全栈项目？那你说说你的token怎么实现的吧

前言

经常我们会被面试官问，账号密码被修改了怎么办，这就涉及到 `token` 了。假设我在淘宝网站登录了账号进入首页，然后我在另一个设备上同样登录该账号，并且将密码修改，原设备刷新数据会被退出登录，这个过程的实现就需要借助 `token` 。 `token` 翻译过来是令牌的意思

`token` 的实现一定是需要你有登录页面，其他页面，以及后端，本文有点长，因为是手把手教，小白可以放心食用

准备工作

前端简单写两个页面，一个登录页，一个首页

前端

```
less
复制代码
npm create vite@latest client -- --template vuecd clientnpm i npm i vue-router@4 npm i vant  // 安装vantnpm i axios
```

路由

```
javascript
复制代码
import { createWebHistory, createRouter } from 'vue-router'const routes = [{        path: '/login',        component: () => import('../views/Login.vue')  },    {        path: '/home',        component: () => import('../views/Home.vue')     }]const router = createRouter({    history: createWebHistory(),    routes: routes})export default router
```

登录页

借助 `vantUI` ，用他的按钮和输入框

自行引入样式~

```ini
复制代码
<template>    <div>        <van-form @submit="onSubmit"            <van-cell-group inset>                <van-field v-model="username" name="username" label="用户名" placeholder="用户名"                    :rules="[{ required: true, message: '请填写用户名' }]" />                <van-field v-model="password" type="password" name="password" label="密码" placeholder="密码"                    :rules="[{ required: true, message: '请填写密码' }]" />            </van-cell-group>            <div style="margin: 16px;"                <van-button round block type="primary" native-type="submit"                    登录                </van-button>            </div>        </van-form>    </div></template>
```

axios发请求

```xml
复制代码
<script setupimport axios from 'axios'import { ref } from 'vue';const username = ref('')const password = ref('') const onSubmit = (values) => {  axios.post('http://localhost:3000/login', values) // 希望后端有这么个地址，并向他传参 一定是个对象，刚好vant就是给你一个对象    .then(res => {  console.log(res);    })}</script
```

后端

```less
复制代码
npm init -ynpm i koanpm i koa2-cors // koa解决跨域npm i koa-bodyparser // koa解析不出post请求npm i koa-router // koa-router写接口
```

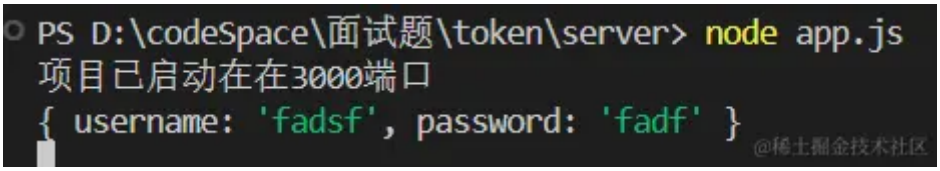
`app.js` 如下

```php
复制代码
const Koa = require('koa')const cors = require('koa2-cors') // 这样不用写相应头，白名单什么的 npm i koa2-corsconst bodyParser= require('koa-bodyparser')const userRouter = require('./routes/user') // 让接口生效const app = new Koa()app.use(bodyParser())// koa无法解析post参数，这是koa一个小问题，安装npm i koa-bodyparserapp.use(cors()) // 处理跨域app.use(userRouter.routes(), userRouter.allowedMethods()) app.listen(3000, () => {    console.log('项目已启动在在3000端口'); })
```

`routes/user.js` 如下

```
1 javascript
2 复制代码
3 // 和用户相关的接口const router = require('koa-router')() // 直接调用// 定义一个
  post请求的接口router.post('/login', (ctx) => {    let user = ctx.request.body //
  拿到前端的参数    console.log(user)})module.exports = router
```

好了，现在前端是可以朝着后端发请求的，并且可以拿到数据



好了，上面的准备工作已经完成了，前端已经可以发请求到后端，后端也能拿到数据

接下来后端的逻辑应该是拿到这个账号密码去数据库核对是否存在，这里我省去这个步骤，假设一定有这个账号密码

`routes/user.js`

```
1 javascript
2 复制代码
3 // 和用户相关的接口const router = require('koa-router')() // 直接调用// 定义一个
  post请求的接口router.post('/login', (ctx) => {    let user = ctx.request.body //
  拿到前端的参数        if (1) {          ctx.body = {            code: 0,
    data: `你好${user.username}`        }    } else { // 数据库中没有这个账号
    ctx.body = {            code: 1,            data: '账号后密码错误'        }
  }})module.exports = router
```

