KAWA spring app

Aplikacja do zarządzania kawiarnią, możemy stworzyć listę oferowanych produktów, ta lista pojawi się na stronie dodawania zamówienia.

Na stronie dodawania zamówienia, możemy dodać klienta i jego zamówienie.

Po zakończeniu zamówienia możemy go zrealizować (w rzeczywistości usunąć go z listy)

Najpierw utworzono projekt Maven, i wszystkie zależności zostały zapisane w `pom.xml`

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>

   <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Jest link do Spring JPA, a także do Spring Web. Używam również silnika szablonów Thymeleaf (zastępując jsp w aplikacjach Spring MVC). Korzystam z bazy danych H2 w pamięci.

Istnieją trzy modele: `Produkt, Customer` i `CustomerOrder.`

`Customer.java`

```
package com.model;


import javax.persistence.*;



@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private long customerId;

    private String firstName;

    private String lastName;

    public long getCustomerId() {
        return customerId;
    }

    public void setCustomerId(long customerId) {
        this.customerId = customerId;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

CustomerOrder.java

Zestaw produktów w tabeli zamówień ma relacji wiele do wielu, klient ma relacje jeden do jednego. Najważniejsze jest to, że zamówienie może zawierać kilka produktów, a kilka zamówień może zawierać produkt.

CascadeType do scalenia, ponieważ kiedy usuwamy zamówienie sprzedaży, chcemy, aby zostało usunięte oddzielnie od klienta lub produktu. Jeśli tak się nie zrobić, to wraz z zamówieniem klient i produkt zostaną usunięte z bazy danych

```
package com.model;

import javax.persistence.*;
import java.util.Set;

@Entity
public class CustomerOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long orderId;

    private Double total;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE} , fetch =
FetchType.LAZY)
    @JoinTable(name = "ORDER_PRODUCTS", joinColumns = {@JoinColumn(name =
"ORDER_ID")},inverseJoinColumns = {@JoinColumn(name = "PRODUCT_ID")})
    private Set<Product> products;
```

```
    @OneToOne
    private Customer customer;

    public Long getOrderId() {
        return orderId;
    }

    public void setOrderId(Long orderId) {
        this.orderId = orderId;
    }


    public Double getTotal() {
        return total;
    }

    public void setTotal(Double total) {
        this.total = total;
    }

    public Set<Product> getProducts() {
        return products;
    }

    public void setProducts(Set<Product> products) {
        this.products = products;
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

}
```

Product.java

```
package com.model;

import javax.persistence.*;

@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long productId;

    private String productName;

    private Double productPrice;

    public Long getProductId() {
        return productId;
    }

    public void setProductId(Long productId) {
        productId = productId;
    }
```

```
    public String getProductName() {
        return productName;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }

    public Double getProductPrice() {
        return productPrice;
    }

    public void setProductPrice(Double productPrice) {
        this.productPrice = productPrice;
    }


}
```

Cała klasa znajduje się pod adnotacją @Entity, która jest zdefiniowana w javax.

Następująca adnotacja @Id oznacza, że jest to klucz podstawowy w tabeli.

Adnotacja @GeneratedValue zgłasza, że klucz powinnien zostać wygenerowany automatycznie

Kontrolery

Mamy dwa kontrolery OrdersControler i ProductsControler.

`ProductsController.java`

```
package com.controller;

import com.model.Product;
import com.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;


@Controller
public class ProductsController {

    @Autowired
    ProductRepository productRepository;


    @RequestMapping("/product/{id}")
    public String product(@PathVariable Long id, Model model){
        model.addAttribute("product", productRepository.findOne(id));
        return "product";
    }

    @RequestMapping(value = "/products",method = RequestMethod.GET)
    public String productsList(Model model){
        model.addAttribute("products", productRepository.findAll());
        return "products";
    }

    @RequestMapping(value = "/saveproduct", method = RequestMethod.POST)
    @ResponseBody
    public String saveProduct(@RequestBody Product product) {
```

