****

Katedra Informatyki i Automatyki

Aplikacje internetowe

System dokumentujący E-thing

Wożniak Piotr

Rzeszów, 2016

Spis treści

[1. Wstęp 3](#_Toc470619635)

[2. Opis systemu 3](#_Toc470619636)

[3. Etapy tworzenia projektu 3](#_Toc470619637)

[4. Wykorzystane technologie, wymagania 3](#_Toc470619638)

[5. Funkcjonalności 4](#_Toc470619639)

[6. Opis aplikacji 5](#_Toc470619640)

[7. Wnioski 10](#_Toc470619641)

# Wstęp

Projekt zakłada stworzenie systemu pozwalającego dokumentacje przedmiotów o materialnym charakterze. Ma ona pomóc w nadzorować zmian zachodzącym na określonym dobrze w celu pomocy w operacji sprzedaży. Operacje dokumentujące uwzględniają użycie aplikacji webowej z użyciem komputera lub urządzenia mobilnego.

# Opis systemu

System opisuje podstawowe elementy:

* User – określa użytkownika i jego podstawowe podmioty takie jak np. login, hasło itd.
* Thing – przedmiot posiadany przez użytkownika,
* Thing type – typ przedmiotu,
* Thing image – określa obrazek przedmiotu,
* Feature – cecha przedmiotu określająca jego stan,

# Etapy tworzenia projektu

Podczas wykonywania projektu można było określić następujące etapy:

1. Wybór tematu projektu oraz analiza,
2. Określenie wymagań projektowych oraz wybór technologii,
3. Wybór hostu oraz stworzenie repozytorium,
4. Opracowanie struktury bazy danych oraz aplikacji serwerowej,
5. Opracowanie strony klienta,
6. Testowanie aplikacji,
7. Dokumentacja projektu

# Wykorzystane technologie, wymagania

Wymagane w realizacji projektu było dobranie odpowiednik technologii tworzących aplikacje oraz hostu umożliwiającego udostepnienie jej.

Wykorzystywane technologie:

* Java EE,
* Apache Maven,
* JavaScript,
* Spring framework,
* Hibernate framework,
* HTML
* Ajax,
* Semantic UI

Wykorzystanym hostem w projekcie jest:

* Heroku

Wykorzystana baza danych:

* PostgreSQL

Repozytoria:

* bitBucket,
* heroku

# Funkcjonalności

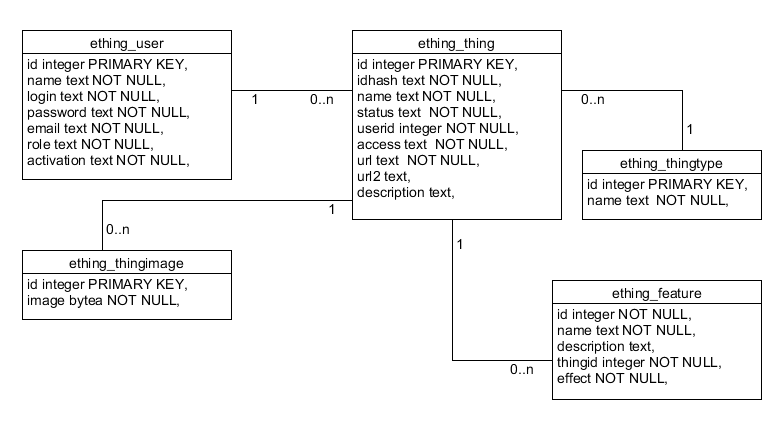
Użytkownik (niezarejestrowany)

* Zarejestrowanie się,
* Przeglądanie profilów oraz przedmiotów publicznych

Użytkownik (zarejestrowany)

* Logowanie się,
* Funkcja przypomnienia hasła,
* Zmiana danych profilu,
* Przeglądanie profilów oraz przedmiotów,
* Dodawanie własnych przedmiotów,
* Edycja własnych przedmiotów,
* Usuwanie własnych przedmiotów,
* Wyszukiwanie publicznych przedmiotów,

Administrator

* (niezaimplementowane)

# Opis aplikacji

* 1. Baza danych

Wykorzystano postgreSQL, który udostępnia wtyczka Heroku Postgres. Skorzystano z pgAdmin 4 z pomocą, którego stworzono bazę danych.

