TEHNIČKA DOKUMENTACIJA

Server Manager Application Tehnical Documentation

Ožujak 2024.

SERVER MANAGER APPLICATION

Tehnical Documentation

Izradio:

univ. mag. inf. Filip Pavliš

Bjelovar, 3. ožujak 2024.

Sadržaj

1. Tehnička dokumentacija

U ovome dokumentu opisat će se tehnička dokumentacije aplikacije Server Manager Application. SMA je aplikacija koja omogućuje korisniku pregled i uređivanje odnosno praćenje servera. Aplikacija nudi različite funkcionalnosti koje će se u sljedećim poglavljima prikazati i detaljnije objasniti. Najprije će se objasniti i navesti sve tehnologije i alati korišteni pri izradi ovog projekta. Nakon toga će se prikazati model baze podataka. Zatim će se objasniti pojedine funkcionalnosti aplikacije detaljnjim obješnjenjem korištenih metoda.

2. Korištene tehnologije i alati

Za izradu aplikacije Server Manager korišteno je više alata. Baza podataka je napravljena koristeći MySQL. MySQL se temelji na modelu klijent-poslužitelj. Jezgra MySQL-a je MySQL poslužitelj koji upravlja svim uputama (ili naredbama) baze podataka.

Za backend dio projekta korišten je IntelliJ IDEA 2023.3.4. U njemu je napravljen Spring Boot projekt pomoću Spring Initializr project wizarda.

Front end dio web aplikacije izrađen je pomoću platforme Angular. Verzija koja se koristila je sljedeća: Angular CLI: 16.0.6, Angular-16.2.12. Angular je besplatni okvir webaplikacije otvorenog koda. Korišteni jezik prilikom izrade frontend dijela je TypeScript. HTML i CSS su također korišteni radi izrade potrebnog dizajna web stranice.

3. Baza podataka

Pošto se radi o dosta jednostavnijoj aplikaciji, baza podataka ima jednu glavnu tablicu iz koje su dobiveni svu potrebni podaci. Ta tablica nazvana je server i ima sljedeće atribute:

```
28 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Server {
    @Id
    @GeneratedValue(strategy = AUTO)
    private Long id;
    @Column(unique = true)
    @NotEmpty(message = "IP Address cannot be empty or NULL")
    private String ipAddress;
    private String name;
    private String location;
    private String type;
    private String imageUrl;
    private Status status;
}
```

Slika 1. Model Tablice Server – Klasa Server

Svaki server ima svoj ID koji je ujedno i primarni ključ tablice. ID je tipa Long. Nadalje, svaki server posjeduje svoju IP adresu. IP adresa je tipa String i ima dva ograničenja: ne smije biti prazna ili NULL, te mora biti jedinstvena.

Svaki server također ima: ime (name), lokaciju (location), tip (type) i url slike (imageUrl) koji su tipa String. Zadnji atribut koji svaki server posjeduje je status tipa Status. Status je enumeracija i može biti („SERVER_UP“ ili „SERVER_DOWN“).

4. Glavne funkcionalnosti aplikacije

U ovome poglavlju opisat će se glavne funkcionalnosti Server Manager Aplikacije. Prikazati će se sve glavne metode koje se nalaze na front i back endu aplikacije uz detaljnije informacije i jasna objašnjenja korištenih metoda.

4.1. Pregled servera

Svaki put kada korisnik uključi aplikaciju treba mu se prikazati popis svih dostupnih servera iz baze podataka. U angularu je to moguće prikazati odmah kada se inicijalizira pojedina komponenta metodom *ngOnInit().*

```
ngOnInit(): void {
  this.appState$ = this.serverService.servers$
  .pipe(
    map(response => {
      this.dataSubject.next(response);
      return { dataState: DataState.LOADED_STATE, appData: { ...response,
        data: { servers: response.data.servers.reverse()}}}
    }),
    startWith({dataState: DataState.LOADING_STATE}),
    catchError((error: string) => {
      return of({ dataState: DataState.ERROR_STATE, error })
    })
  );
}
```

Slika 2. ngOnInit() metoda.

Ta metoda prikazuje popis svih trenutačnih servera u obrnutom redoslijedu. Observeable appState$ ima tri propertya: dataState, appData i error. Property dataState omogućuje praćenje stanja aplikacije (LOADING_STATE, LOADED_STATE ili ERROR_STATE). Property appData omogućuje prikaz podataka korisniku. Property error služi za hvatanje mogućih pogrešaka.

