Customer Management Analysis

Uploaded a file

i done a project OnlineFoodDeliverySystem ,i provide my project pdf , analyse this and i managed customer management

Your project "Online Food Delivery System" includes a comprehensive customer management module that handles user registration, authentication, and profile management. You managed the following key aspects:

1. **Customer Registration and Authentication:** Users can register and log in through the frontend, and the backend handles authentication and profile management.
2. **Profile Management:** Customers can view and update their profile details, including name, email, phone, and address.
3. **Data Handling:** Customer data is stored and processed through the backend, utilizing relational databases such as MySQL, PostgreSQL, or SQL Server.
4. **Frontend and Backend Interaction:** The frontend sends customer data to the backend via REST APIs.

Let me know if you need help refining any part of your project or preparing documentation.

i provide my all codes

Please upload your code files, and I'll analyze them to give you insights and suggestions based on your implementation.

Controller:

```
package com.cts.OnlineFoodDeliverySystem.controller;

import java.time.format.DateTimeFormatter;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import
org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.ModelAttribute;
import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.PostMapping;

import
com.cts.OnlineFoodDeliverySystem.model.Customer;
import
com.cts.OnlineFoodDeliverySystem.model.MenuItems;
import com.cts.OnlineFoodDeliverySystem.model.Order;
import
com.cts.OnlineFoodDeliverySystem.model.RestaurantAdmi
n;
import
com.cts.OnlineFoodDeliverySystem.service.CustomerServic
e;
import
com.cts.OnlineFoodDeliverySystem.service.MenuItemsServi
ce;
import
com.cts.OnlineFoodDeliverySystem.service.OrderService;
import
com.cts.OnlineFoodDeliverySystem.service.RestaurantAdmi
nService;

@Controller
public class CustomerController {

   @Autowired
     private CustomerService customerService;
   @Autowired
     private RestaurantAdminService
restaurantAdminService;
   @Autowired
     private MenuItemsService menuItemService;
   @Autowired
     private OrderService orderService;
```

```
    @GetMapping("/customer/register")
    public String showCustomerRegistrationForm(Model model) {
        model.addAttribute("customer", new Customer());
        return "customer/register";
    }

    @PostMapping("/customer/register")
    public String registerCustomer(@ModelAttribute("customer") Customer customer, Model model) {
        if (!customer.getEmail().contains("@") || !customer.getEmail().contains(".")) {
            model.addAttribute("error", "Invalid email format");
            return "customer/register";
        }
        customerService.registerCustomer(customer);
        model.addAttribute("success", "Registration successful! You can now log in.");
        return "customer/login"; // Or changed here
    }

    @GetMapping("/customer/login")
    public String showCustomerLoginForm() {
        return "customer/login";
    }

    @GetMapping("/customer/dashboard")
    public String customerDashboard(Model model) {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        String currentCustomerEmail = authentication.getName();
        Customer customer = customerService.getCustomerByEmail(currentCustomerEmail);
        List<Order> recentOrders = orderService.getRecentOrdersByCustomer(customer);

        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
        List<String> formattedOrderDates =
recentOrders.stream()
          .map(order ->
order.getOrderDate().format(formatter))
          .toList();

        model.addAttribute("customer", customer);
        model.addAttribute("recentOrders", recentOrders);
        model.addAttribute("formattedOrderDates",
formattedOrderDates); // Add the formatted dates

        return "customer/dashboard";
    }

    @GetMapping("/customer/dashboard/Restaurants")
    public String DisplayRestaurants(Model model) {
        List<RestaurantAdmin>
restaurants=restaurantAdminService.allRestaurant();
        model.addAttribute("restaurants",restaurants);
        return "customer/restaurants";
    }


@GetMapping("/customer/dashboard/Restaurants/{email}/
view")
    public String
DisplayItemsInRestaurant(@PathVariable("email") String
email,Model model) {
        RestaurantAdmin
radmin=restaurantAdminService.findAdminByEmail(email).
get();
        List<MenuItems>
mitems=menuItemService.getMenuItemsByRestaurantId(ra
dmin.getId());
        model.addAttribute("rest",radmin);
        model.addAttribute("items",mitems);
        return "customer/displayItems";
    }
    @GetMapping("/customer/profile")
    public String viewProfile(Model model) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String email = authentication.getName();
        Customer customer =
customerService.getCustomerByEmail(email);
        model.addAttribute("customer", customer);
        return "customer/profile";
    }

    @GetMapping("/customer/edit-profile")
    public String editProfileForm(Model model) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String email = authentication.getName();
        Customer customer =
customerService.getCustomerByEmail(email);
        model.addAttribute("customer", customer);
        return "customer/edit-profile";
    }
    @PostMapping("/customer/edit-profile")
    public String saveProfile(@ModelAttribute Customer
customer) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String email = authentication.getName();
        Customer existingCustomer =
customerService.getCustomerByEmail(email);

        // Ensure the ID of the existing customer is set in the
updated customer object

customer.setCustomerid(existingCustomer.getCustomerid())
;

        // Preserve the existing password

customer.setPassword(existingCustomer.getPassword());

        customerService.updateCustomer(customer);
        return "redirect:/customer/profile?success=Profile
updated successfully!";
    }
```

```
@GetMapping("/customer/dashboard/Restaurants/{email}/
view/display/{id}")
        public String
DisplayItemsInRestaurant(@PathVariable("email") String
email,@PathVariable("id") int id,Model model) {
            RestaurantAdmin
radmin=restaurantAdminService.findAdminByEmail(email).
get();
            List<MenuItems>
mitems=menuItemService.getMenuItemsByRestaurantId(ra
dmin.getId());
            model.addAttribute("rest",radmin);
            model.addAttribute("items",id);
            return "customer/itemcart";
        }

}
```

