# 2.项目目标

该项目的核心目标是建立和实施一个智能问答系统，旨在解决华南理工大学（SCUT）学生和家长在校园生活中遇到的各种 问题，帮助用户快速获取针对性的解决方案，增强他们在校园生活中的体验和满意度。具体而言，项目目标包括但不限于： 1.提供涵盖华南理工大学学生和家长在校园生活中可能遇到的各种问题的智能问答系统，包括学习指导、入学手续、宿舍安 排、选课、校园设施和活动信息等。 2.通过建立学校信息的向量数据库和对大语言模型进行微调等技术，实现高效的问答功能，确保系统能够理解用户的查询并提供准确、全面的答案，以及个性化的解决方案。 3.设计用户友好的界面，便于与系统进行交互。我们在前端页面添加了数字人，并且大语言模型的回复可以通过数字人说出。同时，数字人的外形和声音也是可 定制的。 4.根据用户反馈和需求持续改进和优化系统，不断提升其功能和性能，确保它能持续满足用户的实际需求并保持领先地位。 总之，该项目的核心目标是建立一个智能问答系统，为华南理工大学学生和家长提供全面和个性化的信息服务， 解决校园生活中遇到的各种问题和查询，从而提升校园生活的便利性和舒适度。

# 3.5.前端模块（zbw)

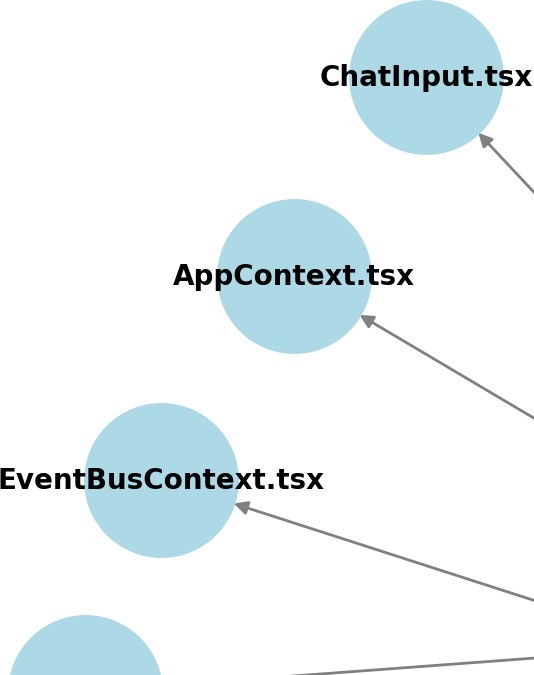
图：具体技术方案 前端模块主要使用了现代的前端框架react和next.js利用React的组件化设计和状态管理实现动态、响应式的用户界面，通过Next.js 的服务器端渲染（SSR）和静态生成（SSG）提升页面加载速度和SEO性能。 前端的具体成果如下：

