

Treće predavanje

Uvod

Ovo predavanje ima 2 cilja:

1. Početi spremati podatke u bazu (dosta je bilo hardkodiranih vrijednosti!)
2. Napraviti REST servise koji će omogućiti rukovanje domenskim objektima - dohvat, stvaranje, ažuriranje i brisanje (naravno, na bazi podataka)

build.gradle

Za početak, dodajmo u svoj `build.gradle` označene biblioteke:

```
dependencies {  
    compile("org.springframework.boot:spring-boot-starter-web:1.1.7.RELEASE")  
    compile("org.springframework.boot:spring-boot-starter-aop:1.1.7.RELEASE")  
    compile("org.springframework.boot:spring-boot-starter-data-jpa:1.1.7.RELEASE")  
    compile("org.hsqldb:hsqldb:2.3.2")  
    compile("com.fasterxml.jackson.datatype:jackson-datatype-hibernate4:2.4.3")  
}
```

Prva, `spring-boot-starter-data-jpa`, u naš će projekt dodati podršku za baze podataka i mogućnost stvaranja repozitorija podataka baziranih na JPA standardu. Pročitajte ukratko o Spring Data JPA modulu na sljedećoj adresi:

<http://projects.spring.io/spring-data-jpa/>

Idući *dependency*, `hsqldb`, naš projekt obogaćuje sa HyperSQL DataBase in-memory bazom podataka. Nju ćemo u nekom kasnijem trenutku zamijeniti sa "pravom" bazom podataka, PostgreSQL, no za sada će nam biti puno brže i jednostavnije razvijati i testirati s ovom.

Konačno, s paketom `jackson-datatype-hibernate4` dodajemo u naš projekt podršku za serijalizaciju Hibernate 4.x objekata. Hibernate je ORM (Object Relational Mapping) koji implementira JPA standard, fino se sljubljuje sa Springom, pa ćemo koristiti baš njega.

<http://hibernate.org/orm/>

<http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/orm.html>

Čisto kao informaciju, Hibernate nije jedina implementacija JPA standarda podržana od Springa (iako je daleko najpopularnija), tu su još *OpenJPA* i *EclipseLink*.

Krenimo s perzistencijom!

Otvorimo paket s našim domenskim modelom i krenimo s prvom klasom, `Game`:

```
@Entity  
public class Game {
```

```

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;

@ManyToOne
@JoinColumn(name = "match_id", nullable = false)
private Match match;

@Enumerated(value = EnumType.STRING)
private Map map;

private Date date;

@ManyToOne
@JoinColumn(name = "winner_id")
private Player winner;

...

```

I to je to.

Dakle, sve što je bilo potrebno napraviti za omogućiti perzistenciju ove klase je dodati nekoliko anotacija. Možete biti sretni što živite u 2014. godini, kada sam ja ovo prvi puta pokušavao prije 10ak godina, bilo je nekoliko dana posla za replicirati ovu funkcionalnost :)

Pa idemo ukratko objasniti koje smo to magične anotacije koristili:

- **@Entity**: An entity is a lightweight persistence domain object. Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes. The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.
- **@Id**: Specifies the primary key of an entity. The field or property to which the Id annotation is applied should be one of the following types: any Java primitive type; any primitive wrapper type; String; java.util.Date; java.sql.Date; java.math.BigDecimal; java.math.BigInteger. The mapped column for the primary key of the entity is assumed to be the primary key of the primary table. If no Column annotation is specified, the primary key column name is assumed to be the name of the primary key property or field.
- **@GeneratedValue(strategy = GenerationType.AUTO)**: A generated id (also known as a surrogate id) is one that is generated by the system. A sequence number in JPA is a sequential id generated by the JPA implementation and automatically assigned to new objects. The benefits of using sequence numbers are that they are guaranteed to be unique, allow all other data of the object to change, are efficient values for querying and indexes, and can be efficiently assigned.

- **@ManyToOne**: Defines a single-valued association to another entity class that has many-to-one multiplicity. It is not normally necessary to specify the target entity explicitly since it can usually be inferred from the type of the object being referenced. If the relationship is bidirectional, the non-owning OneToMany entity side must use the `mappedBy` element to specify the relationship field or property of the entity that is the owner of the relationship.
- **@JoinColumn**: Specifies a column for joining an entity association or element collection.
- **@Enumerated**: Specifies that a persistent property or field should be persisted as an enumerated type.

