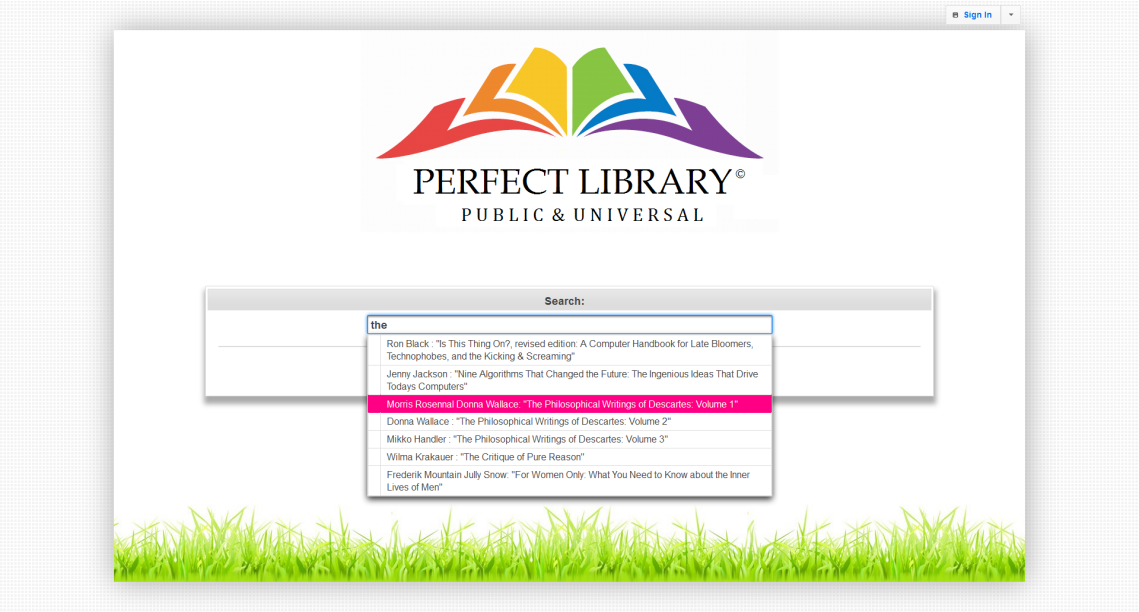
Popis webovej aplikácie: „Perfect Library“

***abstrakt**: v stručnom úvode čitateľ nájde jednoduchý popis a vymenovanie použitých technológií, nasleduje odbornejší rozbor, popis kódu, popis úspechov a neúspechov, v prílohe je návod na inštaláciu, popis konfigurácie aplikačného serveru plus glassfish-resources.xml a dokument s testovacími dátami (cca 200 SQL insertov)*



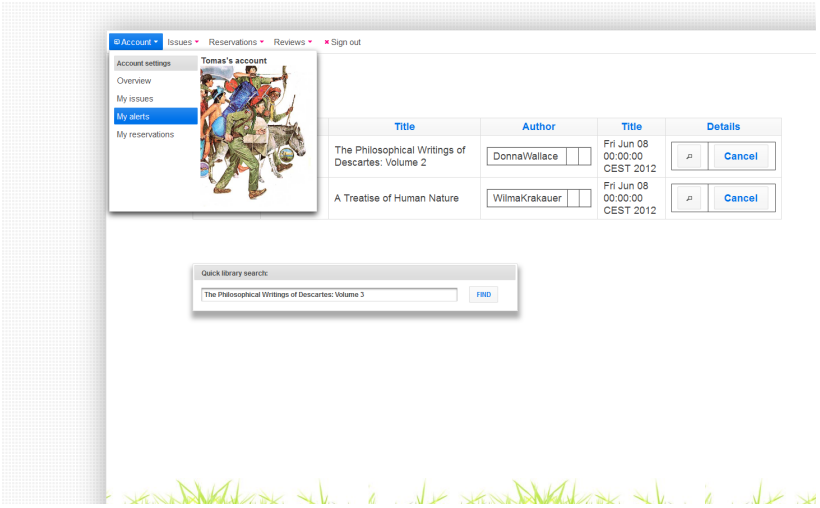
Perfect Library je *“library management system”* aplikácia s podporou lokalizácie umožňujúca návštevníkom fulltextovo vyhľadávať a prezerať dokumenty a čítať si ich recenzie. Registrovaným užívateľom knižnice umožňuje v zabezpečenom prístupe uskutočňovať výpožičky, rezervácie a pridávať recenzie. Perfect Library je *Java EE6* aplikácia bežiaca na servery *Glassfish 3.1.2.* aktívne využívajúca tieto technológie:

Java API for RESTful Web Services (JAX-RS) 1.1
Java Architecture for XML Binding (JAXB) 2.2
JavaServer Faces 2.0 (PrimeFaces 3.2, PrettyFaces 3.3.3.)
Enterprise JavaBeans 3.1
Java Persistence 2.0
Java Message Service API 1.1
JavaMail 1.4
Java Transaction API (JTA) 1.1

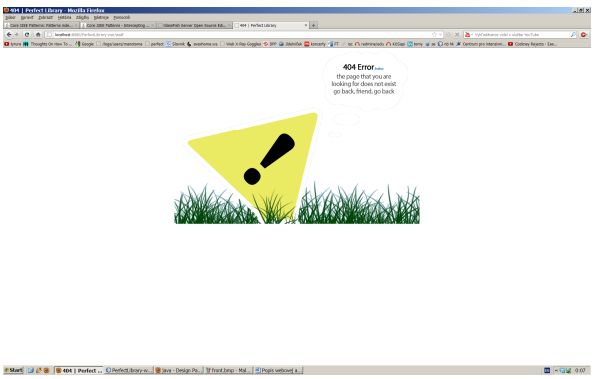
Aplikácia je napísaná elegantným štýlom postavenom na zásadách OOP, vďaka ktorých ju možno veľmi flexibilne modifkovať z minimálnym zásahom do kódu podľa meniacich sa potrieb a politike zákazníka – knižnice. Z dôvodu limitovanej veľkosti tohto dokumentu, nech je ukážkou aspoň nasledovný krátky príklad.
Knižnica sa rozhodne poskytovať novú službu – výpožičku LP platní. O platniach bude chcieť evidovať veci ako *názov, dĺžku, priemer* a požadovať aby tieto informácie mohli vidieť návštevníci jej stránok a fulltextovo vyhľadávať dostupné exempláre. Zásah do kódu je minimálny stačí vytvoriť triedu *GramophoneRecord* ktorá je potomkom abstraktnej triedy *LibraryAudioDocument* (tá je zase potomkom koreňovej abstraktnej triedy *LibraryDocument*) a pridať jej jediný atribút „*recordRadius*”, ktorému pridáme anotáciu *@SpecifiedLibraryDocumentProperty*. Na záver ešte túto triedu oanotujeme anotáciou *@SpecificationsProcessingStrategy(SpecificationsProcessingStrategyValue.REFLEXION)* (ďalšou možnosťou je zabudovaná stratégia VISITOR_PATTERN keď nechceme použiť reflexiu). Kód skompilujeme a po redeployi bude môcť návštevník okamžite fulltextovo

vyhľadávať a prezerať platne a ich priemer a dĺžku, pridávať k nim recenzie, uskutočňovať výpožičky a rezervácie bez akéhokoľvek potrebného zásahu do kódu prezentačnej vrstvy alebo business logiky.

