

УНИВЕРЗИТЕТ У БЕОГРАДУ
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

Лабораторија за мултимедијалне комуникације

Пројектни рад из предмета:
АРХИТЕКТУРА РАЧУНАРА И ОПЕРАТИВНИ СИСТЕМИ

Тема:
ПРИМЕНА КОНКУРЕНТНОГ ПРОГРАМИРАЊА ЗА ИЗРАДУ
ВИШЕНИТНОГ ВЕБ СЕРВЕРА ЗА ПОТРЕБЕ ОРГАНИЗОВАЊА
ШАХОВСКИХ ТУРНИРА

Студент:
Стефан Јовановић
Бр. индекса: 2020/0002

Ментор:
Маја Миљанић

Београд, 2022.

Садржај

1	Прикупљање корисничких захтева	3
1.1	Вербални опис система.....	3
1.2	Случајеви коришћења	3
2	Анализа	4
2.1	Структура софтверског решења – Проширени модел објекти везе (ПМОВ).....	4
2.2	Структура софтверског решења – Релациони модел и речник података	5
3	Пројектовање.....	6
3.1	Конкурентност	6
3.2	Thread Pool модел	7
3.3	Имплементација Thread Pool модела.....	7
3.3.1	Worker структура	7
3.3.2	ThreadPool структура	8
3.4	Веб сервер	11
3.5	Имплементација веб сервера	11
3.5.1	Request структура	11
3.5.2	Route структура	13
3.5.3	Response структура.....	13
3.5.4	Fianchetto структура	13
3.6	Бекенд апликације	17
3.7	Имплементација бекенда	17
3.7.1	Модел базе података	17
3.7.2	Контролери.....	18
3.8	Фронтенд апликације.....	19
3.8.1	Изглед апликације	19
4	Закључак	23

1 Прикупљање корисничких захтева

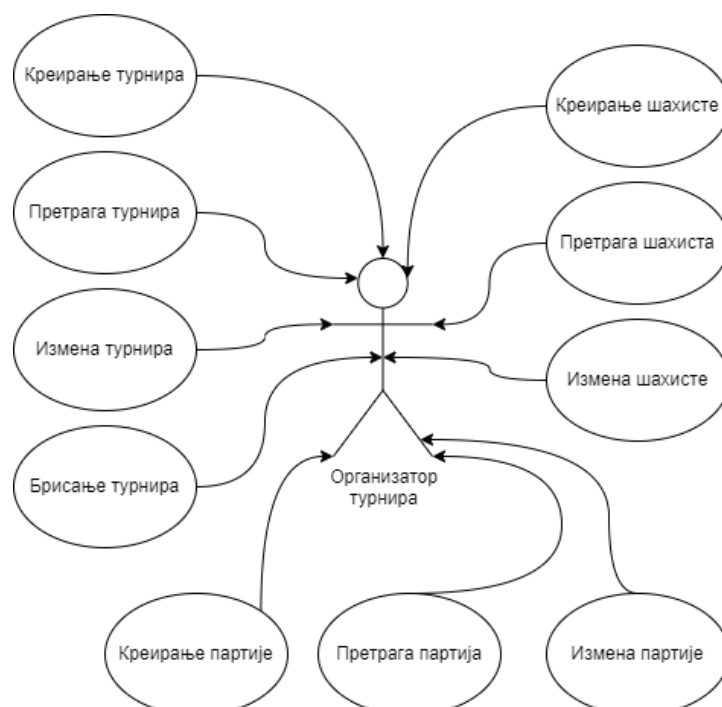
1.1 Вербални опис система

Организатор шаховског турнира организује турнир на одређеној локацији(уживо или онлајн) и са одређеним бројем рунди. Корисник (организатор) може да креира, обрише, прегледа и измени турнире. У оквиру сваког турнира корисник додаје (додаје их у PGN (Portable Game Notation) формату), претражује или мења партије. Партије се играју искључиво у формату „1 на 1“ где корисник може да дода, измени и прегледа све шахисте у систему.

1.2 Случајеви коришћења

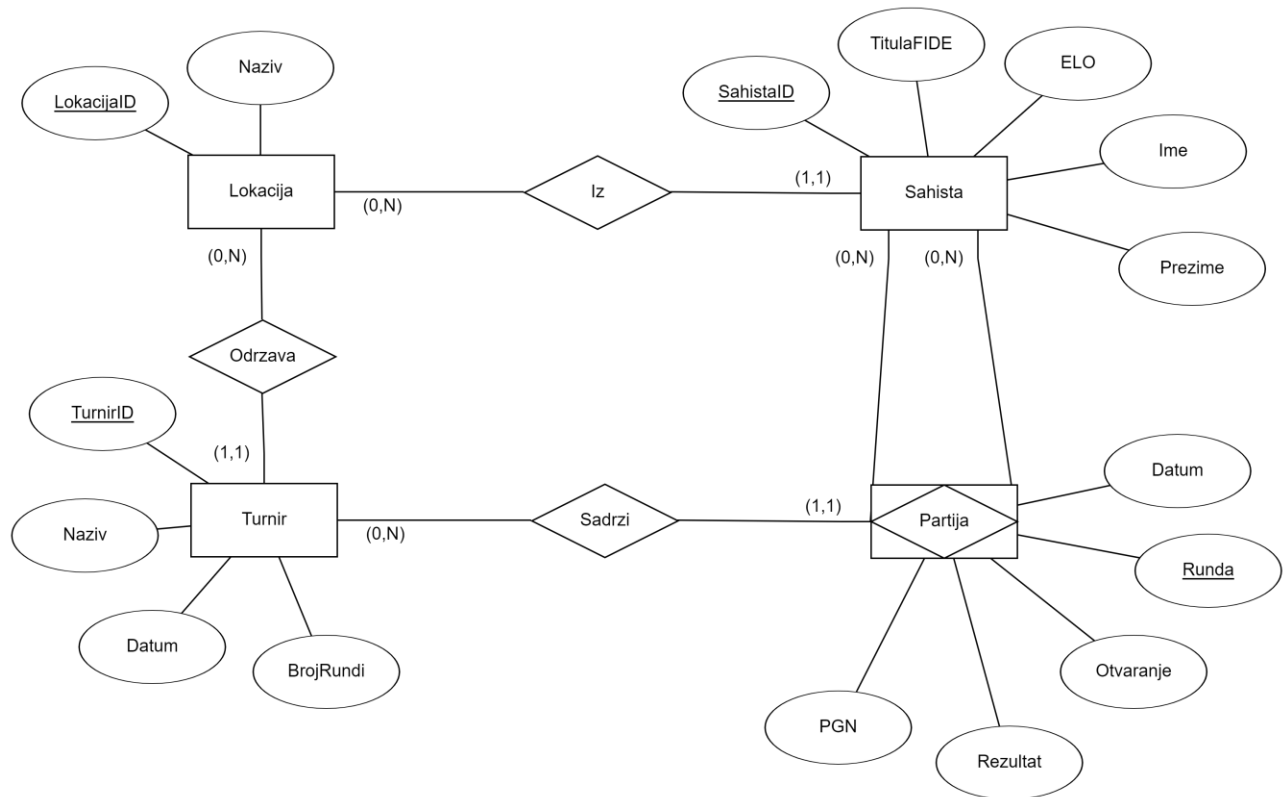
1. Креирање турнира
2. Претрага турнира
3. Измена турнира
4. Брисање турнира
5. Креирање партије
6. Претрага партија
7. Измена партије
8. Креирање шахисте
9. Претрага шахиста
10. Измена шахисте

