Razvoj informacionih sistema

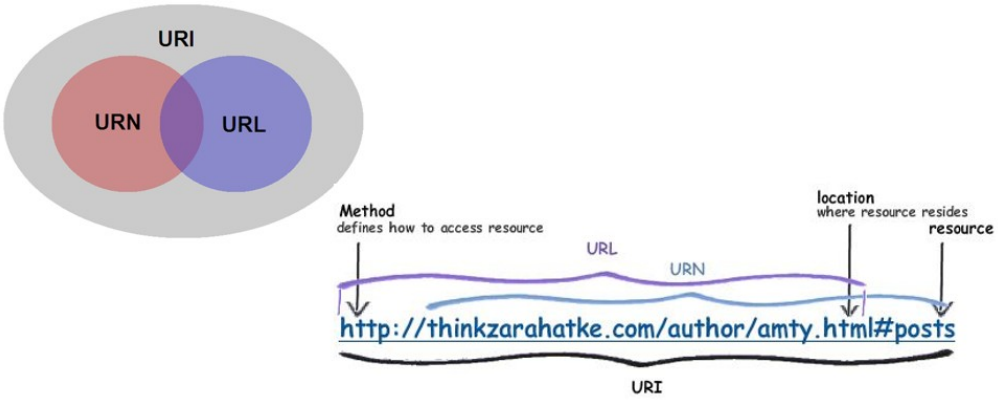
(Pitanja i odgovori za usmeni ispit 2024/2025)

1. Šta je web aplikacija i koji su osnovni koncepti?

- **Veb aplikacija** je aplikacija kojoj korisnici pristupaju preko mreže (Internet, Intranet), ona sadrži statički sadržaj ali može da generiše i dinamički sadržaj.

- **Osnovni koncepti web aplikacije:**

- **HTTP** - ***stateless*** protokol za razmenu informacija na veb-u koji funkcioniše po principu zahteva i odgovora
- **Veb server –** softverska komponenta koja obrađuje HTTP zahteve i prosleđuje HTTP odgovore, to je uglavnom računar na kome je instaliran odgovarajući softver, ima svoju IP adresu i može imati i domain name (URL adresa), *funkcije* su mu: skladištenje, obrada i dostavljanje veb stranica klijentu, primanje sadržaja od klijenta i interpretacija delova URL adrese (path, query)
- **URL –** pomoću URL adrese klijent inicira komunikaciju sa veb serverom, preko njega se takođe vrši identifikacija veb servera, razlikujemo:
    - **URI** (Uniform Resource Identifier) – sekvenca karaktera koja na jedinstven način identifikuje apstraktni ili fizički resurs (primer: files.hp.com), URI može biti URL, URN ili oba
    - **URL** (Uniform Resource Locator) – pored identifikacije resursa obezbeđuje i njegovo pronalaženje, opisuje način pristupa, mora biti naveden protokol za pristup (http, ftp...)
    - **URN** (Uniform Resource Name) – definiše identitet resursa na osnovu imena



http:// je protokol, nesto.com je domen, iza toga je path, posle ? je querz string i iza # je fragment

- **Web Application Framework –** okruženje za razvoj dinamičkih veb sajtova, aplikacija i servisa, primeri: spring-java, angular – typescript, django – python, play – Scala i Java, Laravel – PHP...

- Osnovu izvršavanja veb aplikacije čini komunikacija preko HTTP protokola, osnovna komunikacija se odvija između klijenata i veb servera koji se nalazi na nekoj URL adresi.

- Razlika između veb servera i aplikativnog servera odlikuje se u tome što su veb serveri jednostavniji (primaju zahtev i prosleđuju ga komponenti koja treba da ga obradi), to je samo okruženje u kojima rade serverski programi, drugim rečima, dok veb serveri obrađuju HTTP zahteve, aplikativni serveri obezbeđuju poslovnu logiku aplikacije, podržavaju različite protokole i funkcionalnosti se najčešće nude preko API-ja (npr. EJB, Enterprise Java Beans), *aplikativni server može da bude i veb server ali obrnuto ne može*. Neke poznatije implementacije veb servera uključuju Apache HTTP Server, Apache Tomcat i Jetty, dok za aplikativni server imamo GlassFish, JBoss (Wildfly) i Apache TomEE.

2. HTTP protokol (uopšteno, metodi, ...)

- **HTTP (Hypertext Transfer Protocol)** je ***stateless*** protokol (protokol u kome primalac ne sme da zadrži stanje sesije iz prethodnih zahteva, svaki zahtev se može razumeti izolovano) za razmenu informacija na veb-u koji funkcioniše po principu zahteva i odgovora.

- Http klijent (najčešće veb browser) inicira prenos podataka nakon što se uspostavi TCP/IP veza sa udaljenim veb serverom na određenom portu (po default-u je to na portu 80). Dakle HTTP transakcija započinje zahtevom klijenta i završava se odgovorom servera.

1. Korisnik zahteva URL od browser-a
2. Browser šalje request poruku (zahtev)
3. Server mapira URL na neki fajl ili program u direktorijumu dokumenata
4. Server šalje response poruku (odgovor)
5. Browser formatira taj odgovor (response) i prikazuje ga

- HTTP zahtev (request) se sastoji od:

1. Definicije HTTP metoda, URL adrese, verzije protokola
2. Liste HTTP zaglavlja
3. Tela poruke

- HTTP odgovor (response) se sastoji od:

1. Verzije protokola, statusa i opisa
2. Liste HTTP zaglavlja
3. Tela poruke

- **HTTP metodi:**

1. **GET** – vraća podatke identifikovane putem URL
2. **HEAD –** slično GET samo vraća HTTP zaglavlja, koristi se radi utvrđivanja da li je link još uvek aktivan ili je izmenjen u odnosu na prethodno stanje
3. **POST** – koristi se za slanje podataka web serveru
4. **OPTIONS** – ispitivanje mogućnosti web servera
5. **PUT** – koristi se za ažuriranje podataka iz tela poruke na lokaciju identifikovanu URL-om
6. **DELETE –** briše dokument sa web servera identifikovan URL-om
7. **TRACE** – služi za praćenje zahteva kroz firewall i proxy servere, radi pronalaženja problema na mreži

- HTTP status kodovi:

1XX – informativni (informational)

2XX – uspešni (success) (200 OK, standardni odgovor za uspešno izvršen zahtev)

3XX – redirekcioni (redirection) (301, resurs je trajno pomeren)

4XX – klijentska greška (client error) (401 neautorizovan pristup, 403 zabranjen pristup, 404 resurs nije pronađen)

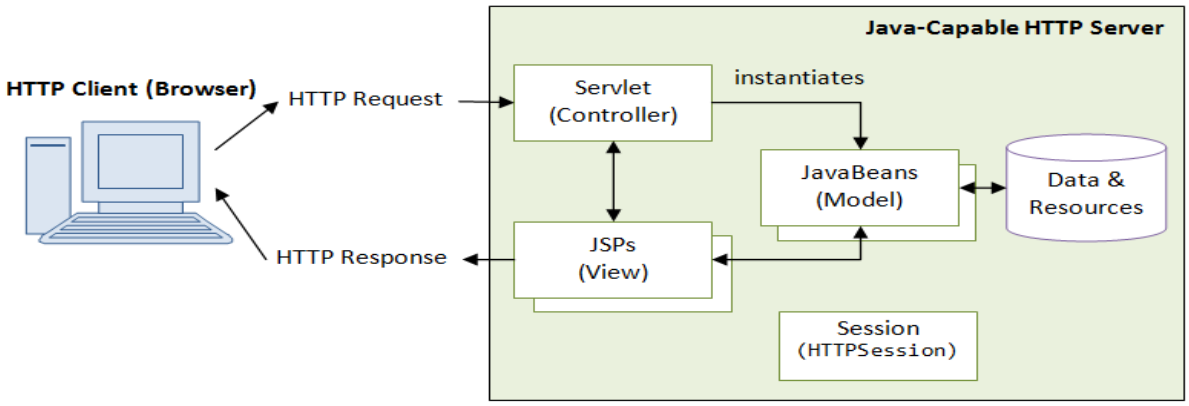
5XX – serverska greška (server error) (500 interna server greška, 503 servis nedostupan)

- HTTPS predstavlja protokol za sigurnu komunikaciju na mreži (podrazumevani port 443), može biti kombinovan sa drugim protokolima (SSL – secure sockets layer, TLS – transport lazer security). Podržava autentifikaciju veb servera, enkripcija u oba smera i sprečava man-in-the-middle-attack (MITM) kada treće lice "upadne" u komunikaciju.

3. Java Servlet (osnovni koncepti, kontejner, životni vek, ...)

- Servleti su osnova svake veb aplikacije. Java servlet možemo definisati kao java klasu koja se izvršava na veb serveru. Java Servleti se zasnivaju na HTTP protokolu, obrađuju HTTP zahtev klijenta i generišu HTTP odgovor.

- Servlet najčešće predstavlja srednji sloj aplikacije i kao takav može da prosledi zahtev donjem sloju aplikacije kao što je na primer baza podataka.



<u>Servlet Container</u>

- Servlet container je deo HTTP veb servera koji upravlja objektom Servlet-a, on upravlja životnim vekom Servlet objekta.

- Funkcionalnost Servlet Container-a je:

- Kreiranje instance Servlet klase
- Prihvatanje zahteva klijenta
- Prosleđivanje zahteva Servlet objektu
- Prosleđivanje rezultata klijentu (rezultat može biti vraćen u formatu TEXT/HTML, XML, PLAIN, IMAGE/JPG ili binarni format)

- Implementacija Java Servleta – Java Servlet klasa nasleđuje apstraktnu klasu **HttpServlet(doGet, doPost...) k**oja nasleđuje drugu apstraktnu klasu GenericServlet a ona implementira tri interfejsa (jakarta.servlet.Servlet, jakarta.servlet.ServletConfig i jakarta.io.Serializable)

public interface Servlet:

- **<u>Životni vek Servleta</u>** se sastoji od poziva sledećih metoda:

- init() – inicijalizacija
- service() – odgovor na zahtev klijenta
- destroy() – uništavanje servleta

- Metode za pristup konfiguraciji su: getServletConfig() i getServletInfo()

public interface ServletConfig:

- **<u>Najčešće korišćene metode:</u>**

- getInitParameter(String name) – vraća vrednost parametra na osnovu imena parametra
- getServletContext() – vraća referencu na ServletContext u okviru Web aplikacije
- getInitParameterNames() – vraća listu imena parametara za inicijalizaciju

public abstract class GenericServlet:

- Nezavisna od protokola, implementira neke metode iz interfejsa Servlet kao što su init() i getServletContext(), kao i neke pomoćne metode za dobijanje informaciji o konfiguraciji servleta.

public abstract class HttpServlet:

- Klasa servleta treba da implementira barem jedan od sledećih metoda:

- **doGet()** – za HTTP get zahteve, get se koristi za preuzimanje podataka ili slika sa Web servera
- **doPost()** – za HTTP post zahteve, post se najčešće koristi za slanje podataka sa ekranske forme Web serveru (get se nekad može koristiti za slanje malih podataka ali ako je sadržaj veliki koristi se post)

- **Interfejs Serializable** omogućava implementaciju mehanizma za praćenje sesija

public abstract interface HttpServletRequest

- Objekat koji predstavlja HTTP zahtev dobijen od klijenta.

- Metode:

- String getParameter(String name) – vraća vrednost parametra GET ili POST zahteva
- HttpSession getSession(Boolean create) – vraća objekat HttpSession koji mogu da koriste sve Web komponente u okviru tekuće sesije klijenta
- getParameters() – vraća nazive svih parametara
- getContentType()
- getCookies()

public abstract interface HttpServletResponse

- Objekat koji predstavlja HTTP odgovor klijentu.