* 1. Strona serwerowa

Plik POM (konfiguracja maven)

Używane artifactId:

* spring-boot-starter-parent,
* spring-boot-starter-web,
* spring-boot-starter-data-rest,
* spring-boot-starter-data-jpa,
* spring-security-crypto,
* spring-boot-starter-thymeleaf,
* postgresql,
* spring-boot-starter-security,
* spring-security-config,
* spring-web,
* spring-boot-autoconfigure,
* mail,
* spring-context-support

Konfiguracja aplikacji (pl.ething.config)

* ApplicationMail – klasa implementuje metody związane z wykorzystaniem poczty email. Zawiera metodę do wysłania wiadomości email aktywującej konto oraz przypominające hasło użytkownika.
* DatabaseConfig – klasa konfiguruje elementy połączenia z bazą danych. Do najistotniejszych zalicza się:

LocalContainerEntityManagerFactoryBean,

DriverManagerDataSource,

JpaTransactionManager,

PersistenceExceptionTranslationPostProcessor.

Określa się miedzy innymi dane do połączenia z bazą danych (lokalna, heroku) oraz paczkę zwierającą model aplikacji.

* SecurityConfiguration – klasa implementuje podstawowe funkcje zabezpieczeń aplikacji. Pozwala na logowanie użytkowników, oraz blokowanie zawartości niedostępną dla określonych użytkowników.

Warstwa controllera (pl.ething.controller)

* MainController – klasa controllera obsługuje metody udostępniające zawartość klienta z użyciem thymeleaf. Metody z użyciem określonego URL, metody oraz repozytorium możliwe jest stworzenie komunikacji strony serwerowej i klienckiej.
* ThingController – klasa obsługuje metody związanie z modyfikacją przedmiotów,
* UserController – klasa obsługuje metody związane z modyfikacją użytkowników,

Warstwa modelu (pl.ething.model)

Warstwa zawiera w sobie klasy modelu reprezentujące encje bazy danych. Z ich użyciem możliwe jest korzystanie z Hibernate Framework i modyfikację danych.

Warstwa repozytorium (pl.ething.repository)

Analogiczna z warstwą modelu, interfejsy budują repozytoria umożliwiające działania na danych. Udostępniają podstawowe operacje np. zapis, usuwanie. Możliwe jest również kreowanie własnych metod.

Warstwa DTO (pl.ething.dto)

Warstwa zawiera w sobie klasy transferowe.

Warstwa main (pl.ething.main)

Zawiera klasę startową aplikacji. Używa odpowiednich adnotacji konfigurujących oraz startujących (Spring boot).

* 1. Strona klienta

Templates

Do stworzenia warstwy prezentacji użyto HMTL wspieranego przez thymeleaf oraz semantic UI. Templaty są zwracane przy użyciu kontrolera. String określa nazwę pliku html i na podstawie jej wyświetlana jest odpowiednia strona. Taki sposób pozwala również pozwala na dostęp do danych zwracanych w kontrolerze z użyciem Model już w samym HMTL-u.

