## 部署命令

修改.env的url地址到azure

npm run build

复制build目录到main/resources/改名为static

mvn clean install

mvn package azure-webapp:deploy

## 命令

apikey必须硬编码到java代码里，放配置文件里会泄露，被openai置失效

npm i uuid @types/uuid

npm install @mui/material @emotion/react @emotion/styled

为了管理密码，要使用react context了！

npm install jsonwebtoken

npm install @types/jsonwebtoken --save-dev

安装完成后，您可以在项目中导入并使用 jsonwebtoken 库。（加了这两个包会报很多错！！！！！！！！！！！！）

保护路由的实现

安装redux

npm i @reduxjs/toolkit react-redux

## 安装daisyui

How to install daisyUI as a Tailwind CSS plugin?

You need Node.js and Tailwind CSS installed.

Install daisyUI:

npm i daisyui

Then add daisyUI to your tailwind.config.js files:

module.exports = {

//...

plugins: [require("daisyui")],

}

The default theme is light (or dark for dark mode) but you can change the default theme from tailwind.config.js

## 下拉菜单选择prompt方案

### claude方案

我的技术栈是：spring boot, react, typescript, tailwind, daisyUI

我的前端有个下拉菜单，我想实现一个功能，点击期中的选项时，向服务端发送一条信息。给我说一下具体思路。我已经有这个下拉菜单了，菜单项是个<li>

Claude

APP 10:42 AM

可以这么实现:

1. 给菜单选项 <li> 添加点击事件,在事件中发送 AJAX 请求到后端

tsx

<li onClick={() => sendMessage(option.value)}> {option.label} </li>

2. 定义一个 sendMessage 函数发送 AJAX 请求:

ts

function sendMessage(message: string) {

fetch('/message', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ message })

})

}

3. 在后端 Spring Boot 中定义一个 /message 的 POST API 来接收消息:

java

@PostMapping("/message")

public void postMessage(@RequestBody Map<String, String> message) {

String msg = message.get("message");

// 接收到消息,进行处理...

}

4. 这样每当点击下拉菜单选项时,就会通过 AJAX 发送一条消息到 /message API,后端就可以接收到用户选择的菜单项了。

主要思路是:通过为菜单项添加点击事件,在事件中使用 AJAX 技术与后端通信,发送用户的选择消息。后端 API 接收到信息后进行处理。这是前端与后端互动的一种常见模式。

### 我的方案，第一版，最简单

先实现下拉菜单，已经有了，在header组件里

下拉菜单的li上有一个onclick回调，然后header组件里有一个处理回调的函数handleLi

handleLi会把li关联的prompt发送给maivc，同时有标记，清空前面的内容。

maivc收到后，把回复发给header，header再通过全局状态告诉msgList组件。

另一个方案，这个方案不利于把消息功能集中化。

我考虑只把整个prompt通过全局变量传递给chatpage，然后chatpage收到之后，按照正常的聊天流程走

## 路由问题汇总

Error: Full authentication is required to access this resource

我把前端打包后放在服务端工程的src\main\resources\static目录下部署之后，我发现localhost:8080不能访问了，我想可能是因为我jwt保护了所有端口，只留下了前端登录服务端的localhost:8080/login 现在我怎么打开这个static需要的端口？？？访问网站报上面的错误

方案

.antMatchers(HttpMethod.GET, "/").permitAll() .antMatchers("/static/\*\*").permitAll()

我部署成功了，但是我发现我只能访问

https://shellgpt.azurewebsites.net

不能访问

https://shellgpt.azurewebsites.net/signin

也不能访问

https://shellgpt.azurewebsites.net/chatpage

上述两个链接会被跳转到，但是如果我直接在地址栏里输入，则不能访问。知道为什么吗

这个问题可能是由于前端路由的配置导致的。在前端应用中，通常使用客户端路由（如React Router）来实现导航。当用户在浏览器地址栏中直接输入URL时，请求会发送到服务器。然而，服务器可能不知道如何处理这些客户端路由，因此会返回404错误或者跳转到根路径。