- Metode:

- setContextType(String type) – govori klijentskom browser-u koji je tip odgovora (text/html, text/xml...)
- PrintWriter getWriter() – vraća output stream za slanja tekstualnih podataka klijentu
- ServletOutputStream getOutputStream() – vraća stream za slanje binarnih podataka klijentu (slike, muzika)
- sendRedirect() – služi i za preusmeravanje zahteva na drugi URL

- Metode init() i destroy() su metode interfejsa Servlet i one se pozivaju samo jednom. Obično ne rade ništa po default-u ali se mogu redefinisati da na primer otvorimo/zatvorimo konekciju nad bazom podataka.

init() – kada web server prvi put pokreće servlet, pre nego što servlet može da odgovori na HTTP zahtev

destroy() – kada se web server stopira ili servlet skloni sa servera

- Postoje dva načina da se pozove servlet: 1) u okviru web.xml fajla, 2) pomoću anotacije @WebServlet (češće)
- Tipičan način za poziv HTTP Servlet-a od strane veb klijenta je korišćenjem HTTP forme (u okviru atributa action HTML elementa <form> specifira se Servlet koji se poziva, default tip je get) i još jedan način e preko hyperlink-a.

- POST zahtev se poziva kucanjem URL adrese ili preko linka a GET zahtev se poziva kucanjem URL adrese (http://<host>/<servletName>?<parameterName>=<parameterValue>, ako ima više parametara razdvajaju se znakom “&”, znak “?” predstavlja početak tzv. query stringa)

4. Šta je JSP?

- JSP (Java Server Pages) predstavlja deo Java EE platforme, to je tehnologija za razvoj Java veb aplikacija i proširenje Java Servlet tehnologije. JSP fajl ima ekstenziju jsp.

- JSP tehnologija omogućava jednostavno i brzo kreiranje dinamičnog veb sadržaja. Osnovna zamisao je ugradnja Java koda u HTML stranicu. (html kod je bio unutar tagova **<%=** html kod primer **%>**), kasnije je ovo zamenjeno sa jsp akcijama, bibliotekama tagova i expression jezikom.

- Problem sa Java Servlet tehnologijom je to što su dizajn stranica (HTML) i programska obrada (Java) pomešani u istim datotekama pa zato dizajn može biti veoma komplikovan i teško je razdvojiti funkcije dizajnera i programera. Takođe je lakše menjati HTML nego upravljati velikim brojem println naredbi.

- JSP stranica se prevodi u Java Servlet prvi put kada se pozove.

JSP realizacija

Dobijeni servlet se kompajlira i obrađuje zahteve JSP stranice, rezultat njegovog izvršavanja tj. obrade zahteva je rezultat izvršavanja JSP stranice. Kod sledećih poziva iste stranice, veb server poziva odgovarajući servlet (ne kompajlira iznova jsp stranicu). Generisanje servleta, njegovo kompajliranje i pozivanje je zadatak servlet kontejnera.

JSP naspram drugih tehnologija

- Slična je sa idejom PHP i ASP (Active Server Pages).

- Za razliku od statičke HTML stranice ima mogućnost generisanja dinamičkih sadržaja.

- **JavaScript se izvršava u browser-u dok se JSP renderuje u html na serveru**

- Potrebna klasa za kombinovanje JSP i Servleta je RequestDispatcher i metoda forward.

- Elementi JSP stranice:

1. Skripting elementi
2. Implicitni objekti (predefinisane promenljive)
3. JSP akcije: alternativa za skriptlete, izraze i deklaracije
4. EL – expression language
5. Dodatni tagovi – JSTL biblioteka (Java Strandard Tag Library)

5. Skripting elementi (direktive, deklaracije, izrazi, skriptleti)

- Skripting elementi se pišu unutar znakova <% %>, neki elementi imaju i svoju xml reprezentaciju (bez znakova <% %>), na primer: <jsp:expressions> izraz </jsp:expresions>

- Postoji 5 različitih skripting elemenata:

komentar	**<%-- komentar --%>**
direktiva	**<%@ direktiva %>**
deklaracija	**<%! deklaracija %>**
skriptlet	**<% skriptlet %>**
izraz	**<%= izraz %**

Direktive

- JSP direktive se koriste za import biblioteke tagova ili Java paketa i klasa, obradu grešaka, uključivanje drugih stranica.

- Sintaksa: **<%@ directiveType {attribute="value"}%>**

- Postoje standardne jsp direktive:

<%@ page ... %>	definiše osobine vezane za obradu JSP stranice
<%@ include ... %>	uključivanje drugog fajla u JSP stranicu
<%@ taglib ... %>	import biblioteke tagova

- **Page**:
  - Obično prvi element JSP stranice, predstavlja velik broj atributa kojima se specificiraju različiti parametri za obradu JSP stranice
  - Atributi:
    - import – uvoz java klase ili paketa <%@ page import="java.util.Date" %>
    - contentType – definiše format rezultujuće strane
    - isThreadSafe="true/false" – true: višestruki zahtevi se obrađuju istovremeno od jedne instance servleta, false: zahtevi se obrađuju jedan po jedan
    - errorPage="URL" – url strane kojoj se na obradu prosleđuju izuzeci
    - isErrorPage="true/false" – definiše da li je data strana errorpage za neku drugu JSP stranu
    - pageEncoding="UTF-8" – kodni raspored JSP stranice
- **Include:**
  - Omogućava umetanje sadržaja drugih datoteka (HTML stranica, tekstualni fajl ili JSP stranica) u trenutku prevođenja JSP strane u servlet.
  - Sintaksa: <%@ include file="url"%>
- **Taglib:**
  - Omogućava proširenje skupa standardnih tagova sa dodatnom bibliotekom tagova
  - Sintaksa: <%@ taglib uri="taglibLibraryURI" prefix="tagPrefix" %>
  - Primer: JSTL biblioteka specijalnih tagova <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

Deklaracija

- Omogućava definisanje metoda ili atributa koji se pri prevođenju JSP strane umeću u telo servlet klase.

- Deklaracije ne proizvode nikakav izlaz.

- Sadržaj deklaracije je validan Java izraz ili više Java izraza razdvojenih ";".

- Sintaksa: **<%! Java code %>** primer: <%! int i=0; Double d %>

JSP izraz

- Rezultat izvršavanja se pretvara u String i umeće u rezultujuću stranu. Ne stavlja se ; na kraju

- Sintaksa: **<%= izraz %>**

- Primer: <%= (2*5) %> ili recimo <%= username %> ...

Skriptlet

- Omogućavaju umetanje Java koda u JSP stranicu, java kod skriptleta se izvršava kada JSP stranica obrađuje zahtev (u servletu se stavlja u _jspService() metod koji obrađuje zahtev).

- Sintaksa: **<% Java code %>**

6. Implicitni objekti

- Odnose se na neke najčešće korišćene koncepte u veb tehnologijama npr. pristup određenom opsegu vidljivosti.

implicitni objekat	opis
request	objekat tipa HttpServletRequest odnosi se na trenutni HTTP zahtev
response	objekat tipa HttpServletRequest odnosi se na tekući HTTP odgovor koji se prosleđuje brauzeru
out	JspWriter objekat se odnosi na output stream HTTP odgovora
session	objekat klase HttpSession odnosi se na tekuću sesiju
application	objekat klase ServletContext odnosi se na celu veb aplikaciju
page	objekat ekvivalentan promenljivoj this u Javi i koristi se za poziv metoda servleta koji nastaje na osnovu ove JSP stranice

0

- Generalna preporuka je da se ne koriste izrazi, skriptlete i deklaracije tj. da se Java kod ne ugrađuje u JSP stranicu jer je teško za održavanje, nečitko, HTML dizajn bude pomešan sa Java kodom i narušava se MVC patern (u skriptletama se može greškom naći nešto što pripada kontroleru). Alternativa su JSP akcije, JSTL i Expression language (EL).

(pročitati još o implicitnim objektima u narednom pitanju)

7. Expression language i JSTL biblioteka

Expression language

- Dodat je sa ciljem da se izbace skriptleti iz JSP stranica.

- Sintaksa: **${expr}** expr je neki validan EL izraz

- Može se koristiti na dva načina:

1. Kao vrednost atributa u tagovima <jsp:include page="${location}">
2. Kao ispis u HTML stranici <h1>Welcome ${name}</h1>

- EL može koristiti **implicitne objekte** za pristup promenljivama iz opsega.

Pristup promeljivima u različitom opsegu vidljivosti: (tim redosledom se i traži objekat)

- pageScope
- requestScope
- sessionScope
- applicationScope

Pristup parametrima

- param
- paramValues - niz stringova koji sadrži sve HTTP request parametre

Pristup podacima iz zaglavlja HTTP zahteva – header

- Ne moraju se koristiti implicitni objekti za pristup promenljivima u opsegu, npr. dovoljno je ${user} ne mora ${requestScope.user}

- Moguća je navigacija kroz povezane objekte, samo treba paziti na lenjo učitavanje ako je objekat dobijen iz baze, npr. ${user.fakultet.address}

EL sintaksa

- Pristup kolekcijama:

- ${niz[indeks_ili_ključ]}
- ${primerak.zaduzenjes[0]} - vraća prvo zaduženje iz liste zaduzenjes

- Provera uslova:

- ${test? opt1, opt2} – ako je test true prikazuje se opt1 u suprotnom je opt2

- Podržava standardne aritmetičke operacije i logičke operatore.

JSTL (Java Standard Tag Library) biblioteka

- Standardna biblioteka tagova koja se koristi za implementaciju dinamičkih elemenata u JSP stranici, zamenjuje skriptlete.

- Potrebno je dodati biblioteke jstl.jar na path

JSTL grupe tagova:

1. **Osnovni tagovi (Core tags**) – za implementaciju petlji, grananja, postavljanja promenljivih u neki opseg vidljivosti, potrebno je uvesti biblioteku u JSP stranicu pomoću taglib direktive, primeri definisanih tagova:

<c:out>	zamena za JSP izraze
<c:set>	postavlja vrednost neke promenljive u određenom opsegu
<c:remove>	uklanja vrednost iz opsega
<c:if>	uslovno grananje
<c:forEach>	definisanje petlje
<c:redirect>	redirektuje odgovor na navedeni URL

2. **Tagovi za formatiranje** – Potrebno je uvesti biblioteku *fmt*

<fmt:formatNumber>	formatiranje brojeva, valuta, procenata
<fmt:parseNumber>	parsiranje stringa i pretvaranje u broj
<fmt:formatDate>	datum pretvara u String reprezentaciju
<fmt:parseDate>	- parsira String i pretvara u datum
...	

3. **SQL tagovi**
4. **XML tagovi**
5. **JSTL funkcije** – Funkcije za manipulaciju stringovima, uvodi se biblioteka *fn*

fn:contains()	fn:toLowerCase()
fn:endsWith()	fn:toUpperCase()
fn:indexOf()	fn:substringBefore()
fn:split()	fn:substringAfter()
fn:trim()	fn:replace()

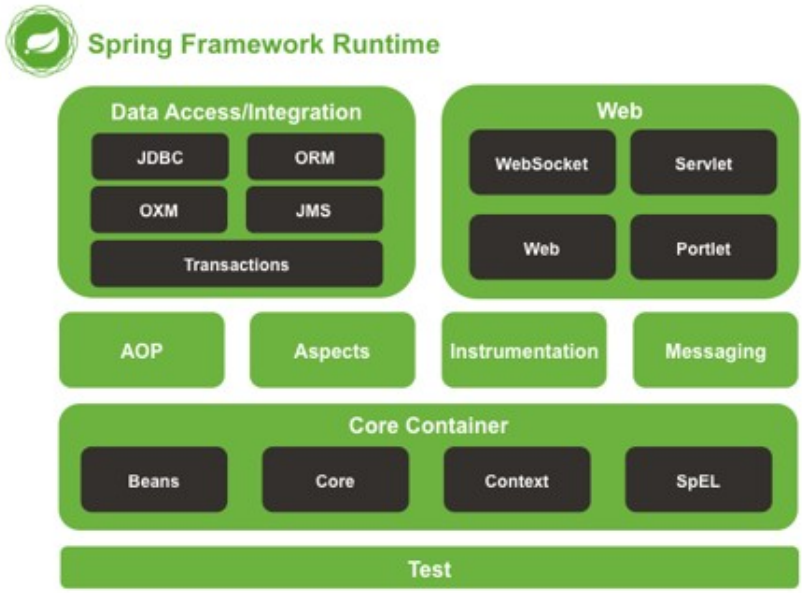
- Tipičan patern za komunikaciju servleta i jsp je da servlet bude zadužen:

    1. Za primanje HTTP zahteva
    2. Implementaciju poslovne logike
    3. Prosleđivanje modela jsp stranici koja će znati da prikaže taj model

    (Potrebna klasa RequestDispatcher i metoda forward)

Nešto malo o Springu:

- Spring je Java open source framework kreiran 2003. godine.
- To nije samo jedna biblioteka već skup različitih biblioteka različitih namena.



