

In diesem Arbeitsblatt wollen wir Cookies dazu verwenden, um Sessions in einem Express Server zu verwenden. Dazu werden wir auch einen Client, selbstverständlich mit Vue 3, programmieren.

Hier der Ablauf mit speziellem Blick auf die Server Seite:

- **express-session** erzeugt für jeden neuen Client, der den Server kontaktiert, ein Session-Objekt. Dieses **Session Objekt** hat als Property eine eindeutige **Session ID**.
- Der Server fügt bei allen HTTP-Anfragen das **Session Objekt** zum **req Objekt** als Property **session** hinzu. Es steht somit in jeder Route zur Verfügung!
- Du kannst auch eigene Properties in dem **Session Objekt** abspeichern. Diese werden in einer internen Session Tabelle abgespeichert. Die **Session ID** ist der Key für die richtige Zeile in dieser Tabelle. Defaultmäßig ist die Tabelle im Hauptspeicher (**MemoryStore**). Das hat Nachteile (siehe weiter unten).
- Nur die **Session ID** wird an den Client mit einem Cookie übermittelt (signiert) und dort gespeichert. Zur Signierung des Cookie Payloads dient das **SESSION_SECRET**.
- Der Client schickt, solange das Cookie existiert, bei jeder HTTP Anfrage an den Server das Cookie mit.
- Bei jeder HTTP Anfrage rekonstruiert **express-session** aus dem mitgeschickten Cookie die **Session ID**, schaut in der internen Session Tabelle nach und fügt die dort gespeicherten Properties zu dem **Session Objekt** hinzu.
- In der Route stehen somit wieder auch alle vorher hinzugefügten eigenen Properties zur Verfügung.
- Am Ende der Session wird das Cookie am Client zerstört und die Session am Server aus der internen Session Tabelle entfernt.

Aufgabe 1. Starte mit dem Code von **START Sessions Server.zip** und **START Sessions Client.zip**.

Dieser Server bietet bereits folgende Routen (die aber noch nicht implementiert sind):

