Development Scenario: Insurance Claim Processing System

Day 1: HTML, CSS, and JavaScript - User Authentication and Profile Setup

Task 1: Design and code the HTML forms for user registration and login, ensuring accessibility standards are met.

**Registration Form:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Registration Form</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<div class="container mt-5">

<h2>Register</h2>

<form id="registrationForm">

<div class="form-group">

<label for="username">Username</label>

<input type="text" class="form-control" id="username" placeholder="Enter username" aria-describedby="usernameHelp" required>

<small id="usernameHelp" class="form-text text-muted">Your username must be unique.</small>

</div>

<div class="form-group">

<label for="email">Email address</label>

<input type="email" class="form-control" id="email" placeholder="Enter email" required>

</div>

<div class="form-group">

<label for="password">Password</label>

<input type="password" class="form-control" id="password" placeholder="Password" required>

</div>

<div class="form-group">

<label for="confirmPassword">Confirm Password</label>

<input type="password" class="form-control" id="confirmPassword" placeholder="Confirm Password" required>

</div>

<button type="submit" class="btn btn-primary">Register</button>

</form>

</div>

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.3/dist/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

</body>

</html>

**Login Form:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Login Form</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<div class="container mt-5">

<h2>Login</h2>

<form id="loginForm">

<div class="form-group">

<label for="loginEmail">Email address</label>

<input type="email" class="form-control" id="loginEmail" placeholder="Enter email" required>

</div>

<div class="form-group">

<label for="loginPassword">Password</label>

<input type="password" class="form-control" id="loginPassword" placeholder="Password" required>

</div>

<button type="submit" class="btn btn-primary">Login</button>

</form>

</div>

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.3/dist/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

</body>

</html>

Task 2: Apply CSS to style the forms for a consistent look and feel that aligns with the company's branding.

body {

font-family: Arial, sans-serif;

background-color: #f8f9fa;

color: #333;

}

.container {

max-width: 500px;

margin: auto;

}

h2 {

margin-bottom: 20px;

color: #007bff;

}

.form-group {

margin-bottom: 15px;

}

.btn-primary {

background-color: #007bff;

border-color: #007bff;

}

.btn-primary:hover {

background-color: #0056b3;

border-color: #0056b3;

}

.form-text {

color: #6c757d;

}

Task 3: Implement JavaScript form validations to provide immediate feedback on user input errors before submission.

<script>

document.getElementById('registrationForm').addEventListener('submit', function(event) {

event.preventDefault();

const username = document.getElementById('username').value;

const email = document.getElementById('email').value;

const password = document.getElementById('password').value;

const confirmPassword = document.getElementById('confirmPassword').value;

let valid = true;

document.querySelectorAll('.error').forEach(el => el.remove());

if (username.length < 3) {

valid = false;

showError('username', 'Username must be at least 3 characters long.');

}

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (!emailRegex.test(email)) {

valid = false;

showError('email', 'Please enter a valid email address.');

}

if (password.length < 6) {

valid = false;

showError('password', 'Password must be at least 6 characters long.');

}

if (password !== confirmPassword) {

valid = false;

showError('confirmPassword', 'Passwords do not match.');

}

if (valid) {

alert('Registration successful!');

}

});

function showError(inputId, message) {

const inputElement = document.getElementById(inputId);

const errorElement = document.createElement('div');

errorElement.className = 'error text-danger';

errorElement.textContent = message;

inputElement.parentNode.appendChild(errorElement);

}

document.getElementById('loginForm').addEventListener('submit', function(event) {

event.preventDefault();

const email = document.getElementById('loginEmail').value;

const password = document.getElementById('loginPassword').value;

let valid = true;

document.querySelectorAll('.error').forEach(el => el.remove());

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (!emailRegex.test(email)) {

valid = false;

showError('loginEmail', 'Please enter a valid email address.');

}

if (password.length < 6) {

valid = false;

showError('loginPassword', 'Password must be at least 6 characters long.');

}

if (valid) {

alert('Login successful!');

}

});

function showError(inputId, message) {

const inputElement = document.getElementById(inputId);

const errorElement = document.createElement('div');

errorElement.className = 'error text-danger';

errorElement.textContent = message;

inputElement.parentNode.appendChild(errorElement);

}

</script>

Day 2: JavaScript/Bootstrap - Responsive Dashboard for Policy Management

Task 1: Create a dashboard layout with Bootstrap ensuring responsiveness across devices.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Responsive Dashboard</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

<style>

body {

font-family: Arial, sans-serif;

background-color: #f8f9fa;

color: #333;

}

.sidebar {

height: 100vh;

position: fixed;

top: 0;

left: 0;

padding: 1rem;

background-color: #343a40;

color: white;

}

.sidebar a {

color: white;

text-decoration: none;

}

.sidebar a:hover {

color: #007bff;

}

.content {

margin-left: 250px;

padding: 1rem;

}

@media (max-width: 768px) {

.sidebar {

position: static;

height: auto;

margin-bottom: 1rem;

}

.content {

margin-left: 0;

}

}

</style>

</head>

<body>

<div class="sidebar">

<h3>Dashboard</h3>

<ul class="nav flex-column">

<li class="nav-item">

<a class="nav-link" href="#">Home</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Policies</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Claims</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Reports</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Settings</a>

</li>

</ul>

</div>

<div class="content">

<div class="container-fluid">

<h2>Welcome to the Policy Management Dashboard</h2>

<div class="row">

<div class="col-md-6 col-lg-4 mb-4">

<div class="card">

<div class="card-body">

<h5 class="card-title">Policy Overview</h5>

<p class="card-text">Details about policies.</p>

<a href="#" class="btn btn-primary">View Details</a>

</div>

</div>

</div>

<div class="col-md-6 col-lg-4 mb-4">

<div class="card">

<div class="card-body">

<h5 class="card-title">Claim Statistics</h5>

<p class="card-text">Overview of claims.</p>

<a href="#" class="btn btn-primary">View Details</a>

</div>

</div>

</div>

<div class="col-md-6 col-lg-4 mb-4">

<div class="card">

<div class="card-body">

<h5 class="card-title">Reports</h5>

<p class="card-text">Various reports.</p>

<a href="#" class="btn btn-primary">View Details</a>

</div>

</div>

</div>

</div>

</div>

</div>

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.3/dist/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

</body>

</html>

Task 2: Utilize Bootstrap's JavaScript components like tabs and modals to enrich the policy management interface.

