StudyBuddy

Inhalts- und Dateiverzeichnis

▶ Dateiverzeichnis

```
.
├── db
│   └── StudyBuddy
│       ├── docs.bson
│       ├── docs.metadata.json
│       ├── documents.bson
│       ├── documents.metadata.json
│       ├── users.bson
│       └── users.metadata.json
├── documentation
│   ├── img
│   │   ├── 1_Erster_Entwurf.png
│   │   ├── 2_Sitemap.png
│   │   ├── 3_KompDia_Homepage.png
│   │   ├── 4_KompDia_Login.png
│   │   ├── 5_KompDia_Signup.png
│   │   ├── 6_KompDia_Impressum.png
│   │   ├── 7_KompDia_Browse.png
│   │   ├── 8_KompDia_Share.png
│   │   ├── browse.png
│   │   ├── download.png
│   │   ├── image-1.png
```

```
│   │   ├── image-10.png
│   │   ├── image-11.png
│   │   ├── image-12.png
│   │   ├── image-2.png
│   │   ├── image-3.png
│   │   ├── image-4.png
│   │   ├── image-5.png
│   │   ├── image-6.png
│   │   ├── image-7.png
│   │   ├── image-8.png
│   │   ├── image-9.png
│   │   ├── image.png
│   │   └── upload.png
│   ├── StudyBuddy.md
│   └── StudyBuddy.pdf
├── public
│   ├── css
│   │   └── style.css
│   ├── img
│   │   ├── browse_placeholder.png
│   │   ├── favicon-96x96.png
│   │   ├── favicon.svg
│   │   ├── group_picture.png
│   │   ├── hp_card1.jpeg
│   │   ├── hp_card2.jpeg
│   │   ├── login_email_icon.svg
│   │   ├── login_lock_icon.svg
│   │   ├── login_person_icon.svg
│   │   ├── logo.png
│   │   └── search.png
│   └── js
│       ├── frontendBrowse.js
│       ├── navbar.js
│       ├── upload.js
│       └── validation.js
├── src
│   ├── models
│   │   ├── docModel.js
│   │   └── userModel.js
│   ├── routes
│   │   ├── browse.js
│   │   ├── homepage.js
│   │   ├── impressum.js
│   │   ├── login.js
│   │   ├── share.js
│   │   ├── signup.js
│   │   ├── uploadRoute.js
│   │   └── userRoutes.js
│   └── app.js
├── static
│   ├── browse.html
│   ├── homepage.html
│   ├── impressum.html
│   ├── login.html
```

```
│   ├── share.html
│   └── signup.html
├── Installationsanleitung.md
├── README.md
├── StudyBuddy.pdf
├── package-lock.json
└── package.json
```

Frontend: HTML und CSS

Mithilfe von HTML und CSS können die Struktur, der Inhalt und das Design von Webseiten erstellt werden. Beide Sprachen arbeiten Hand in Hand, um einen ansprechenden und funktionalen Web-Auftritt zu erstellen.

Dieses Kapitel beschreibt den Entwicklungsprozess von StudyBuddy hinsichtlich der Sprachen HTML und CSS. Es betrachtet die Entstehung von der Zieldefinition über den ersten Webseitenentwurf bis hin zur finalen Gestaltung. Zudem werden die eingesetzten Hilfsmittel beschrieben, die diesen Prozess begleitet haben. Weiterhin wird anhand ausgewählter Code-Beispiele die Funktionsweise des Codes erläutert.

Zieldefinition

Für das Design der Wep-Applikation "StudyBuddy" hat sich das Team folgende Ziele gesetzt, damit die Bedürfnisse der Zielgruppe Schüler und Studenten*innen optimal erfüllt werden:

- **Übersichtlichkeit und Klarheit:**
  Zweck der Website ist u. A. Schüler*innen und Studierende beim Lernen zu unterstützen. Dabei kann das Lernen für Prüfungen mit unter eine emotionale Belastung darstellen. Sei es der Leistungsdruck, der gefühlte Zwang sich mit Themen auseinander zu setzen, für die man wenig bis kein Interesse aufbringen kann oder die Überforderung bei dem Versuch ein neues Thema zu verstehen, das unbegreiflich scheint.
  Aus diesem Grund soll StudyBuddy eine übersichtliche und klare Gestaltung haben, damit die Applikation einfach zu bedienen ist und keinen weiteren Stress im Lernprozess bedingt.

- **Ansprechendes, durchgängiges Farbdesign:**
  Die farbliche Gestaltung des Web-Auftritts soll eine stressfreie, motivierende und konzentrationsfördernde Atmosphäre schaffen. Deshalb fokussiert sich die Farbauswahl auf Blau- und Lilatöne. Diese zeichnen sich nach der PAD-Theorie[^1] durch eine beruhigende physiologische und emotionale Stimulation, sowie durch eine angenehme Farbwahrnehmung aus.[^2]
  Weiterhin wurden im :root-Element Farben für das User-Interface definiert, die auf den Webseiten angewendet werden. Dadurch ergibt sich ein einheitliches und konstantes Farbbild.

- **Flexible Navigation:** Den Nutzer*innen soll eine flexible Navigation zwischen der Hauptseite und den Unterseiten ermöglicht werden. Dadurch soll eine möglichst effiziente Nutzung der Website gewährleistet werden.