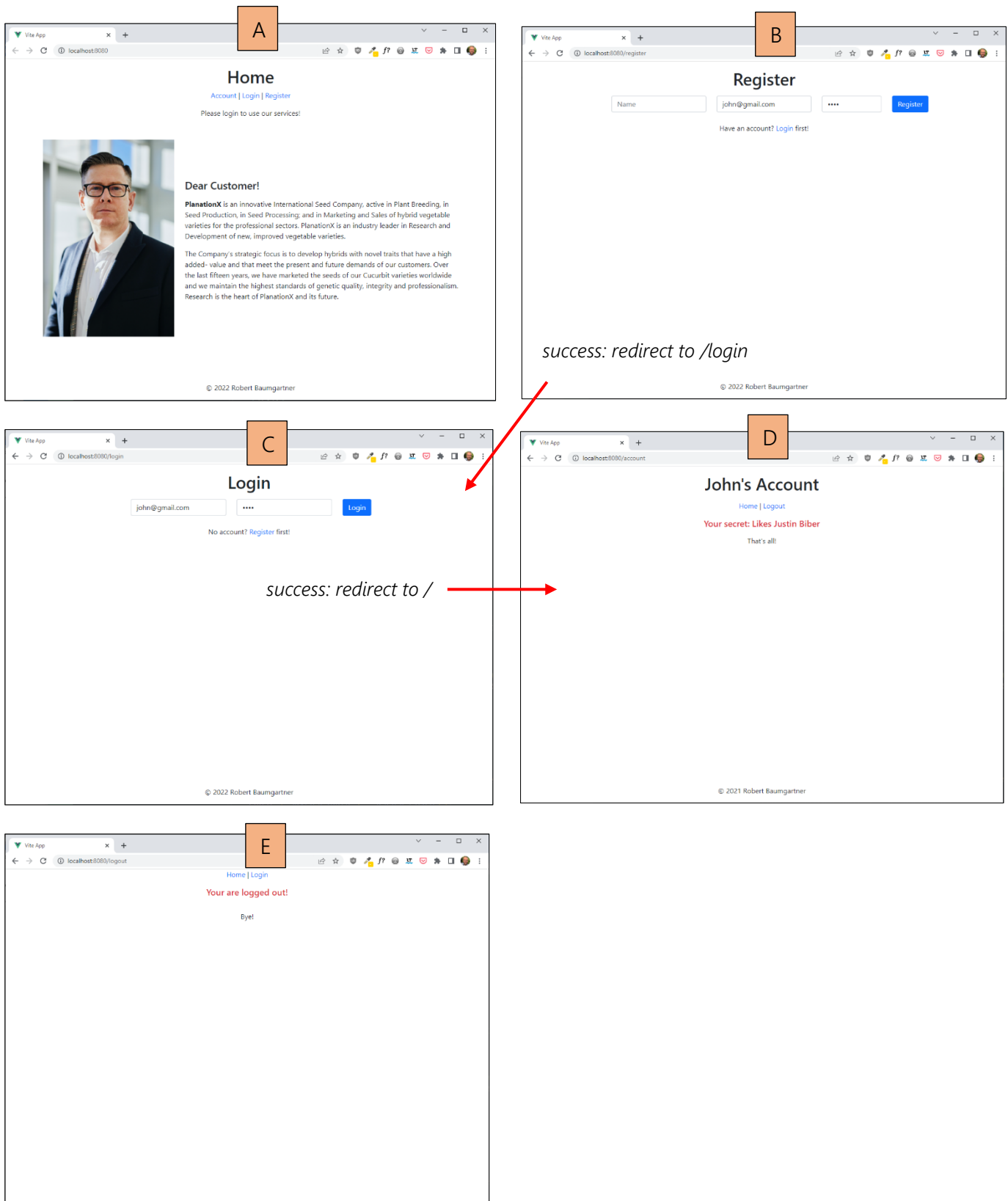
HTTP Verb	Pfad	Bedeutung
POST	/api/register	Übermitteln der Daten zum Registrieren
POST	/api/login	Übermitteln der Daten zum Login
GET	/api/users/:id/secret	Geheime Daten
GET	/api/logout	Löschen der aktuellen Session und des Cookies

Der Client bietet bereits die Views und folgende Routen:

Pfad	Bedeutung
/	Anzeige der Willkommen-Seite
/register	Formular zur Registrierung des Users
/login	Anzeigen des Formulars zum Einloggen des Users
/logout	Bestätigung, dass der User ausgeloggt ist
/account	Geheime User Daten

Die Links funktionieren den Namen entsprechend, d.h. **Home** leitet immer zu **/home** um. Ausnahmen:

- In **Home**: Die Route **/account** führt, wenn man nicht eingeloggt ist, zu **/login**.
- In **Login**: Der Button **Login** führt, wenn man nicht eingeloggt ist, zu **/login**.

