

# IT LAB REPORT

Shuvayan Ghosh Dastidar  
001810501044  
BCSE-III  
Third Year Second Semester  
2021

Assignment 4  
Design of an online apparel store using Spring  
Boot.

## Problem Statement

Design an online apparel store using either the *Spring framework* or *servlets*. The store keeps records for its items in a database where some items may be discounted and some other items should be displayed as “new arrivals”. A user may search for a specific item. By default, when a user signs in, based on his/her profile (male/female etc.), show him/her preferred set of clothing. Users will be divided into two groups: some users looking for discounted items mainly, some others looking for new arrivals. So, initially, depending on their preference already set in the database, the order of the displayed list would vary. By default, discounted items will be displayed first. This order may change depending on usage behavior at runtime.

You may apply the concept of “dependency injection” here. *Dependency injection (DI)* is a technique where one object supplies the dependencies of another object. Basically you have an interface and a number of java beans implementing them. The major benefit of DI is loose coupling and ease of use. DI makes classes more cohesive because they have fewer responsibilities. You should use `SessionListener` or `ServletContextListener`.

Four main features of the backend of your application should be – (i) session handling, (ii) database connectivity, (iii) event handling, and (iv) security (e.g. https). DI is made optional here.

.

## Tech Stack

The tech stack used are :

- **Spring Boot** : Spring Boot helps in creating stand-alone Spring applications. It has the facility of embedding Tomcat, Jetty or Undertow directly without the need of deploying WAR files. It also provides Dependency Injection that allows for the loose coupling of the components.

- **ReactJS** : ReactJs is a front-end framework which uses the concept of a virtual Dom for manipulating content on a single HTML page. Websites made with ReactJs and are hence single paged web applications. Due to it's extensive support and ease of development it has been widely used in web development and as well as mobile app development.
- **Postgres** : PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley.

## Structure of the code

```

.
├─ main
│   └─ java
│       └─ com
│           └─ flamup
│               └─ spring
│                   ├── Application.java
│                   ├── Controllers
│                   │   ├── AuthController.java
│                   │   ├── CartServiceController.java
│                   │   ├── ProductController.java
│                   │   ├── RegistrationController.java
│                   │   ├── SessionController.java
│                   │   └─ TestGetController.java
│                   ├── DTO
│                   │   ├── OrderDTO.java
│                   │   └─ UpdateOrderDTO.java
│                   ├── Models
│                   │   ├── AppUserRole.java
│                   │   ├── ApplicationUser.java
│                   │   ├── OrderItem.java
│                   │   ├── Product.java
│                   │   ├── RegistrationRequest.java
│                   │   └─ ShoppingCart.java
│                   ├── Repositories
│                   │   ├── ApplicationUserRepository.java
│                   │   ├── CartRepository.java
│                   │   ├── OrderRepository.java
│                   │   └─ ProductRepository.java
│                   ├── Services
│                   │   ├── CartService.java
│                   │   ├── CustomLogoutHandler.java
│                   │   ├── ProductService.java
│                   │   └─ RegistrationService.java
│                   ├── auth
│                   │   └─ AppUserService.java
│                   └─ security
│                       ├── PasswordEncoder.java
│                       ├── RestAuthEntryPoint.java
│                       ├── SwaggerConfig.java
│                       └─ WebSecurityConfig.java
└─ resources
    ├── application.properties
    ├── application.yml
    └─ data-postgres.sql

```

```

|      └─ static
└─ test
    └─ java
        └─ com
            └─ flamup
                └─ spring
                    └─ ApplicationTests.java

```

The following code has been made following the MVC design pattern. The models required for the project are kept under the models folder. It has also the controllers for communication of the API with the client requests and the services to provide a layer of abstraction between the models or the repositories and the controllers to actually manage the business logic of the application. Then finally we have the repositories which are responsible for performing DDL operations with the database through JDBC.

Let's look at each component closely,

For the spring Boot application, maven has been used as a build tool. Let's look at the dependencies.

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-jdbc</artifactId>
  </dependency>
  <dependency>

```

```

        <groupId>io.springfox</groupId>
        <artifactId>springfox-boot-starter</artifactId>
        <version>3.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

