardni Rust trejt implementiran na sve primitivne tipove u biblioteci. To omogućava kompajleru da „zna" kako da prikaže ove tipove podataka (transformišući ih u ljudima čitljiv ispis). Naš prilagođeni tip nije deo standardne biblioteke i stoga ne implementira taj trejt.

Kako da krenemo dalje i saznamo kako da sami implementiramo taj trejt? Odgovor se opet nalazi u Rust dokumentaciji. Možemo potražiti `Display` (http://mng.bz/G1wq) i kliknuti na `[src]` da bismo pronašli implementaciju. Odeljak comment sadrži primer implementacije.

Listing 2.15 Primer implementacije trejta Display

```
// https://doc.rust-lang.org/src/core/fmt/mod.rs.html#743-767

/// # Examples
///
/// ```
/// use std::fmt;
///
/// struct Position {
///     longitude: f32,
///     latitude: f32,
/// }
///
/// impl fmt::Display for Position {
///     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
///         write!(f, "({}, {})", self.longitude, self.latitude)
///     }
/// }
///
/// assert_eq!("(1.987, 2.983)",
///            format!(
///              "{}",
///              Position {
///                longitude: 1.987, latitude: 2.983,
///              }
///            )
/// );
/// ```
```

Pomoću primera koda (prikazanog u listingu 2.14) iz Rust dokumentacije možete polako naučiti kako da koristite dokumentaciju za trejtove da biste ih primenili na svoje tipove. Prvo ćete kopirati osnovni primer u bazu koda i pokušati da zamenite primer structa (u ovom slučaju structa `Position`), vašim structom `Question`. Evo isečka koda koji implementira trejt `Display` za `Question`, sa izmenama napisanim podebljanim slovima:

```
impl std::fmt::Display for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error>
     {
        write!(
            f,
            "{}, title: {}, content: {}, tags: {:?}",
            self.id, self.title, self.content, self.tags
         )
    }
}
```

Kada pokušavamo da odštampamo `Question` pomoću makroa `println!`, pozivamo funkciju `fmt` koju smo upravo implementirali za `Question`. Ova funkcija poziva makro `write!`, koji štampa tekst u konzoli, a mi definišemo šta će tačno biti napisano, prosleđujući argumente tom makrou.

Međutim, evo dva upozorenja: naš struct `Question` ima prilagođeni struct koji se zove `QuestionId`, a koji takođe ne implementira trejt `Display` po podrazumevanoj vrednosti, a mi imamo složeniji struct za naše tagove - to je vektor. Ne možete da koristite trejt `Display` za složeniji struct, kao što je vektor, pa, umesto toga, morate da koristite trejt `Debug`. U sledećem listingu je prikazana naša datoteka main.rs, sa implementiranim trejtovima `Display` i `Debug`.

Listing 2.16 Primena trejta Display na naš trejt Question

```
…

impl std::fmt::Display for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter)
         -> Result<(), std::fmt::Error> {
        write!(
           f,
           "{}, title: {}, content: {}, tags: {:?}",
            self.id, self.title, self.content, self.tags
        )
    }
}

impl std::fmt::Display for QuestionId {
    fn fmt(&self, f: &mut std::fmt::Formatter)
         -> Result<(), std::fmt::Error> {
        write!(f, "id: {}", self.0)
    }
}

impl std::fmt::Debug for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>)
          -> Result<(), std::fmt::Error> {
        write!(f, "{:?}", self.tags)
    }
}

…
```

Implementacija `Debug` izgleda prilično slično implementaciji trejta `Display`. Ovde koristimo `{:?}`, umesto `{}`, u makrou `write!`. Bilo bi prilično zamorno pisati sav ovaj kod uvek kada želimo da implementiramo i prikažemo prilagođeni tip.

Rust standardna biblioteka obezbeđuje proceduralni makro za ovu implementaciju, koji se zove `derive`, a možemo ga postaviti na vrh naše struct definicije pomoću linije koda `#[derive]`. Dokumentacija za trejt `Display` takođe sadrži važnu činjenicu: `Display` *je sličan* `Debugu`*, ali služi za ispis koji je usmeren ka korisniku, pa ne može da bude izveden.*

Deklarativni makroi

Makroe u Rustu možete uočiti po znaku uzvika na kraju naziva. Taj znak ukazuje na deklarativni makro ili na *proceduralni makro* sličan funkciji (koji se razlikuje od proceduralnog makroa). Najpoznatiji deklarativni makro koji ćete koristiti u Rustu na početku je `println!`, a on štampa tekst u konzoli.

Makroi prihvataju enkapsulirani kod i generišu standardni Rust kod iz njega neposredno pre nego što kompajler preuzme sav Rust kod i kreira binarnu datoteku.

Takođe možete kreirati svoje makroe, iako tamo ulazite u novi „svet", skoro bez ikakvih pravila. Kreiranje vaših makroa trebalo bi da sledi nakon što postanete dovoljno vešti u korišćenju standardnog Rusta.

Stoga, idemo dalje i izvodimo trejt `Debug`. Posle toga, moramo da koristimo `Debug` u našem makrou `println!` pomoću makroa `{:?}`, umesto `{}`. Ažurirani kod je naveden u sledećem listingu koda, a nakon pokretanja programa, biće prikazan sadržaj našeg pitanja u konzoli (za sada, ignorišite upozorenja).

