# РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## 4.1 Быстрое добавление в корзину

Для удобства бизнес-покупателей в приложении необходимо реализовать возможность создания быстрого заказа, что позволит быстро выбрать и сразу сформировать корзину из нужных устройств.

В качестве реализации механизма быстрого добавления в корзину были написаны методы: makeQuickOrder(quickOrderElementsDto: QuickOrderElementsDto, bindingResult: BindingResult, redirectAttributes: RedirectAttributes, model: Model), addToCart(quickOrderElementsDto: QuickOrderElementsDto, cart: Cart, bindingResult: BindingResult, indexInList: int) класса QuickOrderController.

Для обработки переданного пользователем запроса используется метод makeQuickOrder(quickOrderElementsDto: QuickOrd- erElementsDto, bindingResult: BindingResult, redirectAttributes: RedirectAttributes, model: Model). Рассмотрим его алгоритм по шагам:

Параметры функции makeQuickOrder(quickOrderElementsDto QuickOrderElementsDto, bindingResult: BindingResult, redirectAttributes: RedirectAttributes, model: Model):

* quickOrderElementsDto – объект, представляющий запрос

пользователя, состоящий из моделей устройств и соответственно количества устройств для добавления;

– bindingResult – объект, заполняемый возникшими ошибками;

* redirectAttributes – объект, заполняемый параметрами,

которые требуется передать при перенаправлении при успехе;

– model – в этот объект передаются атрибуты, которые будут использованы при визуализации.

Для реализации метода makeQuickOrder() используются следующие входные данные:

* httpSessionCartService – объект, предоставляющий

инструменты работы с сессионной корзиной пользователя.

Шаг 1. Получение сессионной корзины пользователя.

Cart cart = httpSessionCartService.getCart(httpSession);

В случае уже существующей корзины, она будет получена из http-сессии, если такой атрибут сессии пуст(корзина пуста), будет создан новый объект корзины и сохранен в http-сессии.

Шаг 2. Цикл поочерёдного добавления элементов запроса в корзину.

quickOrderElementsDto.getQuickOrderElements() .forEach(quickOrderElement ->

{  
 int indexInList = quickOrderElementsDto

.getQuickOrderElements()

.indexOf(quickOrderElement);  
 addToCart(quickOrderElementsDto, cart, bindingResult, indexInList);  
});

В цикле происходит попытка добавления каждого элемента запроса в корзину, высчитывается индекс элемента (для позиционирования ошибки). Добавление происходит с помощью служебного метода addToCart().

Шаг 3. Проверка на наличие ошибок и сохранение аттрибутов.

После цикла поочерёдного добавления в корзину, происходит проверка на наличие ошибок, если ошибки есть в модель будут добавлены, как успешные результаты, так и ошибочные.

if (bindingResult.hasFieldErrors())  
{  
 model.addAttribute("successfulPhones", phoneList);  
 model.addAttribute("errors", bindingResult);  
 return PhoneshopPages.QuickOrderPage;  
}

Пользователь будет перенаправлен на ту же страницу, на которой будут отображены результаты добавления в корзину. В случае успешного добавления всех элементов запрос пользователя будет переадресован на ту же страницу, с отображением сообщения об успехе.

redirectAttributes.addFlashAttribute("success", true);  
return "redirect:" + PhoneshopPages.QuickOrderPage;

Шаг 4. Конец алгоритма.

В предыдущем методе используется служебный метод addToCart(quickOrderElementsDto: QuickOrderElementsDto, cart: Cart, bindingResult: BindingResult, indexInList: int). Он используется для добавления одного из элементов запроса из цикла и оптимизации количества кода. Рассмотрим его алгоритм подробнее: Параметры функции addToCart(quickOrderElementsDto: QuickOrderElementsDto, cart: Cart, bindingResult: BindingResult, indexInList: int):

* quickOrderElementsDto – объект, представляющий запрос

пользователя, состоящий из моделей устройств и соответственно количества устройств для добавления;

* cart – объект, представляющий сессионную корзину пользователя;

– bindingResult – объект, заполняемый возникшими ошибками;

– indexInList – номер добавляемого элемента в списке.

