Veebirakendused Java baasil konspekt

LOENGUD: https://echo360.org.uk/section/87663d08-a7b2-451e-963a-5a25a8da8db1/home

Loeng 1 – Java EE, korraldusest

- Java EE ehk Enterprise Edition. Java SE ehk Standard Edition. Java EE on spetsifikatsioon, mis määrab, millist funktsionaalsust näiteks server peab toetama. Kui keegi tahab end nimetada Java veebiserveriks, peab ta just mingi kindlat viisi pidi toetama rakenduste paigaldamist.
- Php-s kopid faili serverisse ja töötab. Javas faili serverisse panemiseks vaja teha pakett, millel kindel formaat (web archive .war). Keeruline sest rakendus peab töötama erinevates keskkondades.
- Veebiserver võtab mingi päringu vastu, annab meie programmile edasi, programm otsustab mis sellega teha ja genereerib väljundi ning läbi veebiserveri saadetakse see brauserisse tagasi.
- Veebiserver tagab käivitamise keskkonna, võimaldab ligipääsu ressurssidele (andmebaas nt), cache, transaktsioonid jne.
- Annotatsioon on kompileeritav märgend (kui teed vea, siis ei kompileeru. Kui märgendiks oleks kommentaar, siis viga kompileerimisel välja ei tule)
- loe pakendamise kohta. .war (web archive) fail ja selle sisu (struktuur)

Harjutus 1

Rakenduse käsurealt käivitamine:

1) kompileerimine: javac Hello.java
2) jooksutamine: java Hello

Aga kui rakendus kasutab välist teeki siis asi nii kergelt ei lähe:

1) navigeeri teegi pakkuja leheküljele ning leia vastav teek (StringUtils nt mvnrepository.com → otsing commons lang 3 → vali õige teek → vali sobiv versioon); kompileerimisel on seda vaja staatiliseks tüübi kontrolliks, jooksutamisel vaja rakendus otseslt sellele teegi koodile ligipääsu
2) tõmba alla jar fail samasse kausta kus rakendus
3) nüüd saab kompileerida rakendust kuid peab andma teada et seda teeki vaja arvesse võtta:
javac -cp <teegi faili nimi nt commons-lang3-3.11.jar> Hello2.java
4) nüüd jooksuta javac -cp <teegi faili nimi nt commons-lang3-3.11.jar>**;.** Hello2

Pane tähele semikoolonit ja punkti teegi nime lõpus. Vaikimisi -cp kirjutab classpathi üle ehk nagu Hello2 klassi otsitaks ka sealt. Semikooloniga (linuxil koolon) eraldatakse erinevaid asukohti, et otsida korraga mitmest asukohast. Andes asukohana punkti ütled, et otsi seda faile ka praegusest asukohast kust see Hello2 ka leitakse.

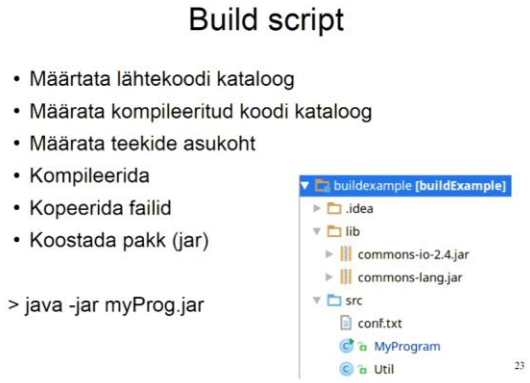
Loeng 2 – Projekti ehitamine

- Lähtekood vs programm – IDEs on sul lähtekood ja seal saad käivitada siis on nagu programm ja lähtekood koos aga kui tahaksid programmi kellelegi teisele edasi anda siis ei saa ju seda IDEga saata.
- Projekti ehitamine automaatselt: kui reposse midagi commitid, siis seatud webhook teatab süsteemile, et repos on muudatus toimunud. Kood laetakse alla serverisse ja ehitatakse projekt kokku ning kompileeritakse ning käivitatakse.
- Kompileermisel tekkviad küsimused: kus asub lähtekood, kus asuvad teegid, kuhu läheb kompileeritud kood?
- Kompileerimisel on staatiline kontroll, mis tähendab seda, et kontrollikse süntaksi vigu jms kas koodi on üldse mõtet käivitada.
- -cp käsk ütleb JDK töövahenditele ja applikatsioonidele kust leida kolmandate osapoolte defineeritud klasse – klasse, mis pole Java laiendid või osa Java platvormist. Ilma neid täpsustamata kompileerimine ebaõnnestub. Seda tuleb täpsustada nii kompileerimisel ja käivitamisel. Kompileerimisel staatline kontroll ja käivitamisel peab otseselt koodi käivitama ning endiselt teeki vaja, kuid kompileerimisel seda ju klassi faili juurde ei kirjutata kus see teegi fail on, eriti et vahel võis selle faili asukoht muutuda.
- Teegid on klassifailid aga kokku zipitud aga .jar laiendiga. Siit saab tuletada ka et .jar on kokku zipitud klassifailide kogum (+ teatud ekstra failid nt metadata ja pildid jms).
- Failide transformeerimine – programm võib muuta sinule mugaval kujul oleva faili enda jaoks mugavaks.
- Miks mitte kohe teha projekti ehitamine automaatselt kohe koodis? Ei ole hea mõte. Üks põhjus nt sest me

Projekti ehitamine (hüpoteetiline)

```
import tld.company.buildtools.Compiler;
...

public class Builder {

    public static void main(String[] args) {

        Compiler c = new Compiler();

        c.addSourceDir("src");
        c.addBuildDir("bin");
        c.addLibDir("lib");

        c.compileJava();
    }
}
```

ei tahaks, et meie build skript oleks kompileeritavas keeles (mida Java on), sest kui skripti muutame, siis peaksime teda ennast ka veel enne kompileerima käsitsi. Teine põhjus on keerukus.
- Domain specific language (DSL) ehk piiratud funktsionaalsusega keel konkreetse eesmärgi saavutamiseks. Mõeldud ainult kindla töö tegemiseks. Näiteks SQL, regex.
- Gradle on üks tööriist Java projektide ehitamiseks. Ta on kirjutatud JVM-le ja selleks peab olema sdk installitud. Gradle tahab kompileerida ja tal endal seda pole.
- Build scripti jaoks oleks vaja siis (vt 9. Sept 12.00 loeng 43:00)



- build.gradle failis on plugins all plugin “id ‘java’”. See ütleb et gradle abil üritatakse ehitada java projekti ning mis tekitab võimaluse luua sinna task **compileJava**, mis annab ette konfi kuidas kompileerida kust kuidas mida leida. **Jar** task annab ette mis jar paketti sisse läheb, andes ette kust failid saada, millise klassi main meetod käima panna (manifest), kust teegid saada.
- .war pakett on selle jaoks kui tahad programmi kuskile teise masinasse saata
- mida build.gradle faili siis kirjutada, et ehitatakse kokku see .war struktuur oma programmist:

> gradle init

1) vaja tõmmata vajalikud teegid
    - kõigepealt ütled kust asju tõmmata (repositories task alla nt jcenter()). See on tegelikult lihtsalt ühte urli avalikus serveris kuhu poole pöörduda.
    - teegid on kirjeldatud taski dependencies all. (taski all compileOnly ‘<teekide grupp>:< teegi nimi>:<versioon>’). CompileOnly asmele võib olla muu käsk. CompileOnly nimetatakse skoobiks ja see tähendab seda kus teek vajalik on ehk compileonly puhul teeki on vaja ainult kompileerimisel, sest nt serverisse pole vaja käivitamiseks kaasa panna sest igas serveris on antud teek juba olemas. Teine variant oleks nt implementation ehk mõlemal juhul vaja. Variante on veelgi.
2) kompileerida kood
3) kopeeri conf failid

4) teha war pakett või käivitada rakendus

- plugins sektsiooni plugin “id ‘war’”.

- gradle wrapper ehk gradlew: genereerib projekti neli faili (gradle-wrapper.jar [programm], gradle-wrapper.properties [siin on konkreetne gradle versioon olemas], gradlew, gradlew.bat. [kaks viimast on samad skriptid aga esimene mac/linux, teine win]). Kui öelda ‘gradle appRun’ eeldatakse gralde on arvutisse installitud, aga kui öelda ‘gradlew appRun’ või sama asi ‘gradlew.bat appRun’, siis installitakse properties failis olev gradle õige versioon ja kasutatakse seda.

Harjutus 2

Jar faili ehitamine gradle abil (käivitamine java -jar myapp.jar):

a) Genereerige kataloogistruktuur ja build.gradle fail.

```
> gradlew init        # (mac-i peal on "gradlew" asemel "./gradlew")
```

tehke järgmised valikud: application, Java, Groovi,
JUnit Jupiter, <vaikeväärtus>, <vaikeväärtus>.

b) Kopeerige failid just genereeritud kataloogidesse sobivale kohale.

> source failid kausta src/main/java

Kui programmil on mingi package siis java kausta teha sama package ja siis sinna kopeerida .java failid. **Conf** fail src/main/resources sisse. **Teek** läheb algkaustas olevasse kausta nt “lib”.

c) Kompileerige kood

build.gradle faili peaksite lisama kirje. Nt vt netist vastav teek nt apache commonsist midagi ja seal teegi lehel peaks ette antama gradle sektsiooni kirje midal lisada dependencies alla ja kompileerimisel kasutatakse teeki otse netist.

```
dependencies {
```

```
    implementation fileTree(include: ['*.jar'], dir: 'lib')
}
```

See näide ütleb seda et kohalikust lib kataloogist otsi kõik jar failid ehk kui oled teegi alla tõmmanud ja annad selle talle ette, et ta võtaks seda miite internetist vaid kohalikust arvutist.

d) Käivitage rakendus läbi Gradle.

build.gradle faili peaksite kirjutama. See on oma tehtud task nagu “init” või “build”.

```
task runMyCode(type: JavaExec) {
  main = '<klassi nimi koos paktetidega. nt. mypackage.MyClass>'

  classpath = sourceSets.main.runtimeClasspath
}
```

Käivitage kood läbi Gradle: gradlew runMyCode

e) Koostage jar pakk ja käivitage rekendus pakist (java -jar myapp.jar).