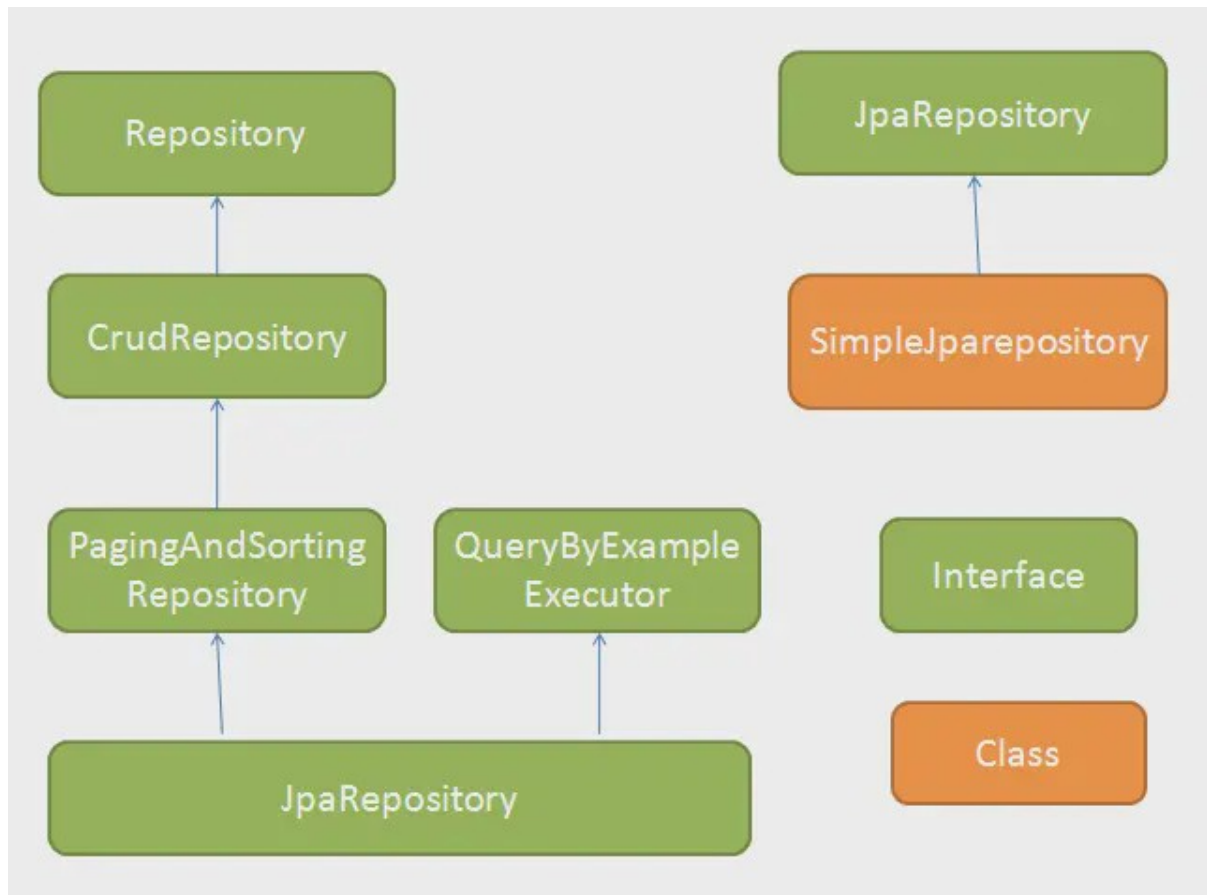
```

The project is structured as follows

**Models.** It contains the entity POJOs (Plain Old Java Object) that represents a table in a relational database which in our case is postgres. The `javax.persistence` package has many annotations that are suitable for making spring boot understand the details of a POJO and converting them to a table of relational database.

**Controllers.** This provides configuration for communication of the API with the outside world. All kinds of request mappings such as GET, PUT, POST, DELETE can be configured in the controllers and are used to make api endpoints for the outside world.

**Repositories.** These are usually interfaces which extend other classes, which provide some basic methods to work with the CRUD operations and other complicated tasks.



As the hierarchy, the `CrudRepository` is at the top, `Paging and Sorting repository` has the methods of `CrudRepository` and the support for pagination and sorting of results. The `JPA repository` is at the highest level and provides support for all kinds of functions that `CrudRepository` and `Paging Repository` provides.

**Services.** These are the actual classes which get called by the controllers and which in turn call the functions of the repositories to interact with the database. Thus services act as a bridge between the database or the repositories and the controllers or the world outside the API.

Let's jump into the code.

The user class has been made by implementing the UserDetails of spring security.

```
import lombok.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import javax.persistence.*;
import java.util.Collection;
import java.util.Collections;
```

```
@Entity
@NoArgsConstructor
@Data
public class ApplicationUser implements UserDetails {
```

```
    public enum Gender{
        MALE,
        FEMALE
    }
```

```
    @SequenceGenerator(
        name = "student_sequence",
        sequenceName = "student_sequence",
        allocationSize = 1
    )
```

```
    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "student_sequence"
    )
```

```
    private Long id;
    private String firstName;
    private String lastName;
    private String username;
```

```
    private String password;
    private String email;
    @Enumerated(EnumType.STRING)
    private AppUserRole appUserRole;
    @Enumerated(EnumType.STRING)
    private Gender gender;
```

```
    public ApplicationUser(String firstName,
                           String lastName,
                           String username,
                           String password,
                           String email,
                           AppUserRole appUserRole,
                           Gender gender) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.username = username;
        this.password = password;
        this.email = email;
```



