Završni rad

Izrada veb radnog okvira u programskom jeziku Java

Oktobar 2022.

Mentor:

Prof. dr Vladimir Milićević

Kandidat:

Nikola Tasić 3698

Sadržaj

Uvod

U ovom radu bavićemo se kreiranjem veb radnog okvira za *Java* programski jezik pod nazivom *Grain*. Pored kreiranja samog radnog okvira, cilj rada je istraživanje osnovnih i nekih od naprednijih koncepata koji čine svaki radni okvir i samim tim veb aplikacije. Istražićemo razne odluke koje nastaju prilikom dizajniranja različitih naprednih funkcionalnosti, kao što su *sistem za umetanje zavisnosti* (eng. *dependency injection*) ili *templejting jezik* (eng. *templating lanaguage*) za kreiranje dinamičkih stranica generisnanih na serveru (eng. *server-side renderer*). Sam radni okvir biće po svojoj strukturi i funkciji nalik na *Spring* radni okvir, koji mu je i bio inspiracija.

Objasnićemo i teorijski obraditi pojmove kao što su:

- Umetanje zavisnosti
- HTTP zahtevi i sesija
- Autentikacija i autorizacija bazirana na rolama
- Dizajniranje programskog jezika i kreiranje interpretatora
- MVC arhitektura softverskih projekata
- Objektno-relaciono mapiranje i komunikacija sa bazama podataka

Takođe, na kraju rada ćemo obraditi primer gotove aplikacije napisane u *Grain* radnom okviru. Aplikacija će pratiti MVC arhitekturu ali pored toga će koristiti asinhrone HTTP pozive za dobavljanje određenih informacija. Naravno, aplikacija će imati autentikaciju i autorizaciju za odgovarajuće putanje.

Pojam radnog okvira

Šta je radni okvir

Pojam radni okvir predstavlja strukturu ili skup pravila prateći koja lakše dolazimo do cilja. Kada kažemo radni okvir (eng. *framework*) najčešće mislimo na softverski frejmvork koji služi za izradu neke od različitih vrsta aplikacija (desktop, veb, mobilnih itd.). Naravno, radni okviri postoje i u drugim sferama industrije. Jedan primer, blizak softveru, koji je takođe iz softverske industrije, je upravljanje projektima uz korišćenje jednog od agilnih radnih okvira kao što su npr. *Scrum*, *ekstremno programiranje*, *Lean* itd. Pravila i struktura nametnuta od strane tih okvira dovode do toga da se manje vremena i energije (resursa) troši na kreiranje njih samih prilikom početka svakog projekta, postavljaju smislena osnovna pravila koja su se u praksi pokazala kao efikasna prilikom uspešnog vođenja projekata i naravno dovode do lakšeg uključenja novih ljudi na projekat koji poznaju dati radni okvir.

Slična situacija nastaje i u softverskom svetu. Ne želimo da svaki put kada krećemo sa novim projektom kreiramo osnovne alate koje ćemo koristiti od nule (parsiranje HTTP zahteva, *web security* biblioteka, logovanje itd.). Upotreba radnih okvira umnogome smanjuje vreme razvoja nekog sistema. Takođe, zbog toga što mnogo timova radi na različitim projektima koristeći iste alate, dolazi do toga da se sam alat bolje testira i samim tim prilagođava promenama. Pored svega toga, prilikom promene članova tima proces upoznavanja člana tima sa projektom je znatno kraći i sam taj proces se može fokusirati isključivo na upoznavanje novog

člana sa domenskim problemima, jer će sva tehnička infrastruktura biti manje-više ista (podrazumevajući, naravno, da je član upoznat sa radnim okvirom). Postoji dobra misao iz knjige *"Jasan Kod"* (eng. *"Clean Code"*) Roberta C. Martina, gde Vord Kaningam (kreator vebsajta *Wikipedia*) navodi da je jedna od odlika čistog koda da programer pri čitanju istog nailazi na linije i konstrukcije koje su upravo ono što je očekivao. Radni okviri nam tu pomažu jer će uvek postojati zajednička polazna tačka za sve projekte i programeri će, baš tako, nailaziti na konstrukcije koje su upravo ono što su i očekivali.

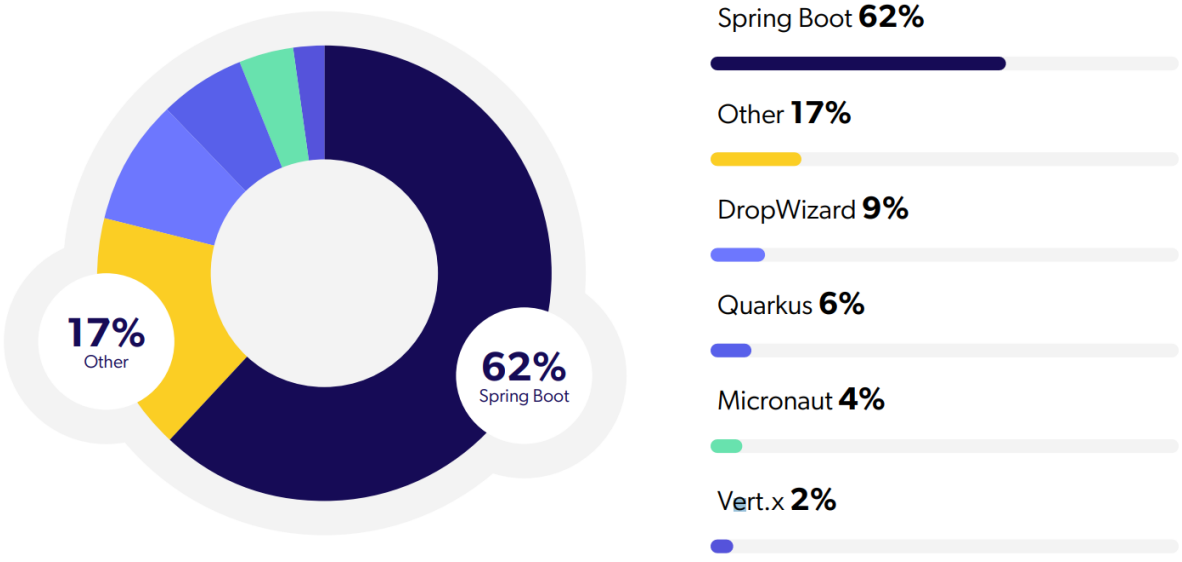
Radni okvir u kontekstu programiranja je temelj povrh kojeg konstruišemo softver uz pomoć skupa biblioteka i pravila. Takođe, radni okvir i pomenute biblioteke sadrže implementaciju često upotrebljavanih funkcionalnosti koje se tiču domena za koji je on specijalizovan. U ovom radu ćemo se fokusirati na domen veb frejmvorka, tako da će se te podrazumevane implementacije učestalih funkcionalnosti ticati parsiranja HTTP zahteva, implementacije veb bezbednosti (eng. *web security*), komunikacije sa bazom podataka, kreiranja dinamičkih stranica itd.

U zavisnosti od toga kako je implementiran, radni okvir može biti modularan i monolitan. Modularan radni okvir je, na primer, *Spring* radni okvir. *Spring* sam po sebi ne donosi mnogo više od implementacije umetanja zavisnosti (eng. *dependency injection* - *DI*) ali kombinovanje sa njegovim ostalim njegovim modulima (bibliotekama) nam daje kompletno rešenje. Mana ovog pristupa su potencijalni problemi sa različitim zavisnostima i malo veća kompleksnost konfiguracije. Velika prednost je to što programer bira koje će module izabrati i tako dolazi do minimalnog rešenja koje će zadovoljiti potrebe projekta. U drugom slučaju, kod monolitnih radnih okvira, kakav je ujedno i **Grain**, celokupno rešenje je u obliku jedne zavisnosti i programer samo bira da li će koristiti neke funkcionalnosti ili ne. Ovde možemo doći do problema da je brdo nekorišćenih funkcionalnosti, a samim tim i koda, ubačeno u projekat. Ovo uglavnom ne predstavlja problem u mnogim situacijama, ali to naravno zavisi od projekta i veličine framework-a.

Dakle, s obzirom na to da ustanovili smo šta predstavlja radni okvir - neizostavan alat modernog developera - možemo da pređemo na opis nekih najpopularnijih rešenja.

Veb radni okviri u *Java* programskom jeziku

U ovom delu pričaćemo o nekim od popularnijih Java biblioteka i radnih okvira i videćemo koliko se procentualno koriste na projektima gde je upravo *Java* glavna tehnologija. Referenciraćemo *Perforce*-ov *2021 Java Developer Productivity Report*, gde možemo naći pregršt različitih informacija o *Java* ekosistemu.



Procenat zastupljenosti radnih okvira - Perforce - 2021 Java Developer Productivity Report[1]

Kao što možemo da vidimo, *Spring Boot* ubedljivo zauzima prvo mesto po popularnosti među radnim okvirima na *Java* projektima. *Spring Boot*, doduše, beleži pad sa 83% koje beleži prošle godine. Na trećem mestu sa tek 9% se nalazi *Micronaut*, koji je kao i četvrtoplasirani *Quarkus* (6%) doživeo pomak sa svojih 1% koje je imao prethodne godine (2020).

Vert.x

Vert.x je *open-source*, reaktivni i poliglotski (može biti pisan u bilo kom JVM jeziku) programski *toolkit* koji nam dolazi od *Eclipse*-a. *Vert.x* je modularan, brz i lagan, a dizajniran je za korišćenje u mikroservisnim aplikacijama. Takođe, pored dizajna koji ima mikroservise na umu, *Vert.x* je pogodan za reaktivno programiranje jer se bazira na *event loop*-u poput tehnologije kao što je *Node.js*. (preuzeto sa https://vertx.io)

Micronaut

Micronaut je, kao što mu i samo ime govori, *micro-framework* koji je dizajniran tako da ne koristi *Java Reflection API* za konfiguraciju, stoga sve njegove funkcionalnosti koje bi inače bile konfigurisane u toku izvršenja se zapravo konfigurišu u toku kompajliranja. Ovo dovodi do znatnog umanjenja zahteva za radnom memorijom, a takođe i smanjuje vreme pokretanja. Ovo je, naravno, idealno za korišćenje u mikroservisnim i *cloud-native* aplikacijama. Takođe, *Micronaut* je *open-source*. (preuzeto sa https://micronaut.io)

Quarkus

Quarkus je *Java* framework prilagođen za *Kubernetes* deployment. Glavne tehnologije koje omogućuju *Quarkus* su *OpenJDK HotSpot* i *GraalVM*. Ideja *Quarkus*-a je da načini *Javu* vodećom platformom u *Kubernetes* i *serverless* okruženjima dok pruža developerima jedinstven imperativni i reaktivni model za optimalno korišćenje u širokom spektru arhitektura. (preuzeto sa https://quarkus.io)

DropWizard

DropWizard je *open-source Java* radni okvir za razvoj visoko performantnih, *ops-friendly*, *RESTful* veb servisa. *DropWizard* sa sobom povlači stabilne i zrele *Java* biblioteke iz *Java* ekosistema u jednostavan, lak paket koji omogućava programerima da se fokusiraju na obavljanje sbog posla. (preuzeto sa https://www.dropwizard.io)

Spring Boot

Spring Framework je *open-source* radni okvir i *IoC* (inverzija kontrole - eng. *inversion of control*) kontejner za *Java* platformu. Jedna od glavnih odlika *Spring*-a je da se on može koristiti u bilo kojoj *Java* aplikaciji ali se najčešće koristi za izradu web aplikacija na *Java EE* (*Java Enterprise Edition*) platformi. *Spring*, zajedno sa *Spring Boot*-om, je *de facto* standard za izradu web aplikacija. (preuzeto sa https://spring.io)

Teorijska postavka

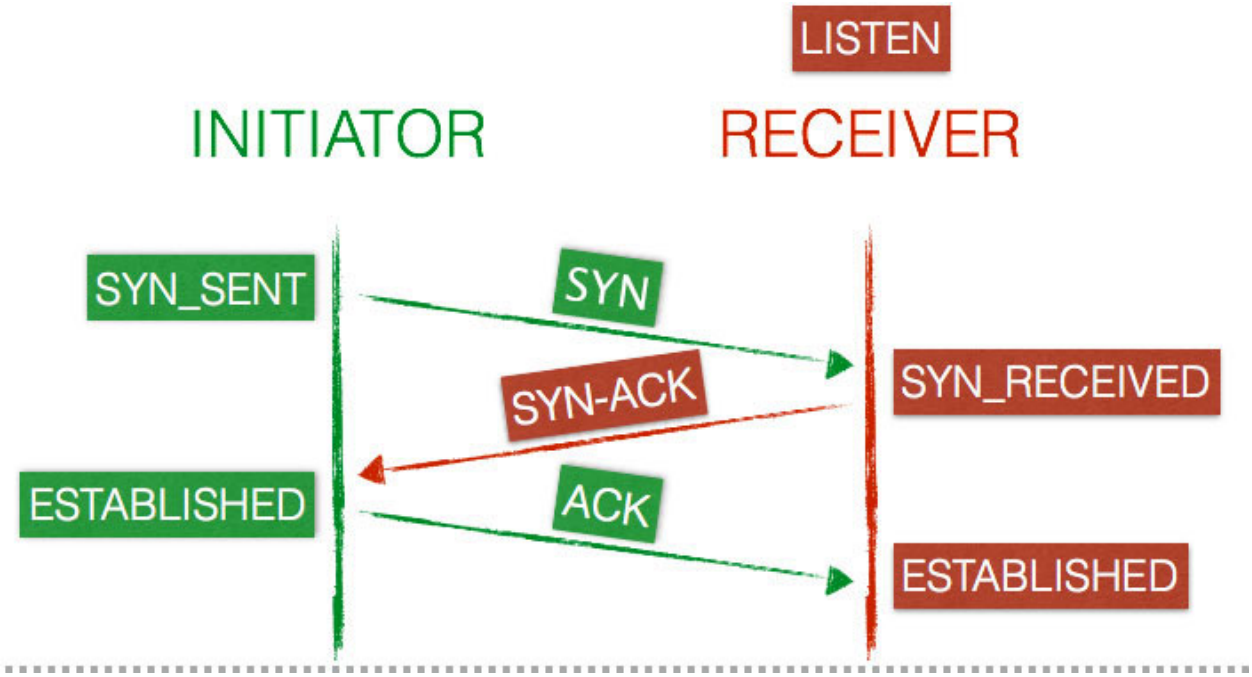
Pre nego što predstavimo studiju slučaja koja će biti pokrivena u ovom radu, bilo bi korisno proći kroz sve veće bitne koncepte koje Grain framework pokriva. Koncepti će se ticati različitih podležućih tehnologija koje primarno koristimo na vebu, ali i nekih specifičnih kao što je, na primer, pisanje programskog jezika.

HTTP

HTTP ili *hyper-text transfer protocol* je najkorišćeniji protokol za razmenu informacija na vebu. HTTP je započeo svoju evoluciju, ako možemo tako da se izrazimo, u kolevci Interneta - u CERN-u 1989. godine. Prva verzija (1.0) se pojavila 1991. godine a trenutna standardna najpodržanija verzija je 1.1, koja je standardizovana 1999. godine (RFC 2616 sa dopunama 2014. godine RFC 7230-7235). Postoje i verzije HTTP/2 i HTTP/3 koje menjaju način prenosa podataka preko HTTP-a ali i ne i njegovu semantiku iz ugla aplikacija koje ga koriste. HTTP/2 uvodi binarnu kompresiju podataka u zaglavlju (više o tome kasnije),

korišćenje jedne TCP konekcije za razmenu podataka i *push*-ovanje podataka. HTTP/3 je revizija HTTP/2 protokola koji koristi QUIC i UDP transportni protokol umesto TCP-a. U daljem tekstu kada govorimo o HTTP-u mislićemo na HTTP/1.1 verziju, jer je to verzija koju podržava *Grain* radni okvir.

HTTP je protokol u aplikativnom sloju i osnova komunikacije na *World Wide Web*-u. Ovaj protokol funkcioniše kao zahtev-odgovor protokol između klijenta i servera. Klijentska aplikacija kreira zahtev ka serveru. Server nakon obrade podataka vraća odgovor klijentu, koji može da sadrži različite informacije u svom zaglavlju (eng. *headers*) i zahtevani resurs u telu (eng. *body*). Komunikacija između servera i klijenta se ostvaruje pomoću TCP transportnog protokola. U osnovi TCP protokola se nalazi *three-way-handshake* koji osigurava da su podaci isporučeni (za razliku od UDP-a).



Dijagram TCP-a - blog.confirm.ch/tcp-connection-states

Kao što vidimo, komunikacija se ostvaruje u tri koraka:

1. U prvom koraku klijent šalje **SYN** zahtev ka serveru.
2. Server odgovara sa **SYN-ACK** i potvrđuje da je dobio zahtev.
3. Na kraju, klijent šalje **ACK** nazad serveru.

Glavne odlike (razlike) u odnosu na **UDP** su sledeće:

- Uređen redosled paketa
- Ponovna transmisija izgubljenih paketa: svi podaci koji nisu dobili **ACK** odgovor se ponovo transmituju
- Transfer bez grešaka - svi paketi sa greškom se tretiraju izgubljenim i ponovo se transmituju
- Kontrola toka - kontrola brzine transfera da bi osiguralo dostavljanje podatka
- Kontrola zagušenja - izgubljeni paketi smanjuju brzinu toka

HTTP zahtev i odgovor

Kao što smo već naglasili, HTTP je protokol koji se zasniva na zahtevu i odgovoru, tako da ćemo ta dva koncepta bolje objasniti u ovom poglavlju.

Tipovi poruka HTTP zahtevi i HTTP odgovori koriste generički format za poruke definisan u RFC 822. Oba tipa poruka imaju početnu liniju (eng. *start-line*), nijedan ili više polja zaglavlja, praznu liniju (linija koja nema nijedan karakter pre CRLF) koja označava kraj poljima zaglavlja i telo poruke (eng. *message body*) koje je opciono.

Šema poruka bi izgledala ovako:

```
generic-message = start-line
          *(message-header CRLF)
          CRLF
          [ message-body ]
start-line    = Request-Line | Status-Line
```

Moramo napomenuti da je **CRLF** oznaka za kraj reda standardna na Windows sistemima (*NIX* sistemi koriste samo **LF**, izuzev starijih verzija OS X-a koji koriste **CR**). **CR** predstavlja *carriage return* - znak za povratak na početak reda, dok **LF** predstavlja *line feed*, odnosno novi red. Ovi karakteri postoje u svakom tekstu, naravno u zavisnosti od operativnog sistema, s tim što su nevidljivi. Karakter **CR** je `0D` u heksadecimalnom zapisu i tekstualna reprezantacija mu je `\r`. Karakter **LF** je `0A` u hexadecimalnom, a tekstualno se prikazuje kao `\n`. Ova nomenklatura je konvencija je zaostavština pisaćih mašina.

HTTP zaglavlje Polja zaglavlja se dele na tri tipa:

- Generička - mogu ih koristiti i zahtev i odgovor (Cache-Control, Connection itd.)
- Zahtev - koriste se isključivo u zahtevu (Host, User-Agent itd.)
- Odgovor - koriste se isključivo u odgovoru (Age, Location itd.)
- Entitetska - koriste se u zahtevu i odgovoru za opis tela zahteva (Content-Type, Content-Length itd.)