- Osnovna ideja Springa je da pojednostavi razvoj JEE aplikacija koje su lake za održavanje, testiranje, dobro struktuirane.
- Može se koristiti za razvoj standalone Java aplikacija ali i web aplikacija.
- Zasnovan je na principu Dependency Injection i upotrebe anotacija.
- **Prednosti:**
    - Primenjuje templejt patern za JDBC, Hibernate, JPA, čime se sakrivaju detalji i smanjuje količina koda koji se piše
    - Poštuje princip Loose Coupling zbog primene Dependency Injection paterna
    - Primena Dependency Injection paterna olakšava razvoj i testiranje aplikacije
    - Koristi POJO klase koje ne moraju da nasleđuju posebne interfejse
    - Omogućava dodavanje nefunkcionalnih zahteva kao što su keširanje, rad sa transakcijama, formatiranjem validacija pomoću anotacija…
    - Spring pruža API za JavaEE specifikacije

8. Alati za pakovanje aplikacija – Maven

- Osnovne komponente Building Tool-ova su build skripte i executable (program koji se izvršava). (***Maven***,*Ant*,*Ivy*,***Gradle***)
- Neke od operacija Building Tool-ova:

    - Kompajlirane izvornog koda
    - Kopiranje resursa
    - Kompajliranje i izvršavanje testova
    - Pakovanje projekata (jar, war)
    - Deploy
    - Brisanje nepotrebnih resursa
    - Upravljanje povezanim bibliotekama i projektima – dependencies

- **POM** (Project Object Model) je XML reprezentacija Maven projekta. Svaki Maven projekat sadrži pom.xml fajl. U njemu su specificirani svi artifakti vezani za projekat (plugins, goals, build profil, dodatne biblioteke…).

- U njemu imamo

1. **modelVersion** – verzija Maven-a po kojoj je kreiran pom.xml, obavezno se navodi
2. **groupId, projectId** – jedinstveno identifikuju maven projekat u okviru neke organizacije
3. **version** – tokom vremena projekat može imati više verzija
4. **packaging –** definiše tip artefakta koji će se kreirati na osnovu projekta, nije obavezan i podrazumevano je jar (može pom, jar, maven-plugin, war, rar…)

- Elementi **build** i **reporting**. U build je opisan proces kompajliranja i pakovanja aplikacije (definiše lokaciju resursa i upravlja plugin-ovima), a sa reporting se opisuje proces kreiranja dokumentacije za projekat.

- Podelementi build-a: sourceDirectory (sadrži izvorni kod aplikacije), defaultGoal (podrazumevana akcija koja će se izvršiti), resources (koji resursi se koriste u procesu bildovanja), finalName (naziv kreiranog resursa), resource (opisuje specifičnu grupu resursa), directory (u njih se snimaju kreirani resursi), targetPath (specifira strukturu direktorijuma u javi – jar, war), includes (skup file pattern-a koji će biti uključeni), excludes (skup file pattern-a koji će biti preskočeni).

- **Maven plugin:** Svaki zadatak se izvršava preko plugin-a. Dve vrste:

- **build plugins** – pakovanje i kreiranje izvršne verzije projekta, konfiguracija u <build> u pom.xml
  - clean - brisanje nepotrebnih resursa nakon kompajliranja/pakovanja aplikacije
  - compiler - kompajliranje java fajlova
  - deploy – „kopiranje" kreiranih artefekta na udaljeni repozitorijum
  - install - instaliranje kreiranih artefakta u lokalni repozitorijum
  - resources - kopiranje resursa u poseban direktorijum
  - ear, jar, war, rar - pluginovi za pakovanje projekta u arhivu određenog tipa
- **reporting plugins** – generisanje izveštaja/dokumentacije, konfiguracija u <reporting> u pom.xml
  - changelog - generiše listu poslednjih promena
  - javadoc - generisanje dokumentacije
  - project-info-reports - generisanje izveštaja za projekat

- **Plugin** element ima artefactId, version i configuration

- **Goal** predstavlja specifičan zadatak plugina koji je deo procesa pakovanja aplikacije. Primeri: generate-sources, install, clean, compile, test,... Postoje zavisnosti između ovih zadataka, izvršavanje jednog podrazumeva izvršavanje nekih drugih. (desni klik na pom.xml -> run as, i onda se pojave te opcije).

Maven definiše strukturu projekta:

target - direktorijum u koji se smešta rezultat pakovanja aplikacije
src - izvorni kod aplikacije
src/main - sav izvorni kod aplikacije koji će biti upakovan u arhivu (ili osnovni artefakt)
src/test - izvorni kod koji se koristi za testiranje
src/main/java - java fajlovi koji se pakuju u glavni artefakt
src/main/webapp - resursi vezani za web aplikaciju
src/main/resources - fajlovi koji se ne kompajliraju
...

- Dependencies su druge biblioteke ili projekti koje koristimo za razvoj našeg programa. Jedan od osnovnih zadataka alata za pakovanje aplikacija je upravljanje tim dodatnim bibliotekama.

- **JAR HELL** je stanje u kom se nalazi aplikacija ako ima problema sa dependency-jima. Problemi koji mogu dovesti do toga:

1. Za rad nekog jar-a trebaju nam drugi jar fajlovi, da bi smo otkrili koji su to fajlovi čitamo dokumentaciju ili pokušavamo da eliminišemo jedan po jedan error
2. Na class-path-u se nalaze jarovi koji imaju iste klase, ili postoje dva ista jara ali različitih verzija. Prilikom učitavanja jar fajlova učitaće se onaj koji je prvi na redu a ostali će biti sakriveni (shadowing)
3. Verzije jar-ova nisu kompatibilne

- Maven rešenje za te probleme je lokalni repozitorijum biblioteka.

- Element **dependancy**:

- groupId, artifactId, version - jedinstvena identifikacija povezane biblioteke
- type - packaging type povezane biblioteke
- scope - u kom delu "života" softvera će se koristiti biblioteka (compile, runtime, test)
- optional – definiše da li je korišćenje biblioteka obavezno

Maven repozitorijum:
- Biblioteke specificirene u pom.xml-u kao dependencies preuzimaju se iz centralnog repozitorijuma preko http-a. Preuzete biblioteke se onda čuvaju u lokalnom repozitorijumu (${user.home}/.m2/repository), fajlovi su u njemu organizovani na sledeći način:

{groupId}/{artifactId}/{version}/{artifactId}-{version}.jar

Razlika između Maven i Gradle – Gradle ne koristi XML već Groovy jezik.

9. Principi Dependency Injection, IoC kontejner, Spell jezik

Dependency Injection (DI)

- Dependency Injection je princip koji omogućava odvajanje kreiranja objekata od njihove upotrebe.

- Klasa ne stvara objekte od kojih zavisi, već ih dobija (injektuje) spolja, takođe ne zavisi direktno od drugih klasa već samo od interfejsa ili apstrakcija (SOLID principi – Single responsibility princip, Open/closed princip, Liskov princip supstitucije, Interface segregation princip, Dependency inversion princip).

- Zavisnosti se mogu lako zameniti za test verzije ili mock objekte što omogućava lakše unit testiranje.

- Razlikujemo Constructor-based DI i Setter-based DI

- **Ideja** je da se objekti inicijalizuju u vremenu izvršavanja a ne u vremenu kompajliranja od strane kontejnera (primenom DI, objekte instancira Spring kontejner, dovoljno je samo naznačiti da će se određeni objekat inicijalizovati u vremenu izvršavanja). *Konkretne objekte u Springu povezujemo pomoću XML konfiguracionih fajlova ili određenih anotacija.*

Spring IoC kontejner

- Spring kontejner omogućava kreiranje objekata i dodavanje svih objekata sa kojima je povezan. Kreiranje objekata je prebačeno sa nivoa aplikacije na kontejner i otuda naziv IoC (Inversion of Control).

- Osnova Spring IoC kontejnera su paketi org.springframework.beans i org.springframework.context.

- Objekti koji se nalaze pod kontrolom Spring IoC kontejnera se nazivaju binovi (beans). Spring kontejner će upravljati binovima ali programer mora da definiše koje to binove ima u svojoj aplikaciji i kakva je njihova međusobna veza. Kontejner zna koje binove da kreira na osnovu metadata konfiguracije koja može biti u obliku XML, Java klasa ili anotacija. Prilikom kreiranja binova kontejner kreira objekte koji se zovu kao i njihove klase a ako postoji više binova iste klase u kontejneru potrebno je definisati koji bin se umeće i za to se koristi anotacija **@Qualifier** (u suprotnom se javlja greška NoUniqueBeanDefinitionException)

```
@Configuration
public class Config {

@Bean("john")
public Employee johnEmployee() {
      return new Employee("John");
}
```

```
public class EmployeeService {

      @Autowired
      @Qualifier("john")
      private Employee employee;
}
```

- ApplicationContext interfejs predstavlja IoC kontejner zadužen za instanciranje binova a ApplicationContext klasa predstavlja način za implementaciju instance Spring kontejnera, svaka Spring aplikacija mora imati bar jednu klasu koja implementira ApplicationContext. Postoje različite vrste ovih klasa: AnnotationConfigApplicationContext (učitava konfiguraciju za standalone Spring aplikaciju), FileSystemXmlApplicationContext (učitava konfiguraciju za standalone Spring aplikaciju iz XML fajlova u fajl sistem), AnnotationConfigWebApplicationContext (učitava konfiguraciju za web Spring aplikaciju iz Java konfiguracionih klasa), XmlWebApplicationContext (učitava konfiguraciju za web Spring aplikaciju iz XML fajlova).

- Klasa koja predstavlja konfiguraciju označena je sa anotacijom **@Configuration**.

- Spring konfiguracija sadrži definiciju bar jednog bina i za to se koristi anotacija **@Bean** koja se postavlja na metod

SpEL jezik (Spring Expression Language)

- Jezik koji se koristi unutar Spring konfiguracije kako bi omogućio evaluaciju izraza i manipulaciju podacima tokom razvoja aplikacija koje koriste Spring, može se koristiti sa XML konfiguracijama ili konfiguracijama koje koriste anotacije

- SpEL izrazi počinju sa simbolom # i navode se u okviru { } , npr. #{${someProperty} + 2}

- Ukoliko želimo nekoj promenljivoj da dodelimo vrednost SpEL izraza koristimo anotaciju **@Value**, pomoću SpEL izraza možemo pristupiti metodama I promenljivim koji su definisani u nekom binu.

```
@Component("workersHolder")
public class WorkersHolder {
      private List<String> workers = new LinkedList<>();
      private Map<String, Integer> salaryByWorkers = new
      HashMap<>();
      public WorkersHolder() {
            workers.add("John");
            workers.add("Susie");
            salaryByWorkers.put("John", 35000);
            salaryByWorkers.put("Susie", 47000);
      }
//Getters and setters
}
```

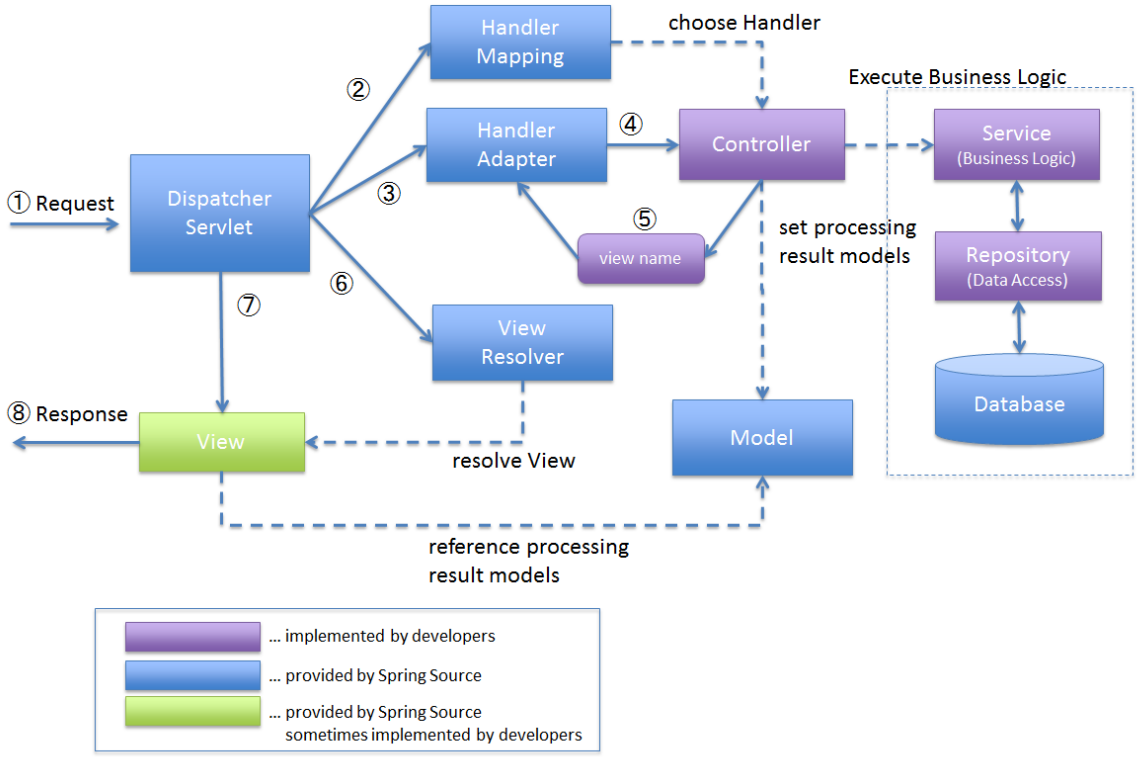
```
@Value("#{workersHolder.salaryByWorkers['John']}")
private Integer johnSalary;

@Value("#{workersHolder.workers[0]}") // John
private String firstWorker;
```

10. Arhitektura web aplikacije bazirane na Spring MVC

- Spring MVC je jedan od modula iz Spring frameworka-a koji je zasnovan na Servlet tehnologijama i arhitektonskom paternu Model-View-Controler.