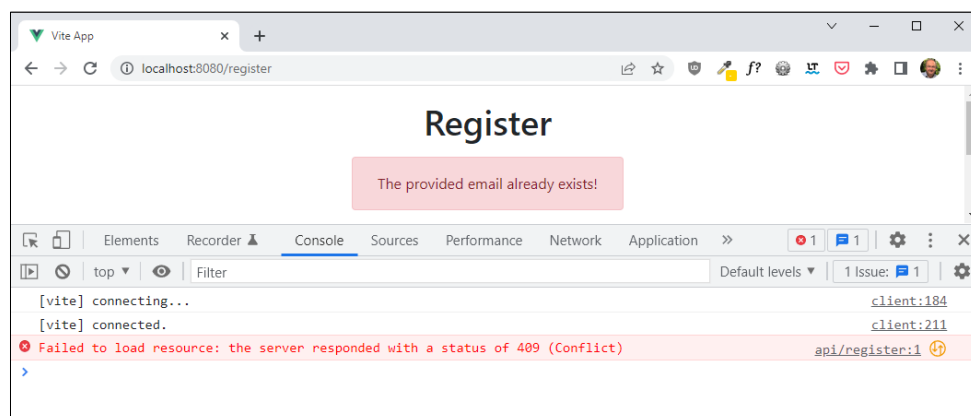
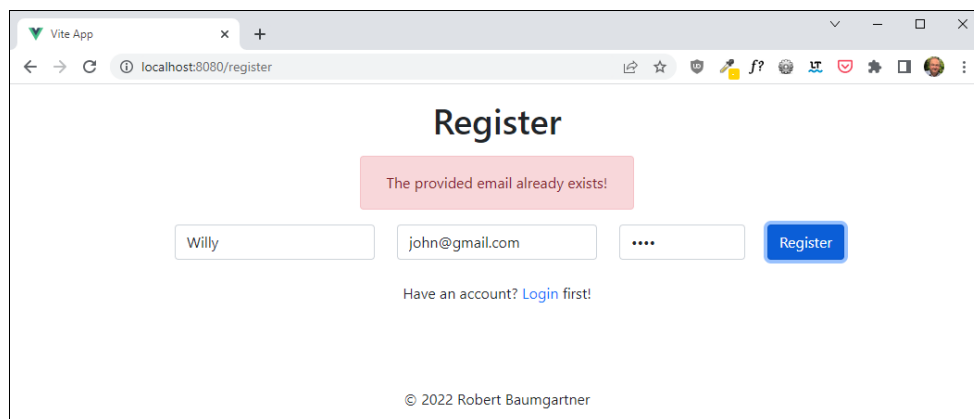


Der User wird zunächst auf der Home Seite (A) begrüßt. Ist er bereits eingeloggt, kann er gleich zur Account Seite (D) wechseln. Auf der Account-Seite werden persönliche Daten angezeigt (rot). Ist er noch nicht registriert, muss er sich registrieren (B). Dabei darf eine E-Mail-Adresse nur einmal für eine Registrierung verwendet werden.

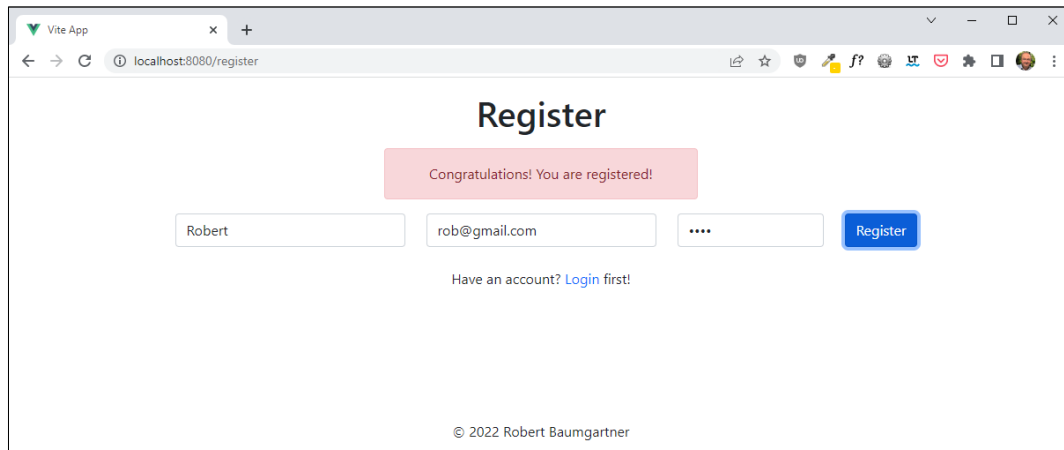
Am Server gibt es bereits eine Liste mit folgenden Daten.

```
export const users = [  
  {  
    id: 1,  
    name: 'John',  
    email: 'john@gmail.com',  
    password: '1234',  
    secret: 'Likes Justin Biber',  
  },  
  {  
    id: 2,  
    name: 'Sandy',  
    email: 'sandy@gmail.com',  
    password: '1234',  
    secret: 'Hates Deutschrap',  
  },  
  {  
    id: 3,  
    name: 'Willy',  
    email: 'willy@gmail.com',  
    password: '1234',  
    secret: 'Sleeps with a gun under his pillow',  
  },  
];
```