为了解决这个问题，您需要确保服务器在遇到未知的路由时始终返回主页（通常是index.html）。这样一来，React应用会加载并处理相应的客户端路由。

在Spring Boot中，您可以通过以下方式配置：

在src/main/resources/static目录下创建一个名为index.html的文件，将React应用的入口HTML代码放入其中。

在您的@SpringBootApplication主类中添加以下方法：

javaCopy code

@Bean

public WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> webServerFactoryCustomizer() {

return factory -> factory.addErrorPages(new ErrorPage(HttpStatus.NOT\_FOUND, "/"));

}

此方法将为404错误配置一个错误页面，当服务器收到未知路由请求时，将返回index.html文件。这将允许React应用加载并处理客户端路由。

添加此代码后，您应该能够直接访问https://shellgpt.azurewebsites.net/signin和https://shellgpt.azurewebsites.net/chatpage链接。

保护路由的最终方案参考代码app.tsx

## 保护路由方案

《《《《《《《《《《《《《《《

我的意思是，当我已经登录成功，保存了jwt。我重新打开网页时，就不希望再登录了。但是我希望每次重新打开网页时，都去服务端验证一下这个Jwt的签名。如果我验证过了这个jwt，我就不需要在客户端的路由跳转中再到服务端验证了，但是在客户端的路由跳转中，我会只在客户端验证这个jwt的存在性和时效性。

我这个流程对不对？

您的流程基本上是正确的。我明白您希望在重新打开网页时自动登录，并且希望在此时验证JWT签名。以下是根据您的需求调整的流程：

1、用户登录成功后，将JWT存储在localStorage中。这样，即使用户关闭浏览器，JWT仍然可用。

2、当用户重新打开网页时，在应用程序启动时检查localStorage中是否存在JWT。如果存在JWT，将其发送给服务端进行验证（签名和过期状态）。服务端验证成功后，允许客户端访问受保护资源。

3、在客户端路由跳转期间，只在客户端验证JWT的存在性和时效性。如果JWT不存在或已过期，引导用户返回到登录页面。

这个流程是可行的，符合您的需求。请注意，在将JWT存储在localStorage中时，要确保实施适当的安全措施，以防止潜在的XSS攻击。

在许多场景下，这种实践是可以接受的，但是请务必确保实施适当的安全措施，如使用Content Security Policy（CSP）来降低XSS风险。如果您的应用程序需要更高的安全性，您可以考虑使用HTTP-only cookies来存储JWT，并使用CSRF tokens来防止跨站请求伪造（CSRF）攻击。

总之，您的流程在非高安全需求的应用程序中是常见的实践，但请确保实施其他安全措施来保护用户数据和会话。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

JWT密钥在服务端用于对JWT签名和验证。通常，密钥应该是一个长期保留的配置值，而不是每次都随机生成。这是因为，如果密钥每次都随机生成，那么在服务重启或部署新版本时，之前发给客户端的所有令牌将会失效，因为新密钥无法验证它们。因此，最佳实践是生成一个安全的密钥并将其保存在服务器的配置文件中，或使用环境变量来存储密钥。

客户端不需要知道这个密钥。事实上，从安全性角度考虑，客户端最好不要知道密钥，因为这可能导致攻击者使用该密钥篡改或伪造令牌。JWT的核心思想是，服务端使用密钥对JWT进行签名，客户端不需要知道密钥。客户端只需要在访问需要身份验证的资源时，将JWT发送给服务端。然后，服务端使用密钥验证JWT的有效性和完整性。

至于客户端如何存储JWT，一般有以下两种方法：

使用localStorage：将JWT存储在客户端的本地存储中。这样，即使在关闭浏览器或刷新页面后，JWT仍然可用。然而，这种方法可能会导致跨站脚本攻击（XSS）的风险，因为攻击者可能会利用脚本访问本地存储并窃取JWT。