V EJB vrstve používam dva typy *session bean* – *stateless* a *singleton*. Môj usecase pre singleton: každá služba, ktorú knižnica poskytuje (napr. výpožička) môže mať nejakú špecifickú politku (napr. výpožičná politika), ktorú môže tvoriť viacero pravidiel a ktoré treba dodržiavať (napr. od každého dokumentu možem mať požičaný iba jeden kus). Je to záležitosť, u ktorej je veľká pravdepodobnosť zmien a preto nie je vhodné kontrolovať takéto podmienky na tvrdo zakodované v metodách ako statementy pred zapisom do databázy a potom ich hľadať ked sa niečo zmení. Vhodnejší prístup je každé pravidlo navrhnúť ako triedu a potom jej objekt zaregistrovať v nejakom centralizovanom providerovi takýchto pravidiel, ku ktorému by sa dalo kedykoľvek pristúpiť keď je treba a ktoré môžu potom používať nejaké validátory. Práve takýto provider je u mňa singleton (+ anotacia @javax.ejb.Startup). Je to potomok abstraktnej triedy *LibraryServicePolicyProvider*, má jedinú public metódu, ktorá je od predka: "*final public Set<ServicePolicyRule> getPolicyRules()*" (vždy vracia *return Collections.unmodifiableSet(rules);*) z anotáciu *@javax.ejb.Lock(LockType.READ)* pretože: "Allows *simultaneous* access to methods designated as *READ*".



V stateless beanach používam rôzne *TransactionTypes* (konkrétne *SUPPORT*, *REQUIRED*, *REQUIRED_NEW*). Typ *REQUIRED_NEW* som použil hlavne pri potvrdzovaní eventu pomocou emailu, pretože som nechcel ak by niečo pri spracovaní zlyhalo bola rollbacknutá celá tranzakcia.



Potvrdzovaciu časť eventov v systéme, v ktorej som chcel využiť asynchrónne spracovanie, má na starosti *MessageDrivenBeana* - *ConfirmationMessageBean*. Ako messaging model som použil *Point-to-Point(Queue)*, pretože "every message metters". Táto časť je navrhnutá dômyselne. MessageDrivenBeana dokáže spracovať iba objekty typu *ConfirmationRequiredEvent* (čo je inteface) z ktorých sa nemôže dozvedieť žiadne detaily, ktorého účtu a služby sa event týka. Samotné poslanie mailu je vďaka zabudovanej podpore v glassfishi veľmi jednoduché a v prílohe je popis príslušnej konfigurácie. Ako primitívnu alternatívu JMS som si v jednom stateless beane vyskúšal anotáciu @Asynchronous (pri vytvarani užívateľskej recenzie), kde sa mi zdalo príhodné nasadiť metodiku *"fire and forget"*.

Aplikácia poskytuje REST rozhranie umožňujúce vďaka JAXB vrátiť XML reprezentáciu (1) dokumentu podľa ID, (2) recenziu dokumentu špecifikovaného podľa jeho ID, (3) všetky recenzie užívateľa podľa jeho užívateľského mena alebo (4) celého mena. Ukážky URI:
(1) *http://localhost:8080/PerfectLibrary-war/resources/document/2*
(2) *http://localhost:8080/PerfectLibrary-war/resources/document/review/10*
(3) *http://localhost:8080/PerfectLibrary-war/resources/reviews/tommy*
(4) *http://localhost:8080/PerfectLibrary-war/resources/reviews/Tomas-Mano*
(5) *http://localhost:8080/PerfectLibrary-war/resources/app/ping (ak beží odpovedie "alive")*
(6) *http://localhost:8080/PerfectLibrary-war/resources/app/whoareyou*
Trieda *PenaltyTimerService* (ktorá je opäť dalším *javax.ejb.singleton)* je realizáciou Schedulera z konfiguráciou *timerConfig.setPersistent(false)*. V zadanom čase môžu vzniknúť objekty triedy NewPenaltyEvent (implementujúce spomínaný interface *ConfirmationRequiredEvent),* ktorý potom dostane JMS *ConfirmationMessageBean*, ktorá spustí rutinu generácie a odoslania potvrdzujúceho e-mailu o vystavení upomienky.