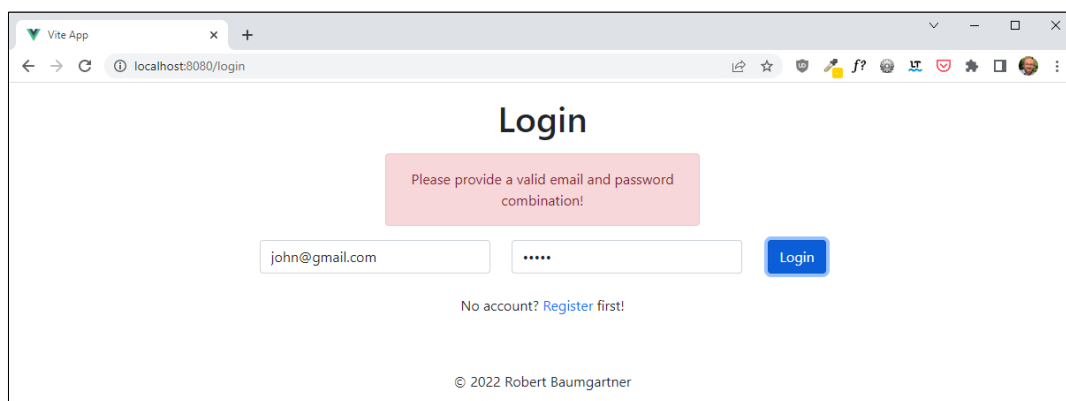
Beispiel für fehlgeschlagene Registrierung:



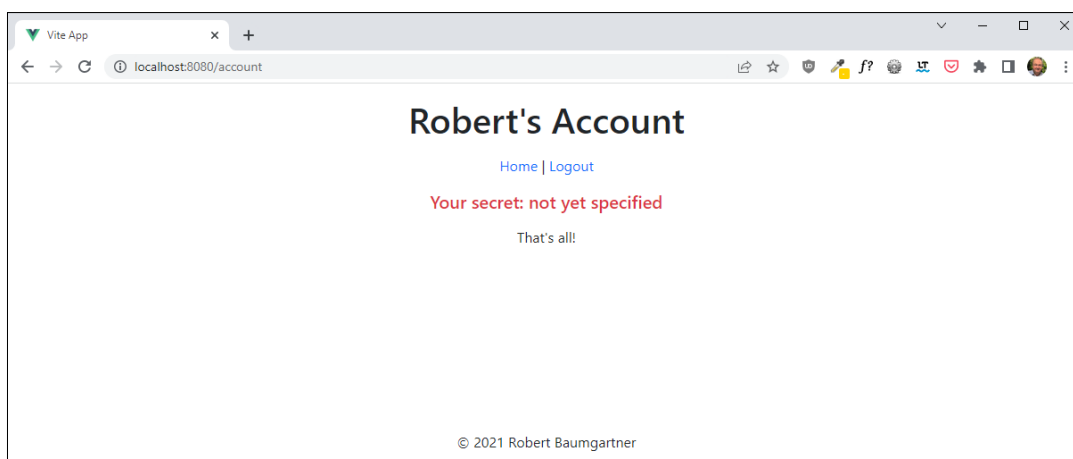
Beispiel für erfolgreiche Registrierung:



Geht das Login schief, bekommt der User eine Nachricht:



Nach erfolgreichem Login wird der User auf die Account Seite weitergeleitet. Neue User, die sich gerade registriert haben, besitzen noch kein Geheimnis. In dem Fall wird auf der Account Seite folgendes angezeigt:



So weit die Übersicht.

