

# Rust

## Uvod

### Šta je Rust?

Rust je moderan programski jezik, zvanično objavljen 2015. godine. Rust se predstavljao kao sigurnija opcija od C i C++, omogućava rad sa memorijom uz dodatne sigurnosne funkcionalnosti koje sprečavaju loše upravljanje memorijom. Rust takođe omogućava pronalazak većinu programskih grešaka za vreme kompajliranja.

### U čemu je Rust dobar?

Rust je odličan u situacijama kada su potrebni sigurnost i performanse. Kompajliranjem dobijamo mašinski kod čije performanse su slične C-u ili C++-u. To omogućava Rust-u primenljivost u svim sferama programiranja.

### Razlika između Rust-a i C++-a?

Glavna razlika je to što Rust forsira sigurnost memorije tako što svaka memorija mora da ima svog vlasnika i uz pomoć sistema pozajmljivanja koji se proveravaju za vreme kompajliranja. To sprečava trku za resursom ili loše pristupanje memoriji. Takođe zbog tih funkcionalnosti Rust nema potrebu za garbage kolektorom jer se memorija briše u pravo vreme. Kao rezultat toga Rust značajno smanjuje šansu za greškom prilikom izvršavanja programa ili dolaska u nedefinisano stanje.

### Neke funkcionalnosti Rust-a

- Odsutnost garbage kolektora  
Memorija se briše kad izađe iz svog opsega.
- Sistem vlasništva  
Svako parče memorije ima svog vlasnika koji je zadužen za njegov životni vek.
- Pozajmljivanje varijabli i njihove reference.  
Rust omogućava pozajmljivanje memorije bez promene vlasnika. Uz to proverava ko i kada pristupa tim podacima.
- Rukovanje greškama  
Rust omogućava funkcijama da vrate varijablu tipa Result koja u sebi sadrži kod greške ako postoji i vrednost koju funkcija želi da vrati.

- Unos tipova

U Rust-u nije potrebno specifično naglasiti tip prilikom kreiranje varijable već kompajler može sam da pretpostavi za vreme kompajliranja.

## Server MMO igre

### High-level arhitektura

Server se sastoji iz više modula kako bi se rasteretila sinhronizacija sveta. Struktura tih modula bi predstavljala stablo koja bi izolirala manje bitna dešavanja od viših slojeva, a takođe mogla da propagira dešavanja iz sveta ka nižim slojevima. Primeri slojeva: svet (vrhovni sloj), kontinent, oblast, regije i u slučaju većih gradova lokacije. Ovo omogućava da najniži slojevi stabla ne moraju imati konekcije između sebe već da mogu da proslede informacije sloju iznad i da će on da propagira informacije do prave lokacije. Takođe je potreban jedan ili više multipleksera koji će da otvara klijentske pakete i prosleđuje na pravi node i da vraća informacije nazad do klijenta.

### Rukovanje sa unosima igrača

Svaku akciju koju igrač izvrši neophodno je da server proveri da li je validna. Neke od tih provera su: da li igrač može da se pomeri na željenu lokaciju, da li može da napadne željenu žrtvu, da li je količina štete naneta dobra, da li može da pokupi neki predmet.

### Sinhronizacija sveta

Modul u okviru sveta bi bio zadužen za prosleđivanje komunikacije između igrača i upravljanje Guild-ovima.

Modul u okviru kontinenta i oblasti služe kao izolacijski slojevi.

Modul za regije je dužan da vodi računa o borbama, poziciji igrača, zadacima igrača.

Serveri bi radili na tick-rate petlji gde bi posle svakog tick-a slale informacije igraču o trenutnom stanju lokacije gde se nalazi. Pošto MMO igre nisu brze možemo imati mali tick-rate, na primer 10, i to bi značilo da se server ažurira svakih 100ms.

## Komponente Rust-a

### Rust kompajler

Rust koristi *rustc* kao svoj kompajler. On izvorni kod pretvara u objektni kod koji se pomoću linkera spaja u jedan izvršni fajl.

## LLVM

LLVM je set kompajlerskih i toolchain tehnologija potrebnih za pravljenje programskog jezika. On predstavlja osnovu rustc kompajlera i pomaže u optimizaciji. Prednost korišćenja LLVM je to što je standardizovan za korišćenje bez obzira na jezik za koji ga kompajler koristi.