See osa teeb nagu selle kogu paketi kokku, et seda saaks nt kellelegi teisele edasi anda ja tema seda saaks käivitada.

build.gradle faili peaksite lisama kirje. (jari konf)

```
jar {
    manifest {
        attributes 'Main-Class': '<klassi nimi koos paktetidega. nt. mypackage.MyClass>'
    }

    duplicatesStrategy DuplicatesStrategy.EXCLUDE

    from(configurations.runtimeClasspath.collect
            { it.isDirectory() ? it : zipTree(it) })
}
```

Manifestis öeldakse mis käima tuleb panna. From osa on see mis pakki kaasa läheb. Paki koostab käsk.

> gradle jar

Paki leiate kataloogist build/libs. Seda siis saab käivitada navigeerides sellesse kausta kus see jar sees ning käsk java -jar <jar faili nimi>.jar.

Loeng 3 - Servlet API, Json, Lombok

- Kui server käima panna, hakkab ta klasside (vaikimisi kõikide) hulgast otsima, kas miski võiks talle huvi pakkuda. Eriti otsib ta just kas klassil on @WebServlet annotatsioon ehk kas tegemist on servletiga, kuid veel tuleb kontrollida, kas servlet vastab nõuetele (tal peab olema doGet meetod override). Igaühest, mis leitakse, teeb ta serveri käima minekul objekti, mis elavad niikaua kui server käib.
- Server vahendab kliendi tetud päringuid servlettidele. Millist servletti kasutada seda otsustab server urli põhjal.
- Peab oskama päringu näidet kirjeldada:

Projekti kirjeldusest

Päringu näide:

```
POST /api/orders HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{ "orderNumber": "A123" }
```

Pildi kirjeldus: tehtud on POST päring /api/orders aadressile, (musti asju ise ei määra v.a port?) content-type ehk missugusel kujul on saadetud andmed ja viimaseks siis reaalne sisu.

- Java → Json: String json = new ObjectMapper().writeValueAsString(<object>); build.gradle failis jacksoni implementation.
- Json → Java: Post post = new ObjectMapper.readValue(json, Post.class);

- HTTP protokoll on tekstipõhine
- Servlet query parameetri saamine: request.getParameter("name");
- Servlet headeri väärtuse saamine: request.getHeader("Content-Type");
- Servlet kogu päringu sisu saamine: InputStream is = request.getInputStream();
- Stream: pikkus pole teada ja lugeda saab ühe korra.

Servleti elutsükkel

- init()
- doGet(), doPost(), doPut(), doDelete()
- destroy()

- init() kutsutakse välja servleti mällu laadimisel. Selles pole hea muutujaid defineerida sest sul võib olla kümneid servlette ja tekiks küsimus milline servleti puhul.

- Webfilter läheb servleti ja serveri vahele. Kui tuleb päring mingile servletile, siis sõna saab alati kõigepealt filter.

- Lombok
- @AllArgsConstructor puhul peab olema ka @NoArgsConstructor, sest kui klassis konstruktorit ei ole, siis kompilaator ise automaatselt genereerib ilma argumentiteta konstruktori. Aga kui kõikide argumentidega on juba olemas, siis kompilaator ilma argumentiteta konstruktorit ei tee. Raamistikud aga ise tahavad neid kasutada, kuid nemad vajavad tühja konstruktorit selleks.
- build.gradle failis on rida annotationProcessor '..lombok…'. See teeb teegi selliseks eriliseks. Javal on võimalus et mingi klass on nn "Agent", mida on võimalik JVM kaasa anda. Agendil on võimalik teatud protsessidele vahele segada nt klassi kompileerimise ajal. Tema näeb klassi kompileerimise ajal klassi annotatsioone (lomboki omi) ning teab mis nendega teha ja genereerib vajalikku baitkoodi juurde.

Harjutus 3

- Objektist json string saamine:

> vaja on ObjectMapperit. Build.gradle dependencies alla panna

```
implementation 'com.fasterxml.jackson.core:jackson-databind:2.11.2'
```

```
> String json = new ObjectMapper().writeValueAsString(<objekt>);
```

- Stringist objekti saamine:

```
> Objekt objekt = new ObjectMapper().readValue(json, Objekt.class);
```

- Post request sisendi lugemine:

```
> String input = Util.readStream(request.getInputStream())
```

Util klass readStream meetod:

```
public static String readStream(InputStream is) {
	try (Scanner scanner = new Scanner(is, StandardCharsets.UTF_8.name())) {
		return scanner.useDelimiter(\\A).next();
	}
 }
```

- Postmani nt vastuse saatmine:

Enne vastuse saatmist oleks hea content-type seadistada:

```
> response.setContentType(“application/json”);
```

Vastuse saatmine:

```
> response.getWriter().println(<vastus>);
```

- Rakenduse teavitamisest teavitav Listener (servlet kaust ikka):

```
@WebListener
public class MyListener implements ServletContextListener {

	@Override
	public void contextInitialized(ServletContextEvent sce) {
		System.out.println(“Started…”);
	}
}
```

- Listenerist servletile saatmine (võib ka servlet -> servlet jne):

```
@WebListener
public class MyListener implements ServletContextListener {

	@Override
	public void contextInitialized(ServletContextEvent sce) {
		ServletContext context = sce.getServletContext();
		Post post = new Post();
		post.setTitle(“from                                    context”);
		context.setAttribute(“contextAttribute”, post);
	}
}
```

Selle saab servletis nt doGet meetodis kätte:

```
> ServletContext context = getServletContext();
> context.getAttribute(“contextAttribute”);
```

See aga tagastab Object tüüpi instantsi, kuigi meie panime sinna Post tüüpi instantsi.

Et saada tagasi Post siis tuleb castida:

```
> (Post) context.getAttribute(“contextAttribute”);
```

- Uue servleti manuaalselt lisamine (@WebServlet annotatsioon leitakse automaatselt, aga jätame selle ära, siis peame listeneris servleti seadistama)

```
@WebListener
public class MyListener implements ServletContextListener {

	@Override
	public void contextInitialized(ServletContextEvent sce) {
		ServletContext context = sce.getServletContext();
		OtherServlet servlet = new OtherServlet();
		ServletRegistration reg = context.addServlet(“otherServlet”,
			servlet); // servleti registreerimine
		reg.addMapping(“/otherServlet”);  // mappingu lisamine
	}
}
```

- Lombok

Build.gradle failis:

> compileOnly “org.projectlombok:lombok:1.18.12”

annotationProcessor “org.projectlombok:lombok:1.18.12”

Getterid ja setterid käivad @Getter või @Setter annotatsiooniga, kas klassi peale või property peale. Klassi peal lisab kõikidele propertitele. Annotatsioon @Data lisab korraga nii getteri, setteri, kui ka teeb hashcode, equals ja toString meetodid ka.

Üks mugav annotatsioon on veel @Builder. see võimaldab meil ühe reaga objekti luua ja sellele valikuliselt väärtusi anda:

```
Post post = Post.builder()
	.id(1L)
	.title("A")
	.build();
```

- PMD stiilireeglid

Settings → Plugins → PMDPlugin

Saad kontrollida stiili nt terve src kataloogi peal:

Paremklikk src peale → Run PMD.

Et teha Custom Rules:

Tõmba alla projekt https://bitbucket.org/mkalmo/hwtests

Settings → Other Settings → PMD → Lisa projekti ruleset.xml faili asukoht → Apply → OK

Loeng 4 - Jdbc, PostgreSql

- Javas on kaht liiki erindid: checked või unchecked. Checked erindeid tulevad välja kompileerimisel, unchecked runtime'is. Checked erindid peavad olema kontrollitud kas try-catch plokis või deklareeritud *throws* keywordiga.
- Jdbc – Java Database Connectivity. On andmebaasi API
- API mõiste – tarkvaraliides, mis võimaldab rakendustel omavahel suhelda
- Iga andmebaasisüsteemi tegija mõtleb ise välja protokolli kuidas selle süsteemiga suhelda, sest andmebaasiga ei ole ühiseid kõigile sama standardit kuidas seda suhtlust korraldada. Klient, kes andmebaasiga suhelda tahab, aga ei tea kuidas seda protokolli kasutada. Selleks peab ka süsteemi looja looma vastava draiveri, kes oskaks kliendi ja süsteemi vahelist suhtlust korraldada. Igal andmebaasisüsteemil peaks olema oma protokoll ja oma draiverid jms. Jdbc töö ongi see asi ära abstraheerida, et ükskõik mis andmebaasisüsteemiga suhelda klient soovib, kõik päringud ja ühendused käivad samamoodi. Kuidas aga jdbc teab kuidas iga süsteemiga suhelda? Jdbc'le saab laadida sisse iga süsteemile vastava driveri. Kliendile näeb kõik samasugune välja, aga kuidas jdbc andmebaasiga suhtleb, see tuleb kuskilt välisest driverist.
- Ressurss (resource) – objekt, mille peab kindlasti sulgema pärast seda kui programm on lõpetanud selle kasutamise.
- *Try with resources* – kõik objektid (ressursid), mis try-le argumentidena antakse (jah, sulud peale try-d), nendelt eeldatakse et nad implementeerivad AutoClosable liidest ja on tagatud et igal juhul see suletakse. Kui tekib exception, suletakse enne catchi jõudmist, kui ei teki, siis try blocki lõpus.

```
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

- Miks nii? Sest SQLException on checked, aga kui näiteks ei saa andmebaasiga ühendust siis ongi asi katki ja ei tahagi sellega tegeleda ning viskad unchecked exceptioni edasi.

Draiveri laadimine

```
static {
    try {
        Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

- Siin static on nagu konstruktor, aga seda plokki ei panda käima mitte siis kui klass luuakse nagu konstruktori puhul, vaid siis kui see klass mällu laetakse. Programmi kohta laetakse klass mällu ühe korra ehk see plokk kutsutakse välja kokku ühe korra, aga konstruktorit kutsutakse välja nii palju kordi kui sellest klassist instantse luuakse. Antud pildil tagatakse see et draiver laetakse mällu ainult ühe korra ehk draiver ise ka läheb käima ainult ühe korra.