[^1]: Die PAD-Theorie (Pleasure-Arousal-Dominance-Theorie) aus der Farbpsychologie beschäftigt sich mit der Wirkung von Farben. Pleasure beschreibt, wie angenehm oder unangenehm eine Farbe empfunden wird. Arousal zeigt die physiologische und emotionale Stimulation. Dominance stellt dar, wie stark eine Farbe ein Gefühl von Kontrolle oder Macht vermittelt.

[^2]: vgl. Valdez, P., & Mehrabian, A. (1994). Effects of color on emotions. Journal of Experimental Psychology: General, 123(4), 394–409

Erster Entwurf

Zu Beginn des Projekts wurde ein erster Entwurf der Homepage erstellt. Dafür wurde das Design-Programm Figma genutzt. Damit war es möglich, als Team gemeinsam an einem Entwurf zu arbeiten.

Abbildung 1 stellt den ersten Entwurf dar.

Bereits beim initialen Design wurde die Auffassung geteilt, dass die Homepage möglichst übersichtlich gestaltet sein soll. Dazu gehörte, dass sich das Design auf die Funktionen Up- und Download (später umbenannt in Share und Browse) ausrichtet, denn diese bilden die Hauptfunktionen der Webapplikation.

Weiterhin wurde sich für den Namen "StudyBuddy" entschieden, da die Applikation genau dies für ihre Nutzenden darstellen soll - eine angenehme, positive Unterstützung beim Lernen. Außerdem impliziert "Buddy" - im Sinne von Freund - dass man mit anderen Lernenden zusammenarbeiten kann, indem man seine Lernunterlagen austauscht.

Abweichend vom ursprünglichen Entwurf wurde die Farbauswahl getroffen. Wie in der Einleitung beschrieben, wurden Blau- und Lilatöne als Hauptfarben definiert.
Zudem wurden Anpassungen bei den Bildern vorgenommen. Wie in Abbildung 1 zu sehen ist, wirken die ersten Grafiken tendenziell kalt und wenig ansprechend. Deshalb wurden mit Microsoft Designer neue Bilder generiert.
Eine weitere Veränderung ist, dass die Verlinkung zu "About Us", also dem Impressum, nicht mehr im Header zu finden ist. Da diese Unterseite keine der Hauptfunktionen von StudyBuddy beinhaltet, wurde die Verknüpfung im footer der Webseiten aufgenommen.

Hiermit wurde der initiale Entwurf des Webseiten-Designs aufgezeigt. Im folgenden Teil wird auf die Erstellung der Struktur und des Inhalts mittels HTML eingegangen. Anschließend wird die Gestaltung mit CSS dokumentiert.

Hypertext Markup Language (HTML)

Die Hypertext Markup Language (HTML) ist eine standardisierte Sprache zur Erstellung von Webseiten. Mit ihr können die Struktur und die Inhalte einer Webseite programmiert werden.

Dieses Kapitel beschäftigt sich mit der HTML-Programmierung von StudyBuddy. Dabei wird zuerst auf die Sitemap der Webapplikation eingegangen, um ein grundsätzliches Verständnis für die Verknüpfung der Webseiten zu schaffen. Anschließend wird deren Aufbau mittels Kompositionsdiagramme dargestellt. Abschließend wird anhand eines Code-Beispiels die Funktionsweise aufgezeigt.

Sitemap

Abbildung 2 zeigt die Sitemap von StudyBuddy.

Die Homepage stellt den Ausgangspunkt der Webapplikation dar. Von dort ermöglicht die Struktur eine bidirektionale Navigation zwischen den Unterseiten "Login", "Impressum", "Browse" und "Share". Eine Ausnahme von der vollständigen Verlinkung stellt die Seite "Signup" dar. Diese ist nur vom "Login" aus zu

erreichen. Dieser Aufbau wurde gewählt, da der "Signup" aus Sicht des Users nur einmal aufgerufen werden muss und bei der anschließenden Nutzung der "Login" ausreichend ist.

Kompositionsdiagramme

Kompositionsdiagramme dienen dazu, den Inhalt und die Struktur der Webseiten zu visualisieren.

Die Darstellung erfolgt in Form von rechteckigen Containern. Der grundlegende Container ist meist das body-Element, da dieses das Fundament des sichtbaren Bereiches einer Webseite ist. In jedem Container können weitere Sub-Container platziert werden, die wiederum untergeordnete Container enthalten können.

Ziel ist es, den Aufbau der HTML-Elemente und deren Platzierung zu- bzw. ineinander grafisch darzustellen.

Der Webauftritt von StudyBuddy zeichnet sich dadurch aus, dass alle Webseiten innerhalb des body-Elements die Subelemente "header", "main" und "footer" haben. Dadurch wird ein einheitliches Design geschaffen und der Wiedererkennungswert für die Nutzenden erhöht.

Eine weitere Gemeinsamkeit der Homepage und der Unterseiten "Impressum", "Browse" und "Share" ist die Sektion der Klasse "introduction". Diese Sektion befindet sich zu Beginn des main-Elements. Aufgrund der Gestaltung dieser Klasse erhalten diese Webseiten eine kohärent designte Einleitung bzw. Überschrift.

Neben den Gemeinsamkeiten haben die Webseiten einige individuelle Strukturen. Diese werden im anschließenden Teil genauer beleuchtet.

Kompositionsdiagramm Homepage

Die Abbildung 3 zeigt das Kompositionsdiagramm der Homepage.

Neben des bereits beschriebenen Aufbaus enthält das main-Element ein svg-Element und eine Sektion der Klasse "hp-main-container". Die skalierbare Vektorgrafik (svg) stellt einen pinken Pfeil dar, der den Fokus des Webseitenbesuchers auf den Browse-Button lenkt. Dadurch soll der Nutzende zur Interaktion mit diesem Button animiert werden.