Listing 2.17 Korišćenje makroa derive za implementaciju trejta Debug

```
#[derive(Debug)]
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

#[derive(Debug)]
struct QuestionId(String);

impl Question {
    fn new(
        id: QuestionId,
        title: String,
        content: String,
        tags: Option<Vec<String>>
    ) -> Self {
        Question {
            id,
            title,
```

```
            content,
            tags,
        }
    }
}

fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

Još ima šanse za poboljšanje. Izdvojili smo ID pitanja iza structa `QuestionId`, ali još uvek treba da saznamo da li ovaj struct prihvata `String` kao ulaz. Možemo sakriti ovaj detalj implementacije i olakšati korisniku da generiše ID za pitanje.

Rust obezbeđuje uobičajene funkcionalnosti. Jedna od njih je trejt `FromStr`, koji je sličan trejtu `ToString`, o kojem je bilo reči ranije. Možemo koristiti `FromStr` na sledeći način:

```
let id = QuestionId::from_str("1").unwrap(); // from_str() can fail
```

Jednostavno ovaj kod se čita ovako: *Kreiraj tip Xiz tipa &str*. Rust nema implicitnu konverziju, samo eksplicitnu. Dakle, uvek ukazujemo da li želimo da konvertujemo jedan tip u drugi.

Listing 2.18 Primena trejta FromStr na trejt QuestionId

```
use std::io::{Error, ErrorKind};
use std::str::FromStr;

…

impl FromStr for QuestionId {
    type Err = std::io::Error;

    fn from_str(id: &str) -> Result<Self, Self::Err> {
        match id.is_empty() {
            false => Ok(QuestionId(id.to_string())),
            true => Err(
              Error::new(ErrorKind::InvalidInput, "No id provided")
            ),
        }
    }
}

…
```

Potpis trejta omogućava da prihvatimo `&str` i vratimo naš tip (`QuestionId`) ili grešku u slučaju da je ID prazan.

Parametru dodeljujemo naziv `id` i tip `&str` (pošto ćemo taj tip primiti). Naziv (`id` u ovom primeru) može biti bilo šta što izaberemo. Zatim, postižemo podudaranje ako `id` nije prazan i vraćamo odgovarajući tip `QuestionId` sa jednim poljem unutar njega koje transformišemo u `String`, baš kao što smo naveli u structu.

Možemo da promenimo način na koji kreiramo ID pitanja u funkciji `main`. Umesto da koristimo `.to_string`, pozivamo `::from_str` u trejtu `QuestionId`. U implementaciji trejta možemo videti da metod `from_str` ne prihvata tip `&self`, pa to nije metod koji možemo pozvati pomoću tačke (`.`), već je pridružena funkcija koju pozivamo pomoću duple dvotačke (`::`).

Listing 2.19 Korišćenje trejta FromStr za kreiranje QuestionIda iz &stra

```
...

fn main() {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

2.1.7 Rukovanje rezultatima

Zašto moramo da dodamo `.expect` posle funkcije? Ako pažljivo pogledamo, vidimo da implementacija trejta `FromStr` vraća `Result`. *Rezultati* su skoro kao opcije i mogu imati dve moguće povratne vrednosti: uspeh (success) ili grešku (error). U slučaju uspeha, enkapsuliraju vrednost, koristeći `Ok(value)`. U slučaju greške, enkapsuliraju grešku u `Err(error)`. Tip `Result` je takođe implementiran kao enum i izgleda ovako:

Listing 2.20 Definicija Result u Rust standardnoj biblioteci

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Baš kao i `Option`, `Result` ima različite implementirane metode i trejtove. Jedan od ovih metoda je `expect`, koji, na osnovu dokumentacije, vraća sadržanu vrednost `Ok`. Možemo uočiti da zaista vraćamo trejt `QuestionId` omotan u `Ok`, što znači da metod `expect` vraća vrednost unutar njega ili generiše poruku o grešci, koju navedemo.

Za pravilno rukovanje greškama potreban je iskaz `match`, u kojem bismo primili grešku iz funkcije `from_str` i obradili je u nekom obliku. U našem jednostavnom primeru, međutim, dovoljan je način obrade grešaka koji smo upotrebili. Još jedan uobičajeni metod na koji ćete naići je unwrap - on je bolji za čitanje, ali će „paničiti" bez poruke o grešci koju ste naveli.

Nemojte koristiti metode `unwrap` ili `expect` u finalnoj verziji, jer oni dovode do „panike" i otkazivanja vaše aplikacije. Uvek rešavajte slučajeve grešaka pomoću iskaza `match` ili se uverite da ste „uhvatili" greške i elegantno ih vratite.

Možete lako da koristite enum `Result` i za svoje funkcije. Povratni potpis će izgledati ovako `- >Result<T, E>`, u kojem `T` označava vaše podatke koje želite da vratite, a `E` je greška koja može biti prilagođena ili greška iz standardne biblioteke.

`Result` izgleda isto kao i `Option`; glavna razlika je u varijanti greške (`Error`). `Option` se koristi kad god neki podaci mogu ili ne mogu biti prisutni, ali kada ne izazivaju probleme. `Result` se koristi tamo gde zaista očekujete da će nešto biti prisutno i morate aktivno da upravljate slučajem kada nešto nije prisutno.