Aufgabe 2. Installiere das npm Modul **express-session**.

express-session ist eine Middleware, welche Tätigkeiten in Bezug auf die Session für uns erledigt, d. h. das Erstellen der Session, das Setzen des Session-Cookies und das Erstellen des Session-Objektes im **req**-Objekt.

In diesem Arbeitsblatt verwenden wir den Defaultort zum Speichern von Sitzungen, d. h. **MemoryStore**. Natürlich sind dann alle Sessions weg, wenn wir den Server restarten oder wenn er crashed!

Außerdem aus der Doku:

Warning The default server-side session storage, `MemoryStore`, is **purposely** not designed for a production environment. It will leak memory under most conditions, does not scale past a single process, and is meant for debugging and developing.¹

Daher verwende niemals den `MemoryStore` in der Produktion.

Wir werden im nächsten Arbeitsblatt PostgreSQL dazu verwenden.

Alternative: **cookie-session**. Diese Middleware speichert die Daten im Cookie am Client (Bsp: Shopping Cart).

Aufgabe 3. Um die **express-session** Middleware zu registrieren, verwende in `app.js` die Middleware Registrierung `app.use`. Bei dieser Gelegenheit kann **express-session** konfiguriert werden.

Wiederum macht es Sinn, Werte im `.env` File abzuspeichern.

Hier ein Beispiel:

```
PORT=3000
SESSION_LIFETIME=2
SESSION_NAME=mysid
SESSION_SECRET=I love SEW and INSY
NODE_ENV=development
```

und diese dann als Konstante in `app.js` zu verwenden.

```
const {PORT, NODE_ENV, SESSION_LIFETIME, SESSION_NAME, SESSION_SECRET} = process.env;
```

Registrierung der Middleware:

```
app.use(
  session({
    secret: SESSION_SECRET,
    name: SESSION_NAME,
    saveUninitialized: false,
    resave: false,
    cookie: {
      maxAge: SESSION_LIFETIME * 1000 * 60 * 60,
      httpOnly: false,
      sameSite: false,
      secure: NODE_ENV === 'production',
    },
  }),
);
```

SESSION_SECRET: Ein String, der von **express-session** für das Signieren des Cookies verwendet wird. Im Endeffekt wird ein Hashwert aus dem Secret und den Daten des Cookies berechnet.

SESSION_NAME: Name für das Session Cookie. Der Standardwert ist `sid`.

saveUninitialized: Ob erzwungen werden soll, dass nicht initialisierte (neue, aber nicht geänderte) Sessions in den Speicher übertragen werden. Der Standardwert ist `true` (veraltet). Für Login Sessions ergibt es keinen Sinn, leere Sitzungen für nicht authentifizierte Anfragen zu speichern, da sie noch nicht mit wertvollen Daten verknüpft sind und somit Speicherplatz verschwenden. Wir speichern die neue Sitzung erst, wenn sich der Benutzer angemeldet hat.

¹ <http://expressjs.com/en/resources/middleware/session.html>

`resave`: Ob erzwungen werden soll, dass nicht geänderte Sessions in den Speicher zurückgeschrieben werden sollen. Standard ist `true` (veraltet). Wir speichern nur geänderte Sessions!

Soweit unsere Session Parameter. Die anderen Parameter sind Cookie Parameter. Siehe voriges Arbeitsblatt.

Unter `/models/users.js` sind bereits einige User eingetragen. Diese können wir für den Login Prozess verwenden. Wenn der User das Login Formular ausfüllt (C) und auf den Login Button drückt, soll er eingeloggt werden und eine personalisierte Home Page (D) angezeigt werden.

Dazu ist eine **POST** Route nötig, welche die übermittelten Daten mit den registrierten Usern vergleicht.

Aufgabe 4. Erstelle die **POST /api/login** Route.

Suche zunächst den User in dem Array aus `/models/users.js`.