Warstwa JavaScript

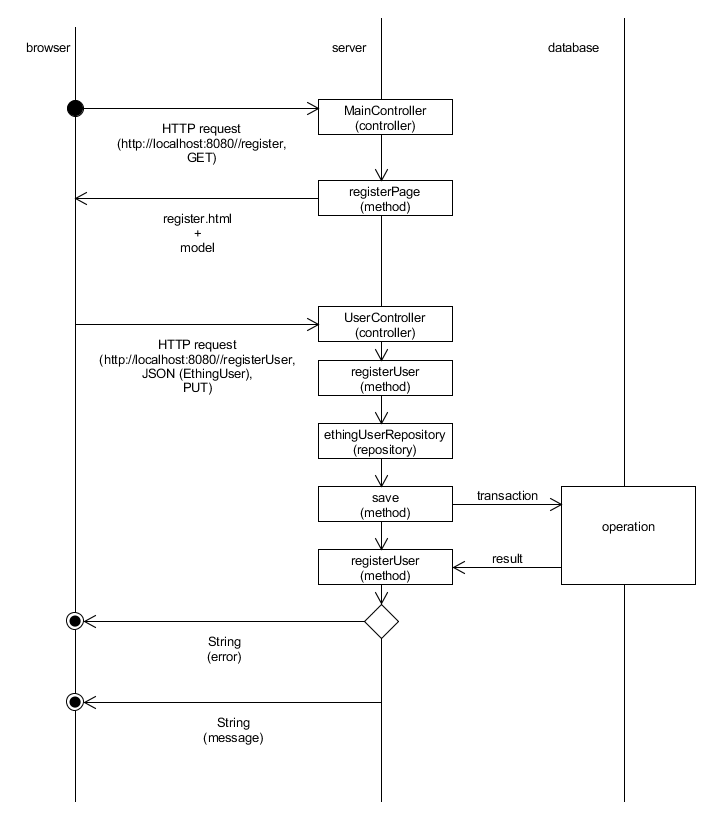
Logika stron oprogramowana jest za pomocą skryptów JS. Implementują one funkcjonalności interfejsu graficznego jak i komunikacji klient-serwer z użyciem Ajax-a. Warstwa wspierana jest również przez jQuery oraz semantic UI.

* 1. Komunikacja klient/serwer

Komunikacja pomiędzy stroną serwerową i kliencką odbywa się na dwa sposoby:

* Z użyciem technologii Ajax - dane pobierane są z formularza HTML z użyciem JS i jQuery. Następnie upakowane są odpowiednim skryptem do obiektu analogicznego jak obiekt modelu. Kolejno dane zostają przekazane do funkcji wykorzystującej Ajax. Json wysłany na podstawie URL do kontrolera oraz mapowany na obiekt Java modelu. Następuje obsługa danych po stronie serwerowej.
* Z użyciem zależności thymeleaf tworzącej szablony HTML. W kontrolerze możliwe jest zwracanie szablonów na podstawie stringu. Dane mogą być zwracane, jako atrybut zmiennej Model i wykorzystane w szablonie. Ułatwia to dostęp do danych w postaci wypisania ich.
  1. Działanie systemu

Rejestracja użytkownika.



Po wprowadzeniu określonego URL wyświetlana jest strona HTML. Na podstawie adresu wywoływana jest metoda registerPage w MainController. Implementowana z użyciem adnotacji RequestMapping określającej adres http oraz metodę. Kluczowe są również wprowadzane do metody obiekty HttpServletRequest oraz Principal. Pierwszy pozwala określić adres URL bazowy aplikacji, drugi zaś pomaga w stwierdzeniu czy użytkownik jest zalogowany.

@RequestMapping(value = "/register", method = RequestMethod.GET)

public String registerPage(HttpServletRequest request, Model model, Principal principal) {

String mainPage = getMainPage(1, request);

if (principal == null) {

noLoginModelAttribut(model, mainPage);

} else {

loginModelAttribut(model, mainPage, principal);

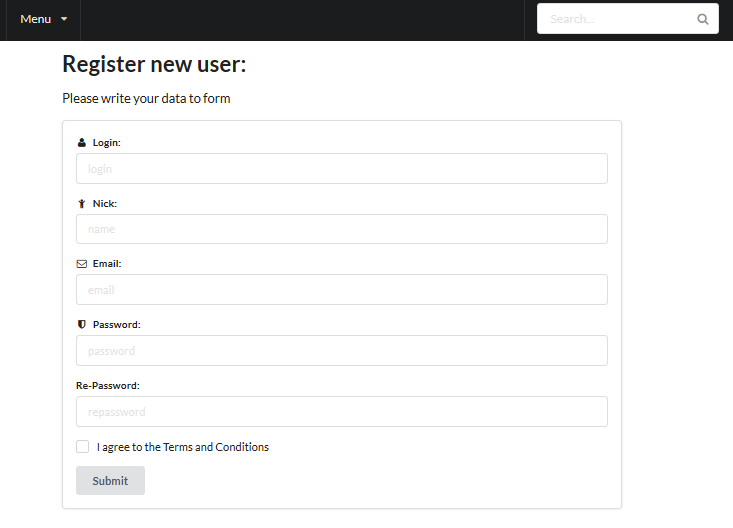
}

publicModelAttribut(model, mainPage);

return "register";

}

Zwraca ona wartość String a na podstawie jego wartości rezultatem jest strona HTML. Obsługiwana jest z użyciem thymeleaf. Wraz z tamplatem możliwe jest zwrócenie danych do modelu. Strona serwerowa zwraca formularz gotowy do wypełnienia przez użytkownik.