Model:

```
package com.cts.OnlineFoodDeliverySystem.model;
import org.springframework.stereotype.Component;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.NoArgsConstructor;
@Component
@Entity
@Table(name = "customer")
@NoArgsConstructor
public class Customer {
    @Id
    @Column(name="customerid",length=10)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int customerid;

    @Column(name="cname",length=50,nullable = false)
    private String name;

    @Column(name="email",length=50,unique = true,
nullable = false)
    private String email;

    @Column(name = "password", length = 100, nullable =
false)
    private String password;


    @Column(name="phone",length=10)
    private long phone;

    @Column(name="address",length=255)
    private String address;

    public int getCustomerid() {
        return customerid;
    }

    public void setCustomerid(int customerid) {
        this.customerid = customerid;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

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

    public String getPassword() {
        return password;
    }

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

    public long getPhone() {
        return phone;
    }

    public void setPhone(long phone) {
        this.phone = phone;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
    public Customer() {

    }
    public Customer(String name, String email, String password, long phone, String address) {
        super();
        this.name = name;
        this.email = email;
        this.password = password;
        this.phone = phone;
        this.address = address;
    }

}


Repository:
package com.cts.OnlineFoodDeliverySystem.repository;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.cts.OnlineFoodDeliverySystem.model.Customer;
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer>{
    Optional<Customer> findByEmail(String email);
}
```

