三、系统重要功能实现

3.1 用户鉴权

我们参照现代网页前端设计，实现了token鉴权机制。当用户尝试进行登录时，服务端会鉴别用户身份并返回一个token。客户端会将这个token保存在cookie中。在登录成功后用户每次向服务端发出请求时都将附带上这个token。服务端可以依据校验该token，判断用户是否成功登录，并进一步获取用户的具体身份。用户鉴权的主要实现逻辑位于`frontend/src/utils/request.js`。

```
// 用户登录成功后，调用该函数保存用户认证信息：保存用户token，以及设置token过期时间
function setAuthorization(auth) {
      Cookie.set(xsrfHeaderName, 'Bearer ' + auth.token, {expires:
auth.expireAt})
}

// 用户退出登录后，调用该函数移除用户认证信息
function removeAuthorization() {
      Cookie.remove(xsrfHeaderName)
}
```

3.2 路由守卫

为了防止未获取权限的用户直接通过网址跳转到需要对应权限的页面，我们设计了全局路由守卫。即在用户进入每个页面之前都对用户的认证信息进行检测：如果检测通过，则跳转到对应页面；如果检测不通过，则跳转到错误处理页面。路由守卫主要处理逻辑位于`frontend/src/router/guards.js`中。

我们会在页面路由设置文件(`frontend/src/touter/config.js`)中规定进入相应页面所需的权限，如下所示：

```
{
  path: 'announcementPage',    // 这是公告界面，所有登录用户均可查看
  name: '公告',
  meta: {
    icon: 'dashboard',
    page: {
      closable: false
    },
  },
  component: () => import('@/pages/announcementPage'),
},
{
  path: 'jobSeekPage', // 这是针对教师的求职页面
  name: "广场",
  meta: {
    icon: "team",
    authority: {
      role: 'teacher',      // 设置该页面只有role为'teacher'的用户才能进入
    }
  },
  component: () => import('@/pages/jobSeekPage')
},
{
  path : "hirePage", // 这是针对学生的招聘页面
```

```
  name: "广场",
  meta: {
    icon: "team",
    authority: {
      role: 'student'      // 设置该页面只有role为'student'的用户才能进入
    }
  },
  component: () => import('@/pages/hirePage')
}
```

当用户尝试进入某个页面时，首先会经过登录守卫的检测

```
const loginGuard = (to, from, next, options) => {
  const {message} = options
  if (!loginIgnore.includes(to) && !checkAuthorization()) {
    message.warning('登录已失效，请重新登录')   // 鉴权失败，重定向到登录界面
    next({path: '/login'})
  } else {
    next()    // 鉴权成功，跳转到目标界面
  }
}
```

用户通过登录守卫验证，表示用户已经成功登录。之后用户还需要经过权限守卫的检测

```
const authorityGuard = (to, from, next, options) => {
  const {store, message} = options
  const roles = store.getters['account/roles']
  if (!hasAuthority(to, roles)) {
    message.warning(`对不起，您无权访问页面: ${to.fullPath}，请联系管理员`)  // 鉴权失
败，跳转到错误处理页面
    next({path: '/403'})
    // NProgress.done()
  } else {
    next()  // 鉴权成功，跳转到目标页面
  }
}

// 检测用户是否具有role权限
function hasAuthority(route, roles) {
  const authorities = [...route.meta.pAuthorities, route.meta.authority]
  for (let authority of authorities) {
    if (!hasRole(authority, roles)) {
      return false
    }
  }
  return true
}

// 检测用户的roles列表中是否含有页面所需的role权限
function hasRole(authority, roles) {
  let required = undefined
  if (typeof authority === 'object') {
    required = authority.role
  }
```

```
  return hasAnyItem(required, roles, (r, t) => !!(r === t || r === t.id))
}
```