- Svaka od komponenti je zadužena za određeni deo posla i na taj način se smanjuje zavisnost između njih.

- Centralna komponenta je DispetcherServlet (često se naziva front kontrolor). Zadužen je da prima sve HTTP zahteve od klijenta i rutira do određenog kontrolora. Kontrolori onda obrađuju zahtev i delegiraju posao drugim komponentama.



1: zahtev stiže do DispetcherServleta

2: aplikacija može da ima više kontrolera, Handler mapper komponente pomažu DispecherServletu da prepozna kome je zahtev upućen

3: DispetcherServlet sada zna kome treba da uputi zahtev i poziva odgovarajućeg kontrolora

4: Kontrolor obrađuje zahtev. Ako treba da vrati neke podatke klijentu pakuje ih u model i definiše logičko ime View komponente koja treba taj model da prikaže.
5: DispatcherServlet konsultuje komponentu View Resolver koja mu na osnovu logičkog imena View komponete da njenu stvarnu implementaciju

6: DispatcherServlet prosleđuje model View komponenti
7: View komponenta renderuje podatke i prikazuje ih korisniku u browseru

- Da bi mogli da razvijamo web aplikaciju pomoću Spring MVC potrebno je da konfigurišemo DispatcherServlet. To je moguće pomoću web.xml ili pomoću jave (programski, npr. nasleđivanjem klase AbstractAnnotationConfigDispatcherServletInitializer – time se automatski kreira odgovarajuća instanca klase ApplicationContext, i potrebno je redefinisati metode: **getServletMappings()** definiše URL adrese koje će osluškivati DispatcherServlet, **getServletConfigClasses()** vraća klase koje konfigurišu binove koji će se koristit u web aplikaciji (npr. kontrolori, view resolveri, handler mappere), **getRootConfigClasses()** učitava binove koji nisu deo web aplikacije).

- Najlakši način da se konfiguriše Spring MVC aplikacija je upotrebom **Spring Boot**-a.

- Spring Boot je wrapper oko Spring aplikacije i rešava probleme koji se pojavljuju u svim aplikacijama a programer se koncentriše samo na poslovnu logiku aplikacije

Prednosti:

- Eliminiše potrebu za konfiguriranjem
- Omogućava kreiranje profila koji sadrže konfiguracije za različita okruženja
- Rešava probleme sa dependency-jima →postoje starteri za različite tipove aplikacija
- **Embedovan Tomcat web server**
- Pokretanje web aplikacije kao jar fajla
- Command line interfejs za pokretanje Groovy skripti

- Spring Boot projekat može da se kreira na više načina:

- Kreiranjem maven projekta i dodavanjem dependencija (npr. spring-boot-starter-web)
- Korišćenjem Spring Tool alata iz Eclipsa
- Korišćenjem Spring Initializr na zvaničnom Spring sajtu

- Spring Boot aplikacija mora imati klasu sa anotacijom @SpringBootApplication, ta klasa ima main metod i pokretanjem te klase se pokreće aplikacija. Spring Boot automatski pokreće veb server Tomcat i aplikacija je dostupna na adresi http://localhost:8080/. Treba da se nalazi u root projekta da bi skenirala binove koji su ispod.

- Spring Initializr kreira fajl application.properties u kome se mogu dodati vrednosti za Spring Boot parametre (npr. port web servera, parametri konekcije za bazu…). Ne moramo ga učitavati. Umesto njega se često koristi application.yml fajl. *Postoji i anotacija @PropertySource koja omogućava da se definišu drugi konfiguracioni fajlovi.*

- Spring Boot Actuator je komponenta koja omogućava monitoring Spring Boot aplikacije u trenutku izvršavanja, omogućava da vidimo koji su binovi kreirani, koji su parametri podešeni itd. (spring-boot-starter-actuator u pom.xml). Kada ga dodamo postanu nam dostupni HTTP endpoint-ovi koji počinju sa /actuator, međutim moraju se postaviti u application_properties.

- Spring profili omogućavaju konfigurisanje određenih beanova u zavisnosti od toga šta je podešeno kao okruženje.

11. Spring kontrolor, redirekcija u Springu

- Kontrolori su zaduženi za obradu HTTP zahteva. To su klase koje imaju anotaciju @Contoller i čiji metodi imaju anotaciju @RequestMapping. Anotacija @Controller omogućava da Spring prepozna automatski ovu komponentu i da je inicijalizuje. (*zamenili Servlete*)

```
@Controller
public class HomeController {
    @RequestMapping(value="/", method =RequestMethod.GET)
   public String home(){
      return "home";
   }
}
```

Prosleđivanje parametara kontroleru:

- Anotacija @RequestMapping ima atribute:

  - value: definiše koji url patern treba da obradi metoda definisana u kontroloru
  - method: definiše koji tip HTTP metoda se obrađuje

- Umesto @RequestMapping možemo koristiti anotacije @GetMapping i @PostMaping. Ako se stavi iznad klase, time se definiše da je ta klasa zadužena za određeni URL, a ako se stavi iznad metoda tada detaljnije opisuju URL paterne na koje će te metode da reaguju.

```
@Controller
@RequestMapping(value ="/")
public class HomeController {

  @RequestMapping( value =”home” method =RequestMethod.GET)
   public String home() {.....

   @GetMapping (“wellcome”)
    public String wellcome(){ ……
```

- Metode u kontroloru mogu imati parametre tipa HTTPServletRequest, HTTPServletResponse, HTTPSession

```
@RequestMapping(value="enrol", method=RequestMethod.POST)
public String saveStudent(HttpServletRequest r){

   Student s= new Student(r.getParameter("name"),r.getParameter("surname"));
   sr.enrol(s);
   return "ok";
}
```

- Parametri u request objektu se mogu definisati i anotacijom @RequestParam koja ima atribut required kojim definišemo da li je parametar obavezan. Podrazumevana vrednost je true i doći će do greške ako metod ne dobije taj parametar.

```
@RequestMapping(value="enrol", method=RequestMethod.POST)

public String saveStudent(@RequestParam("name") String n,@RequestParam("surname")String sn){

        Student s= new Student(n,sn);
        sr.enrol(s);
         return "ok";
}
```

- Parametri se mogu prosleđivati i kroz url adresu kao Path varijable (npr. localhost:8080/helloSpring/students/get/147 je path varijabla, želimo da pozovemo kontrolorev method koji ima URL “/students/get”i parametar koji prosleđujemo je

147)

```
@RequestMapping(value="get/{noIndex}", method=RequestMethod.GET)

public String getStudent(@PathVariable("noIndex") String br, Model m){
    m.addAttribute("student", sr.getStudent(br));
    return "findStudent";
}
```

- Ako želimo da pročitamo vrednost koja se nalazi u headeru HTTP zahteva koristimo anotaciju **@RequestHeader**. Ako želimo da pročitamo vrednost koja se nalazi u cookijima HTTP zahteva koristimo anotaciju **@CookieValue**.

Ali to se sve koristi ako šaljemo mali broj parametara. Ako šaljemo veći broj parametara, onda je bolje poslati ceo objekat – potrebno je samo da imamo prazan konstruktor i da se polja u JSP stranici zovu isto kao polja u objektu, i to se naziva data-binding.

Redirekcija

- Ukoliko želimo da iz jednog metod kontrolora pozovemo drugi metod potrebno je samo da vratimo string koji počinje sa rečju redirect:

```
@RequestMapping(value = "delete/{numIndex}", method = RequestMethod.GET)
    public String deleteStudent(@PathVariable("numIndex") String br) {
        sr.delete(Integer.parseInt(br));
         return "redirect:/students/subscribed";
}
```

- Ali ako želimo da pošaljemo i neke podatke, ako ih stavimo u model oni neće biti vidljivi jer se sa redirekcijom kreira novi request a model je samo vidljiv u onom requestu u kom je kreiran. Zato imamo dve opcije onda:

1. koristiti URL templejt
    a. treba da poštujemo pravila u kreiranju redirect stringa i da sve što želimo da pošaljemo stavimo u model
    b.
```
@RequestMapping(value="/register", method=POST)
public String processRegistration(Student student, Model model) {
    sr.save(student);
    model.addAttribute("username", student.getUsername());
    return "redirect:/student/{username}";
}
```
2. koristiti Flash atribute
    a. kada želimo da prenosimo složenije podatke
    b. način da podatak privremeno smestimo u HTTP sesiju, ali podatak je vidljiv samo u sledećem requestu i Spring vodi računa da ga izbriše iz sesije
    c. Koristiti se metoda addFlushAttribute() nad modelom tipa RedirectAttributes podklasa klase Model

```
model.addFlashAttribute("student", s);
```

12. Controlor-Service-Repository patern

- U njemu:

- **Repository** je zadužen za perzistenciju podataka, tj za dobavljanje i čuvanje podataka u različitim tipovima skladišta
- Repozitorijumi imaju anotaciju @Repository, kako bi omogućili jednostavno instanciranje pomoću Dependency Injection tehnike
- Obično jedan repozitorijum je odgovoran za jedan entitet i operacije nad njim

- **Kontroler** ima anotaciju @Controllor i njegova uloga je da prima poruke od View komponente i da ih prosleđuje ka Servisu i obrnuto, da odgovor Servisa prosleđuje View komponentama.

- **Servis** je komponenta iznad repozitorijuma i ona koristi njegove usluge kao i usluge drugih servisa. Anotacija @Service, implementiraju interfejs. Servisi enkapsuliraju poslovnu logiku aplikacije. Operacija servisa može pozivati različite repozitorijume i ili će se sve uspešno izvršiti ili se ništa neće izvršiti.