<div class="content">

<div class="container-fluid">

<h2>Welcome to the Policy Management Dashboard</h2>

<ul class="nav nav-tabs mb-4" id="policyTabs" role="tablist">

<li class="nav-item">

<a class="nav-link active" id="home-tab" data-toggle="tab" href="#home" role="tab" aria-controls="home" aria-selected="true">Home Insurance</a>

</li>

<li class="nav-item">

<a class="nav-link" id="auto-tab" data-toggle="tab" href="#auto" role="tab" aria-controls="auto" aria-selected="false">Auto Insurance</a>

</li>

<li class="nav-item">

<a class="nav-link" id="health-tab" data-toggle="tab" href="#health" role="tab" aria-controls="health" aria-selected="false">Health Insurance</a>

</li>

</ul>

<div class="tab-content" id="policyTabsContent">

<div class="tab-pane fade show active" id="home" role="tabpanel" aria-labelledby="home-tab">

<!-- Content for Home Insurance -->

<div class="row">

<div class="col-md-6 col-lg-4 mb-4">

<div class="card">

<div class="card-body">

<h5 class="card-title">Policy 1</h5>

<p class="card-text">Details about Policy 1.</p>

<a href="#" class="btn btn-primary">View Details</a>

</div>

</div>

</div>

<!-- Add more cards or content as needed -->

</div>

</div>

<div class="tab-pane fade" id="auto" role="tabpanel" aria-labelledby="auto-tab">

<!-- Content for Auto Insurance -->

<p>Content for Auto Insurance policies.</p>

</div>

<div class="tab-pane fade" id="health" role="tabpanel" aria-labelledby="health-tab">

<!-- Content for Health Insurance -->

<p>Content for Health Insurance policies.</p>

</div>

</div>

</div>

</div>

Task 3: Enhance dashboard interactivity with JavaScript for policy sorting and detailed views.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Responsive Dashboard with JavaScript Interactivity</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

<style>

body {

font-family: Arial, sans-serif;

background-color: #f8f9fa;

color: #333;

}

.sidebar {

height: 100vh;

position: fixed;

top: 0;

left: 0;

padding: 1rem;

background-color: #343a40;

color: white;

}

.sidebar a {

color: white;

text-decoration: none;

}

.sidebar a:hover {

color: #007bff;

}

.content {

margin-left: 250px;

padding: 1rem;

}

@media (max-width: 768px) {

.sidebar {

position: static;

height: auto;

margin-bottom: 1rem;

}

.content {

margin-left: 0;

}

}

</style>

</head>

<body>

<div class="sidebar">

<h3>Dashboard</h3>

<ul class="nav flex-column">

<li class="nav-item">

<a class="nav-link active" id="home-tab" data-toggle="tab" href="#home" role="tab" aria-controls="home" aria-selected="true">Home Insurance</a>

</li>

<li class="nav-item">

<a class="nav-link" id="auto-tab" data-toggle="tab" href="#auto" role="tab" aria-controls="auto" aria-selected="false">Auto Insurance</a>

</li>

<li class="nav-item">

<a class="nav-link" id="health-tab" data-toggle="tab" href="#health" role="tab" aria-controls="health" aria-selected="false">Health Insurance</a>

</li>

</ul>

</div>

<div class="content">

<div class="container-fluid">

<h2>Welcome to the Policy Management Dashboard</h2>

<div class="tab-content" id="policyTabsContent">

<div class="tab-pane fade show active" id="home" role="tabpanel" aria-labelledby="home-tab">

<div class="row" id="policyList">

<!-- Policy cards will be dynamically generated here -->

</div>

</div>

<div class="tab-pane fade" id="auto" role="tabpanel" aria-labelledby="auto-tab">

<p>Content for Auto Insurance policies.</p>

</div>

<div class="tab-pane fade" id="health" role="tabpanel" aria-labelledby="health-tab">

<p>Content for Health Insurance policies.</p>

</div>

</div>

</div>

</div>

<!-- Modal for Policy Details -->

<div class="modal fade" id="policyDetailsModal" tabindex="-1" role="dialog" aria-labelledby="policyDetailsModalLabel" aria-hidden="true">

<div class="modal-dialog modal-dialog-centered" role="document">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title" id="policyDetailsModalLabel">Policy Details</h5>

<button type="button" class="close" data-dismiss="modal" aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

</div>

<div class="modal-body" id="policyDetailsContent">

<!-- Policy details will be dynamically inserted here -->

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>

<button type="button" class="btn btn-primary">Save changes</button>

</div>

</div>

</div>

</div>

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.3/dist/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

<script>

$(document).ready(function() {

// Dummy data for policies

var policies = [

{ name: "Policy 1", type: "Home Insurance", details: "Details about Policy 1." },

{ name: "Policy 2", type: "Home Insurance", details: "Details about Policy 2." },

{ name: "Policy 3", type: "Auto Insurance", details: "Details about Policy 3." },

{ name: "Policy 4", type: "Health Insurance", details: "Details about Policy 4." }

];

// Function to render policy cards

function renderPolicyCards(policies) {

var html = '';

policies.forEach(function(policy) {

html += `

<div class="col-md-6 col-lg-4 mb-4">

<div class="card">

<div class="card-body">

<h5 class="card-title">${policy.name}</h5>

<p class="card-text">${policy.details}</p>

<a href="#" class="btn btn-primary" data-toggle="modal" data-target="#policyDetailsModal" data-policy='${JSON.stringify(policy)}'>View Details</a>

</div>

</div>

</div>

`;

});

$('#policyList').html(html);

}

// Initial rendering of policy cards

renderPolicyCards(policies);