Наведене случајеве користи организатор турнира.



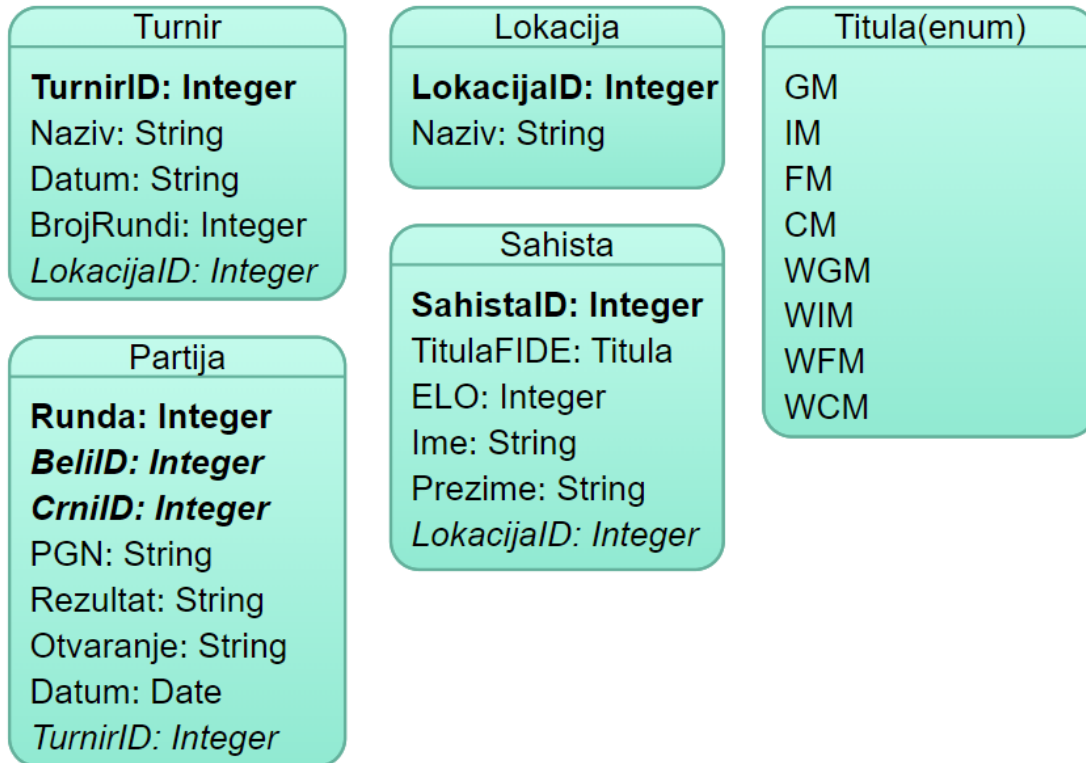
2 Анализа

2.1 Структура софтверског решења – Проширени модел објекти везе (ПМОВ)



2.2 Структура софтверског решења – Релациони модел и речник података

На основу проширеног модела објекти везе прави се одговарајући **релациони модел** и **речник података**.

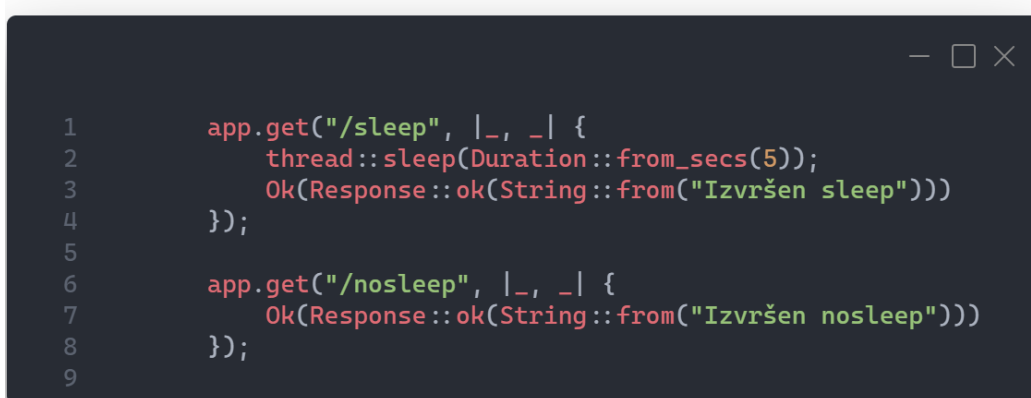


3 Пројектовање

3.1 Конкурентност

Читава примена конкурентности код веб сервера заснива се на идеји да се више HTTP захтева подели између више нити при чему ће онда свака нит бити одговорна за задати захтев и на њој је да тај захтев обради и пошаље одговарајући одговор. Ипак, поставља се питање зашто све захтеве не би обрађивала једна нит. Проблем са тим приступом најбоље ћемо илустровати на примеру.