13. Templejt patern

- Spring framework omogućava povezivanje aplikacije sa bazom podataka na različite načine:

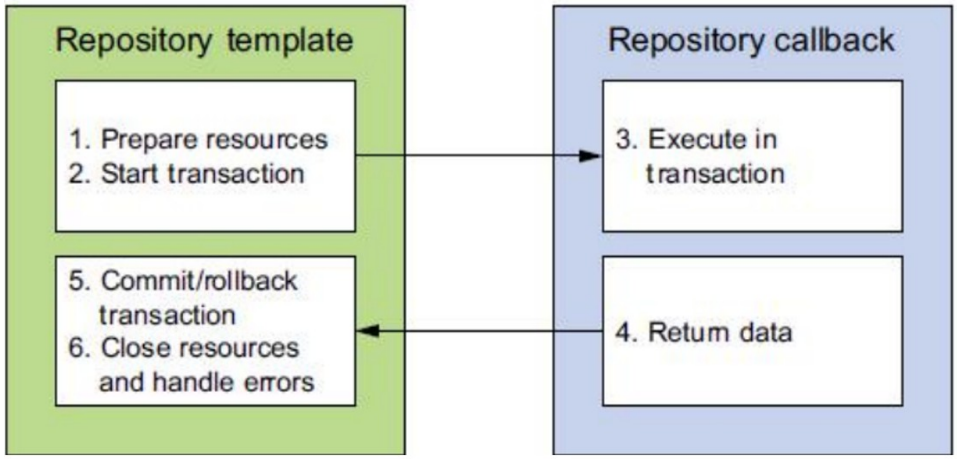
- preko JDBC konekcije
- primenom ORM frameworka (JPA, Hibernate…)
- primenom Spring Data modula

- Ideja templejt paterna je da su određene aktivnosti za rad sa bazom podataka uvek iste i da se menjaju samo neke specifične aktivnosti koje zavise od izabrane tehnologije

- uvek se mora otvoriti konekcija bez obzira koja tehnologija se korisiti
- bazi se postavljaju upiti ali ti upiti su specifični za tehnologiju

- Spring koristi

- template klase kako bi definisao fiksne aktivnosti
- callback klase kako bi definisao aktivnosti koje su specifične



Spring JDBC Template sakriva sve one aktivnosti koje bi inače morali da izvršimo kada koristimo JDBC za konekciju sa bazom. Da bi mogli da ga koristimo moramo definisati Datasource tj. da u application.properties unesemo parametre za konekciju na bazu podataka (url, username, password).

Preporuka je da se koristi connection pool jer on definiše maksimalan broj konekcija u jednom trenutku i vodi računa da vrati konekciju koja je slobodna. Spring po default-u koristi HikariCP pooling, moguće ga je podesiti u application.properties.

Upotreba JDBC Template: Potrebno je u klasi označenoj sa anotacijom @Repository, @Controller ili @Service kreirati objekat klase JdbcTemplate i označiti ga anotacijom **@Autowired,** potrebno je implementirati interfejs RowMapper kako bi se rezultati iz baze mapirali na Java klase. Na metod ili na celu klasu dodati anotaciju @Transactional kako bi Spring kontejner vodio računa o rollback operaciji ukoliko dođe do greške.

- JdbcTemplate klasa ima operacije:

- **query** – omogućava dobavljanje podataka iz baze
- **update** - omogućava insert, update i delete podataka

14. Spring data

- Spring framework nudi dodatan modul SpringData koji je wrapper oko JPA. Ideja je da Spring na osnovu imena operacije prepozna šta treba da se uradi. Potrebno je da naš repozitorijum nasledi klasu JpaRepository <T, I> da bismo radili sa JPA. *(T je tip entiteta a I je tip ključa u entitetu).*

- Potrebno je dodati dependencije (spring-boot-starter-data-jpa i konektor za bazu).

- Time što nasleđuje JpaRepository kreirani interfejs automatski može da koristi metode:

- findAll():List<T>
- findOne(I)
- count()
- delete(T)
- deleteAll()
- save(T)
- exists(I) …..

- Spring Data prepoznaje imena metode koje imaju imena u skladu sa definisanim paternima i izvršava ih, bez da mi te metode moramo implementirati. Stavimo read/find/get/count onda BY i onda deo koji treba izvršiti.

- Npr: public List<Student> findByPrezimeOrderByImeAsc(String p) ili recimo findByImeOrPrezimeIgnoringCase(String ime, String p)…

Možemo definisati i svoju metodu tako što stavimo @Query i definišemo JPQL naredbu:

```
@Query("select s from Student s where s.email like '%gmail.com'")
public List<Student> findAllGmailStudents();
```

Može i sa parametrima: (mogu i imenovani parametri)

```
@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")
User findUserByStatusAndName(Integer status, String name)
```

Ako želimo da koristimo update moramo iznad @Query staviti **@Modifying** i tada je povratna vrednost metode broj redova u tabeli koji su ažurirani.

Kombinacija Spring Data i EntityManager-a

- Ukoliko neki upit ne može da se definiše kroz konvenciju imena ili kroz Query anotaciju može se koristiti standardni način za izvršavanje upita preko EntityManagera. Potrebno je kreirati klasu koje će imati sufiks Impl a početak imena je isti i kao interfejs koji nasledjuje JpaRepository.

- U kreiranoj klasi implementiramo određeni metod koji smo definisali u interfejsu i na taj način smo zadržali i mogućnosti JpaRepository klase plus smo dodali nove metode.

- Definišemo interfejs u kome definišemo specifičan metod

```
public interface StudentReportRepository {
        public List <Object[]> groupByEmail();
}
```

- Dodamo da Spring JPA Data repozitorijum nasleđuje taj interfejs

```
@Repository
@Transactional
public interface StudentRepository extends JpaRepository<Student, Integer>, StudentReportRepository{

}
```

- Implementiramo interfejs i klasa se zove kao i Spring Data repozitori plus dodamo sufiks Impl

```
@Repository
public class StudentRepositoryImpl implements StudentReportRepository {
  @PersistenceContext
   private EntityManager em;

   public List <Object[]> groupByEmail() {
        String query="select substring(s.email,instr(s.email,'@')+1),count(*)from  Student s
                                    group by substring(s.email,instr(s.email,'@'))";
         List <Object[]> res=em.createQuery(query).getResultList();
          return res;
   }
}
```

Dinamički upiti: ako parametri zavise od korisničkog unosa, tada struktura upita može biti promenljiva. Tada korisimo JPA Criteria API u kastomizovanom repozitorijumu.

15. Validacija input polja na formi

- Validacija predstavlja proveru da li objekti koji su stigli do kontrolora zadovoljavaju određene uslove.

- Dva načina za validaciju:

1. upotreba Java Bean Validator specifikacije - implementirano kroz Hibernate Validator
2. upotreba org.springframework.validation.Validator interfejsa

Java Bean Validation API

- U kontroloru koji prima objekat koji hoćemo da validiramo, dodati anotaciju **@Valid** kako bi naznačili da se primenjuje validacija. Ako validacija ne prođe Spring javlja grešku 400 Bad request.

- Kontrolor može imati još jedan parametar tipa BindingResult ili Errors koji će služiti za obradu grešaka pri validaciji.

```
public class Student {

@NotNull
@Size(min=2, max=30)
String name;

@NotNull
String surname;

@Min(0)
int noIndex;
.....
}
```

```
@RequestMapping(value="enrol", method=RequestMethod.POST)
 public String saveStudent(@Valid Student s,Errors e ){
      if (e.hasErrors()) {
          return "save";
      }
       sr.enrol(s);
       return "ok";
}
```

Neke anotacije su

- @Size - definiše dužinu karaktera u input polju
- @Past - proverava da li je datum manji ili jednak sa tekućim u sistem
- @Future - - proverava da li je datum veći ili jednak sa tekućim u sistem
- @NotNull - vrednost nije null
- @NotBlank - vrednost nije prazna
- @Digit - vrednost mora biti broj
- @Max, @Min - vrednost je broj veći/ manji od zadatog broja
- @Pattern - omogućava definisanje regularnog izraza koji mora da važi

- Moguće je validirati i Path varijablu ili Query parametar koji su u HTTP GET metodi i u tom slučaju stavljamo **@Validated** na kontrolor i validaciju na sam parametar:

```
@Controller
@Validated
public class TestController {

   @GetMapping("get/{id}")
   public Student getStudent( @PathVariable("id") @Min(5) int id) {
            .......
  }
}
```

Implementacija korisnički definisanog validatora

- Ovaj proces uključuje kreiranje:

- korisničkog validacionog interfejsa - klase koja implementira ConstraintValidator
- anotacije koja označava gde će se ta validacija koristiti

➢ Potrebno je definisati sopstvenu anotaciju koju ćemo staviti iznad atributa koji treba da je primeni.

```
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = IpAddressValidator.class)
public @interface IpAddress {
   String message() default "{IpAddress.invalid}";
   Class<?>[] groups() default { };
   Class<? extends Payload>[] payload() default { };
}
```

➢ Potrebno je implementirati Validator koji implementira ConstraintValidator interfejs

```
public class IpAddressValidator implements ConstraintValidator<IpAddress, String> {

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
         bollean valid
       // logika provere da li je nešto validno

      return valid:
  }

}
```

- Onda te naše anotacije možemo koristiti kao i sve ostale.

16. Spring Converter i Spring Data binding

Spring Data binding

- Koncept koji se odnosi na mapiranje elemenata HTML forme na Java objekat. Koristimo ga kada imamo objekat koji se sastoji od drugih objekata (pošto po default Spring zna da mapira samo proste tipove). Koristi se u kombinaciji sa Spring form jsp tagovima.

- Moguće ga je implementirati kroz:

- InitBinder anotaciju (@InitBinder)
    - Postavlja se na metod koji nema povratnu vrednost i omogućava mapiranje HTTP parametara na model objekat
    - Njome se instancira objekat klase WebDataBinder na koji se registruju konverteri i formateri
- ModelAttribute anotaciju (@ModelAttribute)
    - Može se staviti na metod ili na parametar metode
    - Omogućava da se izvrši inicijalizacija java objekta, to će se desiti kada se prvi put pozove kontrolor koji ima metod sa ovom anotacijom
    - Objekat je vidljiv samo u kontroloru u kome je nastao, ako želimo da ga ostali vide, kreiramo klasu sa @ControllerAdvice i u njoj definišemo @ModelAttribute metode
    - Objekat će biti povezan sa formom pomoću Spring JSP form taga i modelAttribute

@SessionAtributes – koristimo je na nivou klase, kada želimo da objekat bude vidljiv posle nekoliko HTTP zahteva
- Kada se prvi put pozove metod sa @ModelAtribute, objekat će se staviti u sesiju ali je objekat vidljiv na nivou tog kontrolera.
- Za brisanje objekta iz sesije koristi se metod klase SessionStatus.

Spring Converter

- Konverteri su klase koje omogućava mapiranje jednog tipa podataka na drugi. Spring ima ugrađene konvertere npr. koji pretvaraju String u Integer, ali češća je varijanta da nam trebaju naši konverteri npr. String u Date

- Potrebno je da kreiramo klasu koja implementira interfejs Converter<S, T> gde je S tip iz kog konvertujemo a T tip u koji konvertujemo
- Ovu klasu treba anotirati sa anotacijom @Component kako bi je Spring kontejner registrovao
- Metod u kontroloru će automatski pokušati da izvrši ovu konverziju ako dobije odgovarajući tip

17. Asinhrono i automatsko izvršavanje servisa
- **Automatsko** izvršavanje operacije podrazumeva da ne mora postojati direktan poziv te operacije od strane klijenta, već se operacija samostalno izvršava kada se desi definisani trenutak u vremenu. Da bi ga omogućili, main klasa Spring projekta mora da ima anotaciju **@EnableScheduling**, a metod koji se automatski poziva mora imati anotaciju **@Scheduled** i ne može imati parametre.

- Anotacija @Scheduled može imati parametre koji definišu kada počinje izvršavanje neke operacije nakon što se web aplikacija deploy-jue i koliko često se izvršava (svaki minut, sat…).

Primer

```
@Service
public class NotificationService {

 @Scheduled(fixedRate = 2000)
 public void sendNotificationEveryTwoSeconds() {

     System.out.println(new Date(System.currentTimeMillis()));
     System.out.println("Ovde ide implementacija slanja mejla");
   }

}
```

- Često se koristi sa cron izrazima koji se sastoje od 6 oznaka koje predstavljaju <second> <minute> <hour> <day of month> <month> <day of week>

```
@Scheduled(cron = "0 15 10 15 * ?")
public void scheduleTaskUsingCronExpression() {

  //TODO implementirati poslovnu logiku
}
```

- **Asinhrono** izvršavanje operacije podrazumeva izvršavanje u posebnoj niti. Pošto upravljanje nitima u web kontejneru može biti problematično (jer može dovesti do nedostatka operativne memorije), koristi se „Thread pool“ koja osigurava da se koriste postojeće niti a ne da se iznova kreiraju nove i kada više nema niti u „pool“-u operacije se stavljaju u red čekanja.