Jednostavan primer je naš primer prethodnog koda. Tagove označavamo kao opcione u našem structu `Question`, pošto možemo da kreiramo pitanje i bez njih. Međutim, ID je potreban, a ako struct `QuestionId` ne može da kreira ID iz `&stra`, kreiranje nije uspešno i moramo da vratimo grešku onome ko pozove ovaj metod. Nakon što su osnovna struktura i tipovi implementirani, dodajmo veb server našoj aplikaciji da bismo mogli da obezbedimo prve primere podataka korisnicima.

2.2 Kreiranje veb servera

Razmotrili smo koje funkcije Rust sadrži i ne sadrži za izradu veb usluga. Da rezimiramo suštinu:

- Rust se ne isporučuje sa runtimeom koji može da obradi asinhroni rad u pozadini.
- Rust obezbeđuje sintaksu za izražavanje blokova asinhronog koda.
- Rust uključuje tip `Future` za rezultate koji imaju stanje i povratni tip.
- Rust implementira TCP (i UDP), ali ne HTTP.
- Veb radni okvir koji smo izabrali isporučuje se sa HTTP-om i svime što je implementirano.
- Runtime diktira veb radni okvir koji izaberemo.

Naš veb radni okvir diktira mnogo štošta „iza kulisa": runtime, apstrakciju pomoću HTTP-a (i implementaciju HTTP servera) i dizajn načina na koji se prosleđuju zahtevi hendelerima ruta. Stoga bi trebalo da budete zadovoljni dizajnom i runtimeom koji je izabrao određeni veb radni okvir.

Da biste razumeli sintaksu i teme koje ćemo razmatrati, u listingu 2.21 je prikazan pregled primera koji će biti završen do kraja ovog poglavlja. Nakon što pročitate taj poslednji deo ovog poglavlja, razumećete šta ovaj kod radi. Želimo samo da istaknemo neke oblasti.