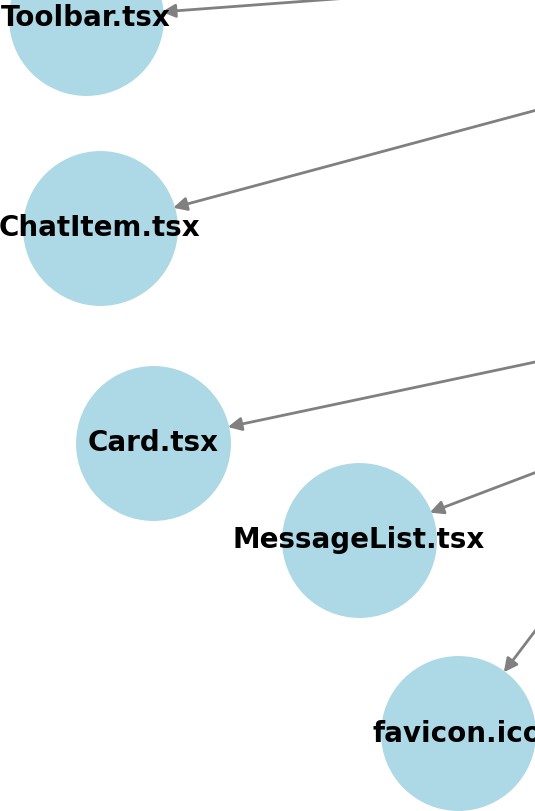




图：前端具体成果 其主要分为5个模块：登录模块、左边的聊天列表模块，中间的消息显示模块与消息输入模块，以及右边的数字人显示模块。下面将分别对这五个模块进行描述。

## 总体架构描述







图：前端项目框架 这是项目的总体架构图，react的基本思想是component is everything，所以在本项目中基本上所有的功能实现都是以component为基础的。整个前端就是由Navigation，Main,Metahuman以及Login四大component构成 的，而这四大component又是由其他小的组件构成的。 由react的基本思想出发，可以得到react项目的最优构建方法，即先将component中的静态网页部分写好，再写各个组件之间的信息传递逻辑。而在信息传递中最重要的就是各类信息数据 结构的设计，接下来介绍用于信息传递的数据结构。

export interface Chat { chatId: string description: string time:number

}

export interface ChatDTO { chatid: string

description: string time:string user:string

}

export interface Message { id: string

role: "user" | "assistant" message: string chatid:string

time: string

}

export interface MessageRequestBody { messages: Message[]

model: string

}

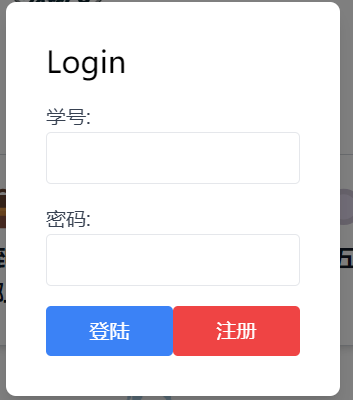
export type State = { user:string displayNavigation: boolean themeMode: "dark" | "light" currentModel: string messageList: Message[] streamingId: string selectedChat?:Chat message:string chatList:Chat[]

}

这五类数据结构就囊括了整个前端项目中大部分的数据传递。 Chat是聊天列表中一个聊天的信息，这里只保存了聊天的id 以及聊天的标题，并没有保存所有聊天的记录。这是因为如果把聊天记录全部保存下来会消耗大量的内存，同时也会让数据 库的查询更频繁。此处的设计是在空间和时间中进行平衡，只在本地存储聊天列表，如果要访问具体的聊天记录，就直接向 后端发请求进行访问。 ChatDTO是一个专门用于向后端传输消息的数据结构，当前端想向后端传输一个Chat时，必须先将

他转化为ChatDTO才能传输。 Message是聊天中的消息，其储存了聊天中的一条消息，包括消息的内容以及具体消息的主体是谁发送的，同时还有消息的发送时间。 MessageRequestBody是一个专门向后端传输消息列表的数据结构。 State是整个前端的全局信息，其包括了当前使用的用户，当前聊天的所有消息，当前的聊天，当前要发送的消息以及当前的所有聊 天。通过对State进行操控就可以实现前端的各种功能。

## 登录模块

 图：登录功能展示 登录功能的具体展示如上图所示，用户输入学号和密码，点击登录按钮就会进行登录检测，点击注册按钮就会进行注册。

async function check(){

const body = JSON.stringify({ "id":username, "password":password,

})

const response = await fetch("http://10.40.20.16:8080/login",

{

method:"POST", headers:{

"Content-Type":"application/json"

},

body:body,

}

)

const responseData = await response.json(); if (responseData.success){

onLogin(); dispatch({type:ActionType.Login,value:username})