Дат је следећи сегмент кода написан у програмском језику Rust:



```
1  app.get("/sleep", |_, _| {
2      thread::sleep(Duration::from_secs(5));
3      Ok(Response::ok(String::from("Izvršen sleep")))
4  });
5
6  app.get("/nosleep", |_, _| {
7      Ok(Response::ok(String::from("Izvršen nosleep")))
8  });
9
```

Слика 1 Две GET методе

Код садржи две GET методе, једну за руту **/sleep** и другу за руту **/nosleep**. Прва метода суспендује нит која извршава ту методу (у случају једнонитног веб сервера то је главна нит) на 5 секунди након чега враћа одговор статусног кода 200 са поруком **Izvršen sleep**. Друга метода, за разлику од прве, не суспендује тренутну нит већ само враћа одговор статусног кода 200 са поруком **Izvršen nosleep**. Уколико бисмо са једнонитног веб сервера послали GET захтев ка рути **/sleep**, а одмах након тога и ка рути **/nosleep** десило би се то да би нит била у суспендованом стању 5 секунди, након чега бисмо добили одговор од руте **/sleep**, а одмах након тога и од руте **/nosleep**. Другим речима, суспендованост нити узроковала је да за добијање одговора од обе руте морамо чекати приближно 5 секунди, иако само једна GET метода суспендује нит. Овај пример у ствари илуструје сценарио када један корисник захтева од сервера ресурсе за које је потребно дуже време обраде и због њега сваки други корисник који затражи било који други ресурс мора да сачека да се прво заврши обрада ресурса првог корисника како би процесор постао расположив и за остале захтеве. Управо овај проблем решава се применом конкурентног програмирања тако што обезбеђујемо више нити где би свака нит обрађивала по један кориснички захтев. Када би пример са **слике 1** покренули на вишенитном веб серверу оно што би се десило јесте да би сваки захтев добио своју нит која би га извршавала, што значи да суспендовање нити која извршава захтев на рути **/sleep** не би утицало на обраду захтева на рути **/nosleep**.

Крајњи излаз би био тај да би се захтев на рути /nosleep одмах извршио, док би се након приближно 5 секунди обрадио и захтев на рути /sleep.

3.2 Thread Pool модел

Thread pool је софтверски патерн који се састоји од групе нити које су спремне и чекају да им буде додељен задатак. Када добије задатак, програм га додељује једној од слободних нити у pool-у. Када та нит изврши додељени задатак она се враћа у pool и поново је доступна за неки нови задатак. Овај модел нам омогућава да захтеве обрађујемо конкурентно чиме повећавамо пропусност нашег сервера.

Ограничићемо број нити у нашем моделу на мањи број. Разлог овога је тај што се тиме штитимо од потенцијалних DoS (Denial of Service) напада. Да смо омогућили да програм ствара по једну нову нит за сваки захтев, малициозни корисник би могао да пошаље неколико милиона захтева што би узроковало пад нашег сервера.


Ово је само један од начина како можемо повећати пропусност нашег сервера. Од других модела истичу се fork/join модел и једнонитни асинхрони У/И модел.

3.3 Имплементација Thread Pool модела

За саму имплементацију овог модела битне су нам две структуре: **ThreadPool** и **Worker**.

3.3.1 Worker структура

Worker структура у ствари представља нит која ће извршавати задатке који јој буду додељени. Следећа слика представља приказ поља структуре:



```
1 pub struct Worker {
2     pub id: usize,
3     pub thread: Option<thread::JoinHandle<()>>,
4 }
5
```

Слика 2 Поља структуре Worker

Структура има два поља:

- **id** - представља идентификатор сваке нити
- **thread** – поље које представља саму нит

Над конструктором је имплементирана метода **new** која има улогу конструктора:

```
1 pub fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) -> Worker {
2     let thread = thread::spawn(move || loop {
3         let message = receiver.lock().unwrap().recv().unwrap();
4         match message {
5             Message::NewJob(job) => {
6                 println!("Worker {} got a job; executing.", id);
7
8                 job();
9             }
10
11             Message::Terminate => {
12                 println!("Worker {} was told to terminate.", id);
13                 break;
14             }
15         }
16     });
17
18     Worker {
19         id,
20         thread: Some(thread),
21     }
22 }
23
```

Слика 3 new метода структуре Worker

На почетку иницијализујемо променљиву thread методом thread::spawn којој прослеђујемо closure (анонимна функција). Променљива message представља поруку добијену од ThreadPool-а и примање и слање порука је омогућено помоћу библиотеке mpsc (multi-producer, single-consumer). Порука је enum и може имати вредности типа NewJob и Terminate. У првом случају нит добија задатак и извршава га. У другом случају терминирамо нит. На крају прослеђујемо тог новог Worker-а.

3.3.2 ThreadPool структура

ThreadPool структура руководи свим Worker-има тако што креира листу нити и додељује им задатке.

```
1 pub struct ThreadPool {
2     workers: Vec<Worker>,
3     sender: mpsc::Sender<Message>,
4 }
```

Слика 4 Поља структуре ThreadPool

Структура има два поља:

- **workers** - представља вектор који садржи Worker-e
- **sender** – променљива за слање задатака нитима, део библиотеке mpsc

Над структуром су имплементиране три методе: **new**, **execute** и **drop**.



```
1 pub fn new(size: usize) -> ThreadPool {
2     assert!(size > 0);
3
4     let (sender, receiver) = mpsc::channel();
5     let receiver = Arc::new(Mutex::new(receiver));
6
7     let mut workers = Vec::with_capacity(size);
8
9     for id in 0..size {
10         workers.push(Worker::new(id, Arc::clone(&receiver)));
11     }
12
13     ThreadPool { workers, sender }
14 }
```

Слика 5 new метода структуре ThreadPool

На слици 5 приказана је метода **new**. Метода прима параметар **size** који представља број Worker-a у pool-у. Креира се нови канал за комуникацију методом `mpsc::channel()` која враћа две инстанце које бивају смештене у променљиве **sender** и **receiver**. Помоћу променљиве **sender** ће **ThreadPool** слати поруке сваком од Worker-a, а они ће их примати помоћу променљиве **receiver**. С обзиром да једну поруку треба да прими само једна нит, **receiver** ће заправо бити **mutex** што постижемо помоћу методе `Mutex::new` (значење библиотеке **Arc** (Atomically Reference Counted) неће бити овде објашњено с обзиром да се ради о једној специфичности програмског језика Rust, више о томе се може прочитати овде: <https://doc.rust-lang.org/book/ch16-03-shared-state.html>). Након тога смештамо у низ **workers** онолико Worker-a колико је методи прослеђено.

Следећа метода је метода **execute**:

```
1 pub fn execute<F>(&self, f: F)
2 where
3     F: FnOnce() + Send + 'static,
4 {
5     let job = Box::new(f);
6
7     self.sender.send(Message::NewJob(job)).unwrap();
8 }
```

Слика 6 execute метода структуре ThreadPool

Ова метода као параметар прима анонимну функцију. У променљивој job креира се паметни показивач (smart pointer) који ће показивати на ту функцију и представља задатак за неког од Worker-a. Потом се помоћу променљиве sender шаље овај посао неком од слободних Worker-a.