Listing 2.21 Minimalni Rust HTTP server sa Warpom

```
use warp::Filter;

#[tokio::main]
async fn main() {
```

```
    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

Treća linija koda `#[tokio::main]` označava runtime koji koristimo. U odeljku 2.2.1 je razmotren runtime. Međutim, da biste mogli da zamislite u svojoj glavi kako izgleda runtime, predstavili smo kako on izgleda u vašem Rust izvornom kodu. Sledeće, u liniji 4 je funkcija `main`, koju označavamo kao asinhronu - to radimo da bismo istovremeno obradili više zahteva (zahvaljujući runtimeu), a unutar asinhronih funkcija možemo da koristimo ključnu reč `.await` da bismo naznačili da je funkcija asinhrona po prirodi i da ne generiše rezultat odmah. Ovo su već tri (od četiri) gradivna bloka asinhrone „priče" u Rustu.

2.2.1 Rukovanje višestrukim zahtevima odjednom

Kada pišete serverske aplikacije, obično morate da opslužujete više od jednog klijenta odjednom. Čak i ako sve vaše veze ne stižu u istoj milisekundi, čitanje iz baza podataka ili otvaranje datoteka na čvrstom disku zahtevaju vreme.

Umesto da čekate da se svaki proces u potpunosti završi i pustite stotine, ako ne i hiljade drugih zahteva da čekaju i da se gomilaju, možete izabrati da pokrenete proces (na primer, upit baze podataka) i omogućite da budete obavešteni kada se on završi. U međuvremenu možete početi da opslužujete druge klijente.

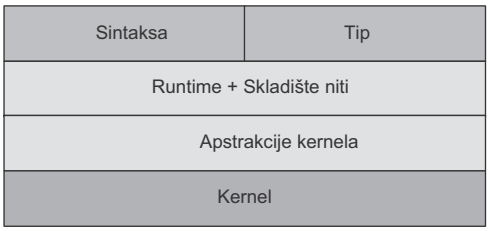
> **Zelene niti**
>
> U asinhronom programiranju su uvek uključene programske niti koje se kreiraju („mreste") pomoću nekog procesa i nalaze se unutar njega. Njima obično rukuje operativni sistem (unutar kernela), pa su skupi za upravljanje iz perspektive korisničkog prostora (zbog stalnog prekidanja kernela).
>
> Stoga je stvoren koncept „zelene niti" (http://mng.bz/nemK). Ove niti se nalaze u potpunosti u korisničkom prostoru i njima upravlja runtime.

Pisanje asinhronih aplikacija omogućava da upravo sprečite gomilanje zahteva. Kada pogledate naš kod `minimal-tcp` iz Poglavlja 1 (https://github.com/Rust-Web-Development/code/blob/main/ch_01/minimal-tcp/src/main.rs), videćete da on obrađuje jedan strim za drugim na blokirajući način. To znači da završavamo obradu jednog strima pre nego što pređemo na sledeći.

U okruženju sa više niti mogli bismo da stavimo svaki strim u posebnu nit, da pustimo da se strimovi obrađuju u pozadini i vratimo ih u prvi plan i da pošaljemo odgovor nazad klijentu koji je uputio zahtev. U drugoj dizajnerskoj odluci koristimo jednu nit, koja preuzima zadatak kad god je to moguće. Ključ je u tome da dugotrajni metod može vratiti kontrolu runtimeu i

signalizirati da mu je potrebno više vremena da bude izvršen. Runtime onda može da izvrši druge obrade i da ponovo proveri da li je ovaj dugotrajni metod završio obradu.

Da bismo asinhrono obradili dolazne HTTP zahteve, potreban nam je programski jezik koji razume asinhrone koncepte i obezbeđuje tipove i sintaksu da označimo kod koji bi trebalo da se izvršava asinhrono. Neophodan nam je i runtime koji prihvata naš kod i koji „zna" kako da ga izvrši na način koji nije blokirajući. Na slici 2.9 prikazani su potrebni „sastojci".



Slika 2.9 Za okruženje asinhronog programiranja potrebna su četiri sastojka (sintaksa, tip, runtime i apstrakcije kernela) da bi funkcionisalo.

Da sumiramo četiri „sastojka" okruženja asinhronog programiranja:

- iskorišćavanje kernelovog API-a za asinhrono čitanje i pisanje `epoll/select/poll` (za dodatnu literaturu pogledajte link http://mng.bz/09zx)
- mogućnost rasterećivanja dugotrajnih zadataka u korisničkom prostoru i posedovanja nekog oblika mehanizma da nas obavesti kada zadatak bude završen kako bismo mogli da napredujemo u radu. Runtime kreira i zelene niti i upravlja njima.
- sintaksa unutar programskog jezika, tako da možemo da označimo asinhrone blokove u našem kodu da bi kompajler znao šta da radi sa njima
- specifičan tip u standardnoj biblioteci za međusobno isključivi pristup i modifikaciju. Umesto tipa, kao što je *broj*, u kojem se čuva određena vrednost, tip za asinhrono programiranje treba da skladišti ne samo vrednosti, već i trenutno stanje dugotrajne operacije.

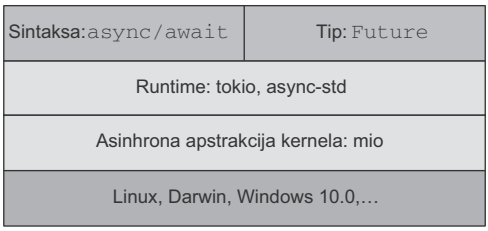
Rust ne implementira apstrakciju pomoću asinhronog API-a kernela, niti standardna biblioteka obezbeđuje runtime. To je u suprotnosti sa NodeJS i Go jezicima koji se isporučuju sa izvornim runtimeom i apstrakcijom pomoću API-a kernela.

Rust obezbeđuje sintaksu i tip. Sam Rust razume asinhrone koncepte i isporučuje dovoljno „sastojaka" da izradimo runtime i izvršimo apstrahovanje pomoću API-a kernela.

2.2.2 Rustovo asinhrono okruženje

Da bi Rust imao manji memorijski otisak, ne uključuje se runtime, niti apstrakcija pomoću asinhronog API-a kernela. To daje programeru šansu da izabere runtime koji odgovara potrebama projekta i takođe prilagođava Rust budućim potrebama ukoliko vremenom dođe do velikog napretka u runtimeima.

Već ste videli glavne blokove asinhrone „priče” na slici 2.8. Rust se isporučuje sa sintaksom i tipom. Tu su i dobro testirani izabrani runtimei (kao što su Tokio i async-std) i apstrakcija pomoću asinhronog API-a kernela `mio`. Na slici 2.10 prikazani su gradivni blokovi izvedeni iz Rust eko-sistema.



Slika 2.10 Gradivni blokovi asinhronog Rust eko-sistema

Za svoju sintaksu Rust obezbeđuje kombinaciju `async` i `await` ključnih reči. Označite funkciju `async` tako da možete da koristite `await` unutar nje. Funkcija `await` vraća tip `Future`, koji ima tip vrednosti koji vraćate u slučaju uspešnog izvršenja, i metod `poll`, koji izvršava dugotrajni proces i vraća `Pending` ili `Ready`. Stanje `Ready` može sadržati ili `Error` ili uspešno vraćenu vrednost.

Često nećete morati da razumete detalje tipa `Future`. On je od pomoći u daljem razumevanju osnovnog sistema, tako da ga možete kreirati kada je potrebno, ali, za početak, dovoljno je da znate zašto postoji i kako funkcioniše zajedno sa ostatkom eko-sistema.

U svakoj asinhronoj aplikaciji u Rustu je važno koji runtime ćete izabrati. Runtime će već uključivati apstrakciju pomoću asinhronog API-a kernela (koji je u većini slučajeva biblioteka koja se zove Mio). Najpre ćemo da pogledamo šta Rust isporučuje, tj. sintaksu i tip.

2.2.3 Rustovo rukovanje sintaksom async/await

Na vrhu runtimea postoje dva gradivna bloka integrisana u Rust. Prva je sintaksa `async/await`. Pogledajmo ponovo naš isečak koda iz Poglavlja 1 koji izvršava asinhroni HTTP poziv.

Listing 2.22 Primer HTTP poziva async

```
use std::collections::HashMap;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let resp = reqwest::get("https://httpbin.org/ip")
        .await?
        .json::<HashMap<String, String>>()
        .await?;
    println!("{:#?}", resp);
    Ok(())
}
```

Imajte na umu da morate da dodate cratese Tokio i Reqwest u vaš Cargo.toml da bi ovaj isečak koda funkcionisao. Mnogi Rust cratesi dele svoju logiku na funkcije, što omogućava samo uključivanje manjeg podskupa funkcionalnosti, tako da vaša aplikacija neće sadržati kod koji vam nije potreban.