这样，就实现了后端将数据返回给前端，前端成功拿到数据

现在有个问题，就是我可以在用户界面不输入账号密码，直接修改 `url` ，跳去 `home` 主页去，照样是能过去的，首页一定是根据账号来进行展现的，因此账号不同首页数据不同，如何进行区分呢？后端返回一个 `code: 0` 一定是不够的，还需要返回一个 `token`

token

> 参考：JSON Web Token 入门教程 - 阮一峰的网络日志 (ruanyifeng.com)

`JWT` (JSON Web Token)是目前最流行的跨域 `认证` 解决方案

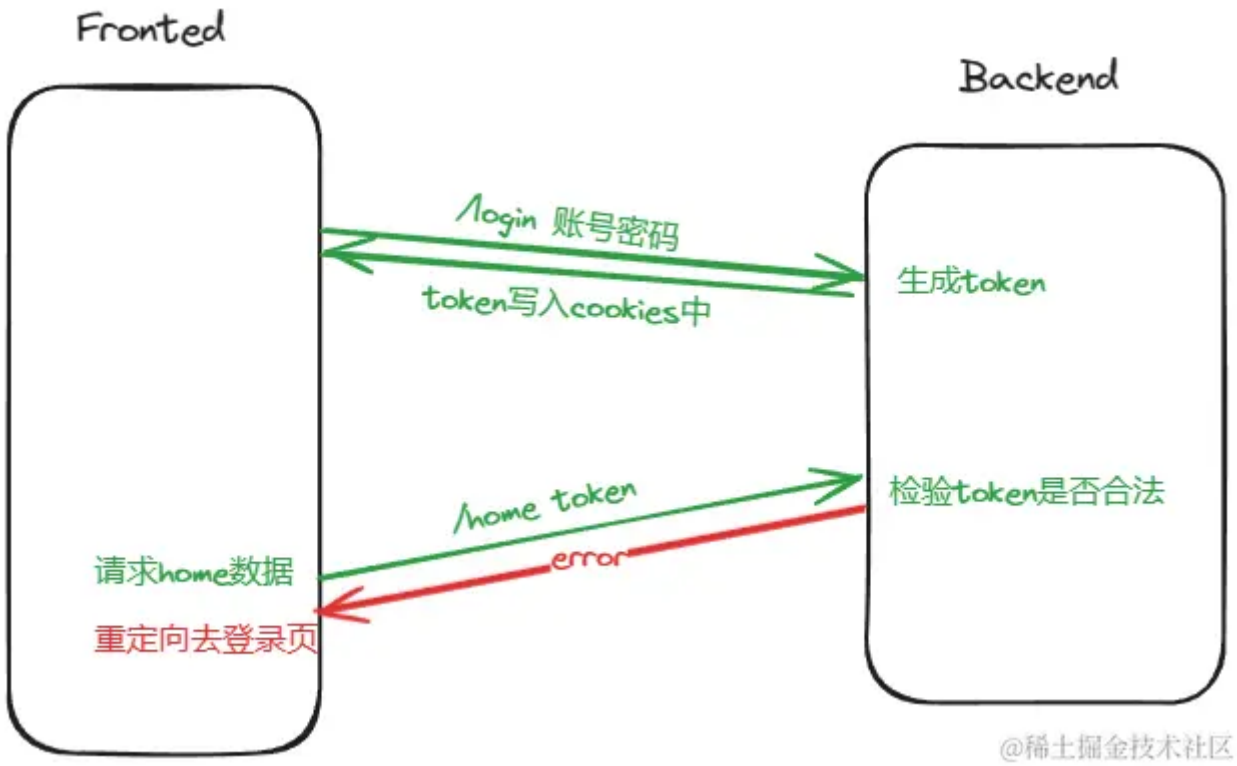
> 跨域认证重点在认证二字

跨域 `认证` 流程

1. 用户向服务器发送用户名和密码。
2. 服务器验证通过后，在当前对话（session）里面保存相关数据，比如用户角色、登录时间等等。
    1. 这是后端的 `session` 会话，非前端的 `sessionStorage` ，存在运存中
3. 服务器向用户返回一个 session_id，写入用户的 Cookie。
    1. 这个 `session_id` 就是 `token` 令牌，大多数情况下浏览器的 `Cookie` 由后端控制
4. 用户随后的每一次请求，都会通过 Cookie，将 session_id 传回服务器。
5. 服务器收到 session_id，找到前期保存的数据，由此得知用户的身份。

> `cookie` 虽然是浏览器的内存空间，但是受后端掌控，后端将登录令牌保存在 `cookie` 中
>
> 而 `cookie` 有个特点，所有保存在 `cookie` 中的数据，都会在 `http` 请求时自动被携带在请求头中