Последња метода је метода **drop** и она има улогу деструктора:

```
1 fn drop(&mut self) {
2     println!("Sending terminate message to all workers.");
3
4     for _ in &self.workers {
5         self.sender.send(Message::Terminate).unwrap();
6     }
7
8     println!("Shutting down all workers.");
9
10    for worker in &mut self.workers {
11        println!("Shutting down worker {}", worker.id);
12
13        if let Some(thread) = worker.thread.take() {
14            thread.join().unwrap();
15        }
16    }
17 }
```

Слика 7 drop метода структуре ThreadPool

Прва for петља одговорна је за слање терминирајуће поруке свим Worker-има. Друга for петља за сваку нит позива методу join() која обезбеђује да се све нити терминирају пре него што се заврши извршавање програма.

3.4 Веб сервер

Након што је имплементиран ThreadPool модел следеће на реду је имплементација самог веб сервера. С обзиром да је намена веб сервера израда платформе за организацију шаховских турнира сервер је симболично назван **Fianchetto** (шаховски развој у којем је ловац развијен на другом реду суседне б или г колоне).

Развој сервера се састојао из неколико корака:

- Успоставити TCP конекцију на некон порту наше машине
- Парсирати пристигле HTTP захтеве
- Обезбедити одговарајуће одговоре за пристигле захтеве
- Укључити у цео сервер претходно имплементиран Thread Pool model

Дакле, два главна протокола коришћена у овом пројекту су **TCP** (Transmission Control Protocol) и **HTTP** (Hypertext Transfer Protocol). TCP је протокол транспортног слоја у ОСИ моделу и као такав описује детаље везане за то како се информације преносе са једног сервера на други, али не описује шта су заправо те информације. HTTP је протокол апликационог слоја у ОСИ моделу и као такав даје форму информацијама дефинишући HTTP захтеве и одговоре.

3.5 Имплементација веб сервера

У саму имплементацију укључено је неколико структура:

- **Fianchetto** – представља сам веб сервер и укључује методе попут get, post, delete, put
- **Request** – представља HTTP захтев и задужена је за парсирање добијеног садржаја
- **Route** – представља руту и укључује путању до те руте, HTTP методу као и функцију коју треба позвати на тој рути
- **Response** – представља различите HTTP одговоре

3.5.1 Request структура

У наставку је дата Request структура:

```
1 pub struct Request<'a> {
2     pub method: &'a str,
3     pub path: &'a str,
4     pub content_type: &'a str,
5     pub content_length: usize,
6     pub content: serde_json::Value,
7 }
```

Слика 8 Поља структуре Request

Сва поља наведена горе су неки од стандарних делова HTTP захтева. Сва ова поља добијају се парсирањем целокупног захтева који је преко TCP протокола добијен као низ бајтова. Ово парсирање врши се у методи `new` ове структуре чији је код дат у наставку:

```
1 pub fn new(request: &str) → Result<Request, serde_json::Error> {
2     let lines: Vec<&str> = request.split("\r\n").collect();
3     let first: Vec<&str> = lines.get(0).unwrap().split(" ").collect();
4
5     let mut method = "";
6     if let Some(m) = first.get(0) {
7         method = m;
8     }
9     let mut path = "";
10    if let Some(p) = first.get(1) {
11        path = p;
12    }
13
14    let mut content_type = "";
15    let mut content_length: usize = 0;
16    let mut content_started = false;
17    let mut content = String::from("");
18
19    for line in lines {
20        if line.contains("Content-Type") {
21            let content_type_vec: Vec<&str> = line.split(":").collect();
22            content_type = content_type_vec.get(1).unwrap();
23            content_type = &content_type[1..];
24        } else if line.contains("Content-Length") {
25            let content_length_vec: Vec<&str> = line.split(":").collect();
26            let mut length: &str = content_length_vec.get(1).unwrap();
27            length = &length[1..];
28            content_length = length.parse().unwrap();
29        } else if line.contains("{") {
30            content_started = true;
31        }
32
33        if content_started {
34            content += line;
35        }
36    }
37    let mut content_json = Value::Null;
38    if content_type.to_lowercase().contains("json") {
39        let json = content.trim_matches('\u{0}');
40        content_json = serde_json::from_str(json)?;
41    }
42
43    Ok(Request {
44        method,
45        path,
46        content_type,
47        content_length,
48        content: content_json,
49    })
50 }
```

Слика 9 `new` метода структуре `Request`

3.5.2 Route структура

Route структура представља руту и укључује путању до те руте, HTTP методу као и функцију коју треба позвати на тој рути:

```
1 pub type RouteAction = Box<
2     dyn Fn(super::request::Request, &Params) → Result<String, Box<dyn std::error::Error>>
3     + Send
4     + Sync,
5 >;
6
7 pub struct Route {
8     pub path: String,
9     pub method: String,
10    pub action: RouteAction,
11 }
```

Слика 10 Поља структуре Route

Структура садржи поља за путању, HTTP методу као и за одговарајућу акцију која треба бити предузета, у облику анонимне функције.

3.5.3 Response структура

Ова структура одговорна је за обезбеђивање HTTP одговора одговарајућег формата. Ово укључује одговоре са статусним кодовима 200, 201, 400, 404 и 204.

3.5.4 Fianchetto структура

Fianchetto структура садржи следећа поља:

```
1 pub struct Fianchetto {
2     listener: TcpListener,
3     pool: ThreadPool,
4     routes: HashMap<'static str, Vec<Route>>,
5 }
```

Слика 11 Поља структуре Fianchetto

Поље **listener** представља променљиву типа TcpListener помоћу којег ћемо успоставити TCP конекцију на нашем серверу. Поље **pool**, као што се може и видети, представља инстанцу структуре ThreadPool. У пољу **routes** смештаће се све методе за појединачне руте. Ово је остварено помоћу структуре података HashMap где се као кључ користи

string који представља релативну путању, а као вредност вектор чије су вредности типа Route. Сва ова поља се иницијализују у методи **new**:

```
1 pub fn new(address: &str, num_of_threads: usize) → Self {
2     let listener = TcpListener::bind(address).unwrap();
3     let pool = ThreadPool::new(num_of_threads);
4     let routes = HashMap::new();
5
6     Fianchetto {
7         listener,
8         pool,
9         routes,
10    }
11 }
```

Слика 12 new метода структуре Fianchetto

Променљивој routes се додају одговарајуће руте у методи **request**:

```
1 fn request<F>(&mut self, path: &'static str, callback: F, method: String)
2 where
3     F: Fn(Request, &Params) → Result<String, Box<dyn std::error::Error>>
4         + Send
5         + Sync
6         + 'static,
7 {
8     let action = Box::new(callback);
9
10    let route = Route {
11        path: String::from(path),
12        method,
13        action,
14    };
15
16    if !self.routes.contains_key(path) {
17        let vec = vec![route];
18        self.routes.insert(path, vec);
19    } else {
20        self.routes.get_mut(path).unwrap().push(route);
21    }
22 }
```

Слика 13 request метода структуре Fianchetto

Проверава се да ли HashMap-а већ садржи дати кључ и на основу тога се дата рута или додаје у постојећи вектор, или се креира нови вектор.