}

else{

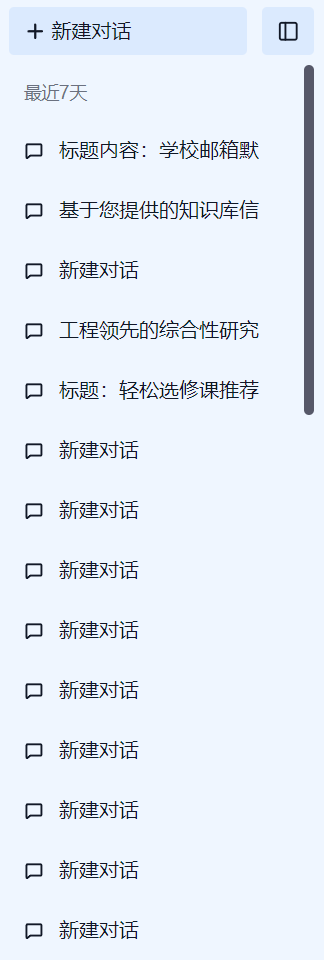
alert("Login failed")

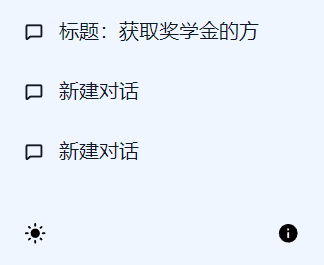
}

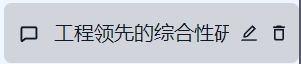
}

登录功能的实现比较简单，具体的代码如上。其逻辑就是将用户输入的代码传输给后端的login接口，后端将用户名密码与 数据库中的做比对，再将结果返回给前端。注册功能就是调用另一个regist接口，先检查数据库中是否有相同的账户，如果 没有就将用户密码对加入数据库中。

## 聊天列表模块



图：聊天列表展示 聊天列表模块主要用于选择聊天以及一些其他的功能。点击具体的某个聊天就可以进入到这个聊天中，此时消息显示模块就会显示出此聊天的消息。点击最上面的新建对

话即可以从当前的聊天中退出并进入初始状态。  图：聊天列表展示 当点击了这个聊天后，最右边会出现修改标题和删除聊天两个按钮，点击这两个按钮即可实现对应的功能。

async function getChatlist(){ if(user=="") return

const response = await fetch(`http://10.40.20.16:8080/chat/${user}`,{ method:"POST",

headers:{

"Content-Type":"application/json"

},

}

)

const responseData = await response.json(); const list = (responseData["data"]) as Chat[]; console.log(list)

setChatList(list)

}

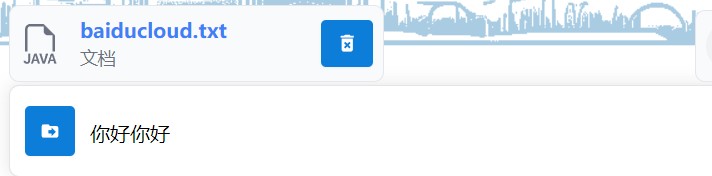
useEffect(()=>{

getChatlist(); console.log(chatList)

},[user,selectedChat])

首先介绍的是获取聊天列表的实现，实际上就是调用相应的接口进行信息获取。更为重要的是什么时候去调用这个函数，显 而易见的是当用户成功登录的时候应该调用，同时当当前选择的聊天变化的时候应该调用，这样设计的原因是我们希望所有 的数据处理都停留在后端，前端只需要进行数据的发送和接受，所以当用户新建一个数据库没有的对话时，前端会做的是将 这个对话发送给后端并让后端保存。此时就需要前端去再次访问数据库，得到最新的聊天列表。而这个判断条件就是当前的 对话改变了。 改变聊天的标题以及删除聊天也基本上是这个逻辑，就不过多赘述了。 新建对话的实现就是将当前选择的对话设置为空，这样消息显示模块就不会显示对话而是显示初始页面。