```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }
```

Flegovi funkcija

Kada dodajemo `tokio` našim zavisnostima unutar datoteke Cargo.toml, moramo da dodamo flegove funkcija. *Flegovi funkcija* omogućavaju programerima da samo uključe podskup cratea, pri čemu se štedi vreme za kompajliranje projekta i takođe se smanjuje veličina projekta.

Ne podržavaju svi cratesi flegove funkcija. Imajte na umu da, ako uključite crate i želite da koristite određene funkcije, kompajler vas neće obavestiti da funkcija nije uključena u datoteku Cargo.toml. Najsigurniji „ulog" na početku je da uključite sve funkcije cratea, a nakon što završite, proverite da li možete da smanjite količinu koda koji dodajete, koristeći samo određene funkcije.

Imenovanje flegova funkcija nije standardizovano - prepušteno je da vlasnik cratea imenuje njegove funkcije. Za Tokio i primer u knjizi koristimo fleg funkcije `full`:

```
tokio = { version = "1", features = ["full"] }
```

Poziv funkcije `reqwest::get(„https://httpbin.org/ip")` vraća `Future` koji omotava tip return. Ovaj poziv vraća našu trenutnu IP adresu u obliku objekta, sa ključem i vrednošću. U Rustu to se može izraziti pomoću HashMapa: `HashMap<String, String>`. Crate Reqwest podrazumevano vraća `Future` (https://docs.rs/reqwest/latest/reqwest/#making-a-get-request). Ako želite blokirajući način kreiranja HTTP zahteva, koristite klijent `reqwest::blocking`: https://docs.rs/reqwest/latest/reqwest/blocking/index.html.

Očekujemo `HashMap` omotan u `Future` kao odgovor: `Future<Output=HashMap <String, String>>`. Onda možemo pozvati `await` u tip Future, pa ga naš runtime preuzima i pokušava da izvrši funkciju unutar njega. Pretpostavljamo da će to izvršavanje potrajati duže, tako da će runtime upravljati zadatkom u pozadini i popuniti našu promenljivu sadržaja nakon što je datoteka pročitana.

Obično nećete naići na definisanje vaših Futurea, bar ne na početku kada koristite Rust i veb usluge. Važno je da znate da, prilikom upotrebe cratesa ili tuđeg koda kada su funkcije označene kao `async`, možete koristiti `await` u njima.

Ova sintaksa treba da pisanje asinhronog Rust koda učini sinhronim, pri čemu se blokira kod za programera. U pozadini Rust transformiše ovaj deo koda u mašinu stanja u kojoj svaki drugačiji `await` predstavlja stanje. Kada su sva stanja spremna, funkcija nastavlja da se izvršava do poslednje linije koda i vraća rezultat.

Zbog sintakse koja izgleda kao blokirajući konkurentni proces, može vam biti teško da shvatite prirodu i zamke asinhronog programiranja. Razmotrićemo detalje kada budemo implementirali našu prvu aplikaciju. Za sada je dovoljno da razumete „sastojke". Najpre ćemo da pogledamo unutrašnjost Futurea da biste razumeli šta koristimo, odnosno šta koristi runtime koji smo izabrali.

2.2.4 Rustov tip Future

U listingu 2.22 možete videti da naša `await` funkcija vraća nešto što čuvamo u promenljivoj `resp`. Ovde „u igru" ulazi naš tip `Future`. Kao što je pomenuto, Future je složeniji tip koji ima sledeći potpis (prikazan u listingu 2.3). Ne morate da u potpunosti razumete šta taj tip radi, ali isečak koda ima dva uključena linka tako da možete dodatno istraživati. Objašnjenje ključne funkcionalnosti sledi odmah nakon listinga.

Listing 2.23 Rustov trejt Future