Weiterhin umfasst die Sektion "hp-main-container" 4 Elemente: zwei Buttons und zwei Bilder. Bei den Buttons handelt es sich um Verlinkungen der Hauptfunktionen "Browse" und "Share", während die Grafiken zur optischen Aufwertung eingesetzt werden. Das Ziel dieser Struktur ist es, ein Elternelement für die Buttons und Bilder zu schaffen, welches sich als CSS-Flexbox zur Anordnung der Inhalte nutzen lässt.

Die Homepage soll insgesamt als ansprechende Aufmachung der WebApplikation dienen und den Nutzenden umgehend zu einer weiteren Interaktion einladen.

Kompositionsdiagramm Login und Signup

Die Struktur des Logins und des Signups lassen sich der Abbildung 4 und Abbildung 5 entnehmen.

Die beiden Webseiten enthalten in dem main-Element ein Formular und eine Verlinkung untereinander.

Innerhalb des form-Elements wird beim Login die E-Mail-Adresse und das Passwort eines Nutzenden abgefragt. Diese Informationen werden benötigt, um eine Identifizierung und den damit verbundenen Login zu ermöglichen.

Das Formular der Signup-Seite unterscheidet sich insofern als dass das Login-Formular um die beiden Eingabe-Elemente "firstname" und "repeat password" erweitert wird. Diese Daten werden einmalig bei der Erstellung eines neuen Users erfasst. Der Vorname des Users wird benötigt, um diesen nach dem Login im Header anzuzeigen. Die Wiederholung des Passworts dient dazu, Tippfehlern im Passwort bei dessen Initialisierung entgegenzuwirken.

Die Verlinkung der beiden Webseiten untereinander wird durch die Einbettung eines Anker-Elements innerhalb eines Paragrafen realisiert.

Die beiden Seiten haben den Zweck, neue Nutzer*innen für die Webapplikation zu erfassen und diese anschließend zu identifizieren.

Kompositionsdiagramm Impressum

Die Abbildung 6 zeigt das Kompositionsdiagramm des Impressums.

Der Aufbau des Impressums ist relativ schlicht gehalten.
Innerhalb des main-Elements wird nach der Einleitung ein sektion-Element mit der Klasse "impressum-container" eingebettet. Dieses enthält eine h2-Überschrift und drei Paragrafen in Form von p-Elementen.

Durch diese Struktur wird ein übersichtliches Impressum mit einem klaren Fokus auf die enthaltenen Kontaktdaten geschaffen.

Kompositionsdiagramm Browse

Die Struktur der Browse-Seite wird in der Abbildung 7 dargestellt.

Hier folgt auf die Einleitungssektion ein div-Element der Klasse "browse-search". Dieses enthält ein Formular, das mithilfe von CSS als Suchleiste gestaltet wird.

Anschließend wird ein div-Element mit der Klasse "browse-container" erzeugt. Dieses umfasst eine Seitenleiste und einen Hauptbereich.

Die Seitenleiste wird durch ein div-Element dargestellt und bietet Filteroptionen für die Suchergebnisse. Zur Auswahl stehen die Tags, die beim Hochladen einer Datei festgelegt werden können.
Hier ist besonders zu erwähnen, dass sowohl die Suchleiste als auch die Seitenleiste dasselbe Form-Attribut mit dem Wert "search-form" haben. Dadurch ist es möglich, den Suchbegriff und die gewählten Filter innerhalb einer GET-Request zu übermitteln.

Im Hauptbereich werden die Suchergebnisse ausgegeben. Dazu werden dynamisch - in Abhängigkeit der Suchtreffer - div-Elemente über ein Skript erstellt.

Kompositionsdiagramm Share

Die Abbildung 8 zeigt das Kompositionsdiagramm der Share-Seite.

Die Share-Seite hat den Zweck, dass Nutzer*innen neue Lernunterlagen hochladen können. Deshalb enthält das main-Element einen Container ("share-container") mit einem Upload-Formular. In diesem können Dateiname, Beschreibung und Tags der Lernunterlage definiert werden. Bei der Auswahl der Tags kann nur eine der drei Optionen "exercises", "summary" und "scribbeld Notes" auf eine Lernunterlage zutreffen.

Deshalb werden die Tags über Radiobuttons abgefragt, die nur die Auswahl einer Option zulassen. Weiterhin wird ein input-Element mit dem Attribut type="file" genutzt, um das Hochladen einer Datei zu ermöglichen. Abschließend kann der Upload über button-Element abgeschlossen werden.

Die Share-Seite stellt neben der Browse-Seite eine Hauptfunktion von StudyBuddy dar. Deshalb ist auch hier der Fokus auf der Benutzerfreundlichkeit. Diese steht allerdings im Konflikt mit dem Ziel eine möglichst gute Informationslage zu der hochgeladenen Datei zu erfassen. Somit musste abgewogen werden, wie viele Details vom User zu einer Datei abgefragt werden können, bevor sich dieser gegen einen ggf. aufwändigen Upload entscheidet würde. Mit der aktuellen Lösung wird versucht, den beiden Zielen bestmöglich gerecht zu werden.

Cascading Style Sheet (CSS)

CSS bietet die Möglichkeit, eine HTML-Struktur visuell zu gestalten. Dadurch kann einer Webseite ein gewünschtes Design gegeben werden.