Security:

```
package com.cts.OnlineFoodDeliverySystem.security;

import java.util.Collections;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEn
```

```
coder;
import
org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.util.matcher.AntPathReq
uestMatcher;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private CustomUserDetailsService
customerUserDetailsService;

    @Lazy
    @Autowired
    @Qualifier("restaurantAdminDetailsService")
    private UserDetailsService
restaurantAdminUserDetailsService;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public DaoAuthenticationProvider
customerAuthProvider() {
        DaoAuthenticationProvider authProvider = new
DaoAuthenticationProvider();

authProvider.setUserDetailsService(customerUserDetailsSer
vice);
        authProvider.setPasswordEncoder(passwordEncoder());
        authProvider.setAuthoritiesMapper(user ->
Collections.singletonList(() -> "CUSTOMER")); // Assign
CUSTOMER role
        return authProvider;
    }

    @Bean
    public DaoAuthenticationProvider adminAuthProvider() {
        DaoAuthenticationProvider authProvider = new
DaoAuthenticationProvider();

authProvider.setUserDetailsService(restaurantAdminUserD
etailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        authProvider.setAuthoritiesMapper(user ->
Collections.singletonList(() -> "ADMIN")); // Assign ADMIN
role
        return authProvider;
    }

    @Bean
    public SecurityFilterChain
customerFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/customer/**", "/",
"/cart/**","/order/**") // Include /cart/** in this filter chain
            .authorizeHttpRequests((requests) -> requests
                .requestMatchers("/", "/images/**", "/webjars/**",
"/css/**", "/js/**", "/customer/register").permitAll()
                .requestMatchers("/customer/**",
"/cart/**","/order/**").hasAuthority("CUSTOMER") // Secure
/customer/** and /cart/** for CUSTOMER role
                .anyRequest().authenticated()
            )
            .formLogin((form) -> form
                .loginPage("/customer/login")
                .loginProcessingUrl("/customer/login")
                .defaultSuccessUrl("/customer/dashboard")
                .usernameParameter("email")
                .passwordParameter("password")
                .failureUrl("/customer/login?error")
                .permitAll()
            )
            .logout((logout) -> logout
                .logoutRequestMatcher(new
AntPathRequestMatcher("/logout"))
                .logoutSuccessUrl("/")
                .permitAll()
                .deleteCookies("JSESSIONID")
            )
            .authenticationProvider(customerAuthProvider());
        return http.build();
    }

    @Bean
    public SecurityFilterChain adminFilterChain(HttpSecurity
http) throws Exception {
        http
            .securityMatcher("/admin/**")
            .authorizeHttpRequests((requests) -> requests
                .requestMatchers("/admin/register", "/images/**",
"/webjars/**", "/css/**", "/js/**").permitAll() // Add any
public admin resources

.requestMatchers("/admin/**").hasAuthority("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin((form) -> form
                .loginPage("/admin/login")
                .loginProcessingUrl("/admin/login")
                .defaultSuccessUrl("/admin/dashboard")
                .usernameParameter("email")
                .passwordParameter("password")
                .failureUrl("/admin/login?error")
                .permitAll()
            )
            .logout((logout) -> logout
                .logoutRequestMatcher(new
AntPathRequestMatcher("/admin/logout")) // Optional:
specific admin logout
                .logoutSuccessUrl("/admin/login?logout")
                .permitAll()
                .deleteCookies("JSESSIONID")
            )
            .authenticationProvider(adminAuthProvider());
        return http.build();
    }
}

Service:
```

```
package com.cts.OnlineFoodDeliverySystem.service;

import java.util.Optional;

import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Lazy;
import
org.springframework.security.core.userdetails.UsernameN
otFoundException;
import
org.springframework.security.crypto.password.PasswordEn
coder;
import org.springframework.stereotype.Service;

import
com.cts.OnlineFoodDeliverySystem.model.Customer;
import
com.cts.OnlineFoodDeliverySystem.repository.CustomerRe
pository;

@Service
public class CustomerServiceImpl implements
CustomerService{

    @Autowired
    private CustomerRepository customerRepository;
    @Lazy
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Override
    public void registerCustomer(Customer customer) {
        // TODO Auto-generated method stub

customer.setPassword(passwordEncoder.encode(customer.
getPassword())); // Encode the password
        customerRepository.save(customer);

    }

    @Override
    public Optional<Customer> findCustomerByEmail(String
email) {
        // TODO Auto-generated method stub

        return customerRepository.findByEmail(email);
    }

    @Override
    public Customer getCustomerByEmail(String email) {
        // TODO Auto-generated method stub
        return customerRepository.findByEmail(email)
                .orElseThrow(() -> new
UsernameNotFoundException("Customer not found with
email: " + email));
    }

    @Override
    public void updateCustomer(Customer customer) {
        // TODO Auto-generated method stub
        customerRepository.save(customer);
    }

}
```