S obzirom da osnovno poznavanja relacijskih baza podataka uzimamo kao pretpostavku na ovoj vještini, mislim da su anotacije uglavnom jasne već iz naziva. Ukratko, domensku klasu anotirali smo s `@Entity` kako bi označili da ju želimo perzistirati. Hibernate će, s obzirom da nismo drugačije naveli, koristiti istoimenu tablicu u bazi podataka (koju će nam čak i automatski stvoriti!). Nakon toga specificirali smo kolonu koja će u bazi predstavljati primarni ključ, označili je kao autogeneriranu (što znači da će se dodijeljivati nova vrijednost za novostvorene objekte na razini baze podataka). Za kraj, označili smo veze prema drugim entitetima.

Proceduru ponavljamo za ostale domenske klase.

Repozitoriji

Novi paket u našoj projektnoj strukturi zove se `repositories`, a sadržavati će repozitorije za pojedine domenske klase. Kao primjer za analizu, uzeti ćemo `GameRepository`:

```
public interface GameRepository extends CrudRepository<Game, Long> {
    List<Game> findByMatchId(Long matchId);
}
```

Za početak, primjećujemo da naš repozitorij nasljeđuje sučelje `CrudRepository` (koje, pak, nasljeđuje baznu Springovu `Repository` apstrakciju repozitorija). `CrudRepository` nam olakšava život u smislu smanjivanja *boilerplate* programskog koda implementacijom najčešće korištenih metoda repozitorija:

- spremanje entiteta (novog ili ažuriranog postojećeg); `save(S entity)`
- dohvat entiteta po primarnog ključu; `findOne(ID primaryKey)`
- dohvat svih entiteta određenog tipa; `findAll()`
- dohvat broja entiteta određenog tipa; `count()`
- brisanje entiteta; `delete(T entity)`
- provjera postoji li entitet; `exists(ID primaryKey)`

Zgodna stvar, puno manje posla za nas. Morali smo dodati samo još jednu metodu koja će nam trebati, a specifična je za naš slučaj - dohvat liste `Game` objekata za zadani `Match`.

Primjetite magiju - naša metoda `findByMatchId` uopće nije implementirana! Spring Data je iz potpisa naše metode shvatio da želimo dohvatiti listu `Game` objekata za zadanu vrijednost njegovog atributa `Match`.

I to je to, napravili smo sve što nam je trebalo za prvu ruku, što se tiče perzistencije podataka. Idemo sada napraviti REST sučelje na “prednjoj strani”, gdje ćemo zaprimati zahtjeve za manipulacijom podacima u bazi podataka.

Nadogradnja servisa

Idemo za početak nadograditi naše servise metodama za dohvat, stvaranje i ažuriranje. Koristiti ćemo *GameService* kao ogledni primjer:

```
public interface GameService {  
    public Game get(Long id);  
    public List<Game> listByMatch(Long matchId);  
    public Game create(Long matchId, Game game);  
    public Game update(Long id, Game updatedGame);  
    public void delete(Long id);  
}
```

Naravno, to moramo i implementirati u *GameServiceImpl*:

```
@Transactional  
@Service  
public class GameServiceImpl implements GameService {  
  
    @Autowired  
    GameRepository gameRepository;  
  
    @Autowired  
    MatchRepository matchRepository;  
  
    @Override  
    public Game get(Long id) {  
        return gameRepository.findOne(id);  
    }  
  
    @Override  
    public List<Game> listByMatch(Long matchId) {  
        return gameRepository.findByMatchId(matchId);  
    }  
  
    @Override  
    public Game create(Long matchId, Game game) {  
        game.setMatch(matchRepository.findOne(matchId));  
        return gameRepository.save(game);  
    }  
  
    @Override  
    public Game update(Long id, Game updatedGame) {  
        Game game = gameRepository.findOne(id);
```

```

        BeanUtils.copyProperties(updatedGame, game, "id", "match");
        return game;
    }

    @Override
    public void delete(Long id) {
        gameRepository.delete(id);
    }
}

```

Koristeći, vama već dobro poznatu, metodu injekcije ovisnosti (zanimljiva sintagma, kad prevedeteš na hrvatski) “gurnuti” ćemo si u servis repozitorije koje smo malo prije napravili. Nakon toga, “kačimo” naše servisne metode na odgovarajuće metode repozitorija.