- jdbc ühenduse loomine

String url = “jdbc:protokoll (nt postgresql):host”;

Connection connection = DriverManager.getConnection(url, username, password);

- tabeli loomine (executeUpdate tähendab, et selline päring ei tagasta midagi, select puhul executeQuery vt järgmine pilt)

Tabeli loomine

```
try (Connection conn = DriverManager.getConnection(...);
     Statement stmt = conn.createStatement()) {

    stmt.executeUpdate(
        "CREATE TABLE person (id INT, name VARCHAR(100))");

} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

Tagastusega päring

```
try (Connection conn = DriverManager.getConnection(...);
     Statement stmt = conn.createStatement()) {

    ResultSet rset = stmt.executeQuery(
            "SELECT id, name FROM person");

    while (rset.next()) {
        System.out.println(
             rset.getLong("id")
             + ", "
             + rset.getString("name"));
    }

} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

```
1, John
2, Jack
3, Jill
```

20

- ResultSet puhul ei loeta vastust kohe mällu, muidu ei saaks suuri päringuid teha mis palju andmeid tagastavad (mälu saab täis). ResultSet on selline aktiivne objekt, mille käest saab andmeid küsima hakata (rset.next() – kontrollib kas järgmine rida on olemas ning kui on, seab lugeja sellele reale ning saab lugeda andmed välja rset.getLong("id") jne). Ridu ei tooda ühekaupa vaid suuremas hulgas ja vajadusel tuuakse juurde.).

Parameetritega päring

```
try (Connection conn = DriverManager.getConnection(...);
     PreparedStatement ps = conn.prepareStatement(
             "SELECT id, name FROM person WHERE id = ?")) {

    ps.setLong(1, 3L);

    ResultSet rset = ps.executeQuery();

    ...
```

```
"...&id = " + id; // EI  TOHI!!!
```

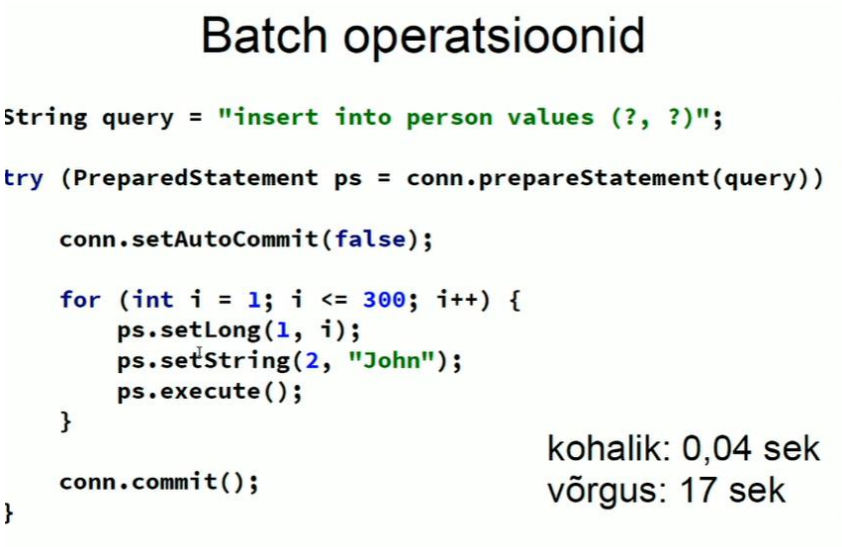
- PreparedStatement puhul kui liidad sqllausele otsa midagi nt id, siis sqlinjection oht. Liites midagi otsa ei oska kompilaator seda ka efektiivselt ära kasutada (kui varem on selline päring tehtud oskaks ta seda ära kasutada ja peaks vähem tööd tegema). Kasutades parameetritega päringut ei oleks seda probleemi.

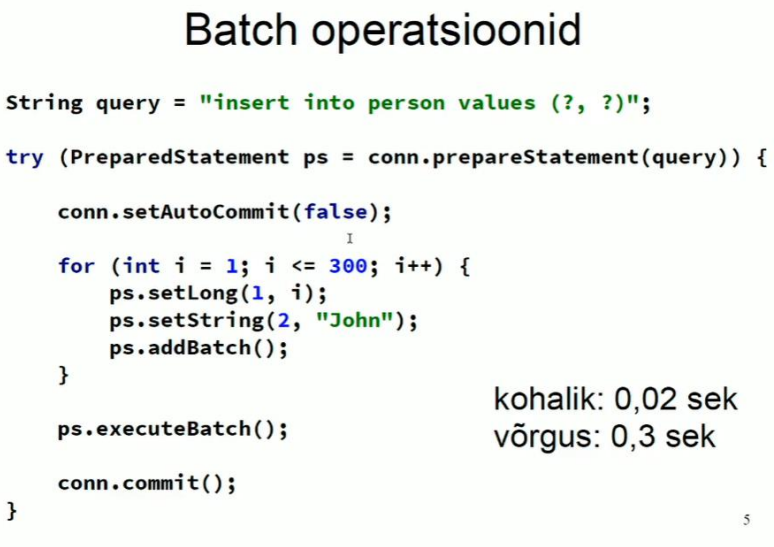
- prepareStatment'ile võib anda teise parameetrina new <tüüp>[] { "<välja nimi" }, mis tagastab genereeritud info.

```
PreparedStatement ps =
      conn.prepareStatement(sql, new String[] { "id" });

...

ps.executeUpdate();

ResultSet rs = ps.getGeneratedKeys();

if(!rs.next()) {
    throw new RuntimeException("unexpected errror!");
}

System.out.println(rs.getLong("id"));
```

61

- Kõik eelnevad andmebaasi API käsud (DriverManager, Conenction, Statement jne) asuvad java.sql pakis.
- Ühenduste puul – ühenduste loomine on üsna kallis tegevus, enne igat päringut ei taha teha uut ühendust. Aga kui on mitu klienti siis ühest ühenduskanalist jääb väheks. Vahepealne variant oleks mingi hunnik ühendusi (pool), mis tehakse valmis rakenduse käivitamise hetkel ja neid hakatakse siis sealt kasutama. Kui kõik ühendused kasutusel siis peab ootama, kui keegi lõpetab saab ühendus vabaks, aga ei panda vahepeal kinni
- javax.sql.DataSource – abstraktne koht ühenduste saamiseks (ühenduste pool üks võimalus).
- UUID – *Universally Unique Identifier* – 16 baiti (sisu on kuueteistkümnend numbrid, stringinga 32). Andmebaasi peab reaalselt see tüüp olemas, muidu on ta tavaline string (ehk 32 baiti). On unikaalne ülemaailma, sinna sisse arvutatakse kellaaeg, masin kohta infot jms. PROS: 1) aitab varjata infot; 2) ei ole äraarvatav; 3) saab genereerida baasist väljas; 4) alati unikaalne üle kõigi serverite; CONS: 1) võib anda infot (masina koht mis genereeris nt); 2) arenduse ja testimise ajal ebamugav kasutada (nt annad id lingis parameetrina kaasa); 3) andmemaht; 4) ühilduvus erinevate andmebaaside ja raamistikega; 5) jõudlus
- DAO muster – point eraldada andmebaasikood ülejäänud rakenduse koodist. Disaini mõiste läks kood jube koledaks kui kõik on näiteks kontrolleritesse kirjutatud. SQL laused ja ühendused on peidetud teise interface'i taha ära ja vajadusel kutsutakse seda välja ning saadakse see.





- Batch operatioonid (transaktioon) – kõik või mitte midagi. Ei ole variant et üks asi õnnestub ja teine mitte. Kui üks ei õnnestu ei tehta ka teisi.
- Autocommit mõte – pildil on seatud false, et ei juhtuks seda, et peale igat execute'ti see autocommititakse. Kui aga teha autocommit siis võib juhtuda see et kui on palju neid execute'te ja ühega neist vahepeal tuleb error, siis pooled said tehtud, pooled mitte. Et seda vältida tehakse commit alles lõpus – see on "kõik või mitte midagi" metoodika. Commiti ajal tehakse ka muude piirangute kontroll. Selle metoodika puhul tehakse see kontroll siis ühe korra lõpus, mitte peale igat commiti. Siin on siiski aga ka halb pool, sest endiselt peale igat execute'i saadetakse serverisse päring (pildil foreach käib 300x). Kasutades batch'i saadetakse kõik päringud teele ühe korraga ja nii hoitakse võrguliikluse arvelt palju kokku.

Harjutus 4

- kui teha sql insert käsk on hea kirjutada käsku sisse ka välja nimed => **insert into** person (name, age, sex) **values** ('Juku' , 20, 'M'); . Kui seda ei kirjuta, siis tuleb alati väärtustada kõik väljad, aga kui kunagi lisatakse lisaatribuut siis see insert käsk ei töötaks.

Loeng 5 – Silumine

- Kordamiseks: Mis on JDBC mõte – teeb kõikidega baasidega suhtlemise ühesuguseks, teeb ühesuguse API nendele, ühtlustada erinevate andmbeaasidega suhtlemist.
- Invariants – mingisugused tingimused mis peavad koguaeg kehtima

Loeng 6 – Spring Core raamistik

- Kordamiseks: mis on javax.sql.Datasource – liides, mille abil saab luua ühendusi tõelise andmekogumiga, luua connection pool jms
- Spring Core ülesandeks on süsteemi eri osade (service, dao, andmebaas jne) kokkusidumine. See tõstab paindlikkust (ei kasutata new keywordi, Spring seetõttu saab teha igasugu põnevaid asju)
- Konfiguratsiooni klassi (annotatsiooniga @Configuration) saab panna @Bean annotatsiooniga meetodeid, mis tagastavad mingit tüübi objekti. Siit oskab Spring leida vastavat tüüpi objekte kui neid konteksti käest küsida:

```
ConfigurableApplicationContext ctx =
        new AnnotationConfigApplicationContext(Config.class);


PersonDao dao = ctx.getBean(PersonDao.class);
```

- @ComponentScan() anna ette package ja siis suudab Spring leida selles pakis olevatest klassidest.
- @Component – märgib klassi, et Spring suudaks üles leida @ComponentScan abil.

@ComponentScan

```
@Configuration
@ComponentScan(basePackages = {"mypackage"})
public class Config {

}
```

```
package mypackage;

@Component
public class ReportService {
   ...
}
```

```
package mypackage;

@Component
public class PersonDao {
   ...
}
```

15

- Kokku siis kaks võimalus leida vajalikku klassi: 1) @Bean meetod konfi klassis või klass tähistatud @Component ja konfis antud @ComponentScan'iga ette asukoht kus teda leida võib. Kui luua mõlemad korraga, annab Spring errori, et vaja üks eemaldada VÕI seadistada nii, et üks kehtib ühe profiili ja teine teise puhul.
- Dependency Injection – sõltuvuste süstimine raamistiku poolt ehk koodi ei ole kirjutatud, mis näiteks mingi välja väärtuseks on vaid käivitamise ajal süstitakse väljale sisse sobilik versioon vajalikust asjast.