## Paket menadžer

Paket menadžer služi za preuzimanje, kompajliranje i distribuciju biblioteka. Benefit paket menadžera je lakše upravljanje zavisnostima. Za Rust, standardni paket menadžer je Cargo. On korisničke pakete čuva na zvaničnom sajtu, ali može da preuzme neophodne pakete sa Git-a, lokalnog sistema ili nekog drugog eksternog izvora.

## Formatiranje koda

Formatiranje služi da ulepša i standardizuje izgled koda tako da sav kod, nebitno ko ga napisao, izgleda slično. Rustfmt je standardizovan alat za formatiranje Rust koda.

## Alat za sintaktičku proveru

Alat za sintaktičku proveru služi za proveru sintakse, grešaka prilikom kucanja koda pre kompajliranja. Prednost korišćenja ovog alata je ubrzanje generisanja koda i ispravljanje grešaka pre kompajliranja. Alat Clippy se koristi kao standardizovan za Rust.

## Alat za detekciju nepredviđenog stanja

Ovaj alat analizira izvršni kod u potrazi za greškama u logici. Neki primeri takvih grešaka su pristup memoriji izvan opsega ili memoriji koja je obrisana, pristupanje promenljivoj koja nije inicijalizovana i proverava da li je neki nedohvatljiv kod zapravo dohvatljiv.

## Std::env

Pomoću ove standardne biblioteke možemo da koristimo ulazne argumente, putanju i naziv izvršnog programa, i čitamo i dodajemo varijable okruženja trenutnog procesa.

## Std::fs

Standardna biblioteka za upravljanje sa fajl sistemom. Omogućava listanje, kreiranje i brisanje direktorijuma, i čitanje i pisanje fajlova.

## Std::time

Standardna biblioteka za rad sa vremenom. Pomoću strukture *Duration* lako upravljati vremenskim intervalima. Pomoću strukture *Instant* možemo uzeti trenutno vreme i meriti koliko je prošlo od kreiranje te strukture.

## Std::error

Standardizovana biblioteka za upravljanje sa greškama. Omogućava funkcijama da vraćaju rezultat u strukturi *Result* pomoću koje možemo lako proveriti da li se funkcija izvršila uspešno.

## Std::fmt

Standardna biblioteka za formatiranje i ispisivanje poruka. Omogućava upotrebu *format!* makroa koji prima običan string koji i menja {} u prosledene varijable. Ovo olakšava kreiranje stringova i logovanje poruka.

## Std::collections

Služi kao kolekcija standardnih struktura podataka. Neke od struktura koje su korisne su Vec koji predstavlja dinamički niz, LinkedList koji predstavlja listu, HashMap predstavlja mapu i HashSet predstavlja skup.

## Std::thread

Standardna biblioteka upravljanje nitima prirodnim za taj operativni sistem. Jedna od korisnih funkcionalnosti je postojanje Builder-a koja omogućava konfigurisanje niti pre njenog pravljenja.

## Std::net

Osnovna mrežna biblioteka. Omogućava kreiranje i upravljanje TCP/UDP konekcija, kao i upravljanje IP adresama.

## Hyper\_rustls

Ova biblioteka omogućava uspostavljanje HTTPS konekcije. Ona ujedinjuje dve biblioteke, Hyper koja upravlja HTTP konekcijom i Rustls koja olakšava rad sa sertifikatima.

## Quinn

Biblioteka za korišćenje QUIC protokola. Sa ovom bibliotekom uspostavljamo podršku za HTTP/3. Takođe je korisno kada nam je potreban veliki i brz protok gde redosled i pouzdanost paketa nije previše bitan.