使用sessionStorage：将JWT存储在客户端的会话存储中。这样，在关闭浏览器时，JWT将被清除。这种方法在一定程度上降低了XSS风险，但并不能完全消除。

另一种选择是使用HTTP-only cookies存储JWT。这种方法可以有效降低XSS风险，因为HTTP-only cookies不能通过客户端脚本访问。但这可能会导致跨站请求伪造（CSRF）的风险。要解决这个问题，可以结合使用CSRF tokens来提高安全性。总之，在选择存储方法时，请权衡安全性和易用性，并根据您的应用需求做出合适的决策。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

那你重新说一下方案

在 React TypeScript 项目中实现 JWT 验证的最佳实践方案如下：

使用以下命令安装 jsonwebtoken 库及其类型定义：

sql

Copy code

npm install jsonwebtoken

npm install @types/jsonwebtoken --save-dev

创建一个名为 validateJWT 的工具函数。该函数接受 JWT 作为参数，并使用 jsonwebtoken 库解码它。然后，验证 JWT 的有效性，例如检查签名、过期时间等。

创建一个新的组件，例如 ProtectedRoute。这个组件将负责检查 JWT 是否有效，并决定是否允许用户访问特定的路由。

在 ProtectedRoute 组件内部，调用 validateJWT 函数来验证 JWT 的有效性。根据验证结果，ProtectedRoute 组件将决定是否渲染其子组件（即允许用户访问特定路由）或重定向到登录页面。

在您的路由配置中，使用新创建的 ProtectedRoute 组件包装那些需要进行 JWT 验证的路由。例如，将 ChatPage 和 SpeechTestPage 放在 ProtectedRoute 中。

通过这种方法，您将在客户端验证 JWT，这将减少不必要的服务器请求。但是，请注意，客户端验证仅确保 JWT 没有被篡改；您仍然需要在处理敏感数据和操作时在服务器端进行验证。

edwinbiz@hotmail.com

结合我给你的代码， 大致给我一下代码示例吧

根据您提供的代码和要求，以下是一个简化的代码示例：

首先，安装 jsonwebtoken 及其类型定义：

sql

Copy code

npm install jsonwebtoken

npm install @types/jsonwebtoken --save-dev

创建一个 validateJWT.ts 文件，用于验证 JWT：

typescript

Copy code

import jwtDecode from 'jsonwebtoken';

export const validateJWT = (token: string) => {

try {

const decoded = jwtDecode.verify(token, process.env.REACT\_APP\_JWT\_SECRET as string);

const currentTime = Math.floor(Date.now() / 1000);

if (decoded.exp && decoded.exp < currentTime) {

return false;

}

return true;

} catch (error) {

return false;

}

};

创建一个 ProtectedRoute.tsx 组件：

tsx

Copy code

import React from 'react';

import { Route, Redirect, RouteProps } from 'react-router-dom';

import { validateJWT } from './validateJWT';

interface ProtectedRouteProps extends RouteProps {

token: string | null;

}

const ProtectedRoute: React.FC<ProtectedRouteProps> = ({ token, ...routeProps }) => {

if (token && validateJWT(token)) {

return <Route {...routeProps} />;

}

return <Redirect to="/signin" />;

};

export default ProtectedRoute;

修改您的路由配置以使用 ProtectedRoute 组件：

tsx

Copy code

import React from 'react';

import { RouterProvider } from '@uirouter/react';

import { AppProvider } from './AppProvider';

import SignInPage from './SignInPage';

import ChatPage from './ChatPage';

import SpeechTestPage from './SpeechTestPage';

import ProtectedRoute from './ProtectedRoute';

import HeaderBar from './HeaderBar';

const router = createBrowserRouter([

{

path: '/',

element: <Navigate to="signin" />

},

{

path: '/signin',

element: <SignInPage />

},

{

path: '/chatpage',

element: <ProtectedRoute token={localStorage.getItem('jwtToken')} component={ChatPage} />