再一段话解释下流程：用户发送账号密码给后端校验，后端通过校验，并在 `session` 中存入唯一数据，然后返回一个 `token` 给前端，将其写入到浏览器的 `cookie` 中，之后用户的每次请求都会把 `cookie` 带上，而 `cookie` 中又会有 `token` ，因此就是每次请求(比如这里的请求首页)都会把 `token` 带回给后端，后端会比较这个 `token` 和之前的自己生成的 `token` 是否一致，一致则同一账号



另外，再来想想一个情景：假设我有个账号已经登录在自己电脑里了，然后我在另外一个设备中也登录了这个账号，两个设备登录之后后端都会返回一个 `token` 回去，两个 `token` 会一样吗？必然是不一样的，因为 `token` 里面还会带上时间戳，在后端看来，我仅仅只维护最新的 `token` ，因此原来的设备你在个人首页走向个人详情页的时候，会带上一个 `token` 给到后端，后端发现，诶？我的 `token` 不是已经更新掉了吗，因此，这个 `token` 是非法的，后端会给你返回一个登录失效

> 这就是面试官常问的，账号在别处修改了账号密码原设备被顶掉了怎么办

当然，这其实是有两种情况的，一个是你在别处修改密码后，原设备没有发任何请求，你也被告知下线，这是借助 `websocket` 来实现的实时通信；还有一种是原设备可以留在原地，不受影响，但是一旦你点击刷新或者去到别处页面发请求，后端才会通知你登录失效

> 当然 `token` 也可以设置过期，过期同样登录失效

一个注意点：登录注册接口是不携带 `token` 的，因为 `token` 就是需要你先返回一个账号密码再去生成；

还有就是，将 `token` 存入 `cookies` 中是后端常干的行为，但是后端也可以偷懒，不给你存入，后端就相当于直接将 `token` 当成字符串直接返回给前端， `ctx.body` 中多个 `token` 。既然如此，那么这个工作只能让前端来干了，前端自己将 `token` 存入 `cookie` 中去。当然浏览器的三大存储 `Local storage` ， `Session storage` ， `Cookies` ，浏览器都可以进行操作，所以前端做这个工作，你爱存哪里存哪里，有时候前端可能做的是后台管理系统，登录需要就在当前会话中生效，退出页面就会失效，因此会将token存入 `Session storage` 。若是存入 `cookie` ，你可以设置过期时间，没有过期就不会失效

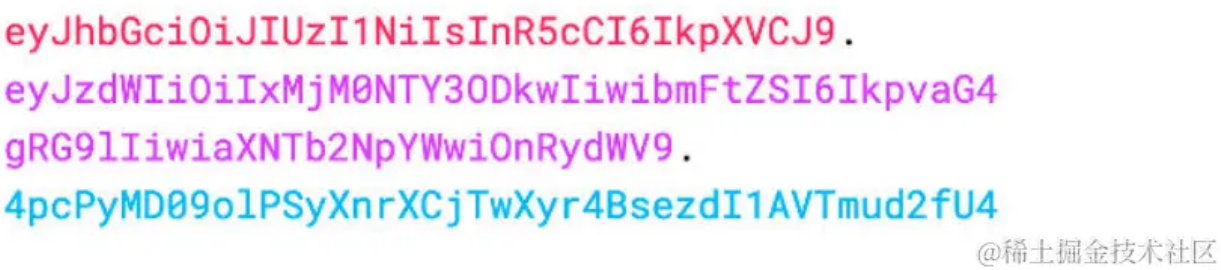
> 接下来的 `token` 实现我就前端来存入 `LocalStorage` 中

另外， `token` 如果长成这个样子会怎么办

这样不就是明文传输吗，被别人看到了怎么办，非常不安全！

> 假设后端真这样把 `token` 明文传给前端，小明在网吧准备给账号氪金，但是先去上了趟洗手间，这个过程中法外狂徒张三过来点击 `F12` 进入调试页面，将你的 `token` 中的账号字段改成张三自己的账号，最后小明氪金氪给了张三账号……
>
> 另外， `token` 中是不会包含账号的密码的

但是后端生成的 `token` 确确实实是这样的对象，为保证安全，后端还会将其进行加密，最终长成如下这样



组成

标准的 `token` 包含三部分：

- Header（头部）
  - 存放账号信息
- Payload（负载）
  - 存放账号的描述
- Signature（签名）
  - 签名就是标记的意思

这里特意强调 `标准` 的 `token` 有着三部分，因此后端你也可以就是将 `token` 仅仅包含账号信息，这样很多需求就实现不了了

`token` 实现

紧接着上面的情景，我现在就是让后端生成一个 `token` ，让前端自己存储，第一步，后端生成 `token` ，并进行加密并返回给前端，然后前端存入 `LocalStorage` 中，并跳转页面发请求