用户通过权限守卫，表示用户拥有目标页面所需的权限。

用户通过登录守卫与权限守卫后，即可进入目标页面。

3.3 动态路由匹配

我们有时需要把某种模式匹配到的所有路由全都映射到同样的组件中。例如**学生主页**页面，**家教主页**页面，**帖子详情**页面，这时我们可以采用动态路由匹配：一组页面的组件设计相同，客户端通过具体指定的参数来决定具体跳转的目标页面。如下所示：

```
{
    path: 'user/studentHomePage/:id',
    name: '学生主页',
    meta: {
    invisible: true
    },
    component: () => import('@/pages/studentHomePage')
}
```

在“学生主页”页面中，我们使用了动态路由参数，该页面的具体路径将由`$route.params.id`决定。

当我们需要跳转到`id`为`22373090`的页面时，可以这么编写代码

```
this.$router.push({ name: '学生主页', params: { id: "22373090" } })
```

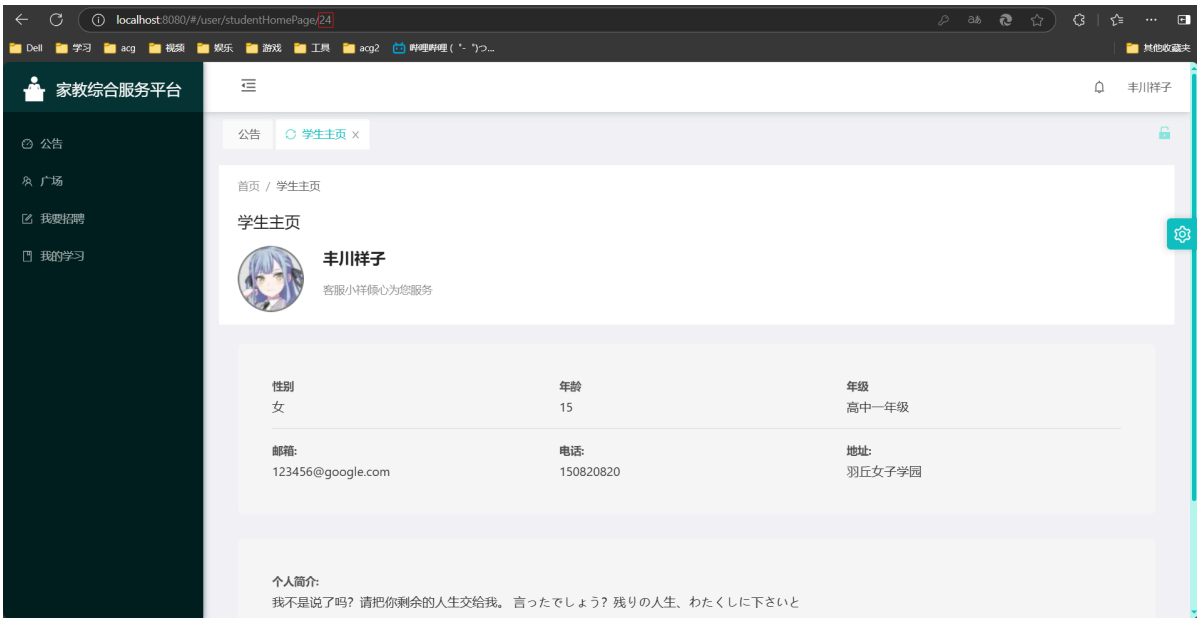
动态路由参数通过`route`直接获取

```
userId: this.$route.params.id // 获取路由参数中的id
```

页面的具体信息可以通过`api`从服务端调取

```
created() {    // 创建“学生主页”页面时，调用fetchUserInfo获取用户信息
    this.fetchUserInfo();
},
methods: {
    fetchUserInfo() {
        // 调用 getUserInfo api，获取用户信息
        getStudentInfo(this.userId)
            .then(response => {
                // ...
            })
            .catch(error => {
                // ...
            });
    }
}
```