- Profiilide kasutus: erinev andmebaas, ühenduste puulimine, teavituste saatmine, logimine, ligipääs… kõik igasugune stuff, mida arenduse ajal teha ei taha või ei tohi üldse juhtuda võrreldes live tootega.
- JdbcTemplate – pealisehitus Jdbc-le, tuleb kaasa Springi raamistikuga, teeb andmebaasiga suhtlemise mugavamaks. Peidab ära igasugused connectioni küsimused, preparedStatementid jms.

JdbcTemplate (sisestus)

```
@Repository
public class PersonDao {

    public JdbcTemplate template;

    public PersonDao(JdbcTemplate template) {
        this.template = template;
    }

    public void save(Person person) {
        String sql = "INSERT INTO person (name, age) values (?, ?)";

        template.update(sql, person.getName(), person.getAge());
    }
}
```

- Ülemisel pildil kuidas sisestada andmeid. Teha sql lause, kasutada template.update millele parameetritena sql ja varargsidena väärtused. Query ei tagasta midagi.

JdbcTemplate (tagastusega)

```
@Repository
public class PersonDao {

    ...

    public String getPersonName(Integer id) {
        return template.queryForObject(
                "select name from person where id = ?",
                new Object[] { id }, String.class);
    }
```

- Ülemisel pildil kuidas teha tagastusega päringut. Viimane parameeter on tagastaustüüp (nagu ObjectMapperi puhulgi oli) ja seetõttu ei saa varargs kasutada väärtuste puhul. Väärtused lähevad

massiivina sql ja tüübi vahele vastavas järjekorras. Aga kui pärida terve hulga väljade väärtusi, siis ei saa ju tagastusväärtuseks lihtsalt String panna? Sel juhul tuleb teha mapper:

JdbcTemplate (objekti tagastusega)

```
public List<Person> findAllPersons() {
    return template.query("select id, name from person",
            new PersonMapper());
}

private class PersonMapper implements RowMapper<Person> {
    public Person mapRow(ResultSet rs, int rowNum)
            throws SQLException {

        Person person = new Person();
        person.setId(rs.getLong("id"));
        person.setName(rs.getString("name"));
        ...

        return person;
    }
}
```

28

- Ülemisel pildil mapper kuidas päringu tulemused mappida objektiks.
- Kasutades reflektsiooni saab mappimise veelgi paremaks teha. Kui reflektsiooni saab kasutada ainult siis kui päringu tulemused väljad klapivad üks-ühele objekti väljadega:

```
public List<Person> findAllPersonsAutoMapper() {
    return template.query("select id, name from person",
            new BeanPropertyRowMapper<Person>(Person.class));
}
```

- Ülemisel pildil: reflektsiooni abil mappimine kasutades BeanPropertyRowMapperit.
- RowMapperitega tekib siiski probleem kui päringus on JOIN klauslid ning päringut vaja rohkem töödelda. Selleks on RowCallbackHandler:

```
private class PersonRowHandler implements RowCallbackHandler {
    List<Person> result = ...

    public void processRow(ResultSet rs) throws SQLException {

        String name = rs.getString("name");

    }

    public List<Person> getResult() { ...
}
```

- Ülemisel pildil: processRow meetod teeb sama töötluse mis muidu peaks kuskile Dao klassis ära tegema, et päringu vastuse ridu töödelda. Peale rea töötlemist tulemus result listi. GetResult tagastab listi. RowHandler klassi anda sama moodi sisse nagu mapperit päringu ehitamisel:

```
var handler = new PersonRowHandler();

template.query(sql, handler);

handler.getResult();
```

SimpleJdbcInsert

```
var data = Map.of("name", "Alice",
                  "age", 20);

Number id = new SimpleJdbcInsert(template)
        .withTableName("person")
        .usingGeneratedKeyColumns("id")
        .executeAndReturnKey(data);
```

- Ülemine pilt: jdbc abil sisestamine (Number on ükskõik mis numbriline tüüp, saab küsida getLongValue, getIntValue vms). Andmed võib jällegi anda ette mugavamalt kui väljad klapivad:

```
var data = new BeanPropertySqlParameterSource(person);
```

```java
@Bean
public JdbcTemplate getTemplate(DataSource dataSource) {

    var populator = new ResourceDatabasePopulator(
            new ClassPathResource("schema.sql"),
            new ClassPathResource("data.sql"));

    DatabasePopulatorUtils.execute(populator, dataSource);

    return new JdbcTemplate(dataSource);
}
```

- Ülemisel pildil: skeemi loomine. ClassPathResource loeb classpathi pealt faili, classpathi lähevad .war paketi ehitamisel kõik resource kataloogis olevad failid.

Sama tüüpi klassid

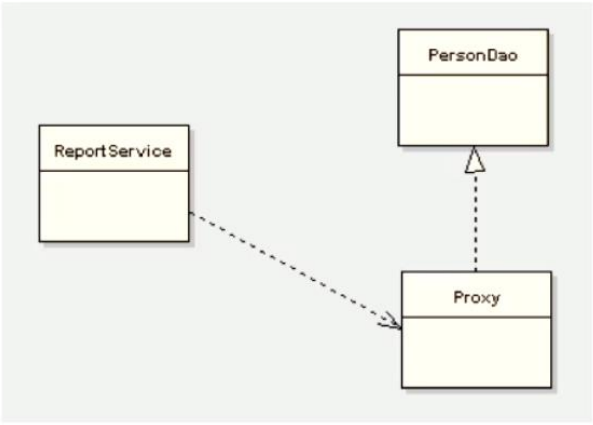
```java
@Configuration
public class Config {

    @Bean(name = "postgreDs")
    public DataSource postgreDataSource() { ...

    @Bean(name = "mongoDs")
    public DataSource mongoDataSource() { ...
```

```java
@Repository
public class Dao {

    private DataSource postgreDs;

    private DataSource mongoDs;

    public Dao(@Qualifier("postgreDs") DataSource postgreDs,
               @Qualifier("postgreDs") DataSource mongoDs) {

        this.postgreDs = postgreDs;
        this.mongoDs = mongoDs;
    }
```

- Aspect Oriented Programming (AOP) – eesmärk on tegevuste (ehk programmi erinevate aspektide nt logimine, transaktsioonid jne), mis jagunevad üle programmi, tsentraliseerimine, et

saavutada see, et ei peaks neid (sarnaseid) tegevusi käsitsi igal pool programmis tegema, vaid oleks ühtne koht.

AOP



execution(public * example.PersonDao.*(..))

- Ülemisel pildil: reportService vajab tööks personDao'd. Seda küsib ta läbi proksi, Proksisid saab aga panna tegema igasuguseid lisategevusi. Et näiteks kui tulevad mingit teatud sorti tegevused, siis tuleb teatud muid asju ka teha. Selleks on loodud ka eraldi keel (pildil näide ütleb, et kui käivitatakse **public** meetod kataloogist **example.PersonDao,** mille tagastustüüp ning klassi nimi võib olla ükskõik millised ja parameetrid ei ole olulised, SIIS pane midagi käima).
- HyperSQL andmebaas – võrreldes postgre, mysql jms andmebaasidega on hsql liides kõigest 1.4 Mb (teised on sadades Mb-des). Ta on mälupõhine andmebaas ehk ei jäta midagi meelde (ei kirjuta midagi kettale). Hsql-il on kõik sql pool samamoodi olemas, aga kuna andmebaaside puhul enamik jõudu kulub ketttale kirjutamisega, siis hsql liidese maht on väljapaistev. Hsql pole siiski päris andmebaas, tal puuduvad teistega võrreldes kõiksugu protokollid jms et see andmebaas kettale võimalikult efektiivselt kirjutatakse. Hsql paneb pigem kirja kõik sql laused kuidas baasi seisu taas luua ja kui rakendus käima läheb siis need kõik käivitatakse.

Välised parameetrid

```
@Configuration
@PropertySource("classpath:/application.properties")
public class Config {

    @Bean
    public DataSource hsqlDataSource(Environment env) {
        ...
        ds.setUrl(env.getProperty("db.url"));

        return ds;
    }
```

- Ülemisel pildil: väliste parameetrite lugemine, et ei tahaks koodi otse kirjutada. @PropertySource viitab välisele konfiguratsioonile, mille andmed loetakse Environment objekti, mis tuleb ise Springiga kaasa (Spring ise süstib selle sisse automaatselt). Sellest objektist saab need välised andmed välja lugeda.

Cache

```
@Repository
public class MemoryMessageRepository {

    @Override
    @Cacheable(value = "message")
    public Message getMessage(String key) {
        System.out.println("Fetching message");
        return messages.get(key);
    }

    @Override
    @CacheEvict(value = "message", key = "#message.key")
    public void save(Message message) {
        messages.put(message.getKey(), message);
    }
```

Jäta meelde! :

- ctx.getBean(<x>.class) - Spring-ilt x tüüpi objekti küsimine.
- @Repository ja @Service - Annotatsioonid, mis ütlevad Spring-ile, et tema peab märgistatud klassi tundma.
- @ComponentScan(basePackages = {"service", "..."}) - Ütleb Spring-ile, et nendest pakkettidest tuleks bean-e otsida.
- System.identityHashCode(x) - Ütleb objekti x identifikaatori. Kui kaks muutujat annavad sama tulemuse, siis viitavad nad samale objektile.
- @Scope(BeanDefinition.SCOPE_PROTOTYPE) - ütleb, et sellest bean-ist tuleb iga kord uus koopia anda (default on ta singleton).
- @PropertySource("classpath:/<faili nimi>") - viide välisele konfiguratsioonile, mis laetakse Environment objekti.
- Baasist lugemine (select)
    - var mapper1 = klass mis implementeerib RowMapper liidest.
    - var mapper2 = new BeanPropertyRowMapper<>(<klass, milleks soovime tulemuse mappida>)
    - new JdbcTemplate(<viide DataSource objektile>).query(sql, <mapper1 või mapper2>);
- Baasi kirjutamine (insert)
    - var data = new BeanPropertySqlParameterSource(<DTO objekt>);
    - new SimpleJdbcInsert(<viide JdbcTemplate objektile>)
      .withTableName(<millisesse tabelisse>)
      .usingGeneratedKeyColumns(<id välja nimi>)
      .executeAndReturnKey(data);
- Skeemi loomine ja näidisandmete sisestus
    - var populator = new ResourceDatabasePopulator(
      new ClassPathResource("schema.sql"),
      new ClassPathResource("data.sql"));
    - DatabasePopulatorUtils.execute(populator, dataSource);
- @Profile("<profiili nimi>") - Ütleb, et see konfiguratsioon või bean kehtib ainult seadistatud profiili jaoks. Kui profiil on seatud näiteks “test”, siis konfiguratsiooni, mis

kehtib profiili “dev” puhul üldse ei arvestata, isegi et ta on konteksti lisatud. Profiil valitakse keskkonnamuutuja järgi

- @EnableAspectJAutoProxy - Luba AOP-d konfiguratsiooni beanidele ja komponentidele.
- @Aspect - AOP element mis, milles olevat koodi võib dünaamiliselt liita muule koodile. Sellega tähistatud klass on komponent, mis leitav konfi poolt ning et teha lisategevusi näiteks logimist enne mingit tegevust siis logimise meetodi peale annotatsioon @Before(“<siia see aop lause, mille täitumisel meetod täidetakse>”).
- @Before("execution(* packagename.ClassName.*(..))") - tähistb meetodit, mis tuleb käivitada määratud enne määratud meetodite väljakutset.