```
        productRepository.save(product);
        return product.getProductId().toString();
    }

}
```

Za pomocą tej adnotacji framework Spring znajdzie pożądaną bean i zastąpi go wartość we właściwości, które jest oznaczone adnotacjej @Autowired.

@Controller identyfikuje tę klasę jako kontroler

@RequestMapping pozwala zdefiniować trasę HTTP, a także akcję dla każdej metody

@ResponseBody tworzy komunikat do wysłania jako proste String wartości lub może to być XML i JSON

Repozytoria

`ProductRepository.java`

```
package com.repository;

import com.model.Product;
import org.springframework.data.repository.CrudRepository;


public interface ProductRepository extends CrudRepository<Product, Long> {

}
```

Ten interfejs rozszerza CrudRepository. Metody, których używamy, takie jak findOne () delete () itp., Są wszystkie zdefiniowane w tym interfejsie, dlatego nie musimy pisać implementacji dla tych metod CRUD

`OrderController.java`

```
package com.controller;

import com.model.Customer;
import com.model.CustomerOrder;
import com.model.Product;
import com.repository.CustomerRepository;
import com.repository.OrderRepository;
import com.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.HashSet;
import java.util.Set;
```

```
@Controller
public class OrdersController {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private CustomerRepository customerRepository;

    @RequestMapping(value = "/orders", method = RequestMethod.GET)
    public String productsList(Model model){
        model.addAttribute("products", productRepository.findAll());
        model.addAttribute("orders", orderRepository.findAll());
        return "orders";
    }

    @RequestMapping(value="/createorder", method = RequestMethod.POST)
    @ResponseBody
    public String saveOrder(@RequestParam String firstName, @RequestParam String
lastName, @RequestParam(value="productIds[]") Long[] productIds){

        Customer customer = new Customer();
        customer.setFirstName(firstName);
        customer.setLastName(lastName);
        customerRepository.save(customer);
        CustomerOrder customerOrder = new CustomerOrder();

customerOrder.setCustomer(customerRepository.findOne(customer.getCustomerId()));
        Set<Product> productSet = new HashSet<Product>();
        for (Long productId:productIds){
            productSet.add(productRepository.findOne(productId));
        }
        customerOrder.setProducts(productSet);
        Double total = 0.0;
        for (Long productId:productIds){
            total = total + (productRepository.findOne(productId).getProductPrice());
        }
        customerOrder.setTotal(total);
        orderRepository.save(customerOrder);

        return customerOrder.getOrderId().toString();
    }


    @RequestMapping(value = "/removeorder", method = RequestMethod.POST)
    @ResponseBody
    public String removeOrder(@RequestParam Long Id){
        orderRepository.delete(Id);
        return Id.toString();
    }

}
```

Мы вызываем метод delete, передающий Id, и он сопоставляется с маршрутом removeorder(удаления заказа). И идентификатор передается в контроллер из jQuery ajax, который может видеть в @RequestParam

Wyzywamy metodę delete, która przekazuje identyfikator, i mapuje ona na trasę removeorder(usunięcie zamowienia). I identyfikator jest przekazywany do kontrolera z jQuery ajax, które można zobaczyć w @RequestParam.

`Funkcja Compeleteorder, która jest wywoływana z jQuery AJAX:`

```
.......
$('.delete-order').on("click", function(e){
    e.preventDefault();

    if(confirm("Czy na pewno zamówienie jest kompletne?")){
        var Id = parseInt($(this).closest("td").attr("id"));