Čo všetko som si ďalej vyskúšal stručne (mimo vyhrátok z JPA v 2. checkpointe):
- JSF: nepoužívam anotácie, všetko je v faces-config.xml, celá navigacia pomocou navigation cases (tomu som rád, do budúcna to ani inak robiť nebudem, škoda že nevyšiel čas vyhodiť všetky anotácie z entít do ejb-jar.xml) a customizáciu JSF error messages prekritím javax.faces.component.UIInput.REQUIRED a zmenu levelov (nemám rád ked sa na užívateľa valia správy z výkričnikom v červenej farbe za hlúpy preklik)
- Lokalizácia pre SK a EN
- Vlastný konverter, je to obyčajná trieda, tak som si tam vyskušal ručný inject EJB pomocou JNDI lookupu (čo robilo potom problémy, lebo som dosť refraktoroval package väčšinou som ten String v lookup zabudol prepísať a hladať aktuálne JNDI nebolo vždy med lízať)
- Customizovaný error page (pre 404) vo web.xml
- Použitie rôznych primefaces themes
- URL rewriting pomocou frameworku PrettyFaces (nie moc úspešne)
- JAXB – veľmi zlý dojem, pred tým som trochu robil s (http://simple.sourceforge.net/)a JAXB mi prišlo veľmi neinuitívne a odfláknuté (potreba ID-čiek aby sa zabránilo nekonečnej rekurzii pri vykreslovaní stromu kde je vzťah many to many, ako id akceptuje iba stringy, na long si treba napísať vlastný adaptér atď.)
- Ručný encryption hesla zo stringu na hash s použitím salt (som za to rád i keď sú tam použité proprietárne knihonvy a pre prax to riešenie je asi skôr nepoužiteľné, kvôli možnosti vystopovať heslo, keď putuje od browseru na server)
- Implementácia vlastného security realmu podporujúceho „osolenie" hesla do glassfishu rozšírením AppservPasswordLoginModule.class a DigestRealmBase.class

Čo by som vylepšil:
- popridával viac logovania a Junit testy (nebolo moc času lebo som chcel hlavne skušať java EE veci)
- nepodarilo sa mi rozchodiť JSF templaty, dosť to špatí kód
- málo času vymyslieť niečo rozumné na využitie AOP, škoda
- viacej javadocu
- radšej viacero menších ejb modulov než jeden vačší (už som si na to netrúfol kôli tomu ako by som riešil videteľnosť atď.)

Čo sa týka **neočakávaných problémov**, tých bolo milión, jednu dobu som to zapisoval, tak na konci prikladám v prílohe zoznam necelých 20, najhoršie boli tie, ktoré sa objavili z ničoho nič a samy bez zjavnej príčiny zmyzli, kvôli tomu by som si v budúcnosti chcel radšej vyskúšať JBoss, TOP 4:

- *“org.apache.catalina.LifecycleException: java.lang.IllegalArgumentException: javax.servlet.ServletException: com.sun.enterprise.container.common.spi.util.InjectionException: Error creating managed object for class: class org.jboss.weld.servlet.WeldListener”*
- *java.lang.Exception: WEB0113: Virtual server [server] already has a web module [PerfectLibrary-war.war] loaded at [/PerfectLibrary-war]; therefore web module [PerfectLibrary-war] cannot be loaded at this context path on this virtual server. . Please see server.log for more details.*
- *java.lang.VerifyError: (class: edu/simulation/perfectlibrary/model/human/MemberAccount_, method: <init> signature: ()V) Constructor must call super() or this() VerifyError exception docs says: Thrown when the "verifier" detects that a class file, though well formed, contains some sort of internal inconsistency or security problem*
- *Error occurred during deployment: Exception while deploying the app [PerfectLibrary5] : Cannot resolve reference Local ejb-ref name=edu.simulation.perfectlibrary.business.LibraryIssueService/memberAccountService,Local 3.x interface =edu.simulation.perfectlibrary.business.MemberAccountServiceLocal,ejb-link=null,lookup=,mappedName=,jndi-name=,refType=Session because there are 2 ejbs in the application with interface edu.simulation.perfectlibrary.business.MemberAccountServiceLocal.*

(popis konfigurácie glassfishu na ďalšej strane)

Popis konfigurácie glassfishu:

1. JDBC resources

Edit JDBC Resource

Edit an existing JDBC data source.