Loeng 7 – Spring MVC, Valideerimine (JSR 303)

- MVC – *Model-View-Controller* = Äriloogika-Esitlus-Päringud
Model tegeleb andmetega, loogikaga ja rakenduse reeglitega. View on nagu äriloogika esitlus kasutajale. Controller võtab vastu sisse tulevaid päringud ning tõlgib need käskudeks rakendusele. Controllerite kaudu samamoodi saadetakse päringuid.
- Vaatekihti jäävad nii controller kui view. Teenus- ja andmekiht on mudeli all.
- Servlettide asemele tuleb üks servlet DispatcherServlet mis teeb palju meie eest juba ära, nätieks konteksti loomine.
- Spring MVC kasu; 1) integreerib springi veebirakendusse; 2) sunnib peale mvc arhitektuuri; 3) pakub mugavust ja funktsionaalsust
- Konteksti loomine on ära peidetud (AnnotationConfigApplicationContext(Config.class)). Vaja teha controllerid kus Spring ise juba injectib asju peale
- Varem oli nii et kui Servlet käima läheb hakkab ta otsima klasse millel on WebServlet annotatsioon. Kui ta sellise leidis, vaatas et ta laiendaks HttpServlet klassi ning seejärel pani ta vastava aadressi peale üles.
- Spring MVC-s on nii, et spring mvc jar’is on on kataloog META-INF/services, mille sees oleva faili sisu paneb server tööle. See fail on ServletContextInitializer, mis otsib meie classpathi pealt klassi, mis implementeerib (mingil sügavusel vähemalt) WebApplicationInitializer interface’i. Selles on meetod startup() ja see pannakse tööle.

Raamistiku laadimine

```
public class ApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/api/*" };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MvcConfig.class };
    }

    ...
}
```

16

- Üleval pildil: raamistiku laadimine, AbstractAnnotation…Initializer'isse on peidetud kogu DispatcherServleti loomine (mis on frontController ametis) ning implementeerib WebApplicationInitializerit, milles oleva startup() meetodi paneb Spring MVC rakenduse käivitumisel tööle. GetServletMappings() meetodis antakse ette mis aadressi peale rakendus püsti panna (sinna otsa liidetakse kontrollerites täpsustatud aadressid) ning getServletConfigClasses() meetodis antakse ette konfi fail(id).

- POST meetod:

```
@PostMapping("customers")
public void save(@RequestBody Customer customer) { ...
```

- GET meetod:

```
@GetMapping("customers/{id}")
public Customer getById(@PathVariable Long id) {
    return dao.findById(id);
}
```

- Parameetrite kätte saamine:

```
@GetMapping("customers/search")
public List<Customer> search(
            @RequestParam(defaultValue = "") String key) {

    return dao.search(key);
}
```

- Üleval pildil: @RequestParam'iga tähistatud välja nimi on queryString urlis. Kui defaultvalue't pole või öeldud et väli optional (nt String asendades Optional<String>), siis on väärtus nõutud.
- Konfigureerimine päris sarnane nagu varem:

Konfiguratsioon

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = {"hw8.customer"})
@PropertySource("classpath:/application.properties")
public class MvcConfig {

    @Bean
    public DataSource dataSource(Environment env) {
        ...
    }

    ...
```

- @EnableWebMvc paneb tööle mvc annotatsioonid nagu @RestController, @PathVariable, @RequestParam jne
- Funktsionaalne programmeerimine – kõik funktsioonid käituvad nagu matemaatikas funktsioonid, neil on sisend ja väljund ning muid kõrvalisi tegureid ei ole.
- Funtkstionaalne programmeerimine vs objekt-orienteeritud
- JSR 303 – Java Specification Request. Valideerimise teek kas väljad vastavad nõuetele. Objektide väljadel erinevad annotatsioonid nt @NotNull, Size(min = 0, max = 2). Need määravad reeglid väljale, missugune väärtus olema peab. Kui on väli mille tüüp on näiteks list, mille sees on alamobjektid (nt List<Contacts>), tuleb see tähistada annotatsiooniga @Valid, et kontrollitaks ka

nende alamobjektide korrektsust. See @Valid annotatsioon lisatakse ka nt kontrolleris sisendi ette, et seda valideerida:

```
@PostMapping("customers")
public void save(@RequestBody @Valid Customer customer) {
    ...
}
```

Neid annotatsioone saab ka ise juurde kirjutada.

- Mis saab aga kui ei vasta nõuetele? Viskab exceptioni, kuid seda üldiselt ei taha, vaid tahaks korralikku nt json sõnumit tagasi saada. Selleks saab Springis kirjutada meetodi kontrollerisse, mis tegeleb vigade püüdmisega:

Vigade püüdmine (kontroller)

```
@ExceptionHandler
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ValidationErrors handleValidationErrors(
          MethodArgumentNotValidException exception) {

    ValidationErrors errors = new ValidationErrors();

    ... = exception.getBindingResult().getFieldErrors();

    ...

    return errors;
}
```

Valideerimise vastus JSON

```
{
  "errors": [
    { "code": "NotNull.customer.code",
      "arguments": []
    },
    { "code": "Size.customer.firstName",
      "arguments": ["15", "2"]
    }
  ]
}
```

- Et mitte igas kontrolleris samu ExceptionHandlereid korrata oleks võimalus nt teha mingi ülemkontroller kust kõik kontrollerid pärinevad ja need handlerid sinna sisse. Parem võimalus oleks aga AOPd kasutada:

Kasutamise näide (advice)

```
@RestControllerAdvice
public class ValidationAdvice {

    @ExceptionHandler
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ValidationErrors handleMethodArgumentNotValid( ...

        ...

}
```

- Ülemisel pildil: eraldi klass kontrollerite kõrval. Kõik loogika, mis siia sisse on kirjutatud, on kirjutatud ka kõikidesse teistese kontrolleritesse. Mõiste ‘advice’ tähendab koodi, mida on võimalik kuhugile vahele toppida.
- @JsonFormat(pattern=”yyyy-MM-dd”) – käivad LocaleDate välja peale DTO-des et konvertida jsoniks. Selleks peab projektile lisama ka sõltuvuse teegist jackson-datatype-jsr310
- Kui vaja saata post päringus parameetrina LocalDate objekti, siis panna ette @DateTimeFormat:

```
@PostMapping("date")
public String readAndConvertDateParameter(
        @RequestParam
        @DateTimeFormat(pattern = "yyyy-MM-dd")
        LocalDate date) {
```

Loeng 8 – Spring Boot

- Genereerimine vs automaatne konfiguratsioon – koodi genereerimine enamasti väga hästi ei toimi, sest piiravalt keerulise konfiguratsiooni puhul võib tulla tahtmine seda hiljem muuta, aga

generaator meie muudatusi üldjuhul ei arvesta. Töötab ainult siis kui see genereeritud asja pole hiljem enam tahtmine muuta.

- Spring Boot ise on hästi õhuke raamistik, kuid lihtsalt taustal konfigureerib meie eest hulga asju ära.

- Spring Boot miinused:

  - Kõike ei saa automaatselt – palju asju genereeritakse, aga kui sa isegi ei tea et need asjad seal sees on, siis sa ei oska neid otsida ja tuunida.
  - Suur võit algaja jaoks – kui sa ei tea täpselt kuidas mida konfda siis on hea et ette ära tehakse, aga kui tead, siis sa pigem eelistaksid isegi käsitsi teha.
  - Eelmised kaks kokku – vähem koodi, aga ka vähem kontrolli
  - Suurem paki maht (palju ülearuseid teeke)

- Spring Data: repo

- REST – *Representational State Transfer* – arhitektuuriline lahendus, mis seab paika kindlad tavad/reeglid, mida kasutatakse veebiteenuste arendamisel.

  - Üks peamisi ideid on vahemälu kasutamine ja skaleeritavus ehk ei ole otseselt sisse kirjutatud kus keegi peab asuma ehk kelle poole klient täpselt pöördub. Klient ei pöördu konkreetselt mingi kindla serveri poole, vaid nende vahel on hoopis mingi vahendaja, kelle külge on võimalik ühendada nt võimsuse tõttu lisaservereid. Vahemälu saab efektiivsuse tõstmiseks kasutada, nt kui tehakse GET päring, mida on alles hiljuti tehtud (HTTP protokollis on kirjutatud loogika, kas tohib vahemälust otse tagastada või tuleb ikka serveri poole pöörduda), siis vahendaja vahel ei suhtle üldse serveritega, vaid saadab eelmise sama GET päringu vastuse tagasi.
  - Resource identification in requests – kui on mingi kindel *endpoint* (nt /orders), mille poole peaksid kõik päringud minema, siis tekiks skaleeritavusega probleem. Kui on aga /orders/1/orderrow/3, /orders?search?product=computer, siis saab ressursse paremini tükeldada. Cahce'imisel on aga väike aga kus järgmise kaks endpointi on siiski erinevad

    /api/customers/search?startDate=...&endDate

    /api/customers/search?endDate=...&startDate

    Parameetrid cache'itakse täpselt samas järjekorras nagu sisse tulid. Selle probleemi vältimiseks täpsustatakse parameetrid tähestikulises järjekorras.

- Self-descriptive messages – HTTP ei peaks hoidma eelnevate päringute kohta infot. Iga päring peab sisaldama vajalikku infot päringu teostamiseks.
- Hypermedia – klient ei pea teadma kõik linke, mis lehel olemas, vaid lehel saab ise klikkides navigeerida edasi teistele lehtedele. Seda aga siiski praktikas siiski tihti ei kasutata teatud probleemide tõttu.

- RESTi puhul tähtsad mõisted:
  - Resource -igasugune asi mille kohta on võimalik infot saada.
- Swagger – põhimõte kirjeldada API-t.

Loeng 9 – JPA

- JPA – *Java Persistance API*
- JPA on ORM-i standard, määrab liidesed ja annotatsioonid.
- Puudused: paindumatus, keerukus, harjumatus, raportid. Eelised: sõltuvus andmebaasist päris väiksemaks, testkeskkonna ja live keskkonna andmebaasi eristamine
- ORM – *Object Relational Mapping* – objektid on meil programmis, andmebaasis on relatsioonid või tabelid ja nende vahel toimub mappimine.
- @Transactional – meetodi alguses alustatakse transaktsiooniga ja lõpus pannakse kinni. JPA puhul AutoCommit on default *false*.

- *Entity* – olem, millel on omaette elutsükkel (saab omaette eksisteerida). Kui on mingi olem (objekt), mis ei saa ilma teiseta eksisteerida, siis pole ta päris *entity* ning talle pole tegelikult mõtet id-d omistada. *Entity* poolel tuleks see suhe siis märkida @ElementCollection (nt inimese ja telefoninumbri vaheline suhe – telefoni numbril pole enam mõtet kui inimest enam andmebaasis ei ole). Teisele (mitte *entity* poolel märkida object @Embeddable ja tal pole id-d.
- EntityManager meetodid:

Olemi laadimine:

```
em.find(Order.class, 1L);
```

Olemi muutmine:

```
Order o = em.find(Order.class, 2L);

if (o != null) {
    o.setNumber("A123");
}
```

Olemi kustutamine:

```
Employee employee = em.find(Employee.class, 1L);

if (employee != null) {
    em.remove(employee);
}
```

Eelmised toimivad kui vaja teha teha ühte operatsiooni nt kustutada üks kirje ära, kuid kui vaja kustutada mitu kirjet siis vaja teha tavaline query (NB! Panna tähele, et tegemist ei ole tavalise SQL-iga, vaid JPQL (*Java Persistence Query Language*)):

```
Query query = em.createQuery("delete from Person");

query.executeUpdate();
```

Parameetritega query (query käivitab getResultList(), ühe väärtuse jaoks getSingleResult()):

```
TypedQuery<Employee> query = em.createQuery(
        "select e from Employee e where e.id = :id",
        Employee.class);

query.setParameter("id", 4L);

List<Employee> employees = query.getResultList();
```

Olemi salvestamine (persist kui on uus objekt, merge kui on juba eksisteeriv objekt (tal oleks id) . EntityManager puhul vaikimisi autocommit on väljas, persist tuleb teha transaktsioonina ehk meetodi alguses peab olema @Transactional (või käsitsi seadistama). Selleks et transaktsioone kasutada peab konfis olema peale annotatsioon @EnableTransactionManagement:

```
Person jack = new Person("Jack");
em.persist(jack);
```

```
Person jill = new Person("Jill");
person.setId(1L);

em.persist(jill); // error
em.merge(jill); // ok
```

- JPQL räägib objektidest ja nende väljadest, mitte tabelitest ja väljadest ning ei sõltu konkreetsest andmebaasist (nt Oracle).
- Database First – API tegemist alustatakse andmebaasi skeemist. 1) See hoiab rakendust koos, sest kui arendajad hakkavad arendama, siis nad näevad, mis moodi on kõik rakenduse jaoks vajalikud andmed kirjeldatud ning nii on nende andmete ümber mugav rakendust ehitada. 2) Paremad võimalused optimeerimiseks. 3) negatiivne külg et baasi skeem ja GUI mõjutavad rakenduse arhitektuuri (koodi), ei tahaks et mõne nädalaga kirjutatud asi (baasi skeem) hakkaks mõjutama asja, mille peale läheb aasta või rohkem (arhitektuuri püsti panek ja kood).
- Code First – alustamine koodi kirjutamisest. 1) mugavam – ei pea baasile mõtlema, kohe koodi. 2) baasi skeem ei mõjuta arhitektuuri ja hea koodi disaini tegemine lihtsam.
- Code First ja Database First kesktee – alustame koodist ning fikseerime ja optimeerime baasi skeemi hiljem Nii saame alustada parimast vaatest koodi poolel ja samas teha minimaalseid valikuid/optimeerimisi baasi pooleks. Pluss: mugavam; pole baasi skeemi ees saab programmi disainida ilma baasile mõtlemata. Halb: kirjutada koodi ilma baasile mõtlemata viib lihtsalt otsejoones võssa.
- ElementCollection:

```
@Entity
public class Person {
    @Id
    private Long id;

    @ElementCollection
    private List<Phone> phones;
```

```
@Embeddable
public class Phone {
    // no id field
    private String number;
    ...
}
```

- JPA konf:

```
@Bean
public EntityManagerFactory
          entityManagerFactory(DataSource dataSource) {

    LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();
    factory.setPersistenceProviderClass(
            HibernatePersistenceProvider.class);
    factory.setPackagesToScan("model");
    factory.setDataSource(dataSource);
    factory.setJpaProperties(additionalProperties());
    factory.afterPropertiesSet();

    return factory.getObject();
}
```

- Üleval pildil: setPackagesToScan sisse antakse paketi nimi kus on sees kõik entity'd. Samuti ette anda datasoure ja jpa jaoks mõned lisa valikud:

```
private Properties additionalProperties() {
    Properties properties = new Properties();

    properties.setProperty("hibernate.dialect",
      "org.hibernate.dialect.PostgreSQL10Dialect");

    properties.setProperty(
            "hibernate.show_sql", "true");

    properties.setProperty(
            "hibernate.hbm2ddl.auto", "create");


    return properties;
}
```

51

- Üleval pildil: dialect on sinu andmebaasi dialect et jpa oskaks suhelda; show_sql on debugimiseks mis prindib konsooli tehtud sql laused; viimane on mis saab tabelitega kui neid ei ole aga neid päritakse – peamised väärtused: 1) "create" mingi entity pärimisel mida ei tunta ära

ise loob vastava tabeli entity klassi annotatsioonide järgi, rakenduse käivitamisel loob kõik otsast ja varasemad andmed kaovad; 2) “validate” – kontrollib entity’te vastavaust skeemile, kui leiab mingi puuduva entity siis mitte ei loo seda nagu “create” puhul, vaid annab teada, et seda ei ole (ei tunne ära); 3) “update” uuendab andmebaasi kui midagi on puudu või ei vasta skeemile - nt entity’l puudub väli ehk ei vasta skeemile, siis ta teeb välja juurde.

- HSQL server:

```
public static void main(String[] args) {
    Server server = new Server();

    server.setDatabasePath(0,
                           "mem:db1;sql.syntax_pgs=true");

    server.setDatabaseName(0, "db1");

    server.start();
}
```

- CascadeType.: (neid rohkem) 1) persist – kui entity on persisted, siis temaga seotud alam-entity’d on lisatakse samuti persisted (persist oli andmebaasi uue kirje lisamine); 2) all – kõik operatsioonid; 3) merge; 4) remove

```
@OneToOne(cascade = CascadeType.PERSIST)
private Address address;
```

```
Person person = new Person("Alice");
person.setAddress(new Address("Pine Street"));

em.persist(person);
```

- FetchType: LAZY - pärid tabelist, mis seotud teise tabeliga, siis esialgu ei võeta teise tabeli andmeid kaasa, vaid alles siis kui tõeliselt neid vaja. Kannatab jõudlus, nt pärid daost andmeid ja siis kui teed päringu alamandmetele, siis vahel juba ühendus kinni pandud (peab konfima nii et seda ei juhtuks). Probleemiks võib tulla ühenduse katkemine.


```
@ElementCollection(fetch = FetchType.LAZY)
private List<Phone> phones;
```

```
Person person = em.find(Person.class, 1L);