Ist er vorhanden, dann setze die `userId` in dem Session Objekt, dass von **express-session** automatisch als Property vom `req` Objekt erzeugt wird!

```
if (!validateLogin(req.body)) return res.status(400).send('Your input has a wrong format!');
const { email, password } = req.body;
const user = users.find((el) => el.email === email && el.password === password);
if (user) {
  const { id, name } = user;
  req.session.userId = id;
  return res.status(200).json({ id, name });
}
res.status(401).end();
```

Wie du siehst, kommen vom Server `id` und `name` zurück. Diese sollen im Pinia Store `userStore.js` mit der Funktion `saveUserData` gespeichert werden.

Redirecte auf **AccountView** mit `router.push('/account');`

In **AccountView** hole dir die `id` aus dem Store, rufe in `onMounted` mit **GET /api/users/:id/secret** das Geheimnis auf. Setze die Variable und zeige sie an. Anmerkung: Die Server Route `.../secret` wird später abgesichert.

Um jedoch den Development Server verwenden zu können und relative Routen wie `/api/login` etc. am Client zu haben sowie das CORS Cookie Problem zu umgehen, stellen wir einen Proxy Server ein.

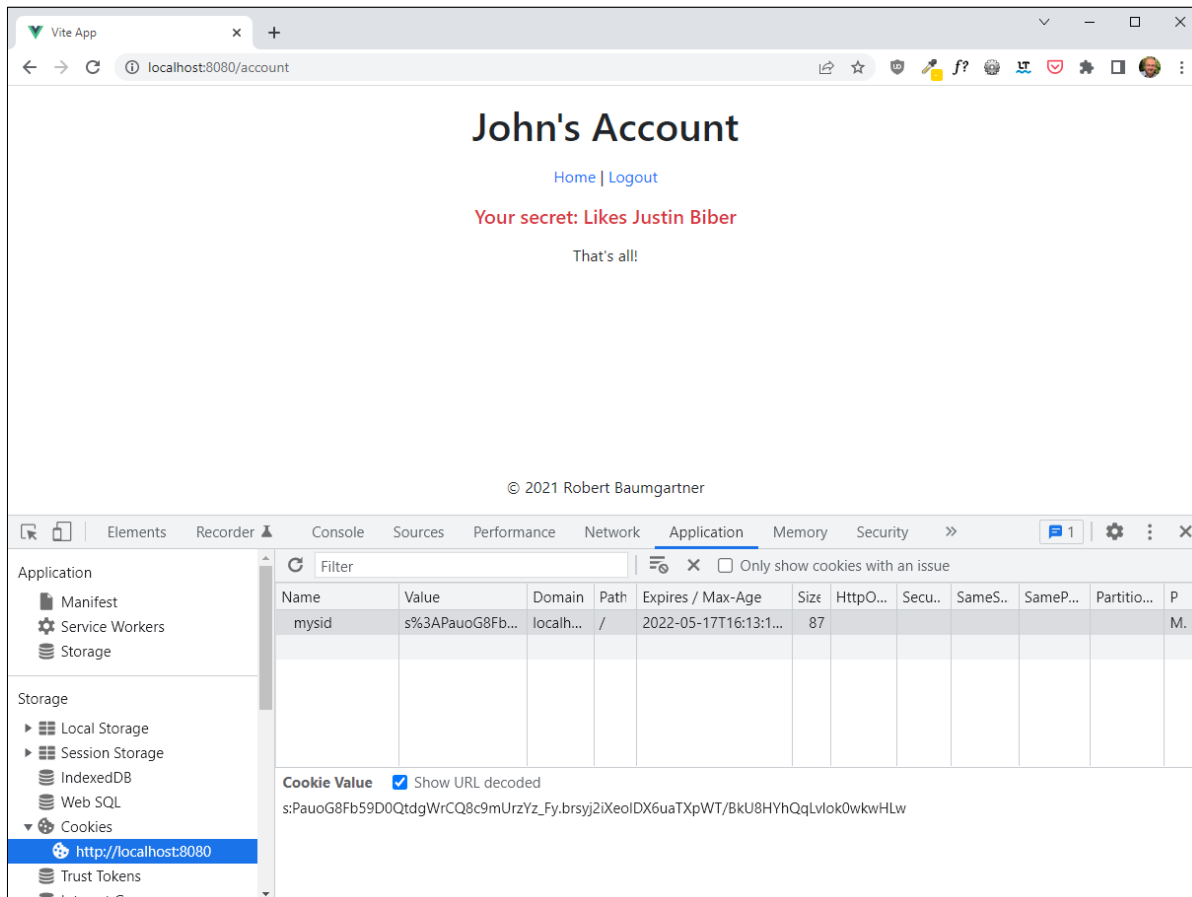
Füge dazu die folgenden Zeilen zu `vite.config.js` hinzu!

```
proxy: {
  '/api': {
    target: 'http://127.0.0.1:3000',
    secure: false,
  },
},
```