```

        this.appUserRole = appUserRole;
        this.gender = gender;
    }

    public Long getId(){
        return id;
    }
    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;
    }

    public String getEmail() { return username; }

    public void setEmail(String email) { this.email = email; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        SimpleGrantedAuthority authority =
            new SimpleGrantedAuthority(appUserRole.name());
        return Collections.singletonList(authority);
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

```

```

@Override
public boolean isEnabled() {
    return true;
}

public void setPassword( String password){
    this.password = password;
}

public void setUsername( String username ){
    this.username = username;
}
}

```

However as we can see there are lot of repetitive getters and setters, which can be avoided by **Lombok** annotations, **@Getter** and **@Setter**. Here's the model class for the Product item.

```

import lombok.*;

import javax.persistence.*;

@Entity
@Table(name = "clothes")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Product {

    public enum sex{
        MALE,
        FEMALE
    }

    @Id
    Long id;
    @Enumerated(EnumType.STRING)

    @Column(name = "sex")
    sex a_sex;
    @Column(name = "dresstype")
    String b_dresstype;
    @Column(name = "image")
    String c_image;
    @Column(name = "price")
    String d_price;
    @Column(name = "arrival")
    String e_arrival;
    @Column(name = "discount")
    int f_discount;

}

```

Notice the “a\_”, “b\_” like syntax. This is done intentionally to make them get stored in the database in the order intended as Spring JPA stores the attributes as columns of the table in alphabetical order.

Here’s the model for each of the Order Item in a shopping cart which is specific to a single user( defined by his/her email). The ManyToOne annotation serves this purpose.

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.flamup.spring.DTO.OrderDTO;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

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

@Entity
@Table(name = "item")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class OrderItem {

    @Id
    private String id;

    @ManyToOne
    private Product product;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "cart_id" , nullable = false)
    @JsonIgnore
    private ShoppingCart cart;

    private int quantity;

    public void fromDto(Product p, ShoppingCart cart, int quantity ){
        id = UUID.randomUUID().toString().replace("-", "");
        product = p;
        this.cart = cart;
        this.quantity = quantity;
    }
}
```

And here's the model class for the **ShoppingCart**.

```
import lombok.*;

import javax.persistence.*;
import java.time.LocalDateTime;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "cart")
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@ToString
public class ShoppingCart {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "user_id")
    private String userId;

    @OneToMany( mappedBy = "cart", cascade = CascadeType.ALL, fetch =
FetchType.LAZY )
    private Set<OrderItem> items = new HashSet<>();
}
```

Now let's look at the repositories which accepts calls from services and works with the models.

Let's see how the authentication process works.

First we have the registration controller, which in calls the register method of the registrationService on getting a POST request to **"/api/register"**.

```
@RestController
@RequestMapping(path = "api/register")
@AllArgsConstructor
public class RegistrationController {

    private RegistrationService registrationService;

    @PostMapping
```

```

    public ResponseEntity<String> register(@RequestBody RegistrationRequest request)
    {
        return new
        ResponseEntity<String>(registrationService.register(request),HttpStatus.OK);
    }
}

```

The service has a function which checks for duplicate emails and thus encodes the plaintext password with a **password encoder** and then saves the user details on to the database. During login these details are checked for existence in the database.

```

public String signupUser( ApplicationUser applicationUser){
    boolean userExists = applicationUserRepository
        .findApplicationUsersByEmail(applicationUser.getUsername())
        .isPresent();

    if ( userExists ){
        throw new IllegalStateException("email already used, try logging in");
    }

    String encodedPassword =
    bCryptPasswordEncoder.encode(applicationUser.getPassword());
    applicationUser.setPassword(encodedPassword);

    System.out.println("IN appUserService" + applicationUser);
    applicationUserRepository.save(applicationUser);
    return "SUCCESS";
}

```

We will come later to the part of login as it will be a part of the spring security configuration.

Let's look at how **Pagination** happens for servicing GET requests to the server.

**Pagination greatly helps in reducing the load of the server.**

We have the products controller as

```

import javax.servlet.http.HttpServletRequest;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

```

```

import java.util.Map;
import java.util.stream.Collectors;

@RestController
@RequestMapping(path = "api/v1/")
@AllArgsConstructor
public class ProductController {

    private final ProductService productService;

    @GetMapping(path = "dresses")
    public HashMap<String, Object>
getProductsByDressTypeOnSearch(@RequestParam(name = "dress") String dress){
        HashMap<String, Object> hs = new HashMap<>();
        List<Product> products = productService.getProductsByDressType(dress);
        hs.put("products", products);
        hs.put("length" , products.size());
        return hs;
    }

    @GetMapping( path = "clothes")
    public HashMap<String, Object> getProductsBySexAndSession(
        @RequestParam(name = "sex") String sex,
        @RequestParam(defaultValue = "10") Integer items,
        @RequestParam(defaultValue = "0") Integer page,
        HttpServletRequest request){

        List<String> messages = (List<String>)
request.getSession().getAttribute("SESSION_STORE");

        if ( messages == null){
            messages = new ArrayList<>();
        }

        Map<Boolean, Long> countByType = messages.stream().collect(
            Collectors.partitioningBy(
                (String msg) -> (msg.equals("0")),
                Collectors.counting() ));

        return productService.getProductsBySex(sex, items, page, countByType);
    }

}