// Sort policies by name

$('#sortByName').click(function() {

policies.sort(function(a, b) {

var nameA = a.name.toUpperCase();

var nameB = b.name.toUpperCase();

if (nameA < nameB) {

return -1;

}

if (nameA > nameB) {

return 1;

}

return 0;

});

renderPolicyCards(policies);

});

// Sort policies by type

$('#sortByType').click(function() {

policies.sort(function(a, b) {

var typeA = a.type.toUpperCase();

var typeB = b.type.toUpperCase();

if (typeA < typeB) {

return -1;

}

if (typeA > typeB) {

return 1;

}

return 0;

});

renderPolicyCards(policies);

});

// Display policy details in modal

$('#policyDetailsModal').on('show.bs.modal', function(event) {

var button = $(event.relatedTarget);

var policy = button.data('policy');

var modal = $(this);

modal.find('.modal-title').text(policy.name);

modal.find('.modal-body').html(`<p>${policy.details}</p>`);

});

});

</script>

</body>

</html>

**Day 3:**

Servlet/JSP, Introduction to JSP - Claims Submission Process

Task 1: Develop Servlets to manage the workflow of submitting insurance claims.

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/submitClaim")

public class SubmitClaimServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String policyNumber = request.getParameter("policyNumber");

String claimAmount = request.getParameter("claimAmount");

String claimDetails = request.getParameter("claimDetails");

PrintWriter out = response.getWriter();

out.println("<html><body>");

out.println("<h1>Claim Submission Successful</h1>");

out.println("<p>Policy Number: " + policyNumber + "</p>");

out.println("<p>Claim Amount: " + claimAmount + "</p>");

out.println("<p>Claim Details: " + claimDetails + "</p>");

out.println("</body></html>");

}

}

Task 2: Construct JSP pages for entering claim information and confirmations.

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Claim Submission Form</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<h2>Claim Submission Form</h2>

<form action="submitClaim" method="post">

<label for="policyNumber">Policy Number:</label>

<input type="text" id="policyNumber" name="policyNumber" required><br><br>

<label for="claimAmount">Claim Amount:</label>

<input type="text" id="claimAmount" name="claimAmount" required><br><br>

<label for="claimDetails">Claim Details:</label><br>

<textarea id="claimDetails" name="claimDetails" rows="4" cols="50" required></textarea><br><br>

<input type="submit" value="Submit Claim">

</form>

</body>

</html>

Task 3: Employ JavaBeans to manage the transition of data in the claim submission process.

package com.example.beans;

import java.io.Serializable;

public class ClaimBean implements Serializable {

private static final long serialVersionUID = 1L;

private String policyNumber;

private double claimAmount;

private String claimDetails;

// Default constructor (mandatory for JavaBeans)

public ClaimBean() {

}

public String getPolicyNumber() {

return policyNumber;

}

public void setPolicyNumber(String policyNumber) {

this.policyNumber = policyNumber;

}

public double getClaimAmount() {

return claimAmount;

}

public void setClaimAmount(double claimAmount) {

this.claimAmount = claimAmount;

}

public String getClaimDetails() {

return claimDetails;

}

public void setClaimDetails(String claimDetails) {

this.claimDetails = claimDetails;

}

}

Day 4: Spring Core - Policy Administration Backend

Task 1: Refactor policy-related operations to utilize Spring Beans and Dependency Injection.

package com.example.policy;

public class Policy {

private String policyNumber;

private String policyHolder;

private double premiumAmount;

public String getPolicyNumber() {

return policyNumber;

}

public void setPolicyNumber(String policyNumber) {

this.policyNumber = policyNumber;

}

public String getPolicyHolder() {

return policyHolder;

}

public void setPolicyHolder(String policyHolder) {

this.policyHolder = policyHolder;

}

public double getPremiumAmount() {

return premiumAmount;

}

public void setPremiumAmount(double premiumAmount) {

this.premiumAmount = premiumAmount;

}

}

Task 2: Implement Spring validation on the server side to ensure policy data integrity.

package com.example.policy;

import org.springframework.validation.Errors;

import org.springframework.validation.ValidationUtils;

import org.springframework.validation.Validator;

public class PolicyValidator implements Validator {

@Override

public boolean supports(Class<?> clazz) {

return Policy.class.equals(clazz);

}

@Override

public void validate(Object target, Errors errors) {

Policy policy = (Policy) target;

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "policyNumber", "policyNumber.required", "Policy number is required.");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "policyHolder", "policyHolder.required", "Policy holder is required.");

if (policy.getPremiumAmount() <= 0) {

errors.rejectValue("premiumAmount", "premiumAmount.invalid", "Premium amount must be greater than zero.");

}

}

}

Task 3: Set up Application Context and Bean Factory for a scalable backend structure.

package com.example.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import com.example.policy.PolicyService;

import com.example.policy.PolicyValidator;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "com.example.controller", "com.example.policy" })

public class AppConfig {

@Bean

public PolicyService policyService() {

return new PolicyService(policyRepository());

}

@Bean

public PolicyValidator policyValidator() {

return new PolicyValidator();

}

}

package com.example.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer { @Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] { AppConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

return null;

}

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

Day 5: Spring MVC - User Claim Interaction Workflow

Task 1: Migrate front-end form handling to Spring MVC controllers.

First we have to create html page

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Claim Submission Form</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<h2>Claim Submission Form</h2>

<form action="${pageContext.request.contextPath}/submitClaim" method="post">

<label for="policyNumber">Policy Number:</label>

<input type="text" id="policyNumber" name="policyNumber" required><br><br>

<label for="claimAmount">Claim Amount:</label>

<input type="text" id="claimAmount" name="claimAmount" required><br><br>

<label for="claimDetails">Claim Details:</label><br>

<textarea id="claimDetails" name="claimDetails" rows="4" cols="50" required></textarea><br><br>

<input type="submit" value="Submit Claim">

</form>

</body>

</html>

Now we have to create mvc controller

package com.example.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

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

@Controller

@RequestMapping("/claim")

public class ClaimController {

@PostMapping("/submitClaim")

public String submitClaim(@RequestParam("policyNumber") String policyNumber,

@RequestParam("claimAmount") Double claimAmount,

@RequestParam("claimDetails") String claimDetails,

Model model) {

model.addAttribute("successMessage", "Claim submitted successfully!");

return "claimConfirmation";

}

}

Task 2: Configure Thymeleaf as the view layer for dynamic content rendering in Spring MVC.