大概模拟下，就是如下这样

```
javascript
复制代码
const router = require('koa-router')() // 直接调用router.post('/login', (ctx) => {    let user = ctx.request.body // 拿到前端的参数        if (1) {    ctx.body = {            code: 0,            data: `你好${user.username}`    }    } else { // 数据库中没有这个账号        ctx.body = {            code: 1,        data: '账号后密码错误',            token: 'xxxxxxx'  // 大概这样子    }    }})module.exports = router
```

生成token

> 参考：jsonwebtoken - npm (npmjs.com)

国外已经有人造好了轮子，我们不需要自己写算法实现

后端安装好 `npm i jsonwebtoken`

我把生成 `token` 的函数封装到 `server/utils/jwt.js` 中

server/utils/jwt.js

写个 `sign` 函数用于生成 `token` ，既然 `token` 的生成依据账号，那就需要传入前端返回的账号对象，另外还需要个加密方式，这个参数称之为 `加盐` ，我就写个 `666`

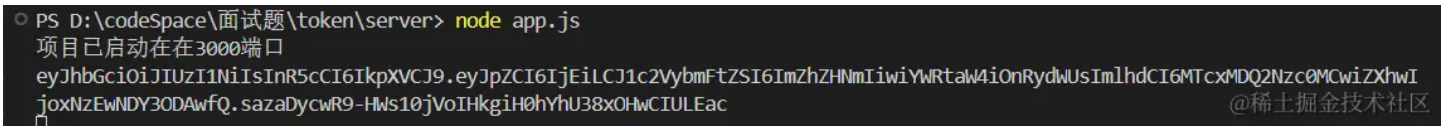
```
java
复制代码
// 封装一个可用于创建token的函数const jwt = require('jsonwebtoken') function sign (option) { // 生成token    return jwt.sign(option, '666', { // 第二个参数为加盐，666放到了账号中去        expiresIn: 60 // token的有效时长 单位s    })}module.exports = {    sign}
```

好了，现在拿到 `server/routes/user.js` 中调用这个函数生成 `token`

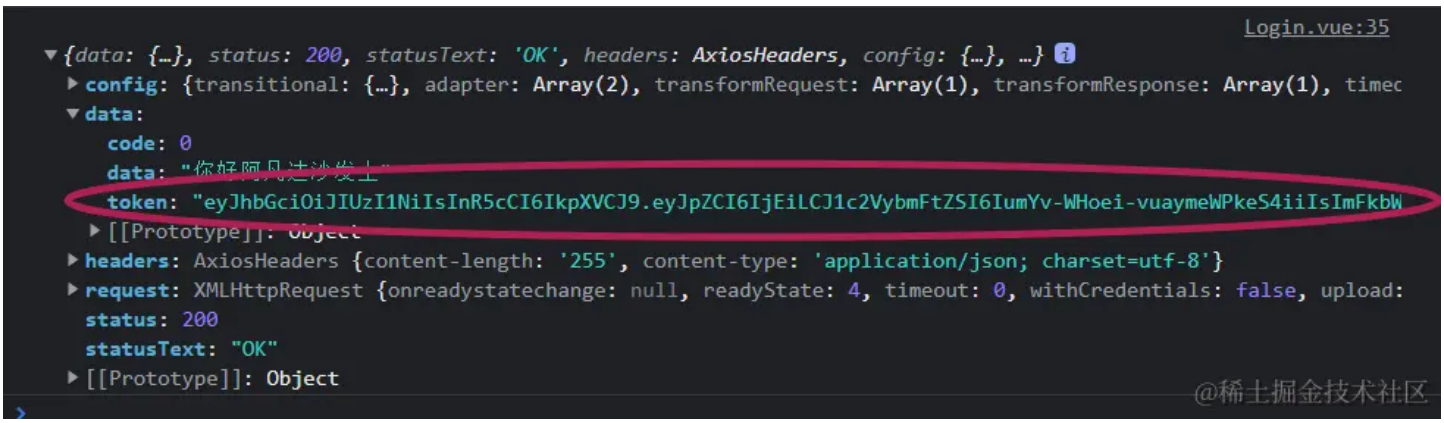
```
javascript
复制代码
// 和用户相关的接口const router = require('koa-router')() // 直接调用const jwt = require('../utils/jwt.js')// 定义一个post请求的接口router.post('/login', (ctx) => {    let user = ctx.request.body    if (1) { // 模拟验证成功        // 创建一个token        let jwtToken = jwt.sign({id: '1', username: user.username, admin: true}) // admin: true意思是管理员的token，权限        console.log(jwtToken); // 已经加密好了        ctx.body = {            code: 0,            data: `你好${user.username}`,            token: jwtToken        }    } else { // 数据库中没
```

```
有这个账号          ctx.body = {              code: 1,              data: '账号后密码错
误'          }     }})module.exports = router
```