Analyse all these and wait for my new propmpt

Got it! I'll analyze your code thoroughly. Let me know when you're ready with the next prompt.

these are all my module codes , so tomorrow i have interview and i wanted to know about every annotation and each line in codes, and give me the flow of of my module with diagram

I'll start by explaining the annotations and code lines, and then I'll create a flow diagram to illustrate the module interactions. Let's begin with the annotations and code breakdown.

1. Annotations Breakdown:

Controller Annotations:

- `@Controller`: Marks the class as a Spring MVC controller. It is responsible for processing HTTP requests and returning views.

- `@Autowired`: Enables automatic dependency injection. Spring injects the required bean at runtime.
- `@GetMapping` / `@PostMapping`: Map HTTP GET/POST requests to specific handler methods.
- `@ModelAttribute`: Binds form data to a model object.
- `@PathVariable`: Extracts data from the URI.

Model Annotations:

- `@Component`: Marks the class as a Spring bean.
- `@Entity`: Specifies that the class is a JPA entity (mapped to a database table).
- `@Table(name = "customer")`: Specifies the database table name.
- `@Id`: Marks the primary key.
- `@Column`: Maps a field to a database column. Options include length, nullable, unique, etc.
- `@GeneratedValue(strategy = GenerationType.AUTO)`: Specifies auto-incremented primary key.

Repository Annotations:

- `@Repository`: Indicates that the interface is a DAO (Data Access Object) component that interacts with the database.
- `@JpaRepository`: Provides JPA functionalities such as CRUD operations.

Security Annotations:

- `@Configuration`: Marks the class as a source of bean definitions.
- `@EnableWebSecurity`: Enables Spring Security for the application.
- `@Bean`: Indicates that a method returns a bean that will be managed by the Spring container.
- `@Qualifier`: Used when there are multiple beans of the same type to specify which one should be injected.
- `@Lazy`: Delays the initialization of a bean until it is needed.

Service Annotations:

- `@Service`: Marks the class as a Spring service, which contains business logic.

---

2. Code Flow:

1. **Registration:**
   - Customer submits the registration form.
   - The controller validates the email and saves the customer after encoding the password.
   - Success or error message is displayed.
2. **Login:**
   - The customer logs in using email and password.
   - The system verifies credentials using the `UserDetailsService`.
3. **Dashboard:**
   - After successful login, the customer dashboard displays recent orders.
   - Uses `SecurityContextHolder` to get the authenticated user.
4. **Restaurant Viewing:**
   - The customer can view the list of available restaurants.
   - Upon selecting a restaurant, the menu items are displayed.
5. **Profile Management:**
   - The customer can view and edit their profile.
   - Changes are saved and updated in the database.

I'll now create a flow diagram for your module.

explain with code , like first start with controller and explain everything in that code

Let's break down the **CustomerController** class from your project, step by step.

---

1. Class Definition and Annotations:

```java
@Controller
public class CustomerController {
```

- `@Controller`: Marks this class as a Spring MVC controller to handle web requests.

---

2. Dependency Injection:

```java
@Autowired
private CustomerService customerService;

@Autowired
private RestaurantAdminService restaurantAdminService;

@Autowired
private MenuItemsService menuItemService;

@Autowired
private OrderService orderService;
```

- `@Autowired`: Automatically injects instances of the specified services.
- **Services Used:**
  - `CustomerService`: Handles customer-related operations.
  - `RestaurantAdminService`: Manages restaurant admin-related operations.
  - `MenuItemsService`: Fetches menu items.
  - `OrderService`: Handles order-related tasks.