- Da bi omogućili asinhrono izvršavanje main klasa mora imati @EnableAsync, a metod koji hoćemo asihrono da pozovemo mora imati @Async. Ako se metod izvršava asinhrono a želimo da vraća nešto, ta povratna vrednost mora biti tipa Future

Primer asinhrone metod koja nešto vraća

```
@Async
public Future<String> asyncMethodWithReturnType() {
    try {
      //TODO implementacija poslovne logike

    } catch (InterruptedException e) {
        //obrada izuzetka
    }
    return null;
}
```

Kada pozovemo asinhronu metodu treba da osluškujemo objekat Future

```
Future<String> future = asyncMethodWithReturnType();
while (true) {
    if (future.isDone()) {
                  String rez = future.get();
                 //TODO obrada rezultata izvršavanja
        break;
    }
    System.out.println("Continue doing something else. ");
   //TODO implementacija neke poslovne logike
}
```

18. Internacionalizacija i lokalizacija

- **Internacionalizacija** (i18n) - tehnika koju koristimo kada želimo da lako promenimo aplikaciju tako da prikazuje sadržaj na različitim jezicima a da se pri tome osnovni kod ne menja u zavisnosti od promene jezika

- Postoje dva pristupa:
  1) Napraviti kopije svih stranica na odgovarajućim jezicima (npr. neki statički sajt, da broj stranica nije velik)
  2) Kreirati posebne fajlove message.properties za svaki jezik i onda iz stranica se referencirati na pojedinačna polja u .properties fajlovima (properties fajlovi su tekstualni fajlovi koji su po principu ključ vrednost parova u novom redu, primer imena fajla: messages_en_US.properties). Potrebno je u application.properties definisati parametar **spring.messages.basename** koji sadrži putanju do message fajla i osnovno ime fajla. Fajlovi sa porukama se nalaze u foderu resources. Treba da postoji i messages.properties (default fajl)

- **Lokalizacija** (l10n) – prilagođavanje internacionalizovane aplikacije lokalnim parametrima koji su specifični za različita geografska područja (npr. format datuma 11.02.2017 ili 2017/02/11).

- Klasa java.util.Locale. Pri kreiranju objekta ove klase potrebno je definisati jezik, državu i platformu na kojoj se izvršava aplikacija (za jezik i za države postoje šifrirani ISO kodovi – npr. za Srbiju je kod za državu RS a za srpski latinicu je sr)
- Locale.getDefault() vraća vrednosti koje su podešene na računaru
- Potrebno je definisati i bean tipa LocaleResolver koji čita vrednost objekta Locale na osnovu cookie-ja, ili paramtera u sesiji ili Accept-Language header-a

Primer čitanja objekta Locale iz sesije

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver slr = new SessionLocaleResolver();
    slr.setDefaultLocale(Locale.US);
    return slr;
}
```

⇒ Potrebno je definisati i interceptor koji presreće sve HTTP zahteve i podešava vrednost Locale na novu vrednost

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("lang");
    return lci;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
```

- Za čitanje vrednosti iz properties fajlova koristimo spring jsp taglib biblioteku

```
<%@taglib prefix="s" uri="http://www.springframework.org/tags"%>
```

koristimo tag s:message gde u atributu code definišemo ključ čiju vrednost čitamo iz .properties fajla

```
<h1><s:message code="home.welcome" /></h1>
```

19. Upravljanje izuzecima u Springu

- Pojedini tipovi izuzetaka su automatski mapirani na HTTP statuse
- Spring Boot će tražiti stranicu sa nazivom error.jsp ili error.html i nju prikazati kad se desi greška
- Ukoliko nema takve stranice prikazaće generičku stranicu "Whitelabel Error Page"
- U application.properties dodati server.error.whitelabel.enabled=false ukoliko ne želimo da se prikazuje default stranica sa greskom
- Dodati i property server.error.path=/ kojipokazuje gde se nalaze error stranice

Spring exception	HTTP status code
BindException	400 - Bad Request
ConversionNotSupportedException	500 - Internal Server Error
HttpMediaTypeNotAcceptableException	406 - Not Acceptable
HttpMediaTypeNotSupportedException	415 - Unsupported Media Type
HttpMessageNotReadableException	400 - Bad Request
HttpMessageNotWritableException	500 - Internal Server Error
HttpRequestMethodNotSupportedException	405 - Method Not Allowed
MethodArgumentNotValidException	400 - Bad Request
MissingServletRequestParameterException	400 - Bad Request
MissingServletRequestPartException	400 - Bad Request
NoSuchRequestHandlingMethodException	404 - Not Found
TypeMismatchException	400 - Bad Request

- Izuzetak se može označiti anotacijom @ResponseStatus i mapirati na određeni HTTP status, tako što kreiramo klasu sa tom anotacijom. (default – ako ne nađe status izuzetka vraća HTTP 500)

Anotacija @ExceptionHandler

- Koristimo je ukoliko želimo da obradimo neki izuzetak a da to ne radimo u svakom metodu u try catch bloku. Treba samo da u kontroloru definišemo metod sa tom anotacijom i kao parametre stavimo klase koje obrađuje.

```
@ExceptionHandler(value=RuntimeException.class)
public String handleError(){
    return "error";
}
```

Ako koristimo ovu anotaciju na nivou kontrolora, onda ona obrađuje samo izuzetke koji su se desili u tom kontroloru. Da je ne bi dodavali u sve kontrolore koristimo anotaciju @ControllerAdvice koja nudi obradu izuzetaka na nivou cele aplikacije.

20. RESTFul servisi

- REST (REpresentational State Transfer) je skup ograničenja koja kada se primene u dizajnu sistema definišu arhitektnoski patern. Alternativa Restfull servisima su SOAP servisi (zastereo koncept ali još uvek se negde primenjuje).

- Ograničenja koja nameće REST stil:

1. Sistem mora imati klijent server arhitekturu
2. Sistem mora biti stateless
3. Mora biti podržan mehanizam keširanja
4. Svi resursi moraju imati jedinstvene identifikatore (URI)
5. Servisi se moraju opisati hipertekstom

Za sistem koji podržava ova ograničenja kažemo da je RESTful (ograničenja ne utiču na izbor tehnologije za implementaciju REST sistema).

- RESTfull resurs je sve ono čemu se putem mreže može pristupiti i što se može razmeniti između klijenta i servera. Npr. temperatura u NS u 17:00, neka novinska vest, rezultat pretrage po nekom indeksu na Google-u…

- Svaki resurs ima svoj jedinstveni identifikator i u web servisima je to obično hiperlink. Taj identifikator ne sme da se menja tokom vremena (uvek mora ostati isti).

- Reprezentacija predstavlja oblik u kome se resursi razmenjuju između klijenta i servera i ona može da se menja. Klijent u svojoj komunikaciji sa serverom definiše u kom obliku (reprezentaciji) želi da preuzme resurs. (slika, XML, HTML, JSON...)

Primer resursa: podaci o studentu u nekoj bazi

XML reprezentacija

```
<student>
  <ime>Pera</ime>
  <prezime>Perić</prezime>
  <fakultet>PMF</fakultet>
</student>
```

JSON reprezentacija

```
{"student":{
      "ime":"Pera",
      "prezime":"Perić",
      "fakultet":"PMF"
      }
}
```

- RESTfull sistem koristi HTTP protokol za razmenu reprezentacija resursa, koristi HTTP metode:

- GET se koristi za izvršavanje upita nad resursima i dobavljanje reprezentacije određenog resursa
- POST se koristi za kreiranje novog resursa
- PUT se koristi za ažuriranje postojećeg resursa
- DELETE se koristi za brisanje resursa

- REST servisi u Springu podrazumevaju upotrebu **@RestController** anotacije i metodi kontrolora vraćaju objekte koji su u različitoj reprezentaciji (najčešće JSON) i klijent koji poziva servis je zadužen za generisanje HTML stranice i popunjavanje podacima koji su stigli od servera.

Primer REST kontrolora

```
@RestController
public class BookshopControler {

		@Autowired
		BookshopService bookshopService;

		@GetMapping("/books")
		public List<Book> getAllBooks(){
			return bookshopService.getBookshop();
		}
		@GetMapping("/books/{title}")
		public ResponseEntity<List<Book>> getBooksByTitle(@PathVariable("title") String title){
			List books = bookshopService.getBookByTitle(title);
			if ((books == null)||(books.isEmpty())) {
				return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
			}else {
				return ResponseEntity.ok(books);
			}
		}
}
```

Znači za obradu HTTP zahteva i dalje koristimo iste anotacije @GetMapping, @PostMapping, @PutMapping, @DeleteMapping ali sada metode vraćaju objekte (najčešće tipa ResponseEntity – omogućava da se klijentu vrati HTTP odgovor koji će sadržati HTTP status u zaglavlju a sami objekti će biti smešteni u telu odgovora).

- Za definisanje reprezentacije koja će biti vraćena koristiti se dodatni parametar produce u okviru mapiranja za metod

```
@GetMapping (value="/books/", produces ="application/xml")
```

ako želimo da definišemo u kojoj reprezentaciji metod prihvata resurs koristićemo atribute consumes

```
@PostMapping (value="/save/book/",consumes = "application/xml")
```

- U slučaju kada se pozivaju POST ili PUT metode potrebno je da metod definiše tip objekta koji se nalazi u HTTP zahtevu i za to se koristi anotacija **@RequestBody**

```
@PostMapping("/save/book")
public ResponseEntity saveBook(@RequestBody Book book)
```

- **Spring REST klijenti** omogućavaju da iz jednog servisa pozovemo REST servis koji se nalazi u drugoj aplikacij

- Spring pruža različite načine za ostvarivanje komunikacije: (oba su sinhroni HTTP klijenti)

- RestTemplate
  - Prvobitan način (moguće depricated)
  - Da bi se koristio potrebno je registrovati bean u konfiguracionoj klasi

```
@Bean
RestTemplate restTemplate(RestTemplateBuilder builder) {
		return builder.build();
}
```

- Poseduje odgovarajuće metode za pozivanje HTTP zahteva: getForObject, getForEntity, postForObject, postForEntity, delete, exchange, execute…
- Upotreba: Injektovati rest templejt u servis ili drugi kontrolor i kreirati HTTP zahtev

- RestClient
  - Alternativa za RestTemplate
  - Dovoljno je u servisu ili kontroloru kreirati instacu objekta i pozvati metodu za HTTP protokol

```
RestClient restClient = RestClient.create();

String result = restClient.get()
                          .uri("https://example.com")
                          .retrieve()
                          .body(String.class);
```

```
Pet pet = new Pet()
....
ResponseEntity<Void> response = restClient.post()
                .uri("https://petclinic.example.com/pets/new")
                .body(pet)
                .retrieve()
                .toBodilessEntity();
```

- Za dokumentovanje Spring REST API-ja može se koristiti OpenAPI3 specifikacija, doda se određena biblioteka i onda je dokumentacija dostupna na http://localhost:8080/v3/api-docs , ta adresa se može promeniti u application.properties

- Swagger UI je interaktivni korisnički interfejs koji omogućava vizualizaciju i testiranje RESTful API-ja koji su dokumentovani pomoću OpenAPI specifikacije. Omogućava programerima da istraže i interagiraju sa API-jem bez potrebe da direktno šalju zahteve putem komandne linije ili drugih alata. Dostupan nam je na adresi http://localhost:8080/swagger-ui/index.html čim dodamo OpenAPI biblioteku. (putanja se opet može podesiti).

-HATEOAS (Hypertext As The Engine Of Application State) je arhitektonski patern u izgradnji web servisa. Ideja je da odgovor web servisa sadrži linkove ka drugim resursima. Za opis sintakse tih linkova se može koristiti HAL – on opisuje linkove koristeći JSON, resurs mora biti validan JSON dokument, postoje dve rezervisane reči _links i _embedded. Prednost HAL dizajna je transparentnost lokacije i smanjenje grešaka.

- **Spring Data REST** omogućava da se metode iz Spring repozitorijuma izlože kao REST servisi. Odgovor je formatiran u skladu sa HAL formatom. Treba dodati spring-boot-starter-data-rest biblioteku u pom.xml, i da bi se repozitorijum izložio kao REST servis potrebno je staviti anotaciju **@RepositoryRestResource(collectionResourceRel = "people", path = "people")** kojom definišemo URL putanju do resursa.