具体实现效果如下



3.4 文件上传

在我们使用的Vue2中，文件需要包装在`FormData`对象的`file`段中，才能与后端进行通信。如果此外还有其它数据，可以以`json`的形式包装在`FormData`对象的`data`段中，如下所示

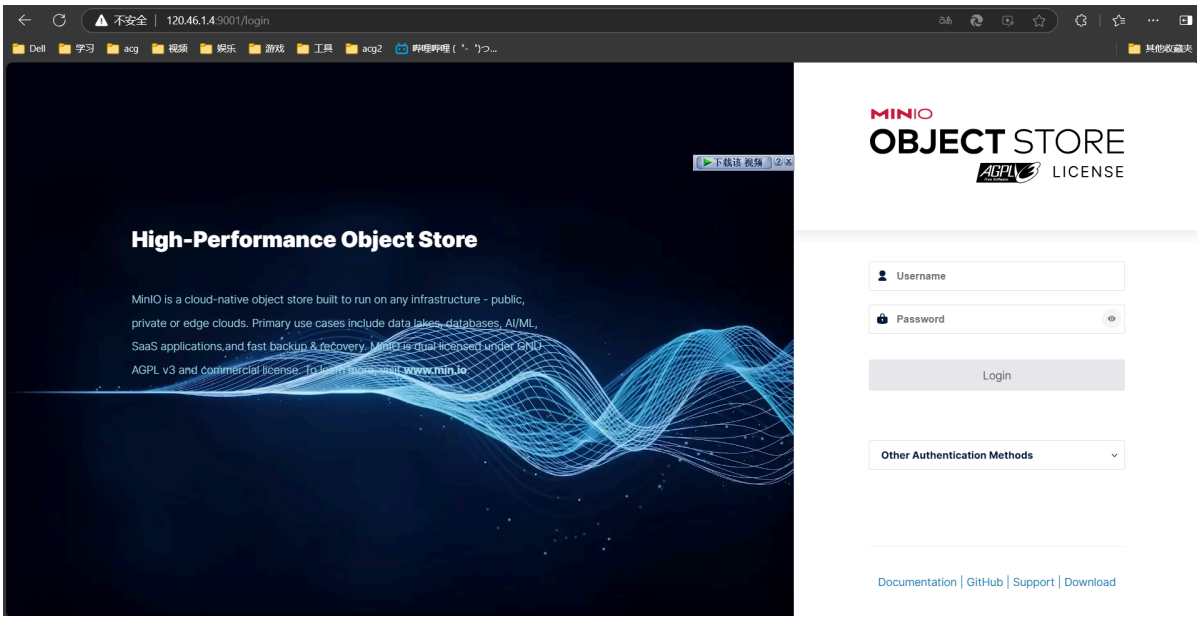
```
submitLearningMaterial() {
    const formData = new FormData();
    formData.append('file', this.fileList[0]);    // 将文件添加到formData的file段
    const jsonData = {
        id: this.currUser.id,
        teacher: this.currUser.name,
        studentId: this.curStudentId,
    };
    formData.append('data', JSON.stringify(jsonData));  // 将其他数据以json形式添加到data段中

    submitLearningMaterial(formData).then(res => {    // 通过api与后端进行通信
        // ...
    }).catch(error => {
        // ...
    });
}
```