In diesem Teil wird dokumentiert, wie der CSS-Code von StudyBuddy aufgebaut ist und wie sich dieser auf die HTML-Dokumente auswirkt. Die Funktionsweise wird anhand eines Code-Beispiels ausführlich erläutert.

Aufbau des Codes

Im ersten Teil des CSS-Codes werden Gestaltungsmerkmale festgelegt, die auf alle oder einige Webseiten Einfluss haben. Dazu gehören das :root-Element, der Header der Webseiten, das main-Element inkl. der Klasse ".introduction" sowie der footer.

Anschließend wird die Webseiten-spezifische Gestaltung codiert. Dazu wird immer für eine Unterseite das komplette Design dieser Webseite gestaltet, bevor auf die Nächste eingegangen wird. Die Elemente und Klassen einer Seite werden in der Reihenfolge gestaltet, in der sie in den zugehörigen HTML-Dateien erstellt werden.

Im Anschluss an das Webseitendesign folgen die Anpassungen für das Responsive Design. und die Programmierung von Animationen.

Für den gesamten CSS-Code gilt: Innerhalb eines Selektors sind die Eigenschaft-Wert-Paare alphabetisch nach der Eigenschaft sortiert.

Es ergibt sich nachstehende Grob-Gliederung:

```
1 :root
2 all websites
    2.1  header
        2.1.1 header logo
        2.1.2 header navigation
    2.2 main incl. main-introduction
    2.3 footer
3 homepage
4 login/signup
5 impressum
6 browse
    6.1 browse searchbar
```

```
        6.2 browse sidebar
        6.3 browse main
    7 share
    8 responsive design
    9 animations
```

Erläuterung eines Code-Beispiels

Einen besonderen Gestaltungsaufwand haben die Buttons der Klasse ".hp-button" mit sich gebracht. Daher dient diese Klasse als exemplarisches Code-Beispiele zur Erläuterung eines CSS-Designs.

Auszug HTML-Code zu .hp-button

```html
<button class="hp-button"
onclick="window.location.href='./browse';">Browse Notes</button>
```

Auszug CSS-Code zu .hp-button

```css
.hp-button {
    appearance: none;
    background-color: var(--ui-blue);
    border: 2px solid var(--ui-blue);
    border-radius: 15px;
    color: white;
    cursor: pointer;
    font-size: var(--ui-font-size-big);
    font-weight: 550;
    margin: 0 auto;
    min-height: 60px;
    outline: none;
    padding: 1rem;
    text-align: center;
    text-decoration: none;
    transition: all 0.3s ease-in-out;
    width:45%;
    will-change: transform;
}

.hp-button:hover {
    box-shadow: rgba(0, 0, 0, 0.25) 0 8px 15px;
    transform: translateY(-3px);
}

.hp-button:active {
    box-shadow: none;
    transform: translateY(0);
}
```

Erklärung Funktionsweise

.hp-button
Durch diesen Selektor werden alle Elemente mit der Klasse "hp-button" angesprochen.

appearance: none;
Das plattform- bzw. betriebssystemspezifische Aussehen des Buttons wird versteckt.

background-color: var(--ui-blue);
Der Button bekommt die Hintergrundfarbe "-ui-blue" zugewiesen. Diese Farbe wurde im :root-Element definiert und hier über die Var()-Funktion aufgerufen. Bei der Definition im :root-Element können z. B. Farben oder Schriftarten für die gesamte CSS-Datei bestimmt werden. Dies ermöglicht eine effiziente und flexible Verwendung wiederkehrender Eigenschaften und Werten, die einmal definiert und anschließend für beliebig viele Elemente angewendet werden können.

border: 2px solid var(--ui-blue);
Diese Kurzschreibweise beschreibt die Breite (border-width), den Style (border-style) und die Farbe (border-color) des Rahmens. Der hier programmierte Rahmen ist 2 Pixel breit, hat eine durchgezogene Linie und hat die Farbe "-ui-blue" zugewiesen. Alle 4 Seiten des Rahmens haben die gleiche Gestaltung.

border-radius: 15px;
Der Radius des Rahmens von 15 Pixeln gibt eine Abrundung der Ecken des Rahmens an.

color: white;
Die Schriftfarbe des Buttons ist weiß.

cursor: pointer;
Die Darstellung des Mauszeigers verändert sich zu einer kleinen Hand, wenn der Mauszeiger über den Button bewegt wird. Dadurch wird dem Nutzenden angezeigt, dass eine Verlinkung und die Möglichkeit, diese anzuklicken, existiert.

font-size: var(--ui-font-size-big);
Die Schriftgröße des Buttontextes wird auf den Wert "--ui-font-size-big" festgelegt. Hierbei handelt es sich ebenfalls um eine benutzerdefinierte Größe, die im :root-Element definiert wird. Der Wert ist max(1rem, 2vw). Durch die Max-Funktion wird stets der größere der beiden aufgeführten Werte angenommen. Da es sich bei 16px um einen absoluten Wert handelt, wird dieser nur angenommen, wenn 2vw (2 % der Viewport-Breite) kleiner als 16 Pixel ist. So passt sich die Schriftgröße responsiv an die Viewport-Breite an.

font-weight: 550;
Mit font-weight kann die Stärke (Fettdruck) der Schrift eingestellt werden. Der Normalwert einer Schrift liegt bei 400, weshalb der hier eingestelle Wert von 550 die Schrift fetter darstellt als normal.

margin: 0 auto;
Der Abstand um das Element herum wird oben und unten auf 0 gesetzt (es wird kein Abstand definiert). Der Wert auto für die linke und rechte Seite um das Element bedingen eine zentrierte Ausrichtung.