Load Defaults

JNDI Name: jdbc/perfectlibrary

Pool Name: mysql_perfect_lib_rootPool

Use the JDBC Connection Pools page to create new pools

Description:

Status: Enabled

Additional Properties (0)

2. JDBC Connection pools

Edit JDBC Connection Pool

Modify an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular database.

Load Defaults | Flush | Ping

General Settings

Pool Name: mysql_perfect_lib_rootPool

Resource Type: javax.sql.DataSource

Must be specified if the datasource class implements more than 1 of the interface.

Datasource Classname: com.mysql.jdbc.jdbc2.optional.MysqlDataSource

Vendor-specific classname that implements the DataSource and/or XADataSource APIs

3. JMS Resources – Connection Factories

Edit JMS Connection Factory

Editing a Java Message Service (JMS) connection factory also modifies the associated connector connection pool and connector resource.

Load Defaults

General Settings

Pool Name: * jms/PerfectLibraryConfirmationQueue

JNDI Name: jms/PerfectLibraryConfirmationQueue

Resource Type: * javax.jms.QueueConnectionFactory

Description:

Status: Enabled

4. JMS Resources – Destination Resources

Edit JMS Destination Resource

Editing a Java Message Service (JMS) destination resource also modifies the associated admin object resource.

Load Defaults

JNDI Name: confirmationRequestsQueue

Physical Destination Name * confirmationRequestsQueue

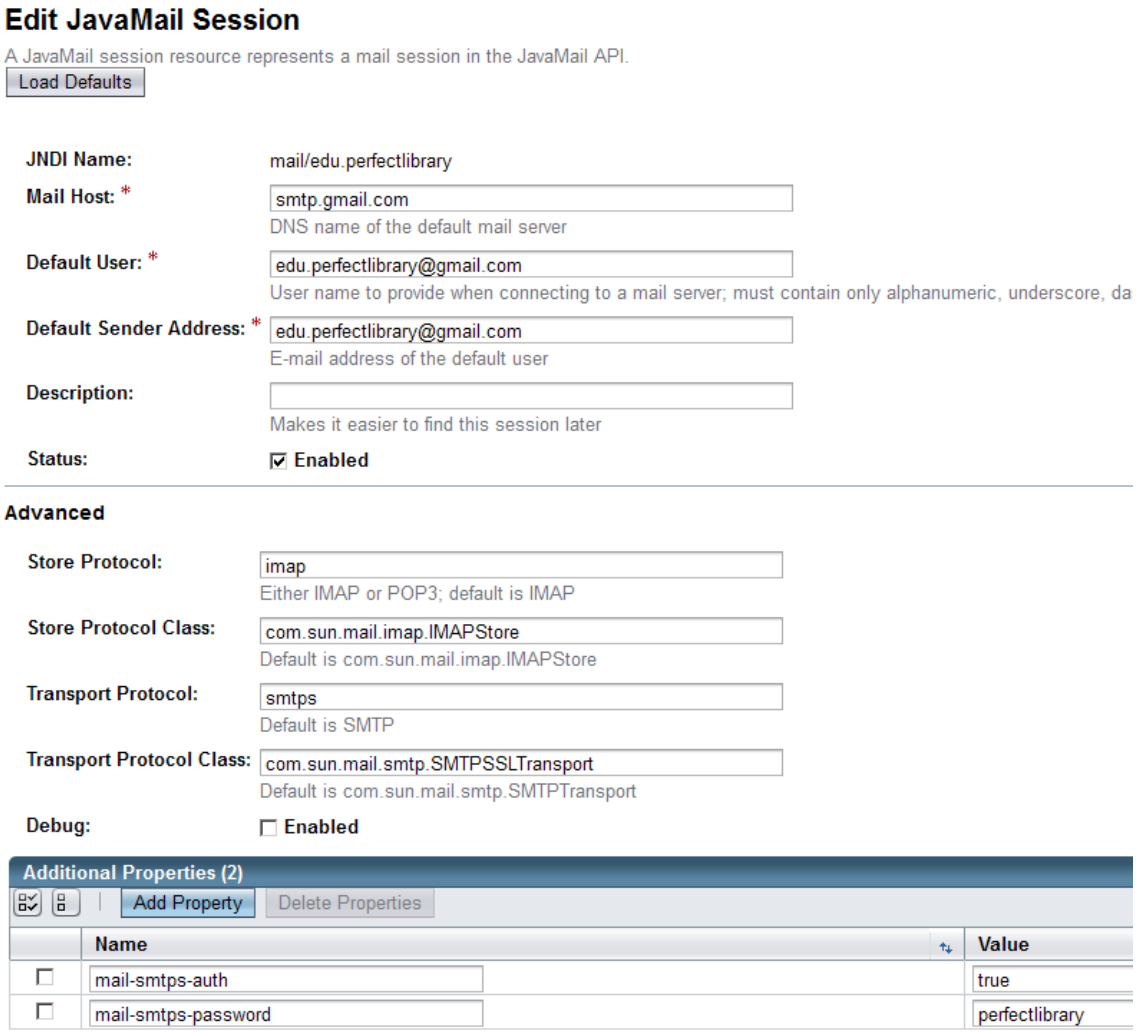
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: * javax.jms.Queue