Nadogradnja controllera

Zadnji korak je nadogradnja naših kontrolera, koji će znati primati različite tipove HTTP zahtjeva (GET, POST, PUT, DELETE) i protumačiti to kao odgovarajuću akciju. U ovom primjeru ćemo pokriti samo neke od tih akcija, ali shvatiti ćete princip:

```

@Controller
public class GameController {

    @Autowired
    GameService gameService;

    @RequestMapping(value =
"/tournaments/{tournamentId}/rounds/{roundId}/matches/{matchId}/games", method =
RequestMethod.GET)
    public ResponseEntity listGames(@PathVariable Long matchId) {
        return new ResponseEntity(gameService.listByMatch(matchId), HttpStatus.OK);
    }

    @RequestMapping(value =
"/tournaments/{tournamentId}/rounds/{roundId}/matches/{matchId}/games", method =
RequestMethod.POST)
    public ResponseEntity postGame(@PathVariable Long matchId, @RequestBody Game
game) {
        return new ResponseEntity(gameService.create(matchId, game), HttpStatus.CREATED);
    }

    @RequestMapping(value =
"/tournaments/{tournamentId}/rounds/{roundId}/matches/{matchId}/games/{id}", method
= RequestMethod.GET)
    public ResponseEntity getGame(@PathVariable Long id) {
        return new ResponseEntity(gameService.get(id), HttpStatus.OK);
    }
}

```

Jednostavno “kačimo” pojedini tip HTTP zahtjeva na odgovarajuću servisnu metodu. Magija se u ovom koraku nalazi eventualno u dijelu automatske (de)serijalizacija podataka.

Što to znači?

Uzmimo za primjer *postGame* metodu. Ona služi za stvaranje novog *Game* objekta, uz pridjeljivanje identifikatora Match-a kojem pripada. Dakle, HTTP POST zahtjev koji će sadržavati JSON reprezentaciju *Game* objekta (naravno, u obliku običnog niza znakova) mora biti deserijaliziran u pravi pravcati objekt tipa *Game*, koji će onda biti proslijeđen u servis na perzistiranje.

Još valja obratiti pozornost na RequestMapping i korištenje {vrijednosti u vitičastim zagradama}. Kao što možete pretpostaviti, Spring će se pobrinuti da vrijednost koju pročita na tim pozicijama u URL stringu mapira na odgovarajuća polja.

Evo primjera koji će koristiti navedeni mehanizam, kako bi nam vratio listu svih *Game* objekata za turnir 1, rundu 1 i match 1:

<http://localhost:8080/tournaments/1/rounds/1/matches/1/games>

Dodatna konfiguracija

Da bi Jackson znao pravilno serijalizirati objekte koje je mapirao Hibernate, trebamo mu dodatno naznačiti da se koristi Hibernate.

Ako bi pokušali bez toga serijalizirati neke klase u JSON, dobili bi *circular mapping error* jer npr. klasa *Tournament* ima property `List<Round> rounds`, a klasa *Round* ima property `Tournament tournament`, pa bi se Jackson zavrtio u krug pokušavajući serijalizirati atribut *rounds* u klasi *Tournament* i atribut *tournament* u klasi *Round*.

Da bi to izbjegli koristimo činjenicu da Hibernate po defaultu koristi *lazy loading* za OneToMany veze, pa moramo nekako Jackson konverteru naznačiti da ne forsira serijalizaciju atributa koje Hibernate učitava na taj, *lijeni*, način. Za one koji se još nisu susreli s konceptom *lazy loadinga*, radi se o oblikovnom obrascu (jel se tako kaže design pattern?) koji odgađa inicijalizaciju objekata sve do trenutka do kada im se prvi puta ne pristupi.

Kod Spring Boot-a većina stvari se konfigurira preko Spring Bean-ova pa tako i Jackson konverter. Ako Spring Boot prilikom pokretanja ne detektira u našoj aplikaciji potrebni *bean* za npr. Jackson konverter, on će ubaciti neki predefinirani. Ako želimo ubaciti svoju konfiguraciju dovoljno je deklarirati *bean* s željenom konfiguracijom (*bean*-ove deklariramo u klasama koje su anotirane s `@Configuration`), a Spring Boot će to detektirati i uzeti našu konfiguraciju umjesto predefinirane.

```
@Bean
```

```
public MappingJackson2HttpMessageConverter jacksonMessageConverter(){
```

```
MappingJackson2HttpMessageConverter messageConverter = new
MappingJackson2HttpMessageConverter();
    ObjectMapper mapper = new ObjectMapper();
    Hibernate4Module hm = new Hibernate4Module();
    hm.configure(Hibernate4Module.Feature.FORCE_LAZY_LOADING, false);
    mapper.registerModule(hm);
    messageConverter.setObjectMapper(mapper);
    return messageConverter;
}
```

