**I] Setup in Windows OS :-**

1. Installing java JDK 21.0.2 - [Link](https://www.azul.com/downloads/?package=jdk#zulu)
2. Downloading Temporal CLI for Windows:-

i) Download the CLI Binary Choose the appropriate binary for your system:

For Windows amd64: Download link

For Windows arm64: Download link

ii) Extract the Archive Use your preferred archive tool to extract the downloaded binary.

iii) Update System PATH Add the path to the temporal.exe binary to your system's PATH environment variable to access the CLI from any command prompt.  
  
iv) Verify Configuration:

Open a new Command Prompt window (or PowerShell) and type temporal to verify that the Temporal CLI is now accessible from any command prompt.

You should see the Temporal CLI help message indicating that the command is recognized.

1. Install Docker Desktop - [Link](https://docs.docker.com/desktop/install/windows-install/)

**II] Program Approach :-**

src/

├── main/

│ ├── java/

│ │ ├── com/

│ │ │ └── midas/

│ │ │ ├── app/

│ │ │ │ ├── activities/

|\_\_\_ UserActivity.java (if

│ │ │ │ ├── controllers/

│ │ │ │ │ └── ApplicationController.java

│ │ │ │ ├── models/

│ │ │ │ │ └── UserModel.java

│ │ │ │ └── utils/

│ │ │ │ └── StripeIntegration.java

│ │ │ └── Application.java

└── test/

├── java/

│ └── com/

│ └── midas/

│ ├── app/

│ │ ├── controllers/

│ │ │ └── ApplicationControllerTest.java

│ │ └── utils/

│ │ └── StripeIntegrationTest.java

│ └── ApplicationTest.java

└── resources/ (if needed)

1. We will be creating src folder in our root directory.
2. In src/main we will be creating java (our Sources root), resource( folder imported from midas github project) and tests(our test sources root).

**III] Programs Insight and Code:-**

1. UserActivity.java (in activities package):

- This file should contain the implementation of the Temporal workflow for user signup.

- Inside this class, you will integrate the Stripe Create Customer API to create a new customer in Stripe upon a new user signup.

- It will orchestrate the necessary steps for user signup, including creating a Stripe customer and storing the generated Stripe customer ID.  
  
package com.midas.app.activities;

import org.springframework.stereotype.Component;

@Component

public class UserActivity {

// Example method to perform some user-related action

public void createUser(String username, String email) {

// Logic to create a user

System.*out*.println("User created with username: " + username + " and email: " + email);

}

}

2. ApplicationController.java (in controllers package):

- This file should be updated to handle the new fields in the user model and manage the user signup process.

- Add logic to generate and store the providerId appropriately during the user signup process.

- Update the endpoints to handle the user signup and expose the GET /accounts endpoint for testing purposes.  
  
package com.midas.app.controllers;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ApplicationController {

private final midas.app.utils.StripeIntegration stripeIntegration;

// Constructor to inject StripeIntegration dependency

public ApplicationController(midas.app.utils.StripeIntegration stripeIntegration) {

this.stripeIntegration = stripeIntegration;

}

@PostMapping("/signup")

public void signup(@RequestBody midas.app.models.UserModel user) {

try {

// Call Stripe integration to create customer

String customerId = stripeIntegration.*createCustomer*(user.getEmail());

// Update user model with new fields

user.setProviderType("stripe");

user.setProviderId(customerId);

// Save user to database or perform further operations

} catch (Exception e) {

e.printStackTrace();

// Handle error

}

}

}

3. UserModel.java (in models package):

- Add a new field named `providerType` to the user model with an enum type having values like `stripe`.

- Add a field named `providerId` to store the generated Stripe customer ID.  
  
package com.midas.app.models;

public class UserModel {

private String email;

private String providerType;

private String providerId;

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getProviderType() {

return providerType;

}

public void setProviderType(String providerType) {

this.providerType = providerType;

}

public String getProviderId() {

return providerId;

}

public void setProviderId(String providerId) {

this.providerId = providerId;

}

}

4. StripeIntegration.java (in utils package):

- This file should contain the implementation of the integration with the Stripe SDK.

- Implement methods to interact with the Stripe API, such as creating a new customer, handling errors, etc.