```

The function **getProductsBySexAndSession** takes in two optional parameters for the number of items in a page and the current page. The **countByType** counts

the number of interactions with the new apparel vs the old apparel. This calls a function of productService.

```
public HashMap<String, Object> getProductsBySex(String sex, Integer pageSize,
Integer page, Map<Boolean, Long> countBySex){
```

```
    Pageable paging = PageRequest.of(page , pageSize);
    Page<Product> prods = productRepository.findProductByA_sex(sex, paging);
    List<Product> newProds =
productRepository.findProductsByE_arrivalAndA_sex("new" ,sex);
    Integer totalPages = prods.getTotalPages();
    List<Product> products = prods.getContent();
    products = rearrange(products, countBySex, newProds);
    Integer current = page;
    HashMap<String, Object> hs = new HashMap<>();
    hs.put("current" , current);
    hs.put("products" , products);
    hs.put("total" , totalPages);
    return hs;
}
```

```
private List<Product> rearrange( List<Product> products, Map<Boolean, Long>
countBySex, List<Product> newProds){
```

```
    List<Product> ll = new ArrayList<>(products);
    Product ptemp = ll.get(0);
    ll.remove(ptemp);
    ll.add(0, newProds.get(0));
```

```
    if ( (long)( countBySex.get(true) + countBySex.get(false) ) == 0){
        return ll;
    }
```

```
    Integer toShift =
        (int)(long)( (double) countBySex.get(false) / (countBySex.get(true) +
countBySex.get(false)) * products.size() ) ;
```

```
    System.out.println("toShift" + toShift);
    int pivot = Math.max( 0 , (int) (Math.random() * 50) - 5 );
    for ( int start =0; start < Math.min(Math.min(Math.min( products.size() ,
toShift), newProds.size() ), 7) ; start++){
        Product p = ll.get(start );
        ll.remove(p);
        ll.add(0, newProds.get(start + pivot ));
    }
    return ll;
}
```

Above is the algorithm for returning product Items based on number of items per page and their interaction of the associated website. Below is the code for the productRepository.

```
@Repository
@Transactional
public interface ProductRepository extends JpaRepository<Product, Long> {

    Optional<Product> findProductById( Long id);

    @Query("select p from Product p where LOWER(p.b_dresstype) LIKE
LOWER(CONCAT('%', :type, '%'))")
    List<Product> findProductsByB_Dresstype(@Param("type") String type);

    @Query(value = "select p from Product p where LOWER(p.a_sex)=lower(:sex) order
by p.e_arrival desc ")
    Page<Product> findProductByA_sex(@Param("sex") String sex, Pageable paging);

    @Query(value = "select p from Product p where
LOWER(p.e_arrival)=LOWER(:arrival) and LOWER(p.a_sex)=LOWER(:sex)")
    List<Product> findProductsByE_arrivalAndA_sex(@Param("arrival") String arrival,
@Param("sex") String sex);

}
```

Here the query annotation is used for writing custom queries for our purposes. Spring boot has the support for native queries, ie, which include raw SQL statements rather than Hibernate created SQL statements.

Thus we are only left with the shopping cart service, which has the functions of adding an item in the cart, updating the quantity of the item in the cart and deleting the item from the cart.

Here's the controller for defining those endpoints to which the backend will be called,



```

@RestController
@RequestMapping(path = "api/v1/")
@AllArgsConstructor
public class CartServiceController {

    private final CartService cartService;

    @PostMapping( path = "add")
    public ResponseEntity<String> addProduct(@RequestBody OrderDTO orderDTO ){
        return new ResponseEntity<String>(cartService.addToCart(orderDTO) ,
        HttpStatus.CREATED);
    }

    @GetMapping( path = "orders")
    public ResponseEntity<HashMap<String , Object>> getOrders(){
        return new ResponseEntity<>(cartService.getProducts(), HttpStatus.OK);
    }

    @PutMapping(path = "order")
    public ResponseEntity<String> updateOrder(@RequestBody UpdateOrderDTO
updateOrderDTO){
        return new ResponseEntity<>(cartService.updateOrder( updateOrderDTO ),
        HttpStatus.CREATED);
    }

    @DeleteMapping( path = "order/{id}")
    public ResponseEntity<String> deleteOrder(@PathVariable String id){
        return new ResponseEntity<>(cartService.deleteOrder(id), HttpStatus.OK);
    }

}

```

We have two Data Transfer Objects(DTO). A DTO is used to transfer information from the client side and between services of the API.