import.sql

U datoteku `import.sql` smjestili smo malo testnih podataka kako na baza ne bi bila prazna. Spring Boot će automatski izvršiti datoteku ovog posebnog imena, ako se nalazi u `classpath`-u:

```
INSERT INTO tournament(id, name) VALUES(1, 'tournament_2014');

INSERT INTO player(id, name, nickname, race) VALUES(1, 'name_1', 'nick_1', 'Zerg');
INSERT INTO player(id, name, nickname, race) VALUES(2, 'name_2', 'nick_2', 'Protoss');
INSERT INTO player(id, name, nickname, race) VALUES(3, 'name_3', 'nick_3', 'Random');
INSERT INTO player(id, name, nickname, race) VALUES(4, 'name_4', 'nick_4', 'Terran');

INSERT INTO round(id, tournament_id) VALUES(1, 1);
INSERT INTO round(id, tournament_id) VALUES(2, 1);

INSERT INTO match(id, round_id, player_1_id, player_2_id) VALUES (1, 1, 1, 2);
INSERT INTO match(id, round_id, player_1_id, player_2_id) VALUES (2, 1, 3, 4);

INSERT INTO game(id, match_id, map, date, winner_id) VALUES(1, 1, 'OVERGROWTH', DATE
'2014-01-01' ,1);
INSERT INTO game(id, match_id, map, date, winner_id) VALUES(2, 1, 'CATALLENA', DATE
'2014-01-02' ,2);
```

Idemo sada to sve malo testirati

<<Tomo ovdje ti malo napisi kako se testira, POSTER plugin, mozes par screenshota opalit>>

GET zahtjeve je lako testirati, jednostavno u browseru navedemo adresu, recimo:

<http://localhost:8080/tournaments/1/rounds/1/matches/1/games>

Ako pritom imate dodatak za svoj web preglednik koji zna prepoznati i lijepo formatirati JSON, stvar bi trebala izgledati otprilike ovako:

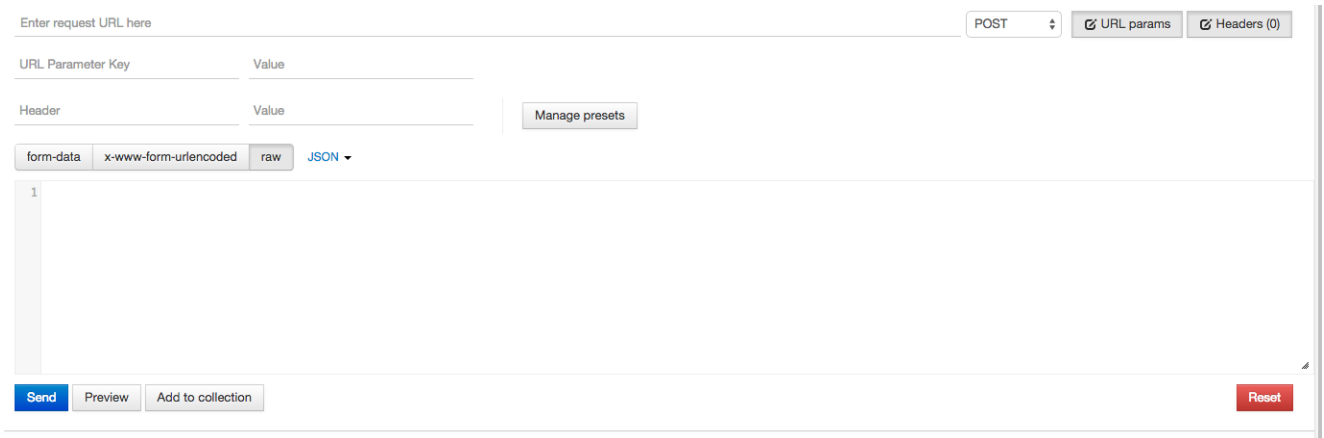
```

[
  - {
    id: 1,
    - match: {
      id: 1,
      - round: {
        id: 1,
        - tournament: {
          id: 1,
          rounds: null,
          name: "tournament_2014"
        },
        matches: null
      },
      games: null,
      - player1: {
        id: 1,
        wonMatches: null,
        name: "name_1",
        nickname: "nick_1",
        race: "Zerg"
      },
      - player2: {
        id: 2,
        wonMatches: null,
        name: "name_2",
        nickname: "nick_2",
        race: "Protoss"
      },
      winner: null
    },
    map: "OVERGROWTH",
    date: 1388530800000,
    - winner: {
      id: 1,
      wonMatches: null,
      name: "name_1",
      nickname: "nick_1",
      race: "Zerg"
    }
  },
  - {
    id: 2,