},

{

path: '/setting',

element: <ProtectedRoute token={localStorage.getItem('jwtToken')} component={SpeechTestPage} />

}

]);

function App() {

return (

<div className="min-w-[240px] min-h-screen">

<AppProvider>

<HeaderBar />

<RouterProvider router={router} />

</AppProvider>

</div>

);

}

export default App;

注意：在这个示例中，我假设 JWT 令牌存储在本地存储中，键名为 'jwtToken'。您可以根据需要修改这个部分。

这个简化的示例应该可以帮助您实现所需的功能。

》》》》》》》》》》》》》》》》

## 服务端本地生成和保存jwt secret key的方法

《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《《

我的服务端是spring boot + spring security + jwt。这是我生成jwt的方法，用的是临时生成的secret。请修改这个代码，让它从配置文件中读取jwt，这个配置文件是application.yaml吗？然后你还要告诉我如何生成这个secret，用什么网站生成，还是其他什么工具。

您可以使用Spring Boot的配置文件application.yaml来存储JWT的密钥。首先，您需要在配置文件中添加密钥：

yaml

Copy code

jwt:

secret: your-base64-encoded-secret-key

将your-base64-encoded-secret-key替换为您的实际Base64编码的密钥。

接下来，您需要在JwtService类中使用@Value注解从配置文件中读取密钥，然后将其转换为Key类型。这里是修改后的JwtService类：

java

Copy code

import io.jsonwebtoken.SignatureAlgorithm;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.security.Keys;

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

import org.springframework.stereotype.Component;

import javax.crypto.spec.SecretKeySpec;

import java.security.Key;

import java.util.Base64;

import java.util.Date;

@Component

public class JwtService {

static final long EXPIRATIONTIME = 86400000;

static final String PREFIX = "Bearer";

@Value("${jwt.secret}")

private String secret;

private Key getSigningKey() {

byte[] decodedKey = Base64.getDecoder().decode(secret);

return new SecretKeySpec(decodedKey, 0, decodedKey.length, "HmacSHA256");

}

public String getToken(String username) {

String token = Jwts.builder()

.setSubject(username)

.setExpiration(new Date(System.currentTimeMillis() + EXPIRATIONTIME))

.signWith(getSigningKey())

.compact();

return token;

}

}

要生成Base64编码的密钥，您可以使用Java的SecretKey和Base64类：

java

Copy code

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import java.security.NoSuchAlgorithmException;

import java.util.Base64;

// 这个文件放在spring工程的test目录下，可以直接运行

public class JwtKeyGenerator {

public static void main(String[] args) throws NoSuchAlgorithmException {

KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA256");

SecretKey secretKey = keyGenerator.generateKey();

String encodedKey = Base64.getEncoder().encodeToString(secretKey.getEncoded());

System.out.println("Base64 Encoded Secret Key: " + encodedKey);

}

}

运行这个JwtKeyGenerator类，将输出一个Base64编码的密钥。将其复制并替换application.yaml中的your-base64-encoded-secret-key。

请注意，生成的密钥应该保密，并且在生产环境中需要保护好。不要将其上传到公共代码仓库。

》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》

## 解决postman有返回，网页没返回的问题

《《《《《《《《《《《《《《《《

我选择了JSONObject方案。但是还有问题，我能收到body里的jwt，但是headers里却没有Authorization字段。但是postman调用时，body和header都有。这是怎么回事

这可能是由于浏览器的 CORS 策略所导致的。在跨域请求的情况下，浏览器对哪些 header 可以暴露给客户端有限制。要让 Authorization header 在跨域请求时对客户端可见，您需要在服务器端明确允许暴露这个 header。在您的 Spring Boot 应用中，您需要更新 CorsConfiguration 配置。