Ze strony klienta po odpowiednim sprawdzeniu danych w formularzu i oprawieniu ich w formę obiektu wywoływana jest funkcja sendUserData. Obsługuje ona działanie Ajax-a.

function sendUserData(userData)

{

var result = null;

$.ajax({

url: getMainPage()+"/registerUser",

type: "PUT",

data: JSON.stringify(userData),

async: false,

dataType: "json",

contentType: 'application/json',

success: function (response) {

result = response;

},

error: function (error)

{

result = error;

}

});

return result;

}

Obiekt wprowadzony zawiera dane nowo rejestrowanego użytkownika. Do najważniejszych atrybutów w funkcji ajax należą:

* url – adres HTTP obsługiwany przez metodę kontrolera warstwy serwerowej.
* type – typ żądania np. GET, POST,
* data – dane przesyłane z żądaniem,
* async – określenie synchronizacji żądania,
* dataType – typ danych przesyłanych z żądaniem,
* contentType – atrybut używany w przypadku wysyłania danych do serwera,
* success – funkcja wykonana w razie poprawnie wykonanego żądania,
* error – funkcja wykonana w razie błędu żądania.

Cała funkcja obsługuje komunikację od strony klienckiej.

Po wysłaniu żądania uruchamiane jest działanie po stronie serwerowej, czyli metoda kontrolera obsługująca działanie na użytkownikach (UserController). Wywoływana jest metoda registerUser. Tak jak wcześniej stworzona zostaje metoda. Z wyjątkiem zmienia się adres URL oraz metoda. Zmienia się również charakter zwracanego obiektu. Nie jest to już szablon, lecz czysty String określający poprawność operacji.

@RequestMapping(value = "/registerUser", method = RequestMethod.PUT)

public @ResponseBody

String registerUser(HttpServletRequest request, @RequestBody EthingUser ethingUser, Model model) {

try {

String mainPage = new String(request.getRequestURL().

toString().substring(0, request.getRequestURL().

toString().lastIndexOf("/")));

EthingUser newEthingUser = new EthingUser();

newEthingUser.setLogin(ethingUser.getLogin());

PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

String hashedPassword = passwordEncoder.encode(ethingUser.getPassword());

newEthingUser.setPassword(hashedPassword);

newEthingUser.setName(ethingUser.getName());

newEthingUser.setEmail(ethingUser.getEmail());

newEthingUser.setEthingThingSet(null);

newEthingUser.setRole("USER");

int activation = (ethingUser.getLogin() + ethingUser.getPassword() + ethingUser.getEmail()).hashCode();

activation = Math.abs(activation);

newEthingUser.setActivation(activation + "");

this.ethingUserRepository.save(newEthingUser);

this.senderMail.sendEmailActivationUser(newEthingUser, mainPage);

return "message";

} catch (Exception e) {

System.out.printf(e.getMessage());

return "error";

}

}

Metoda implementuje użycie usług email związanych z rejestracją oraz repozytorium działających na bazie danych. Zapisywany jest nowy użytkownik przez save obiektu EthingUserRepository. Po wykonaniu operacji zwracany jest rezultat i odpowiednia wartość z metody kontrolera do strony klienckiej systemu. W HTML-u pojawia się adekwatna wiadomość wywołana z JS.

Dodawanie przedmiotu.

W analogiczny sposób działa dodawanie przedmiotów. Wpierw zwracany jest na podstawie odpowiedniego URL szablon HTML. Z tą jednak różnica, że zwracane są dane dotyczące typów przedmiotu. W metodzie thingaddPage poza Stringiem określającym HTML do modelu przypisany zostaje rezultat metody findAll repozytorium EthingThingTypeRepository. Uzyskuje się stronę, która dzięki skryptom uzyskuje odpowiedni formularz. Następnie użytkownik wypełnia go i potwierdza.