        $.ajax({
            type:"POST",
            url:"/removeorder",
            data:{Id:Id},
            success:function (data) {
                $(".delete-
order").closest("td#"+data).parent("tr").fadeOut("slow",function(){
                    $(".delete-order").closest("td#"+data).parent("tr").remove();
                });
.........
```

Powyższy przykład to prosta procedura obsługi jQuery do klikania przycisku.

Po naciśnięciu przycisku otrzymuje identyfikator przycisku, który jest wciśnięty w DOM.

Identyfikator został wygenerowany za pomocą kontrolera poprzez dodawanie dynamicznych id z bazy danych.

W taki sposób, identyfikator-jest to jeden, który jest mapowany do rzeczywistej bazie danych, a następnie przekazujemy ID i sterownik wyłącza się, a my odpowiemy temu ze zdalnym ID.

Używamy z jQuery animacje aby usunąć tego, wierszy w tabeli.

Kod HTML strony i tabeli produktów, których Thymeleaf używa do tworzenia dynamicznej tabeli

Mam kilka produktów zarejestrowanych w głównym pliku `KavaAplication.java`

`KavaAplication.java`

```
@Override
public void run(String... strings) throws Exception {

    Product americano = new Product();
    americano.setProductName("Americano");
    americano.setProductPrice(3.95);
```

```
    Product capuccinno = new Product();
    capuccinno.setProductName("Capuccinno");
    capuccinno.setProductPrice(4.95);

    Product latte = new Product();
    latte.setProductName("Latte");
    latte.setProductPrice(5.00);

    Product espresso = new Product();
    espresso.setProductName("Espresso");
    espresso.setProductPrice(3.5);

    Product herbata = new Product();
    herbata.setProductName("Herbata");
    herbata.setProductPrice(2.00);

    productRepository.save(americano);
    productRepository.save(capuccinno);
    productRepository.save(latte);
    productRepository.save(espresso);
    productRepository.save(herbata);


}
```

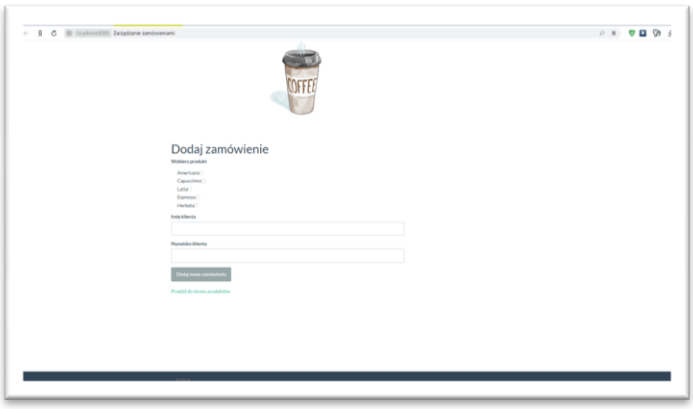
Kod paterna Thymeleaf z products.html

```
<table class="table table-striped table-hover">
    <thead>
    <tr>
        <th>Imę</th>
        <th>Cena</th>
    </tr>
    </thead>
    <tbody>