请按照以下步骤更新您的 SecurityConfig 类：

在 corsConfigurationSource() 方法中，更新 CorsConfiguration 对象以允许暴露 Authorization header。在调用 config.applyPermitDefaultValues() 之前，使用 config.setExposedHeaders() 方法添加 Authorization 到允许暴露的 header 列表中。

java

Copy code

@Bean

CorsConfigurationSource corsConfigurationSource() {

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

CorsConfiguration config = new CorsConfiguration();

// 将这行替换为允许指定的域名

config.setAllowedOrigins(Arrays.asList("http://localhost:3000"));

config.setAllowedMethods(Arrays.asList("\*"));

config.setAllowedHeaders(Arrays.asList("\*"));

config.setAllowCredentials(true);

// 添加 Authorization 到允许暴露的 header 列表中

config.setExposedHeaders(Arrays.asList("Authorization"));

config.applyPermitDefaultValues();

source.registerCorsConfiguration("/\*\*", config);

return source;

}

另外，将 config.setAllowCredentials() 设置为 true。这样客户端才能在跨域请求中接收和发送凭证（如 cookies 和 Authorization header）。

现在，您的浏览器应该能够访问到 Authorization header。请注意，当调试跨域问题时，确保您的客户端和服务器端代码已更新，并且浏览器缓存已清除。这样可以避免因缓存导致的问题。

————————————————————————

当发送跨域请求时，服务器可能返回一个带有 CORS 头部的响应，这表明浏览器可以读取响应内容。在这种情况下，响应的类型为 cors。

当您发送 HTTP 请求时，如果服务器支持 CORS，并且在响应中包含了适当的 CORS 头部，那么得到一个 cors 类型的响应是正常的。这意味着您的请求已经成功发送到服务器，并且浏览器允许您读取响应内容。

我注意到您在读取响应头 Authorization 时，可能在错误的地方执行了此操作。您应该在 response.json() 之前读取响应头。这是因为一旦您调用 response.json()，响应对象将会被标记为已使用，之后您可能无法再从中获取数据。请尝试调整代码顺序

login Response2: Response {type: 'cors', url: 'http://localhost:8080/login', redirected: false, status: 200, ok: true, …}

VM16:1 Uncaught (in promise) SyntaxError: Unexpected end of JSON input

at login (Login.tsx:38:1)

l

报错如上所示

从您提供的错误信息来看，问题出在尝试解析 JSON 时。错误提示 "Unexpected end of JSON input" 表明响应体可能为空，或者不是有效的 JSON 格式。

要解决这个问题，请首先检查您的后端服务器是否返回了一个有效的 JSON 对象。您可以通过在服务器端代码中打印响应内容来检查。

在客户端，您可以修改代码，以便在尝试解析 JSON 之前检查响应是否包含内容。您可以通过检查 response.headers.get('Content-Type') 是否包含 "application/json" 来实现这一点。

if (response.headers.get('Content-Type')?.includes('application/json')) {

const responseBody = (await response.json()) as any;

console.log('Response body:', responseBody);

} else {

console.error('Response does not contain JSON');

}

从您提供的控制台输出中，我们可以看到 Response does not contain JSON，这意味着响应头 Content-Type 并未包含 "application/json"。这可能是服务器没有返回有效的 JSON 对象，或者响应头设置有误。

根据之前的讨论，您在使用 Postman 时得到了正确的结果，但在使用网页请求时没有得到预期结果。因此，我们需要首先确定问题出在哪里。

首先，请确保在后端服务器的响应中设置了正确的 Content-Type，如下所示：

java

Copy code

response.setContentType("application/json");

然后，请确保您的服务器在处理请求时正确地返回了 JSON 对象。您可以通过在服务器端代码中打印响应内容来检查。

return ResponseEntity.ok()

.header(HttpHeaders.AUTHORIZATION, "Bearer " + jwtToken)

.header(HttpHeaders.ACCESS\_CONTROL\_ALLOW\_HEADERS, "Authorization")

.build();

