Clientseitige Implementierung

Grundlegende Struktur

Die clientseitige JavaScript Architektur folgt dem Prinzip der Seperation of Concers, wodurch hie die unterschiedlichen Verantwortungen klar voneinander getrennt sind.

Die für viele Funtkionalitäten benötigten Routen sind unter `src/routes` defniert und werden verwiesen unter `app.js`.

Die im Verzeichnis `public/js` befindlichen JavaScript Dateien `frontendBrowse.js`, `navbar.js`, `upload.js` und `validation.js`, gewährleisten mittels Einbindungen in die statischen HTML-Seiten im Verzeichnis `/static`, und über Kommunikationsschnittstellen mit dem Backend, ein angenehmes User-Interface und eine sinnvolle Interaktionssteuerung.

Login/Signup-Handling in `validation.js`

`validation.js` wird cleintseitig zur Abwicklung der Signup und Login Funktionen verwendet und ist eingebunden in `/static/login.html` und `/static/signup.html`.

Zunächst wird über einen EventListener geprüft ob, das HTML Dokument fertig geladen und geparst ist. Im Anschluss werden die Login- und Signup-Formulare über die jeweils festgelegte ID aufgerufen:

```
document.addEventListener('DOMContentLoaded', () => {
    const signupForm = document.getElementById('signup-form');
    const loginForm = document.getElementById('login-form');
    ...
});
```

Signup

Das Skript prüft zunächst, ob das signupForm-Element auf der Seite existiert. Falls ja, wird ein Event Listener hinzugefügt, der auf das submit-Ereignis des Formulars reagiert:

```
if (signupForm) {
      // event listener for the signup form
      signupForm.addEventListener('submit', async (e) => {
        e.preventDefault();
        ...
    });
}
```

Im Anschluss werden die Eingaben `firstName`, `email` und `password` aus dem Formular über die festgelgte ElementId ausgelesen:

```
const firstName = document.getElementById('firstname-input').value;
const email = document.getElementById('email-input').value;
const password = document.getElementById('password-input').value;
```

Innerhalb einer try-catch funktion werden nun die Usereingaben an das Backend gesendet, wobei über `catch` mögliche Serverfehler abgefangen werden.

Hierbei wird fetch() verwendet, um eine asynchrone POST-Anfrage an die URL `/api/users/signup` zu senden. Die Userdaten werden dabei in einen JSON-String umgewandelt, was über `headers:` dem Empfänger mitgeteilt wird.

`if(res.ok)` überprüft ob die Antwort von `/api/users/signup` zwischen 200-299 liegt, da Normalfall `res.status(200)` erhalten wird, erscheint dann die Mitteilung einer erfolgreichen Registrierung und der User wird auf die Loginseite weitergeleitet wird. Falls die Antwort nicht zwischen 200 und 299 liegt, wird eine Fehlermeldung ausgegeben mit der übermittelten Message.

Falls die Kommunikation mit `/api/users/signup` gänzlich fehltschlägt wird das über `catch` abgefangen und es erscheint ebenfalls eine Fehlermeldung:

```
try {
    const res = await fetch('/api/users/signup', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ firstName, email, password })
          });

    const data = await res.json();

    if (res.ok) {
        alert('Registrierung erfolgreich!');
        window.location.href = './login';
    } else {
        alert(`Fehler: ${data.message}`);
    }
}   catch (error) {
    alert('Serverfehler!');
    console.error('Fehler beim Signup:', error);
}
```

Login

Die Kommunikation des Login funktioniert nach dem gleichen Muster wie beim Signup, mit der Ausnahme, das hier `email` und `password` ausreichend sind, und der `firstName` nicht übertragen wird

Das Skript prüft zunächst, ob das loginForm-Element auf der Seite existiert. Falls ja, wird ein Event Listener hinzugefügt, der auf das submit-Ereignis des Formulars reagiert:

```
if (loginForm) {
      // event listener for the signup form
      loginForm.addEventListener('submit', async (e) => {
        e.preventDefault();
        ...
});
}
```

Im Anschluss werden die Eingaben `email` und `password` aus dem Formular über die festgelgte ElementId ausgelesen:

```
const email = document.getElementById('email-input').value;
const password = document.getElementById('password-input').value;
```

Innerhalb einer try-catch funktion werden nun die Usereingaben an das Backend gesendet, wobei über `catch` mögliche Serverfehler abgefangen werden.