---

3. Customer Registration:

```java
```

```java
@GetMapping("/customer/register")
public String showCustomerRegistrationForm(Model model) {
    model.addAttribute("customer", new Customer());
    return "customer/register";
}
```

- `@GetMapping("/customer/register")`: Handles GET requests for customer registration.
- Adds a new `Customer` object to the model to be used in the registration form.
- Returns the view name `"customer/register"`.

```java
@PostMapping("/customer/register")
public String registerCustomer(@ModelAttribute("customer") Customer customer, Model
model) {
    if (!customer.getEmail().contains("@") || !customer.getEmail().contains(".")) {
        model.addAttribute("error", "Invalid email format");
        return "customer/register";
    }
    customerService.registerCustomer(customer);
    model.addAttribute("success", "Registration successful! You can now log in.");
    return "customer/login";
}
```

- `@PostMapping("/customer/register")`: Handles POST requests to register a customer.
- Uses `@ModelAttribute` to bind form data to the `Customer` object.
- Validates the email format.
- Calls `customerService.registerCustomer(customer)` to save the customer.
- On success, redirects to the login page.

---

4. Customer Login:

```java
```

```
@GetMapping("/customer/login")
public String showCustomerLoginForm() {
    return "customer/login";
}
```

- Renders the customer login page.

---

5. Customer Dashboard:

```java
@GetMapping("/customer/dashboard")
public String customerDashboard(Model model) {
    Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
    String currentCustomerEmail = authentication.getName();
    Customer customer = customerService.getCustomerByEmail(currentCustomerEmail);
    List<Order> recentOrders = orderService.getRecentOrdersByCustomer(customer);

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
    List<String> formattedOrderDates = recentOrders.stream()
            .map(order -> order.getOrderDate().format(formatter))
            .toList();

    model.addAttribute("customer", customer);
    model.addAttribute("recentOrders", recentOrders);
    model.addAttribute("formattedOrderDates", formattedOrderDates);
    return "customer/dashboard";
}
```

- **Retrieving the Current User:**
    - Uses `SecurityContextHolder` to get the authenticated user's email.
- **Fetching Customer Data:**
    - Uses `customerService.getCustomerByEmail()` to get customer details.
- **Fetching Recent Orders:**
    - Calls `orderService.getRecentOrdersByCustomer()` to get recent orders.

  - Formats the order dates for display.
- **Model Attributes:**
  - `customer`: Holds customer details.
  - `recentOrders`: Holds the list of recent orders.
  - `formattedOrderDates`: Holds formatted order dates.
- **Return Value:**
  - Returns the `"customer/dashboard"` view.

---

6. Viewing Restaurants:

```java
@GetMapping("/customer/dashboard/Restaurants")
public String DisplayRestaurants(Model model) {
    List<RestaurantAdmin> restaurants = restaurantAdminService.allRestaurant();
    model.addAttribute("restaurants", restaurants);
    return "customer/restaurants";
}
```

- Calls `restaurantAdminService.allRestaurant()` to fetch all restaurants.
- Adds the list of restaurants to the model.
- Renders the `"customer/restaurants"` view.

---

7. Viewing Menu Items from a Restaurant:

```java
@GetMapping("/customer/dashboard/Restaurants/{email}/view")
public String DisplayItemsInRestaurant(@PathVariable("email") String email, Model
model) {
    RestaurantAdmin radmin = restaurantAdminService.findAdminByEmail(email).get();
    List<MenuItems> mitems =
menuItemService.getMenuItemsByRestaurantId(radmin.getId());
```

```
    model.addAttribute("rest", radmin);
    model.addAttribute("items", mitems);
    return "customer/displayItems";
}
```

- Uses `@PathVariable` to extract the restaurant's email from the URL.
- Fetches restaurant admin details and menu items.
- Adds the restaurant and menu items to the model.
- Renders the `"customer/displayItems"` view.