## 消息输入模块



图：消息输入模块 用户可以通过打字输入文本消息，可以通过点击文件上传按钮或者将文件拖入输入框来上传文件，也可以点击录音按钮来语音上传。

文字输入模块比较简单，就是直接将文字框里的文字对应到变量中。 文件输入也较为简单，html中的input标签中自带了文件的输入，也就是说可以直接上传文件，typescript中也自带了file这个数据类型，直接上传保存到变量中。最后在发送的 时候将文件保存在formdata中。FormData 是一种用于构建和格式化要发送到服务器的数据的对象，通常用于通过 HTML 表单提交数据时。它允许你轻松地通过 JavaScript 操作和发送表单数据，而不需要刷新页面。FormData 对象可以用于XMLHttpRequest 和 Fetch API，以便通过 POST 请求发送键值对数据。 语音输入调用了javascript中自带的mediaRecoder库，他会生成一个临时的文件，传输时传输这个文件即可。

async function send(content:string) { await changeToNewChat()

const responseData = await vsend() const audio = responseData.audio || ""

const message: Message = { id: uuidv4(),

role: "user",

message: content + audio, chatid:chatIdRef.current, time:getCurrentTimeInUTC8()

}

await send2sql(message)

dispatch({ type: ActionType.ADD\_MESSAGE, message }) const messages = messageList.concat([message]) await doSend(messages)

if (!selectedChat?.description || selectedChat.description === "新建对话") { console.log(messages.length)

if(messages.length>2){ updateChatTitle(message)

}

}

}

当用户输入完消息后，前端会对其做一定的处理并将其发送给后端以及大模型端。首先前端会检测当前的聊天是否有改变， 然后会将文件以及语音发送给大模型端，大模型端将这两个转化为文字返回给前端，前端将返回的文字与用户输入的文字叠 加到一起，再发送给大模型得到回复。当完成了以上所有流程后，前端将文字发送给标题生成接口生成标题。

## 消息显示模块



图：消息显示模块 在发送消息后，用户输入的消息会显示在屏幕上，同时当大模型推理时会将输出以流式传输的形式传输给前端，前端就可以以流式输出的形式显示在屏幕上。用户可以点击右边的播放按钮来让数字人读这段文字。

<ul>

{messageList.map((message) => {

const isUser = message.role === "user" return (

<li

key={message.id} className={`${

isUser

? "bg-blue-50 dark:bg-gray-800"

: "bg-gray-50 dark:bg-gray-800"

}`}

>

<div className='w-full max-w-4xl mx-auto flex space-x-6 px-4 py-6 text-lg'>

<div className='text-3xl leading-[1]'>

{isUser ? "" : <div className="my-icon"/>}

</div>

<div className='flex-1'>

<Markdown>{`${message.message}${ message.id === streamingId ? "▍" : ""

}`}</Markdown>

</div>

<div>

{!isUser &&

<Button icon={FaVolumeUp}

onClick={()=>send(message.message)}

>

</Button>}

</div>

</div>

</li>

)

})}

</ul>

此处的逻辑就是将全局变量messageList中的信息全部显示出来 ，并通过message里面的role来确定消息的来源。此处并没有直接的实现流式输出的功能。流式输出是间接实现的，实际上，大模型返回的消息是一个词一个词返回的，每返回一个 词，就更改messageList最后一个元素的值，此时显示出来就会多出一个词，就实现了流式输出。但这里是与我们最初的设 想是相悖的，我们其实更希望前端不进行数据的处理，而是留到后端去做。但这样就会增加很多与后端的交互，现在最终的 逻辑也是一种耦合性和复杂性的妥协。

## 数字人显示模块

 图：数字人显示模块 用户可以通过点击播放按钮让数字人说话。

React.useEffect(() => {

// Make sure Video.js player is only initialized once if (!playerRef.current) {

// The Video.js player needs to be \_inside\_ the component el for React 18 Strict Mode. const videoElement = document.createElement("video-js");

videoElement.classList.add('vjs-big-play-centered'); videoRef.current.appendChild(videoElement);

const player = playerRef.current = videojs(videoElement, options, () => { videojs.log('player is ready');

onReady && onReady(player);

});

// You could update an existing player in the `else` block here

// on prop change, for example:

} else {

const player = playerRef.current;

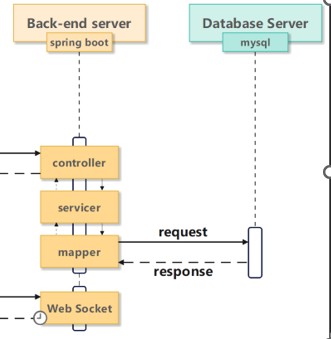
player.autoplay(options.autoplay); player.src(options.sources);