Hierbei wird fetch() verwendet, um eine asynchrone POST-Anfrage an die URL `/api/users/login` zu senden. Die Userdaten werden dabei in einen JSON-String umgewandelt, was über `headers:` dem Empfänger mitgeteilt wird. Zudem zeigt `credentials: 'include'` dem Backend, dass der Client noch keine Cookies hat, da sie dadurch normalerweise mitgesendet würden, allerdings noch nicht vorhanden sind. Zusätzlich akzeptiert der Browser aufgrunddessen Cookies, die vom Server gesendet werden. Als Resultat erstellt der Server ein Cookie, der dem Client gesendet wird und bei zukünftigen Anfragen an den Server mitgesendet wird.

`if(res.ok)` überprüft ob die Antwort von `/api/users/signup` im 200er Bereich liegt, da Normalfall `res.status(200)` erhalten wird, erscheint dann die Mitteilung eines ergolgreichen Logins und der User wird auf die `./homepage` weitergeleitet. Zudem wird der firstName im `localStorage` des Browsers gespeichert.Falls die Antwort nicht zwischen 200 und 299 liegt, wird eine Fehlermeldung ausgegeben mit der übermittelten Message.

Falls die Kommunikation mit `/api/users/signup` gänzlich fehltschlägt wird das über `catch` abgefangen und es erscheint ebenfalls eine Fehlermeldung:

```
try {
    const res = await fetch('/api/users/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        credentials: 'include',
        body: JSON.stringify({ email, password })
    });

    const data = await res.json();

    if (res.ok) {
        localStorage.setItem("firstName", data.firstName);
        alert("Login erfolgreich!");
        window.location.href = "./homepage";
    } else {
        alert(`Fehler: ${data.message}`);
    }
} catch (error) {
    alert('Serverfehler!');
    console.error('Fehler beim Login:', error);
}
```

Verarbeitung des Uploads in `upload.js`

Das Dokument `upload.js` ist im Rahmen der Dateiverarbeitungslogik dafür zuständig, die vom User ausgewählte Datei zusammen mit vom User gewähltem Titel, Beschreibung, Tag an den Server zu übermitteln, damit dieser das File dann in der Mongo-Datenbank speichern kann. Einbindung: `upload.js` ist in `share.html` eingebunden.

Zunächst wird ein EventListener für den Upload-Button erstellt, der auf das Event `'click'` reagiert. Falls das Event eintrit, werden der Titel und die Beschreibung, ausgewählte File sowie der Tag ausgelesen. Falls es einen Tag gibt, wird außerdem, der dazugehörige value erfasst (hier:Exercises, Summary, Scribbled Notes) Sollte kein Tag ausgewählt sein, erscheint ein Altert mit der Aufforderung, einen Tag auszuwählen:

```
document.getElementById('upload-btn').addEventListener('click', async function(event)
{
  event.preventDefault();

  // collect form data
  const documentTitle = document.getElementById('title').value;
  const documentDescription = document.getElementById('description').value;
  const uploadFile = document.getElementById('upload').files[0];

  // collect tag
  const selectedTag = document.querySelector('input[name="tag"]:checked');
  if (!selectedTag) {
    alert('Bitte wähle einen Tag aus.');
    return;
  }
  const tag = selectedTag.value;
})
```

Für die Fehlerbehandlung in der Kommunikation wird `try-catch` verwendet. Zunächst wird eine GET-Anfrage an dei Route `/api/users/status` gesendet, wodurch überprüft wird, ob der User eingeloggt ist. Die Antwort wird dass in ein JavaScript-Obejekt umgewandelt.

```
const response = await fetch('/api/users/status');
const status = await response.json();
```

Die Antwort des Servers könnte beispielsweise so aussehen:

```
{
  "loggedIn": true,
  "user": {
    "id": "12345",
    "firstName": "Bob"
  }
}
```