```
use std::collections::HashMap;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let resp = reqwest::get("https://httpbin.org/ip")
        .await?
        .json::<HashMap<String, String>>()
        .await?;
    println!("{:#?}", resp);
    Ok(())
}
```

Rustov trejt `Future` ima pridruženi tip `Output`, koji može biti, na primer, `file` ili `String`, i metod, koji se zove `poll`. Ovaj metod se često poziva da bi se videlo da li je `Future` spreman. Metod `poll` vraća tip `Poll` koji može biti `Pending` ili `Ready`. Kada je spreman, `poll` vraća tip naveden u drugoj liniji koda ili grešku. Kada je `Future` spreman (`Ready`), vraća rezultat koji se, zatim, dodeljuje našoj promenljivoj.

Možete videti da je `Future` trejt. To je korisno, jer možete da primenite ovaj trejt na bilo koji tip koji imate u svom programu.

Posebno treba istaći da se u Rustu ni jedan `Future` aktivno ne pokreće. U drugim jezicima, kao što su Go ili JavaScript, prilikom dodeljivanja promenljive Promisu ili kreiranja go rutine, svaki njihov runtime će odmah početi da se izvršava. U Rustu morate aktivno da ispitujete future (`poll`), što je, inače, posao runtimea.

2.2.5 Odabir runtimea

Runtime je u centru vaše asinhrone veb usluge. Njegov dizajn i performanse su važni za osnovne performanse i bezbednost vaše aplikacije. U NodeJS-u imate „Googleov" V8 mehanizam koji se bavi odabirom runtimea. Go ima svoj runtime koji je takođe razvio „Google".

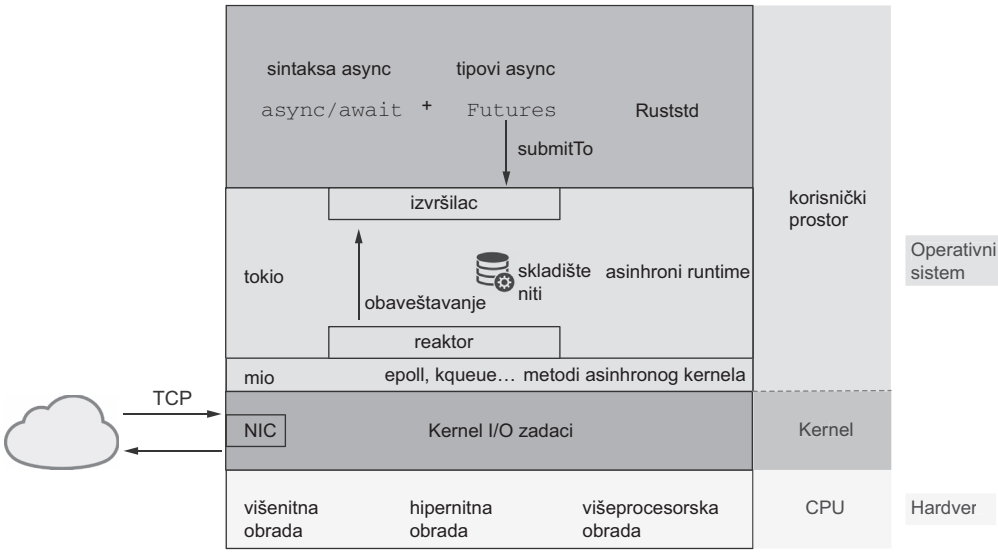
Ne morate da znate detaljno kako runtime funkcioniše i kako izvršava vaš asinhroni kod. Međutim, dobro je bar da upoznate njegove termine i koncepte. Možda ćete kasnije naići na probleme u svom kodu u kojem vam znanje o funkcionisanju runtimea koji ste izabrali može pomoći da rešite probleme ili da ponovo napišete svoj kod.

Mnogi ljudi kritikuju što se Rust ne isporučuje sa runtimeom, jer je runtime jednostavno centralni deo svake veb usluge. Sa druge strane, mogućnost odabira određenog runtimea za vaše potrebe ima prednost u prilagođavanju vaše aplikacije potrebama performansi i platforme.

Jedan od najpopularnijih runtimea se zove Tokio i često se koristi u mnogim većim kompanijama. Stoga je on prvi siguran ulog u vašoj aplikaciji. Odabraćemo `ga` za naš primer, a kasnije će biti više reči o odabiru runtimea za vaše potrebe.

Runtime je odgovoran za kreiranje niti i ispitivanje tipova Future i za njihov završetak. Takođe je odgovoran za prosleđivanje posla kernelu i iskorišćavanje asinhronog API-a kernela da ne bi došlo do „uskih grla”. Tokio koristi crate Mio (https://github.com/tokio-rs/mio) za asinhronu komunikaciju sa kernelom operativnog sistema. Kao programer, verovatno nikada nećete dirati kod u biblioteci Mio, ali dobro je da znate koje tipove cratea i slojeva apstrakcije „uvlačite” u svoj projekat kada razvijate veb servere pomoću Rusta.

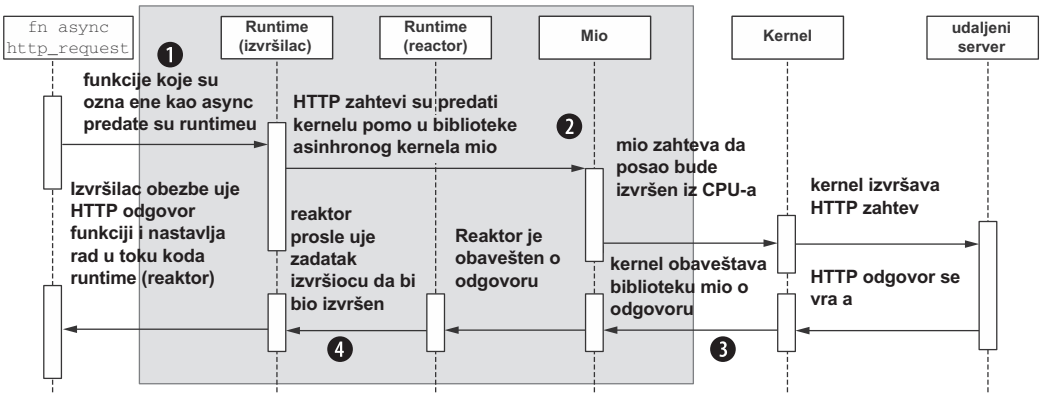
Kao što vidite na slici 2.11, runtime ima prilično značajnu ulogu u veb usluzi. Kada se kod označava kao `async`, kompajler će predati kod runtimeu i onda od implementacije zavisi koliko će vaše izvršavanje ovog zadatka biti brzo, tačno i bez grešaka. Hajde da pogledamo primer zadatka i da vidimo šta se dešava „iza kulisa”.



Slika 2.11 Kompletno asinhrono Rust okruženje