Description:

Status: Enabled

5. JavaMail Sessions



glassfish-resources.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Resource Definitions//EN" "http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
<resources>
   <jdbc-connection-pool allow-non-component-callers="false" associate-with-thread="false" connection-creation-retry-attempts="0" connection-creation-retry-interval-in-seconds="10" connection-leak-reclaim="false" connection-leak-timeout-in-seconds="0" connection-validation-method="auto-commit" datasource-classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource" fail-all-connections="false" idle-timeout-in-seconds="300" is-connection-validation-required="false" is-isolation-level-guaranteed="true" lazy-connection-association="false" lazy-connection-enlistment="false" match-connections="false" max-connection-usage-count="0" max-pool-size="32" max-wait-time-in-millis="60000" name="mysql_perfect_lib_rootPool" non-transactional-connections="false" pool-resize-quantity="2" res-type="javax.sql.DataSource" statement-timeout-in-seconds="-1" steady-pool-size="8" validate-atmost-once-period-in-seconds="0" wrap-jdbc-objects="false">
      <property name="serverName" value="localhost"/>
      <property name="portNumber" value="3306"/>
```

```
        <property name="databaseName" value="perfect_lib"/>
        <property name="User" value="root"/>
        <property name="Password" value="password"/>
        <property name="URL" value="jdbc:mysql://localhost:3306/perfect_lib"/>
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="jdbc/perfectlibrary" object-type="user" pool-name="mysql_perfect_lib_rootPool"/>
    <admin-object-resource enabled="true" jndi-name="jms/PerfectLibraryConfirmationQueue" res-type="javax.jms.Queue" res-adapter="jmsra">
        <property name="Name" value="PhysicalQueue"/>
    </admin-object-resource>
    <connector-connection-pool name="jms/PerfectLibraryConfirmationQueueFactoryPool" connection-definition-name="javax.jms.QueueConnectionFactory" resource-adapter-name="jmsra"/>
    <connector-resource enabled="true" jndi-name="jms/PerfectLibraryConfirmationQueueFactory" pool-name="jms/PerfectLibraryConfirmationQueueFactoryPool" />
</resources>
```