@RequestMapping("/thingadd")

public String thingaddPage(HttpServletRequest request, Model model, Principal principal) {

String mainPage = getMainPage(1, request);

if (principal == null) {

noLoginModelAttribut(model, mainPage);

} else {

loginModelAttribut(model, mainPage, principal);

}

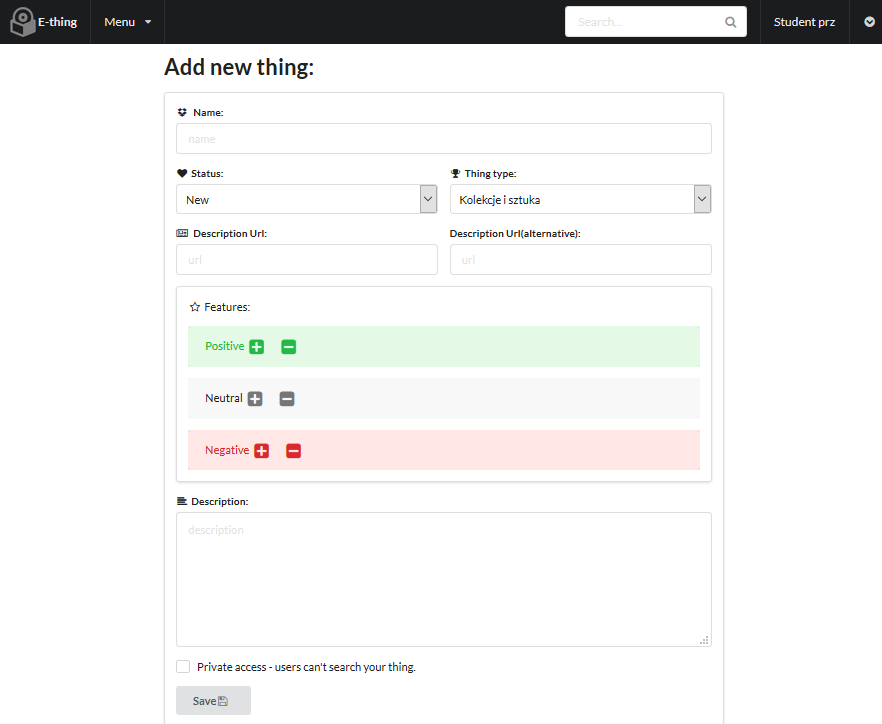
publicModelAttribut(model, mainPage);

List<EthingThingtype> thingType = ethingThingTypeRepository.findAll();

model.addAttribute("thingtype", thingType);

return "thingadd";

}



Po raz kolejny wysyłane zostaje żądanie http z odpowiednim protokole i danych JSON. W metodzie addThing kontrolera ThingController wywoływane są kolejno metody repozytoriów. Cały schemat operacji prezentują poniższe diagramy. Określa się poprawność operacji na bazie danych z użyciem metod repozytoriów i zwracany jest rezultat to przeglądarki obsługiwany przez skrypty.

Tak jak w przypadku rejestracji przy dodawaniu przedmiotu skorzystano z odpowiednio zaimplementowanej w js funkcji ajax-owej. Zmieniony zostaje jednak adres Http przyporządkowany do obsługi odebrania danych związanych z przedmiotem.

function sendThingData(thingData)

{

var result = null;

$.ajax({

url: getMainPage() + "/addThing",

type: "PUT",

data: JSON.stringify(thingData),

async: false,

dataType: "json",

contentType: 'application/json',

success: function (response) {

result = response;

},

error: function (error)

{

result = error;

}

});

return result;

}

@RequestMapping(value = "/addThing", method = RequestMethod.PUT)

public @ResponseBody

String addThing(HttpServletRequest request, @RequestBody EthingThingAndFeatures ethingThingAndFeatures, Principal principal) {

try {

EthingThing newThing = ethingThingAndFeatures.getEthingThing();

newThing.setEthingThingimageSet(null);

int idhash = (newThing.getName() + newThing.getStatus() + newThing.getUrl()).hashCode();

idhash = Math.abs(idhash);

newThing.setIdhash(idhash + "");