Falls der Benutzer nicht eingeloggt, wird ein Alert mit `'Du bist nicht eingeloggt'` generiert.

```
alert('Du bist nicht eingeloggt.')
```

Ist der Benutzer allerdings eingeloggt, wird ein FormData-Objekt erstellt, um die Daten für den Upload zu speichern. Infolgedessen wird dieses Objekt per POST-Anfrage an die API-Route `/api/upload` gesendet, wo es über weitere Verarbeitungsschritte in der Datenbank gespeichert wird:

```
if (status.loggedIn) {
      const userID = status.user.id;
```

```
      // create form data object containing all necessary information
      const formData = new FormData();
      formData.append('docTitle', documentTitle);
      formData.append('description', documentDescription);
      formData.append('uploadFile', uploadFile);
      formData.append('tag', tag);
      formData.append('userID', userID);

      // send POST request with fromData to /api/upload
      const uploadResponse = await fetch('/api/upload', {
        method: 'POST',
        body: formData
      });
      ...
}
```

In Anbhängigkeit der Antwort von der API-Route werden Alerts über den Status des Hochladens generiert:

```
if (uploadResponse.status === 200) {
    alert('Dokument erfolgreich hochgeladen!');
    document.getElementById('upload-form').reset(); // Formular zurücksetzen
} else if (uploadResponse.status === 400) {
    alert('Fehler: UserID nicht gefunden.');
} else {
    alert('Serverfehler. Bitte versuche es später erneut.');
}
```

Sollte die Kommunikation mit dem Server fehlschlagen, wird der `catch` ausgelöst:

```
} catch (error) {
    console.error('Fehler:', error);
    alert('Netzwerkfehler. Überprüfe deine Verbindung.');
}
```

Anpassung der Navigationsleiste, wenn Benutzer eingeloggt in `navbar.js`:

Anpassung Navigationsleiste

Mittels des Dokuments `navbar.js` wird in die Navigationsleiste nach erfolgreichem Einloggen ein Logout-Button integriert. Dies geschieht auf allen statischen HTML-Seiten. Ebenso wird der logout hier über die Schnittstelle `/api/users/logout` abgewickelt und eine GateKeeper Funktion sorgt dafür, dass nur eingeloggt Benutzer auf die Share-Seite gelangen.

Zu Beginn stellt der DOMContentLoaded-Event Listener sicher, dass die Seite fertig geladen ist, bevor das Sktipt ausgeführt wird. Darauffolgend wird durch eine GET-Anfrage überprüft ob der Nutzer eingeloggt ist, wobei die Antowort des Servers in JSON geparst wird:

```
fetch('/api/users/status', { credentials: 'include' })
  .then(response => response.json())
```

```
  .then(data => { ... })
  .catch(err => console.error("Error fetching user status:", err));
```

Eine Antwort des Servers kann beispielsweise so aussehen:

```
{
  "loggedIn": true,
  "user": {
    "id": "12345",
    "firstName": "Bob"
  }
}
```

Bei erfolgreichem Login und wenn das DOM-Element `user-menu` existiert, wird das `user-menu` ersetzt durch eine personalisierte Bergrüßung: `Hallo {firstName}` und einen Logout-Button:

```
if (data.loggedIn && userMenu) {
    userMenu.innerHTML = `
        <div class="user-hover">
        <span class="greeting">Hallo ${data.user.firstName}</span>
        <span class="logout">Logout</span>
        </div>
    `;
    ...
}
```

Logout-Verarbeitung

Auf den Logout im darauffolgend ein EventListener gesetzt, der beim klick auf Logout eine POST-Anfrage an `/api/users/logout` sendet und die Cookies mitsendet. Wenn die Antwort des Servers im 200er Bereich liegt, wird der User auf die Login-Seite navigiert:

```
const logoutEl = userMenu.querySelector(".logout");
logoutEl.addEventListener("click", async () => {
    try {
    const res = await fetch('/api/users/logout', {
        method: 'POST',
        credentials: 'include'
    });
    // if successful logout, redirect to login page
    if (res.ok) {
        window.location.href = "/login";
    }
    } catch (error) {
    console.error("Logout Error:", error);
    }
});
```