First add thymeleaf dependency

package com.example.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

import org.thymeleaf.spring5.SpringTemplateEngine;

import org.thymeleaf.spring5.view.ThymeleafViewResolver;

import org.thymeleaf.templatemode.TemplateMode;

import org.thymeleaf.templateresolver.ClassLoaderTemplateResolver;

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Bean

public ClassLoaderTemplateResolver templateResolver() {

ClassLoaderTemplateResolver templateResolver = new ClassLoaderTemplateResolver();

templateResolver.setPrefix("/WEB-INF/views/");

templateResolver.setSuffix(".html");

templateResolver.setTemplateMode(TemplateMode.HTML);

return templateResolver;

}

@Bean

public SpringTemplateEngine templateEngine() {

SpringTemplateEngine templateEngine = new SpringTemplateEngine();

templateEngine.setTemplateResolver(templateResolver());

return templateEngine;

}

@Bean

public ViewResolver thymeleafViewResolver() {

ThymeleafViewResolver resolver = new ThymeleafViewResolver();

resolver.setTemplateEngine(templateEngine());

resolver.setCharacterEncoding("UTF-8");

return resolver;

}

}

Task 3: Implement data binding and server-side validation within the Spring MVC framework.

package com.example.model;

import javax.validation.constraints.NotBlank;

import javax.validation.constraints.NotNull;

public class Claim {

@NotBlank(message = "Policy number is required")

private String policyNumber;

@NotNull(message = "Claim amount is required")

private Double claimAmount;

@NotBlank(message = "Claim details are required")

private String claimDetails;

public String getPolicyNumber() {

return policyNumber;

}

public void setPolicyNumber(String policyNumber) {

this.policyNumber = policyNumber;

}

public Double getClaimAmount() {

return claimAmount;

}

public void setClaimAmount(Double claimAmount) {

this.claimAmount = claimAmount;

}

public String getClaimDetails() {

return claimDetails;

}

public void setClaimDetails(String claimDetails) {

this.claimDetails = claimDetails;

}

}

Day 6: Object Relational Mapping and Hibernate - Database Integration for Claims and Policies

Task 1: Define Hibernate entity mappings for claim and policy data models.

First create entity class

package com.example.model;

import javax.persistence.\*;

import java.util.Date;

@Entity

@Table(name = "claims")

public class Claim {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "claim\_id")

private Long id;

@Column(name = "policy\_number")

private String policyNumber;

@Column(name = "claim\_amount")

private Double claimAmount;

@Column(name = "claim\_details")

private String claimDetails;

@Column(name = "claim\_date")

private Date claimDate;

}

Task 2: Develop Hibernate DAOs to handle CRUD operations for claims and policies.

package com.example.dao;

import com.example.model.Claim;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

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

import org.springframework.stereotype.Repository;

import javax.persistence.criteria.CriteriaBuilder;

import javax.persistence.criteria.CriteriaQuery;

import javax.persistence.criteria.Root;

import java.util.List;

@Repository

public class ClaimDAO {

@Autowired

private SessionFactory sessionFactory;

public void save(Claim claim) {

getCurrentSession().saveOrUpdate(claim);

}

public Claim findById(Long id) {

return getCurrentSession().get(Claim.class, id);

} public List<Claim> findAll() {

Session session = getCurrentSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Claim> criteria = builder.createQuery(Claim.class);

Root<Claim> root = criteria.from(Claim.class);

criteria.select(root);

return session.createQuery(criteria).getResultList();

}

public void delete(Long id) {

Session session = getCurrentSession();

Claim claim = session.load(Claim.class, id);

session.delete(claim);

}private Session getCurrentSession() {

return sessionFactory.getCurrentSession();

}

}

Task 3: Write and test HQL and Criteria queries for advanced data retrieval and reporting.

package com.example.dao;

import com.example.model.Claim;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

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

import org.springframework.stereotype.Repository;

import javax.persistence.criteria.CriteriaBuilder;

import javax.persistence.criteria.CriteriaQuery;

import javax.persistence.criteria.Root;

import java.util.List;

@Repository

public class ClaimDAO {

@Autowired

private SessionFactory sessionFactory;

private Session getCurrentSession() {

return sessionFactory.getCurrentSession();

} public void save(Claim claim) {

getCurrentSession().saveOrUpdate(claim);

}

public Claim findById(Long id) {

return getCurrentSession().get(Claim.class, id);

}

public List<Claim> findAll() {

Session session = getCurrentSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Claim> criteria = builder.createQuery(Claim.class);

Root<Claim> root = criteria.from(Claim.class);

criteria.select(root);

return session.createQuery(criteria).getResultList();

}

public void delete(Long id) {

Session session = getCurrentSession();

Claim claim = session.load(Claim.class, id);

session.delete(claim);

}

public List<Claim> findByPolicyNumber(String policyNumber) {

String hql = "FROM Claim WHERE policyNumber = :policyNumber";

return getCurrentSession().createQuery(hql, Claim.class)

.setParameter("policyNumber", policyNumber)

.getResultList();

}

public List<Claim> findClaimsAboveAmount(Double amount) {

Session session = getCurrentSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Claim> criteria = builder.createQuery(Claim.class);

Root<Claim> root = criteria.from(Claim.class);

criteria.select(root).where(builder.gt(root.get("claimAmount"), amount));

return session.createQuery(criteria).getResultList();

}

}

Day 7: Spring Boot and Microservices - Microservices for Claim Processing

**1. Define Microservices Architecture**

Begin by outlining the architecture of your microservices. Typically, for claim processing, you might have microservices for:

* **Claim Management**: Handles CRUD operations for claims.
* **Policy Management**: Manages policies and related operations.
* **User Management**: Manages user authentication, profiles, and permissions.
* **Notification Service**: Sends notifications for claim status updates.
* **Reporting Service**: Generates reports and analytics on claims data.

