**Asish Kumar Panda 2006315 I.T**

Backend Engineer Code Assessment Documentation

Overview:

This document details the Backend Engineer Code Assessment, designed to evaluate your skills in integrating Stripe payment processing into a pre-existing Spring Boot application leveraging Temporal Workflow Engine.

Project Setup:

* Base Application: Spring Boot application with Java 21.
* External Service: Temporal Workflow Engine for orchestrating business logic.
* Task Focus: Stripe integration for customer creation and data management.
* Repository: <https://github.com/Midas-Labs/backend-engineer-assessment>

Tasks:

1. Stripe Integration for Customer Creation using Temporal Workflow:
   * Implement a workflow using Temporal to create a new Stripe customer upon user signup using the provided Stripe Create Customer API.
   * Bootstrap code and SDK are provided.
2. Add New Fields to User Model:
   * Include a providerType field with an enum for "stripe".
   * Add a providerId field to store the Stripe customer ID.
   * Update the application controller to handle these fields during user signup and store the providerId appropriately.
3. API Implementation:
   * Utilize the existing GET /accounts endpoint for verification and testing.

Bonus:

* Write unit and integration tests for your implementation, covering Stripe integration, user model changes, signup process, and GET /accounts functionality.

Project Submission Guidelines:

* Code Quality: Clean, well-documented, adhering to standard practices.
* Testing: Include tests for all new functionalities. Bonus points for comprehensive testing.
* Documentation: Provide a README file with:
  + Setup instructions.
  + Test execution instructions.
  + Brief explanation of your implementation approach and assumptions.

Evaluation Criteria:

* Functionality: Meets all requirements and works as expected.
* Code Quality: Well-organized, clean, readable, and utilizes good practices.
* Testing: Covers critical paths and edge cases, ensuring application stability.
* Documentation: Clear setup instructions and development insights.

Additional Notes:

* Feel free to use any libraries or frameworks that enhance your solution.
* This assessment is designed to be completed within a specified timeframe.
* Be prepared to discuss your approach and answer questions during the evaluation.

AccountController code:

package com.midas.app.controllers;  
  
import com.midas.app.mappers.Mapper;  
import com.midas.app.models.Account;  
import com.midas.app.models.User.ProviderType;  
import com.midas.app.services.AccountService;  
import com.midas.generated.api.AccountsApi;  
import com.midas.generated.model.AccountDto;  
import com.midas.generated.model.CreateAccountDto;  
import io.temporal.client.WorkflowClient;  
import io.temporal.client.WorkflowOptions;  
import java.util.List;  
import lombok.RequiredArgsConstructor;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.\*;  
  
@Controller  
@RequestMapping("/api/accounts")  
@RequiredArgsConstructor  
public class AccountController implements AccountsApi {  
  
 private final AccountService accountService;  
 private final WorkflowClient workflowClient;  
 private final Logger logger = LoggerFactory.*getLogger*(AccountController.class);  
  
 @Override  
 @PostMapping("/signup")  
 public ResponseEntity<AccountDto> createUserAccount(  
 @RequestBody CreateAccountDto createAccountDto) {  
 logger.info("Creating account for user with email: {}", createAccountDto.getEmail());  
  
 // Set providerType as STRIPE  
 createAccountDto.setProviderType(ProviderType.*STRIPE*);  
  
 WorkflowOptions options =  
 WorkflowOptions.*newBuilder*().setTaskQueue("user-signup-tasks").build();  
 UserSignupWorkflow workflow = workflowClient.newWorkflowStub(UserSignupWorkflow.class, options);  
  
 // Execute the workflow for user signup  
 Account account = workflow.signupUser(createAccountDto);  
  
 return new ResponseEntity<>(Mapper.*toAccountDto*(account), HttpStatus.*CREATED*);  
 }  
  
 @Override  
 @GetMapping  
 public ResponseEntity<List<AccountDto>> getUserAccounts() {  
 logger.info("Retrieving all accounts");  
  
 List<Account> accounts = accountService.getAccounts();  
 List<AccountDto> accountsDto = accounts.stream().map(Mapper::*toAccountDto*).toList();  
  
 return new ResponseEntity<>(accountsDto, HttpStatus.*OK*);  
 }  
}

User/Accounts information:

package com.midas.app.models;  
  
import jakarta.persistence.\*;  
import java.time.OffsetDateTime;  
import java.util.UUID;  
import lombok.\*;  
import org.hibernate.annotations.CreationTimestamp;  
import org.hibernate.annotations.UpdateTimestamp;  
  
@Setter  
@Getter  
@RequiredArgsConstructor  
@AllArgsConstructor  
@Builder  
@Entity  
@Table(name = "accounts")  
public class Account {  
  
 @Id  
 @Column(name = "id")  
 @GeneratedValue  
 private UUID id;  
  
 @Column(name = "first\_name")  
 private String firstName;  
  
 @Column(name = "last\_name")  
 private String lastName;  
  
 @Column(name = "email")  
 private String email;  
  
 @Enumerated(EnumType.*STRING*)  
 @Column(name = "provider\_type")  
 private User.ProviderType providerType; // Added field  
  
 @Column(name = "provider\_id")  
 private String providerId; // Added field  
  
 @Column(name = "created\_at")  
 @CreationTimestamp  
 private OffsetDateTime createdAt;  
  
 @Column(name = "updated\_at")  
 @UpdateTimestamp  
 private OffsetDateTime updatedAt;  
}

1. AccountController:

* Role: Handles HTTP requests related to user accounts.
* Key Methods:
  + createUserAccount: Creates a new user account using a Temporal workflow.
  + getUserAccounts: Retrieves a list of existing accounts.