Для реализации метода addToCart(quickOrderElementsDto: QuickOrderElementsDto, cart: Cart, bindingResult: BindingResult, indexInList: int)использовались следующие входные данные:

* phoneServiceImpl – объект предоставляющий инструменты для

работы с устройствами в системе;

* httpSessionCartService – объект, предоставляющий

инструменты для работы с сессионной корзиной пользователя.

Шаг 1. Получение значений требуемых для добавления в корзину.

final QuickOrderElement quickOrderElement = quickOrderElementsDto.getQuickOrderElements()

.get(indexInList);  
final Long quantity = quickOrderElement.getQuantity();  
final String modelPhone = quickOrderElement.getModel();

Значение modelPhone необходимо для поиска устройства в базе данных, quantity – количество добавляемых устройств.

Шаг 2. Проверка полученных значений на пустоту.

После получения значений происходит проверка на их пустоту:

if (quantity == null && modelPhone.isEmpty())  
{  
 return;  
}

Если оба поля пустые, метод завершает работу (необходимо для создания опционально заполненного запроса). Иначе выполнение продолжается.

Шаг 3. Получение устройства из базы данных по названию его модели.

final Optional<Phone> optionalPhone =

phoneServiceImpl.getPhoneByModel(modelPhone);

Полученное значение будет использовано при добавлении в корзину.

Шаг 4. Добавление устройства в базу данных.

Изначально происходит проверка, получено ли успешно устройство. В случае наличия устройства происходит проверка на пустоту значения quantity. Если quantity не пустое происходит попытка добавления устройства в корзину, но если устройства нет в наличии произойдет исключительная ситуация, обрабатываемая блоком catch.

optionalPhone.ifPresent(phone ->  
{  
 if (quantity != null)  
 {  
 try  
 {  
 httpSessionCartService.addPhone(phone

.getId(), quantity, cart);  
 quickOrderElementsDto.getQuickOrderElements()

.set(indexInList, new QuickOrderElement());  
 }  
 catch (RuntimeException e)  
 {  
 bindingResult.rejectValue("quickOrderElements[" + indexInList + "].quantity",  
 "errors.quantity.outOfStock", "Out of stock!");  
 }  
 phoneList.add(phone);  
 }

else  
{  
 bindingResult.rejectValue("quickOrderElements[" + indexInList + "].quantity",  
 "errors.emptyField", "Empty field!");  
}

Шаг 5. Конец алгоритма.

**4.2 Регистрация пользователя**

При открытии приложения, неавторизованный пользователь может

просматривать информацию о товарах, искать нужный ему продукт. Однако

для совершения каких-либо действий направленных на заказ устройств, ему

требуется зарегистрироваться и авторизоваться.

В качестве реализации механизма регистрации были разработаны методы: userRegistration(userName: String, password1: String, model: Model) класса RegisterPageController, registerUser(userName: String, password: String) класса UserServiceImpl.

Для обработки запроса пользователя на регистрацию используется метод userRegistration(userName: String, password1: String, model: Model). Рассмотрим алгоритм по шагам:

Параметры функции userRegistration(userName: String, password1: String, model: Model):

– userName – переданное от клиента имя пользователя;

– password – переданный от клиента пароль;

– model – в этот объект передаются атрибуты, которые будут использованы при визуализации.

Для реализации метода userRegistration(userName: String, password1: String, model: Model) используются следующие входные данные:

* userServiceImpl – объект, предоставляющий инструменты

работы с пользователями приложения.

Шаг 1. Проверка параметров запроса на пустоту и соответствие значениям.

if (username == null || username.isEmpty() || password1 == null || password1.isEmpty() || username.length() < 5 || password1.length() < 5)  
{  
 model.addAttribute("errorEmpty", true);  
 return PhoneshopPages.UserPages.RegisterPage;  
}

Если один параметр, не соответствует заданным критериям (например пустота и длина строки), устанавливается атрибут ошибки и страница перезагружается, с сообщением об ошибке. Если все параметры соответствуют критерием выполнение кода продолжается.