min-height: 60px;
Der Button hat eine Mindesthöhe von 60px.

outline: none;
Die Umrandung des Buttons wird deaktiviert.

padding: 1rem;
Der Abstand zwischen Inhalt und Rahmen des Elementes wird auf 1 rem (root em) festgelegt. Diese Schreibweise gibt den Abstand für alle 4 Seiten um den Inhalt an. 1 rem entspricht der Schriftgröße, die im html-Element oder :root-Element definiert wurde. In diesem Programm sind 1 rem 16 Pixel bei einer Bildschirmbreite von mindestens 768 Pixeln.

text-align: center;
Der Text wird zentriert ausgerichtet.

text-decoration: none;
Die Textdekorationen wie z. B. eine Unterstreichung werden entfernt.

transition: all 0.3s ease-in-out;
Dem Button wird ein Übergang zugewiesen. Der erste Wert all beschreibt, dass alle veränderbaren Eigenschaften wie z. B. die Position des Buttons beeinflusst werden. Der zweite Wert 0.3s definiert, dass der Übergang 0.3 Sekunden, also 300 Millisekunden dauert. Der letzte Wert ease-in-out legt den Ablauf der Animation fest. Die Animation startet langsam, beschleunigt in der Mitte und endet langsam. Dadurch soll ein sanfter Effekt entstehen.

width:45%;
Die Breite des Buttons wird auf 45 % des Eltern-Elements festgelegt.

will-change: transform;
Dem Browser wird mitgeteilt, dass dem Button eine Transformation bevorsteht. Dadurch können Animationen und Übergänge flüssiger dargestellt werden.

.hp-button:hover
Mittels des Selektor :hover wird die Gestaltung definiert, die sichtbar wird, sobald der Mauszeiger über dem Button-Element schwebt.

box-shadow: rgba(0, 0, 0, 0.25) 0 8px 15px;
Der Button bekommt einen Schatten zugewiesen. Der RGBA-Wert definiert dabei die Farbe des Schattens. RGBA steht für Rot, Grün, Blau und Alpha. Mittels dieser Parameter kann eine Farbe codiert werden. Dabei legen Rot, Grün und Blau den jeweiligen Anteil in der Farbe fest und Alpha steht für die Deckkraft bzw. Transparenz. In diesem Beispiel wurde eine schwarze Farbe mit einer Deckkraft von 25 % bzw. einer Transparenz von 75 % gewählt.
Die darauf folgenden drei Werte legen die Breite, Höhe und Unschärfe des Schattens fest.
Der erste Wert ist die horizontale Breite - hier 0 Pixel.
Der zweite Wert bestimmt die vertikale Höhe des Schattens - hier 8 Pixel.
Der letzte Wert definiert die Unschärfe, also wie verschwommen der Schatten sein soll. Die im Code gewählte Unschärfe von 15 Pixeln sorgt dafür, dass der Schatten optisch einen fließenden Übergang hat und keine harten Kanten entstehen.

transform: translateY(-3px); Es wird eine Transformation des Elements definiert. Die Funktion translateY beschreibt eine Verschiebung entlang der vertikalen Achse. Der Wert -3px legt dabei fest, dass die

Verschiebung um 3 Pixel nach oben sein soll.

Die Kombination aus box-shadow und transform sorgt dafür, dass der Button etwas nach oben zu schweben scheint, wenn man mit dem Mauszeiger über ihn fährt. Diese Animation hat das Ziel, den Nutzenden zum Anklicken des Buttons anzuregen.

.hp-button:active
Mit dem Selektor :active wird das Design beim Anklicken des Buttons bestimmt.

box-shadow: none;
Der zuvor bei :hover definiert Schatten wird aufgehoben.

transform: translateY(0);
Die zuvor bei :hover festgelegte Verschiebung entlang der vertikalen Achse wird entfernt. Der Button kehrt auf seine Ausgangsposition zurück. Der Mausklick wird so visuell bestätigt.

Responsives Design

StudyBuddy soll eine Webapplikation für verschiedene Endgeräte sein. Deshalb wird die App responsiv gestaltet, um eine optimale Nutzererfahrung auf allen Bildschirmgrößen zu ermöglichen. Damit soll das ansprechende Design und die Lesbarkeit für alle Endgeräte sichergestellt werden.

Breakpoints

Als Grundlage für das Responsive Design wurden Breakpoints festgelegt. Dabei handelt es sich um Bildschirmgrößen, bei denen eine maßgebliche Veränderung der Gestaltung angewendet wird.

In Anlehnung an die Breakpoints des ChromeDevTool (Developer Tool) sind folgende Intervalle definiert:

bis 425 Pixel Bildschirmbreite:
Damit werden voranging kleine Endgeräte wie Smartphones angesprochen. Bei diesen ist die Bildschirmbreite im Verhältnis zur Höhe gering, weshalb beim Design Elemente bevorzugt untereinander und nicht nebeneinander angeordnet werden.

426 bis 768 Pixel Bildschirmbreite:
Für mittelgroße Endgeräte wie z. B. Tablets.

Über 768 Pixel Bildschirmbreite: Diese Gestaltung richtet sich an große Endgeräte wie Laptops und Bildschirme. Bei diesen ist der Bildschirm meist breiter als hoch, weshalb Elemente nebeneinander dargestellt werden können.

Umsetzung der Responsivität

Zur praktischen Umsetzung des responsiven Designs wurden CSS-Flexboxen, die max()- und min()-Funktion, relative Maßeinheiten und Media Queries genutzt.