打印下，确实有，并且还已经加密好了，所以肯定给到了前端



前端也确确实实接收到了



好了，现在前端自己存到 `LocalStorage` 中去

前端存储

紧接着前端存储下来，然后跳转到首页，跳转页面用 `router.push()` ，记得引入路由

`client/src/views/Login.vue`

```
1 xml
2 复制代码
3 <script setupimport axios from 'axios'import { ref } from 'vue';import {
  useRouter } from 'vue-router'const router = useRouter()const username =
  ref('')const password = ref('') const onSubmit = (values) => {
  console.log(values)    // // 发请求 发请求 封装axios 或者ajax的fetch，vue官方推荐你
  用axios，很好用的库    axios.post('http://localhost:3000/login', values)
  .then(res => {        console.log(res.data.token);
  localStorage.setItem('token', res.data.token)        router.push('/home') // 来
  到首页需要发接口请求，然后把token给到后端    })}</script
```

好了，现在来到了首页，我们的 `token` 工作还远远没有完成，你还需要来到首页后发接口请求，并将 `token` 带入给后端

`client/src/views/Home.vue`

```
xml
复制代码
<script setupimport { onMounted } from 'vue';onMounted(() => {
axios.get('/home')      .then((res) => {         console.log(res);     })
.catch(err => {         console.log(err);     })})</script
```

如果这样写，直接用 `axios` 发请求，那就和 `token` 没有任何关系，并且你甚至可以在登录页直接输入 `url` 到 `home` 页

因此我们需要在 `axios` 发请求的时候把 `token` 带给后端，我们不建议将 `token` 以？拼接传入后端，这样的话每个接口请求都会长这样，很丑~

因此，我们自己封装 `axios`

封装axios

封装的目的就是保证每个请求可以把 `token` 带给后端，另外还需要做一个响应拦截，请求失败了或者 `token` 过期了也应该交给 `axios`

不过是前端发请求，还是后端返回的响应都会经过 `axios` ，因此需要统一进行判断，如果 `code` 不是 `0` 那就是逻辑性错误，比如密码错了，如果这样就返回一个 `Promise` 出来，方便捕获错误，易于调试，这样刚才首页发接口请求就可以 `then` 后面写 `catch` 了

