Théotime HUYBRECHTS

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

# Lab 4 Web development : Microsoft authentification

## Question 1 :

Why should you not commit credentials on git ?

We should never commit credentials such as API keys, secrets, or authentication tokens on Git because it poses a serious security risk. Once committed, these credentials can be easily accessed by anyone with access to the repository, potentially leading to data breaches.

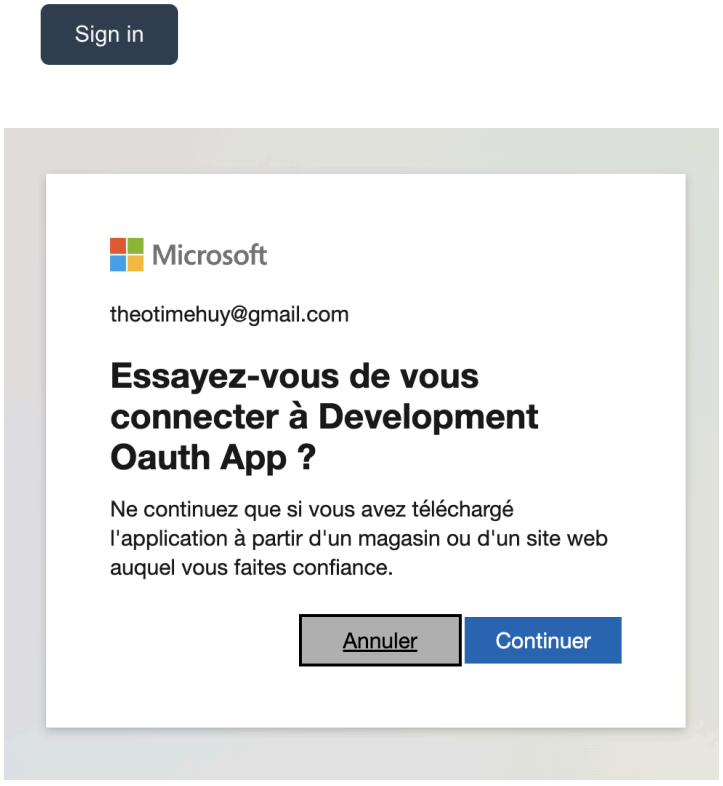
## Question 2 :

Why may you want different configurations depending on the environment ? Give an example?

Different environments (e.g., development, testing, staging, production) often have unique requirements in terms of security, performance, and debugging. For example, in a development environment, we want to Verbose logging and detailed error messages to assist in debugging. As in production we want to minimize logging to avoid revealing sensitive information.

## Exercise 1 :

I already wrote helpers on top of msla (see the file src/lib/microsoftGraph.js above). Import them for use in your SigninButton component. For now, SigninButton stores the resolved user in- side its own state (the data property of your component) and displays it in the template.




I did create a button that calls the microsoft auth.

# Exercise 2:

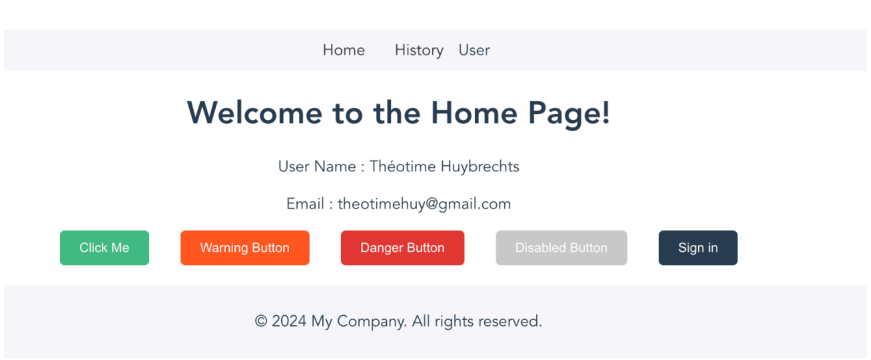
## Use props to share the user with both SigninComponent and HomePage. Then use events to update the shared user from SigninComponent. Eventually, display the user name in HomePage.

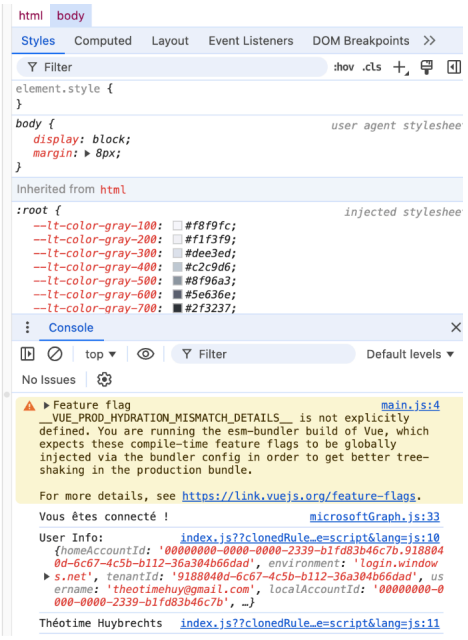
I followed what was said and displayed the infos in the HomePage.

They are “stored” in the App.vue

```vue
<template>
  <base-layout>
    <home-page :user="user" @userChanged="handleUserChange"/>
  </base-layout>
</template>

<script>
import HomePage from './components/HomePage.vue';
import BaseLayout from './components/BaseLayout.vue';

2 usages  Thimoclesse
export default {
  name: 'App',
  components: {
    HomePage,
    BaseLayout,
  },
  data(){
    return {
      user: null,
    };
  },
  methods: {
    handleUserChange(newUser){
      this.user = newUser;
    }
  }
};
</script>

<style>

</style>
```

and catch up in the button that show it in the HomePage



## Question 3:

### While being a well-working solution, it suffers from maintainability issues. Please expose and discuss them.

Each component that needs the user information depends on its parent component to pass it down via props. This can make the application harder to refactor, especially as the app grows.
As the app grows, managing state through props can become unmanageable, especially if many components require access to the same state.

## Question 5:

### Build a comparison table between the various state manage ment strategies available, especially about pro and cons. Optionally, feel free to explore other ways not covered in that tutorial.

| **Strategy** | Pros | Cons |
|---|---|---|
| **Strategy** | - Simple for small apps | Harder to manage as the app grows |
| **Vuex** | Easy to share state | -Can add complexity |
| **Provide/Inject** | Modern alternative to Vuex | Still requires setup |
| **Provide/Inject** | Avoids props drilling | Limited to parent-child relationships |

## Question 6:

### Imagine a developer in your team suggests to exclusively manage the state with stores. Therefore, it recommends not to rely on props and provide anymore. Would you accept this? An argued answer is expected.

While managing state exclusively with stores like Vuex or Pinia offers benefits, I would not recommend completely abandoning props and provide/inject. Here's why:

- Stores are great for global state, but using them exclusively can be overkill for smaller, isolated components that don't need to share state across the app.
- Props and provide/inject offer simpler, more performant solutions when you only need to pass data between parent and child components.

- Mixing state management strategies allows you to choose the best tool for the specific use case, resulting in a more flexible and maintainable codebase.

## Question 7: What is the performance difference between:

### • <a href="/conversations">Conversation</a>

**Page reloads** when clicked, as the browser interprets it as a full page navigation.

**Slower** performance due to the complete reload of the page, reinitializing the Vue app.

### • <router-link to="/conversations">Conversations</router-link>

**No full page reload**. Vue Router intercepts the click. It's **Faster** because only the required components are updated, not the entire page, making the experience more seamless and efficient.