2. Account Model:

* Purpose: Represents a user account in the application.
* Key Fields:
  + id: Unique identifier for the account.
  + firstName, lastName, email: Personal information.
  + providerType: The type of payment provider (e.g., Stripe).
  + providerId: The ID associated with the payment provider.
  + createdAt, updatedAt: Timestamps for account creation and updates.

To delve deeper, please specify:

* Specific functionalities: Which parts of the code's behavior do you want to understand?
* Key areas of interest: Are you more interested in the controller logic, model structure, or other aspects?

Creating Accountdto

package com.midas.generated.model;  
  
import java.net.URI;  
import java.util.Objects;  
import com.fasterxml.jackson.annotation.JsonProperty;  
import com.fasterxml.jackson.annotation.JsonCreator;  
import com.fasterxml.jackson.annotation.JsonTypeName;  
import java.time.OffsetDateTime;  
import jakarta.validation.Valid;  
import jakarta.validation.constraints.\*;  
import io.swagger.v3.oas.annotations.media.Schema;  
  
  
import java.util.\*;  
import jakarta.annotation.Generated;  
  
*/\*\*  
 \* CreateAccountDto  
 \*/*@JsonTypeName("createAccount")  
@Generated(value = "org.openapitools.codegen.languages.SpringCodegen", date = "2024-02-06T00:41:20.524414800+05:30[Asia/Calcutta]")  
public class CreateAccountDto {  
  
 private String firstName;  
  
 private String lastName;  
  
 private String email;  
  
 public CreateAccountDto firstName(String firstName) {  
 this.firstName = firstName;  
 return this;  
 }  
  
 */\*\*  
 \* User's first name  
 \** ***@return*** *firstName  
 \*/* @Schema(name = "firstName", example = "John", description = "User's first name", requiredMode = Schema.RequiredMode.*NOT\_REQUIRED*)  
 @JsonProperty("firstName")  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public CreateAccountDto lastName(String lastName) {  
 this.lastName = lastName;  
 return this;  
 }  
  
 */\*\*  
 \* User's last name  
 \** ***@return*** *lastName  
 \*/* @Schema(name = "lastName", example = "Doe", description = "User's last name", requiredMode = Schema.RequiredMode.*NOT\_REQUIRED*)  
 @JsonProperty("lastName")  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public CreateAccountDto email(String email) {  
 this.email = email;  
 return this;  
 }  
  
 */\*\*  
 \* Email of user  
 \** ***@return*** *email  
 \*/* @Schema(name = "email", example = "john@doe.com", description = "Email of user", requiredMode = Schema.RequiredMode.*NOT\_REQUIRED*)  
 @JsonProperty("email")  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) {  
 return true;  
 }  
 if (o == null || getClass() != o.getClass()) {  
 return false;  
 }  
 CreateAccountDto createAccount = (CreateAccountDto) o;  
 return Objects.*equals*(this.firstName, createAccount.firstName) &&  
 Objects.*equals*(this.lastName, createAccount.lastName) &&  
 Objects.*equals*(this.email, createAccount.email);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(firstName, lastName, email);  
 }  
  
 @Override  
 public String toString() {  
 StringBuilder sb = new StringBuilder();  
 sb.append("class CreateAccountDto {\n");  
 sb.append(" firstName: ").append(toIndentedString(firstName)).append("\n");  
 sb.append(" lastName: ").append(toIndentedString(lastName)).append("\n");  
 sb.append(" email: ").append(toIndentedString(email)).append("\n");  
 sb.append("}");  
 return sb.toString();  
 }  
  
 */\*\*  
 \* Convert the given object to string with each line indented by 4 spaces  
 \* (except the first line).  
 \*/* private String toIndentedString(Object o) {  
 if (o == null) {  
 return "null";  
 }  
 return o.toString().replace("\n", "\n ");  
 }  
}

Purpose:

* This class serves as a data transfer object (DTO) to encapsulate information required for creating a new user account. It acts as a carrier of data between different parts of the application.

Key Fields:

* firstName: Stores the user's first name (String).
* lastName: Stores the user's last name (String).
* email: Stores the user's email address (String).

Annotations:

* @JsonTypeName("createAccount"): This annotation from Jackson library designates a logical type name for this DTO, useful for serialization and deserialization.
* @Generated: Indicates that the code was generated automatically, likely using a tool like Spring Codegen.
* @Schema: Annotations from OpenAPI 3 for defining API documentation elements, such as examples and descriptions.
* @JsonProperty: Specifies the names of properties for JSON serialization and deserialization.

Generated Methods:

* Getters and Setters: Standard methods for accessing and modifying the fields of the DTO.
* equals() and hashCode(): Methods for comparing objects and generating hash codes, used for operations like checking object equality and creating hash-based collections.
* toString(): Provides a string representation of the DTO, primarily for debugging and logging purposes.

Key Points:

* This DTO is likely used by controllers or services to receive account creation requests and transfer the data within the application.
* It doesn't contain any business logic or processing itself; it's solely a data structure for information exchange.