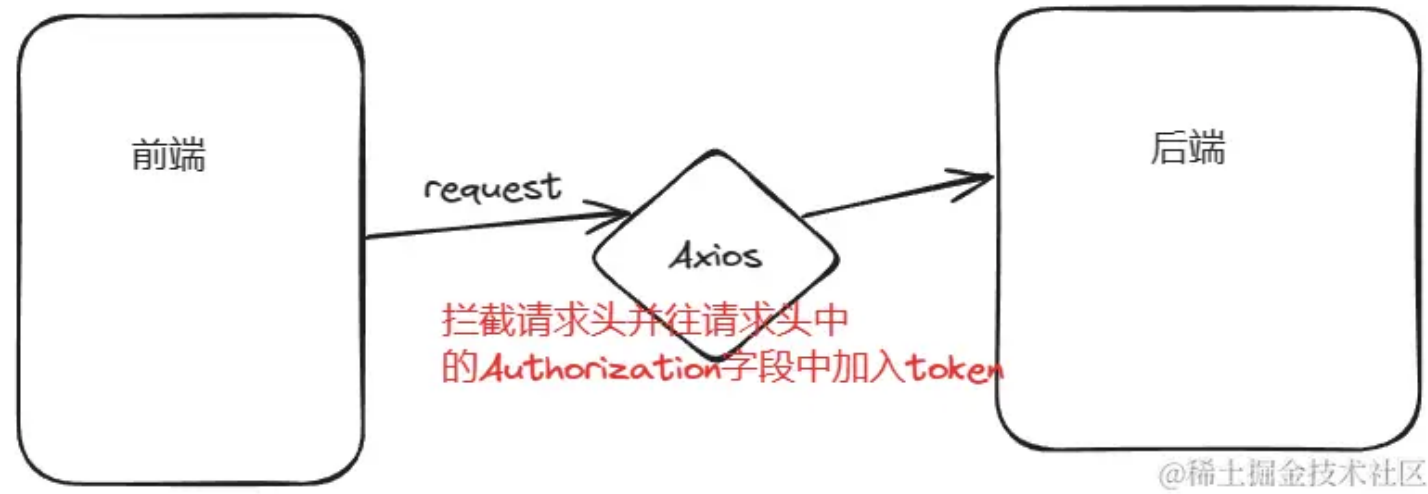
> 逻辑性错误就是后端没有崩掉，是业务逻辑出错，后端崩掉是程序性错误

`client/src/api/index.js`

```
javascript
复制代码
import axios from 'axios'import router from '../router'axios.defaults.baseURL
= "http://localhost:3000"// 请求拦截 axios.interceptors.request.use(config => {
   let token = localStorage.getItem('token')    if (token) {
config.headers.Authorization = token    }    return config // 把请求拦截下来，并往
请求头中加入token，然后return })// 做一个响应拦截，比如登录失败需要提示登录失败 发请求和
接受都需要经过axios的手axios.interceptors.response.use(    (res) => {        if
(res.data.code && res.data.code !== 0) { // 逻辑性错误，比如密码敲错了，并不是程序性
错误            return Promise.reject(res.data.error) // 这么做的意义是，让axios好
去调试，可以捕获错误        }        if (res.data.status= 400 && res.data.status
< 500) { // 程序性错误            // 状态码在[400, 500) 就认为用户没有权限，就强行把
你重定向到登录页面            router.push('/login')            return
Promise.reject(res.data)        }        return res  // 响应的内容没有问题
})export function post(url, body) {     return axios.post(url, body)}
```

- 请求拦截就是把 `token` 拿到，如果存在 `token` ，存入到 `header` 中的 `Authorization` 字段中，这个字段是我们自己命名的

- 如果状态码在 `[400, 500)` 之间就说明用户没有权限，比如没有登录就来请求数据，那就把他强行送到登录页



然后登录和首页的接口请求我再封装下，我把 `post` 刚刚被抛出的请求引入到下面这里

`client/src/api/user.js`

```
1 javascript
2 复制代码
3 import { post } from './index.js'export function login(body) {    return
  post('/login', body).then(res => {        return res.data    })}
```

这样我的 `Login` 页面，就不需要原来 `axios` 请求了，直接把 `login` 函数拿来调用下就可以了，优雅！

`Login.vue`

```
1 xml
2 复制代码
3 <script setupimport { ref } from 'vue';import { useRouter } from 'vue-
  router'import { login } from '../api/user.js'const router = useRouter()const
  username = ref('')const password = ref('') const onSubmit = (values) => {
  login(values).then(res => {        console.log(res);
  localStorage.setItem('token', res.token)        router.push('/home') // 来到首页
  需要发接口请求，然后把token给到后端    })}</script
```

同样， `home` 页的请求我也封装好

`client/src/api/user.js`

```
1 javascript
```

```
2 复制代码
3 import { post } from './index.js'export function login(body) {    return
  post('/login', body).then(res => {        return res.data    })}export function
   home() {    return post('/home').then(res => {        return res.data    })}
```

`Home.vue`

```
1 xml
2 复制代码
3 <script setupimport { home } from '../api/user.js'import { onMounted } from
  'vue';onMounted(() => {    // axios.get('/home') // 不建议把token以? 接在url后面,
  这样用了,所有页面都这样用很难看,所以封装下,复用    home()    .then((res) => {
    console.log(res);    })    .catch(err => {        console.log(err);    })})
  </script
```

目前，后端还没有 `home` 接口，因此会报一个 `404` 的错误，并且这个请求头中会有 `authorization` 字段存放 `token` ，只要是 `404` 那么就会把你送回登录页