Diese werden im Folgenden genauer beleuchtet.

CSS-Flexboxen

Mit Flexboxen können die Kindelemente eines Elements flexibel angeordnet werden.
Zum einen wird die Position der Kindelemente über justify-content (horizontal) und align-items (vertikal) relativ zueinander oder innerhalb des Elternelements festgelegt. Dadurch ergibt sich der Vorteil, dass keine absoluten Größen benötigt werden.
Zum anderen ist es möglich, das Verhalten der untergeordneten Elemente bei einer Vergrößerung, Verkleinerung und deren Größe im Elternelement zu steuern. Mittels der Eigenschaft flex können der Wachstums-, der Verkleinerungsfaktor und die Basisgröße definiert werden. Ändert sich die Größe des Anzeigefensters, können sich die Kindelemente flexibel der Größenänderung anpassen.

max()- und min()-Funktion:

Die beiden Funktionen können genutzt werden, einen von mehreren Werten auszuwählen. Die max()-Funktion gibt immer den größten Wert zurück. Bei der min()-Funktion ist es umgekehrt.

Für die Gestaltung der Schriftgrößen von StudyBuddy wurde die max()-Funktion im :root-Element genutzt.

```
    --ui-font-size-big: max(1rem, 2vw);
    --ui-font-size-small: max(0.8rem, 2vw);
    --ui-font-size-text: max(0.6rem, 1.4vw);
```

Dadurch passen sich die Schriftgrößen abhängig von der Bildschirmbreite an. Beträgt die Bildschirmbreite bspw. 2000 Pixel, so würde die max()-Funktion der --ui-font-size-big zwischen den Werten 1 rem (16 Pixel) und 2000*0,02 = 40 Pixel die Schriftgröße von 40 Pixeln auswählen. Sobald der Wert 2vw (Prozentualer Anteil der Bildschirmbreite, view width) kleiner als die Größe 1 rem wird, verbleibt die Mindestschriftgröße bei 1 rem.

So kann sichergestellt werden, dass die Schriftgröße nicht kleiner als 1 rem wird, diese sich aber bei großen Bildschirmen entsprechend vergrößert.

Die min()-Funktion wurde im Zusammenhang mit der Breite des Signups verwendet.

```
.signup{
      width: min(500px, 100%);
  }
```

Hier wird bestimmt, dass das Element der Klasse "signup" maximal 500 Pixel breit ist. Diese Breite wird unterschritten, sobald der Wert 100 % einer Breite von weniger als 500 Pixel entspricht.

Neben den Schriftgrößen wurden diese Funktionen für die dynamische Gestaltung von Elementhöhen, -breiten und paddings verwendet. Die Funktionen, in Verbindung mit mindestens einer relativen Größe als Parameter, ermöglichen eine flexible Anpassung an die Bildschirmgrößen.

relative Maßeinheiten

Beim responsiven Design spielen relative Maßeinheiten eine wichtige Rolle, da auch sie es ermöglichen, Inhalte flexibel an Bildschirmbreiten anzupassen.
Anstatt absoluter Pixelwerte kommen Einheiten wie Prozent %, em oder rem, Viewport height vh oder

Viewport width vw zu Einsatz. Mit Prozent wird die Breite oder Höhe eines Elements im Verhältnis zum übergeordneten Element definiert. Verändert sich die Größe des Elternelements, wird auch die Größe des untergeordneten Elements angepasst.
Em und rem ermöglichen eine Skalierung von z. B. Schriftarten basierend auf der Größe des übergeordneten Elements (em) oder des root-Elements (rem). Die Viewport-Größen vh und vw passen sich direkt an die Bildschirmgröße an und eignen sich deshalb besonders für Elemente, deren Größe im direkten Verhältnis zur Bildschirmgröße steht.

Für StudyBuddy wurden relative Maßeinheiten bspw. folgendermaßen angewendet:

```
.browse-search{
    box-sizing: border-box;
    flex: 1 1 90%;
    margin: 0 1rem;
}
```

Hier wird der Klasse "browse-search" eine Anfangsbreite von 90 % des Elternelements zugewiesen. Weiterhin wurde die Margin - also der Außenabstand des Elements - links und rechts auf 1rem gesetzt. Dadurch passt sich dieses Element flexibel an die Größe des Elternelements an. Die Margin passt sich mit Veränderung der font-size, die im root-Element definiert wird, an.

Media Queries Media Queries sind ein Werkzeug, um Designanpassung in Abhängigkeit der Bildschirmgröße oder Gerätebedingungen zu machen. Sie ermöglichen es, Gestaltungsregeln, aufgrund von z. B. der Breite, Höhe oder Ausrichtung des Viewports, zu aktivieren. Dadurch lässt sich eine Differenzierung des Designs je nach Beschaffenheit des Endgeräts einrichten.

Für StudyBuddy wurden Media Queries genutzt, um auf die Webseitengestaltung bei mittleren und kleinen Geräten einzuwirken.

```
@media screen and (min-width: 426px) and (max-width: 768px){
    html{
        font-size: 12px;
    }
    .signup{
        width: min(500px, 100%);
    }
}
```

In diesem Code-Auszug wird das Design für die mittelgroßen Endgeräte, gemäß der zuvor definierten Breakpoints, angepasst. Über den Selektor wird die Bildschirmgröße abgefragt und wenn diese zwischen 426 Pixeln und 768 Pixeln liegt, wird die folgende Gestaltung angewendet. In diesem Fall wird die Schriftgröße des hmtl-Elements von 16 Pixel auf 12 Pixel angepasst. Diese Schriftgröße ist die Basis für die Einheit rem. Daraus folgt, dass alle Eigenschaften mit einem Wert der Einheit rem ebenfalls proportional verkleinert werden.