Svako polje se sastoji od imena polja koje je praćeno dvotačkom (:) i vrednosti polja. Ime polja je *case-insensitive*. Između dvotačke i vrednosti polja može postojati bilo koji broj belih polja (eng. *whitespace*) Vrednost polja može biti bilo koji tekst, ali ne sme da sadrži **CRLF**. Polja zaglavlja se razdvajaju **CRLF** karakterom. Redosled gore navedenih polja nije bitan, ali je preporučljivo da se prvo navode generička polja, pa onda polja zahteva i polja odgovora, a na kraju entitetska polja.

```
message-header = field-name ":" [ field-value ]
field-name   = token
field-value  = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
        and consisting of either *TEXT or combinations
        of token, separators, and quoted-string>
```

Telo poruke Telo poruke je zaduženo za prenos entiteta (ovde ne mislimo na entitete u kontekstu baza podataka). Postojanje tela poruke je indikovano postojanjem **Content-Length** ili **Transfer-Encoding** zaglavlja. Ukoliko ne postoji nijedan od ta dva zaglavlja, telo poruke bi trebalo biti ignorisano. Postoje tipovi poruka koji po svojoj semantici i konvenciji ne bi trebalo da imaju telo a to su: svi informacioni odgovori (status kod 1xx - više o ovim kodovima kasnije), 204 (nema sadržaja - eng. *no content*) i 304 (nije izmenjeno - eng. *not modified*). Svi ostali odgovori imaju telo, makar ono bilo dužine nula.

S obzirom na to da telo poruke dolazi na kraju same HTTP poruke, postojane nekog od **Transfer-Encoding** i **Content-Lenght** *header*-a je neophodno - u suprotnom korisnik ne bi znao kada da prestane sa čitanjem poruke.

Zahtev Zahtev klijenta ka serveru se sastoji od **linije zahteva** (eng. *Request-Line*), zaglavlja i tela poruke.

```
Request = Request-Line
      *(( general-header
       | request-header
       | entity-header ) CRLF)
      CRLF
      [ message-body ]
```

Linija zahteva se sastoji iz tri dela: HTTP metode, zahtevanog resursa (URI - *uniform resource identifier*) i verzije HTTP protokola (HTTP/1.1, HTTP/2, HTTP/3) praćene **CRLF**-om. Sva tri dela linije zahteva su razdvojeni jednim razmakom (*space*).

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Metoda zahteva opisuje koja HTTP metoda će biti primenjena na resursu identifikovanim od strane *Request-URI*-a. Metode su *case-sensitive*.

```
Method = "OPTIONS"
       | "GET"
       | "HEAD"
       | "POST"
       | "PUT"
       | "DELETE"
       | "TRACE"
       | "CONNECT"
       | extension-method
extension-method = token
```

Sve ove metode sa sobom nose određenu semantiku, ali naravno server može odlučiti da ih po svojoj volji implementira. Dobro je, naravno, pratiti konvenciju i poštovati njihovu ulogu. Nećemo zalaziti detaljno u ulogu svake od ovih metoda, ali poenta jeste da npr. treba koristiti `GET` za dobavljanje podataka, `POST` za kreiranje, `PUT` i `PATCH` za celovito ili parcijalno ažuriranje, `DELETE` za brisanje i tako dalje. Naravno, u specifikaciji stoji da ova metoda može biti proširena bilo kojom novom metodom - na primer možemo implementirati metodu `HELLO` koja šalje pozdrav ali to nije u standardu i nijedan pretraživač neće znati šta da uradi sa njom. Po specifikaciji, naravno, stoji da server mora da implementira `GET` i `HEAD` metode dok su sve druge opcione.

Resurs identifikator predstavlja apsolutnu putanju resursa na serveru. Ovo može biti direktno mapirano u fajl na fajl sistemu, ili proizvoljno mapirano na bilo koji drugi resurs u sistemu koji implementira HTTP. Kao što smo rekli, *Request-URI* služi da identifikuje resurs nad kojim će biti primenjena HTTP metoda.

```
Request-URI  = "*" | absoluteURI | abs_path | authority
```

Primer *Request-Line*-a koji dobavlja početnu stranu nekog sajta bi izgledao ovako:

```
GET http://7aske.com HTTP/1.1
```

Ovo je primer *Request-Line*-a sa apsolutnim *URI*, dok sledeći navodi identičan zahtev ali koristeći apsolutnu putanju:

```
GET / HTTP/1.1
Host: http://7aske.com
```

Odgovor Posle obrade zahteva server vraća HTTP odgovor nazad. HTTP odgovor izgleda identično kao i zahtev s tim da umesto *Request-Line*-a ima **Status-Line** (statusna linija).

```
Response = Status-Line
      *(( general-header
       | response-header
```

```
     | entity-header ) CRLF)
    CRLF
    [ message-body ]
```

Statusna linija se sastoji iz tri dela: HTTP verzije (HTTP/1.1, HTTP/2, HTTP/3), status koda i tekstom koji opisuje razlog status koda praćenim CRLF-om. Sva tri dela linije zahteva su razdvojeni jednim razmakom (space).

Status kod je trocifreni broj koji po specifikaciji ima svoje značenje. Neki od primera kodova su: 101 Switching Protocols, 200 OK, 304 Not Modified, 404 Not Found. Kodovi se po svom tipu dele na grupacije:

- 1xx - informacioni
- 2xx - uspešni
- 3xx - preusmeravački
- 4xx - greške klijenta
- 5xx - greške servera

Svaki status ima svoj kod i razlog (eng. *reason*). Razlog je predviđen isključivo za čitanje i razumevanje od strane čoveka, dok sam kod čita mašina. Implementacija klijenta nije u obavezi da parsira tekst razloga na bilo koji način.

```
Status-Code  =
  "100"  ; Continue
  | "101"  ; Switching Protocols
  | "200"  ; OK
  | "201"  ; Created
  | "202"  ; Accepted
  | "203"  ; Non-Authoritative Information
  | "204"  ; No Content
  | "205"  ; Reset Content
  | "206"  ; Partial Content
  | "300"  ; Multiple Choices
  | "301"  ; Moved Permanently
  | "302"  ; Found
  | "303"  ; See Other
  | "304"  ; Not Modified
  | "305"  ; Use Proxy
  | "307"  ; Temporary Redirect
  | "400"  ; Bad Request
  | "401"  ; Unauthorized
  | "402"  ; Payment Required
  | "403"  ; Forbidden
  | "404"  ; Not Found
  | "405"  ; Method Not Allowed
  | "406"  ; Not Acceptable
  | "407"  ; Proxy Authentication Required
  | "408"  ; Request Time-out
  | "409"  ; Conflict
  | "410"  ; Gone
  | "411"  ; Length Required
  | "412"  ; Precondition Failed
  | "413"  ; Request Entity Too Large
  | "414"  ; Request-URI Too Large
  | "415"  ; Unsupported Media Type
```

```
  | "416"  ; Requested range not satisfiable
  | "417"  ; Expectation Failed
  | "500"  ; Internal Server Error
  | "501"  ; Not Implemented
  | "502"  ; Bad Gateway
  | "503"  ; Service Unavailable
  | "504"  ; Gateway Time-out
  | "505"  ; HTTP Version not supported
  | extension-code
```

Kao što je i slučaj sa metodama, serverska implementacija može deklarisati i parsirati dodatne kodove.

Sada kada imamo informacije, možemo da damo primer kako bi izgledao jedan kompletan HTTP zahtev-odgvor ka serveru kreiranom u *Grain* frejmvorku:

```
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: insomnia/2022.6.0
> Accept: */*

< HTTP/1.1 200 OK
< Content-Length: 4679
< Content-Type: text/html
```

Kolačići (Cookies) Kolačići su mali tekstualni fajlovi koji se čuvaju na klijentu (pretraživaču) i koji služe da čuvaju informacije o klijentu. Kolačići se šalju svaki put kada se šalje zahtev ka serveru. Oni se koriste za različite stvari, od praćenja aktivnosti klijenta na sajtu, do čuvanja informacija o korisniku. Jedna od osnovnih uloga kolačića je da identifikuju zahtev na serveru tako da bi server mogao da zna o kom korisniku se radi. Svaki korisnik može da dobije unikatni identifikator preko kolačića i preko njega će server znati o kome se radi. U prevodu, preko kolačića server prati sesiju klijenta/korisnika.

Kolačići se šalju kroz **Cookie** header:

```
...
Cookie: <ime-kolačića>=<vrednost-kolačića>
...
```

Na primer, *cookie* koje identifikuje korisnika može da izgleda ovako:

```
Cookie: JSSID=1234567890
```

Kolačići se mogu postaviti na klijentu kroz **Set-Cookie** header. Na primer *cookie* koje identifikuje korisnika može da izgleda ovako:

```
Set-Cookie: JSSID=1234567890
```

Svi kolačići se mogu poništiti kroz **Set-Cookie** header sa vrednošću **max-age=0**. Na primer:

```
Set-Cookie: JSSID=1234567890; max-age=0
```

Klijent automatski ignoriše kolačiće koji su istekli. Svi kolačići se automatski šalju kroz svaki `GET`/`POST` zahtev koji je iniciran iz pretraživača kroz link ili formu. Zahtevi koji se iniciraju koristeći JS *fetch API*, *XHR* ili koristeći neku HTTP biblioteku, neće sadržati kolačiće i oni moraju biti dodati ručno kroz **Cookie** heder. U tim slučajevima se verovatno radi o nekoj *stateless* komunikaciji (bez stanja - suprotno od onoga za šta su kolačići namenjeni) i u tom slučaju je bolje koristiti neki vid *stateless* autentikacije. Više o tome u narednim poglavljima.

Kolačići i njihova zloupotreba je jedan od velikih rizika *web*-a i stoga se mora obazrivo rukovati istim.

Parsiranje JSON podataka

Sada kada smo obradili kako funkcioniše HTTP protokol, možemo da se osvrnemo na jedan od važnijih tipova podataka koji se koristi u *web*-u. U *Grain* frejmvorku za parsiranje JSON podataka se ne koristi biblioteka već postoji minimalistična implementacija koja se trudi da pokrije većinu JSON specifikacije.

JSON je tip podataka koji se koristi za razmenu podataka između klijenta i servera i predstavlja tekstualni format podataka koji je čitak za čoveka i lako parsiran od strane mašine. Skraćenica JSON stoji za *javascript object notation*. JSON je sličan JavaScript objektima ali je nešto jednostavniji i ne podržava sve funkcionalnosti koje JavaScript objekti imaju. Sastavljen od dva tipa podataka: objekata i nizova. Objekat je kolekcija imenovanih vrednosti, dok niz predstavlja kolekciju neimenovanih vrednosti. Vrednosti mogu biti tipa string, broj, boolean, null, objekat ili niz.

Jedna od odlika JSON oblika podataka, pored čitljivosti, jeste da je *language-independent*, tj. da njegov oblik ne zavisi od jezika u kome se koristi, i zbog toga je idealan za razmenu podataka između aplikacija koje su kreirane u različitim jezicima. Pored toga mnogi jezici u svojoj standardnoj biblioteci imaju implementaciju JSON parsiranja, a ako nemaju onda sigurno postoji popularna biblioteka za to.

Primer JSON tipa podatka:

```
{
  "port": 8080,
  "routes": [
    {
      "path": "/",
      "method": "GET",
      "handler": "index"
    }
  ],
  "handlers": {
    "index": {
      "type": "file",
      "path": "index.html"
    }
  }
}
```

Iznad vidimo primer JSON objekta koji možemo da očekujemo u nekom od HTTP odgovora. Da bi klijent pročitao HTTP odgovor JSON tipa, i na adekvatan načina parsirao podatke, moramo podesiti **Content-Type** zaglavlje na tip `application/json`. Bez tog zaglavlja klijent će verovatno parsirati odgovor kao da je u pitanju podrazumevana vrednost za to zaglavlje, a to je `text/plain`. Primer jednog JSON odgovora bi bio:

```
HTTP/2 200 OK
Server: GitHub.com
Date: Tue, 11 Oct 2022 18:53:40 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 69

{"login":"7aske","id":17355360,"html_url":"https://github.com/7aske"}
```

Ovde vidimo par osnovnih *header*-a i telo odgovora. Ključno je naglasiti da je, pored Content-Type *header*-a, bitno podesiti i **Content-Length** *header* da bi klijentska aplikacija znala koliko karaktera iz samog tela odgovora pročitati.

JSON je odlična alternativa XML-u za pisanje konfiguracionih fajlova jer je lakše pisati i razumeti JSON fajl u odnosu na XML fajl. Takođe, postoji i YAML kao nadskup JSON-a koji ima drugačiju sintaksu i više funkcionalnosti, jedna od kojih je postojanje komentara.

Autentikacija i autorizacija

Autentikacija i autorizacija su osnovni koncepti kada su u pitanju aplikacije sa više nivoa pristupa.

Autentikacija

Autentikacija predstavlja potvrđivanje identiteta korisnika na sistemu. U procesu autentikacije anonimni korisnik postaje autentikovani korisnik kome su pridružene dodatne informacije i kome može biti praćena sesija.

Jedan od najosnovnijih metoda za autentikovanje je HTTP Basic autentikacija. U ovom procesu klijent šalje HTTP zahtev sa zaglavljem **Authorization** koji sadrži podatke o korisničkom imenu i lozinki u formatu `username:password` kodirane u Base64. Primer HTTP zahteva sa Basic autentikacijom:

```
GET / HTTP/1.1
Host: 7aske.com
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Ovaj tip autentikacije je jednostavan za implementaciju, ali je i jednostavan za zloupotrebu. Zbog toga se ne preporučuje za korišćenje u produkciji.

Jedan od alternativnih pristupa je username/password autentikacija. U ovom pristupu klijent šalje HTTP zahtev sa korisničkim imenom i šifrom korisnika, i natrag dobija neku vrstu *token*-a koji će na dalje služiti za identifikaciju tog korisnika na sistemu. Token može biti kolačić, JWT itd.

Autorizacija

Autorizacija predstavlja proces određivanja nivoa pristupa korisnika na sistemu i obično se vrši na osnovu uloga korisnika. Uloga je skup prava koja su povezana sa korisnikom i može biti povezana sa korisničkim imenom ili sa tokenom koji je dobijen u procesu autentikacije.

Na primer, kada korisnik pristupa nekoj stranici na sajtu on može da bude ulogovan ili ne. Ako je ulogovan, korisnik onda može da vidi neke informacije koje nisu vidljive anonimnom korisniku. Ako je korisnik ulogovan i ima određenu ulogu, onda može da vidi još neke informacije koje nisu vidljive korisnicima sa drugim ulogama.

Konkretan primer autorizacije bi bila neka veb strana - na primer internet prodavnica. Anonimni korisnik može da gleda proizvode i isključivo to. Ulogovani korisnik pored svakog proizvoda može imati opciju da naruči taj proizvod ili isti doda u korpu. Pored svega toga, ulogovani korisnik, koji je ujedno i menadžer ili administrator te prodavnice, će pored svakog proizvoda imati dugme *izmeni* i/ili link ka administracionoj strani.

Uloge korisnika i pristup - www.toptal.com/firebase/role-based-firebase-authentication

Suma sumarum, autentikacija je proces potvrđivanja identiteta korisnika na sistemu, dok je autorizacija proces određivanja nivoa pristupa korisnika na sistemu.

Autorizacija se može vršiti, npr. kod HTTP zahteva, slanjem *JWT*-a (eng. *JSON web token*) u headeru **Authorization**. Primer HTTP zahteva sa tokenom u headeru:

```
POST /products/1/order HTTP/1.1
Host: 7aske.com/shop
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaXNzIjoiZ3JhaW4iLCJuYW1l
```

U ovom slučaju HTTP zahtev se prosleđuje na server koji proverava da li je token validan i da li korisnik ima pravo da pristupi resursu koji je zatražio. Ukoliko je token validan i korisnik ima pravo pristupa resursu, server odgovara sa HTTP statusom 200 i podacima o resursu. JWT token sadrži informacije o tome koji je korisnik u pitanju, ko je kreirao token, koliko je vremenski validan token itd. JWT se validira na serveru i šanse za njegovu manipulaciju su svedene na minimum, za razliku od kolačića.

JWT je JSON objekat koji se šalje u headeru HTTP zahteva. JWT se sastoji od tri dela odvojenih tačkom:

1. *Header* - koji sadrži informacije o algoritmu koji je korišćen za enkripciju i tipu tokena
2. *Payload* - koji sadrži informacije o korisniku
3. *Signature* - koji je rezultat enkripcije *header*-a i *payload*-a

JWT objekat je kodiran u *Base64*.

Umetanje zavisnosti

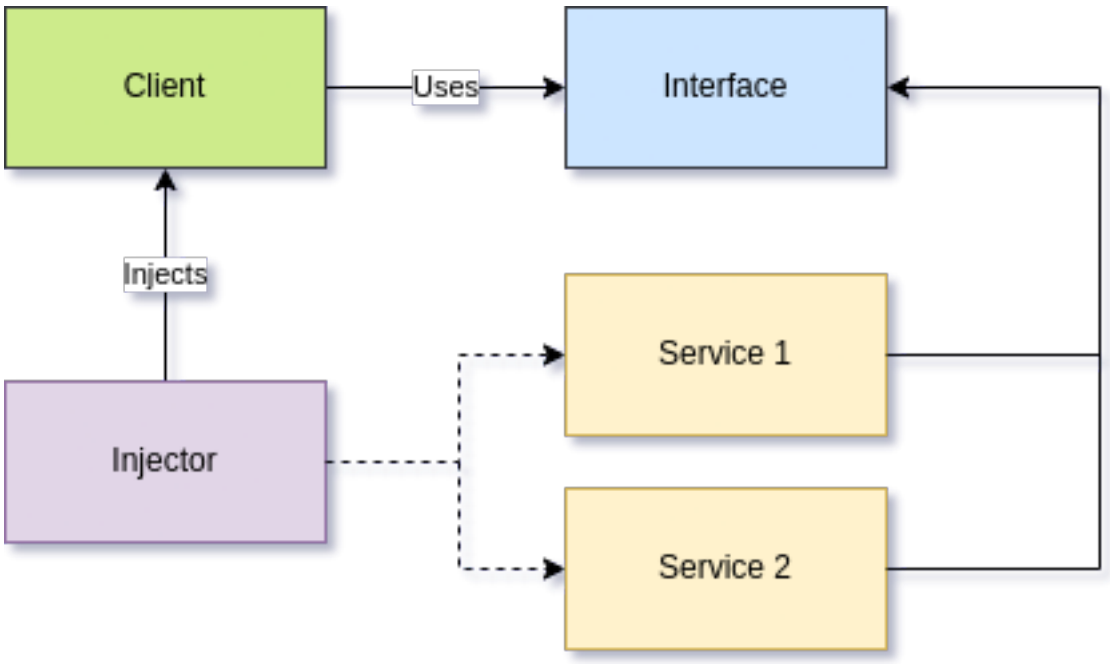
U srži većine modernih veb radnih okvira leži sistem za umetanje zavisnosti (eng. *dependency injection* - DI). DI je dizajn šablon koji za ulogu ima da objektima koje kreira umetne druge objekte (zavisnosti) od kojih zavisi. DI je forma inverzije kontrole (eng. *inversion of control* - IoC) i za cilj ima SoC (eng. *separation of concerns*), odnosno da razdvoji logiku kreiranja objekata od biznis logike koju oni predstavljaju. Ovo za rezultat ima da su objekti međusobno veoma labavo povezani (eng. *loosy coupled*).

U praksi se DI koristiti tako što definišemo klasu i njen konstruktor - tipično se koristi konsturktor za DI, ali postoje i drugi načini o kojima ćemo kasnije pričati. Parametri u konstruktoru biće podrazumevani kao zavisnosti koje sistem za DI treba da razreši. Tipovi parametara mogu da budu klase ili interfejsi.

Primer u pseudo kodu bi izgledao ovako:

```
class UserService:
  constructor(user_repository: UserRepository, role_service: RoleService):
    this.user_repository = user_repository
    this.role_service = role_service
```

U ovom slučaju klasa **UserService** zavisi od **UserRepository** i **RoleService**. Ove zavisnosti se rešavaju tako što se u konstruktoru klase **UserService** prosleđuju objekti koji implementiraju interfejse **UserRepository** i **RoleService**. Kasnije možemo koristiti klasu UserService bez prethodnog znanja o tome da li i kako možemo kreirati objekte klasa/interfejsa UserRepository i RoleService. Na ovaj način se postiže inverzija kontrole - frejmvork je umesto nas, programera, zadužen za kreiranje objekata.

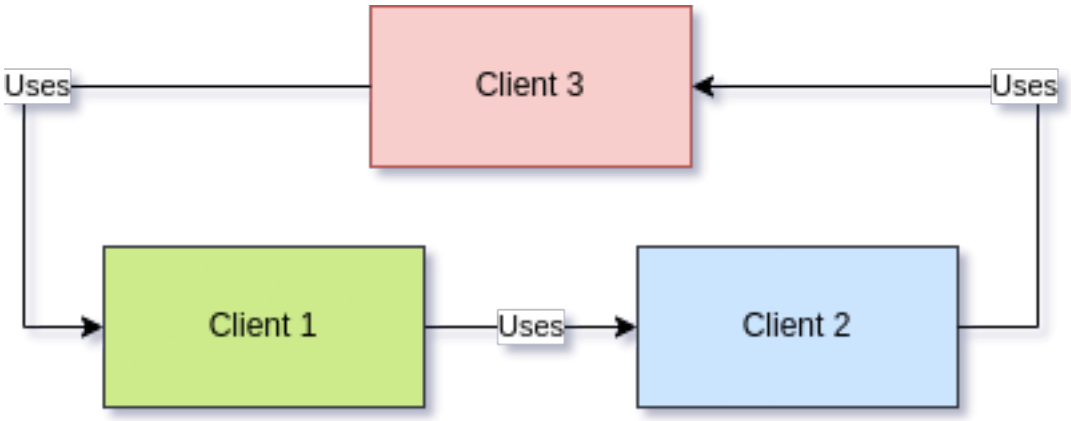


Dijagram umetanja zavisnosti

Na slici vidimo dijagram umetanja zavisnosti. Klasa **Client** ima jednu zavisnost definisanu preko interfejsa **Interface**. U sistemu prikazanom na dijagramu postioje dve klase koje implementiraju dati interfejs. **Injector** u toku izvršenja (eng. *runtime*) može da odabere koja će implementacija zapravo biti korišćena za kreiranje objekta **Client**.

DI umnogome rešava problem kreiranja objekata, ali pri rešavanju tog problema može doći i do problema koji se zove cirkularna zavisnost. Cirkularna zavisnost je problem koji se javlja kada klasa u svom nizu zavisnosti ima samu sebe.

Cirkularna zavisnost



Cirkularna zavisnost

U situaciji opisanoj dijagramom nastaje problem gde će *injector*, ukoliko pokuša da kreira objekat **Client 1** morati prvo da kreira objekat **Client 2**, a usled toga i **Client 3**. Kada se kreira objekat **Client 3** on će

pokušati da kreira objekat **Client 1**. Ovo će se ponavljati u beskonačnost. Rešavanje ovog problema zahteva ili reorganizaciju zavisnosti ili umetanje kroz neki drugi mehanizam koji dozvoljava instanciranje objekata.

Vrste umetanja zavisnosti

Umetanje zavisnosti se može realizovati na više načina. Opisaćemo tipove i njihove prednosti i mane.

1. Umetanje kroz konstruktor - najčešći način umetanja zavisnosti. Prednost ovog načina je što je najjednostavniji za implementaciju i, s obzirom na to da je konstruktor jedini način da se kreira instanca objekta, makar u Javi, pruža atomičnu akciju pri kojoj će sve zavisnosti obavezno biti prisutne u tom trenutku. Mana ovog pristupa je to što se dešava da zbog dizajna aplikacije nećemo moći da pružimo sve zavisnosti. Ostale metode DI rešavaju ovaj problem.
2. Umetanje kroz setere - ovaj način umetanja zavisnosti je sličan prethodnom, ali se zavisnosti umetaju kroz setere. Prednost ovog načina je što se zavisnosti mogu umetati u bilo kom trenutku, a ne samo u konstruktoru. Mana ovog pristupa je to što u trenutku instanciranja objekta mogu, ali ne moraju biti razrešene sve njegove zavisnosti. Ovo često može dovesti do grešaka.
3. Umetanje kroz polja - ovaj način umetanja zavisnosti je takođe sličan svom prethodniku, ali se zavisnosti umetaju kroz polja - ne pozivom metoda. Prednosti i mane su gotovo identične ali što se tiče mana ima jednu dodatnu - umetanje kroz polje uglavnom zahteva da postoji neki oblik introspekcije *runtime*-a u samom jeziku (u Javi je ovo **Reflection API**) i samim tim je sporije i često može doći do grešaka.

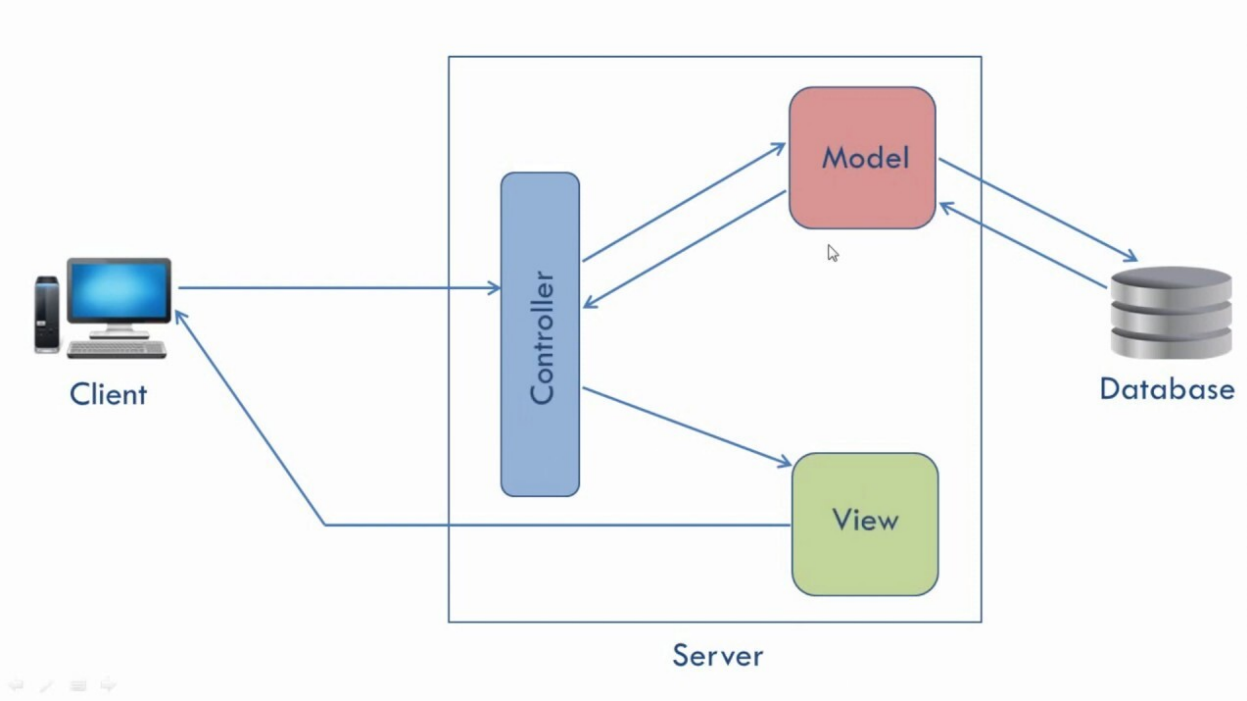
Zaključak je da je najbolje koristiti umetanje kroz konstruktor, ali da se u nekim situacijama može ili mora koristiti i umetanje kroz setere ili polja.

MVC arhitektura

Predstavili smo osnovnu specifikaciju komunikacije koju će naš framework i aplikacija koja je napisana u njemu koristiti. Naredni korak je predstaviti arhitekturu aplikacije odnosno arhitekturalni šablon (eng. *pattern*) koji ćemo koristiti. Arhitekturalni pattern predstavlja skup pravila za struktuiranje i organizaciju projekta. Pored toga, arhitekturalni pattern sadrži i skup pravila koja definišu tok podataka u projektu. Arhitekturalni šabloni su proistekli iz višedecenijskog iskustva u rešavanju sličnih problema i predstavljaju osnovu za razvoj projekta. Ovo, kao i framework sam po sebi olakšava programerima rad na projektu jer definiše pravila za organizaciju istog.

Za aplikaciju smo odabrali MVC arhitekturu/šablon zbog njegove popularnosti i jednostavnosti. MVC je skraćenica za **Model-View-Controller**. Ova arhitektura je jedna od najučestalijih arhitektura u veb aplikacijama. Osnovni principi ove arhitekture su:

- **Model** - predstavlja podatke
- **View** - predstavlja prikaz podataka
- **Controller** - predstavlja logiku aplikacije



MVC arhitektura - Abhay Redkar - What is MVC architecture?

MVC arhitektura se trudi da odvoji logiku aplikacije od prikaza podataka. Ovo omogućava da se aplikacija lakše razvija i održava. Moderni MVC radni okviri prate ovaj šablon i time omogućavaju developerima da pišu čist i struktuiran kod. Ovo im pomaže da beneficiraju od svih nivoa modularnosti.

Tok komunikacije u MVC arhitekturi je sledeći:

1. Klijent (pretraživač) šalje zahtev serveru.
2. Server prima zahtev i prosleđuje ga *Controller*-u.
3. *Controller* obradjuje zahtev i poziva odgovarajući Model.
4. Model vrši potrebne izmene ili komunikaciju sa bazom i vraća *Controller*-u podatke.
5. *Controller* vrši potrebne izmene i vraća *View*-u.
6. *View* vrši potrebne izmene i vraća klijentu.
7. Klijent (pretraživač) prikazuje podatke.

Struktura projekta

Projekti koji prate MVC arhitekturu su često i monolitni po dizajnu tako da ćemo predstaviti jedan takav projekat kao primer:

```
src
├──controller
|   └──UserController.java
├──model
|   └──User.java
└──view
  └──user
    └──index.html
```

Ovo je najčešća struktura MVC projekata. Jedna iteracije ove strukture je uvođenje i servisnog layer-a koji je zadužen za domensku logiku, dok je model layer zadužen isključivo za komunikaciju sa bazom.

```
Browser
↑  ↓
|  UserController.java
|  ↓          ↑ ↓
index.html  User.java
              ↑ ↓
            Database
```

Ovakav pristup omogućava lakši razvoj aplikacija jer se prati modularno struktuiranje klasa u projektu. Sve klase zadužene za *controller* sloj se nalaze u *controller* paketu, *view* klase u *view* paketu itd.

Kreiranje templating jezika

Obradili smo većinu koncepata za koje će radni okvir biti zadužen da apstrahuje. U ovom poglavlju obradićemo *view templating*, odnosno kreiranje templejting jezika koji će nam omogućiti da kreiramo dinamičke strane generisane na serveru (eng. *server-side rendered*).

Templejting jezici

Templejting jezici se koriste na serveru za prikaz dinamičkih stranica. Bez templejtinga ne bismo mogli da na jednostavan način u bilo koju stranu ubacimo dinamičke podatke. Na primer: zamislimo da imamo prodavnicu koja ima proizvode. Mi ne možemo bez prethodnog poznavanja broja proizvoda kreirati statičku stranicu. U najgorem slučaju i možemo ručno napisati HTML strane za svaki od proizvoda, ali šta ćemo kada se doda novi proizvod ili pak promeni cena nekog proizvoda. U takvim situacijama na scenu stupa *view templating*. Templating jezik/sistem se sastoji iz dva dela:

1. **Templating engine** - *engine* koji generiše dinamičke stranice na osnovu *template*-a i podataka.
2. **Template** - *template* je fajl koji sadrži statički deo stranice i specijalne tagove koji se koriste za dinamičko popunjavanje strane podacima.

Templating engine popunjava *template* podacima i vraća rezultujući fajl klijentu. *Templating engine* može da koristi različite *template* jezike. Da bismo lakše objasnili šta je templejting jezik, navešćemo popularne primere istih odnosno primere četiri različita pristupa templejtingu u tri različita programska jezika:

1. **Jinja2** - Python templejting jezik

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flask Jinja2 Example</title>
  </head>
  <body>
    <h1>Flask Jinja2 Example</h1>
    <p>Hello, {{ name }}.</p>
    {{ for item in items }}
      <p>{{ item }}</p>
    {{ endfor }}
  </body>
</html>
```

Jinja2 se zasniva na sintaksi koja je slična *Pythonu*. Ovaj templejting jezik je popularan u *Python* okruženju. Koristi {{ i }} za indikovanje template blokova.

2. **Handlebars** - JavaScript templating jezik

```
<!DOCTYPE html>
<html>
  <head>
    <title>Handlebars Example</title>
  </head>
  <body>
    <h1>Handlebars Example</h1>
    <p>Hello, {{name}}.</p>
    {{#each items}}
      <p>{{this}}</p>
    {{/each}}
  </body>
</html>
```

Handlebars je templejting jezik koji je popularan u JavaScript okruženju. Kao i **Jinja2** koristi {{ i }} za indiciranje *template* blokova. *Handlebars* jezik je sličan *Mustache* jeziku.

3. **JSP** - Java templejting jezik

```
<!DOCTYPE html>
<html>
  <head>
    <title>JSP Example</title>
  </head>
  <body>
    <h1>JSP Example</h1>
    <p>Hello, <%= name %>.</p>
    <% for (String item : items) { %>
      <p><%= item %></p>
    <% } %>
  </body>
</html>
```

JSP je templejting jezik koji je popularan u *Java* okruženju. Ovaj templejting jezik koristi `<%` i `%>` za indiciranje templejt blokova. U *JSP* blokovima, koji se inače zovu *skriptleti* (eng. *scriptlets*), možemo da pišemo gotovo sve funkcionalnosti *Java* programskog jezika. *JSP* je, iako relativno zastareo, veoma moćan jezik.

4. **Thymeleaf** - Java templejting jezik

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Thymeleaf Example</title>
  </head>
  <body>
    <h1>Thymeleaf Example</h1>
    <p th:text="'Hello, ' + ${name} + '.'"></p>
    <p th:each="item : ${items}">
      <span th:text="${item}"></span>
    </p>
  </body>
</html>
```

Thymeleaf je templejting jezik koji je popularan u *Java* okruženju i prirodni naslednik *JSP*-a. Ovaj tem-

plejting jezik koristi `th:` za indiciranje templejt blokova. *Thymeleaf* se umnogome razlikuje od prethodnih templejt jezika u tome što se bez problema može interpretirati kao čista HTML strana jer su sve njegove funkcionalnosti u strane "markirane" kao HTML5 atributi. Ovo može da bude od velike pomoći prilikom razvoja dizajna stranice jer ne moramo da imamo funkcionalni *backend* aplikaciju da bismo prikazali sam HTML sadržaj.

Templating engine

Templating engine je program koji se koristi za generisanje dinamičkih stranica na osnovu *template*-a i podataka. *Templating engine* može da koristi različite *template* jezike. *Templating engine* se sastoji iz dva dela:

1. **Prevodilac** - (eng. *compiler*) prevodilac koji parsira (prevodi) templejt i generiše apstraktno sintaksno stablo - AST (eng. *abstract syntax tree*) *template*-a. *Template* parser se sastoji iz dva dela:
    1. **Lexer** - lekser koji parsira templejt, odnsono tekst programa i generiše tokene. Tokeni su strukture koje sadrže informacije o tipu tokena i njegovoj vrednosti. Primer tokena je `{{` koji označava početak template bloka ili `if` koji označava if petlju.

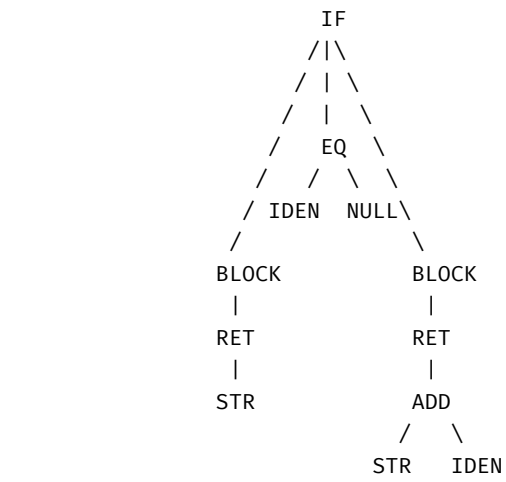
       Lekser za zadatak ima da tekst templejta pretvori u tokene koji imaju veću semantičku vrednosti. Uzmimo za primer sledeći kod: `if (user == null) { return "Hello, guest."; } else { return "Hello, " + user; }` bi se leksiralo u tokene: `IF, LPAREN, IDENTIFIER, EQUAL, NULL, RPAREN, LBRACE, RETURN, STRING, SEMICOLON, RBRACE, ELSE, LBRACE, RETURN, STRING, PLUS, IDENTIFIER, SEMICOLON, RBRACE`

       Ovakav format podataka je mnogo lakši za parsiranje u AST u sledećem koraku.

    2. **Parser** - služi za parsiranje tokena i generisanje AST-a. AST je struktura koja sadrži informacije o strukturi jezika. AST čvor uglavnom ima vrednost i levu i desnu stranu.

       Parser ima za zadatak da tokene, na osnovu sintaksnih pravila programskog jezika pretvori u AST. Ako dođe do ne poklapanja pravila jezika sa očekivanim tokenima u toku parsiranja stabla nastaju tzv. sintaksne greške koje programer mora da ispravi.

       Prethodni primer bi se na sledeći način parsirao u AST: "' # neka imena čvorova su skraćena radi preglednosti



       "'

       Ovakva struktura je manje-više spremna da bude interpretirana i evaluirana u sledećem koraku.

2. **Interpretator** - interpretator (eng. *interpreter*) evaluira AST generisano u koraku kompajliranja i na osnovu njega generiše dinamičku stranu.

Interpretator uzima početni čvor AST-a i redom po pravilima svakog čvora evaluira njegovu vrednost. Na primer, ako se u AST-u nađe čvor `ADD` interpretator će evaluirati vrednost njegovog levog i desnog čvora i zatim će izračunati njihovu vrednost - to će biti vrednost `ADD` čvora. Ako se u AST-u nađe čvor `IF` interpretator će evaluirati vrednost njegovog uslova i zatim će evaluirati vrednost njegovog *if-true* bloka ako je uslov ispunjen. Finalni razultat ove interpretacije, u slučaju *template engine*-a biće validan HTML string koje će biti vraćen klijentu.

Interpretator je najbitniji deo *templating engine*-a jer je on onaj koji evaluira AST i generiše dinamičku stranicu. Interpretator je najčešće implementiran kao rekurzivna funkcija koja prolazi kroz AST i evaluira njegove čvorove. Interpretator pored evaluacije vodi računa o konceptima programskog jezika kao što su globalne i lokalne promenljive, uvoz klasa i fajlova itd. Više o svim ovim konceptima govorićemo kada budemo obrđivali njihovu implementaciju u radnom okviru.

ORM i komunikacija sa bazom

Na kraju, dolazimo do poslednje stavke koju ćemo teorijski obraditi. To je pojam ORM-a i komunikacije sa bazama podataka. Komunikacija sa bazom podataka je jedan od integralnih delova svake veb aplikacije. Osim prezentacionih sajtova, gotovo svaki sistem ima neki vid očuvanja podataka (eng. *data persistence*). Operacije nad bazom podataka mogu biti obavljene na dva načina: direktno koristeći driver/biblioteku ili putem ORM-a koji apstrahuje te operacije u pozive metoda nad objektima/strukturama u jeziku u kome je implementiran. Te objekte/strukture drugačije nazivamo modelima.

Direktan pristup

Kod direktnog pristupa bazi mi uglavnom ručno pišemo i popunjavamo upite ka samoj bazi. Primer ovog pristupa bio bi *JDBC* (*Java Database Connectivity) u Javi:

```
// konekcija na bazu
Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/mydb", "user",
"password");

// kreiranje upita
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?");
stmt.setInt(1, 1);

// izvršavanje upita
ResultSet rs = stmt.executeQuery();

// čitanje rezultata
while (rs.next()) {
  System.out.println(rs.getString("name"));
}
```

Ovaj pristup ima nekoliko nedostataka. Ručno popunjavanje upita je veoma neefikasno i često dovodi do *SQL injection* napada. Kod se često duplira i kod kompleksnijih upita kod često bude teško održiv. Takođe, prilikom promene bilo koje od klasa za koje imamo napisane upite postoji velika šansa da ćemo većinu njih morati ponovo da napišemo. Takođe, bez dodatne konfiguracije upiti ne podležu sintaksnim proverama tako da bismo. bez integracionih testova, greške u upitima otkrili tek u runtime-u. Kada su u pitanju performanse, direktan pristup bazi i maksimalna fleksibilnost u kreiranju upita može da bude pogodna. Često se ORM pristup izbegava kod kompleksnih upita, ili se pribegava pristupu kreiranja pogleda (eng. *view*).

ORM pristup

Object-relational mapping ili ORM je pristup gde biblioteka pruža zajednički interfejs ka bilo kojoj podržanoj bazi podataka kroz klase/strukture u tom jeziku za koji je pisana. Primer ORM-a u Java programskom jeziku je **Hibernate**. *Hibernate*, ili bilo koji drugi ORM, koristi izmene nad objektima modelskih

klasa da kreira upite koji će ekvivalentni red u bazi izmeniti u skladu sa izmenama na objektu. Iz toga proizilazi da je objekat mapiran u bazi podataka, odnosno tako dolazimo do *objektno-relacionog* mapiranja. S obzirom na to da se ORM često konfiguriše na osnovu *runtime* introspekcije podataka (pričali smo o *Reflection API*-u) dolazi do problema sa performansama koje nekad mogu biti razlog zašto ne želimo uvek da koristimo ORM.

Primer korišćenja ORM-a u Javi:

```
// kreiranje sesije

Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();

// kreiranje objekta
User user = new User();
user.setName("John Doe");
user.setAge(25);

// čuvanje objekta u bazi
session.save(user);

// čitanje objekta iz baze
User user = (User) session.get(User.class, 1);

// izmena objekta
user.setName("Jane Doe");

// čuvanje izmena
session.update(user);

// brisanje objekta
session.delete(user);

// zatvaranje sesije
session.getTransaction().commit();
session.close();
```

Kao što vidimo, ORM pruža veoma jednostavan i fleksibilan pristup radu sa bazom podataka. Ne moramo da imamo bilo kakvu informaciju o tome koja je baza podataka u pitanju jer će ORM generisati sve upite prilikom obavljanja operacija. Ukoliko se koristi ORM pristup, često se koristi i *migration* alat koji omogućava da se model baze podataka nesmetano modifikuje u skladu sa modifikacijom klasa, u ORM terminologiji - entiteta. Neki od poznatijih Java migracionih alata su **Flyway** i **Liquibase**. Oni omogućavaju verzioniranje modela baze time što se svaka izmena baze beleži kroz migracione skripte. Na taj način se postiže to da je model baze uvek ponovljiv na novim mašinama i serverima na kojima se vrši instalaicija aplikacije.

Query builder pristup

Još jedan od pristupa za komunikaciju sa bazom podataka je *query builder* pristup. Ovaj pristup leži negde između pisanja upita i korišćenja ORM-a. Kod korišćenja query buildera imamo benefite validacije upita jer koristimo dinamički interfejs (neka vrsta DSL-a - *domain specific language*) i benefite performansi koje nam daje korišćenje upita.

Primer korišćenja **Querydsl** query buildera u Javi:

```
// kreiranje upita
JPAQuery query = new JPAQuery(em);
```

```
QUser user = QUser.user;
List<User> users = query.from(user).where(user.name.eq("John Doe")).list(user);

// izmena user-a
JPAUpdateClause update = new JPAUpdateClause(em, user);
update.where(user.name.eq("John Doe")).set(user.name, "Jane Doe").execute();

// brisanje user-a
JPADeleteClause delete = new JPADeleteClause(em, user);
delete.where(user.name.eq("John Doe")).execute();
```

Zaključak

Koji god pristup izaberemo moramo imati na umu koje zahteve imamo koji se tiču performansi, obim i kompleksnost modela, održavanje itd. ali u većini slučajeva ORM je sasvim zadovoljavajuće rešenje koje umnogome olakšava razvoj web aplikacija.

Sada kada imamo teorijsku osnovu o tome šta sve jedan radni okvir treba da pokrije, možemo da krenemo sa objašnjavanjem kako je svaki od tih koncepata implementiran u **Grain** radnom okviru.

Studija slučaja

Grain

Grain je radni okvir koji je nastao kao rezultat potrebe za jednostavnim i fleksibilnim alatom za razvoj veb aplikacija. Uz pomoć **Grain** radnog okvira možemo da razvijamo veb aplikacije u **Java** programskom jeziku koje lako mogu da budu proširene bilo kojim drugim bibliotekama jer Grain framework sam po sebi podržava **dependency injection**. *Grain* radnom okviru su velika inspiracija **Spring** i **Spring Boot**, što ćemo videti u narednim primerima. Primeri će se sastojati od proširenja koncepta koji su opisani u poglavlju gde smo obrađivali teorijsku postavku, primera u *Grain* radnom okviru, primera u *Spring Boot*-u i opisa implementacije.

HTTP

Prva i osnovna funkcionalnost veb okvira je da ima mogućnost kreiranja servera, odnosno proces koji može da razmenjuje informacije koristeći HTTP protokolo. Za potrebe ove funkcionalnosti proces mora da ima mogućnost da otvori osluškujući (eng. *listening*) *socket* na *host* računaru, zatim da čita podatke koji stižu na taj socket u neblokirajućem (eng. *non-blocking*) režimu - što će reći paralelno koristeći više niti (eng. *thread*). Pročitane podatke mora parsirati u HTTP zahtev i imati mogućnost da HTTP odgovor upiše natrag u isti *socket*.

Primer

Implementacija osnovnog HTTP servera koji vraća bazični odgovor na HTTP zahtev:

```
@SpringBootApplication
public class HttpServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(HttpServerApplication.class, args);
  }

  @Controller
  @RequestMapping("/")
  public class HttpServerController {
```

```
    @GetMapping
    public @ResponseBody String index() {
      return "Hello World!";
    }
  }
}
```

Nakon startovanja ove *Spring Boot* aplikacije ukoliko u pretraživaču odemo na `http://localhost:8080` dobićemo odgovor `Hello World!`. Vidimo da postoji veoma malo koda koji mora biti napisan da bi se dobila ova bazična funkcionalnost. Neki delovi koda mogu i da budu obrisani ali su ostavljeni zbog čitljivosti.

Ekvivalentna implementacija u Grain radnom okviru:

```
public class HttpServerApplication extends GrainApp {
  public static void main(String[] args) {
    GrainAppRunner.run(HttpServerApplication.class);
  }

  @Controller
  @RequestMapping("/")
  public class HttpServerController {
    @GetMapping
    public String index() {
      return "Hello World!";
    }
  }
}
```

Možemo da vidimo da su implementacije ove funkcionalnosti veoma slične u oba radna okvira. Kao i kod *Spring* aplikacije, odlaskom na `http://localhost:8080` dobićemo odgovor `Hello World!`.

Implementacija

U nastavku ćemo detaljnije objasniti kako je implementirana osnovna funkcionalnost HTTP servera u Grain radnom okviru. Iako je ovo veoma bazična funkcionalnost, da bismo došli od `GrainAppRunner.run()` do `Hello World!` u pretraživaču moramo da konfigurišemo nekoliko podsistema.

Prilikom startovanja aplikacije `GrainAppRunner` kreira i konfiguriše instancu klase `GrainApp`, odnosno korisničke klase koja nasleđuje `GrainApp`.

Koraci inicijalizacije su sledeći:

1. Kreiranje `Configuration` objekta i učitavanje konfiguracije

`Configuration` objekat sadrži sve ključ-vrednost parove koji predstavljaju kofiguraciona podešavanja za frejmvork. On je neophodan za inicijalizaciju aplikacije jer postoje parametri koji utiču na istu.

1. Učitavanje aktivnih profila - profili su podešeni koristeći `GRAIN_PROFILES_ACTIVE` environment promenljivu.
2. Učitavanje konfiguracije iz `application.properties` fajla i odgovarajućih `.properties` fajlova na osnovu aktivnog profila.
3. Učitavanje konfiguracije iz okruženja (eng. *environment*). Ova konfiguracija ima prednost u odnosu na `.properties` fajlove.

```
String profilesString = Optional.ofNullable(System.getenv(PROFILES_ENV_VARIABLE))
    .orElse(",");
```

```
// učitavanje profila iz okruženja
List<String> profiles = Arrays.stream(profilesString
    .split("\\s*,\\s*"))
    .collect(Collectors.toList());


// učitavanje konfiguracije iz .properties fajlova
PropertiesResolver propertiesResolver = new PropertiesResolver(profiles);
propertiesResolver.resolve("META-INF/application",
    properties::load);
propertiesResolver.resolve("application",
    properties::load);

// učitavanje konfiguracije iz okruženja
EnvironmentResolver environmentResolver = new EnvironmentResolver();
environmentResolver.resolve(this::set);

// vraćanje konfiguracije u okruženje (environment)
// ovo olakšava pristup konfiguraciji iz drugih delova aplikacije
// koji zahtevaju svoje konfiguracione fajlove tako da i one mogu
// biti konfigurisane preko .properties fajlova
properties.forEach((key, value) ->
    System.setProperty(key.toString(), value.toString()));
```

2. Kreiranje `ApplicationContext` objekta

`ApplicationContext` interfejs predstavlja srž aplikacije. On sadrži `DependencyContainer` koji sadrži sve inicijalizovane komponente koje su kreiranje u toku umetanja zavisnosti. O tome ćemo govoriti u kasnijem poglavlju. Ovaj kontekst objekat ima jednu statičku instancu kojoj se može pristupiti putem `ApplicationContextHolder` singltona. `ApplicationContext` se takođe može ručno instancirati koristeći `ApplicationContextImpl` klasu koja mu je ujedno i jedina implementacija.

1. Odigravanje DI životnog ciklusa - `ApplicationContext` je zadužen za pokretanje DI životnog ciklusa:

```
public ApplicationContextImpl(String basePackage, Configuration configuration) {
  this.basePackage = basePackage;
  this.configuration = configuration;
  // kreiramo injector
  GrainInjector injector = new GrainInjector(configuration);

  // ucitavamo sve klase koje su anotirane sa odgovarajućim anotacijama
  Set<Class<?>> classes = Arrays.stream(new String[]{GrainApp.getBasePackage(), basePackage})
    .flatMap(pkg -> new GrainJarClassLoader(pkg)
      .loadClasses(cl -> !cl.isAnnotation() && isAnnotationPresent(cl, Grain.class))
      .stream())
    .collect(Collectors.toCollection(LinkedHashSet::new));
  // dodajemo sve klase u dependency injection pipeline
  injector.inject(classes);

  // preuzimamo gotov konfigurisan dependency container
  // koji je zadužen za čuvanje svih inicijalizovanih komponenti
  this.dependencyContainer = grainInitializer.getContainer();
}
```

3. Otvaranje socket-a za HTTP server

Socket mora da bude spreman da paralelno obrađuje zahteve, stoga kreiramo *thread-pool* koji je zadužen za to.

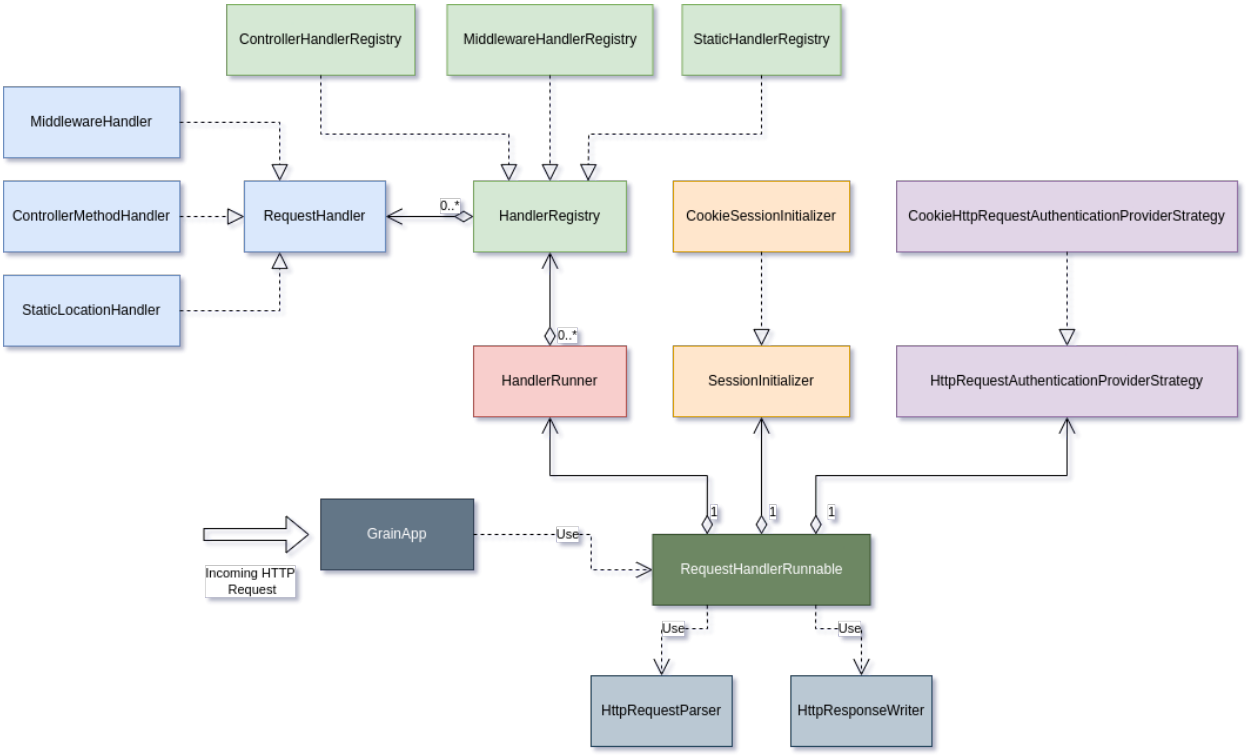
```
// kreiramo thread pool koji će biti zadužen za obradu zahteva
ExecutorService executor =
Executors.newFixedThreadPool(configuration.getInt(ConfigurationKey.SERVER_THREADS));

// kreiramo server socket
try (ServerSocket serverSocket = new
ServerSocket(configuration.getInt(ConfigurationKey.SERVER_PORT), -1,
InetAddress.getByName(configuration.get(ConfigurationKey.SERVER_HOST)))) {

  // sve dok je aplikacija "running" slušamo zahteve
  // i izvršavamo ih u thread pool-u
  while (running) {
    Socket socket = serverSocket.accept();
    executor.execute(new RequestHandlerRunnable(context, socket));
  }

} catch (UnknownHostException e) {
  throw new AppInitializationException("Unable to resolve host " +
  configuration.get(ConfigurationKey.SERVER_HOST), e);
} catch (IOException e) {
  throw new AppInitializationException("Unable to create server socket", e);
}
```

Za opsluživanje zahteva koristimo `RequestHandlerRunnable` koji je zadužen za parsiranje zahteva, pozivanje odgovarajućeg *handlera* i slanje odgovora. Ova klasa naravno implementira `Runnable` interfejs koji joj omugućava da se izvršava na odvojenoj niti. Više o ovoj klasi ćemo govoriti kada budemo pričali o tome kako se procesuira HTTP zahtev.

Po završetku ovih koraka imamo kreiran HTTP server koji je spreman da sluša zahteve na podrazumevanom port-u 8080.

```
17-10-2022 12:37:42.227   DEBUG - [           main] c._.grain.core.component.GrainInjector   :
Evaluating @Value annotations
17-10-2022 12:37:42.227   DEBUG - [           main] c._.grain.core.component.GrainInjector   :
Calling lifecycle methods
17-10-2022 12:37:42.233   DEBUG - [           main] c._.grain.core.component.GrainInjector   :
Loaded 51 Grain classes
17-10-2022 12:37:42.233    INFO - [           main] com._7aske.grain.GrainApp                :
Initialized application context
17-10-2022 12:37:42.233   DEBUG - [           main] com._7aske.grain.GrainAppRunner          :
Startup took 1614ms
17-10-2022 12:37:42.233    INFO - [           main] com._7aske.grain.GrainApp                :
Started Grain application on 0.0.0.0:8080
```

Klasni dijagram toka HTTP zahteva



Tok HTTP zahteva

Na dijagramu vidimo sve klase koje su relevantne za tok jednog HTTP zahteva u Grain radnom okviru. Redom ćemo objasniti koje klase imaju koji svrhu i šta se dešava sa HTTP zahtevom kada on pristigne u aplikaciju.

Inicijalni *"accept loop"* se nalazi u `GrainApp` klasi i on je odgovoran za prihvatanje zahteva metodom `ServerSocket#accept`. Ova metoda je blokirajuća tako da odmah pri ostvarivanju konekcije obradu zahteva šaljemo u *thread pool* pomoću klase `RequestHandlerRunnable`.

`RequestHandlerRunnable` je klasa koja nasleđuje `Runnable` interfejs i odgovorna je za obradu zahteva. Ovde parsiramo HTTP zahtev koristeći `HttpRequestParser` klasu. Ova klasa je dizajnirana tako da parsira HTTP zahtev po HTTP/1.1 specifikaciji o kojoj smo već govorili.