person.getPhones(); // tehakse uus päring
```

EAGER – pärib andmed kohe mõlemast seosest ning loetakse mälust. Probleem, et vahel pole seosetabelist andmeid vaja kuid need võetakse ikka kaasa (aeg ja mälu).

- ORM jõudlus: lasta kõik päringud enda eest nii teha maksab jõudluses kätte. Käsitsi tehes ise otsustad alati mis andmed päritakse, saab teha kõige optimaalsemalt, kuid ORM teeb tõenäoliselt mõnes kohas rohkem päringuid kui vaja. Siiski võita võib ka kui skeem kasvab nii suureks et käsitsi optimeerimine oleks hirmus töö, siis võib ORMist ka parema jõudluse saada.
- Kui mingid model klassidel on ülemklassid (mingi baseEntity ehk väljad mis kõik entity'tel on olemas nt id), peab ülemklassil olema peal @Getter, @Setter et pärida neid välju (@Data päris hea ei oleks). Samuti arvestada @EqualsAndHashCode(callSuper = false) entity klassidele peale. Ülemklassil peab olema ka annotatsioon @MappedSuperClass, sest kui entity klassis vajalik id, kuid see ülemklassi viidud, siis muidu ei leia jpa seda üles. See samuti ütleb, et klass on ülemklass taaskasutamise otstarbeks ja siis jpa ei tee sellest eraldi tabelit andmebaasi.

Õpi!

- @PersistenceContext private EntityManager em;
  ütleb, et siia tuleb süstida EntityManager-i sõltuvus.
- @Autowired ei ole siin piisav, kuna EntityManager peab olema seotud andmebaasiga ja iga kasutaja peab saama uue koopia jne.
- @Entity - ütleb, et seda klassi peab olema võimalik andmebaasi salvestada.
- @Id - ütleb, et seda välja kasutatakse andmebaasis id-na.

- @GeneratedValue - ütleb, et sellele väljale peab andmebaasi ise väärtuse panema.
- @SequenceGenerator(name = "my_seq", sequenceName = "<järjendi nimi>", allocationSize = 1)
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "my_seq")
  See komplekt ütleb, et id väärtused tuleb võtta järjendist (sequence) nimega <järjendi nimi>.
- @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "<välisvõtme veerg>")
  See komplekt mingil väljal ütleb, et see väli viitab seosele teises tabelis. Nii ei looda lisatabelit. Seose hoidmiseks andmebaasis on väli <välisvõtme veerg>.
  CascadeType.ALL ütleb, et kui salvestatakse kirjet, kus antud annotatsioonid on, siis tuleb salvestada ka sõltuval kirel tehtud muudatuse
- @ElementCollection
  @CollectionTable(
  name = "<alamkirjete tabeli nimi>",
  joinColumns=@JoinColumn(name = "<alamtabeli välisvõtme välja nimi>",
  referencedColumnName = "<peatabeli id välja nimi>"))
  See komplekt mingil kollektsiooni tüüpi väljal määrab, et alamkirjed asuvad tabelis <alamkirjete tabeli nimi> ja kirjete vaheline seos on määratud väljadega <alamtabeli välisvõtme välja nimi> ja <peatabeli id välja nimi>
- @Embeddable - ütleb, et seda tüüpi objektid tuleb salvestada eraldi tabelisse.
  Erinevus @Entity-ga märgitud objektidest on see, et nendel pole id välja.
  Seda tüüpi objektid eksisteerivad ainult koos neid koondava objektiga.
- factory.setPackagesToScan("model");
  ütleb, et sellest paketist tuleb otsida @Entity ja @Embeddable annotatsiooniga märgistatud klasse.
- properties.setProperty("hibernate.hbm2ddl.auto", "update");
  ütleb, et olemasolevat skeemi tuleb uuendada või kui see puudub, siis uus luua.
- properties.setProperty("hibernate.show_sql", "true");
  ütleb, et näidata välja millised päringud andmebaasi saadetakse.
- @Transactional
  ütleb, et selle meetodi alguses tuleb alustada transaktsiooniga

ja meetodi lõppedes alustatud transaktsioon lõpetada (commit).

- @Table(name = "<tabeli nimi>") - määrab tabeli nime, kuhu seda tüüpi objektid salvestada.
- @Column(name = "<välja nimi>") - määrab veeru nime, kuhu selle välja sisu salvestada.

Loeng 10 – Andmebaasi versioneerimine

- Kordamiseks: mis probleemi lahendab ORM: vähendada sõltuvust relatsioonilise andmebaasi vahel, sest seal asjad teistmoodi kui objekt-orienteeritud koodis kirjutada tahaksime. Aitab kirjutada oma programmi võimalikult objekt-orienteeritult, sest andmebaasi eripära ei kandu edasi meie programmi.
- Andmebaasi versioneerimine:

- Skeemi versioneerimine – SQL laused

  - Kui midagi resource kataloogi lisatakse siis Gradle kopeerib need classpathi peale kust need leitakse (nt flyway poolt)

  - flyway_schema_history tabelis hoitakse andmeid selle kohta, millised skriptid on juba käima läinud. Iga skripti kohta hoitakse räsi (checksum veerg tabelis), kui skripti muuta, siis räsi ka muutub

- Andmete versioneerimine

  - Atomaarne operatisoon – käsk, mida ei saa mitte miski enam takistada, kui see on juba käima läinud. Mitte ükski teine lõim (*thread*) ei saa seda käsku näha pooleldi olevas seisus.

  - *Synchronized* võtmesõna – saame koodis märkida mingi osa (nt meetod) sünkroniseeritavaks ehk selles osas saab olla ainult üks lõim (saavutad atomaarsed operatsioonid).

  - *Deadlock* – "*a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread*"

  - Krüptoahel: kõikide tabeli kirjete väljadest pannakse kokku räsi ning see lisatakse väljana kirjele juurde. Iga järgneva kirje (mis seotud eelmisega, nt sama id) räsi moodustub kirje väljadest + eelmise kirje räsi. Ehk kui eelmist kirjet (ajalugu) muudetakse, siis muutub ka

hetkel kehtiva kirje räsi. Hetkel kehtiva räsi välja ei tohi saada muuta, see peab olema täiesti lukus.

Loeng 11 – Spring Security, JWT, Oauth2.0

- Kordamiseks: Kuidas toimub Spring MVC laadimine – mingi konf laiendab (..)ServletInitializerit, mis omakorda implementeerib WebApplicationInitializerit, mis kutsub välja onStartup(), mis kutsub getServletMapping(), mille me ise üle kirjutame ja selles paneme dispatcherServleti mingi kindla aadressi peale.

- Http protokoll on olekuta ehk automaatselt ei tehta vahet, kes päringu tegi. Seda tehakse kindlaks sisselogimisega, sisselogimise andmeid hoitakse küpsisega, mida antakse päringuga kaasa. Vaikimisi hoitakse neid andmeid sessioonis 15 minutit ehk kui seni aeg ühtegi päringut ei tee siis kaotatakse need andmed ära serverist.

- Sessioon on serveri poolne mälu. Seal on võti väärtus paarid.

- Spring security laadimine (eraldi initializer sarnaselt spring mvc laadimisele):

Spring Security laadimine

```
public class SecurityWebApplicationInitializer
        extends AbstractSecurityWebApplicationInitializer {

}
```

- Spring Security konf läheb mvc initializeris getRootConfigClasses alla:

```
public class MyApplicationInitializer extends
       AbstractAnnotationConfigDispatcherServletInitializer {

   ...

   @Override
   protected Class<?>[] getServletConfigClasses() {
      return new Class[] { MvcConfig.class };
   }

   @Override
   protected Class<?>[] getRootConfigClasses() {
      return new Class[] { SecurityConfig.class };
   }
```

- Spring Security konf

Seadistamine

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/static/**").permitAll()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/**").authenticated();

        http.formLogin();
    }
}
```

```
/*   - /home
/*   - /home?lang=en
/**  - /home
/**  - /products/88135/details
```

18

- Üleval pildil: kui aadress algab “static” siis kõik läbi lubada. Kui algab “admin” peab roll olema admin. Kõik muu peab olema autenditud
- Kasutajate lisamiseks **testimiseks** kõige basicum kasutada in-memory autentimist (security konfi failis):

```
@Override
protected void configure(AuthenticationManagerBuilder builder) {

    builder.inMemoryAuthentication()
            .passwordEncoder(new BCryptPasswordEncoder())
            .withUser("user")
            .password("$2a$10$3rN/1Dt4gMt4NacjYJn5LeVFXlB7a...")
            .roles("USER")
            .and()
            .withUser("admin")
            .password("$2a$10$WID7JrpmmWCGzQODSHjNoOp6/RC8t...")
            .roles("ADMIN", "USER");
```

- Vajalikud muudatused:

Vajalikud muudatused

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    ...

    http.formLogin();

    http.exceptionHandling()
        .authenticationEntryPoint(new ApiEntryPoint());
    http.exceptionHandling()
        .accessDeniedHandler(new ApiAccessDeniedHandler());

    http.logout()
        .logoutUrl("/api/logout")
        .logoutSuccessHandler(new ApiLogoutSuccessHandler());
```

- Üleval pildil: mitte kasutada default login vormi,edasi exceptionhandlerid, successhandlerid jms. Viimasena logout aadress ning mis saab kui logout õnnestub (vaikimisi suunatakse kuskile avalehele vormi peale)

- Login filter:

Kasutaja info Json-ist

```
@Override
protected void configure(HttpSecurity http) ...

    ...


    var loginFilter = new ApiAuthenticationFilter(
                authenticationManager(), "/api/login");

    http.addFilterAfter(loginFilter, LogoutFilter.class);
```

- Miks Spring Security? Turvaasju pole enamasti mõtet ise teha, sest sisaldavad turvaauke (palju kohti mille peale ei oska mõtelda). Peenemaid autentimise asju teha käsitsi väga raske teha (OpenID, Oauth2.0). Meetodipõhine juurdepääsu kontroll, igasugu mugavused.

- BCrypt võtab salasõna (+ paneb soola otsa), teeb sellest räsi, siis teeb räsist veel räsi jne jne jne.
- Rainbow tables – tabel kus salasõnade räsid (mingi kindla keerukusega) on juba ette arvutatuad. Keerukus on mitu korda on mingi salasõna räsi algoritmist läbi lastud ehk kõigepealt on salssõna mis tehakse räsiks ja siis see räsi tehakse veel omakorda räsiks jne. Keerukus 10 tähendab 2^10=1024 korda niimoodi teha.
- Soola meetod – salasõnale pannakse random jada otsa ja nii igakord kui sama salasõna räsi arvutatakse tuleb tegelikult erinev räsi. See random jada hoitakse meeles.
- Kasutaja näeb ainult enda andmeid (auth ja principal süstitakse springi poolt). Kui keegi küsib nt Toomase andmeid, siis enne kontrollitakse kas kasutaja on ise see Toomas:

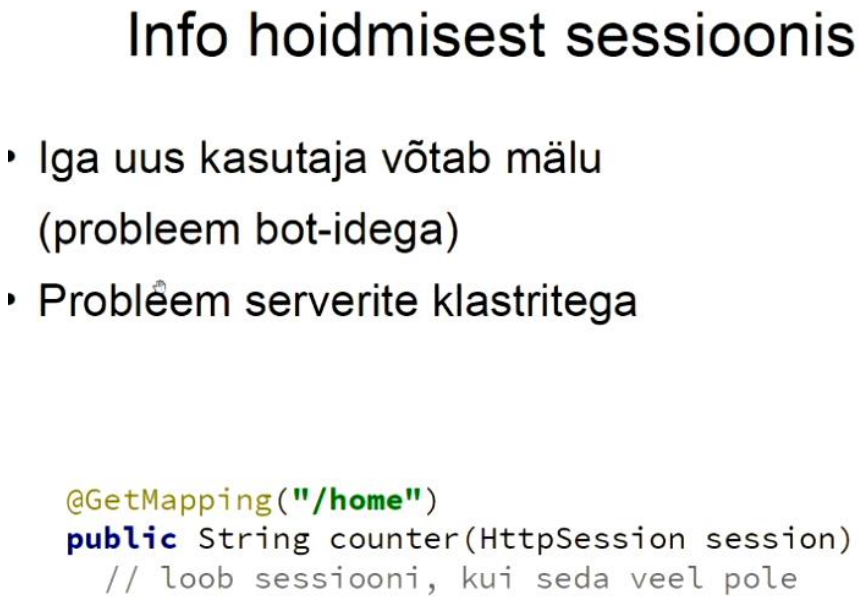
Authentication, Principal

```
@RestController
public class UserController {

    @GetMapping("/users/{username}")
    public User getUserByName(@PathVariable String username,
                              Authentication auth,
                              Principal principal) {

       System.out.println("Principal: " + principal.getName());
       System.out.println("Authorities: " + auth.getAuthorities());
       System.out.println("auth.principal: " + auth.getPrincipal());

       if (...