21. Autentifikacija i autorizacija u Spring Security modulu

- Spring security modul je poseban deo Spring frameworka koji nudi:

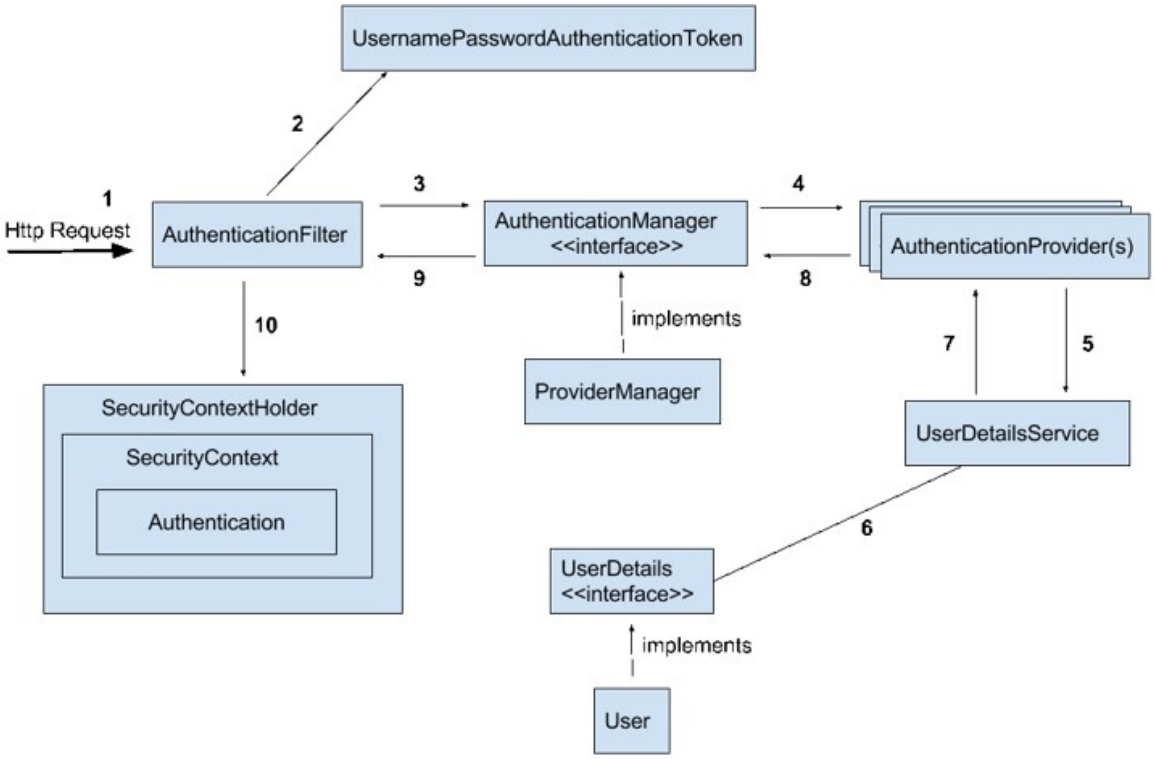
- različite načine autentifikacije
- proveru prava privilegija pristupa
- zaštita na nivou URL-a ali i na nivou metoda
- zaštitu od standardnih napada (CSRF, clickjacking, session fixation)

(više o security modulu u sekciji dodatne informacije)

- **Autentifikacija** je mehanizam pomoću koga određujemo identitet klijenta koji pristupa našoj aplikaciji.

  ⇒ BASIC autentifikacija je najednostavniji protokol za autentifikaciju preko HTTP. Omogućava slanje korisničkog imena i lozinke u okviru header elementa HTTP zahteva
  ⇒ BASIC autentifikacija koristi header element Autorization, username i password su enkodirani po Base64 šemi
  ⇒ Ukoliko klijent pošalje neautentifikovan zahtev, server će vratiti grešku 401 i definisati koji protokol za autentifikaciju očekuje. Informacija o protokolu je smeštena u header element WWW-Authenticate.
  ⇒ Da bi zaštitili lozinku zahteve treba slati kroz HTTPS
  ⇒ Osnovu Spring autentifikacije čini klasa SecurityContextHolder i to je mesto gde se čuva autentifikovani korisnik

  ⇒ Principal je objekat koji opisuje trenutno logovanog korisnika u aplikaciji. On nastaje nakon uspešne autentifikacije.
  ⇒ Credentials je obično password korišćen za autentifikaciju i obično se automatski obriše
  ⇒ Authority je objekat koji opisuje dozvolu/privilegiju koje korisnik ima u aplikaciji, vezuje se za ulogu



  ▫ UsernamePasswordAuthenticationToken je klasa koja implementira objekat Authenitication i koristi se da se u nju smeste username i password koji stignu u HTTP zahtevu

- AuthenticationManager je API koji definiše kako će Spring Security filteri izvršiti autentifikaciju, rezultat je objekat Authenitication koji će biti smešten u SecurityContext-u ako je uspešna autentifikacija
- ProviderManager je implementacija AuthenticationManager i on sadrži listu AuthenticationProvider objekata koji implementiraju različite načine autentifikacije

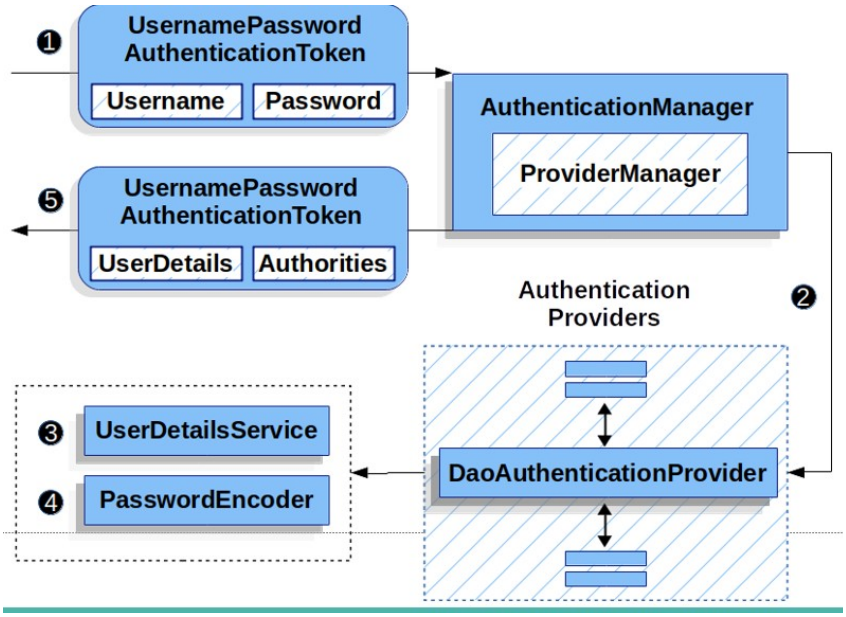
1) **Username/password autentifikacija**

- Potrebno je da kreiramo klasu koja ima anotaciju @EnableWebSecurity i @Configuration, tu ćemo definisati binove za SecurityFilterChain i AuthenticationManager

- Načini konfiguracije Spring aplikacije:

- čuvanje podataka u memoriji
- čuvanje podataka u relacionoj bazi podataka
- pristup podacima preko LDAP servisa

- U binu SecurityFilterChain svašta nešto konfigurišemo (login/logout stranice, remember me, autorizacija itd.)



2) **In memory autentifikacija**

- Najjednostavniji način, dobro za testiranje ali ne i za produkciju.

- Podaci o mogućim korisnicima se čuvaju u memoriji aplikacije

```
@Bean
public UserDetailsService userDetailsService() {
        UserDetails userDetails = User.withDefaultPasswordEncoder()
                .username("user")
                .password("password")
                .roles("USER")
                .build();
        return new InMemoryUserDetailsManager(userDetails);
}
```

3) **Autentifikacija i baza podataka**

- Najčešće se koristi. Potrebno je da postoji tabela sa korisnicima i tabela sa ulogama koje korisnici imaju.

- Potrebno je implementirati interfejs UserDetailsService i njegovu metodu

```
public UserDetails loadUserByUsername(String username)
```

- Takođe potrebno je implementirati i interfejs UserDetails i odgovarajuće metode

- **Autorizacija** je provera da li identifikovani klijent ima pravo pristupa nekom resursu.

1. **Autorizacija na nivou metode**

- Potrebno je dodati anotaciju @EnableMethodSecurity i onda na same metode u okviru Spring beanova (kontrolori, servisi, repozitorijumi) možemo dodati anotacije @PreAuthorize, @PostAuthorize, @PreFilter i @PostFilter

```
@Service
public class MyCustomerService {
        @PreAuthorize("hasRole('ADMIN')")
    public Customer readCustomer(String id) { ... }
}
```

22. Autorizacija preko JWT tokena

- **JWT** (JSON Web Token) je standardni metod za razmenu identiteta izmedju dve aplikacije u obliku JSON objekta. Koristi se za autorizaciju u web aplikacijama. Može se verifikovati jer je digitalno potpisan. Može se i kriptovati pa sadržaj nije vidljiv trećim stranama u komunikaciji. JWT je niz enkodovanih vrednosti po base 64 šemi za enkodiranje.

- Sastoji se od tri dela: header, payload i signature, koji su u formatu header.payload.signature

- ⇒ Header sadrži informaciju o tipu tokena i algoritmu koji se koristi za potpisivanje
- ⇒ Payload sadrži informacije o korisniku koji želi da se autentifikuje
- ⇒ Signature predstavlja potpis koji nastaje tako što uzmemo enkodirani header, enkodirani payload, tajni ključ (nalazi se na serveru) i potpišemo odgovarajućim algoritmom

- Kako radi?

1. Korisnik pošalje svoj username i password
2. Ukoliko su podaci ispravni server generiše JWT token
3. Svaki sledeći put korisnik u Authorization headeru šalje JWT token u sledećem obliku: Bearer <token>
4. Server uzima sadržaj Authorization tokena i proverava da li je token validan
5. Ukoliko je sve ok, omogućava korisniku pristup resurs

Razlika između tokena i sesije

- ❖ Server lako može da poništi sesiju jer se ona nalazi kod njega. Kad klijent pošalje sessionID server će javiti da je sesija istekla. Kod tokena je to komplikovanije.
- ❖ ID sesije se prenosi kao cookie i zauzima manje prostora u prenosu preko mreže nego JWT token
- ❖ JWT su pogodniji za mikroservisnu arhitekturu
- ❖ Sesije troše resurse na serveru, što je značajno ako imamo velik broj korisnika aplikacije

Kako ga integrisati?

1. Potrebno je kreirati konfiguracionu klasu u kojoj definišemo beanove za SecurityFilterChain, AuthenticationManager i PasswordEncoder i implementirati interface UserDetailService koji omogućava autentifikaciju preko baze podataka.