**2. Create Spring Boot Projects**

Each microservice will be a separate Spring Boot project. You can use Spring Initializr or your preferred IDE to generate these projects. Ensure each project includes dependencies like Spring Web, Spring Data JPA, and Spring Boot Actuator.

**3. Implement Microservices**

**Example: Claim Management Microservice**

* **Project Setup**: Create a new Spring Boot project for claim management.
* **Define Entity**: Create a Claim entity class annotated with JPA annotations (@Entity, @Table, @Id, etc.).
* **Repository**: Create a ClaimRepository interface extending JpaRepository<Claim, Long> for CRUD operations.
* **Service Layer**: Implement a ClaimService to encapsulate business logic, using ClaimRepository.
* **Controller**: Create a REST controller (ClaimController) with endpoints for handling HTTP requests (GET, POST, PUT, DELETE) related to claims.
* **Configuration**: Configure database connection (application.properties) and other necessary settings.

**4. Communicate Between Microservices**

Use RESTful APIs or messaging protocols (like Kafka or RabbitMQ) for inter-service communication. Define API contracts and handle service discovery and load balancing using tools like Eureka or Kubernetes.

**5. Implement Security**

Secure microservices with Spring Security and JWT (JSON Web Tokens) for authentication and authorization. Ensure only authenticated users can access protected endpoints.

**6. Testing and Deployment**

* **Unit Testing**: Write unit tests using JUnit and Mockito to test service methods.
* **Integration Testing**: Verify interactions between microservices using tools like SpringBootTest.
* **Deployment**: Deploy microservices independently using Docker containers or deploy to cloud platforms like AWS, Azure, or Google Cloud.

**7. Monitoring and Logging**

Utilize Spring Boot Actuator for monitoring endpoints (/actuator) and integrate with logging frameworks like Logback or Log4j for centralized logging.

**8. Scaling**

Scale microservices horizontally to handle increased loads by deploying multiple instances behind a load balancer.

**Summary**

Implementing microservices with Spring Boot for claim processing involves creating modular, lightweight services that collaborate through well-defined APIs. This approach enhances flexibility, scalability, and maintainability of the system. Customize each microservice to handle specific functionalities (claim management, policy management, etc.), ensure secure communication between services, and deploy using modern DevOps practices for robust, scalable applications.

Task 1: Transition the monolithic application structure to a microservices architecture using Spring Boot.

Transitioning from a monolithic application structure to a microservices architecture using Spring Boot involves breaking down the existing monolith into smaller, independent services that communicate over well-defined APIs. Here’s a structured approach to achieve this transition:

**Task 1: Transition to Microservices Architecture Using Spring Boot**

**1. Identify Microservices Boundaries**

* **Domain Decomposition**: Analyze the existing monolithic application to identify distinct domains or functionalities that can be separated into microservices. For instance, separate services for user management, claims processing, policy management, etc.

**2. Create Spring Boot Projects**

* **Microservice Projects**: Set up individual Spring Boot projects for each microservice. Use Spring Initializr or your IDE to generate projects with necessary dependencies (Spring Web, Spring Data JPA, etc.).

**3. Refactor Existing Functionality**

* **Extract Services**: Identify and extract specific functionalities from the monolith into separate microservices. For example:
  + **User Management**: Create a microservice handling user authentication, registration, and profile management.
  + **Claims Management**: Develop a microservice for managing insurance claims, including CRUD operations and business logic related to claims processing.
  + **Policy Management**: Separate service responsible for policy-related operations, such as creating, updating, and retrieving policies.

**4. Define APIs for Communication**

* **RESTful Endpoints**: Define RESTful APIs for communication between microservices. Use Spring Web to create controllers with endpoints that expose functionalities of each microservice.
* **API Gateway**: Consider implementing an API Gateway (using Spring Cloud Gateway or Zuul) for routing requests to appropriate microservices and handling cross-cutting concerns like authentication and logging.

**5. Implement Data Management**

* **Database per Service**: Each microservice manages its own database. Use Spring Data JPA or another ORM framework to interact with databases, ensuring data isolation and autonomy for each service.
* **Eventual Consistency**: Adopt eventual consistency patterns where needed, ensuring data synchronization across services through messaging (e.g., Kafka, RabbitMQ) or eventual consistency mechanisms.

**6. Implement Service Discovery and Configuration**

* **Service Discovery**: Use tools like Eureka (from Spring Cloud Netflix) or Kubernetes Service Discovery for dynamic service registration and discovery.
* **Configuration Management**: Utilize Spring Cloud Config for centralized configuration management across microservices, ensuring consistency and easy updates.

**7. Implement Security**

* **Authentication and Authorization**: Secure microservices using Spring Security and JWT (JSON Web Tokens). Implement authentication mechanisms to control access to APIs and ensure data security.

**8. Testing and Deployment**

* **Unit and Integration Testing**: Write unit tests for individual microservices and integration tests to validate interactions between services.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines using tools like Jenkins, GitLab CI, or AWS CodePipeline to automate build, test, and deployment processes.

**9. Monitor and Manage Microservices**

* **Monitoring**: Use Spring Boot Actuator for monitoring endpoints (/actuator) to monitor health, metrics, and other aspects of microservices.
* **Logging and Tracing**: Integrate with centralized logging solutions (e.g., ELK stack) and distributed tracing (e.g., Zipkin) for visibility into microservices interactions.

**10. Scale Horizontally**

* **Scalability**: Design microservices to be horizontally scalable. Use container orchestration platforms like Kubernetes for managing deployments and scaling based on demand.