}

}, [options, videoRef]);

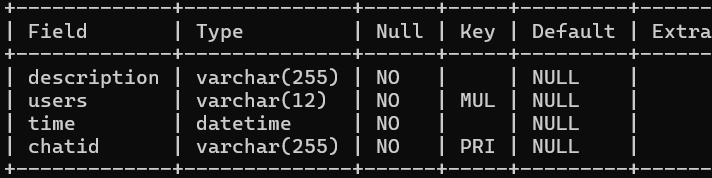
使用 **React.useEffect** 初始化或更新 Video.js 播放器：在初次渲染时创建和配置播放器元素并附加到 DOM 中，随后根据**options** 更新播放器的设置。如果组件卸载，则销毁播放器以防止内存泄漏。函数依赖于 **options** 和 **videoRef** 的变化来重新执行。

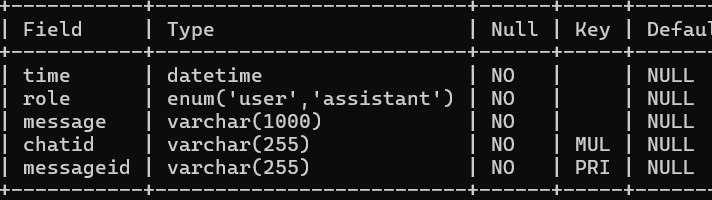
# 3.6 后端模块

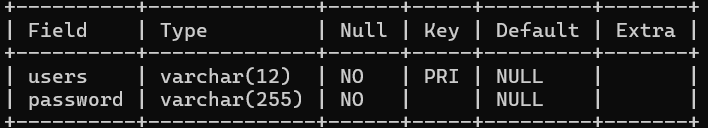
 图：后端的具体架构图 后端主要采用了springboot和mysql的技术路线。主要就是给前端提供对数据库增删查改的操作。 这个项目结构采用了典型的分层架构，每个文件夹都有明确的职责分离：controller层包含控制器类，负责处理客户端的HTTP请求，调用相应的服务方法，并返回响应。控制器是MVC

（Model-View-Controller）模式中的C部分，通常通过注解（如@RestController）定义，并使用URL映射（如@RequestMapping）来处理不同的请求路径。converter层包含转换器类，负责在不同的数据表示形式之间进行转换，例 如将数据库实体对象（Entity）转换为数据传输对象（DTO），或者将输入的JSON数据转换为内部使用的对象。转换器有 助于保持各层之间的解耦，使代码更具模块化和可维护性。dao层包含数据访问对象类，负责与数据库进行交互，DAO层通 过接口和实现类定义数据库操作方法，如查询、插入、更新和删除（CRUD操作），通常使用JPA（Java Persistence API） 或MyBatis等ORM框架实现，提供了对数据库操作的抽象层。dto层包含数据传输对象类，用于在不同层之间传递数据， DTO是简单的Java对象（POJO），包含数据字段和getter/setter方法，不包含业务逻辑，DTO有助于在层之间传递数据而 不暴露数据库实体的内部结构。service层包含服务类，封装业务逻辑，服务层通过调用DAO层的操作方法来处理业务需

求，并在必要时进行事务管理。服务类通常通过接口和实现类定义，以实现松耦合和便于测试。服务层是应用程序的核心， 负责协调数据的处理和业务规则的执行。这种分层结构使得项目的职责分离明确，各层之间的依赖关系清晰，便于团队协作 开发、维护和扩展项目。

在数据库方面设计了三张表来存储信息，分别为chat,message,users。

图：chat表 chat表主要包括了描述，用户，时间以及id四个属性，其中id是主键，users是外键，当users被删除后，相应的chat也会被删除。

图：message表 message表主要包括了时间，角色，信息，聊天id以及id，其中id为主键，聊天id为外键。

图：users表 users表主要包括了用户和密码两个元素，其中用户为主键。