Метода request позива се из четири методе: **get**, **post**, **put** и **delete** које у ствари представљају одговарајуће HTTP методе.

Следећа метода јесте метода **listen**:

```
1 pub fn listen(&mut self) {
2     let mut router = Router::new();
3     self.set_router(&mut router);
4     let arc_router = Arc::new(router);
5
6     for stream in self.listener.incoming() {
7         let stream = stream.unwrap();
8         let router = Arc::clone(&arc_router);
9
10        self.pool.execute(move || {
11            if let Err(result) = handle_connection(stream, router) {
12                println!("Error: {}", result);
13            }
14        });
15    }
16
17    println!("Shutting down server!");
18 }
```

Слика 14 listen метода структуре Fianchetto

На почетку се креира променљива router из библиотеке route_recognizer помоћу која служи да парсира URI и над њом се позива метода **set_router**:

```
1 fn set_router(&mut self, router: &mut VecRouter) {
2     for (path, routes) in self.routes.drain() {
3         router.add(path, routes);
4     }
5 }
```

Слика 15 set_router метода структуре Fianchetto

Ова метода пролази кроз све руте у променљивој routes и додаје их у променљиву router уз одговарајућу путању као кључ. Након тога се за сваки пристигли захтев ангажује по један Worker из pool-а и позива се метода handle_connection за тај пристигли захтев. Уколико дође до неке грешке приликом обраде захтева та грешка се исписује на стандардном излазу.

```

1 fn handle_connection(
2     mut stream: TcpStream,
3     router: Arc<VecRouter>,
4 ) → Result<(), Box<dyn std::error::Error>> {
5     let mut buffer = [0; 8192];
6     stream.read(&mut buffer).unwrap();
7
8     let buffer_str = str::from_utf8(&buffer)?;
9
10    let request = Request::new(buffer_str)?;
11
12    let route_match = router.recognize(request.path)?;
13    let routes: &Vec<Route> = route_match.handler();
14    let mut response = Response::bad_request();
15    for route in routes {
16        //preflight
17        if request.method.eq("OPTIONS") {
18            response = Response::no_content();
19            break;
20        } else if route.method.eq(request.method) {
21            let response_res = (route.action)(request, route_match.params());
22            match response_res {
23                Ok(r) ⇒ response = r,
24                Err(err) ⇒ {
25                    let err = err.to_string();
26                    let err_json = json!({ "err": err });
27                    response = Response::bad_request_body(serde_json::to_string(&err_json)?);
28                }
29            }
30            break;
31        }
32    }
33    stream.write(response.as_bytes())?;
34    stream.flush()?;
35
36    Ok(())
37 }

```

Слика 16 *handle_connection* метода структуре *Fianchetto*


Метода `handle_connection`, као што јој име и каже, обрађује сам захтев који је добијен. Садржај захтева се прво чита као низ бајтова након чега бива конвертован у стринг. Креира се нова инстанца типа `Request` и провера се да ли дата рута може бити обрађена на серверу. Оно што је битно увидети овде јесте линија 17 која је заслужна за обраду `preflight` захтева који је део `CORS` механизма. Након што се нађе одговарајућа рута, позива се акција те руте (линија 21). Метода враћа одговарајући `HTTP` захтев, такође као низ бајтова, и позива наредбу `stream.flush()` којом се сав садржај уписује на дати ток.

3.6 Бекенд апликације

Након што је имплементиран сам бекенд оквир остаје само да се пројектује бекенд наше апликације. Оно што нам и даље фали јесте ORM као апстракција наше базе података. Овде је коришћен **diesel** ORM у комбинацији са PostgreSQL базом података.

3.7 Имплементација бекенда

Почетна тачка наше апликације јесте **main** метода:



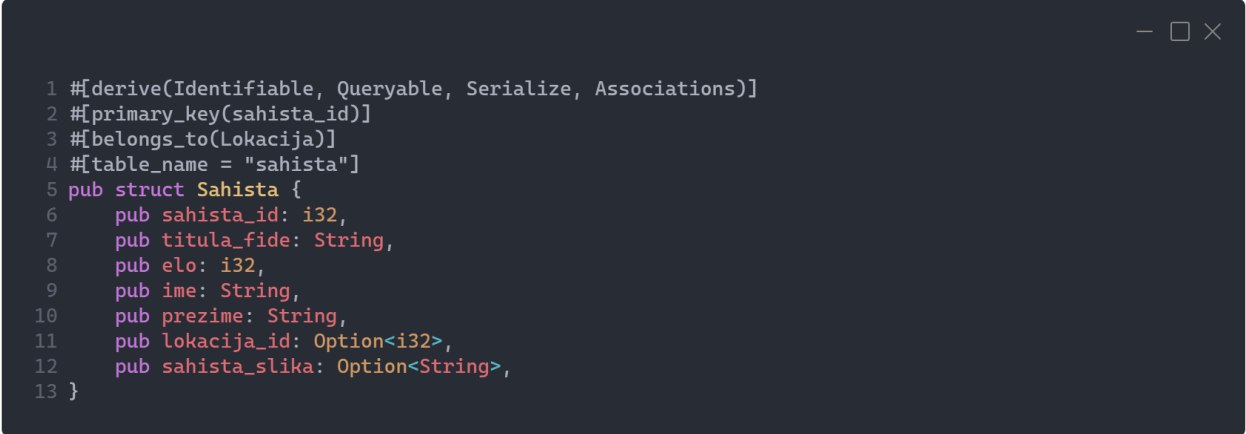
```
1 fn main() {
2     let mut app: Fianchetto = Fianchetto::new("127.0.0.1:1207", 4);
3     let db_conn = Arc::new(establish_connection());
4
5     LokacijaController::routes(&mut app, Arc::clone(&db_conn));
6     TurnirController::routes(&mut app, Arc::clone(&db_conn));
7     SahistaController::routes(&mut app, Arc::clone(&db_conn));
8     PartijaController::routes(&mut app, Arc::clone(&db_conn));
9
10    app.listen();
11 }
```

Слика 17 *main* метода апликације

Променљива **app** представља наш веб сервер и иницијализује се на одређену адресу и порт и прослеђује јој се број нити који ћемо користити. Потом се креира променљива која ће чувати конекцију са базом података. За сваки од модела се потом позивају одговарајући контролери и позива се наредба `app.listen()`.