```
// kreiramo buffer za čitanje zahteva
byte[] byteBuffer = new byte[BYTE_BUF_SIZE];

int n;
do {
  // čitamo zahtev
  n = reader.read(byteBuffer);
  for (int i = 0; i < n; ++i) {
    // kopiramo učitane bajtove u StringBuffer zahteva
    buffer.append((char) byteBuffer[i]);
  }
} while (n == byteBuffer.length);

// proveravamo da li za zahtev završava sa CRLF
```

```
int crlfIndex = buffer.indexOf(CRLF);
if (crlfIndex == -1) {
  throw new HttpParsingException();
}

// parsiramo request line
String requestLineString = buffer.substring(0, crlfIndex);
String[] requestLineParts = requestLineString.split(" ");

// ako request line nije parsiran kako treba bacamo izuzetak
if (requestLineParts.length != 3) {
  throw new HttpParsingException();
}

// parsirane delove request line-a setujemo u HttpRequest objekat
request.setMethod(HttpMethod.resolve(requestLineParts[0]));
request.setPath(requestLineParts[1]);
request.setVersion(requestLineParts[2]);
```

Ovo parče koda u `HttpRequestParser` klasi je odgovorno za učitavanje zahteva u memoriju i parsiranje prvog i osnovnog dela svakog HTTP zahteva - *request line*-a. Kao što smo već imali prilike da pomenemo, *request line* daje osnovne informacije o HTTP zahtevu kao što su HTTP metoda, resurs koji je zahtevan i verzija HTTP protokola. Naredni koraci su učitavanje zaglavlja i tela zahteva.

Nakon parsiranja zahtev je dostupan u `RequestHandlerRunnable` klasi i možemo ga proslediti u odgovarajući *handler* koji će obraditi zahtev i vratiti odgovor. Doduše, pre nego što dođemo do odluke o tome šta ćemo dalje sa zahtevom, moramo da inicijalizujemo sesiju koristeći `SessionInitializer` interfejs i setujemo autentikaciju koristeći `HttpRequestAuthenticationProviderStrategy` interfejs. `SessionInitializer` je interfejs koji po default-u ima samo jednu implementaciju a to je `CookieSessionInitializer`, što znači da će on omogućiti radnom okviru da prati sesiju korisnika koristeći HTTP kolačiće. `SessionInitializer` kreira jedinstveni identifikator za svaki kolačić i taj identifikator čuva u memoriji. Na osnovu identifikatora moguće je čuvati i povratiti informacije o korisniku koristeći `SessionStore`. Rekli smo da se podaci i identifikator sesije čuvaju u memoriji - to se dešava zbog toga što je jedina implementacija `SessionStore` interfejsa `InMemorySessionStore`. Jedan od dizajn fokusa u radnom okviru je bio *Dependency Inversion* - jedan od *SOLID* principa koji nalaže da klase treba da zavise od apstraktnih klasa i interfejsa pre nego od konkretnih klasa. U ovom, kao i mnogim drugim, moguće je implementirati komponentu koja implementira `SessionStore` interfejs i time zamenimo funkcionalnosti čuvanja podataka u memoriji npr. čuvanjem podataka u bazi ili nekom drugom servisu.

Posle inicijalizovanja sesije naredni korak je učitavanje autentikacije. Ovo takođe funkcioniše pomoću `SessionStore` interfejsa u čijoj implementaciji su sačuvani podaci o korisniku koji su vezani za autentifikaciju i autorizaciju. Za konfigurisanje metoda koji koristimo za dobavljanje podataka za autentikaciju koristimo *Strategy* dizajn šablon i interfejs `HttpRequestAuthenticationProviderStrategy`. Više o autentikaciji i autorizaciji u nastavku.

Kada su sesija i autentikacija konfigurisani naredni korak je pronaći *handler* koji će procesuirati zahtev. Svi dostupni *handler*-i moraju da implementiraju `RequestHandler` interfejs koji otkriva metode kao što su `canHandle` i `handle` koje služe za proveru da li *handler* može da procesuira zahtev i procesuiranje zahteva, respektivno. Dostupni *handler*-i su:

1. `StaticLocationHandler` - *handler* koji služi za obradu statičkih resursa kao što su slike, CSS fajlovi, JavaScript fajlovi, itd.

`StaticLocationHandler` se koristi sa serviranje statičkih fajlova. Ovaj *handler* sadrži listu lokacija u kojima će pretraživati fajlove koji odgovaraju URL-u zahteva. Implementacija `handle` metode:

```java
@Override
public void handle(HttpRequest request, HttpResponse response) {
  // kreiramo apsolutnu putanju do fajla imajući u vidu registrovanu lokaciju
  // handlera i URL zahteva
  Path path = Paths.get(location, request.getPath());

  try (InputStream inputStream = getInputStream(path)) {
    // parsiramo ekstenziju fajla u cilju određivanja MIME tipa
    response.setHeader(HttpHeaders.CONTENT_TYPE, probeContentTypeNoThrow(path, "text/html"));

    // čitamo sadržaj fajla i upisujemo ga u telo odgovora
    response.getOutputStream().write(inputStream.readAllBytes());

    // postavljamo status odgovora na 200 OK
    response.setStatus(HttpStatus.OK);

    request.setHandled(true);
  } catch (IOException ex) {
    throw new HttpException.NotFound(request.getPath());
  }
}
```

2. `ControllerMethodHandler` - *handler* koji služi za obradu zahteva pomoću metoda definisanim u kontrolerskim klasama.

`ControllerMethodHandler` je najkompleksniji od svih handlera. On mora da koristi informacije dobijene putem Reflection-a da bi mogao da zna kako da parsira svaki request. Na osnovu povratne metode u kontroleru možemo da vratimo klijentu različite odgovore: tekst, HTML stranu, JSON itd. Takođe, metode kontrolera imaju mogućnost da prime različite vrednosti iz zahteva kao svoje parametre. U zavisnosti od tipa parametra metoda kontrolera može da primi `HttpResponse`, `HttpRequest`, bilo koju vrednost anotiranu sa `@RequestBody`, mapu header-a itd. Kontroler metoda može biti registrovana za određenu putanju korišćenjem `@RequestMapping` anotacije i njenih specijalizacija.

Implementacija `handle` metode:

```java
@Override
public void handle(HttpRequest request, HttpResponse response) throws IOException {

  // parsiramo parametre handler metode
  Parameter[] declaredParams = method.getParameters();
  Object[] params = new Object[declaredParams.length];
  for (int i = 0; i < declaredParams.length; i++) {
    Parameter param = declaredParams[i];
    if (param.getType().equals(HttpRequest.class)) {
      params[i] = request;
    } else if (param.getType().equals(HttpResponse.class)) {
      params[i] = response;
    } else if (param.getType().equals(Session.class)) {
      params[i] = request.getSession();
    } else if (param.isAnnotationPresent(JsonBody.class)) {
      params[i] = new JsonDeserializer<>(param.getType()).deserialize((JsonObject)
request.getBody());
    } else if (param.isAnnotationPresent(FormBody.class)) {
      // ...
    } else if (param.isAnnotationPresent(RequestParam.class)) {
```

```
      // ...
    } else if (param.isAnnotationPresent(PathVariable.class)) {
      // ...
    } else if (Map.class.isAssignableFrom(param.getType())) {
      params[i] = ((JsonObject) request.getBody()).getData();
    }
  }

  // pozivamo handler metodu sa prilagođenim parametrima
  Object result = method.invoke(params);

  // upisujemo body odgovora na osnovu povratnog tipa rezultata metode
  if (result == null) {
    String requestContentType = request.getHeader(CONTENT_TYPE);
    response.setHeader(CONTENT_TYPE, requestContentType == null ? HttpContentType.TEXT_PLAIN :
requestContentType);
  } else if (result instanceof View) {
    viewResolver.resolve((View) result, request, response, request.getSession(),
SecurityContextHolder.getContext().getAuthentication());
  } else if (result instanceof JsonResponse) {
    response.setStatus(((JsonResponse<?>) result).getStatus());
    response.getOutputStream().write(((JsonResponse<?>)
result).getBody().toJsonString().getBytes());
    response.addHeaders(((JsonResponse<?>) result).getHeaders());
  } else if (result instanceof JsonString) {
    response.getOutputStream().write(((JsonString) result).toJsonString().getBytes());
    response.setHeader(CONTENT_TYPE, HttpContentType.APPLICATION_JSON);
  } else if (result instanceof Object[]) {
    response.getOutputStream().write(new JsonArray((Object[])
result).toJsonString().getBytes());
    response.setHeader(CONTENT_TYPE, HttpContentType.APPLICATION_JSON);
  } else if (result instanceof String) {
    // ...
  } else {
    response.getOutputStream().write(result.toString().getBytes());
  if (response.getHeader(CONTENT_TYPE) == null)
    response.setHeader(CONTENT_TYPE, HttpContentType.TEXT_PLAIN);
  }

  request.setHandled(true);
}
```

3. `MiddlewareRequestHandler` - *handler* se poziva pre svakog zahteva i služi za proširivanje funkcionalnosti postojećih *handler*-a, proveru podataka iz zahteva itd.

Middleware ima najprostiju implementaciju od svih handlera. On jednostavno samo poziva `handle` metodu. Implementacija *middleware*-a je u celosti na korisniku. *Middleware* a i svi ostali *handler*-i mogu da imaju `@Order` anotaciju koja određuje prioritet kada postoje *handler*-i koji imaju istu putanju.

Implementacija `handle` metode:

```
@Override
public void handle(HttpRequest request, HttpResponse response) {
  handler.handle(request, response);
}
```

Kao što smo rekli *handler*-i imaju metodu `canHandle` koja služi za proveru da li *handler* može da obradi zahtev. Pomoću nje pronalazimo odgovarajući *handler* koji u zavisnosti od implementacije obrađuje zahtev. Ukoliko *handler* nije pronađen, aplikacija vraća grešku `404`.

JSON

JSON je jedan od najpopularnijih formata za razmenu podataka, stoga je bilo fundamentalno da *Grain* frejmvork ima podršku za njega. Već smo obradili njegovu specifikaciju a sada ćemo pokazati kako je JSON parsiranje implementirano u *Grain* frejmvorku.

Klasa `JsonParser` u *Grain* frejmvorku se koristi za parsiranje JSON stringa u Java objekat. `JsonParser` se trudi ali ne podržava sve stavke iz JSON specifikacije. Na primer, trenutno, ne podržava razmake u ključevima ili eksponencijalnu notaciju za brojeve, ali zato većinu osnovnih funkcionalnosti parsira kako treba. `JsonParser` koristi `StringIterator` koji olakšava parsiranje teksta. Objašnjenje parsiranja JSON string-a biće od koristi kada budemo pričali o leksiranju teksta pri parsiranju template jezika.

JSON objekat se se sastoji iz ključ/vrednost parova od kojih je ključ bilo koji tekst pod navodnicima(“) a vrednost jedan od sledećih tipova:

1. `null` - nulta vrednost
2. `boolean` - `true` ili `false`
3. `number` - broj
4. `string` - tekst pod navodnicima
5. `array` - niz vrednosti
6. `object` - novi objekat koji se sastoji iz ključ/vrednost parova

Svaki od ovih tipova se može identifikovati na osnovu prvog karaktera u JSON stringu. Ukoliko je prvi karakter `{` onda je u pitanju objekat, `[` niz, `"`, tekst, `t` ili `f` boolean, `n` nula vrednost, a u svim ostalim slučajevima broj. Na ovaj način je implementiran i `JsonParser`. Naravno ne smemo da zaboravimo da između svake pročitane vrednosti ili ključa “progutamo” sav beli prostor (eng. *whitespace*).

```
private Object parseValue() {
  String token = iterator.peek();
  switch (token) {
    case "{":
      return parseObject();
    case "[":
      return parseArray();
    case "\"":
      return parseString();
    default:
      return parseOther();
  }
}

private Object parseOther() {
  iterator.eatWhitespace();
  String val = iterator.eatWhile(ch -> !ch.isBlank() && !ch.equals(",") && !ch.equals("}") &&
  !ch.equals("]"));
  if (val.equals("true")) {
    return Boolean.TRUE;
  }
  if (val.equals("false")) {
    return Boolean.FALSE;
  }
```

```
  if (val.equals("null")) {
    return null;
  }
  try {
    double parsed = Double.parseDouble(val);
    if (parsed == (int) parsed) {
      // @Warning this may cause issues with large numbers
      // but removes a lot of headaches with parsing integer types.
      return Integer.parseInt(val);
    } else {
      return parsed;
    }
  } catch (NumberFormatException ex) {
    throw new JsonDeserializationException("Unexpected token '" + val + "' " +
    iterator.getInfo());
  }
}
```

Ovako parsiran string se čuva kao `JsonObject` ili `JsonArray` objekat koji je spreman da bude konvertovan u bilo koji drugi Java objekat. Na primer, parsirani JSON string možemo iskoristiti da popunimo vrednosti atributa bilo koje druge klase uz pomoć `JsonDeserializer` klase. *Reflection API*-jem uzimamo vrednost svakog od atributa (*field*) i na osnovu njegovog tipa parsiramo objekat sačuvan u `JsonObject`-u.

```
if (Byte.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Byte.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Integer.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Float.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Double.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Boolean.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Long.class.isAssignableFrom(field.getType())) {
  // ...
} else if (Character.class.isAssignableFrom(field.getType())) {
  // ...
} else if (String.class.isAssignableFrom(field.getType())) {
  // ...
} else {
  // U svim drugim slučajevima rekurzivno pozivamo JsonDeserializer
}
```

Performanse i podržanost JSON parsiranja u *Grain* frejmvorku nisu idealne ali su dobar *proof of concept*. U budućnosti ćemo implementirati nešto bolje.

Umetanje zavisnosti

Umetanje zavisnosti je jedna od najvažnijih funkcionalnosti svakog modernog frejmvorka i *Grain* nije izuzetak. Umetanje zavisnosti nam omogućava da klase definišemo i kreiramo deklarativno. Na primer, definisali smo klasu `UserController` koja koristi `UserService`. Samim tim što smo `UserService` iskoristili u konstruktoru definisali smo tu klasu kao zavisnost i frejmvork će nam je sam kreirati. Time izbegavamo probleme sa

kreiranjem klasa koje mogu da imaju brdo zavisnih klasa - a da ne pričamo o menadžmentu instanci klasa koje veoma brzo postane neodrživo ako se radi ručno. U *Grain* frejmvorku umetanje zavisnosti je implementirano preko `DependencyContainer`-a koji u *runtime*-u konfiguriše sve komponente(komponente su *Client* objekti definisani ovim šablonom). Komponente su u framework-u nazvane i po njemu samom - zrna od eng. *grain*. `DependencyContainer` interfejs definiše par javnih metoda koje frejmvork interno koristi:

```
public interface DependencyContainer {
    void registerGrain(Object grain);

    <T> T getGrain(Class<T> clazz);

    Collection<Object> getGrainsAnnotatedBy(Class<? extends Annotation> clazz);

    <T> Optional<T> getOptionalGrain(Class<T> clazz);

    <T> Collection<T> getGrains(Class<T> clazz);
}
```

Prilikom kreiranja `ApplicationContext`-a učitavaju se sve klase iz classpath-a koje su anotirane anotacijom `@Grain` ili njenim specijalizacijama se ubacuju u `GrainInjector` koji započinje proces umetanja zavisnosti. Proces umetanja zavisnosti u *Grain* radnom okviru se sastoji iz par koraka koji se ponavljaju za svaku *Client* klasu:

0. Provera `@Condition` uslova

`@Condition` anotacija može da sadrži parče koda pisanog u *Grain* templejting jeziku (koji možemo zvati GTL - *Grain Templating Language*). Interpretator jezika evaluira kod i u zavisnosti od povratne vrednosti biramo da li ćemo datu klasu uključiti u umetanje.

1. Kreiranje `Injectable` objekata

`Injectable` objekti su *wrapper* klase koje pružaju olakšice kada je u pitanju rad sa informacijama o njenim poljima, metodama i konstruktorima. Naravno ovi objekti čuvaju inicijalizovanu komponentu koja će nam kasnije biti dostupna.

2. Umetanje zavisnosti koje su definisane `@Grain` metodama.

Pored klasa, i metode zrna mogu biti anotirane `@Grain` anotacijom. To označava da je ta metoda zapravo *factory* metoda koja definiše novu zavisnost koju možemo umetnuti, kao i bilo koju drugu klasu, odnosno komponentu. Ovo je idealan način da integrišemo eksterne biblioteke u aplikacije pisane u frejmvorku. Naravno, sve `@Grain` metode mogu, kao i klase, da imaju `@Condition` anotaciju čiju vrednost proveravamo takođe u ovom koraku.

3. Dodavanje `Injectable`-a u kontejner.

`DependencyContainer` je zapravio prioritetni red (eng. *priority queue*) koji je definisan tako da prilikom iteracije kroz njega prvo vraća zavisnosti koje imaju manje direktnih drugih zavisnosti. Na ovaj način optimizujemo inicijalizaciju i sprečavamo *backtracking* u tom procesu. `Injectable` klasa razrešava svoje zavisnosti tako što od sopstvenih polja kreira `InejctableReference` objekte koje možemo da tretiramo, za sada, kao "potencijalne objekte". Njihovo razrešenje na realne instance desiće se kasnije, ali za sada su nam potrebni samo da imamo informaciju o tome koliko zavisnosti koja klasa ima.

4. Provera cirkularnih zavisnosti

Pre nego što krenemo sa inicijalizacijom i umetanjem, moramo da proverimo da li neke od naših zavisnosti kreiraju zatvoreni graf. U slučaju da se to desi, ne možemo ostvariti validnu inicijalizaciju. Rešenje ovog problema je prebacivanje neke zavisnosti iz inicijalizacije preko konstruktora u inicijalizaciju preko polja anotacijom `@Inject`. Inicijalizacija preko polja se dešava kasnije u odnosu na inicijalizaciju preko konstruktora i time se može razrešiti cirkularna zavisnost.

5. Inicijalizacija komponente

U ovom koraku se kreira instanca komponente na osnovu njenog konstruktora. Zavisnosti sa najmanje drugih zavisnosti imaju prioritet.

1. Prvi korak je kreiranje instance ove klase. Za ovo je potrebno mapiranje parametara konstruktora na inicijalizovane objekte. Zbog inicijalne provere da li je postoje cirkularne zavisnosti - ovo mapiranje će uvek uspeti. Ukoliko je klasa interfejs kreira se *proxy* intanca koja ima delegate ka *default* metodama tog interfejsa.
2. Pozivaju se sve `@Grain` metode novonastale instance. Rezultujuci objekti se registruju u `Dependency-Container`-u.
3. Setuju se sva `@Inject` polja
4. Popunjavaju se sva polja anotirana sa `@Value`
5. Pozivaju se metode životnog ciklusa (eng. *lifecycle*). Ovo su za sada samo metode anotirane sa `@AfterInit`. U ovim metodama svi parametri su razrešeni na komponente.

Nakon svih ovih koraka sve zavisnosti u `DependencyContainer`-u će biti inicijalizovane.

Primer jedne komponente je klasa koja omogućava integraciju sa Hibernate ORM:

```
@Grain
public class HibernateConfiguration {

  @Grain
  public SessionFactory sessionFactory() {
    org.hibernate.cfg.Configuration configuration = new org.hibernate.cfg.Configuration();

    GrainClassLoader grainClassLoader = new
GrainJarClassLoader(GrainTestApp.class.getPackageName());
    grainClassLoader.loadClasses(cl -> cl.isAnnotationPresent(Entity.class))
      .forEach(configuration::addAnnotatedClass);

    return configuration.buildSessionFactory();
  }
}
```

I upotreba definisane zavisnosti u komponenti:

```
@Grain
public class UserRepository extends BaseRepository<User> {
  public MovieRepository(SessionFactory sessionFactory) {
    super(sessionFactory, User.class);
  }
  // implementation ...
}
```

U ovom slučaju `SessionFactory` je umetnut kroz konstruktor klase `MovieRepository`.

Umetanje zavisnosti nije samo mehanizam koji radni okvir pruža programeru, već i mehanizam koji je i u srži samog radnog okvira. Navešćemo primer autentikacionog servisa:

```
@Grain
public class FormLoginAuthenticationEntryPoint implements AuthenticationEntryPoint {
  @Inject
  private UserService userService;
  @Inject
```

```
  private PasswordEncoder passwordEncoder;
  @Inject
  private SessionStore sessionStore;