    <tr th:each="product : ${products}">
        <td th:text="${product.productName}"></td>
        <td th:text="${product.productPrice}"></td>
    </tr>
    </tbody>
</table>
```

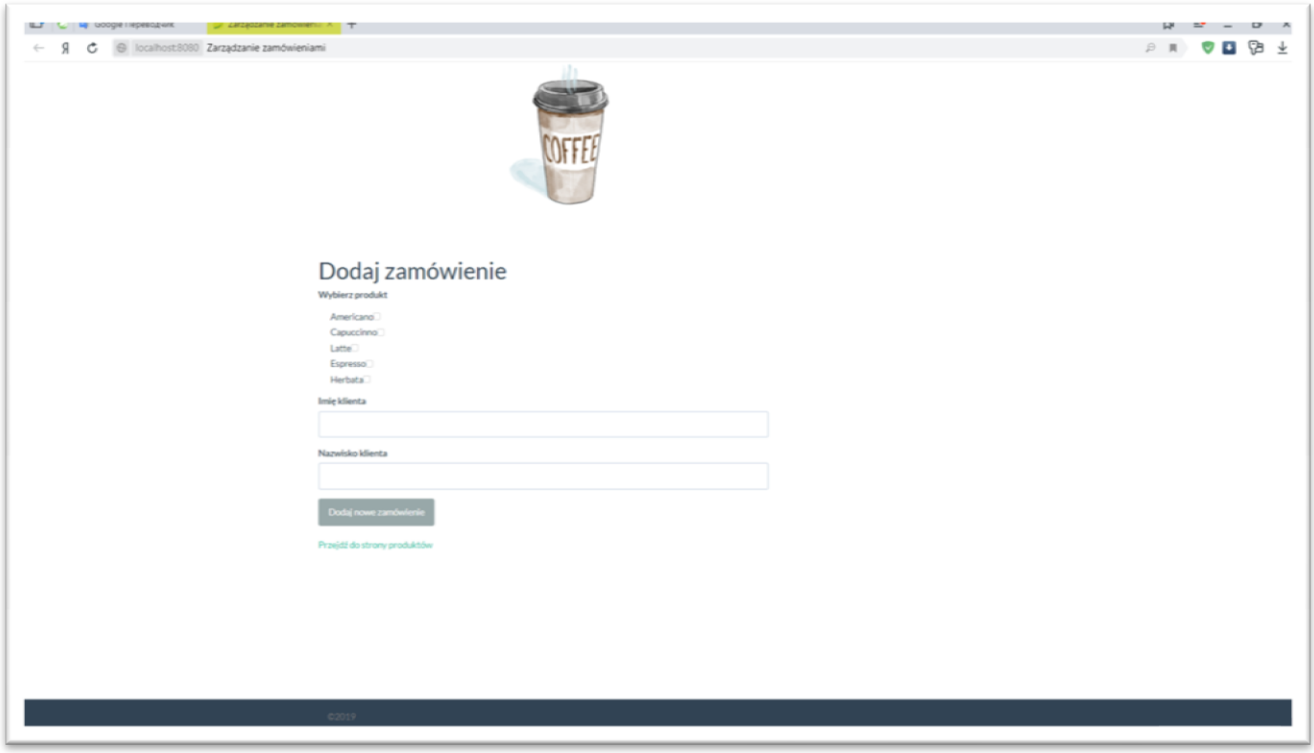
Używamy dialektu Thymeleaf, aby uzyskać dane modelu ze controlera