       ...
    }
```

- Eelmise slaidi lihtsustus:

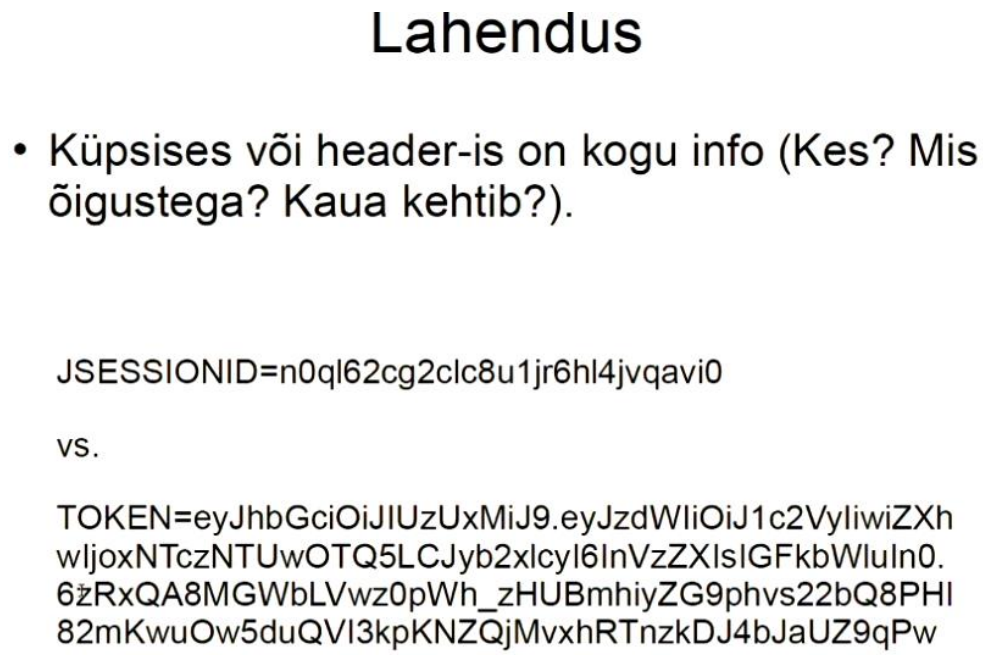
```
@RestController
public class UserController {

    ...

    @GetMapping("/users/{username}")
    @PreAuthorize("#username == authentication.name")
    public User getUserByName(@PathVariable String username) {
        return dao.findByUsername(username);
    }
```

- Sessionid (kuidas hoitakse infot kas oled sisse logitud ja kes sa oled) :

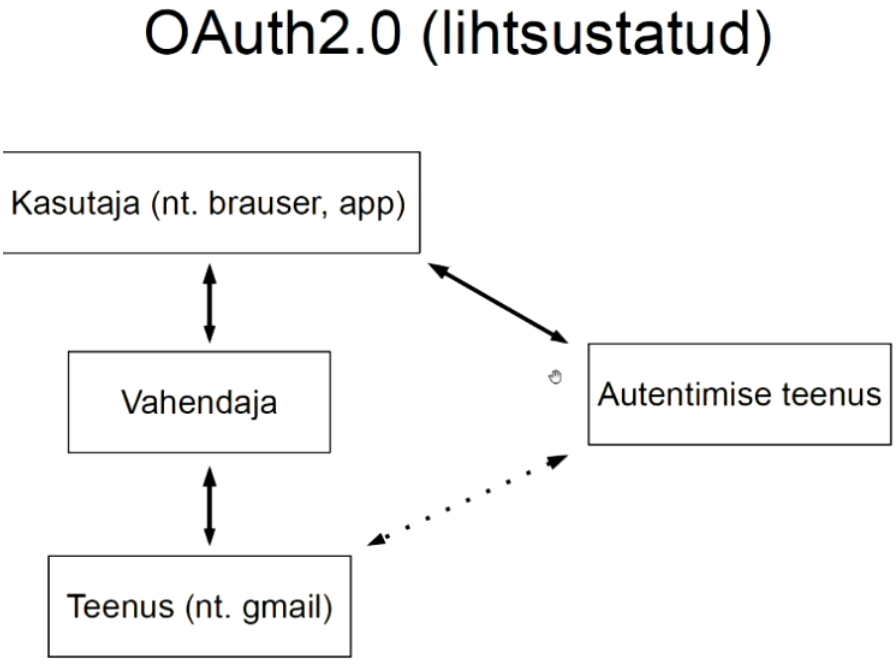



```
@GetMapping("/home")
public String counter(HttpSession session) {
  // loob sessiooni, kui seda veel pole
```

- Üleval pildil: iga sisseloginud kasutaja nõuab serveri poolel mälu. Bottide puhul on see et nad ignoreerivad küpsiseid (ei saada seda päringuga kaasa ning alati iga päringuga reserveeritakse mälu, tõmbavad mälust tühjaks). Samuti probleem klastritega kus kasutatakse loadbalancerit ehk ühendus suunatakse sinna kus vähem liiklust. Iga kasutaja aga vaja suunata alati sinna serverisse kus ta esimest korda suunati. Kui mõni server maha peaks kukkuma siis info kohe läinud.
- Lahendus saata kogu info ühes tokenis (küpsisesse kaasa ainult info kes sa oled). Nii ei pea serveri poolel mälu reserveerima, saata info vahel ühte serverisse, vahel teise. Klient annab koguaeg authorization tokenit kaasa:



- Tokenit aga ei toeta brauserid vaid seda peab rakendus ise meeles hoidma ja kaasa andma.
- JWT info on välja loetav, kuid kuna sisaldab krüptograafilist räsi siis seda muuta ei saa.
- Oauth2.0 (loengu järgi saan aru nii et) kasutatakse nt kui mingi rakendus soovib ligipääsu sinu mingile nt google kasutajale, et ta saaks sinna sinu töid salvestada. Kuidas teha nii et see rakendus tõesti ainult salvestab ja midagi muud ei tee? Sinu rakendus või nt brauser tahab anda luba teisele rakendusele ligipääsuks -> suunatakse autentima (nt google) -> google saadab tagasi tokeni milles kõik õigused jms kirjas -> token edastatakse rakendusele mis ligipääsu soovib -> rakendus edastab selle sinna kuhu ligipääsu soovib (nt gmail või drive) -> nt gmail kontrollib autentimise teenusest uuesti et selle tokeniga jamatud poleks vahel -> kui ok, antakse luba rakendusele ligipääsuks. Kokkuvõttes saavutati see et rakendus saab sulle pakkuda mingit nt gmail või drive teenust ilma et sa talle oma salasõna andsid.
- Sümmeetrilise võtme küptograafia – võti mis on nii krüpteerimiseks kui ka lahti krüpteerimiseks. Mõlemal poolel on see võti.
- Avaliku võtme krüptograafia (sellel pärineb SSH) – algoritm, milles on kaks võtit. Üks võti on krüpteerimiseks (avalik ehk kõigile näha võti), teine on lahti tegemiseks.



- OAuth2.0 probleemid: keerukus (kui ei mõista seda, oluline vigu teha), tokeni uuendamine (tokenis kirjas kaua kehtib, serveri pool mälu ei ole sellest, vaja käia kehtivuse kaotamisel uuesti küsimas), väljalogimine (vahendaja vastustus välja logida)
- Andmebaasipõhine kasutajate haldamine:

```java
@Override
protected void configure(AuthenticationManagerBuilder builder) {

    builder.jdbcAuthentication()
            .dataSource(dataSource)
            .passwordEncoder(new BCryptPasswordEncoder());
}
```

Tabelid

```sql
CREATE TABLE USERS (
  username VARCHAR(255) NOT NULL PRIMARY KEY,
  password VARCHAR(255) NOT NULL,
  enabled BOOLEAN NOT NULL,
  first_name VARCHAR(255) NOT NULL
);

CREATE TABLE AUTHORITIES (
  username VARCHAR(50) NOT NULL,
  authority VARCHAR(50) NOT NULL,
  FOREIGN KEY (username) REFERENCES USERS
    ON DELETE CASCADE
);

CREATE UNIQUE INDEX ix_auth_username
                    ON AUTHORITIES (username, authority);

INSERT INTO users (USERNAME, PASSWORD, ENABLED, FIRST_NAME)
   VALUES ('user', '$2a$04$3rN/1Dt4gMt4...', true, 'Jack');

INSERT INTO users (USERNAME, PASSWORD, ENABLED, FIRST_NAME)
   VALUES ('admin', '$2a$10$WID7JrpmmWC...', true, 'Jill');

INSERT INTO AUTHORITIES (USERNAME, AUTHORITY)
   VALUES ('user', 'ROLE_USER');

INSERT INTO AUTHORITIES (USERNAME, AUTHORITY)
   VALUES ('admin', 'ROLE_ADMIN');
```

- CSRF rünnak – oled loginud nt panka ja brauseris hoitakse su sessioni küpsist. Raha ülekandmisel tehakse täiesti tavaline POST request, mille saab ründaja teha ka kõrvalaknast ilma et peaks üldse panga lehel olema. See toimib sest kui kasutaja on panka sisse logitud siis isegi kui oled kõrvalaknas paneb brauser su sessiooni küpsise kõrvalaknas tehtud requestile kaasa. Kaitse on POST päringule veel random väärtus kaasa anda mis vormis on peidetud. Ründaja seda teada ei saa. Vaikimisi spring mvc ootab iga post päringu puhul seda väärtust kaasa. API põhistel rakendustel pole seda vaja ja saab selle välja lülitada:

```
@Override
protected void configure(HttpSecurity http) {

    http.csrf().disable();

    ...
```

Loeng 12 – Kotlin, evitus (deployment)

- Kotlin – programmeerimiskeel, mille kood kompileerub java baitkoodiks
- Miks Javas olid getterid (ei saa pärida person.name, vaid person.getName()) – põhimõte oli selles, et raske jälgida millal selle välja pealt midagi võtma minnakse. Välja lugemisele ei ole võimalik breakpointi panna et debugida.

Loeng 13 – Klientrakendus

- CDN: *Content Delivery Network*
-