3.7.1 Модели базе података

Модели наше апликације представљени су као структуре, а одговарајуће везе између њих успостављају се коришћењем одговарајућих макроя који су нам доступни из библиотеке **diesel**:



```
1 #[derive(Identifiable, Queryable, Serialize, Associations)]
2 #[primary_key(sahista_id)]
3 #[belongs_to(Lokacija)]
4 #[table_name = "sahista"]
5 pub struct Sahista {
6     pub sahista_id: i32,
7     pub titula_fide: String,
8     pub elo: i32,
9     pub ime: String,
10    pub prezime: String,
11    pub lokacija_id: Option<i32>,
12    pub sahista_slika: Option<String>,
13 }
```

Слика 18 *Модел табеле Sahista*

3.7.2 Контролери

Сви контролери имплементирају trait (може се посматрати као интерфејс код објектно-оријентисаног програмирања) **Controller** који има дефинисану методу **routes**:

```
1 pub trait Controller {  
2     fn routes(app: &mut Fianchetto, conn_pool: Arc<Pool<ConnectionManager<PgConnection>>>);  
3 }
```

Слика 19 Controller trait

У контролерима су дефинисане одговарајуће HTTP методе које су нам доступне од стране нашег бекенд оквира:

```
1 let conn = Arc::clone(&conn_pool);  
2 app.put("/partija/:id", move |req, params| {  
3     let partija_id: i32 = params.find("id").unwrap().parse()?;  
4     let upd_partija: NewPartija = serde_json::from_value(req.content)?;  
5     let partija: Partija =  
6         PartijaController::update_partija(&conn.get().unwrap(), partija_id, upd_partija)?;  
7  
8     let partija_json = serde_json::to_string(&partija)?;  
9     Ok(Response::ok(partija_json))  
10 });
```

Слика 20 Put метода позвана у контролеру за табелу Partija

Можемо видети да је путем URI-ја омогућено и прослеђивање параметара који се смештају у променљиву params.

У контролерима је делимично примењен и repository патерн:

```
1 fn update_partija(conn: &PgConnection, id: i32, partija: NewPartija) -> Result<Partija, Error> {  
2     diesel::update(dsl::partija.find(id))  
3         .set((  
4             dsl::beli_id.eq(partija.beli_id),  
5             dsl::crni_id.eq(partija.crni_id),  
6             dsl::datum.eq(partija.datum),  
7             dsl::otvaranje.eq(partija.otvaranje),  
8             dsl::pgn.eq(partija.pgn),  
9             dsl::rezultat.eq(partija.rezultat),  
10            dsl::runda.eq(partija.runda),  
11            dsl::turnir_id.eq(partija.turnir_id),  
12        ))  
13     .get_result(conn)  
14 }
```

Слика 21 Помоћна метода update_partija дефинисана у контролеру

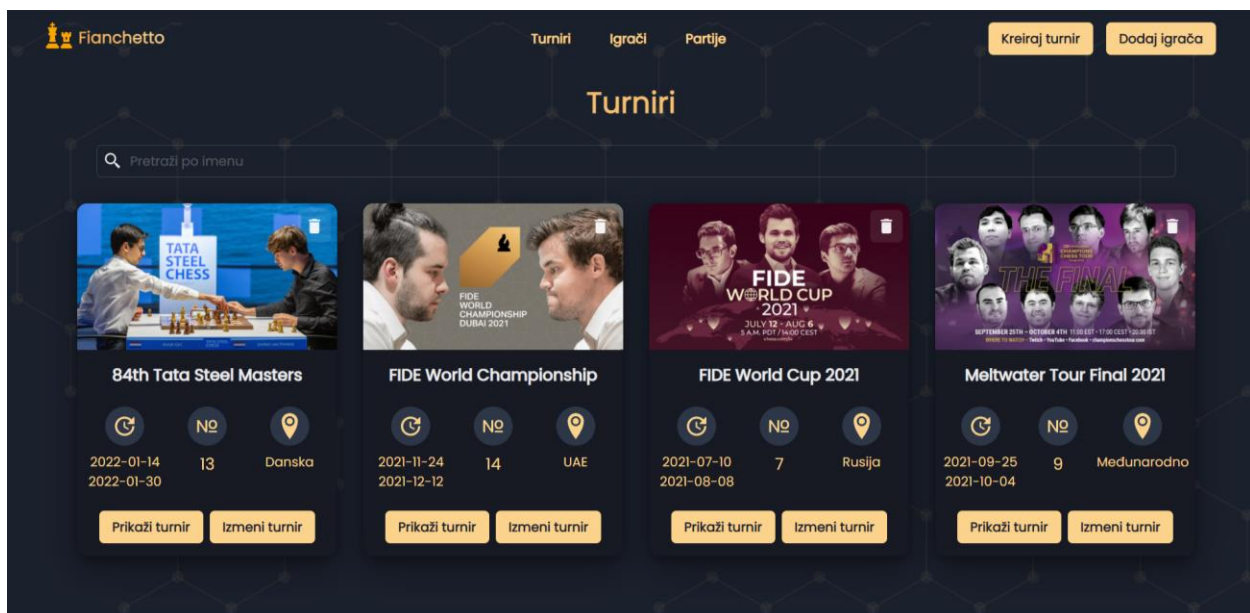
3.8 Фронтенд апликације

Фронтенд апликације имплементиран је помоћу библиотеке React и оквира Next.js, уз CSS оквир Chakra. Језик који је коришћен је typescript. Апликација је направљена као SPA (Single Page Application).