Шаг 2. Попытка регистрации пользователя.

try  
{  
 userServiceImpl.registerUser(username, password1);  
}  
catch (final Exception e)  
{  
 model.addAttribute("error", true);  
 return PhoneshopPages.UserPages.RegisterPage;  
}

Происходит попытка добавления пользователя в систему, если пользователь успешно добавлен, то продолжается выполнение кода. В случае каких-либо ошибок, они будут перехвачены в блоке catch.

Шаг 3. Отправка сообщения об успехе.

После прохождения всех проверок, устанавливается атрибут успеха, пользователь перенаправляется на страницу для авторизации.

model.addAttribute("isRegistered", true);  
return PhoneshopPages.UserPages.LoginPage;

В предыдущем методе используется метод-сервис для добавления пользователей в систему registerUser(userName: String, password: String) класса UserServiceImpl. Рассмотрим его алгоритм подробнее:

Параметры функции registerUser(userName: String, password: String):

– userName – имя пользователя для регистрации;

– password – пароль для регистрации.

Для реализации метода userRegistration(userName: String, password1: String, model: Model) используются следующие входные данные:

* jdbcUserDao – объект, предоставляющий инструменты

работы с таблицей users в базе данных.

Шаг 1. Создание объекта сущности пользователя.

final User user = new User();  
user.setUserName(username);  
user.setIsAccountNonLocked(1);  
user.setRole("ROLE\_CUSTOMER");

Создаётся новая сущность пользователя, с именем пользователя переданным в метод. Новые пользователи всегда разблокированы. Назначается роль customer.

Шаг 2. Шифрование пароля.

Для безопасного хранения пароля пользователя в базе данных и защиты

от хакерских атак, пароль следует зашифровать.

final BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder(ENCRYPTION\_STRENGTH);

user.setPassword(bCryptPasswordEncoder.encode(password));

Шаг 3. Сохранения пользователя в базе данных.

jdbcUserDao.addNewUserToDb(user);

Объект пользователя сохраняется в базе данных, в случае если такой пользователь уже существует будет сгенерировано исключение, обработанное в методе, описанном выше.

* 1. **Поиск и сортировка устройств**

Для удобства пользователей в приложении реализована сортировка элементов по стоимости, по бренду, по модели и по размеру экрана. Также в приложении присутствует поиск элементов по названию модели.

В качестве реализации механизма поиска и сортировки устройств исполюзуются методы: showProductList(search: String, field: String, order: String, page: Long,model: Model) класса ProductListPageController, getPhones(paramsForSearch: ParamsForSearch) класса PhoneServiceImpl, findAll(paramsForSearch: ParamsForSearch) класса JdbcPhoneDao.

Для обработки переданным пользователем запросом используется метод showProductList(search: String, field: String, order: String, page: Long,model: Model). Рассмотрим его алгоритм по шагам:

Параметры функции showProductList(search: String, field: String, order: String, page: Long,model: Model):

– search – поисковый запрос пользователя;

– field – поле для сортировки;

– order – порядок сортировки;

* page – номер страницы;

– model – в этот объект передаются атрибуты, которые будут

использованы при визуализации.

Для реализации метода showProductList(search: String,

field: String, order: String, order: String, page: Long,

model: Model): используются следующие входные данные:

* phoneServiceImpl – объект, предоставляющий инструменты

работы с устройствами в системе;

* httpSessionCartService – объект, предоставляющий

инструменты работы с пользователями приложения.

Шаг 1. Проверка номера страницы.

if (page == null)  
{  
 page = 1L;  
}

Шаг 2. Проверка значения поля сортировки и порядка.

Так как запрос передаётся через поисковую строку браузера, пользователь может случайно написать любое значение в параметр order или field. Чтобы избегнуть поломок приложения происходит проверка на соответствие их заданным параметрам.

if (field != null && order != null)  
{  
 try  
 {  
 field = SortField.valueOf(field.toUpperCase(Locale.ROOT)).name();  
 order = SortOrder.valueOf(order.toUpperCase(Locale.ROOT)).name();  
 }  
 catch (IllegalArgumentException e)  
 {  
 field = null;  
 order = null;  
 }  
}

При несоответствии полученных полей, во избежание ошибок они устанавливаются как пустые.