Hajde da pregledamo primer zadatka da vidimo šta se dešava „iza kulisa” (pogledajte sliku 2.12).

1. U našem Rust kodu označavamo funkciju kao `async`. Kada čekamo povratak funkcije, mi ukazujemo runtimeu tokom kompajliranja da je to funkcija koja vraća tip `Future`.
2. Runtime prihvata ovaj deo koda i predaje ga izvršiocu, koji je odgovoran za pozivanje metoda `poll` u tipu `Future`.
3. Ako je poslat mrežni zahtev, runtime ga predaje Miou koji kreira asinhroni socket u kernelu i zahteva malo CPU vremena da završi zadatak.
4. Kada kernel završi poslove (kao što su slanje zahteva i primanje odgovora), on obaveštava proces čekanja u socketu. Reaktor je tada odgovoran za „buđenje” izvršioca koji nastavlja obradu pomoću rezultata vraćenih iz kernela.



Slika 2.12 Izvršavanje asinhronih HTTP zahteva „iza kulisa“

2.2.6 Izbor veb radnog okvira

S obzirom da je Rust još uvek novo „igralište” za veb usluge, možda će vam biti potrebna aktivnija pomoć od programerskog tima i zajednice da rešite probleme koje možete imati na tom „putu”.

Ovo su prva četiri veb radna okvira koje Rust može da obezbedi:

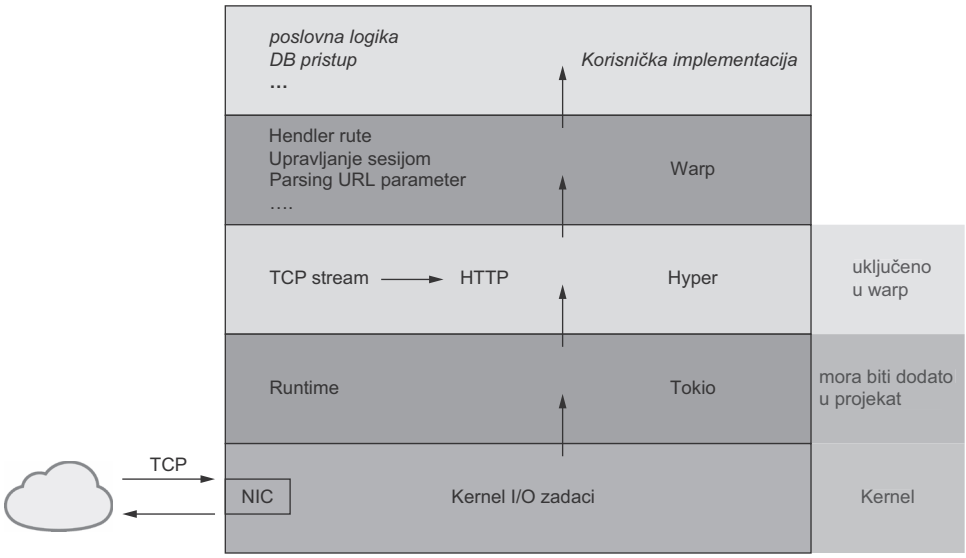
- *Actix Web* je potpuniji, aktivniji korišćeni veb radni okvir i sadrži mnogo funkcija. Ponekad može biti dogmatičan.
- *Rocket* koristi hendlere ruta za anotaciju makroa i ima ugrađeno JSON rasčlanjavanje. To je veoma kompletan radni okvir, sa svim funkcijama za pisanje stabilnih veb servera.
- *Warp* je bio jedan od prvih veb radnih okvira za Rust. Razvijen je u blizini Tokio zajednice i obezbeđuje mnogo slobode. To je najosnovniji radni okvir koji mnoge dizajnerske odluke prepušta programeru.

- *Axum* je najnoviji u grupi radnih okvira i pokušavano je da se što više nadgradi na već postojeće cratese iz tokio eko-sistema, zahvaljujući lekcijama o dizajnu naučenim iz Warpa i drugih radnih okvira.

Actix Web obezbeđuje svoj runtime (ali, možete izabrati i da koristite Tokio). Radni okviri Rocket, Warp i Axum koriste Tokio.

Za ovu knjigu biramo Warp. Dovoljno je mali da se „skloni sa puta" i dovoljno se koristi da se njime aktivno upravlja pomoću veoma aktivnog Discord kanala. Vaše iskustvo može varirati u vašoj kompaniji ili projektu. Važno je da razumete gde radni okvir počinje i gde se završava, šta je čisti Rust i gde će na vaš kod uticati radni okvir koji izaberete. Jasno ćemo istaći te delove da biste znali gde kasnije treba da uključite vaš radni okvir koji izaberete.

U većem delu knjige i koda nismo zagovornici radnog okvira. Kada postavimo server i dodamo hendlere ruta, ponovo smo u čistoj „zemlji" Rust i nećemo videti mnogo radnog okvira kasnije. Jasno ćemo u ovoj knjizi istaći te delove, tako da znate gde da kasnije uključite radni okvir koji ste izabrali.



Slika 2.13 Kada koristite warp, nasleđujete runtime Tokio i Hyper kao HTTP apstrakciju i server „ispod haube".

Kao što vidite na slici 2.11, dolazni TCP zahtevi će morati da budu predati runtimeu Tokio, koji komunicira direktno sa kernelom. Hyper biblioteka će pokrenuti HTTP server i prihvatiti ove dolazne TCP strimove. Osim toga, Warp će se omotati oko funkcija radnog okvira – na primer, HTTP zahtevi će biti prosleđeni odgovarajućim hendlerima ruta. U listingu 2.24 prikazana je ova kompletna akcija.

Listing 2.24 Minimalni Rust HTTP server sa Warpom