好了，现在去后端写首页的接口，以及 `token` 的校验

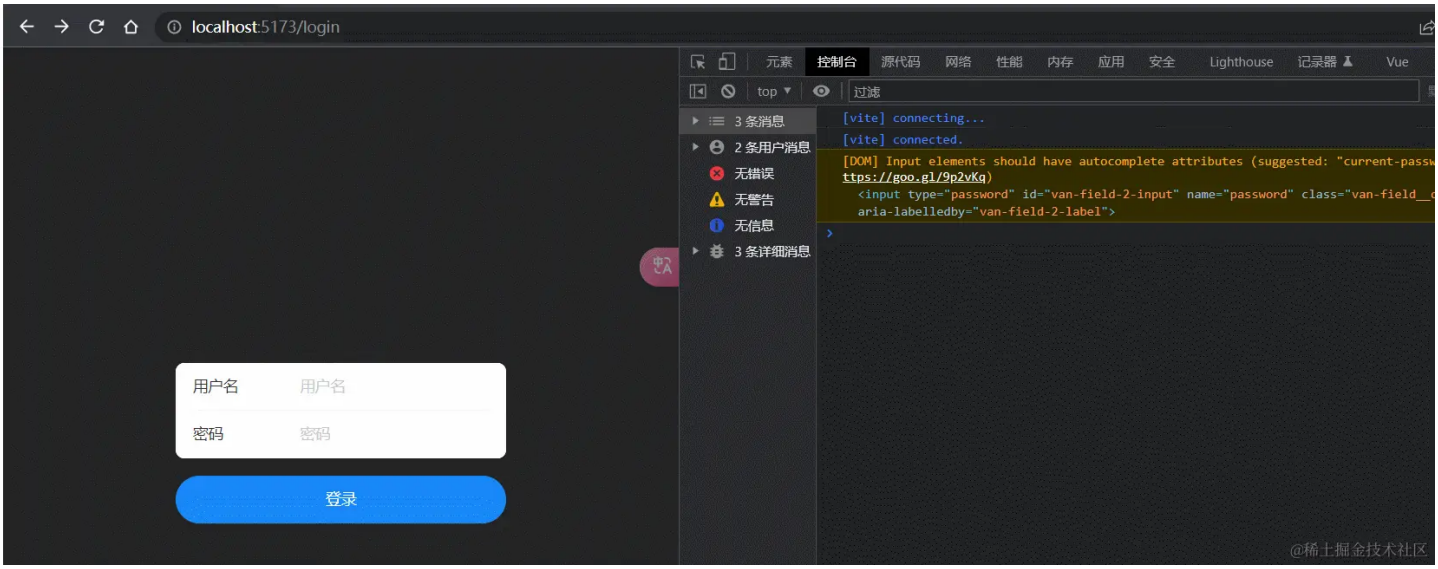
token校验

先把后端的 `home` 接口写好

`server/routes/user.js`

```
1 javascript
2 复制代码
3 router.post('/home', (ctx) => {    ctx.body = {        code: 0,        data: '这
  是首页的数据'    }})
```

这样写没有进行校验，那么你前端就可以从登录页不登录修改 `url` 去到 `home` 页



这样就是鉴权失败!

鉴权不仅仅是校验是否存在 `token` 还要去校验这个 `token` 是否是我当初加密了的 `token` ，当初我的那个 `token` 是加了个 `666` 的，要是你仅仅是看是否存在 `token` ，那么别人可以直接去浏览器应用页面新增乱写的 `token`

这个分析方法同样我把它封装到 `jwt.js` 中， `verify` 验证方法是 `jwt` 自带的，并且需要带上当初生成 `token` 的加盐参数，也就是 `666`

如果前端传过来的 `token` 有问题，那么就向前端返回一个 `401` 的状态码， `401` 就是无权，用户名，密码，令牌错误

校验成功就 `next` 放行，如果没有前端没有传过来 `token` ，那就同样向前端输出 `401`

```
1 javascript
2 复制代码
3 // 封装一个可用于创建token的函数const jwt = require('jsonwebtoken') // npm i
  jsonwebtokenfunction sign (option) { // 生成token    return jwt.sign(option,
  '666', { // 第二个参数为加盐，666放到了账号中去        expiresIn: 60 // token的有效
  时长 单位s    }) }const verify = () => (ctx, next) => { // 校验token是否有效
  let jwtToken = ctx.req.headers.authorization // 前端传过来的authorization需要写成
  小写    if (jwtToken) {        jwt.verify(jwtToken, '666', (err, decoded) => {
          if (err) { // 前端传过来的token有问题                ctx.body = {
            status: 401, // 没有权限                    message: 'token失效'
          }          } else {                // 校验成功
  next() // 放行          }          })          } else {        ctx.body = {
          status: 401,            message: '请提供token'        }
  }}module.exports = {    sign,    verify}
```

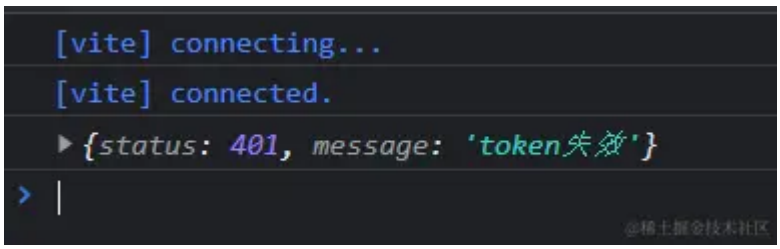
这个方法我拿到 `routes/user.js` 中调用

```
1 javascript
2 复制代码
3 router.post('/home', jwt.verify(), (ctx) => { // 请求这个地址时，校验失败就不走回调
    ctx.body = {        code: 0,        data: '这是首页的数据'    }})
```