**Summary**

Transitioning from a monolithic architecture to microservices using Spring Boot involves breaking down functionality into independent, loosely coupled services that communicate through APIs. This approach enhances scalability, flexibility, and maintainability of applications, enabling faster development cycles and easier deployment. Customize each microservice to handle specific business domains, ensure seamless communication through defined APIs, and leverage modern DevOps practices for efficient management and operation of microservices-based applications.

Task 2: Implement service discovery with Eureka and develop Feign clients for inter-service communication.

To implement service discovery with Eureka and develop Feign clients for inter-service communication in a microservices architecture using Spring Boot, follow these steps:

**Task 2: Implement Service Discovery and Feign Clients**

**1. Set Up Eureka Server**

First, create a Eureka Server to act as a service registry:

**1.1. Create Eureka Server Project**

Create a new Spring Boot project for Eureka Server using Spring Initializr:

* **Dependencies**: Select Eureka Server dependency.

**1.2. Configure Eureka Server**

In the main application class (EurekaServerApplication.java), annotate with @EnableEurekaServer to enable Eureka Server functionality:

java

package com.example.eurekaserver;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

**1.3. Application Properties**

Configure application.properties for Eureka Server (src/main/resources/application.properties):

properties

spring.application.name=eureka-server

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**2. Create Microservices**

Create microservices that register with Eureka Server and communicate using Feign clients.

**2.1. Microservice Project Setup**

For each microservice (e.g., UserService, ClaimsService, PolicyService), set up Spring Boot projects similar to how you created Eureka Server.

* **Dependencies**: Include Eureka Discovery Client, Spring Web, Spring Data JPA, etc.

**2.2. Configure Eureka Client**

In each microservice, configure it as a Eureka Client to register with Eureka Server and discover other services:

* **Application Properties** (application.properties):

properties

spring.application.name=user-service

server.port=8081 # Specify different ports for each service

eureka.client.service-url.default-zone=http://localhost:8761/eureka

* **Main Class** (UserServiceApplication.java):

java

package com.example.userservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class UserServiceApplication {

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

}

**2.3. Define Feign Clients**

Use Feign clients for declarative REST client to simplify HTTP API calls between microservices.

* **Dependencies**: Include Spring Cloud Starter OpenFeign dependency.
* **Feign Client Interface** (UserServiceFeignClient.java):

java

package com.example.userservice.feign;

import org.springframework.cloud.openfeign.FeignClient;

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

@FeignClient(name = "claims-service")

public interface ClaimsServiceFeignClient {

@GetMapping("/claims") // Define endpoints similar to Controller mappings

List<Claim> getAllClaims();

}

**2.4. Use Feign Clients in Controllers or Services**

Inject Feign clients into controllers or services to invoke methods from other microservices:

* **Controller or Service Class** (UserController.java):

java

package com.example.userservice.controller;

import com.example.userservice.feign.ClaimsServiceFeignClient;

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

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

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

@RestController

public class UserController {

@Autowired

private ClaimsServiceFeignClient claimsServiceFeignClient;

@GetMapping("/users/{userId}/claims")

public List<Claim> getUserClaims(@PathVariable Long userId) {

return claimsServiceFeignClient.getAllClaims();

}}

**3. Testing and Deployment**

* **Run Eureka Server**: Start Eureka Server (EurekaServerApplication) first.
* **Run Microservices**: Start each microservice (UserServiceApplication, ClaimsServiceApplication, etc.) to register with Eureka Server.
* **Test Feign Clients**: Verify inter-service communication by invoking endpoints exposed by Feign clients.

Task 3: Set up and configure Spring Cloud Config for centralized configuration management of microservices.

Setting up and configuring Spring Cloud Config for centralized configuration management of microservices involves creating a centralized configuration server and enabling microservices to fetch their configuration from this server. Here’s a step-by-step guide to accomplish Task 3:

**Task 3: Set up and Configure Spring Cloud Config**

**1. Create a Spring Cloud Config Server**

Create a new Spring Boot project for the Config Server:

**1.1. Project Setup**

Use Spring Initializr to generate a new project with the following dependencies:

* **Dependencies**: Select Config Server dependency.

**1.2. Configure as Config Server**

In the main application class (ConfigServerApplication.java), annotate with @EnableConfigServer to enable the Config Server functionality:

java

package com.example.configserver;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

**1.3. Configure Application Properties**

Configure application.properties (src/main/resources/application.properties) or application.yml for the Config Server:

properties

spring.application.name=config-server

server.port=8888 # Port for the Config Server

spring.cloud.config.server.git.uri=https://github.com/your-repo/config-repo.git

# Specify Git repository URL containing configuration files

* **Note**: Replace https://github.com/your-repo/config-repo.git with your actual Git repository URL containing configuration files.

**2. Create Microservices**

Configure microservices to fetch configurations from the Config Server.

**2.1. Project Setup**

Set up each microservice as a Spring Boot project with necessary dependencies (Spring Web, Spring Cloud Starter Config, etc.).

**2.2. Configure as Config Client**

In each microservice’s application.properties or application.yml, configure it as a Config Client to fetch configurations from the Config Server:

* **Application Properties** (application.properties):

properties

Copy code

spring.application.name=user-service

server.port=8081 # Specify different ports for each service

spring.cloud.config.uri=http://localhost:8888

# URL of Config Server

# Specify profile and label if needed

spring.profiles.active=dev

**2.3. Use Configuration Properties**

Inject configuration properties (@Value, @ConfigurationProperties) into microservices to access configurations fetched from the Config Server.

java

package com.example.userservice.config;

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

import org.springframework.context.annotation.Configuration;

@Configuration

public class UserServiceConfig {

@Value("${user-service.max-claims-per-user}")

private int maxClaimsPerUser;

}

**3. Testing and Deployment**

* **Run Config Server**: Start Config Server (ConfigServerApplication) to serve configurations.
* **Run Microservices**: Start each microservice (UserServiceApplication, ClaimsServiceApplication, etc.) to fetch configurations from Config Server.
* **Verify Configuration**: Ensure microservices correctly fetch and utilize configurations from Config Server.

Day 8: Reactive Spring - Real-time Claim Status Updates

Task 1: Introduce Spring WebFlux for handling real-time claim status updates using reactive streams.