```
use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::path("hello")
        .and(warp::path::param())
        .map(|name: String| format!("Hello, {}!", name));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

Funkcija .map je warp Filter koji prihvata (moguće) argumente iz prethodne funkcije i transformiše ih.

Potpis (`.map(||...)`) koristi takozvano zatvaranje (`||`) koje „hvata" promenljive u okruženju i čini ih dostupnim unutar funkcije (`map`). U prethodnom primeru ne koristimo ni jednu promenljivu unutar funkcije `map`. Međutim, ako bi HTTP GET zahtev imao neke parametre, mogli bismo da ih „uhvatimo" i obradimo unutar funkcije `map` pomoću zatvaranja (`||`).Više detalja o zatvaranjima sadrži knjiga o Rustu na adresi https://doc.rust-lang.org/book/ch13-01-closures.html.

Da bismo završili ovo poglavlje o radnom veb serveru, pokušaćemo da uključimo ovaj primer isečka u našu bazu koda. Do sada je naša glavna funkcija izgledala ovako (kasnije može da se menja):

Listing 2.25 Naša funkcija main do ovog trenutka

```
fn main() {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

Uklanjamo kreaciju novog questiona (objasnili smo u Poglavlju 3 kako da vratite JSON), dodajemo runtime i Warp server našem projektu i pokrećemo server u funkciji `main`. Moramo da dodamo dve zavisnosti našem projektu. Hyper crate je uključen u Warp, dok se Tokio mora dodati projektu ručno. U sledećem listingu je prikazana ažurirana datoteka Cargo.toml.

Listing 2.26 Ažurirana datoteka Cargo.toml, sa Tokiom i Warpom koji su dodati projektu

```
[package]
name = "ch_02"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.2", features = ["full"] }
warp = "0.3"
```

Uz dodatnu Tokio zavisnost, možemo da označimo našu glavnu funkciju tako da koristi runtime Tokio i da unutar nje upišemo asinhroni kod koji Warp zahteva. U sledećem listingu je prikazana ažurirana datoteka `main.rs` u našoj fascikli ch_02.

Listing 2.27 Pokretanje warp servera unutar datoteke main.rs

```
use std::str::FromStr;
use std::io::{Error, ErrorKind};

use warp::Filter;

…

#[tokio::main]
async fn main() {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);

    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

Možete pokrenuti server pomoću komande `cargorun` u komandnoj liniji (za sada, ignorišite njena upozorenja). I dalje će biti odštampan primer `question` u komandnoj liniji, ali će takođe biti pokrenut server. Kada otvorite pregledač i otvorite adresu `127.0.0.1:3030`, trebalo bi da vidite pozdrav `Hello, World!`.

Kada je server postavljen, počnite da implementirate vaše REST krajnje tačke – treba da shvatite kako se serijalizuju structovi da biste vratili odgovarajući JSON klijentu koji je uputio zahtev i kako možete da prihvatite parametre upita i na krajnjim tačkama. To i još mnogo štošta vas čeka u Poglavlju 3.

Rezime

- Uvek prvo mapirajte resurse pomoću structova i razmislite o vezama između vaših tipova.
- Pojednostavite svoj život dodavanjem pomoćnih metoda u vaše tipove, kao što je `new`, i transformisanjem jednog tipa u drugi.
- Naučite principe vlasništva i pozajmljivanja u Rustu i kako oni utiču na način pisanja koda i koje greške bi kompajler mogao da generiše.

- Korišćenje trejtova pomaže da vaši prilagođeni tipovi podataka lepo funkcionišu zajedno sa radnim okvirima koje izaberete dodavanjem još funkcionalnosti.
- Korišćenje makroa `derive` za implementaciju trejtova za uobičajene slučajeve upotrebe pomaže da skratite mnogo koda koji sami pišete.
- „Sprijateljite" se sa Rust dokumentacijom, jer ćete je često koristiti za traženje funkcionalnosti u tipovima i radnim okvirima, što vam, zauzvrat, pomaže da bolje razumete Rust jezik.
- Rust se isporučuje sa sintaksom async i tipovima, ali nam je potreban više za pisanje asinhronih aplikacija.
- Runtime se brine o rukovanju višestrukim obradama u isto vreme i obezbeđuje apstrakciju pomoću asinhronog API-a kernela.
- Odaberite veb radni okvir koji se aktivno održava, koji ima veliku zajednicu i podršku i koji možda koriste veće kompanije od onih u kojima poslujete.
- Veb radni okvir koji odaberemo će se apstrahovati pomoću HTTP implementacije, servera i runtimea, tako da možemo da se fokusiramo na pisanje poslovne logike za aplikaciju.