  @Override
  public Authentication authenticate(HttpRequest request, HttpResponse response) throws
  GrainSecurityException {
  // implementation ...
  }
}
```

Ovde vidimo da `FormLoginAuthenticationEntryPoint` ima par definisanih zavisnosti koje su umetnute od strane radnog okvira. Ovo je podrazumevana implementacija `AuthenticationEntryPoint` interfejsa i može biti promenjena od strane korisnika definisanjem komponente koja implementira isti. Sistem za umetanje zavisnosti će prioritizirati komponente definisane van paketa radnog okvira. Ovo je dobar uvod u naše sledeće poglavlje koje se bavi autentikacijom i autorizacijom.

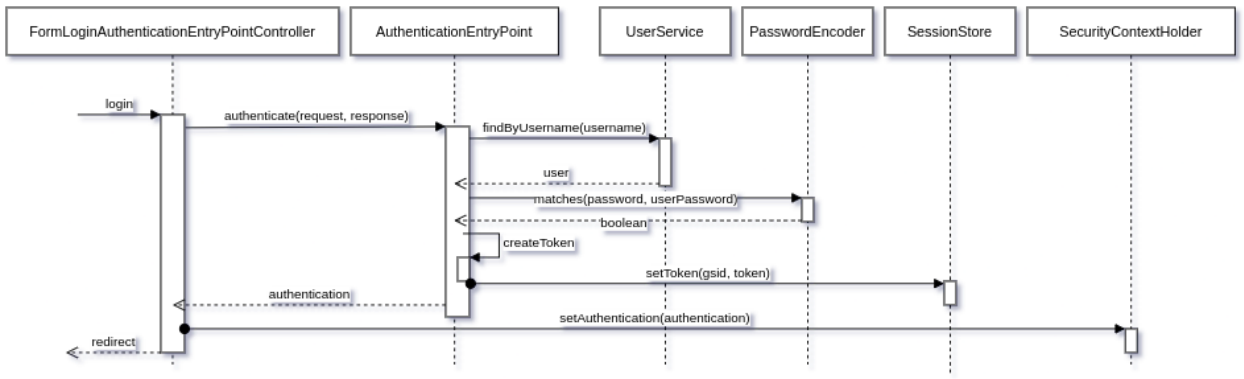
Autentikacija i autorizacija

Jedna od često važnih funkcionalnosti koje radni okvir treba da pruža su mehanizmi za autentikaciju i autorizaciju. Kao što smo već govorili autentikacija je proces identifikacije korisnika na sistemu a autorizacija proces određivanja njegovih privilegija odnosno radnji koje može da uradi.

U *Grain* radnom okviru autentikacija i autorizacija (na dalje ćemo ih zvati objedinjenim terminom - *security*) su implementirani koristeći sesiju identifikovanu kolačićem korisnika.

Autentikacija

U *Grain* radnom okviru autentikacija je implementirana kroz `AuthenticationEntryPoint` interfejs. Ovaj interfejs ima jednu metodu `authenticate` koja prima `HttpRequest` i `HttpResponse` objekte i vraća `Authentication` objekat. Podrazumevana implementacija je `FormLoginAuthenticationEntryPoint`, i kao što joj ime kaže, ona prima `POST` zahtev iz HTML forme koji koristi za autentikaciju. Ovaj interfejs je korisniku izložen kroz `FormLoginAuthenticationEntryPointController` koji sadrži `/login` endpoint.



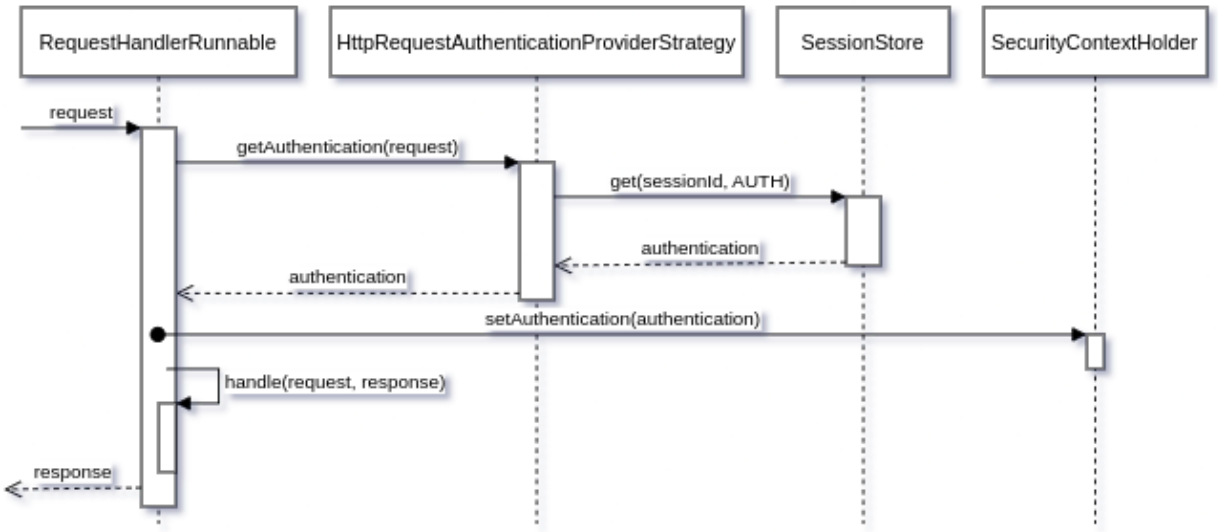
Sekvencijalni dijagram autentikacije

Na dijagramu je prikazan *"happy path"* autentikacionog procesa. Kontroler prihvata zahtev i prosleđuje parsirane `HttpRequest` i `HttpResponse` objekte ka `AutnenticationEntryPoint` objektu. On dalje obrađuje zahtev tako što izvlači kredencijale korisnika i proverava ih koristeći `PasswordEncoder` i `UserService`. Između ostalog, pored provere kredencijala postoji provere: da li je korisnički nalog isključen, da li su mu istekli

kredencijali itd. Nakon uspešne provere validnosti autentikacije kreira se sesija u `SessionStore` objektu i autentikacioni objekat se vraća nazad do `AuthenticationEntryPoint`-a gde se čuva u `SecurityContextHolder`-u.

Autorizacija

Proces autorizacije je malo jednostavniji od autentikacije. Prilikom svakog zahteva proverava se sesija i na osnovu nje se ubacuje sačuvani autentikacioni objekat u `SecurityContextHolder`. Za ovo je zadužen `HttpRequestAuthenticationProviderStrategy`, koji naravno ima podrazumevanu implementaciju koja to radi na osnovu kolačića.



Sekvencijalni dijagram autorizacije

Kao što smo pomenuli, autentikacija koja se koristi za proveru privilegija korisnika se čuva u `SecurityContextHolder` objektu. Sada ćemo objasniti šta to znači. Identifikovati korisnika na sistemu je jedan proces dok je autorizacija nešto sasvim drugo. Osnovna razlika u tome je što je autorizacije proces koji duže traje. Autorizacija korisnika na sistemu mora da bude aktivna i sve dok traje trenutni zahtev koji se obrađuje i nikako ni jedan drugi zahtev ne sme da utiče na njenu validnost. Ovaj veliki problem je rešen relativno lako po uzoru na Spring Security biblioteku. Pomenuli smo ranije da je svaki zahtev u sistemu obrađen na odvojenoj niti da bi se omogućilo obrađivanje puno zahteva odjednom. Ovo nam trenutno umnogome olakšava posao. Sve što je potrebno uraditi na početku svakog zahteva je sačuvati autentikacioni objekat u `ThreadLocal` promenljivu. `ThreadLocal` je java mehanizam koji vrednosti sačuvane u njemu čuva na nivou niti koja se trenutno izvršava. Ovo u prevodu znači da će svaki autorizovani zahtev imati jedinstveno mesto čuvanja autentikacionog objekta i neće dolaziti do potencijalnih problema sinhronizacije niti i potencijalnih bagova vezanih za iste.

```
class ThreadLocalSecurityContextHolderStrategy implements SecurityContextHolderStrategy {
  private final ThreadLocal<SecurityContext> securityContext = new ThreadLocal<>();

  @Override
  public SecurityContext getContext() {
    return securityContext.get();
  }

  @Override
  public void setContext(SecurityContext securityContext) {
```

```
    if (securityContext == null)
      throw new GrainSecurityNullContextException();
    this.securityContext.set(securityContext);
  }

  @Override
  public SecurityContext createDefaultContext() {
    return new SecurityContextImpl();
  }
}
```

Na ovaj način nam je veoma jednostavno "izvući" i proveriti autentikaciju na bilo kom mestu u aplikaciji.

Provera autorizacije

Svaki zahtev u Grain aplikaciji prolazi kroz neku od `AbstractRequestHandlerProxy` implementacija. Ako *security* nije osposobljen RequestHandlerProxy neće uraditi ništa već će samo proslediti zahtev do odgovarajućeg handler-a.

U slučaju da je *security* konfigurisan dolazi do provere autorizacionih pravila na osnovu kojih se određuje da li korsnik može da pristupi određenom endpoint-u.

```
public class SecurityHandlerProxy extends AbstractRequestHandlerProxy {
  private final SecurityConfiguration securityConfiguration;

  public SecurityHandlerProxy(RequestHandler target, SecurityConfiguration
  securityConfiguration) {
    super(target);
    this.securityConfiguration = securityConfiguration;
  }

  @Override
  public void handle(HttpRequest request, HttpResponse response) {
    boolean result = new
    RuleUrlPatternMatcher(securityConfiguration.getRules()).matches(request);
    if (result) {
      try {
        target.handle(request, response);
      } catch (IOException e) {
        throw new GrainRuntimeException(e);
      }
    } else {
      throw new HttpException.Forbidden(HttpStatus.FORBIDDEN.getReason());
    }
  }
}
```

Kao što vidimo vrši se provera zahteva u odnosu na pravila definisana u *Security* konfiguraciji. Ako trenutna autentikacija ne odgovara zahtevima definisanim u pravilima korisniku se vraća `403` greška. Naravno ako za zahtev ne postoje definisana pravila on se odvija bez ikakvih problema. Primer konfiguracije pravila bi izgledao ovako:

```
@Grain
public class SecurityConfig implements SecurityConfigurer {
```

```
  @Override
  public void configure(SecurityConfigurationBuilder sec) {
    sec.withRules()
        .urlPattern("/user/**").authenticated().and()
        .urlPattern("/admin/**").authenticated().roles("ADMIN").and()
        .urlPattern("/posts").unauthenticated()
        .buildRules();
  }
}
```

`SecurityConfigurationBuilder` koristi *Fluent* interfejs šablon koji nam omogućava da definišemo pravila na jednostavan i čitljiv način način. U ovom primeru smo definisali da korisnik mora da bude autentifikovan da bi mogao da pristupi `/user` i `/admin` endpoint-ima. Takođe smo definisali da korisnik mora da ima ulogu (rolu) `ADMIN` da bi mogao da pristupi `/admin` endpoint-u. Na kraju smo definisali da korisnik ne mora da bude autentifikovan da bi mogao da pristupi `/posts` endpoint-u, tj. da je to javni endpoint.

Templating jezik

Sledeći koncept koji ćemo objasniti je dizajn templating jezika. Templating jezik koristimo za dinamičko generisanje HTML stranica. Ovaj templating jezik ima sličnu sintaksu kao **JSP**.

```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo">Grain</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <% if (authentication != null) { %>
        <li><a href="/dashboard"><%= print(authentication.getName()) %></a></li>
        <li><a href="/logout">Logout</a></li>
      <% } else { %>
        <li><a href="/login">Login</a></li>
      <% } %>
    </ul>
  </div>
</nav>
```

U kodu iznad vidimo primer dinamički generisane navigacije za neku od stranica. Korisniku se prikazuju različiti linkovi u zavisnosti od toga da li je ulogovan ili ne.

Pre nego što možemo da objasnimo dati kod počećemo sa jednostavnijim primerima. Najpre ćemo definisati šta je programski jezik.

Programski jezik

Programski jezik možemo podeliti u sve komponente: sintaksa(forma) i semantika(značenje). Sintaksa je skup pravila po kojima se programski jezik sastoji. Semantika je skup pravila po kojima se programski jezik interpretira. Ove dve komponente su definisane specifikacijom po kojoj su kompajleri i interpretatori implementirani.

Kada smo već pomenuli kompajlere i interpretatore naglasićemo da se jezici dele u dve grupe na osnovu toga da li su interpretirani ili kompajlirani. Kompajlirani jezici se prevode iz jezika visokog nivoa u mašinski jezik koji se izvršava direktno na mašini na kojoj se pokreću. Interpretirani jezici se kao što im ime kaže interpretiraju, odnosno različite semantičke konstrukcije pozivaju metode interpretatora koji kasnije izvršava kod na mašini.

Naravno jezici se mogu deliti na statički i dinamički tipizirane. Ova podela nastaje usled prisustva ili odsustva striktnih tipova kada su u pitanju promenljive. Ovaj koncept je semantički i implementiran je direktno u

kompajleru ili interperteru.

Po svom tipu jezike najčešće delimo na: funkcionalne, objektno-orijentisane, itd. Trenutno najpopularnija paradigma programskih jezika je objektno orijentisana paradigma.

Dakle možemo zaključiti da se programski jezik sastoji iz skupa pravila i programa koji je dizajniran da po tom skupu pravila interpretira ili kompajlira tekst odnosno kod.

Templating programski jezik

Osobine koje templating programski jezik treba da ima jesu one koje mu omogućavaju da bude fleksibilan za korišćenje i jednostavan za interpretiranje. Takođe je od veoma velike pomoći da sintaksa jezika nije u preterano velikom konfliktu sa HTML kodom koji će se neminovno naći u datotekama gde će se pisati taj jezik. Najbolji primer ovoga je verovatno **Thymeleaf** koji se nesmetano piše kao deo HTML-a i potpuno je neprimetan. Kontrast toga je naravno **JSP** koji se trudi da bude što je više moguće nalik Javi. Naravno mnogo je jednostavnije implementirati interpretator koji interpretira JSP nego Thymeleaf tako da smo se mi odlučili za jezik koji ima sintaksu nalik na JSP.

GTL

Dakle, jezik treba da ima prostu sintaksu, da bude podržava više paradigmi, da bude dinamički i da bude interpretiran. Naravno dizajn i implementacija programskih jezika je veoma kompleksan i unosan proces. Kreiranje specifikacije, implementacija po specifikaciji itd. Naš pristup u Grain radnom okviru je bio više *ad hoc*. Nismo imali konkretnu specifikaciju već smo radili na parseru i interpretatoru sve dok nismo ispunili neke osnovne ciljeve koji su nam bili potrebni za generisanje stranica. Naravno kasnije su dodavane određene funkcionalnosti da bi jezik bi lakši za korišćenje ali svakako nije bilo nekog konkretnog plana. Zbog toga za predstavljanje jezika koristićemo konkretne primere iz postojećih aplikacija i potencijalno jedinične testove koji su korišćeni u razvoju GTL-a (*Grain Templating Language* - u nedostatku boljeg imena).

```
  @Test
void test_testInterpreterReturn() {
  String code = "2 + 3";
  Interpreter interpreter = new Interpreter(code, null);
  Object retval = interpreter.run();
  assertEquals(5, retval);
}
```

U ovom primeru vidimo interpretiranje prostog izraza `2 + 3`. Ovaj izraz se sastoji iz para AST čvorova celobrojnih literala i jedne binarne operacije sabiranja koji bi u prefiksnoj (poljskoj) notaciji izgledao ovako:

```
+ (2, 3)
```

Kod ovog binarnog čvora imamo levo se nalazi čvor koji predstavlja literala sa vrednošću `2` a desno čvor literala sa vrednošću `3`. Rezultat ovog binarnog čvora sabiranja dobijamo tako što primenimo operaciju sabiranja nad levim i desnim čvorom.

Sledi malo kompleksniji primer koji ilustruje korišćenje binarnih operacija i promenljivih. U ovom primeru vidimo da je rezultat binarne operacije `*` jednak `30` što je rezultat sabiranja `1 + 2` i množenja sa `10`. Ovde vidimo i prioritiziranje operacija usled korišćenja zagrada.

```
@Test
void test_plusOperationWithParenthesis() {
  String code = "a = (1 + 2) * 10";
  Interpreter interpreter = new Interpreter(code, null);
  interpreter.run();
  Object val = interpreter.getSymbolValue("a");
  assertEquals(30, val);
}
```

U poljskoj notaciji ovaj izraz izgleda ovako:

```
= (a, * (+ (1, 2), 10))
```

Sledeći primer ilustruje korišćenje `if` naredbe. U ovom primeru vidimo da je rezultat `if` naredbe jednak `10` jer je uspešno izvršena grana gde se broj `10` dodeljuje promenljivoj `a`:

```
@Test
void test_ifStatement() {
  String code = "if (1 + 2 == 3) { a = 10 } else { a = 20 }";
  Interpreter interpreter = new Interpreter(code, null);
  interpreter.run();
  Object val = interpreter.getSymbolValue("a");
  assertEquals(10, val);
}
```

U poljskoj notaciji ovaj izraz izgleda ovako i ovo je jedan od primera gde AST čvor ima 3 potomka:

```
if (== (+ (1, 2), 3), = (a, 10), = (a, 20))
```

Kod programa se pre parsiranja u AST stablo leksira u tokene. Poslednji primer kada se prevede u leksičke tokene izgelda ovako:

```
IF LPAREN LIT_INT PLUS LIT_INT EQ LIT_INT RPAREN LBRACE
    IDEN ASSN LIT_INT
RBRACE ELSE LBRACE
    IDEN ASSN LIT_INT
RBRACE
```

Lekser je implementiran kao *Finite state machine*. Svi leksički tokeni su definisani kao skup pravila za parsiranje svakog od njih. Lekser pravi odluke na osnovu trenutnog karaktera na kome se nalazi i parsira naredne karaktere na osnovu pravila koje određeno tim karakterom. Na primer, ako lekser pri početku parsiranja naiđe na karakter `"` ili `'` u pitanju je string literal i lekser sve naredne karaktere do dolaska do sledećeg karaktera `"` ili `'` parsira kao string literal. Ako je na primer sledeći karakter `=` u zavisnosti od narednog karaktera token može biti ili dodeljivanje (eng. *assignment*) ako naredni karakter nije `=`. Ukoliko jeste token se parsira kao jednakost (eng. *equality*). Leksička analiza primenjuje samo osnovna pravila i ne služi za proveravanje sintaksnih grešaka. Na primer lekser ne ume da prepozna grešku pisanje dva celobrojna literala jedan za drugim ali ume da prepozna ne zatvaranje string literala. Lekser je implementiran kao jedna `while` petlja koja prolazi kroz sve karaktere programa.