```

Ostale HTTP metode koje smo koristili u implementaciji (POST, PUT i DELETE) ne možemo testirati preko browsera već će nam trebati nekakav alat poput [Postman](#) za Chrome s kojim možemo konfigurirati i slati potrebne upite prema serveru.

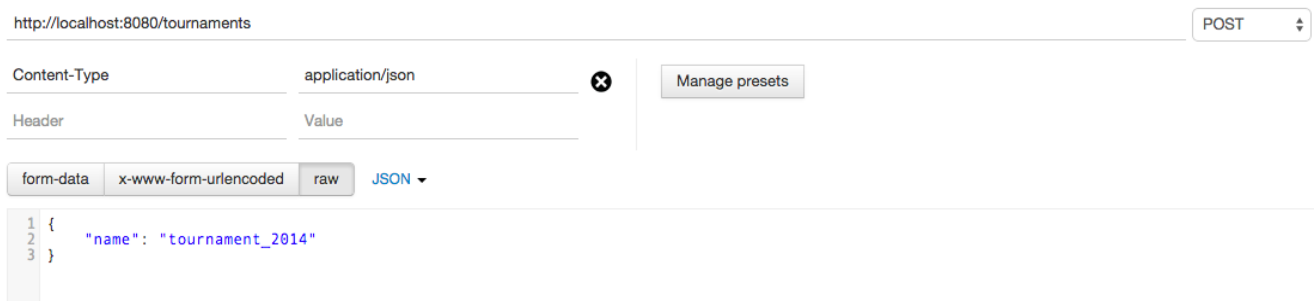
Ovako izgleda forma Postmana za slanje zahtjeva.




Na njoj mozemo definirati HTTP metodu, URL, URL parametre, zaglavlja (*headere*) i tijelo zahtjeva.

Idemo testirati POST metodu za stvaranje turnira.

Postavimo URL da odgovara putanji metode *postTournament* za stvaranje turnira, postavimo HTTP metodu na POST, postavimo zaglavlje na `Content-Type:application/json` i kao tijelo predamo JSON strukturu s imenom turnira.



Server nam kao odgovor vrati status kod 201 Created i JSON stvorenog turnira:



Ako želimo promijeniti ime tog turnira trebamo napraviti PUT zahtjev za taj turnir:

http://localhost:8080/tournaments/2 PUT

Content-Type: application/json × Manage presets

Header: Value

form-data x-www-form-urlencoded raw **JSON**

```

1 {
2   "name": "Novo ime"
3 }

```

Send Preview Add to collection

Body Cookies (1) Headers (4) STATUS 200 OK TIME 92 ms

Pretty Raw Preview **JSON** XML

```

1 {
2   "id": 2,
3   "rounds": null,
4   "name": "Novo ime"
5 }

```

I za izbrisati koristimo metodu DELETE.

http://localhost:8080/tournaments/2 DELETE

Header: Value Manage presets

form-data x-www-form-urlencoded raw **JSON**

```

1

```

Send Preview Add to collection

Body Cookies (1) Headers (3) STATUS 200 OK TIME 98 ms

Pretty Raw Preview **JSON** XML

```

1

```

Ako neka klasa ima kao *property* drugu klasu, dovoljno je u JSON specificirati id (primarni ključ) tog objekta da bi Spring to znao spremiti. No, može se staviti i JSON sa svim propertyima te klase. Npr. ako želimo dodati novi *Match* koji u sebi referencira igrače koji u njemu sudjeluju napravili bi ovakav zahtjev:

http://localhost:8080/tournaments/1/rounds/1/matches POST

Content-Type: application/json × Manage presets

Header: Value

form-data x-www-form-urlencoded raw **JSON**

```

1 {
2   "player1": {
3     "id": "1"
4   },
5   "player2": {
6     "id": "2"
7   }
8 }

```

Send Preview Add to collection

Ovaj upit bi dodao u rundu s id 1 novi match s igracima koji imaju id 1 i 2.