---

8. Customer Profile:

```java
@GetMapping("/customer/profile")
public String viewProfile(Model model) {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    String email = authentication.getName();
    Customer customer = customerService.getCustomerByEmail(email);
    model.addAttribute("customer", customer);
    return "customer/profile";
}
```

- Retrieves the logged-in customer's profile.
- Adds the customer data to the model.
- Returns the `"customer/profile"` view.

---

9. Editing Profile:

```java
@PostMapping("/customer/edit-profile")
public String saveProfile(@ModelAttribute Customer customer) {
    Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
    String email = authentication.getName();
    Customer existingCustomer = customerService.getCustomerByEmail(email);
    customer.setCustomerid(existingCustomer.getCustomerid());
    customer.setPassword(existingCustomer.getPassword());
    customerService.updateCustomer(customer);
    return "redirect:/customer/profile?success=Profile updated successfully!";
}
```

- Retrieves the current customer's details.
- Updates the profile while preserving the existing password.
- Saves the updated profile using `customerService.updateCustomer()`.
- Redirects to the profile page with a success message.

---

10. Displaying Items in Restaurant (Alternative):

```java
@GetMapping("/customer/dashboard/Restaurants/{email}/view/display/{id}")
public String DisplayItemsInRestaurant(@PathVariable("email") String email,
@PathVariable("id") int id, Model model) {
    RestaurantAdmin radmin = restaurantAdminService.findAdminByEmail(email).get();
    List<MenuItems> mitems =
menuItemService.getMenuItemsByRestaurantId(radmin.getId());
    model.addAttribute("rest", radmin);
    model.addAttribute("items", id);
    return "customer/itemcart";
}
```

- Displays items in a specific restaurant based on the restaurant ID.
- Passes both the restaurant and item ID to the model.
- Renders the `"customer/itemcart"` view.

Summary:

1. **Registration and Login:** Handles user registration and login using secure methods.
2. **Dashboard:** Shows recent orders and customer info.
3. **View Restaurants and Menu:** Fetches and displays restaurant details and menu items.
4. **Profile Management:** Allows the customer to view and update profile information.
5. **Data Handling:** Uses various services to interact with the database.

Let me know if you need more detailed explanations of any part!

like this explain for other codes

Let's continue with the breakdown of your other code files, following the same structure as before.

---

2. Customer Model Class:

Class Definition and Annotations:

```java
@Component
@Entity
@Table(name = "customer")
@NoArgsConstructor
public class Customer {
```

- `@Component`: Marks this class as a Spring bean for component scanning.
- `@Entity`: Specifies that this class represents a JPA entity.
- `@Table(name = "customer")`: Maps the class to the `customer` table in the database.
- `@NoArgsConstructor`: Lombok annotation to generate a no-argument constructor.

Fields and Annotations:

java

```java
@Id
@Column(name="customerid", length=10)
@GeneratedValue(strategy = GenerationType.AUTO)
private int customerid;
```

- `@Id`: Marks this field as the primary key.
- `@Column`: Specifies the column name and length.
- `@GeneratedValue(strategy = GenerationType.AUTO)`: Automatically generates the ID.

Other Fields:

java

```java
@Column(name="cname", length=50, nullable = false)
private String name;

@Column(name="email", length=50, unique = true, nullable = false)
private String email;

@Column(name = "password", length = 100, nullable = false)
private String password;

@Column(name="phone", length=10)
private long phone;

@Column(name="address", length=255)
private String address;
```

- `nullable = false`: The field cannot be null in the database.
- `unique = true`: Ensures that the email value is unique.
- The length constraint defines the maximum number of characters.

Constructors:

java

```
public Customer(String name, String email, String password, long phone, String
address) {
    this.name = name;
    this.email = email;
    this.password = password;
    this.phone = phone;
    this.address = address;
}
```

- Constructor to initialize all fields.

Getter and Setter Methods:

- These methods provide controlled access to the private fields.