}

这是服务端代码

感谢您提供了服务端代码。根据您的代码，您已经设置了正确的响应头，包括 Authorization 和 Access-Control-Allow-Headers。然而，这里的问题是响应中没有包含任何 JSON 内容。这就是为什么您在浏览器的控制台中看到“Response does not contain JSON”的错误。

要解决这个问题，您需要在响应中包含 JSON 内容。一种方法是返回一个包含 JWT 令牌的 JSON 对象，如下所示：

// Create a JSON object with the token

JSONObject json = new JSONObject();

json.put("token", "Bearer " + jwtToken);

// build response with generated token and JSON content

return ResponseEntity.ok()

.header(HttpHeaders.AUTHORIZATION, "Bearer " + jwtToken)

.header(HttpHeaders.ACCESS\_CONTROL\_ALLOW\_HEADERS, "Authorization")

.contentType(MediaType.APPLICATION\_JSON)

.body(json.toString());

}

注意，这里我们创建了一个名为 json 的 JSONObject 对象，并添加了 JWT 令牌。我们还设置了 Content-Type 为 application/json，并将 json 对象转换为字符串作为响应的正文。

然后，在客户端，您可以继续使用 response.json() 方法来解析 JSON 内容。现在，您应该能够正确地获取 JWT 令牌并处理响应。

》》》》》》》》》》》》》》》》

## 如何实现会话功能

《《《《《《《《《《《《《《《《

我在用spring boot和react.ts写一个和gpt api聊天的软件。我想让每个用户每打开一次会话（网页上访问我的产品的url），都单独保存这个会话的聊天记录到后端的postgresql里。如果用户把网页关掉了，下一次就是新的会话，不再使用上次的聊天记录。告诉我前端和后端的综合实现方案。

这是一个涉及前端和后端技术的问题。首先，我们需要在前端创建一个唯一的会话ID，然后在后端将每个会话的聊天记录保存在PostgreSQL数据库中。以下是一种实现方案：

前端（React.ts）：

当用户访问您的产品URL时，为每个会话生成一个唯一的会话ID。您可以使用UUID库来生成这个ID。

javascript

Copy code

import { v4 as uuidv4 } from 'uuid';

const sessionId = uuidv4();

当用户与GPT API聊天时，将输入和输出消息及其对应的会话ID发送给后端。您可以使用axios库将数据发送到后端。

javascript

Copy code

import axios from 'axios';

async function sendMessage(message) {

const response = await axios.post('/api/chat', {

sessionId: sessionId,

message: message

});

return response.data;

}

后端（Spring Boot）：

在Spring Boot应用程序中，创建一个ChatMessage实体类，用于存储聊天消息和会话ID。

java

Copy code

@Entity

public class ChatMessage {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String sessionId;

private String message;

private boolean isUserMessage;

// ... getters and setters

}

创建一个ChatMessageRepository接口，继承自JpaRepository，用于操作数据库。

java

Copy code

public interface ChatMessageRepository extends JpaRepository<ChatMessage, Long> {

List<ChatMessage> findBySessionId(String sessionId);

}

创建一个ChatController类，处理前端发来的请求。

java

Copy code

@RestController

@RequestMapping("/api/chat")

public class ChatController {

@Autowired

private ChatMessageRepository chatMessageRepository;

@PostMapping

public ChatMessage saveChatMessage(@RequestBody ChatMessage chatMessage) {

// 保存聊天消息到数据库

return chatMessageRepository.save(chatMessage);

}

}

在application.properties中配置PostgreSQL数据库。

properties

Copy code

spring.datasource.url=jdbc:postgresql://localhost:5432/your\_database\_name

spring.datasource.username=your\_username

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

使用此实现方案，每个用户的会话都将拥有唯一的ID，并且每次打开URL时都将创建新的会话。聊天记录将保存在PostgreSQL数据库中，每次会话的聊天记录都是独立的。当用户关闭网页并重新打开时，将不再使用之前的聊天记录。

》》》》》》》》》》》》》》》》