Let's define those,

```

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class OrderDTO {

    private Long productId;
    private int quantity;

}

@Getter
@Setter
@NoArgsConstructor

```

```
@AllArgsConstructor
public class UpdateOrderDTO {
    private String id;
    private Integer quantity;
}
```

Now let's take a look at the cart service,  
As we can see it has all the support for **POST, GET, PUT, DELETE** making it a perfect example.

```
@Service
public class CartService {
```

```
    private final CartRepository cartRepository;
    private final ProductRepository productRepository;
    private final OrderRepository orderRepository;
```

```
    @Autowired
    public CartService(CartRepository cartRepository,
                      ProductRepository productRepository,
                      OrderRepository orderRepository
                      ) {
        this.cartRepository = cartRepository;
        this.productRepository = productRepository;
        this.orderRepository = orderRepository;
    }
```

```
    public String addToCart(OrderDTO order){
        Object principal =
        SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

```
        if ( principal instanceof UserDetails) {
```

```
            Optional<ShoppingCart> cartFromRepo =
            cartRepository.findById(((UserDetails) principal).getUsername());
            ShoppingCart cart;
            if ( ! cartFromRepo.isPresent() ){
                cart = new ShoppingCart();
                cart.setUserId(((UserDetails) principal).getUsername());
                cartRepository.save(cart);
            }
            else{
                cart = cartFromRepo.get();
            }
        }
```

```
        OrderItem item = new OrderItem();
        Product pt = productRepository.findProductById(order.getProductId())
            .orElseThrow( () -> new IllegalStateException("product not
            found"));
        item.fromDto(pt, cart, order.getQuantity());
```

```

        orderRepository.save(item);
        return "SUCCESS";
    }
    else{
        throw new IllegalStateException("user not authenticated");
    }
}

```

```

private Integer sanitizePrice( String price) {
    // price with Rs.2,244 --> 2244
    price = price.substring(3).replace(",", "");
    return Integer.parseInt(price);
}

```

```

public HashMap<String, Object> getProducts(){
    Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    HashMap<String, Object> hs = new HashMap<>();
    hs.put("products" , new ArrayList<>());
    hs.put("total" , 0);
    if ( principal instanceof UserDetails) {
        Optional<ShoppingCart> cartFromRepo =
cartRepository.findById(((UserDetails) principal).getUsername());
        ShoppingCart cart;
        if ( ! cartFromRepo.isPresent() ){
            return hs;
        }
        cart = cartFromRepo.get();
        ArrayList<OrderItem> orders = new
ArrayList<>(orderRepository.findById(cart.getId()));
        hs.put("products" , orders);
        int price =0;
        for ( OrderItem order : orders ){
            if
( order.getProduct().getE_arrival().toLowerCase(Locale.ROOT).equals("old")){
                price += order.getQuantity() *
order.getProduct().getF_discount();
            }
            else {
                price += order.getQuantity() *
sanitizePrice(order.getProduct().getD_price());
            }
        }
        hs.put("total", price);
        return hs;
    }
    else{
        throw new IllegalStateException("user not authenticated");
    }
}

```

```
}
```

```
public String updateOrder(UpdateOrderDTO updateOrderDTO){
    OrderItem item = orderRepository.findById(updateOrderDTO.getId())
        .orElseThrow(()->new IllegalStateException("order does not
exist"));
```

```
    item.setQuantity(updateOrderDTO.getQuantity());
    orderRepository.save(item);
    return "SUCCESS";
}
```

```
public String deleteOrder( String id){
    OrderItem item = orderRepository.findById(id)
        .orElseThrow(()->new IllegalStateException("order does not
exist"));
    orderRepository.delete(item);
    return "DELETED";
}
```

```
}
```

Now with the services covered, let's define the spring security configuration which will define the **session management and user login and registration to our web-app**. Now we will perform user login through JDBC authentication, there are many other methods of authentication which one can check out in the docs. Spring security is a vast topic and it's worth to learn it. Here's the security config,

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    private final PasswordEncoder passwordEncoder;
    private final AppUserService appUserService;

    @Autowired
    RestAuthEntryPoint restAuthEntryPoint;

    @Autowired
    private CustomLogoutHandler logoutHandler;

    @Autowired
    public WebSecurityConfig(PasswordEncoder passwordEncoder, AppUserService
appUserService) {
```

```

        this.passwordEncoder = passwordEncoder;
        this.appUserService = appUserService;
    }

```

```

@Override
public void configure(WebSecurity registry) throws Exception {
    registry.ignoring()
        .antMatchers("/docs/**")
        .antMatchers("/actuator/**")
        .antMatchers("/swagger-ui.html")
        .antMatchers("/webjars/**");
}