Summary:

- Represents the customer entity with fields mapped to database columns.
- Provides constructors and getter/setter methods.
- Uses annotations for database mapping and dependency injection.

---

3. Customer Repository:

Class Definition and Annotations:

```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
    Optional<Customer> findByEmail(String email);
}
```

- `@Repository`: Marks the interface as a data access layer.

- `JpaRepository<Customer, Integer>`:
  - Extends the `JpaRepository` to provide basic CRUD operations.
  - Uses `Customer` as the entity and `Integer` as the primary key type.

Custom Query Method:

```java
Optional<Customer> findByEmail(String email);
```

- Finds a customer by their email. Returns an `Optional<Customer>` to handle cases where the customer may not be found.

Summary:

- Acts as the data access layer for customer-related database operations.
- Utilizes JPA repository methods for standard CRUD and custom queries.

---

4. Customer Service Implementation:

Class Definition and Annotations:

```java
@Service
public class CustomerServiceImpl implements CustomerService {
```

- `@Service`: Marks this class as a service layer component in Spring.

Dependency Injection:

```java
```

```
@Autowired
private CustomerRepository customerRepository;

@Lazy
@Autowired
private PasswordEncoder passwordEncoder;
```

- Injects the customer repository and password encoder.
- `@Lazy`: Delays the injection until actually needed.

Registering a Customer:

java

```java
@Override
public void registerCustomer(Customer customer) {
    customer.setPassword(passwordEncoder.encode(customer.getPassword()));
    customerRepository.save(customer);
}
```

- Encodes the password before saving.
- Calls `save()` to store the customer in the database.

Finding a Customer by Email:

java

```java
@Override
public Optional<Customer> findCustomerByEmail(String email) {
    return customerRepository.findByEmail(email);
}
```

- Uses the custom query from the repository to find a customer by email.

Getting Customer Details:

```java
@Override
public Customer getCustomerByEmail(String email) {
    return customerRepository.findByEmail(email)
        .orElseThrow(() -> new UsernameNotFoundException("Customer not found with email: " + email));
}
```

- Uses `orElseThrow()` to handle cases where the customer is not found.

Updating a Customer:

```java
@Override
public void updateCustomer(Customer customer) {
    customerRepository.save(customer);
}
```

- Updates the customer details by saving the new data.

Summary:

- Provides business logic for customer-related operations.
- Uses password encoding for secure storage.
- Efficiently fetches and updates customer data.

---

5. Security Configuration:

Class Definition and Annotations:

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

- `@Configuration`: Marks the class as a configuration class.
- `@EnableWebSecurity`: Enables Spring Security.

Password Encoding:

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- Uses BCrypt for hashing passwords.

Authentication Providers:

```java
@Bean
public DaoAuthenticationProvider customerAuthProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(customerUserDetailsService);
    authProvider.setPasswordEncoder(passwordEncoder());
    return authProvider;
}
```

- Configures the authentication provider for customers using the custom user details service.

Security Filter Chain:

```java
@Bean
public SecurityFilterChain customerFilterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/customer/**")
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/", "/customer/register").permitAll()
            .requestMatchers("/customer/**").hasAuthority("CUSTOMER")
            .anyRequest().authenticated()
        )
        .formLogin(form -> form
            .loginPage("/customer/login")
            .defaultSuccessUrl("/customer/dashboard")
            .permitAll()
        )
        .logout(logout -> logout
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
            .logoutSuccessUrl("/")
        );
    return http.build();
}
```

- Configures authentication for customer URLs.
- Allows unrestricted access to registration and login pages.
- Requires the "CUSTOMER" role for other URLs.

Summary:

- Configures security for customer-related URLs.

- Uses BCrypt for password hashing.
- Separates customer and admin authentication configurations.