Шаг 3. Определение смещения в списке устройств.

long offset = (page - 1) \* QUANTITY\_ON\_PAGE;

Смещение требуется для отображения на каждой странице разных

устройств.

Шаг 4. Установка вычисленных параметров поиска и получение списка

устройств.

final ParamsForSearch paramsForSearch = new ParamsForSearch(search,field, order, (int) offset,  
 QUANTITY\_ON\_PAGE.intValue());

final List<Phone> phoneList = phoneServiceImpl.getPhones(paramsForSearch);

Шаг 5. Вычисление количества доступных страниц.

final Long phoneQuantity = phoneServiceImpl.countPhoneQuantity(paramsForSearch);  
  
long numOfPages = phoneQuantity / QUANTITY\_ON\_PAGE;  
  
long lastPage;  
  
if (phoneQuantity % QUANTITY\_ON\_PAGE != 0)  
{  
 lastPage = numOfPages + 1;  
}  
else  
{  
 lastPage = numOfPages;  
}

Шаг 6. Сохранение атрибутов для отображения и отображение

страницы пользователя.

model.addAttribute("phones", phoneList);  
model.addAttribute("cart", cartService.getCart(httpSession));  
model.addAttribute("pages", lastPage);  
model.addAttribute("phoneQuantity", phoneQuantity);  
  
return PhoneshopPages.UserPages.ProductListPage;

Шаг 7. Конец алгоритма.

В описанном выше методе используется метод getPhones(paramsForSearch: ParamsForSearch) класса PhoneServiceImpl, который вызывает метод findAll(paramsForSearch: ParamsForSearch) класса JdbcPhoneDao. Рассмотрим его алгоритм:

Параметры функции findAll(paramsForSearch: ParamsForSearch):

– paramsForSearch – параметры для поиска и сортировки устройств;

Для реализации метода findAll(paramsForSearch: ParamsForSearch): используются следующие входные данные:

* jdbcTemplate – объект, который позволяет отправлять запросы

и упаковывать ответы от базы данных;

* phoneResultSetExtractor – объект, преобразующий

ответ базы данных в список телефонов;

* SQL\_GET\_ALL\_PHONES – базовый sql-запрос для получения

списка всех устройств из базы данных;

* SQL\_WHERE\_SEARCH – дополнение к sql-запросу, ставящее

условие для поиска только товаров в наличии.

Шаг 1. Инициализация значений для формирования sql-запроса.

final String search = paramsForSearch.getSearch();  
final String sortField = paramsForSearch.getSortField();  
final String order = paramsForSearch.getOrder();  
final int offset = paramsForSearch.getOffset();  
final int limit = paramsForSearch.getLimit();  
final List<Object> objects = new ArrayList<>();  
final List<Integer> types = new ArrayList<>();

String query = SQL\_GET\_ALL\_PHONES + SQL\_WHERE\_SEARCH;

Шаг 2. Проверка пустоты параметра search.

if (search != null && !search.isEmpty())  
{  
 query = query + "and lower(model) like lower(?) ";  
 objects.add("%" + search + "%");  
 types.add(Types.VARCHAR);  
}

В случае, если параметр запроса не пустой, sql-запрос дополняется им, и возвращаемый результат будет соответствовать условию поиска.

Шаг 3. Проверка параметра sortField и order.

if (sortField != null && order != null)  
{  
 query = query + String.format("group by phones.id,phone2color.colorId order by %s %s ", sortField, order);  
}

Если эти параметры не пустые, sql-запрос дополняется ими и устройства будут отсортированы в требуемом порядке.

Шаг 4. Установка смещения и ограничения поиска.

query = query + " limit " + limit + " offset " + offset;

Смещение и ограничение поиска устанавливаются для правильного отображения устройств постранично.

Шаг 5. Окончательное формирование и отправка sql-запроса.

final int[] typesArray = types.stream()  
 .mapToInt(i -> i)  
 .toArray();  
return jdbcTemplate

.query(query, objects.toArray(), typesArray, phoneResultSetExtractor);

После отправки запроса и получения ответа из базы данных, возвращается список устройств, соответствующий поиску и(или) с необходимыми сортировками.

Шаг 6. Конец алгоритма.