2. Treba dodati dependency u pom.xml

3. Treba kreirati klasu koja će enkapsulirati logiku za kreiranje JWT tokena, ekstrakciju claims-ova i validaciju tokena

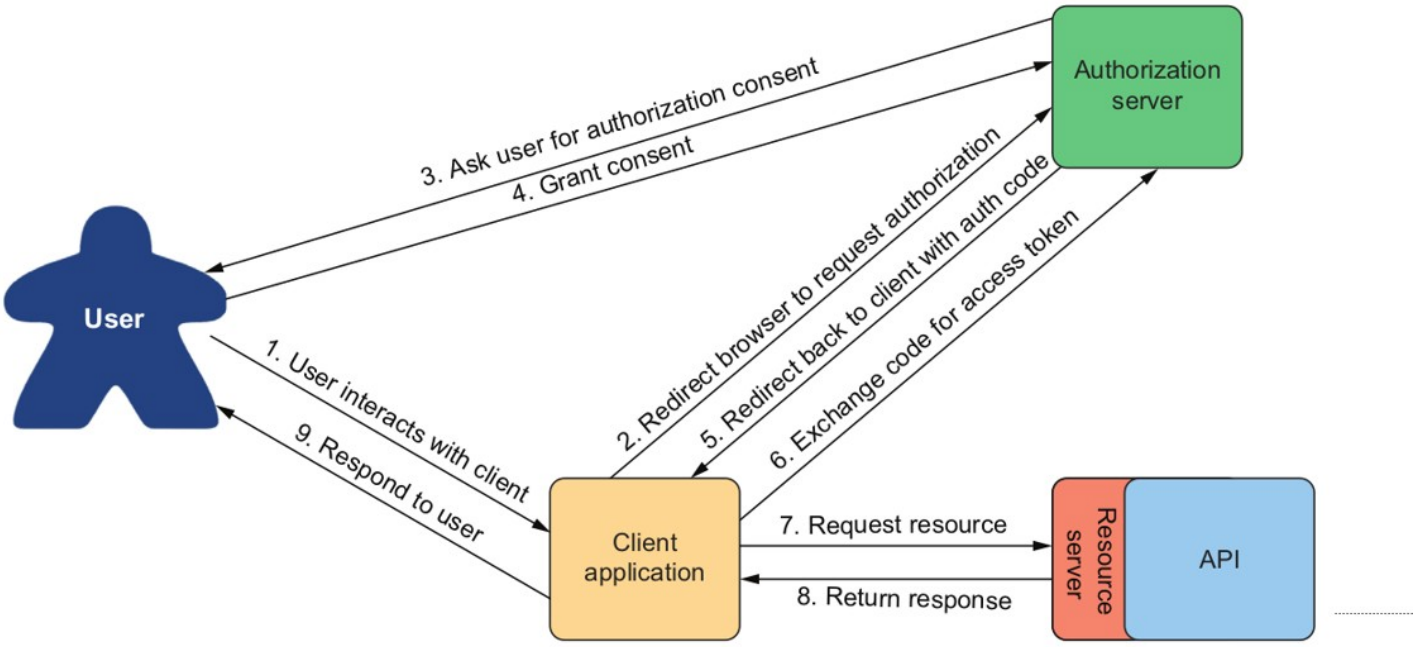
4. Krirati API za autentifikaciju koji prima username i password i vraća generisani JWT token kao response

5. Kreirati Filter koji će presretati sve klijentske HTTP zahteve i svaki put proveravati da li je korisnik poslao token, ako jeste pokušaće da ga validira: validan - prosledi zahtev kontroleru, nije validan - vrati grešku.

23. OAuth2 autentifikacija

- OAuth (Open Authorization) je standardni protokol za davanje ovlašćivanje pristupa između različitih aplikacija. Razvijen je kako bi omogućio siguran pristup podacima korisnika bez deljenja njihove lozinke izmedju aplikacija.

Autorizacija pomoću OAuth 2 protokola:



- ❖ Authorization server – Zadatak: dobij dozvolu od korisnika u ime klijentske aplikacije, ako korisnik odobri dozvolu onda server dodeljuje pristupni token klijentskoj aplikaciji koji koristi za autentifikovan pristup API-u
- ❖ Resource server – backend aplikacija koja je izložila REST API ali zahteva autentifikaciju za njihovo korišćenje
- ❖ Client application – frontend aplikacija koja želi da koristiti REST API
- ❖ Korisnik – osoba koja koristi klijentsku aplikaciju i odobrava aplikaciji pristup API-ju resource servera u njeno ime

- Spring Boot pruža podršku za kreiranje OAuth klijenta kako bi se uspostavil komunikacija sa nekim authorization serverom npr Google ili GitHub:

- Treba dodati biblioteku spring-boot-starter-oauth2-client

- Nakon toga treba podesiti parametre za pristup autorizacionom serveru u application.properties (clientId, clientSecret…)
- Pretpostavka je da se na GitHubu ili Google dozvolili pristup preko OAuth protokola i to podešavanje nema veze sa samim SpringBoot-om

DODATNE INFORMACIJE

- Paginacija je često korisna kada imamo veliki skup podataka i želimo ga prikazati korisniku u manjim delovima. Treba naslediti PagingAndSortingRepository i onda su nam odgovarajuće metode za rad sa paginacijom već dostupne.

```
public interface UserRepository extends JPARepository<User, Long> {

   Page<User> findByLastName(String lastName, Pageable pageable);
}
```

Klasom Pageable definišemo koji deo podataka želimo (broj strane, veličina strane i parametre sortiranja.
Poziv metode:

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("lastName"));
Page<User> usersPage = userRepository.findByLastName("Smith",
pageable);
```

Objekat klase Page sadrži informaciju koliko ukupno ima strana pa tu informaciju koristimo dok u for petlji iteriramo po repozitorijumu.

- Ukoliko nije neophodna paginacija, sortiranje se može definisati i samostalno nad metodom repozitorijuma:

Spring JSP tag biliblioteke

- Spring ima dve taglib biblioteke od kojih prva koristi u JSP za lakše formatiranje izgleda stranice a druga sadrži opšte funkcije.

- Informacije o greškama se mogu prikazati na JSP stranici ako se koristi tag <sf:errors>

JSP tag	Description
<sf:checkbox>	Renders an HTML <input> tag with type set to checkbox.
<sf:checkboxes>	Renders multiple HTML <input> tags with type set to checkbox.
<sf:errors>	Renders field errors in an HTML tag.
<sf:form>	Renders an HTML <form> tag and exposed binding path to inner tags for data-binding.
<sf:hidden>	Renders an HTML <input> tag with type set to hidden.
<sf:input>	Renders an HTML <input> tag with type set to text.
<sf:label>	Renders an HTML <label> tag.
<sf:option>	Renders an HTML <option> tag. The selected attribute is set according to the bound value.
<sf:options>	Renders a list of HTML <option> tags corresponding to the bound collection, array, or map.
<sf:password>	Renders an HTML <input> tag with type set to password.
<sf:radiobutton>	Renders an HTML <input> tag with type set to radio.
<sf:radiobuttons>	Renders multiple HTML <input> tags with type set to radio.
<sf:select>	Renders an HTML <select> tag.
<sf:textarea>	Renders an HTML <textarea> tag.

Slanje mejla

- Potreban je mail server i registrovana email adresa.
- Parametri se postavljaju u application.properties. (host, port,username, password, auth…)
- Treba nam spring-boot-starter-mail u pom.xml.

Nakon svega toga samo injektujemo objekat klase JavaMailSender koji ima metodu send i očekuje objekat tipa SimpleMailMessage

```
@Component
public class EmailServiceImpl implements EmailService {

   @Autowired
   private JavaMailSender emailSender;

   public void sendSimpleMessage(String to, String subject, String text) {
      SimpleMailMessage message = new SimpleMailMessage();
      message.setFrom("rispmf@gmail.com");
      message.setTo(to);
      message.setSubject(subject);
      message.setText(text);
      emailSender.send(message);
   }
}
```

- Ukoliko želimo da šaljemo attachment u mejlu koriste se klase MimeMessage i MimeMessageHelper.

Keširanje

- Ukoliko neka operacija zahteva puno vremena za izvršavanje moguće je keširati rezultat izvršavanja. Operacija se izvršava samo prvi put i nakon sledećeg poziva vrednost će se čitati iz keša.

- Potrebno je staviti anotaciju @EnableCaching na main klasu i anotaciju @Cacheable nad operacijom čiji rezultat želimo da keširamo. Ako

Primer

```
@Component
public class MyMathService {

   @Cacheable("piDecimals")
   public int computePiDecimal(int precision) {
      ...
   }

}
```

ništa ne konfigurišemo Spring će koristiti ConcurrentHashMap objekat za čuvanje vrednosti u memoriji (nije preporučljivo)

Rad sa fajlovima

- Ukoliko želimo da uploadujemo fajlove, Spring koristi objekat tipa MultipartFile

- Potrebno je kreirati odgovarajući metod u kontroloru

```
@PostMapping("/")
public String handleFileUpload(@RequestParam("file") MultipartFile file)
```

- Koristiti HTML formu sa atributom enctype="multipart/form-data"

```
<form method="POST" enctype="multipart/form-data" action="/">
```

- Dodatno u application.properties možemo podesiti veličinu fajla i lokaciju.

Spring modul

- Da bi ga konfigurisali treba nam spring-boot-starter-security. Po default-u svaki metod kontrolora očekuje Basic autentifikaciju. Spring kreira default korisnika sa username-om *user* i hešovanom lozinkom koja se ispisuje u konzoli pri pokretanju aplikacije, da bi smo to isključili treba da isključimo klasu SecurityAutoConfiguration:

```
@SpringBootApplication(exclude = { SecurityAutoConfiguration.class }
```

- Spring security koristi Filtere iz Sevlet tehnologije, to su klase koje presreću HTTP zahtev i omogućavaju manipulaciju zahtevima koji se šalju servletima. Filteri se vezuju u lanac filtera (filter chains). Svaki filter može da odluči da li da obradi zahtev i pusti dalje, sledećem filteru ili da vrati grešku klijentu i prekine niz. Klasa DelegatingFilterProxy posao delegira posebnom Spring beanu (SecurityFilterChain) koji je zadužen za konfigurisanje sigurnosnih mehanizama u našoj aplikaciji.

Spring Security JSP tags omogućavaju da se odredi vidljivost html elemenata na osnovu uloge korisnika koji je logovan

- Treba dodati spring-security-taglibs u pom.xml, i u jsp stranicu uvesti taglib

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

- Ključni tag je authorize

```
<sec:authorize access="hasRole('admin')">
 sadržaj je vidljiv samo onim korisnicima sa ulogom admin
</sec:authorize>
```

- Atribut access može imati poziv neke od funkcija definisanu kao Spring security expression:

- hasRole(), hasAnyRole()
- isAuthenticated()
- isRememberMe()

Tag <sec:authentication> koristi se kada je potrebno prikazati podatke o autentifikovanom korisniku

```
<sec:authorize access="isAuthenticated()">
    Welcome Back, <sec:authentication property="principal.username"/>
</sec:authorize>
```

Tag <sec:csrfToken> omogućava dodavanje skrivenog polja za zaštitu od CSRF napada

```
<form method="post" action="/submit">
    <input type="hidden" name="_csrf" value="<sec:csrfToken />" />
    <!-- ostala input polja -->
</form>
```

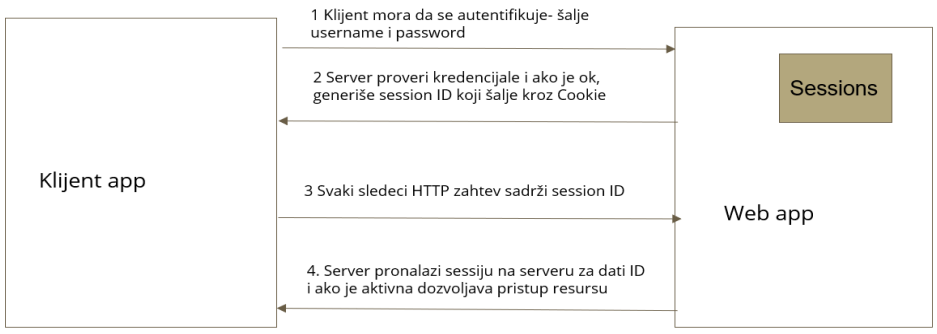
Spring Security i zaštita od napada

- Nekih od ključnih napada koje Spring Security štiti po defaultu:

- Cross-Site Request Forgery (CSRF) - napadač pokušava da iskoristi činjenicu da ako ste logovani na nekoj aplikaciji onda može da uputi zahtev toj aplikaciji bez vašeg znanja i iskoristi vaše kredencijale, obično se dešavaju tako što kliknete na neko dugme na nekoj web stranici a on preusmeri zahtev ka aplikaciji na kojoj ste logovani trenutno u browseru i iskoristi podatke iz sesije vašeg browsera. Upotrebom JavaScripta čak ne morate nigde ni da kliknete dovoljno je da se učita maliciozna stranica i desi će se CSRF napad.
    - Dva načina zaštite: CSRF token - automatski dodaje pri svakom slanju zahteva ka web aplikaciji, aplikacija onda proverava da li je primljeni token identičan onom koji očekuje, obično se u html stranicu dodaje skriveno polje koje sadrži taj token. Drugi način je upotreba SameSite atributa koju server zahteva da se nađe u Cookie-ju ako se zahtev šalje sa iste adrese (vrednosti: Lex, Strinct, None)
- Cross-Site Scripting (XSS) - napadač ubacuje zlonamerni skript (obično JavaScript) u web stranicu koju korisnik pregledava. Fokus na ulaznim podacima.
    - Tri načina zaštite: Sanitizacija i validacija ulaza, izbegavajte direktno uključivanje korisničkih podataka u DOM, koristite sigurnosne HTTP headere

Sav saobraćaj mora ići po HTTPS protokolu

Autentifkacija/autorizacija koristeći sesiju



Problematičan kada imamo mikroservisnu arhitekturu.