Überprüfe, ob alles gesetzt ist!

Erinnere dich: Das Cookie wird automatisch gesetzt und ist das Zeichen, dass der User eingeloggt ist.

Auch der Pinia Store sollte die Daten enthalten! Prüfe das mit den Dev Tools!

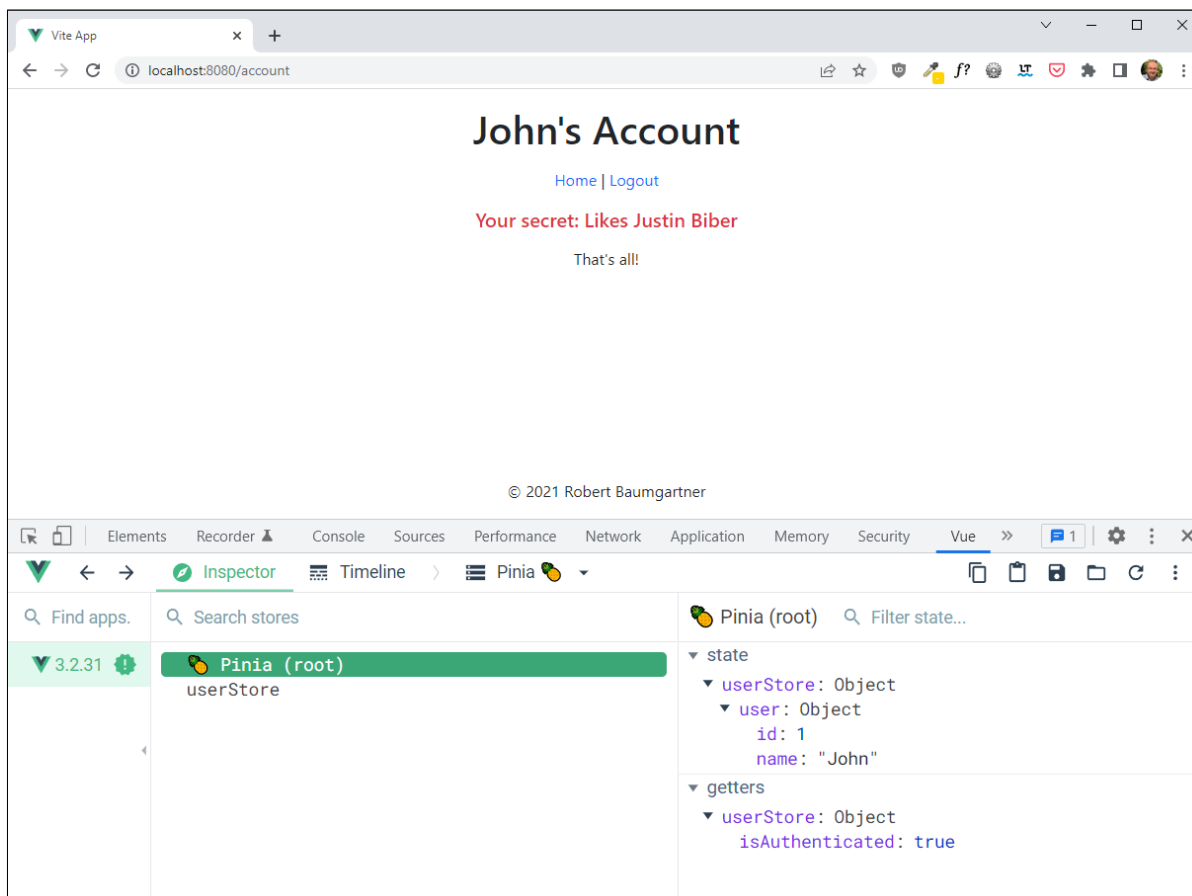


The screenshot shows a web browser at `localhost:8080/account` displaying "John's Account". The page has links for [Home](#) and [Logout](#), and a message: "Your secret: Likes Justin Biber". Below this, it says "That's all!". The footer shows "© 2021 Robert Baumgartner".

The developer tools are open to the **Application** tab. The left sidebar shows the **Storage** section with **Cookies** expanded, listing `http://localhost:8080`. The main pane shows a table of cookies:

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpO...	Secu...	SameS..	SameP...	Partitio...	P
mysid	s%3APauoG8Fb...	localh...	/	2022-05-17T16:13:1...	87						M.

Below the table, the **Cookie Value** is shown as `s:PauoG8Fb59D0QtdgWrCQ8c9mUrZyZ_Fy.brsyj2IXeolDX6uaTXpWT/BkU8HYhQqLvlok0kwHLw`, with the checkbox **Show URL decoded** checked.



The screenshot shows the same web browser at `localhost:8080/account`. The developer tools are now open to the **Vue** tab, specifically the **Pinia** store.

The **Pinia (root)** store is expanded, showing the following state:

```
state
└─ userStore: Object
  └─ user: Object
    ├── id: 1
    └─ name: "John"
getters
└─ userStore: Object
  └─ isAuthenticated: true
```

Aufgabe 5. Erstelle eine GET `/logout` Route.

```
req.session.destroy();  
res.clearCookie(process.env.SESSION_NAME);  
res.status(200).end();
```

In **LogoutView** rufe via Store die Action **logout** auf.