Server service služi za dohvaćanje liste servera koji se nadalje šalju propertyu appData i to u obrnutom redosljedu (servers: response.data.servers.reverse()). Prilikom prihvaćanja

podataka aplikacije je u LOADING_STATEu, prilikom dohvata je u LOADED_STATEu. Ako se dogodi koja pogreška aplikacija se tada nalazi u ERROR_STATEu. Server Service ima observerable servers$ koji šalje HTTP GET request za dohvaćanje svih servera:

```
servers$ = <Observable<CustomResponse>>
this.http.get<CustomResponse>(`${this.apiUrl}/server/list`)
.pipe(
  tap(console.log),
  catchError(this.handleError)
);
```

Slika 3. Observable servers$

Na backend dijelu aplikacije, kontroler metoda getServers() handlea taj HTTP GET request.

```
@GetMapping("/list")
public ResponseEntity<Response> getServers(){
    return ResponseEntity.ok(
            Response.builder() ResponseBuilder<capture of ?, capture of ?>
                    .timestamp(now()) capture of ?
                    .data(Map.of( k1: "servers", serverService.list( limit: 30)))
                    .message("Servers retrieved")
                    .status(OK)
                    .statusCode(OK.value())
                    .build()

    );
}
```

Slika 4. getServers() metoda

Ta metoda dohvaća listu servera, konstruira response object koji sadrži informacije o serverima i ostalim podacima i vraća HTTP status code od 200 (OK). Lista servera se dobije pomoću metode serverService.list(30) (za dohvaćanje maximum 30 servera).

```
1 usage
@Override
public Collection<Server> list(int limit) {
    log.info("Fetching all servers");
    return serverRepo.findAll(PageRequest.of( pageNumber: 0, limit)).toList();
}
```

Slika 5. list(int limit) metoda

Pomoću te metode dohvaća se kolekcija servera koristeći serverRepo koji je tipa ServerRepo. ServerRepo je interface koji nasljeđuje Spring Dana JPA repository.

Tu funkcionalnost konačno završava te se korisniku prikazuje popis svih servera koji se nalaze u bazi podataka.

4.2. Filtriranje servera

Filtriranje servera vrši se pomoću funkcije filterServers(status: Status) koja kao parametar uzima odabran status od strane korisnika.

```
filterServers(status: Status): void {
  this.appState$ = this.serverService.filter$(status, this.dataSubject.value)
  .pipe(
    map(response => {
      return { dataState: DataState.LOADED_STATE, appData: response}
    }),
    startWith({dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}),
    catchError((error: string) => {
      return of({ dataState: DataState.ERROR_STATE, error });
    })
  );
}
```

Slika 6. Metoda filterServers()

To je jednostavna funkcija koja filtrira servere obzirom na njihov status. To čini tako da ažurira appState$ observeable i daje mu odgovarajuće filtrirane podatke odnosno servere za prikaz. Pomoću metode filter$ se dobiju odgovarajući podaci za prikaz.

```
filter$ = (status: Status, response: CustomResponse) => <Observable<CustomResponse>>
new Observable<CustomResponse>(
  Subscriber => {
    console.log(response);
    Subscriber.next(
      status === Status.ALL ? { ...response, message: `Server filtered by ${status} status`} :
      {
        ...response,
        message: response.data.servers.filter(server => server.status === status).length > 0 ? `Server filtered by
        ${status === Status.SERVER_UP ? 'SERVER UP' : 'SERVER DOWN'} status` : `No server of ${status} found`,
        data: { servers: response.data.servers.filter(server => server.status === status)}
      }
    );
    Subscriber.complete();
  }
)
.pipe(
  tap(console.log),
  catchError(this.handleError)
);
```

Slika 7. Metoda filter&

Metoda filter$ uzima odabrani status i trenutni popis servera kao argumente, a vraća observable tipa CustomResponse. Ostatak responsea će uvijek biti isti, no message i data se mijenjaju ovisno o odabranom statusu. Podaci, odnosno data se filtrira pomoću ove funkcije:

```
data: { servers: response.data.servers.filter(server => server.status === status)}
```

Dakle, ako je status (SERVER_UP) funkcija će provjeriti svaki status servera iz danog popisa i odabrati samo one koje zadovoljavaju uvjet. Nakon toga vraća se filtrirana lista servera i prikazuje se korisniku. Tu funkcionalnost završava.