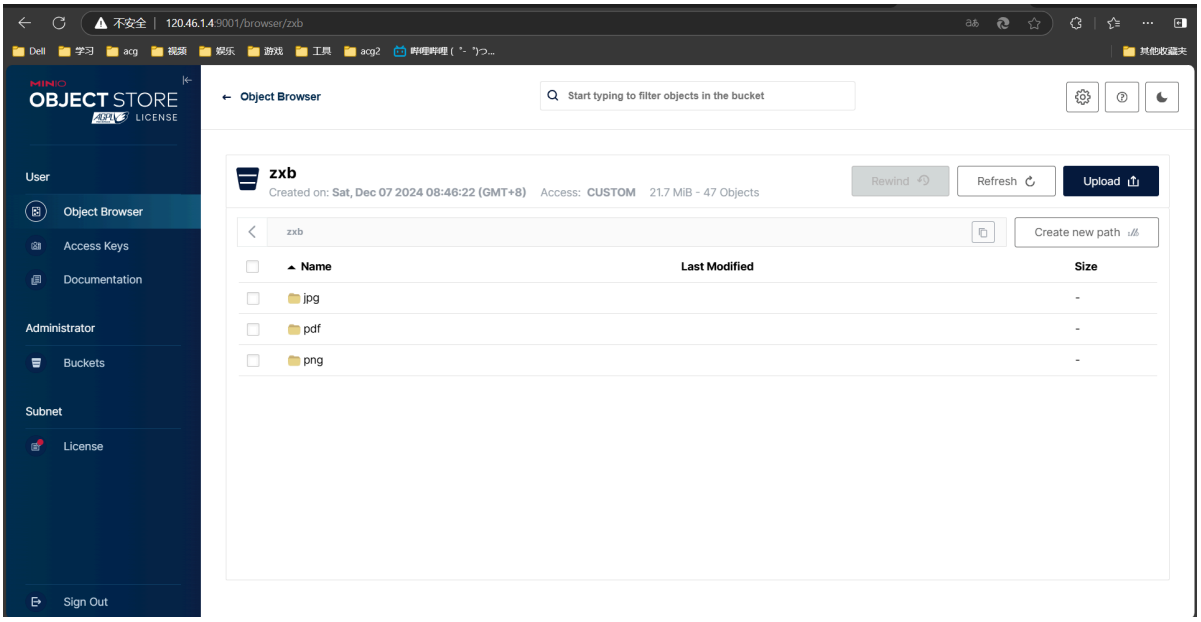
3.5 文件下载

为了实现对用户上传的头像/文件进行云端存储，我们使用了MinIO进行对象存储。

MinIO是一个对象存储解决方案，它提供了与Amazon Web Services S3兼容的API，并支持所有核心S3功能。使用文档参照[MinIO Windows中文文档]

我们从华为云租借了服务器（120.46.1.4），并将MinIO部署在该服务器上。实现效果如下





我们还对上传文件以及下载文件的接口进行了包装，使其使用起来更加方便

```
from minio import Minio

minio_url = "120.46.1.4:9000"
minio_access_key = "****"
minio_secret_key = "****"

def get_download_url(file_name, type) -> str:
    """
    获取文件名为file_name文件的下载链接
    @param file_name: 文件名称
    @param type: 文件类型
    """
    return "http://" + minio_url + "/zxb/" + type + "/" + file_name


class MinioClient:
    def __init__(self):
        self.client = Minio(endpoint=minio_url, access_key=minio_access_key,
secret_key=minio_secret_key, secure=False)
```

```
    def upload_file(self, data, file_name, type)
    """
    上传文件
    @param data: 文件的二进制流数据
    @param file_name：文件存储在云端的名称
    @param type: 文件类型
    """
```

我们可以调用`upload_file`上传文件，调用`get_download_url`获取文件的下载链接，使用示例如下

```
def submitLearningMaterial(request):
    # ...
    file = request.FILES['file']  # 获取上传的文件
    file_name = file.name
    file_type = file_name.split('.')[-1]

    unique_file_name = f"{uuid.uuid4()}.{file_type}" # 使用uuid生成唯一文件名，避免文件名重复发生覆盖

    minio = MinioClient()
    file_data = file.read() # 获取文件的二进制数据
    minio.upload_file(file_data, unique_file_name, file_type) # 上传文件
    download_link = get_download_url(unique_file_name, file_type) # 获取下载链接

    # ...
```

我们通过这种设计，成功实现了”**学习资料**“以及”**用户头像**“相关功能。