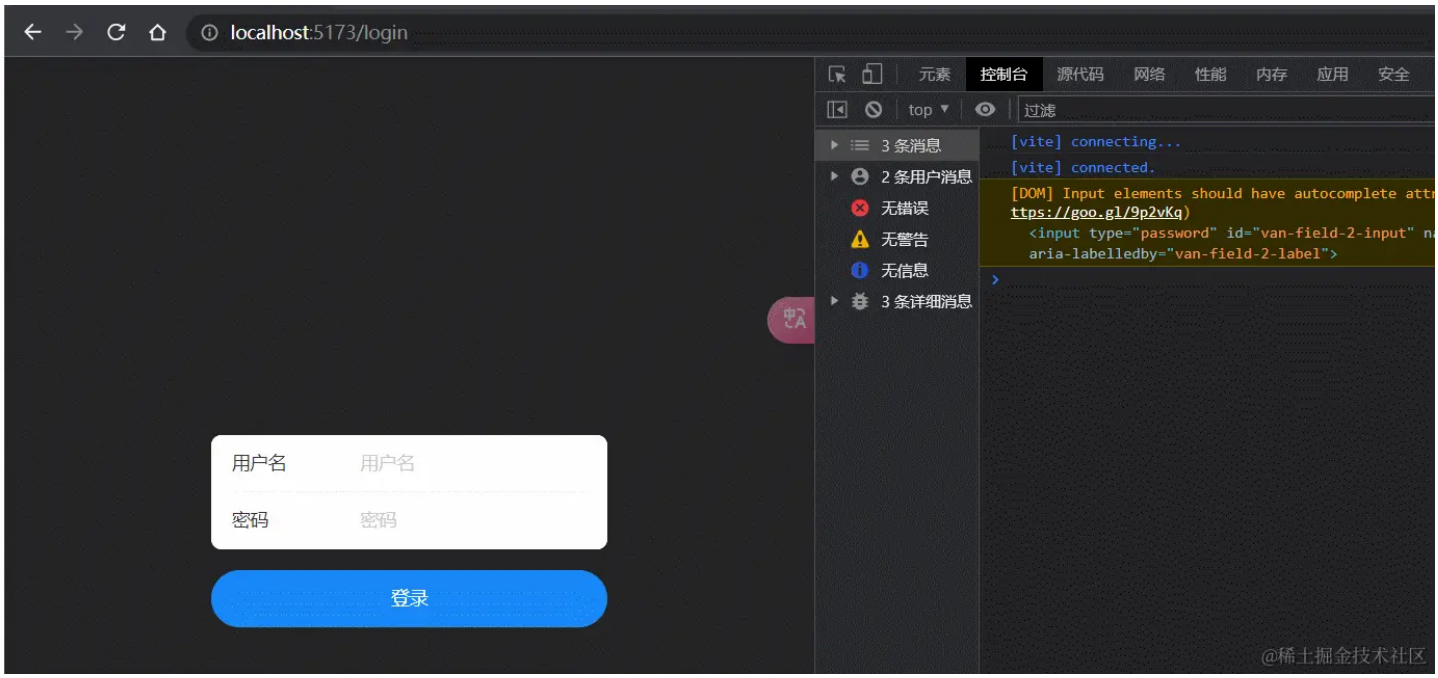
现在检验下

我清空 `token` 数据，然后刷新首页，没有拿到首页数据，这样做同样也是直接输入 `url` 到首页的效果，并且最终被送回到登录页

我再把 `token` 清空掉，自己加一个 `token` 然后刷新首页， `token` 失效，没有拿到首页数据，并且最终被送回到登录页



现在摆在用户面前的是，只有老老实实登录，才能拿到首页数据



至此，整个 `token` 的流程都已经实现完毕，前端登录后，将账号密码返回给后端，后端生成，加密 `token` ，并返回给前端，前端存入 `LocalStorage` 并通过 `axios` 在每一次请求拦截中将 `token` 存入请求头中的 `Authorization` 字段并返回给后端，后端进行校验是否为当初加密了的 `token` ， `token` 合法才返回数据，否则返回 `401` 状态码告诉前端 `token` 失效，实现了登录鉴权~

说完这段话，留心面试官可能会问你浏览器三种存储的区别，以及各个状态码的含义