```
private void doLex() {
  while (true) {
    // preskačemo sve whitespace karaktere
    eatWhitespace();

    // ako smo došli do kraja programa prekidamo leksičku analizu
    if (!hasNext())
      break;

    // parsiramo sledeći karakter
    start = getCharacter();

    // ako je karakter validni početak identifikatora
    // parsiramo ga kao identifikator
    if (isStartOfIdentifier()) {
      String val = eatIdentifier();
      TokenType kwOrIden = classifyToken(val);
      Token token = createToken(kwOrIden, val);
```

```
      emit(token);
      continue;
    }

    // ako je karakter broj ili znak "-" parsiramo celobrojni literal
    if (isStartOfNumberLiteral()) {
      String val = eatFloat();
      TokenType type = getNumberType(val);
      if (type.equals(INVALID)) {
        printSourceCodeLocation();
        throw new LexerException("Invalid number literal " + getInfo());
      }
      Token token = createToken(type, val);
      emit(token);
      continue;
    }

    // ako je karakter neki od validnih navodnika parsiramo ga
    // kao string literal
    if (isStartOfStringLiteral()) {
      String val = eatStringLiteral();
      if (val == null) {
        printSourceCodeLocation();
        throw new LexerException("Invalid string literal " + getInfo());
      }
      Token token = new Token(LIT_STR, val);
      emit(token);
      continue;
    }

    // U svakom drugom slučaju parsiramo karakter kao operator
    // ako ne uspemo da ga prepoznamo kao validan operator
    // prijavljujemo grešku
    Optional<Token> operator = tryParseOperator();
    // ...
```

Parsiranje operatora se je prikazano sledećim kodom:

```
private Optional<Token> tryParseOperator() {
  String curr = next(); // trenutni karakter

  // U zavisnosti od trenutnog i sledećeg karaktera
  // odlučujemo o kom operatoru se radi
  switch (curr) {
    case ":":
      return Token.optional(TERNELSE, curr);
    case "?":
      if (peek().equals("?")) {
        curr += next();
        return Token.optional(DFLT, curr);
      } else {
        return Token.optional(TERNCOND, curr);
      }
    case "!":
```

```
      if (peek().equals("=")) {
        curr += next();
        return Token.optional(NE, curr);
      } else {
        return Token.optional(NOT, curr);
      }
    case "=":
      if (peek().equals("=")) {
        curr += next();
        return Token.optional(EQ, curr);
      } else {
        return Token.optional(ASSN, curr);
      }
  // ostali operatori
  // ...
```

Kada smo imamo listu svih tokena koji čine program možemo da ih parsiramo u AST stablo. Parser je implementiran kao *recursive descent parser*. On funkcioniše slično kao i lekser - u informacije o trenutnom i narednom tokenu odlučuje o kom čvoru se radi. Parsiranje se sastoji od parsiranje dva tipa čvorova: parsiranje izraza (eng. *expression*) i iskaza (eng. *statement*). Iskazi se sastoje iz jednog ili više izraza i mogu da budu na primer: `if`, `for`, `foreach` itd. Izrazi mogu da budu na primer: izrazi dodeljivanja (a = 5), jednakosti (b == 5) itd.

Metoda parsiranja izraza je prikazana sledećim kodom:

```
private AstNode parseStatement() {
  AstNode node = null;
  if (iter.isPeekOfType(LBRACE)) {
    node = parseBlockStatement();
  } else if (iter.isPeekOfType(IF)) {
    node = parseIfStatement();
  } else if (iter.isPeekOfType(FOREACH)) {
    node = parseForEachStatement();
  } else if (iter.isPeekOfType(FOR)) {
    node = parseForStatement();
  } else {
    node = parseExpression();
  }
  return node;
}
```

Ukoliko naredni token nije ni jedan od navedenih parsiramo iskaz kao izraz. Parsiranje izraza je prikazano sledećim kodom:

```
private AstNode parseExpression() {
  Token curr = iter.next();
  AstNode node = null;

  if (iter.isPeekOfType(AND, OR)) {
    node = createNode(curr);
    node = parseBooleanNode(iter.next(), node);
  } else if (iter.isPeekOfType(EQ, NE)) {
    node = createNode(curr);
    node = parseEqualityNode(iter.next(), node);
  } else if (iter.isPeekOfType(GT, LT, GE, LE)) {
```

```
    node = createNode(curr);
    node = parseRelationalNode(iter.next(), node);
  } else if (iter.isPeekOfType(ADD, SUB, DIV, DIV, MUL)) {
    node = createNode(curr);
    node = parseArithmeticNode(iter.next(), node);
  } else if (iter.isPeekOfType(TERNCOND)) {
    node = createNode(curr);
  } else if (iter.isPeekOfType(ASSN)) {
    if (!curr.isOfType(IDEN))
      throw new ParserSyntaxErrorException(getSourceCodeLocation(curr),
          "Cannot assign to '%s'", curr.getType());
  // ostali tipovi izraza
  // ...
  // i na kraju parsiramo preostali token kao običan čvor
  // koji ne predstavlja izraz
  } else {
    node = createNode(curr);
  }
```

Ovde vidimo da se na osnovu trenutnog i sledećeg tokena pravi odluka o tome kako će se parsirati naredni čvor AST stabla. Prilikom parsiranja svakog od čvorova rekurzivno se zovu navedene metode do trenutka kada nemamo više tokena.

Primer parsiranja jednog izraza dodeljivanja je predstavljen sledećim kodom:

```
// token je trenutni token a left predstavlja prethodni parsirani čvor koji će
// postati levi potomak čvora dodeljivanja
private AstNode parseAssignmentNode(Token token, AstNode left) {

  AstAssignmentNode astAssignmentNode = (AstAssignmentNode) createNode(token);
  // setSymbol je delegat metoda metode setLeft
  astAssignmentNode.setSymbol(left);

  // rekurzivni poziv parseExpression
  AstNode value = parseExpression();

  // parsirani izraz dodeljujemo kao vrednost čvora odnosno
  // desnom potomku čvora dodeljivanja
  astAssignmentNode.setValue(value);
  return astAssignmentNode;
}
```

Što se tiče sintaksnih grešaka. Navešćemo primer sintaksne greške koje nastaje kada zaboravimo levu zagradu kod parsiranja `if` iskaza:

```
private AstNode parseIfStatement() {
  Token ifToken = iter.next();
  // ukolikio sledeći token posle if tokena nije (
  // bacamo sintaksnu grešku
  if (!iter.isPeekOfType(LPAREN)) {
    throw new ParserSyntaxErrorException(getSourceCodeLocation(iter.peek()), "Expected '%s'",
    LPAREN.getValue());
  }
```

Uspešnim parsiranjem svih tokena dobijamo AST koje može biti interpretirano od strane *interpretera*.

Interpreter

Interpretator je u našem slučaju klasa koja izvršava interpretaciju AST stabla. Interpretator je implementiran kao *visitor* koji obilazi AST stablo i izvršava odgovarajuće akcije. Interpretator kao *visitor* poziva `run` metode AST čvorova i evaluira njihove vrednosti.

Primer `run` metode `AstBooleanNode`-a:

```
@Override
public Object run(Interpreter interpreter) {
  // imperativ je evaluirati levog potomka
  // kada je u pitanju desni primenićemo tehniku
  // short-circuit evaluation koja će preskočiti evaluaciju
  // levog potomka ukoliko je vrednost desnog potomka
  // dovoljna da se dobije rezultat bulove operacije
  Object leftValue = left.run(interpreter);
  switch (operator) {
    case AND:
      // u slučaju da je levi potomak false
      // desni potomak neće uticati na rezulatat AND operacije
      if (!Boolean.parseBoolean(String.valueOf(leftValue)))
        return false;
      return Boolean.parseBoolean(String.valueOf(right.run(interpreter)));
    case OR:
      // u slučaju da je levi potomak true
      // desni potomak neće uticati na rezulatat OR operacije
      if (Boolean.parseBoolean(String.valueOf(leftValue)))
        return true;
      return Boolean.parseBoolean(String.valueOf(right.run(interpreter)));
  }

  // u slučaju da je operator neki drugi
  // bacamo izuzetak
  throw new IllegalStateException("Unknown operator value " + operator);
}
```

Evaluiranjem svih čvorova dobijamo vrednost koja se može koristiti u daljem toku izvršavanja programa. Interpretator u toku izvršavanja čuva informaciju o trenutnom *scope*-u izvršavanja. Kod izvršavanja `if` čvora možemo videti kako se *scope pop*-uje i *push*-uje na *stack*:

```
@Override
public Object run(Interpreter interpreter) {
  Object value = null;
  // Evaluiramo uslov
  Object conditionValue = this.condition.run(interpreter);

  // ukoliko je uslov true
  // izvršavamo blok koda
  // i pritom kreiramo novi scope
  // kada se blok završi
  // scope se popuje sa stack-a
  if (!isFalsy(conditionValue) && this.getIfTrue() ≠ null){
    interpreter.pushScope();
    value = this.getIfTrue().run(interpreter);
    interpreter.popScope();
  } else if (getIfFalse() ≠ null) {
```

```
    interpreter.pushScope();
    value = this.getIfFalse().run(interpreter);
    interpreter.popScope();
  }

  // na kraju moramo ali i ne moramo vratiti vrednost
  return value;
}
```

Scope-ovi su implementirani kao *dek* mapa stringova (`Deque<Map<String, Object>>`) na objekte.

```
public class Interpreter {
  // ...
  public void pushScope() {
    this.scopeStack.push(new HashMap<>());
  }

  public void popScope() {
    this.scopeStack.pop();
  }

  // dobijanje vrednosti simbola (identifikatora)
  public Object getSymbolValue(String name) {
    Object o = null;
    Map<String, Object> scope = getScopeThatContains(name);

    if (scope.containsKey(name)) {
      o = scope.get(name);
    } else {
      // ukoliko ne postoji simbol sa zadatim imenom
      // pokušavamo da učitamo klasu sa tim imenom
      Optional<Class<?>> clazz = tryLoadClass("java.lang." + name.replaceAll("java.lang.", ""));
      if (clazz.isPresent()) {
        o = clazz.get();
        // čuvamo klasu da ne bi morali ponovo da pozivamo loadClass
        putScopedSymbol(name, o);
      }
    }

    return o;
  }

}
```

Sada primenjena znanja možemo da iskoristimo za implementaciju jednostavnog *templating* jezika.

Templating

Objasnićemo korišćenje jezika na osnovu jednog od prethodnih primera:

```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo">Grain</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <% if (authentication ≠ null) { %>
```

```
        <li><a href="/dashboard"><%= print(authentication.getName()) %></a></li>
        <li><a href="/logout">Logout</a></li>
      <% } else { %>
        <li><a href="/login">Login</a></li>
      <% } %>
    </ul>
  </div>
</nav>
```

Poenta templating jezika je da od ovog HTML *snippet*-a napravi dinamički popunjen HTML fajl. Interpretator je ovde zadužen da delove template-a pretvori u pozive `print` ugrađene metode koja vraća deo template-a bez koda kao string.

```
// print built-in metoda
(AstFunctionCallback) (args) -> {
  String value = (args[0] == null ? "null" : args[0].toString());
  // ovde radimo escape-ing navodnika
  value = value.replaceAll("\\\\'", "'").replaceAll("\\\\\"", "\"");
  // upisujemo u izlazni tok
  write(value);
  return value;
}
```

Prvi korak interpretera je da razdvoji template od blokova koda:

```
print("<nav>\n\t<div class=\"nav-wrapper\">\n\t\t<a href=\"#\"
class=\"brand-logo\">Grain</a>\n\t\t<ul id=\"nav-mobile\" class=\"right
hide-on-med-and-down\">\n")
if (authentication ≠ null) {
  print("\t\t\t<li><a href=\"/dashboard\">")
  print(authentication.getName())
  print("</a></li>\n")
  print("\t\t\t\<li><a href=\"/logout\">Logout</a></li>\n")
} else {
  print("\t\t\t<li><a href=\"/login\">Login</a></li>\n")
}
print("</ul>\n</div>\n</nav>\n")
```

Ovo bi bio rezultat inicijalnog parsiranja template-a. Na kraju imamo kod koji bez problema možemo parsirati interpretatorom. Ukoliko kod sadrži interpolaciju `${val}` ona je evaluirana pre konverzije template-a i menja interpolaciju pozivima `print` metode (npr. `<% print(val) %>`).

Rezultujući template posle evaluacije u slučaju da je korisnik ulogovan izgleda ovako:

```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo">Grain</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li><a href="/dashboard">admin</a></li>
      <li><a href="/logout">Logout</a></li>
    </ul>
  </div>
</nav>
```

Primer gotove aplikacije

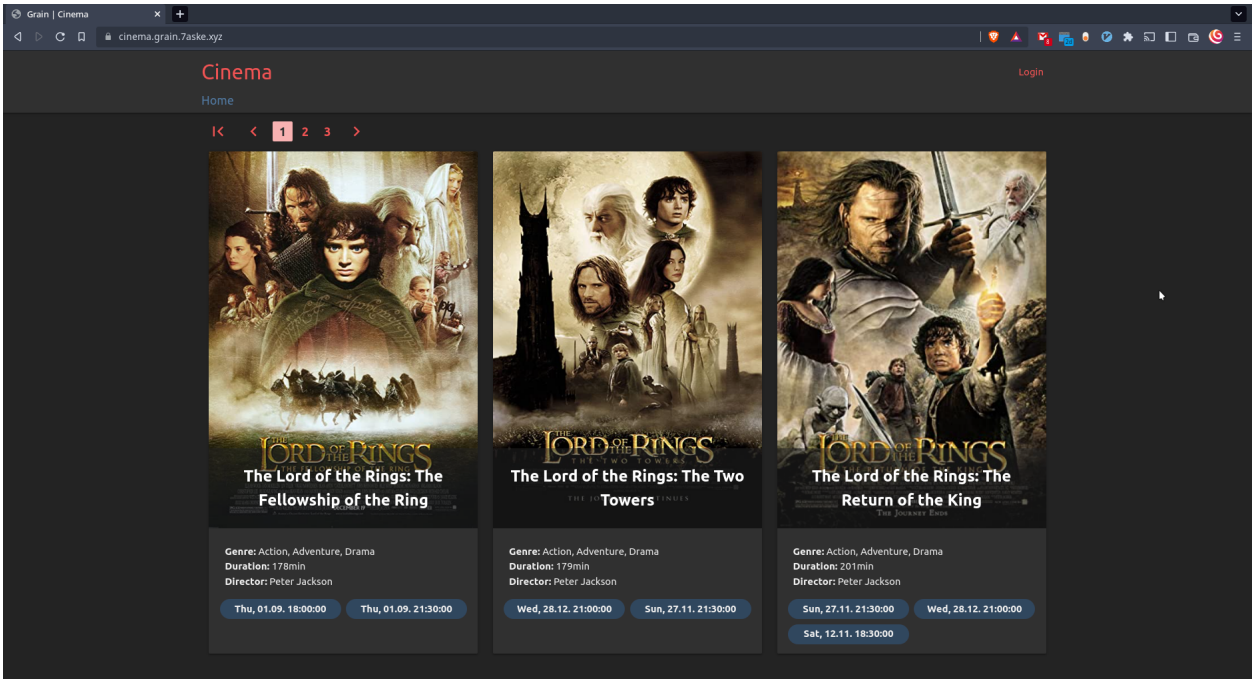
Primer aplikacije pisane u Grain radnom okviru biće aplikacija za rezervaciju mesta u bioskopu. Aplikacije će omogućiti bioskopima da na svom sajtu imaju interfejs za korisnike preko koga će moći da rezervišu mesta u salama za projekcije. Aplikacija će imati sledeće funkcionalnosti:

- Registracija korisnika
- Prijavljivanje korisnika
- Pregled filmova
- Pregled projekcija
- Rezervacija mesta
- Administracija

Aplikacija će biti dostupna na sledećem URL-u (ukoliko nije ugašen server): https://cinema.grain.7aske.xyz

Početna strana

Na početnoj strani će biti prikazani svi trenutno aktivni filmovi zajedno sa svojim projekcijama.



Početna strana

Kao što vidimo korisnik ima listu filmova i ispod njih listu projekcija.

Implementacija

```
@Controller
@RequestMapping
@RequiredArgsConstructor
public class IndexController {
  private final UserService userService;
```

```
  private final MovieService movieService;

  @GetMapping
  public View getIndex(@RequestParam(value = "page", defaultValue = "0,3") Pageable page) {
    TemplateView dataView = new TemplateView("index.gtl");
    dataView.addAttribute("movies", movieService.findAll(page));
    return dataView;
  }

  @GetMapping("/login")
  public View getLogin() {
    return new TemplateView("login.gtl");
  }

  @GetMapping("/register")
  public View getRegister() {
    return new TemplateView("register.gtl");
  }

  @PostMapping("/register")
  public String postRegister(@FormBody RegisterUserDto user) {
    userService.register(user);
    return "redirect:/login?registered";
  }
}
```

Na primeru vidimo implementaciju kontrolera koji pored prikazivanja početne strane ima i endpointe za registraciju i login koji "renderuju" odgovarajuće stranice.

Kontroler u sebi ima umetnute dve zavisnosti: `UserService` za registraciju i `MovieService` za dobavljanje filmova za prikaz na početnoj strani. Takođe, endpoint početne strane ima jedan parametar koji služi za paginaciju.

`MovieService` je jednostavan servis koji implementira **CRUD** operacije nad filmovima:

```
@Grain
@RequiredArgsConstructor
public class MovieServiceImpl implements MovieService {
  private final MovieRepository movieRepository;

  @Override
  public Collection<Movie> findAll() {
    return findAll(null);
  }

  @Override
  public Collection<Movie> findAll(Pageable pageable) {
    return movieRepository.findAll(pageable);
  }

  @Override
  public Movie findById(Long id) {
    return movieRepository.findById(id);
  }
  // ... ostale metode
```

```
}
```

`MovieService` sadrži jednu zavisnost iz *data access* lejera: `MovieRepository`.

```
@Grain
public class MovieRepository extends BaseRepository<Movie> {
  public MovieRepository(SessionFactory sessionFactory) {
    super(sessionFactory, Movie.class);
  }

  public Collection<Movie> search(String search, Pageable pageable) {
    EntityManager entityManager = getEntityManager();
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Movie> cq = cb.createQuery(Movie.class);
    Root<Movie> root = cq.from(Movie.class);
    CriteriaQuery<Movie> where = cq.select(root)
        .where(cb.or(
            cb.like(root.get("title"), wrapLike(search)),
            cb.like(root.get("description"), wrapLike(search)),
            cb.like(root.get("genre"), wrapLike(search)),
            cb.like(root.get("director"), wrapLike(search))
        ));
    return entityManager.createQuery(where)
        .setFirstResult(pageable.getPageOffset())
        .setMaxResults(pageable.getPageSize())
        .getResultList();
  }

  private String wrapLike(String str) {
    return "%" + str + "%";
  }
}
```

U ovoj klasi se nalazi specifična implementacija `BaseRepostory<T>` klase koja je deo integracije za Hibernate bibliotekom. Jedina metoda koja nije klasična **CRUD** operacija jeste `search` metoda koju vidimo u ovoj klasi. Ostale metode se pozivaju direktno iz nad-klase. U klasi `BaseRepository<T>` vidimo generičku implementaciju CRUD operacija za bilo koju entitetsku klasu. Entitetske klase su klase koje manipuliše Hibernate i one su anotirane `@Entity` anotacijom.

```
@RequiredArgsConstructor
public abstract class BaseRepository<E> {
  private final SessionFactory sessionFactory;
  private final Class<E> entityClass;

  protected synchronized EntityManager getEntityManager() {
    return sessionFactory.createEntityManager();
  }

  public List<E> findAll(@Nullable Pageable pageable) {
    EntityManager entityManager = getEntityManager();
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<E> cq = cb.createQuery(entityClass);
    Root<E> root = cq.from(entityClass);
    CriteriaQuery<E> where = cq.select(root);
```

```
    Query query = entityManager.createQuery(where);
    if (pageable ≠ null) {
      query.setFirstResult(pageable.getPageOffset());
      query.setMaxResults(pageable.getPageSize());
    }
    return query.getResultList();
  }

  public E findById(@NotNull Long id) {
    EntityManager entityManager = getEntityManager();
    return entityManager.find(entityClass, id);
  }

  public E save(@NotNull E entity) {
    EntityManager entityManager = getEntityManager();
    entityManager.getTransaction().begin();
    entityManager.persist(entity);
    entityManager.getTransaction().commit();
    return entity;
  }

  public E update(@NotNull E entity) {
    EntityManager entityManager = getEntityManager();
    entityManager.getTransaction().begin();
    entityManager.merge(entity);
    entityManager.getTransaction().commit();
    return entity;
  }

  public void delete(@NotNull E entity) {
    EntityManager entityManager = getEntityManager();
    entityManager.getTransaction().begin();
    entityManager.remove(entity);
    entityManager.getTransaction().commit();
  }
}
```

`SessionFactory` je komponenta takođe definisana kao deo Hibernate integracije.