Aufgabe 6. Die **RegisterView** ist bereits fertig. Wir brauchen aber noch am Server die Route. Erstelle dazu am Server eine POST `/api/register` Route.

Überprüfe ob **email**, **password** und **name** ausgefüllt wurden (**validateRegister**) und suche nach der E-Mail Adresse in den Daten am Server.

Wenn sie nicht vorhanden ist, dann vergib die nächste **id** und speichere die Daten ab.

```
const id = Math.max(...users.map((el) => el.id)) + 1;
```

Sende **200 OK** an den Client. **'Congratulations! You are registered!'**

Im anderen Fall sende **409** **'The provided email already exists!'**.

Bei Fehlern in den Daten, sende **400** **'Your input has a wrong format!'**.

Nun müssen die Routen abgesichert werden. Dazu verwenden wir am Server eine Hilfsfunktion:

```
const redirectLogin = (req, res, next) => {  
  if (!req.session.userid) res.status(400).send('You are not logged in!');  
  else next();  
};
```

Diese prüft, ob die **userId** im Session Objekt gesetzt ist (dann ist der User eingeloggt).

Aufgabe 7. Verwende diese Funktion als Middleware, um die Routen abzusichern:

```
router.get('/users/:id/secret', redirectLogin, getSecret);
```

Jetzt die Client Routen. Verwende dazu den Getter **isAuthenticated** aus dem Store!

```
beforeEnter: (to, from, next) => {  
  const userStore = useUserStore();  
  if (!userStore.isAuthenticated) next({ name: 'Login' });  
  else next();  
},
```

Ein Letztes bleibt noch zu tun: Fange die nicht gefundenen Routen am Client im Router ab!

Verwende dazu die Methode aus dem Arbeitsblatt **Schenes Wean!**



Fertig. Im nächsten Arbeitsblatt verlegen wir den Session Store von Memory in die PostgreSQL Datenbank!