**1. Setup Reactive Spring Boot Project**

Start by creating a new Spring Boot project with WebFlux dependencies to handle reactive programming:

**1.1. Project Setup**

Use Spring Initializr to generate a new project with the following dependencies:

* **Dependencies**: Select Reactive Web and optionally Spring Data Reactive MongoDB or Spring Data Reactive R2DBC depending on your data storage needs.

**1.2. Configure WebFlux**

Ensure your main application class is set up to enable reactive programming:

java

package com.example.reactiveclaims;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ReactiveClaimsApplication {

public static void main(String[] args) {

SpringApplication.run(ReactiveClaimsApplication.class, args);

}

}

**2. Define Claim Entity**

Create a Claim entity class with relevant fields for claim status updates:

java

package com.example.reactiveclaims.model;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

@Document

public class Claim {

@Id

private String id;

private String claimNumber;

private String status;

// Getters and setters

}

**3. Implement Reactive Repository**

If using MongoDB or R2DBC, implement a reactive repository for CRUD operations:

**3.1. Reactive MongoDB Repository**

java

package com.example.reactiveclaims.repository;

import com.example.reactiveclaims.model.Claim;

import org.springframework.data.mongodb.repository.ReactiveMongoRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface ClaimRepository extends ReactiveMongoRepository<Claim, String> {

}

**4. Create Reactive REST Controller**

Define a reactive REST controller to handle real-time updates and queries:

**4.1. Reactive Controller**

java

package com.example.reactiveclaims.controller;

import com.example.reactiveclaims.model.Claim;

import com.example.reactiveclaims.repository.ClaimRepository;

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

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/claims")

public class ClaimController {

@Autowired

private ClaimRepository claimRepository;

@GetMapping

public Flux<Claim> getAllClaims() {

return claimRepository.findAll();

}

@PostMapping

public Mono<Claim> createClaim(@RequestBody Claim claim) {

return claimRepository.save(claim);

}

// Other endpoints for updating and deleting claims

}

**5. Implement Real-time Updates with WebFlux**

Use reactive streams to push real-time updates to clients:

**5.1. Reactive Endpoints**

java

package com.example.reactiveclaims.controller;

import com.example.reactiveclaims.model.Claim;

import com.example.reactiveclaims.repository.ClaimRepository;

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

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/claims")

public class ClaimController {

@Autowired

private ClaimRepository claimRepository;

@GetMapping

public Flux<Claim> getAllClaims() {

return claimRepository.findAll();

}

@PostMapping

public Mono<Claim> createClaim(@RequestBody Claim claim) {

return claimRepository.save(claim);

}

@GetMapping("/{id}")

public Mono<Claim> getClaimById(@PathVariable String id) {

return claimRepository.findById(id);

}

@PutMapping("/{id}")

public Mono<Claim> updateClaim(@PathVariable String id, @RequestBody Claim claim) {

return claimRepository.findById(id)

.flatMap(existingClaim -> {

existingClaim.setStatus(claim.getStatus());

return claimRepository.save(existingClaim);

});

}

@DeleteMapping("/{id}")

public Mono<Void> deleteClaim(@PathVariable String id) {

return claimRepository.deleteById(id);

}

@GetMapping(value = "/{id}/status", produces = "text/event-stream")

public Flux<String> getClaimStatusUpdates(@PathVariable String id) {

return claimRepository.findById(id)

.flatMapMany(claim -> {

return Flux.interval(Duration.ofSeconds(5)) // Emit every 5 seconds

.map(sequence -> claim.getStatus());

});

}

}

**6. Testing and Deployment**

* **Run Application**: Start your Spring Boot application (ReactiveClaimsApplication) to deploy the reactive endpoints.
* **Test Endpoints**: Use tools like Postman or curl to test CRUD operations and real-time updates (/claims/{id}/status) endpoint.
* **Monitor Reactivity**: Monitor reactive streams and performance using Spring Boot Actuator and logging frameworks.

Task 2: Configure R2DBC for reactive database connectivity to update claim status dynamically.

Configuring R2DBC (Reactive Relational Database Connectivity) for reactive database connectivity in Spring Boot allows you to handle database operations reactively, suitable for real-time applications like updating claim statuses dynamically. R2DBC provides non-blocking database access for relational databases, contrasting with traditional blocking JDBC connections. Here’s how you can set it up:

**Task 2: Configure R2DBC for Reactive Database Connectivity**

**1. Add Dependencies**

Start by adding necessary dependencies for R2DBC and the database driver (e.g., PostgreSQL, MySQL):

**1.1. Maven Dependencies**

Add dependencies to your pom.xml for R2DBC and the database driver. For example, using PostgreSQL:

xml

<dependencies>

<!-- R2DBC Postgres driver -->

<dependency>

<groupId>io.r2dbc</groupId>

<artifactId>r2dbc-postgresql</artifactId>

<version>0.9.0.RELEASE</version>

</dependency>

<!-- Spring Boot Starter Data R2DBC -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-r2dbc</artifactId>

</dependency>

<!-- Spring Boot Starter Webflux (if not already included) -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

</dependencies>

**1.2. Gradle Dependencies**

For Gradle, include dependencies in build.gradle:

groovy

dependencies {

implementation 'io.r2dbc:r2dbc-postgresql:0.9.0.RELEASE'

implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'

implementation 'org.springframework.boot:spring-boot-starter-webflux'

}

**2. Configure Database Connection**

Configure R2DBC connection properties in application.properties or application.yml:

**2.1. Application Properties**

properties

# PostgreSQL example

spring.r2dbc.url=r2dbc:postgresql://localhost:5432/yourdatabase

spring.r2dbc.username=db\_username

spring.r2dbc.password=db\_password

spring.r2dbc.pool.initial-size=2

**2.2. Application YAML**

yaml

Copy code

# PostgreSQL example

spring:

r2dbc:

url: r2dbc:postgresql://localhost:5432/yourdatabase

username: db\_username

password: db\_password

pool:

initial-size: 2

**3. Define Entity and Repository**

Define an entity and repository for interacting with the database using R2DBC:

**3.1. Entity Class**

Java

package com.example.reactiveclaims.model;

import org.springframework.data.annotation.Id;

import org.springframework.data.relational.core.mapping.Table;

@Table("claims")

public class Claim {

@Id

private Long id;

private String claimNumber;

private String status;

}

**3.2. Reactive Repository**

Create a reactive repository interface extending ReactiveCrudRepository:

java

package com.example.reactiveclaims.repository;

import com.example.reactiveclaims.model.Claim;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;

public interface ClaimRepository extends ReactiveCrudRepository<Claim, Long> {

}

**4. Update Claim Status Reactively**

Implement reactive endpoints to update claim status dynamically using R2DBC:

**4.1. Reactive Controller**

java

Copy code

package com.example.reactiveclaims.controller;

import com.example.reactiveclaims.model.Claim;

import com.example.reactiveclaims.repository.ClaimRepository;

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

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/claims")

public class ClaimController {

@Autowired

private ClaimRepository claimRepository;

@GetMapping

public Flux<Claim> getAllClaims() {

return claimRepository.findAll();

}

@PostMapping

public Mono<Claim> createClaim(@RequestBody Claim claim) {

return claimRepository.save(claim);

} @GetMapping("/{id}")

public Mono<Claim> getClaimById(@PathVariable Long id) {

return claimRepository.findById(id);

}

@PutMapping("/{id}")

public Mono<Claim> updateClaimStatus(@PathVariable Long id, @RequestBody Claim updatedClaim) {

return claimRepository.findById(id)

.flatMap(existingClaim -> {

existingClaim.setStatus(updatedClaim.getStatus());

return claimRepository.save(existingClaim);

});

}

}

**5. Testing and Deployment**

* **Run Application**: Start your Spring Boot application (ReactiveClaimsApplication) to deploy reactive endpoints using R2DBC.
* **Test Endpoints**: Use tools like Postman or curl to test CRUD operations and verify dynamic updates of claim statuses.
* **Monitor Reactivity**: Monitor reactive streams and performance using Spring Boot Actuator and logging frameworks.

Task 3: Implement WebSocket communication for real-time interaction between the client and the server.

Implementing WebSocket communication in Spring Boot allows for real-time interaction between clients and servers, suitable for applications requiring instant updates and notifications. Here’s a guide to implement Task 3 using WebSocket in Spring Boot:

**Task 3: Implement WebSocket Communication**

**1. Add Dependencies**

Start by adding necessary dependencies for WebSocket support in your pom.xml or build.gradle:

**1.1. Maven Dependencies**

Add dependencies for WebSocket and Spring Boot Starter Web:

xml

Copy code

<dependencies>

<!-- Spring Boot Starter Web for general web support -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Boot Starter WebSocket for WebSocket support -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

</dependencies>

**1.2. Gradle Dependencies**

For Gradle, include dependencies in build.gradle:

groovy

Copy code

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'org.springframework.boot:spring-boot-starter-websocket'

}

**2. Configure WebSocket Endpoint**

Create a WebSocket endpoint handler in your Spring Boot application:

**2.1. WebSocket Handler**

Create a handler class that extends TextWebSocketHandler to manage WebSocket connections and messages:

java

package com.example.reactiveclaims.websocket;

import org.springframework.stereotype.Component;

import org.springframework.web.socket.TextMessage;

import org.springframework.web.socket.WebSocketSession;

import org.springframework.web.socket.handler.TextWebSocketHandler;

import java.io.IOException;

import java.util.HashSet;

import java.util.Set;

@Component

public class ClaimStatusWebSocketHandler extends TextWebSocketHandler {

private Set<WebSocketSession> sessions = new HashSet<>();

@Override

public void afterConnectionEstablished(WebSocketSession session) throws Exception {

sessions.add(session);

}

@Override

protected void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {

// Handle incoming messages (if needed)

}

@Override

public void afterConnectionClosed(WebSocketSession session, org.springframework.web.socket.CloseStatus closeStatus) throws Exception {

sessions.remove(session);

}

// Method to send messages to all connected sessions

public void broadcastClaimStatus(String status) {

TextMessage message = new TextMessage(status);

sessions.forEach(session -> {

try {

session.sendMessage(message);

} catch (IOException e) {

// Handle exception

}

});

}

}

**3. Configure WebSocket Endpoint**

Configure WebSocket endpoint and message broker in Spring Boot:

**3.1. WebSocket Configuration**

Configure WebSocket endpoint and message broker in WebSocketConfig.java:

java

package com.example.reactiveclaims.config;

import com.example.reactiveclaims.websocket.ClaimStatusWebSocketHandler;

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

import org.springframework.context.annotation.Configuration;

import org.springframework.web.socket.config.annotation.EnableWebSocket;

import org.springframework.web.socket.config.annotation.WebSocketConfigurer;

import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Autowired

private ClaimStatusWebSocketHandler webSocketHandler;

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(webSocketHandler, "/ws/claim-status")

.setAllowedOrigins("\*"); // Allow all origins (you may restrict as needed)

}

}

**4. Use WebSocket in Controller**

Use WebSocket to send real-time updates from your controller:

**4.1. Controller Integration**

Inject the ClaimStatusWebSocketHandler into your controller and use it to broadcast updates:

java

package com.example.reactiveclaims.controller;

import com.example.reactiveclaims.websocket.ClaimStatusWebSocketHandler;

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

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/claims")

public class ClaimController {

@Autowired

private ClaimStatusWebSocketHandler webSocketHandler;

@PutMapping("/{id}/status")

public String updateClaimStatus(@PathVariable Long id, @RequestBody String status) {

webSocketHandler.broadcastClaimStatus("Claim ID " + id + " status updated to " + status);

return "Claim status updated successfully";

}

}

**5. Client-side Integration**

Integrate WebSocket on the client-side (e.g., JavaScript) to receive updates from the server and handle WebSocket events.