4.3. Kreiranje servera

Funkcionalnost kreiranja servera započinje funkcijom saveServer koja kao parametre uzima podatke iz forme.

```
saveServer(serverForm: NgForm): void {
  this.isLoading.next(true);
  this.appState$ = this.serverService.save$(serverForm.value as Server)
  .pipe(
    map(response => {
      this.dataSubject.next(
        {...response, data: { servers: [response.data.server, ...this.dataSubject.value.data.servers] } }
      );
      document.getElementById('closeModal').click();
      this.isLoading.next(false);
      serverForm.resetForm({status: this.Status.SERVER_DOWN});
      return { dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}
    }),
    startWith({dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}),
    catchError((error: string) => {
      this.isLoading.next(false);
      return of({ dataState: DataState.ERROR_STATE, error })
    })
  );
}
```

Slika 8. Metoda saveServer()

U suštini, ta funkcija sprema podatke iz forme i šalje ih funkciji save$. Kada se spremi server u bazu dobije nazad podatke o tom spremljenom serveru i šalje ih propertyu appData kako bi korisnik odmah vidio dodani server. Važno je naglasiti da spremljeni server vraća na prvo

mjesto arraya kako bi ga korisnik mogao odmah vidjeti na prvom mjestu u tablici svih servera. To se dobije ovom linijom koda:

```
{ servers: [response.data.server, ...this.dataSubject.value.data.servers] }
```

Metoda save$ šalje HTTP POST request da bi se uspješno spremili podaci o novo dodanom serveru:

```
save$ = (server: Server) => <Observable<CustomResponse>>
this.http.post<CustomResponse>(`${this.apiUrl}/server/save`, server)
.pipe(
  tap(console.log),
  catchError(this.handleError)
);
```

Slika 9. Metoda save$

Nadalje, funkcija saveServer handlea taj HTTP POST request.

```
@PostMapping("/save")
public ResponseEntity<Response> saveServer(@RequestBody @Valid Server server){
    return ResponseEntity.ok(
            Response.builder() ResponseBuilder<capture of ?, capture of ?>
                    .timestamp(now()) capture of ?
                    .data(Map.of( k1: "server", serverService.create(server)))
                    .message("Server created!")
                    .status(CREATED)
                    .statusCode(CREATED.value())
                    .build()
    );
}
```

Slika 10. Metoda saveServer()

Ta funkcija kreira novi Server objekt iz tjela requesta, te ga sprema koristeći serverService.create(server) metodu, te na kraju vraća HTTP response koji indicira uspješnost operacije.

```java
@Override
public Server create(Server server) {
    log.info("Saving new server: {}", server.getName());
    server.setImageUrl(setServerImageUrl());
    return serverRepo.save(server);
}
```

Slika 11. Metoda create()

Metoda create uzima server kao argument. Trenutno su svi podaci servera uneseni osim slike servera. Zbog toga se još jedna linija koda izvršava prije spremanja servera kako bi se dodijelila odgovarajuća slika servera. Nakon toga se koristi serverRepo JPA repositorijom da bi se server spremio u bazu. Tu funkcija kreiranja servera završava.

4.4. Pinganje servera

Funkcija pinganja servera započinje metodom pingServer() koja uzima IP adresu servera kao svoj parametar.

```typescript
pingServer(ipAddress: string): void {
  this.filterSubject.next(ipAddress);
  this.appState$ = this.serverService.ping$(ipAddress)
  .pipe(
    map(response => {
      const index = this.dataSubject.value.data.servers.findIndex(server => server.id === response.data.server.id);
      this.dataSubject.value.data.servers[index] = response.data.server;
      this.filterSubject.next('');
      return { dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}
    }),
    startWith({dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}),
    catchError((error: string) => {
      this.filterSubject.next('');
      return of({ dataState: DataState.ERROR_STATE, error })
    })
  );
}
```

Slika 12. Metoda pingServer()

U suštini, ova metoda pinga server s njegovom ip adresom, nakon toga ažurira UI ovisno o statusu servera. Važno je da se pronađe indeks onog servera kojeg se pinga, te da se samo njegovo stanje ažurira.

Ova metoda poziva ping$ funkciju pomoću koje se poziva HTTP GET request.

```
ping$ = (ipAddress: string) => <Observable<CustomResponse>>
this.http.get<CustomResponse>(`${this.apiUrl}/server/ping/${ipAddress}`)
.pipe(
  tap(console.log),
  catchError(this.handleError)
);
```

Slika 13. Metoda ping$

Funkija pingServer handlea taj HTTP GET request. Pinga server koristeći se servisom, te konstruira odgovor na temelju rezultata pinganja. U odgovoru također vraća pingani server s naredbom Map.of(„server“, server).

```
@GetMapping("/ping/{ipAddress}")
public ResponseEntity<Response> pingServer(@PathVariable("ipAddress") String ipAddress) throws IOException {
    Server server = serverService.ping(ipAddress);
    return ResponseEntity.ok(
            Response.builder() ResponseBuilder<capture of ?, capture of ?>
                    .timestamp(now()) capture of ?
                    .data(Map.of( k1: "server", server))
                    .message(server.getStatus() == Status.SERVER_UP ? "Ping success" : "Ping failed")
                    .status(OK)
                    .statusCode(OK.value())
                    .build()

    );
}
```