```

```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.authenticationProvider(daoAuthenticationProvider());
}

```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.
        csrf().disable()
        .authorizeRequests()
        .antMatchers("/", "/static/**", "index*", "/css/**", "/js/**", "/media/
*", "*.ico", "*.png").permitAll()
        .antMatchers("/api/register").permitAll()
        .antMatchers("/api/auth").permitAll()
        .antMatchers("/api/persist/**").authenticated()

        .antMatchers("/api/v1/**").authenticated()
        .antMatchers("/admin/api/**").hasRole(ADMIN.name())
        .and()
        .exceptionHandling()
        .authenticationEntryPoint(restAuthEntryPoint)
        .and()
        .formLogin()
            .loginProcessingUrl("/api/login")
            .permitAll()
        .and()
        .logout()
            .logoutUrl("/api/logout")
            .invalidateHttpSession(true)
            .deleteCookies("JSESSIONID")
            .clearAuthentication(true)
            .addLogoutHandler(logoutHandler)
            .logoutSuccessHandler(new
HttpStatusReturningLogoutSuccessHandler(HttpStatus.OK));
}

```

```

@Bean
public DaoAuthenticationProvider daoAuthenticationProvider(){
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    provider.setPasswordEncoder(passwordEncoder);
    provider.setUserDetailsService(appUserService);
    return provider;
}
}

```

The antMatchers match any regex provided with an url provided to the server, and are configured as authenticated or permitAll(). We use form-login for the same, ie it accepts a **application/x-www-form-urlencoded** post request to the endpoint "api/login" defined in the config. The logout happens at the url "api/logout" and clears all cookies and authentication. The config is made for a ReactJs based frontend which is a single page web-app or serves a single index.html page from the static content or the resources folder of the application. Let's have a look at our application.properties file.

```

server.error.include-message=always
server.error.include-binding-errors=always

```

```

spring.datasource.url=jdbc:postgresql://db:5432/flamup
spring.datasource.username=postgres
spring.datasource.password=postgres

```

```

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format-sql=true
spring.jpa.show-sql=true