```
@Grain
public class HibernateConfiguration {

  @Grain
  public SessionFactory sessionFactory() {
    org.hibernate.cfg.Configuration configuration = new org.hibernate.cfg.Configuration();

    GrainClassLoader grainClassLoader = new
GrainJarClassLoader(CinemaApp.class.getPackageName());
    grainClassLoader.loadClasses(cl → cl.isAnnotationPresent(Entity.class))
        .forEach(configuration::addAnnotatedClass);

    return configuration.buildSessionFactory();
  }
}
```

```
# application.properties

application.name=Grain Cinema

security.enabled=true

hibernate.dialect=org.hibernate.dialect.MariaDBDialect
hibernate.connection.driver_class=org.mariadb.jdbc.Driver
hibernate.connection.url=jdbc:mariadb://localhost:3306/cinema
hibernate.connection.username=root
hibernate.connection.password=toor
hibernate.hbm2ddl.auto=update
hibernate.show_sql=true
```

Ovde vidimo kako na veoma jednostavan način možemo iskoristiti dva mehanizma koje Grain pruža da konfigurišemo Hibernate. Prvi je da je sva konfiguracija pisana u `application.properties` fajlu automatski deo environment-a iz koga Hibernate takođe čita podatke. Drugi je da možemo da iskoristimo `GrainClassLoader` da dinamički pri pokretanju učitamo i konfigurišemo sve entitetske klase. Ovime imamo veoma elegantnu i minimalističnu Hibernate integraciju.

Template ove strane je `index.gtl` i relativno je jednostavan. U njemu koristimo GTL fragmente koji nam služe kao *reusable* komponente za prikaz kartica sa filmovima:

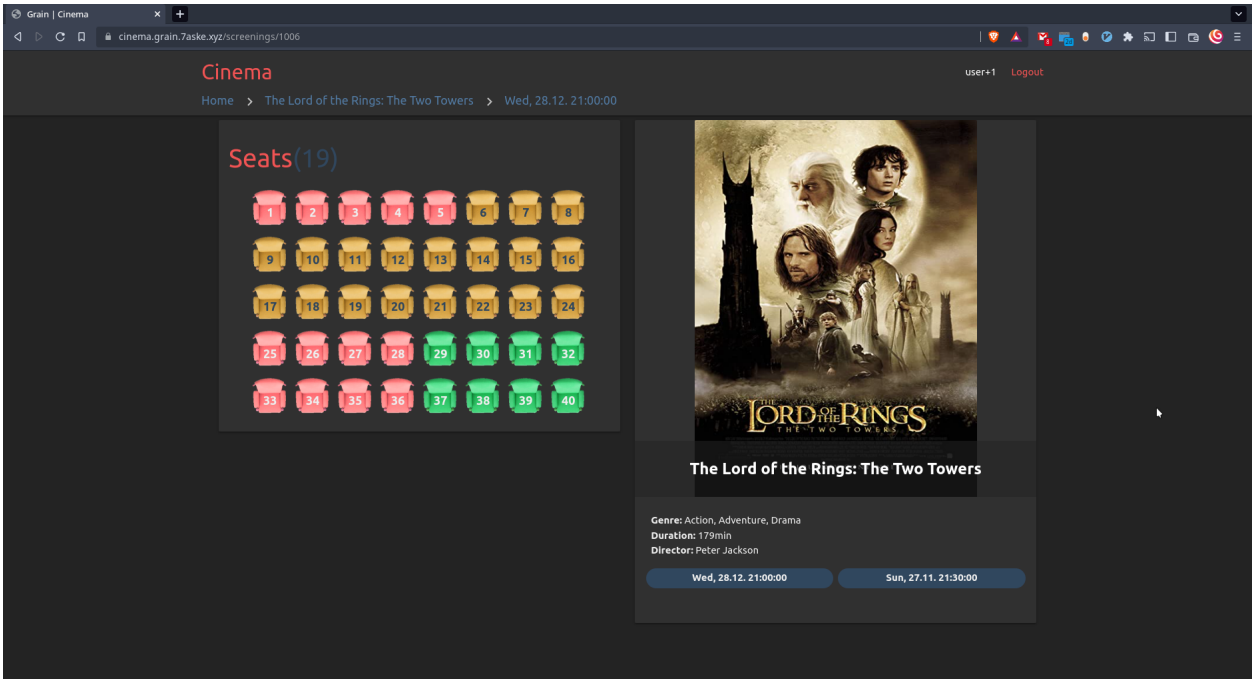
```
<% include "fragments/import.gtl" as Imports; %>
<% include "fragments/nav.gtl" as Nav; %>
<% include "fragments/util/pagination.gtl" as Pagination; %>
<% include "fragments/index/movie-card.gtl" as MovieCard; %>
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
        content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0,
minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <% @Imports() %>
  <title>Grain | Cinema</title>
</head>
<body>
  <% @Nav() %>
<main id="index-page" class="container">
  <div class="row mb-0">
    <% @Pagination() %>
  </div>
  <div class="row">
    <% foreach (movie in movies) @MovieCard(movie=movie); %>
  </div>
</main>
</body>
</html>
```

Na vrhu strane vidimo importovanje GTL fragmenata koje koristimo u samoj strani. `foreach` petlja je najjednostavniji način da prikažemo listu elemenata. Takođe, imamo `Pagination` i `Nav` komponente koje se

ponavljaju na par strana.

Rezervacija

Na strani za rezervacije vidimo interfejs gde ulogovani korisnik može da vidi zauzeta mesta, slobodna mesta i mesta koja je on rezervisao različitim bojama.



Rezervacija

Klikom na slobodno mesto, korisnik može da rezerviše mesto.

Implementacija

Implementacija kontrolera svih strana prati sličan šablon. Developeri koji su radili na Spring projektima primetiće da ima velike sličnosti sa Spring-om.

```
@Controller
@RequestMapping("/screenings")
@RequiredArgsConstructor
public class ScreeningController {
  public final ScreeningService screeningService;

  // prikazivanje strane za rezervaciju
  @GetMapping("/{id}")
  public View index(@PathVariable("id") Long id) {
    TemplateView view = new TemplateView("pages/screening.gtl");
    view.addAttribute("screening", screeningService.findById(id));
    return view;
  }

  // rezervacija mesta
  @PostMapping("/{id}/reservations/{number}")
```

```
  public String reserve(@PathVariable("id") Long id, @PathVariable("number") Integer number) {
    screeningService.reserve(id, number);
    return "redirect:/screenings/" + id;
  }
}
```

Implementacija template-a:

```
<% include "fragments/import.gtl" as Imports; %>
<% include "fragments/nav.gtl" as Nav; %>
<% include "fragments/index/movie-card.gtl" as MovieCard; %>
<% include "fragments/util/spinner.gtl" as Spinner; %>
<% include "fragments/screening/seat.gtl" as Seat; %>
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
      content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0,
minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <% @Imports() %>
  <title>Grain | Cinema</title>
</head>
<body>
<% @Nav() %>
<main id="screening-page">
  <div class="row">
    <div class="col s12 l2"></div>
    <div class="col s12 l4">
      <div class="seats-container card">
        <h2>Seats<span class="free"><% @Spinner() %></span></h2>
        <div class="seats">
          <% foreach(seat in range(screening.getRoom().getSeats()) {
              @Seat(id = screening.id, number = seat + 1, taken = screening.isSeatTaken(seat +
1), self = screening.isSeatTakenBy(seat + 1, #authentication));
            } %>
        </div>
      </div>
    </div>
    <div class="col s12 l4">
      <div class="row pt-2">
        <% @MovieCard(movie = screening.getMovie(), cols = 's12') %>
      </div>
    </div>
  </div>
</main>
<script>
  // dobavljanje broja dostupnih mesta za prikaz u zaglavlju
  document.addEventListener("DOMContentLoaded", () => {
    fetch("/api/screenings/" + ${screening.id} + "/seats")
      .then(res => res.text())
      .then(res => {
        setTimeout(() =>
```

```
          document.querySelector(".free").innerHTML = "("+res+")", 500)
    });
  });
</script>
<script>
  // konfigurisanje click listenera za rezervaciju mesta
  // ovaj listener submit-uje formu koja je render-ovana
  // u sklopu seat.gtl fragmenta
  document.addEventListener("DOMContentLoaded", () => {
    document.querySelectorAll(".seat form").forEach(seat => {
      if (!seat.parentElement.classList.contains("taken") ||
      seat.parentElement.classList.contains("self"))
        seat.addEventListener("click", seat.submit);
    });
  });
</script>
</body>
</html>
```

`Seat` fragment:

```
<div class="seat ${taken ? 'taken' : ''} ${self ? 'self' : ''}">
  <form method="post" action="/screenings/${id}/reservations/${number}">
    <img src="/scaled.png" alt="Seat ${number}">
    <span class="number">${number}</span>
  </form>
</div>
```

Ovaj fragment prikazuje mesto. U zavisnosti od prosleđenih parametara, mesto može biti zauzeto, slobodno ili rezervisano od strane korisnika.

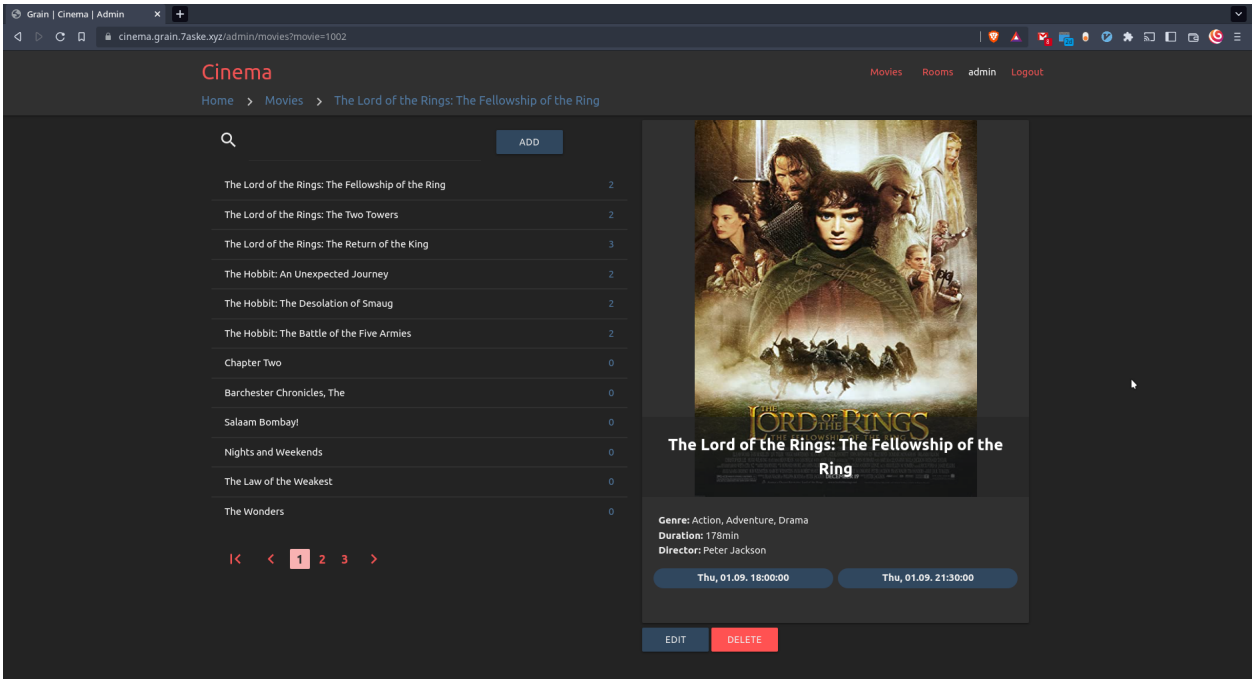
Ova strana je malo kompleksnija od drugih i uključuje API poziv koji vraća broj slobodnih mesta za datu projekciju. Naravno, mogli smo ovu vrednost proslediti template-u pre render-ovanja, ali smo odabrali ovakav pristup da bi demonstrirali asinhrone HTTP pozive u Grain radnom okviru.

```
@Controller
@RequestMapping("/api")
@RequiredArgsConstructor
public class ApiController {
  private final ScreeningService screeningService;

  @GetMapping("/screenings/{screeningId}/seats")
  public String getRemainingSeats(@PathVariable("screeningId") Long id){
    return String.valueOf(screeningService.getRemainingSeats(id));
  }
}
```

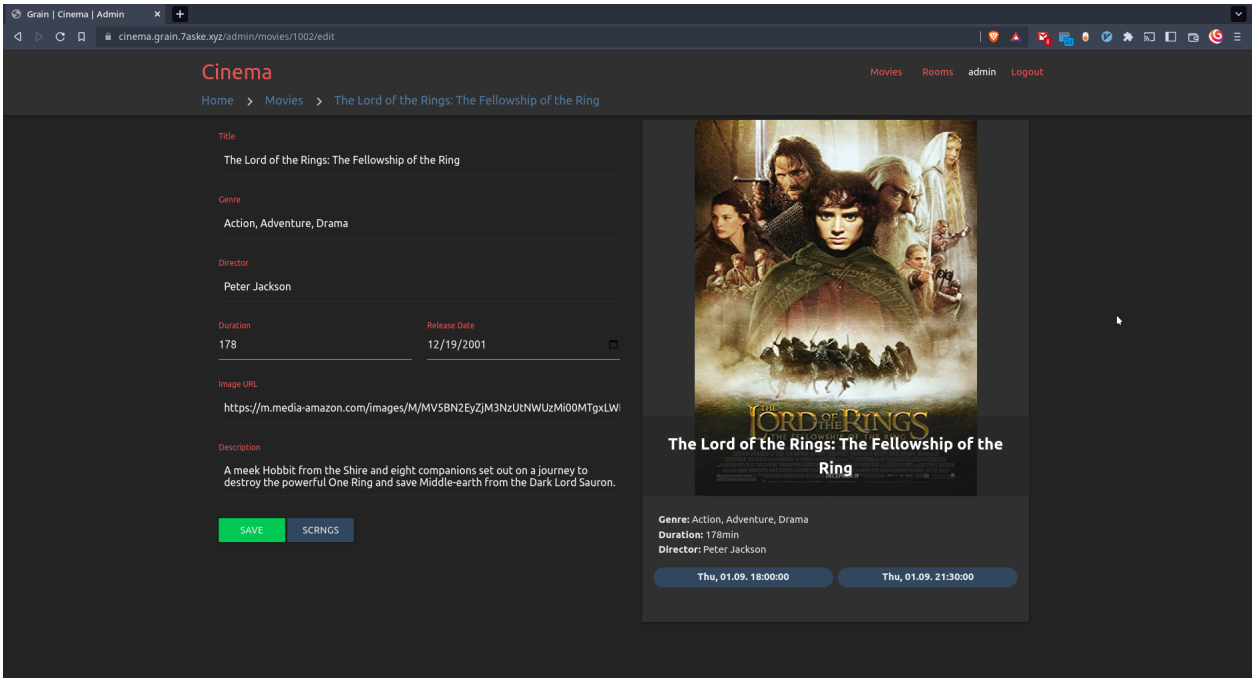
Ovde vidimo implementaciju endpoint-a koji vraća broj slobodnih mesta za datu projekciju.

Administracija filmova

Administracija filmova omogućava dodavanje novih filmova, izmenu postojećih i brisanje filmova. Ova strana je dostupna samo administratorima odnosno korisnicima sa `ADMIN` rolom.



Admin - Pregled filmova



Admin - Izmena filma

Videli smo princip implementacije kontrolera i strana u Grain radnom okviru tako da nećemo ponavljati kod za svaku od strana. Doduše, ono što je sada idealan trenutak za pokazati, kada već pričamo o rolama, jeste *security* konfiguracija:

```java
@Grain
public class SecurityConfig implements SecurityConfigurer {
  private static final Logger logger = LoggerFactory.getLogger(SecurityConfig.class);
  @Override
  public void configure(SecurityConfigurationBuilder sec) {
    logger.debug("Configuring security");
    sec.withRules()
        .urlPattern("/screenings/*/reservations/*").authenticated().and()
        .urlPattern("/admin/**").authenticated().roles("ADMIN")
        .buildRules();
  }
}
```

Vidimo da je su endpointi koji se odnose na rezervacije dostupni samo prijavljenim korisnicima, dok je administracija, odnosno svi endpointi koji počinju sa `/admin`, dostupna samo administratorima. Svi ostali endpointi su javni.

`UserService` iz Grain radnog okvira je implementiran u `UserService`-u ove aplikacije na sledeći način:

```java
@Grain
@RequiredArgsConstructor
public class UserServiceImpl implements UserService,
com._7aske.grain.security.service.UserService {
  private final PasswordEncoder passwordEncoder;
  private final UserRepository userRepository;

  @Override
  public User findByUsername(String s) throws UserNotFoundException {
    return userRepository.findByUsername(s)
        .orElseThrow(UserNotFoundException::new);
  }

  @Override
  public User register(RegisterUserDto dto) {
    if (!Objects.equals(dto.getPassword(), dto.getConfirm())) {
      throw new PasswordsNotMatchingException("Passwords do not match");
    }
    User user = new User();
    user.setUsername(dto.getUsername());
    user.setPassword(passwordEncoder.encode(dto.getPassword()));
    user.setRole(Role.USER);
    return userRepository.save(user);
  }
}
```

Ova komponenta implementira dva servisa pod istim imenom - iz tog razloga jedna od implementacija mora da ima svoj *FQCN* (*fully classified class name*). Ovaj servis je zadužen za dobavljanje korisnika iz baze prilikom login-a.

Zaključak

U ovom radu bavili smo se kreiranjem radnog okvira u programskom jeziku Java. Imali smo prilike da vidimo i objasnimo veliki broj kompleksnih koncepata koje koristimo na dnevnom nivou u svom poslu. Od osnova HTTP-a, preko umetanja zavisnosti do kreiranja novog programskog jezika. Između ostalog prezentovali

smo i realnu aplikaciju koje je kreirana u Grain-u i koja, naravno, može biti postavljena na realan server kome može pristupiti veliki broj klijenata. Mnogi od ovih koncepata su uzimani zdravo za gotovo, ali je veoma korisno izučiti ih i makar delimično ih implementirati radi jos boljeg razumevanja. Kao što je veliki Ričard Fejnman rekao: *What I cannot create, I do not understand.*

Reference

1. Perforce Software Inc., 2022, *2021 Java Developer Productivity Report*, https://www.jrebel.com/resources/java-developer-productivity-report-2021
2. Wikipedia, 2022, *Hypertext Transfer Protocol*, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
3. David H. Crocker, 1982, *RFC 822 - STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES*, Dept. of Electrical Engineering University of Delaware, https://www.rfc-editor.org/rfc/rfc822
4. IETF, 1999, *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*, https://www.rfc-editor.org/rfc/rfc2616
5. IETF, 2017, *RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format*, https://www.rfc-editor.org/rfc/rfc8259
6. IETF, 2015, *RFC 7519 - JSON Web Token (JWT)*, https://www.rfc-editor.org/rfc/rfc7519
7. YouTube, 2014, *Reverse Polish Notation and The Stack - Computerphile*, Computerphile, https://www.youtube.com/watch?v=7ha78yWRDlE
8. YouTube, 2017, *Parser and Lexer — How to Create a Compiler part 1/5 — Converting text into an Abstract Syntax Tree*, Bitwit, https://www.youtube.com/watch?v=eF9qWbuQLuw
9. Univerzitet Metropolitan - Fakultet Informacionih Tehnologija, *Literatura iz predmeta IT355 Veb sistemi 2*, Vladimir Milićević
10. Univerzitet Metropolitan - Fakultet Informacionih Tehnologija, *Literatura iz predmeta CS103 Algoritmi i strukture podataka*, Miljan Milošević
11. GitHub.com, 2022, *Grain*, Nikola Tasić https://github.com/7aske/grain
12. GitHub.com, 2022, *Grain Cinema*, Nikola Tasić https://github.com/7aske/grain-cinema