Slika 14. Metoda pingServer()

Pomoću serverService.ping(ipAddress) funkcije dohvaća server koji se trenutno pinga, te naravno, pinga server.

```
1 usage
@Override
public Server ping(String ipAddress) throws IOException {
    log.info("Pinging server IP: {}", ipAddress);
    Server server = serverRepo.findByIpAddress(ipAddress);
    InetAddress address = InetAddress.getByName(ipAddress);
    server.setStatus(address.isReachable( timeout: 10000) ? SERVER_UP : SERVER_DOWN);
    serverRepo.save(server);
    return server;
}
```

Slika 15. Metoda ping()

U ovoj se metodi konačno dolazi do zapravog pinganja servera. Prvo se dohvaća server koji se pinga pomoću njegove IP adrese s naredbom serverRepo.findByIpAdress(ipAddress).

Nakon toga se ažurira status servera ovisno o rezultatu pinganja („SERVER_UP“ ili „SERVER_DOWN“). U konačnici se ta promjena sprema u bazu te se vraća pingani server. Tu funkcionalnost pinganja servera završava.

4.5. Brisanje servera

Brisanje servera započinje funkcijom deleteServer(server) koja uzima server kao parametar.

```
deleteServer(server: Server): void {
  this.appState$ = this.serverService.delete$(server.id)
  .pipe(
    map(response => {
      this.dataSubject.next(
        {...response, data: { servers: this.dataSubject.value.data.servers.filter(s => s.id !== server.id)}}
      );
      return { dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}
    }),
    startWith({dataState: DataState.LOADED_STATE, appData: this.dataSubject.value}),
    catchError((error: string) => {
      return of({ dataState: DataState.ERROR_STATE, error })
    })
  );
}
```

Slika 16. Metoda deleteServer();

U suštini ona služi za ažuriranje user interfacea kada se obriše server. Filtriraju se serveri na temelju id-a tako da se prikažu samo oni serveri koji nemaju isti id kao server koji se briše. Također prosljeđuje id servera funkciji delete$.

```
delete$ = (serverId: number) => <Observable<CustomResponse>>
this.http.delete<CustomResponse>(`${this.apiUrl}/server/delete/${serverId}`)
.pipe(
  tap(console.log),
  catchError(this.handleError)
);
```

Slika 17. Metoda delete$

Ta funkcija šalje HTTP DELETE request kako bi se obrisao server određenog id-a koji se prosljeđuje.

```
@DeleteMapping("/delete/{id}")
public ResponseEntity<Response> deleteServer(@PathVariable("id") Long id) {

    return ResponseEntity.ok(
            Response.builder() ResponseBuilder<capture of ?, capture of ?>
                    .timestamp(now()) capture of ?
                    .data(Map.of( k1: "deleted", serverService.delete(id)))
                    .message("Server deleted")
                    .status(OK)
                    .statusCode(OK.value())
                    .build()
    );
}
```

Slika 18. Metoda deleteServer

Metoda deleteServer handlea HTTP DELETE request za brisanje servera određenog id-a. Ona poziva funkciju servisa koja briše server na temelju njegovog id-a te konstruira response koji naznačuje uspjeh operacije.

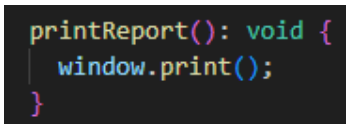
```
1 usage
@Override
public Boolean delete(Long id) {
    log.info("Deleting server by id: {}", id);
    serverRepo.deleteById(id);
    return TRUE;
}
```

Slika 19. Metoda delete

Metoda delete, dakle, poziva funkciju serverRepo.deleteById(id) koja briše određeni server iz baze. Pošto je ta metoda tipa boolean vraća return type tipa true. Tu funkcionalnost brisanja servera završava.

4.6. Printanje izvještaja

Zadnja funkcionalnost ove aplikacije je poprilično jednostavna. Ona omogućuje korisniku spremanje cijelog popisa svih servera iz baze podataka u obliku pdf dokumenta. To čini jednostavnog funkijom printReport() tipa void.



Slika 20. Metoda printReport()

Ova metoda izvršava samo jednu liniju koda a to je window.print() koja omogućuje korisniku spremanje trenutnog dokumenta u obliku pdfa ili nekih drugih oblika dokumenta.

5. Popis slika