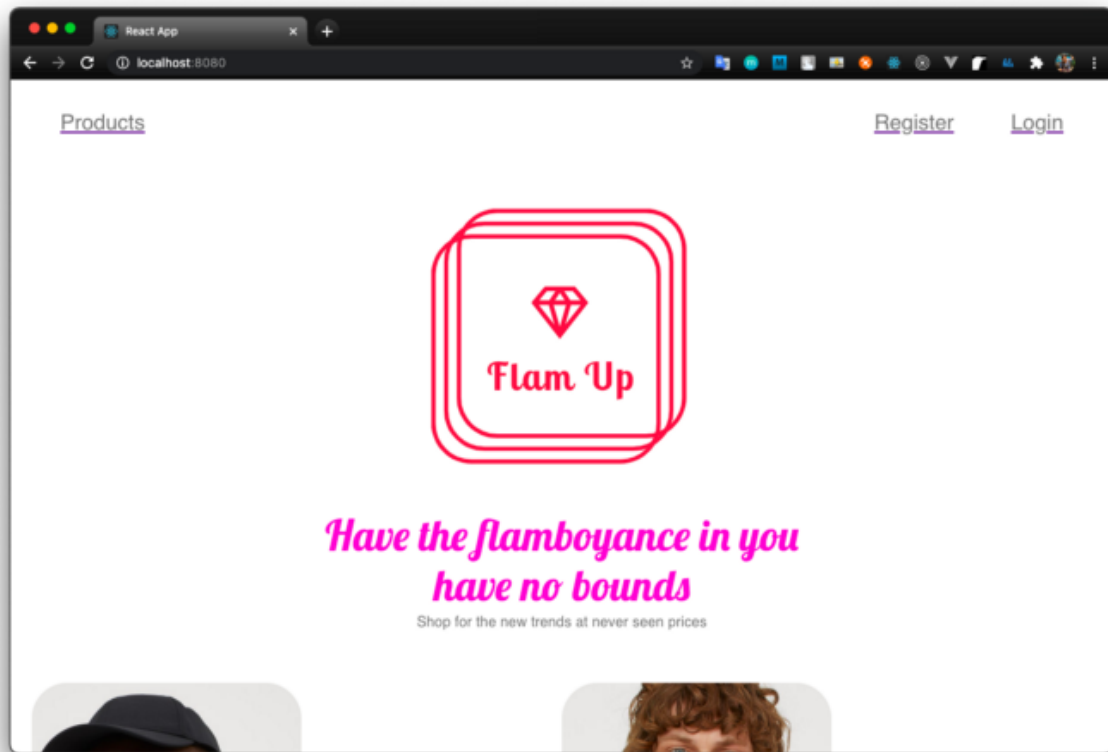
```

```

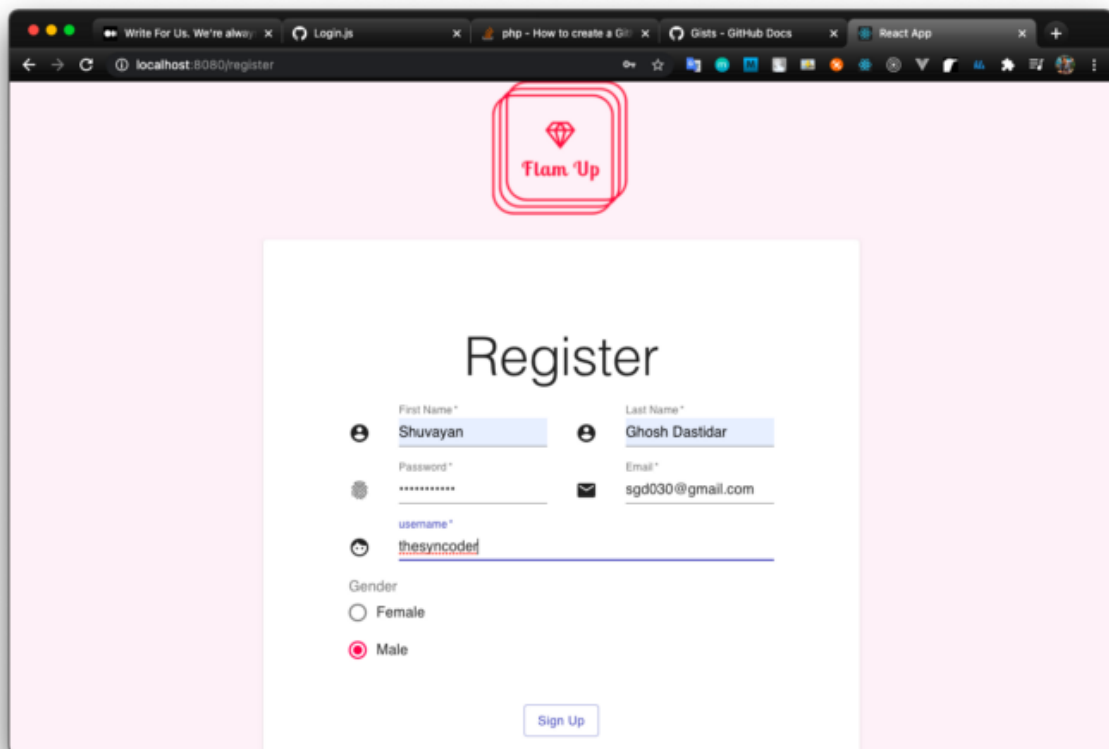
spring.session.jdbc.initialize-schema=always
spring.session.jdbc.table-name=SPRING_SESSION
spring.session.store-type=jdbc

```

The frontend has been made with ReactJs. The frontend has the tasks of pagination, maintaining authentication and pagination for receiving the products from the backend and displaying them in a paginated fashion.

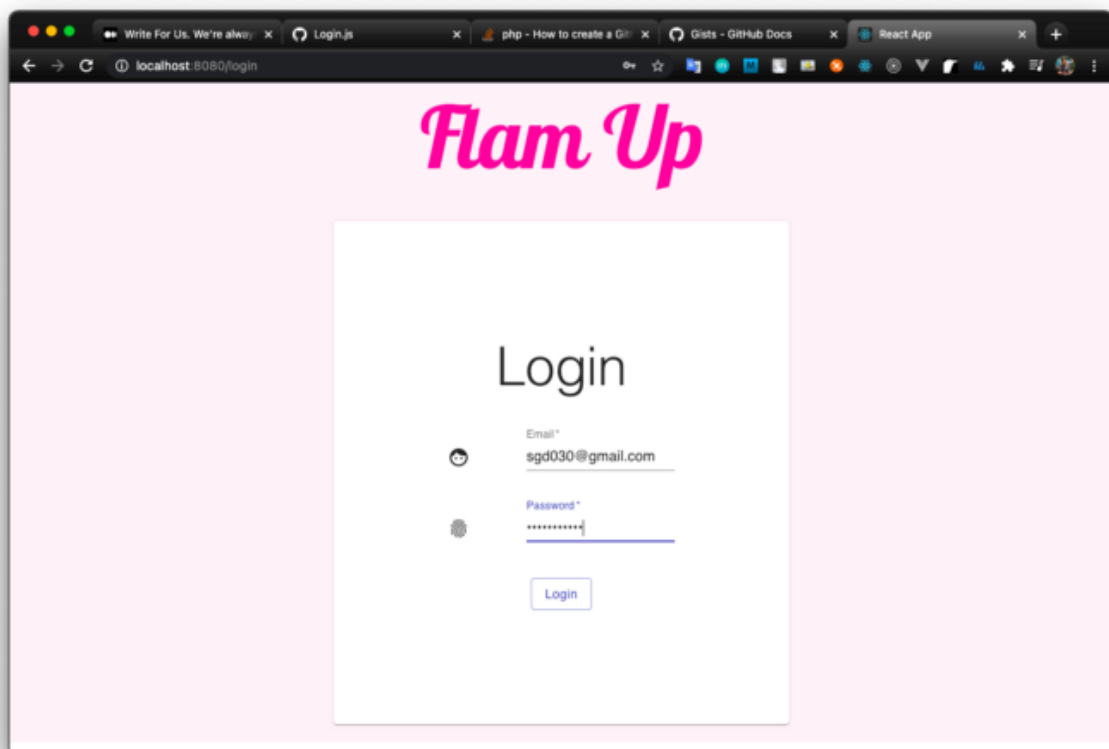


Home Page



The screenshot shows a web browser window with the address bar displaying `localhost:8080/register`. The page has a light pink background. At the top center, there is a logo consisting of a red diamond icon inside a red square frame, with the text "Flam Up" below it. The main content is a white rectangular form titled "Register". Inside the form, there are input fields for "First Name" (containing "Shuvayan"), "Last Name" (containing "Ghosh Dastidar"), "Password" (masked with dots), "Email" (containing "sgd030@gmail.com"), and "username" (containing "thesyncode"). Below these fields, there are radio buttons for "Gender" with "Female" and "Male" options; the "Male" option is selected. A "Sign Up" button is located at the bottom right of the form.

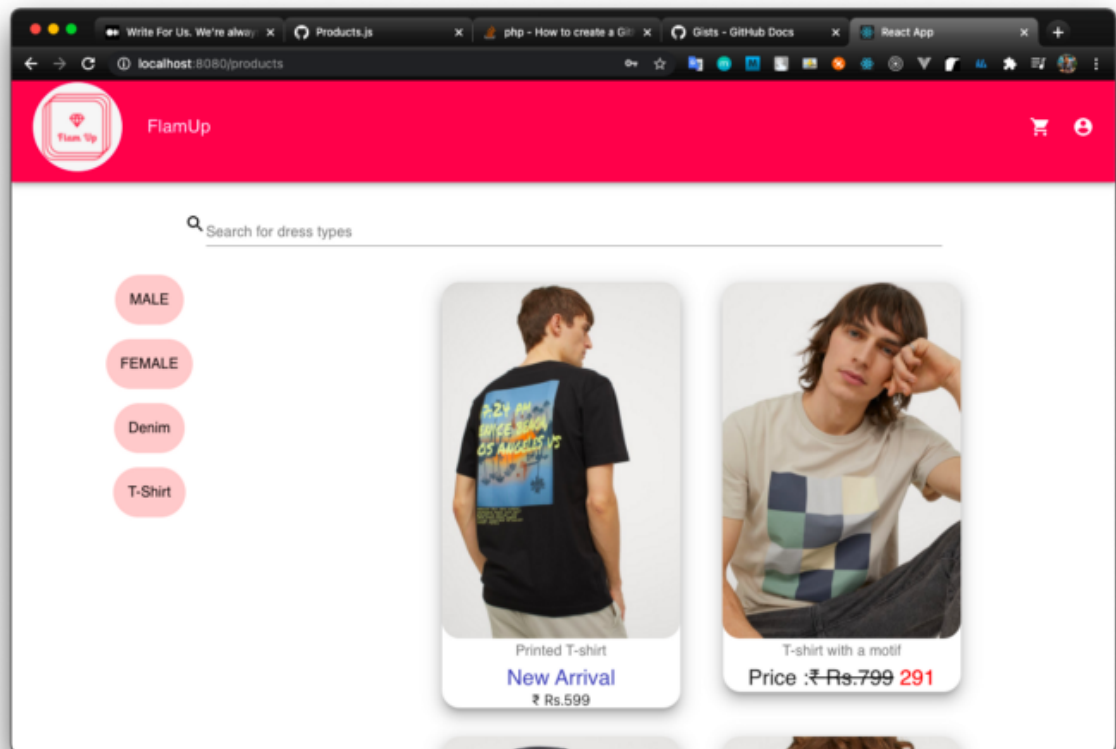
Register page



The screenshot shows a web browser window with the address bar displaying `localhost:8080/login`. The page has a light pink background. At the top center, there is a logo with the text "Flam Up" in a pink, cursive font. The main content is a white rectangular form titled "Login". Inside the form, there are input fields for "Email" (containing "sgd030@gmail.com") and "Password" (masked with dots). A "Login" button is located at the bottom right of the form.

Login Page





Products Page showing Pagination and support for different search items.

## Key Points

- Proper authentication handling in both frontend and backend.
- Session Handling for showing products according to user preference during the current session.
- Support for pagination in both frontend and backend to reduce server loads.