EthingThingtype thingType = ethingThingTypeRepository.findEthingThingtypeByName(newThing.getThingtype().getName());

newThing.setThingtype(thingType);

EthingUser thingUser = ethingUserRepository.findEthingUserByLoginAndActivation(principal.getName(), "1");

newThing.setUserid(thingUser);

EthingThing thing = ethingThingRepository.save(newThing);

List<EthingFeature> ethingFeatures = ethingThingAndFeatures.getEthingFeatures();

for (Iterator<EthingFeature> iterator = ethingFeatures.iterator(); iterator.hasNext();) {

EthingFeature next = iterator.next();

next.setThingid(thing);

}

ethingFeatureRepository.save(ethingFeatures);

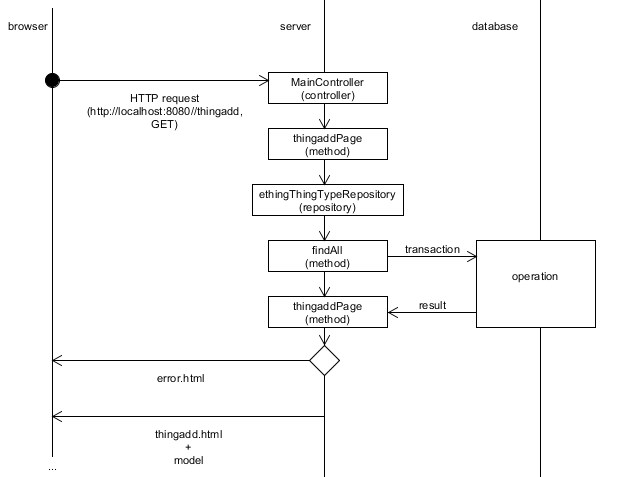
return "message";

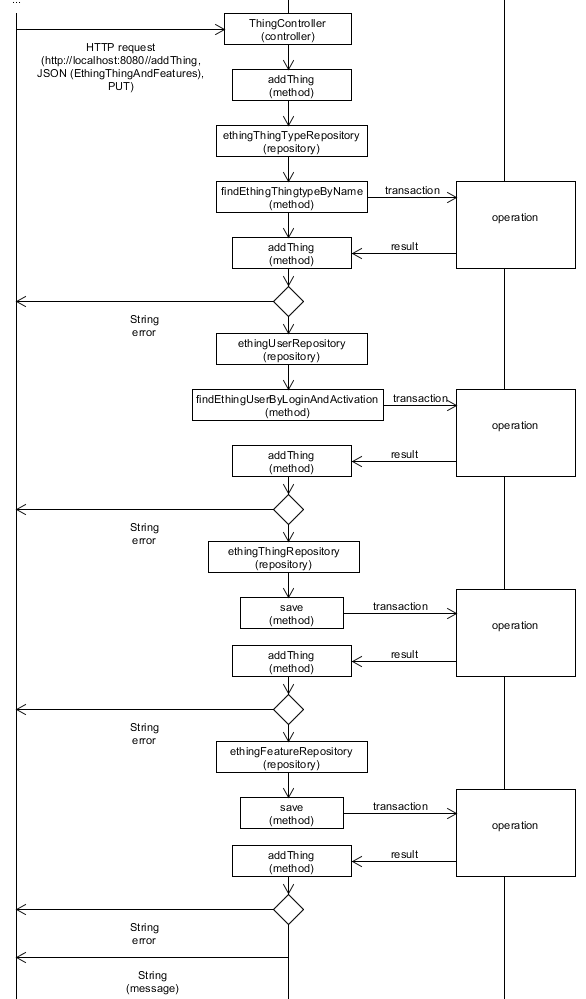
} catch (Exception e) {

return "error";

}

}





# Wnioski

Celem projektu było zrealizowanie systemu (aplikacji internetowej) pozwalającej na ogólna dokumentacje przedmiotów na sprzedaż. Wstępnie planowane funkcjonalności zostały zaimplementowane oraz zrealizowano wszystko do poprawnego działania oraz zahostowania aplikacji.