# Implementacija

## Prihvatanje HTTP zahteva

Uz pomoć Rust paket menadžera možemo lako napraviti server koji može da prihvata HTTP zahteve.

```

use std::collections::HashMap;
use std::net::SocketAddr;
use std::sync::Arc;
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::Mutex;
use tokio_rustls::rustls::{self, ServerConfig};
use tokio_rustls::TlsAcceptor;
use std::time::{Duration, Instant};

```

```

let acceptor = TlsAcceptor::from(Arc::new(config));

// Bind to address
let addr: SocketAddr = "127.0.0.1:8443".parse()?;
let listener = TcpListener::bind(&addr).await?;

println!("Server listening on https://{addr}", addr);
println!("Connection pool enabled for CONNECT requests");

loop {
    let (stream, peer_addr) = listener.accept().await?;
    let acceptor = acceptor.clone();
    let pool = pool.clone();

    tokio::spawn(async move {
        match acceptor.accept(stream).await {
            Ok(tls_stream) => {
                if let Err(e) = handle_client(tls_stream, peer_addr, pool).await {
                    eprintln!("Error handling client {}: {}", peer_addr, e);
                }
            }
            Err(e) => {
                eprintln!("TLS accept error from {}: {}", peer_addr, e);
            }
        }
    });
}

```

## Prihvatanje UDP socket-a

```
// Bind UDP socket
let socket = Arc::new(UdpSocket::bind("127.0.0.1:7777").await?);

// Store connected players: SocketAddr -> player data
let players: Arc<Mutex<HashMap<SocketAddr, String>>> = Arc::new(Mutex::new(HashMap::new()));

let mut buffer = vec![0u8; 1024];

loop {
    // Receive UDP packet
    let (len, addr) = socket.recv_from(&mut buffer).await?;
```

## IPC

Pošto bi mogli bili odvojeni procesi zbog lakše skalabilnosti, potrebno je da imamo definisan komunikaciju između procesa. U našem slučaju ona bi se obavljana na pomoću UDS(Unix Domain Socket). Bismo taj tip jer je veoma brz i omogućava razmenjivanje poruka u oba smera. U slučaju da se moduli nalaze na različitim serverima, za njihovu komunikaciju ćemo koristiti QUIC protokol. On se izvršava pomoću UDP-a, ali zadržava neke od funkcionalnosti TCP protokola.

```
fn spawn_child(role: &str, socket_path: &str) -> Child {
    let exe_path = env::current_exe().expect("Failed to get current exe path");

    Command::new(exe_path)
        .arg(role)
        .arg(socket_path)
        .spawn()
        .expect(&format!("Failed to spawn {}", role))
}
```

```
fn run_child(role: &str, parent_socket: &str) {
    let pid = std::process::id();
    println!("[{} {}] Child process started", role, pid);

    // Wait a moment for parent to set up listener
    thread::sleep(Duration::from_millis(50));

    // Connect to parent
    let mut parent_stream = UnixStream::connect(parent_socket)
        .expect(&format!("[{} {}] Failed to connect to parent", role, pid));

    println!("[{} {}] Connected to parent via {}", role, pid, parent_socket);

    // Spawn grandchildren if this is ChildA or ChildB
    let mut grandchildren: Vec<(Child, UnixStream)> = Vec::new();

    match role {
```

## Komunikacije sa bazama

Svaka baza ima svoju biblioteku sa kojom Rust može da komunicira sa njom.



## PostgreSQL

```
// Connect to database
let mut client = Client::connect(
    "host=localhost user=postgres password=secret dbname=gamedb",
    NoTls,
)?;

client.execute(
    "INSERT INTO players (username, score, level) VALUES ($1, $2, $3)
    ON CONFLICT (username) DO NOTHING",
    &[&username, &score, &level],
)?;
```

## MongoDB

```
// Connect to MongoDB
let client_options = ClientOptions::parse("mongodb://localhost:27017").await?;
let client = Client::with_options(client_options)?;

// Get database and collection
let db = client.database("gamedb");
let collection = db.collection::<Player>("players");

// Find documents
let filter = doc! { "score": { "$gt": 5000 } };
let mut cursor = collection.find(filter, None).await?;
```

## Redis

```
// Connect to Redis
let client = redis::Client::open("redis://127.0.0.1:6379/");
let mut con = client.get_async_connection().await?;

println!("✓ Connected to Redis");

// STRING operations
con.set("player:1001:name", "warrior123").await?;
con.set("player:1001:score", 5000).await?;

let name: String = con.get("player:1001:name").await?;
let score: i32 = con.get("player:1001:score").await?;
```