i używamy Thymeleaf. Każda konstrukcja jest pętlą foreach.

Dla widoku aplikacji korzystał bootstrap.

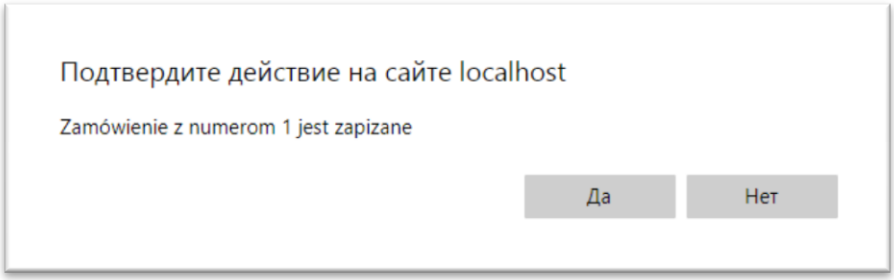


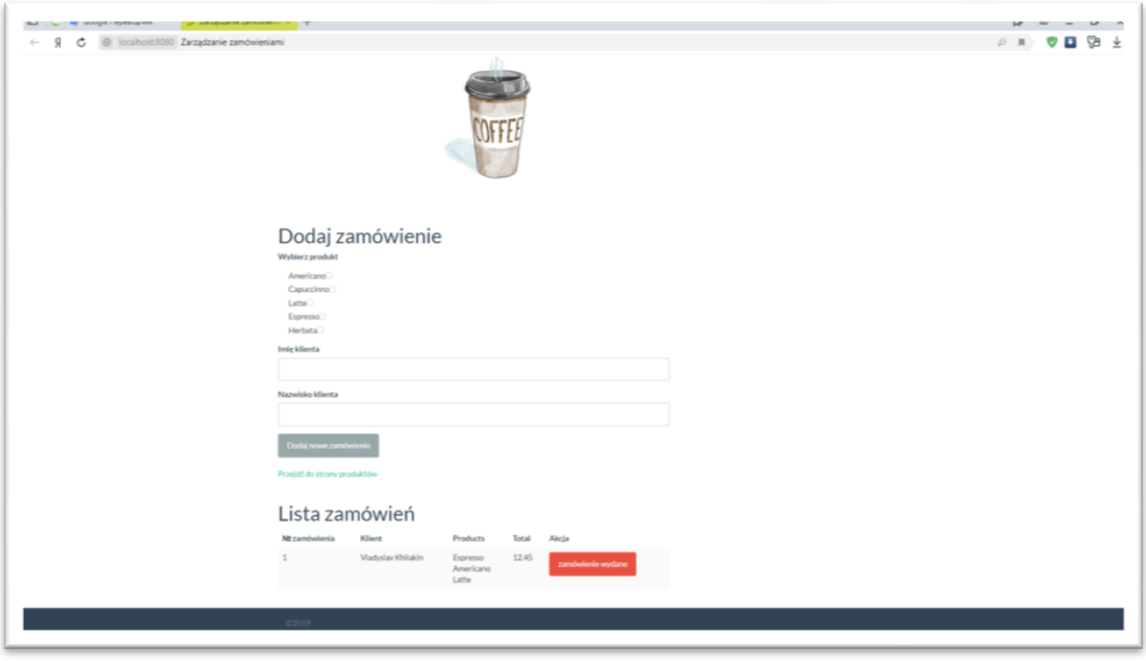
Zrzut aplikacji

Strona główna

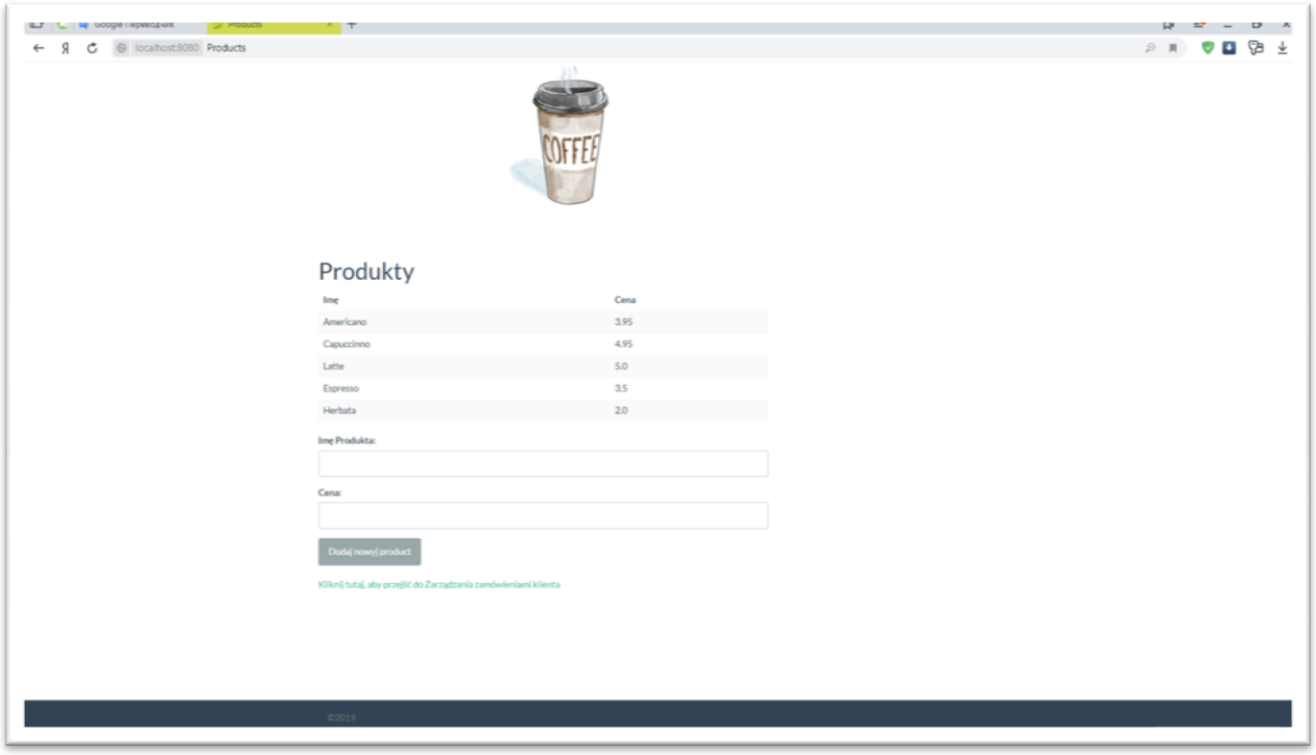


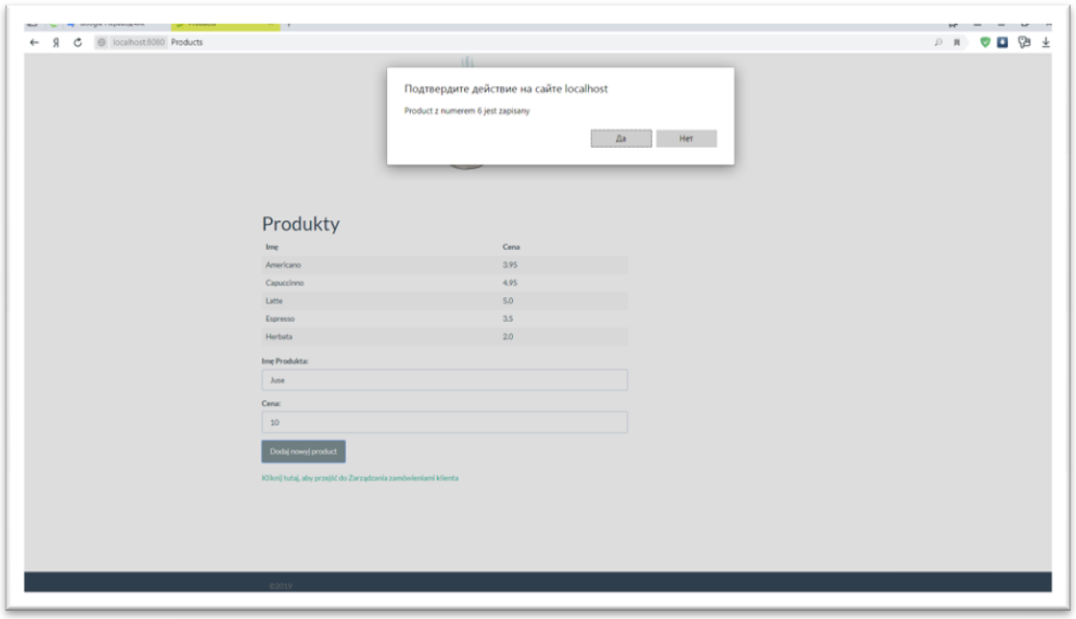
Okno potwierdzenia zamówienia





Okno dodawania nowego produktu





Produkty

Imę	Cena
Americano	3.95
Capuccinno	4.95
Latte	5.0
Espresso	3.5
Herbata	2.0
Juse	10.0

Imę Produkta:

Cena:

Dodaj nowyj product

Kliknij tutaj, aby przejść do Zarządzania zamówieniami klienta