GateKeeper-Funktion für den Share-Button

Die GateKeeper-Funktion überprüft beim betätigen des `share-btn` in der Navigationsleiste ober Benutzer eingeloggt ist. Das ist deshalb relevant, da beim

Hochladen einer Datei eine UserID erwartet wird. Dieser Schutzmechanismus koexistiert mit einer Funktion in `/src/routes/share.js`, die von der Backendseite ebenfalls gewährleisten soll, dass nur eingeloggte Nutzer auf die Share-Seite geroutet werde. Da clientseitige Beschränkungen umgangen werden können, haben wir uns entschieden, dies sowohl im Front- als auch im Backend zu überprüfen.

Per GET-Anfrage wird an die Route `api/users/status` die Anfrage gestellt ob der Benutzer eingeloggt ist und die Antwort wird in JSON geparst. Siehe Oben, wie eine Mögliche Antwort vom Server aussehen kann.

Nachfolgend wird die Antwort überprüft. Ist der Benutzer eingeloggt, wird zu `/share` navigiert, ansonsten zur Login-Seite. Sollte ein Fehler auftreten, der durch try-catch abgefangen wird, wird ebenfalls auf die Login-Seite navigiert:

```
const shareButton = document.getElementById("share-btn");
  if (shareButton) {
    shareButton.addEventListener("click", async (e) => {
      e.preventDefault();
      try {
        // check if user is logged in
        const response = await fetch('/api/users/status', { credentials: 'include' });
        const data = await response.json();
        // grant acess to /share page if user is logged in
        if (data.loggedIn) {
          window.location.href = "/share";
        //if user is not logged in, redirect to login page
        } else {
          window.location.href = "/login?redirect=share";
        }
      } catch (error) {
        console.error("Fehler beim Prüfen des Login-Status:", error);
        window.location.href = "/login?redirect=share";
      }
    });
  }
```

Suchfunktion über `frontendBrowse.js`

Grundfunktionen und Suchanfrage an das Backend

Das Skript `frontendBrowse.js` trägt mit dazu bei, ein anschauliches und funktinieredes User Interface für eine Dokumentensuche zu generieren.

Anfänglich wird ein EventListener auf das Event `DOMContentLoaded` angewandt und auf die DOM-Elemente `search-form` und `search-results` des HTML-Dokumts `browse.html` zugegriffen. `searchForm` verweist auf das Suchformular, `resultsSection` auf den Abschnitt, in dem die Suchergebnisse angezeigt werden sollen und bekommt die CSS-Klasse `browse-card-container`:

```
document.addEventListener("DOMContentLoaded", () => {
  const searchForm = document.getElementById("search-form");
  const resultsSection = document.getElementById("search-results");
  ...
});
```

Um die Suchparameter abzugreifen, wird ein EventListener auf das Event `'submit'` der `searchForm` gelegt, wodurch bei eintreten des Events die searchForm in eine neues FormData-Objekt verpackt, woraus dann der vom User ausgewählte Tag und Suchbegriff extrahiert werden:

```
searchForm.addEventListener("submit", async (e) => {
    e.preventDefault();
    const formData = new FormData(searchForm);
    const searchTerm = formData.get("searchTerm");
    const tag = formData.get("tag");
...
}
```

Aus den extrahierten Suchparametern wird nun eine URL gebaut:

```
let url = `/browse?searchTerm=${encodeURIComponent(searchTerm)}`;
if (tag) {
    url += `&tag=${encodeURIComponent(tag)}`;
}
```

Im Anschluss wird über `fetch()` eine GET-Anfrage an die URL gesendet, wo die Suchparameter aus der URL extrahiert werden und eine Datenbank abfrage gestartet wird. Die Antwort wird in JSON geparst:

```
try {
    const response = await fetch(url);
    const data = await response.json();
...
}
```