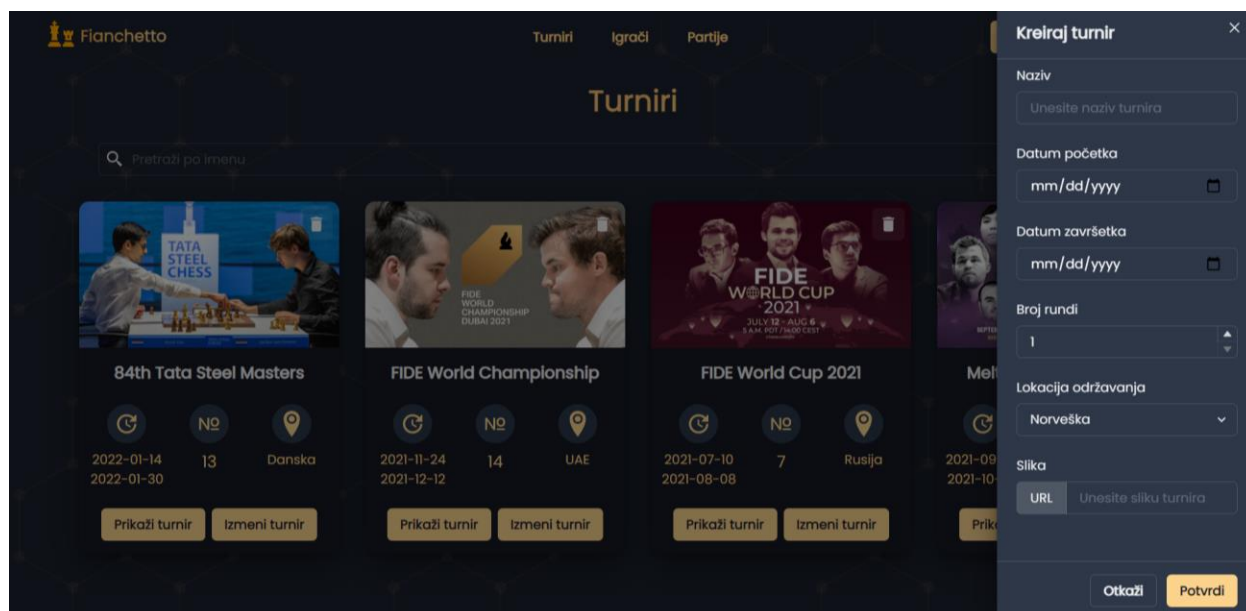
3.8.1 Изглед апликације



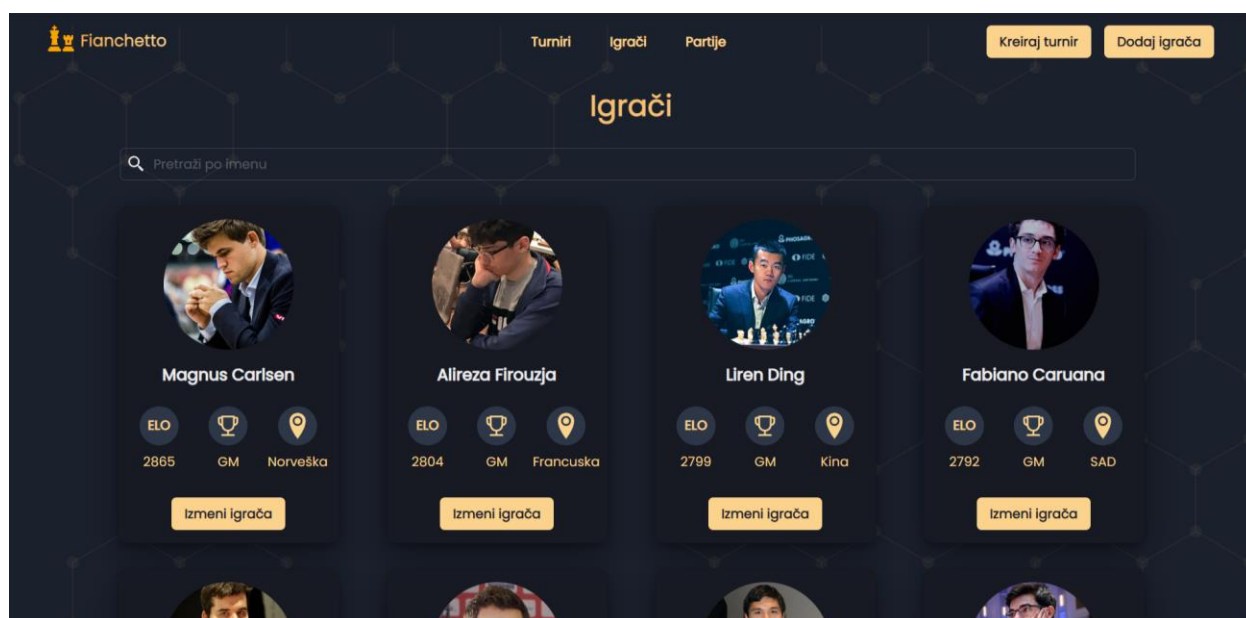
Слика 22 Почетна страна апликације



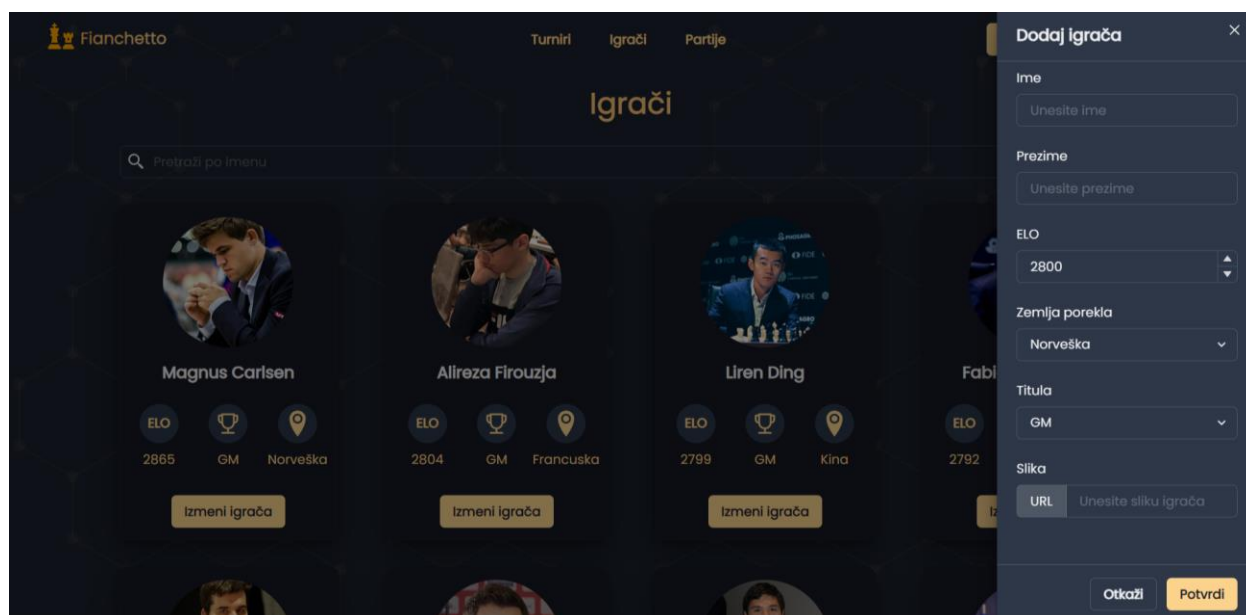
Слика 23 Страница за приказ свих турнира