- This class will be used by the `UserActivity` class to integrate Stripe for user signup.

package com.midas.app.utils;

import com.stripe.Stripe;

import com.stripe.exception.StripeException;

import com.stripe.model.Customer;

import com.stripe.param.CustomerCreateParams;

public class StripeIntegration {

private static final String *STRIPE\_API\_KEY* = "sk\_test\_51OgQ64SEck1MgIokCBvn1SEDNPsiNP1H8EsgvIav7FnsCXjigfBMxNTgWDlydtX6ozLtJqhT8P7XEykAaufORNwJ00jHM6PvHQ";

public static String createCustomer(String email) throws StripeException {

Stripe.*apiKey* = *STRIPE\_API\_KEY*;

Customer customer = Customer.*create*(

CustomerCreateParams.*builder*()

.setEmail(email)

.build()

);

return customer.getId();

}

}

5. MidasApplication.java:

- Main class of your Spring Boot application.

- This class may include Spring Boot configuration, bean definitions, and application startup logic.

- Ensure that all necessary components are properly initialized and configured here.

package com.midas.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MidasApplication {

public static void main(String[] args) {

SpringApplication.*run*(MidasApplication.class, args);

}

}

6. ApplicationControllerTest.java (in controllers package under test):

- Write unit tests to ensure that the `ApplicationController` behaves as expected.

- Test the signup process, including the generation and storage of the `providerId`.

- Mock dependencies such as the `StripeIntegration` class for isolated testing.  
  
package com.midas.app.controllers;

import com.midas.app.models.UserModel;

import org.junit.jupiter.api.Test;

import org.mockito.Mock;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.mockito.Mockito.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;

@SpringBootTest

class ApplicationControllerTest {

private MockMvc mockMvc;

@Mock

private com.midas.app.utils.StripeIntegration stripeIntegration;

@Test

void signup() throws Exception {

ApplicationController controller = new ApplicationController(stripeIntegration);

mockMvc = MockMvcBuilders.*standaloneSetup*(controller).build();

UserModel user = new UserModel();

user.setEmail("test@example.com");

*when*(stripeIntegration.*createCustomer*(*anyString*())).thenReturn("customer123");

mockMvc.perform(MockMvcRequestBuilders.*post*("/signup")

.contentType(MediaType.*APPLICATION\_JSON*)

.content("{\"email\":\"test@example.com\"}"))

.andExpect(*status*().isOk());

*verify*(stripeIntegration, *times*(1)).*createCustomer*("test@example.com");

}

}

7. StripeIntegrationTest.java (in utils package under test):

- Write unit tests to verify the functionality of the `StripeIntegration` class.

- Test methods for creating a customer, handling errors, etc.

- Use mocking or stubbing to isolate the tests and avoid actual calls to the Stripe API.  
  
package com.midas.app.utils;

import com.stripe.Stripe;

import com.stripe.exception.StripeException;

import com.stripe.model.Customer;

import com.stripe.param.CustomerCreateParams;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

class StripeIntegrationTest {

@Test

void createCustomer() {

// Set your Stripe API key

Stripe.*apiKey* = "sk\_test\_51OgQ64SEck1MgIokCBvn1SEDNPsiNP1H8EsgvIav7FnsCXjigfBMxNTgWDlydtX6ozLtJqhT8P7XEykAaufORNwJ00jHM6PvHQ";

try {

// Create a new customer with a test email

Customer customer = Customer.*create*(

CustomerCreateParams.*builder*()

.setEmail("test@example.com")

.build()

);

// Assert that the customer ID is not null or empty

*assertNotNull*(customer.getId());

*assertFalse*(customer.getId().isEmpty());

// Print the customer ID for reference

System.*out*.println("Customer ID: " + customer.getId());

} catch (StripeException e) {

// Handle Stripe API exceptions

e.printStackTrace();

*fail*("StripeException occurred: " + e.getMessage());

}

}

}

8. ApplicationTest.java (in the root of the test package):

- Integration tests to verify the overall functionality of your Spring Boot application.

- Test the signup process end-to-end, including interactions with the Stripe API.

- Test the GET /accounts endpoint to ensure that it returns the expected data.

package com.midas.app;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.context.ApplicationContext;

import static org.junit.jupiter.api.Assertions.*assertNotNull*;

@SpringBootTest

class ApplicationTest {

@Autowired

private ApplicationContext applicationContext;

@Test

void contextLoads() {

// Check if the Spring application context is not null

*assertNotNull*(applicationContext);

}

}