Eine mögliche *vollständige* Antwort des Servers an den Client könnte wie folgt ausssehen, wenn 2 Dokumente zu den Suchparametern `searchTerm`: "JavaScript" und `tag`: "exercises" gefunden werden:

```
res.status(200).json({
  numDocs: 2,
  documents: [
    {
      docID: "645a7e4b8f5d4e1f34b7a9c1",
      docTitle: "Learn JavaScript",
      docDescription: "A beginner's guide to JavaScript programming.",
      docTag: "scribbledNotes",
      docAuthor: "Marc",
      docDate: "2025-02-24T09:23:16.126Z"
    },
    {
      docID: "645a7e4b8f5d4e1f34b7a9c2",
      docTitle: "Advanced JavaScript",
      docDescription: "Deep dive into JavaScript ES6 and beyond.",
      docTag: "summary",
      docAuthor: "Bob",
      docDate: "2025-01-24T09:22:12.126Z"
    }
```

```
  ]
});   const data = await response.json();
```

Bevor die Antwort verarbeitet wird, wird die `resultssection` gecleared, damit es bei aufeinanderfolgenden Suchen keine Konflikte gibt.

Erstellen der Browse-Cards

Werden keine Dokumente gefunen (`data.numDocs === 0`), wird der Schriftzug `"No documents found."` in der resultsSection angezeigt. Andernffalls erstellt das Skript für jedes Dokument ein `"div"` -Element und weist die CSS-Klasse `"browse-grid-container"` zu. Innerhalb des `"div"` -Elements wird nun ein Bild eingefügt, das als Platzhalter dient, für eine mögliche spätere Funktion der Dokumentenvorschau, die wir uns als Zukunftsausblick vorbehalten möchten. Zudem werden der Dokumententitel, die Beschreibung und der Autor angezeigt. Das Uploaddatum wird zum jetztigen Zeitpunkt nicht angezeigt, ist allerdings in der Antwort des Servers vorhanden, um es gegebenenfalls zu einem späteren Zeitpunkt einfach integrieren zu können.

Diese `"div"` -Elemente werden anchließend der resultsSection zugewiesen, also dem Bereich, im HTML-Dokument `browse.html` , wo die Ergebnisse angezeigt werden sollen und jedes `"div"` -Element bekommt einen Download-Button zugewiesen, beidem die `docID` als `data-id` mitgegeben wird..

```
try {
    const response = await fetch(url);
    const data = await response.json();

    resultsSection.innerHTML = "";

    if (data.numDocs === 0) {
        resultsSection.innerHTML = "<p>No documents found.</p>";
    } else {
        data.documents.forEach(doc => {
            const card = document.createElement("div");
            card.classList.add("browse-grid-container");

            card.innerHTML = `
            <div class="browse-card">
            <img src="../public/img/browse_placeholder.png" alt="Search result" />
            <p>${doc.docTitle}</p>
            <p>${doc.docDescription}</p>
            <p>Author: ${doc.docAuthor}</p>
            </div>
            <button class="download-btn" data-id="${doc.docID}">Download</button>
            `;
            resultsSection.appendChild(card);
    });
    }
} catch (err) {
    console.error("Error fetching search results:", err);
    resultsSection.innerHTML = "<p>Error fetching search results.</p>";
}
```

Download

Damit der Benutzer einen Download durchführen kann, wird ein EventListener auf sämtliche `"click"`-Events in der `resultsSection` erstellt. Zusätzlich wird geprüft, ob das `"click"`-Event auf ein Element mit der Klasse `"download-btn"` stattfindet, da nur die aus den Suchergebnissen erstellten `"div"`-Elemente dieser Klasse zugewiesen sind. Wird also eines der vorher erstellen `"div"`-Elemente angeclickt, wird die docID dieses spezifischen Elements ausgelesen und der Client wird zu `/browse/download?docID=${docID}` navigiert, wodurch die `docID` durch das Backend (hier: `/src/routes/browse.js`) genutzt wird um das ausgewählte Dokument zurückzusenden, was den Download auslöst.

```
resultsSection.addEventListener("click", (e) => {
    if (e.target.classList.contains("download-btn")) {
      const docID = e.target.getAttribute("data-id");
      window.location.href = `/browse/download?docID=${docID}`;
    }
});
```