**README**

### 前端部分

#### `index.html`

这是一个简单的HTML页面，包含一个输入框和一个按钮用于发送消息，以及一个列表用于显示历史消息。

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Message App</title>

<style>

body { font-family: Arial, sans-serif; }

.container { width: 50%; margin: auto; }

.message-box { margin-bottom: 20px; }

#messageHistory { list-style-type: none; padding: 0; margin: 0; }

#messageHistory li { margin: 5px 0; }

</style>

</head>

<body>

<div class="container">

<h1>Message App</h1>

<div class="message-box">

<input type="text" id="messageContent" placeholder="Enter your message here" required />

<button type="button" onclick="sendMessage()">Send</button>

</div>

<div class="message-box">

<h2>Message History</h2>

<ul id="messageHistory"></ul>

</div>

</div>

<script>

function sendMessage() {

const content = document.getElementById('messageContent').value;

fetch('/api/messages/send', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(content)

}).then(response => {

if (response.ok) {

loadMessageHistory();

document.getElementById('messageContent').value = ''; // Clear input field

} else {

alert('Failed to send message');

}

});

}

function loadMessageHistory() {

fetch('/api/messages/history')

.then(response => response.json())

.then(data => {

const messageHistory = document.getElementById('messageHistory');

messageHistory.innerHTML = ''; // Clear previous messages

data.forEach(message => {

const li = document.createElement('li');

li.textContent = `${new Date(message.timestamp).toLocaleString()}: ${message.content}`;

messageHistory.appendChild(li);

});

});

}

window.onload = loadMessageHistory; // Load history when page loads

</script>

</body>

</html>

```

#### JavaScript逻辑：

1. \*\*`sendMessage`函数\*\*：当用户点击“Send”按钮时，该函数会被调用。它从输入框获取消息内容，然后通过`fetch` API向后端发送一个POST请求，将消息内容发送到`/api/messages/send`端点。如果请求成功，它会调用`loadMessageHistory`函数来刷新消息历史。

2. \*\*`loadMessageHistory`函数\*\*：这个函数在页面加载时和每次发送消息后被调用。它通过`fetch` API从`/api/messages/history`端点获取所有消息的历史记录，并将它们显示在页面上。

3. \*\*`window.onload`事件\*\*：当页面加载完成时，自动调用`loadMessageHistory`函数以加载初始的消息历史。

### 后端部分

#### `MessageController`类

这个控制器类处理前端发来的HTTP请求。

```java

package com.example.messageapp.controller;

import com.example.messageapp.entity.Message;

import com.example.messageapp.repository.MessageRepository;

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

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

import java.util.List;

@RestController

@RequestMapping("/api/messages")

public class MessageController {

@Autowired

private MessageService messageService;

@Autowired

private MessageRepository messageRepository;

@PostMapping("/send")

public void sendMessage(@RequestBody String content) {

messageService.sendMessage(content);

}

@GetMapping("/history")

public List<Message> getHistory() {

return messageRepository.findAll();

}

}

```

#### 逻辑解释：

1. \*\*`sendMessage`方法\*\*：这个方法映射到`/api/messages/send`端点，并使用`@PostMapping`注解表示它是一个POST请求。它接收一个字符串类型的请求体（即消息内容），然后调用`messageService.sendMessage(content)`方法将消息发送到RabbitMQ队列。

2. \*\*`getHistory`方法\*\*：这个方法映射到`/api/messages/history`端点，并使用`@GetMapping`注解表示它是一个GET请求。它返回所有存储在MySQL数据库中的消息历史记录。

### 服务层和实体层

#### `MessageService`类

这个服务类负责处理业务逻辑，包括将消息发送到RabbitMQ队列。

```java

package com.example.messageapp.service;

import com.example.messageapp.entity.Message;

import com.example.messageapp.repository.MessageRepository;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

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

import org.springframework.stereotype.Service;

@Service

public class MessageService {

@Autowired

private RabbitTemplate rabbitTemplate;

@Autowired

private MessageRepository messageRepository;

public void sendMessage(String content) {

// Create a new message entity and save it to the database

Message message = new Message();

message.setContent(content);

message.setTimestamp(new java.sql.Timestamp(System.currentTimeMillis()));

messageRepository.save(message);

// Send the message to RabbitMQ queue

rabbitTemplate.convertAndSend("myQueue", content);

}

}

```

#### 逻辑解释：

1. \*\*`sendMessage`方法\*\*：首先创建一个新的消息实体并将其保存到MySQL数据库中。然后，它将消息内容发送到RabbitMQ队列。

2. \*\*`RabbitTemplate`\*\*：这是一个Spring提供的模板类，用于简化与RabbitMQ的交互。在这里，我们使用它的`convertAndSend`方法将消息发送到名为`myQueue`的队列。

#### `Message`实体类

这个类表示数据库中的消息表。

```java

package com.example.messageapp.entity;

import javax.persistence.\*;

import java.sql.Timestamp;

@Entity

public class Message {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String content;

private Timestamp timestamp;

// Getters and setters omitted for brevity...

}

```

#### 逻辑解释：

1. \*\*字段\*\*：`id`是主键，`content`是消息内容，`timestamp`是消息的时间戳。

2. \*\*JPA注解\*\*：`@Entity`表示这是一个JPA实体类，`@Id`和`@GeneratedValue`用于标识主键及其生成策略。

### 总结：

1. \*\*前端\*\*：用户通过输入框输入消息并点击“Send”按钮，JavaScript会将消息发送到后端的`/api/messages/send`端点。发送成功后，前端会自动刷新消息历史。

2. \*\*后端\*\*：后端接收到消息后，将其保存到MySQL数据库，并通过RabbitMQ发送到相应的队列。同时，前端可以通过访问`/api/messages/history`端点获取最新的消息历史。