Flow Summary:

1. **Registration:**
    - Customer fills the form → Controller processes → Service encodes password → Data saved.
2. **Login:**
    - User credentials verified using Spring Security → Redirect to dashboard.
3. **Profile Management:**
    - Fetch customer data → Display on profile page → Update details if changed.
4. **Restaurant Viewing:**
    - Display all available restaurants → Select a restaurant → View menu items.
5. **Security:**
    - Uses role-based access control to distinguish between customer and admin users.

The next step is to create the flow diagram to visualize the interactions. Let me know if you want a breakdown of any other module or further explanations.

give me overview of each code

Here’s an overview of each code module in your **Online Food Delivery System** project:

---

1. Customer Controller:

- **Purpose:** Handles HTTP requests related to customer operations such as registration, login, viewing profile, and interacting with restaurants.
- **Key Functions:**

  - **Registration:** Handles GET and POST requests for customer sign-up.
  - **Login:** Renders the login page.
  - **Dashboard:** Displays recent orders and customer information.
  - **Profile Management:** Allows viewing and editing of the customer profile.
  - **Restaurant Viewing:** Displays available restaurants and their menu items.
- **Flow:** Frontend → Controller → Service → Repository → Database → View.
- **Security:** Uses Spring Security for authentication and authorization.

---

2. Customer Model:

- **Purpose:** Represents the customer entity and maps to the database table.
- **Key Annotations:**
  - `@Entity` and `@Table`: Maps the class to a database table named `customer`.
  - `@Id`: Marks the primary key.
  - `@GeneratedValue`: Automatically generates the customer ID.
- **Attributes:** Customer ID, Name, Email, Password, Phone, Address.
- **Flow:** Acts as a data model between the service layer and database.

---

3. Customer Repository:

- **Purpose:** Handles data access for customer-related operations.
- **Extends:** `JpaRepository<Customer, Integer>` to leverage CRUD operations.
- **Custom Query:**
  - `findByEmail(String email)`: Finds a customer by their email.
- **Flow:** Service → Repository → Database.

---

4. Customer Service Implementation:

- **Purpose:** Implements business logic related to customer operations.
- **Methods:**
  - **registerCustomer:** Saves a new customer with an encoded password.
  - **findCustomerByEmail:** Retrieves a customer by email.
  - **getCustomerByEmail:** Retrieves customer data and throws an exception if not found.
  - **updateCustomer:** Updates customer profile information.
- **Flow:** Controller → Service → Repository → Database.
- **Security:** Encodes passwords before saving.

---

5. Security Configuration:

- **Purpose:** Manages security settings for customer and admin access.
- **Components:**
  - **Password Encoder:** Uses BCrypt for hashing passwords.
  - **Authentication Providers:** Separate configurations for customers and admins.
  - **Security Filter Chain:**
    - Allows unrestricted access to registration and login pages.
    - Restricts customer-specific pages to users with the **"CUSTOMER"** role.
    - Redirects after successful login.
- **Flow:** Handles HTTP security and role-based access control.

---

6. Customer Service Interface (Not Provided but Implied):

- **Purpose:** Defines methods for customer-related operations.
- **Common Methods:**
  - Register a customer.
  - Find customer by email.

    - Update customer profile.
- **Flow:** Interface → Implementation → Repository.

---

7. Custom User Details Service (Implied in Security):

- **Purpose:** Loads customer details during login for authentication.
- **Flow:** Spring Security → UserDetailsService → Customer Repository.

Flow Summary:

1. **Registration:**
    - Customer fills registration form → Controller → Service (password encoding) → Repository → Database.
2. **Login:**
    - Customer logs in → Spring Security authenticates using UserDetailsService → Redirect to dashboard.
3. **Profile Update:**
    - Customer updates profile → Controller → Service → Repository → Database.
4. **Restaurant Browsing:**
    - Fetch restaurant data → Display on the dashboard.
5. **Security:**
    - Role-based access control → Different filter chains for customer and admin.

This overview covers the primary functionality and flow of each module in your project. Let me know if you need more insights or any specific section explained!