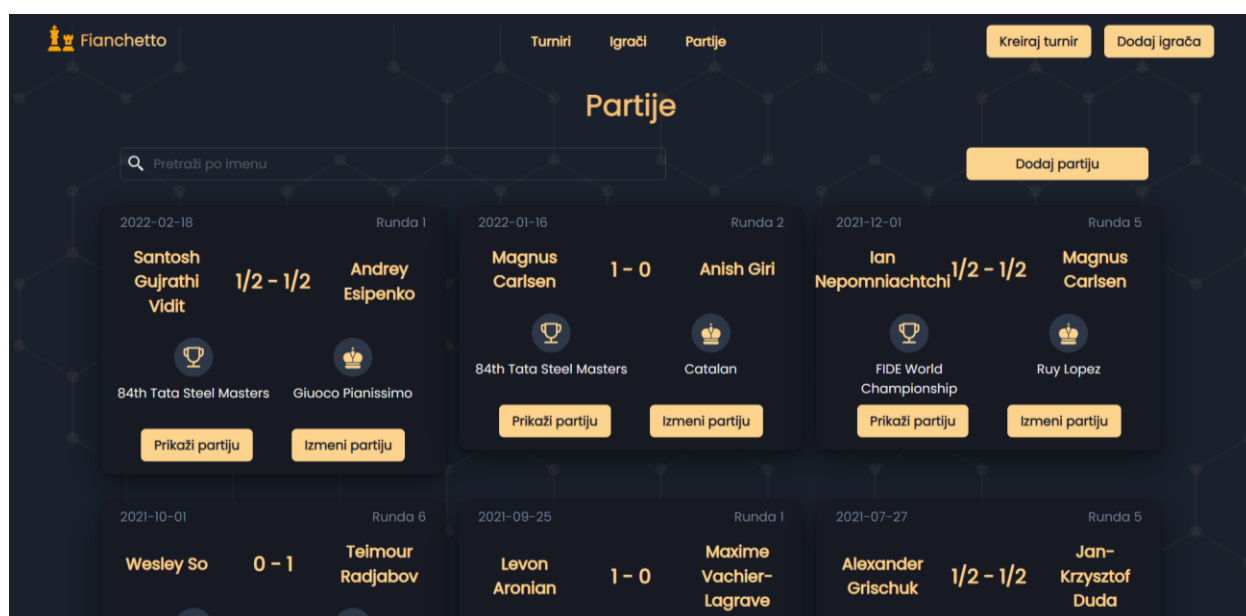
Слика 24 Мени за креирање новог турнира



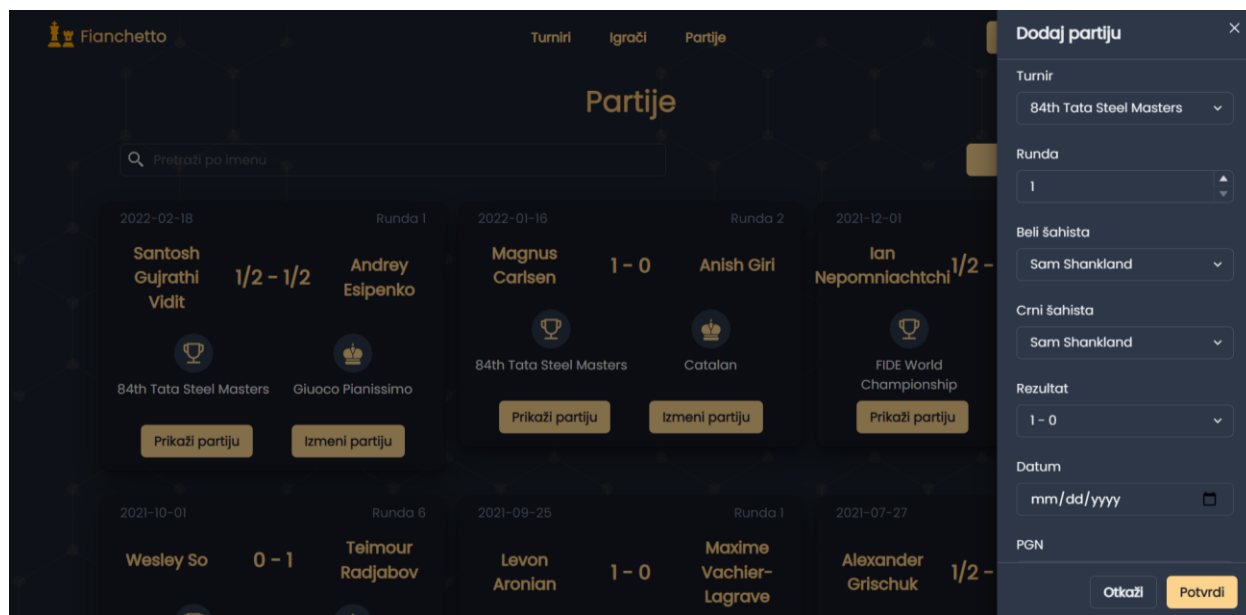
Слика 25 Страница за приказ свих играча



Слика 26 Мени за креирање новог играча



Слика 27 Страница за приказ свих партија



Слика 28 Мени за додавање нове партије



Слика 29 Страница за приказ партије

4 Закључак

У овом пројектном раду приказан је начин на који можемо повећати пропусност нашег система применом конкурентног програмирања. Притом је коришћен програмски језик Rust који нам нуди брзину и ефикасност језика нижег нивоа, попут C-а, али у исто време исправљајући неке од честих проблема као што је управљање меморијом. Језик нуди и бројне функционалности које су присутне у језицима вишег нивоа, као што су концепти из функционалног програмирања, објектно оријентисани патерни и др. Управо ово нам је омогућило да бирамо ниво апстракције који нама одговара, не бринући о томе да ли ћемо приступити неком забрањеном делу меморије или ће доћи до потпуног застоја, притом остварујући високе перформансе (zero cost abstraction).

Остало је ипак још доста ствари које је могуће унапредити у самом пројекту:

- Већи број HTTP одговора
- Правилно управљање CORS механизмом
- Имплементација веб сокета
- Боља интеграција са ORM-ом
- Обрада захтева чији подаци нису у JSON формату
- Примена Rust workspace-а за бољу организацију самог пројекта

Поред свега неимплементираног, сматрам да је ипак постављена одлична база која се може даље развијати и унапређивати и ово би могао бити одличан micro framework за апликације мање комплексности које треба имплементирати у краћем временском периоду.