Weiterhin wird die ursprüngliche Breite der Klasse "signup" überschrieben.

Herausforderungen bei der Umsetzung

Eine Schwierigkeit bei der Implementierung des responsiven Designs stellte die Wahl dar, ob man nach dem Mobile-First- oder Desktop-First-Ansatz vorgeht. Da zu Beginn der Gestaltung mit CSS keine fundierten Kenntnisse zu den beiden Ansätzen vorlagen, war der Einstieg in dieses Thema schwierig. Nachdem erst im fortgeschrittenen Projektverlauf die Frage nach dem geeigneten Prinzip aufkam, wurde sich für den Desktop-First-Ansatz entschieden. Hintergrund war, das bis zu diesem Zeitpunkt eine Desktopansicht entworfen wurde. So konnte im Nachhinein - mit einem gewissen Programmier- und Testaufwand - der Desktop-First-Ansatz implementiert werden.

Letztendlich konnte die Webseite trotz dieser Herausforderung erfolgreich responsiv gestaltet werden.

Clientseitige Implementierung

Grundlegende Struktur

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

Die für viele Funktionalitäten benötigten Routen sind unter `src/routes` definiert 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 clientseitig zur Abwicklung der Signup- und Login-Funktionen verwendet und ist eingebunden in `/static/login.html` sowie `/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 festgelegte 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 Kontrollinstanz 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. 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 fehlschlägt, 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, dass 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 festgelegte 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 URI `/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 aufgrund dessen Cookies, die vom Server gesendet werden. Als Resultat erstellt der Server einen 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 200-er Bereich liegt, da im Normalfall `res.status(200)` erhalten wird, erscheint dann die Mitteilung eines erfolgreichen 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 mit der übermittelten Message ausgegeben.

Falls die Kommunikation mit `/api/users/signup` gänzlich fehlschlägt, 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 eintritt, 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-Objekt 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 Abhängigkeit 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 eingeloggte Benutzer auf die Share-Seite gelangen.

Zu Beginn stellt der DOMContentLoaded-Event Listener sicher, dass die Seite fertig geladen ist, bevor das Skript ausgeführt wird. Darauffolgend wird durch eine GET-Anfrage überprüft, ob der Nutzer eingeloggt ist, wobei die Antwort 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 Begrüßung: `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 wird 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 200-er 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 Bestätigen des `share-btn` in der Navigationsleiste ob der 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 werden. 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 funktionierendes 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-Dokuments `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 ein 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 aussehen, 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 gefunden (`data.numDocs === 0`), wird der Schriftzug `"No documents found."` in der resultsSection angezeigt. Andernfalls 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 jetzigen 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 anschließend 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 angeklickt, 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}`;
    }
});
```

Serverseitige Implementierung

Grundsätzliche Anforderungen

Das Backend soll einen schnellen und effizienten Umgang mit den Requests der User ermöglichen.

Das Backend basiert auf der Node.js-Runtime und ist in der Datei `src/app.js` implementiert. Zudem wird Express.js als Web-Applikations-Framework verwendet, um einfach Handling von Requests und Responses zu ermöglichen. Als Datenbank wird MongoDB verwendet, mehr dazu unter Datenbank.

Besondere Sicherheitsanforderungen werden explizit nicht gestellt. In der Realität wäre es empfehlenswert, neben anderen Sicherheitsvorkehrungen beispielsweise HTTPS statt HTTP zu verwenden. Darauf wird hier jedoch verzichtet, um unnötige Komplexität zu vermeiden und das Projekt auf dessen funktionale Kernbestandteile zu beschränken.

Die grundlegende Struktur des Backends ist Folgende:

- Das Skript `/src/app.js` ist der Startpunkt für den Web-Server und modular aus verschiedenen Skripten für die Verarbeitung der Requests auf verschiedenen Routen zusammengesetzt.
- Diese Routen befinden sich unter `/src/routes`
- Die statischen Pages sollen nicht immer für den User zugänglich sein, d.h. diese befinden sich nicht in `/public/`, sondern unter `/static`. Um außerdem die Notwendigkeit zu umgehen, die URI mit

`.html` abzuschließen, werden die HTML-Dateien unter `/static` über Router bereitgestellt.

Server-Einstiegspunkt `app.js`

`app.js` ist der Einstiegspunkt der Anwendung, zuständig für das Server-Setup und den Aufbau externer Verbindungen, bspw. zu MongoDB.

Zu Beginn erfolgt der Import verschiedener Module und Namespaces:

- `express` als Web Framework zur Erstellung des Servers
- `path` für den Umgang mit Dateipfaden und URIs
- `mongoose` ist ein Modul zur Verwendung von MongoDB-Datenbanken in Node.js-Skripten
- `express-session` wird für die Verwaltung der User-Sessions genutzt
- `express-fileupload` ist eine Middleware zum Umgang mit Dateiuploads vom Client

Nach dem Import der externen Module werden die Router verschiedener interner Module importiert, mehr dazu folgt unter Routing.

Wichtige Konstante, die definiert werden, sind die `app` als `express`-Instanz, der `PORT`, unter welchem die Anwendung auf dem `localhost` läuft, sowie die URI der MongoDB-Datenbank.

Im Anschluss wird die benötigte Middleware, welche zu Beginn des Scripts importiert wird, eingebunden. `cors` wird folgendermaßen konfiguriert:

```
app.use(cors({
  origin: `http://localhost:${PORT}`,
  credentials: true
}));
```

Mit dieser Konfiguration ist der Server selbst der einzige zulässige Ursprung für Requests und alle Requests sind nur mit den Session-`credentials` erlaubt.

Die Middleware hierfür wird im Folgenden konfiguriert:

```
app.use(session({
  secret: 'deinGeheimerSchluessel',
  resave: false,
  saveUninitialized: false,
  cookie: { secure: false }
}));
```

- `secret` ist der geheime Schlüssel, mit dem Session-Cookies gesigned und verifiziert werden.
- `resave: false` stellt sicher, dass Session-Informationen nur dann neu gespeichert werden, wenn sie sich ändern.
- Mit `saveUnitialized: false` wird konfiguriert, dass nicht bei jeder HTTP-Request automatisch ein neues Session-Objekt kreiert und gespeichert wird, sondern nur dann, wenn der Client eine neue Session initialisiert.

- `cookie: { secure: false }` bedeutet, dass die Session-Cookies per HTTP versandt werden, und nicht per HTTPS

Um für diese Ressourcen keine extra Router konfigurieren zu müssen, werden Bilder, CSS-Dateien und die clientseitig auszuführenden JavaScripte, welche sich allesamt im Verzeichnis `/public` befinde, uneingeschränkt für den Client bereitgestellt:

```
app.use("/public", express.static(path.join(__dirname, "../public")));
```

Andere Routen, wie zur Navigation auf die Subpages der Website, oder auf deren API für den User-Login und Dokumenten-Upload, greifen auf die dedizierten Router zurück:

```
// API route for user management (login, singup)
app.use("/api/users", userRoutes);
// API route for document upload
app.use("/api/upload", docRoutes); // Integration of document upload route

// Routes for subpages
app.use("/", homepage);
app.use("/browse", browse);
app.use("/login", login);
app.use("/signup", signup);
app.use("/impressum", impressum);
app.use("/share", share);
```

Abschließend wird mit den passenden URIs eine Verbindung zur MongoDB-Datenbank aufgebaut und der Server gestartet:

```
// Establishing MongoDB connection
mongoose
  .connect(MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Verbunden mit MongoDB"))
  .catch((err) => console.error("Fehler bei der Verbindung zu MongoDB:",
err));

// Starting up server
app.listen(PORT, () => {
  console.log(`Server läuft auf http://localhost:${PORT}`);
});
```

Routing und API-Endpunkte

Die Router der verschiedenen Unterseiten befinden sich im Verzeichnis `/src/routes` und sind schematisch gleich aufgebaut, beispielhaft soll `/src/routes/login.js` hier der Erläuterung dienen:

Das Skript beginnt mit dem Import des `express`-Moduls und `path`-Namespace, deren Verwendungszweck jeweils bereits oben erklärt wurde.

Ein Unterschied zu `app.js` stellt hier die Instanziierung eines Routers dar, da hier kein vollständiger Server, sondern lediglich eine valide URI des bereits in `app.js` erstellten Servers kreiert werden soll.

Für diesen Router wird anschließend dessen Antwort auf HTTP-GET-Requests auf dessen path (`"/"`) definiert. Hier erfolgt serverseitig eine Konsolenausgabe der vom User aufgerufenen Seite und anschließend wird das HTML-File der angeforderten Seite als Response an den Client gesendet.

Zum Schluss wird der Router exportiert, also beim Import der jeweiligen Datei in `app.js` per verfügbar gemacht.

```
// login route
router.get('/', (req, res) =>{
    console.log("User opening login subpage");
    res.sendFile(path.join(__dirname, "../../static/login.html"));
});

module.exports = router;
```

Ausnahmen von diesem Schema stellen `uploadRoute.js`, `userRoutes.js`, `browse.js` und `homepage.js` dar. Letzteres Skript unterscheidet sich nur, indem neben dem Pfad `/` auch der Pfad `/homepage` die dazugehörige Datei zur Verfügung stellen.

Die Besonderheiten der anderen Router werden im Folgenden erläutert:

Dokumenten-Browsing mit `browse.js`

Dieses Skript importiert neben den zuvor genannten zwei Modulen auch die `mongoose`-Schemata `Doc` und `User`, welche jeweils unter `/src/models` als Modelle von Kollektionen der MongoDB-Datenbank erstellt werden.

Dieser Router verfolgt eine von den anderen Unterseiten-Routern abweichende Logik: Die GET-Request dient hier nicht dazu, dem Client in der Response das korrespondiere HTML-File bereitzustellen. Stattdessen werden der Request hier zwei Queries `searchTerm` und `tags` entnommen.

Abbildung 9 stellt schematisch die verkürzten HTTP-Request und Response bei fehlerfreiem Ablauf der Browse-GET-Request dar.

Die Aufgabe der Browse-Seite ist es, dem User die Möglichkeit zum Durchsuchen der in der Datenbank hinterlegten Dokumente zu geben. `searchTerm` ist hierbei der Suchbegriff, welcher im Titel oder der Beschreibung eines Dokuments vorkommen muss, damit das Dokument als mögliches Suchergebnis in Frage kommt.

Darüber hinaus kann der User nach Tags filtern, d.h. nur Dokumente mit einem der spezifizierten Tags werden ihm angezeigt. Dieser Filter wird ebenfalls bereits in der serverseitigen Business-Logik angewendet.

Es wird also erst versucht, der Request diese beiden Queries zu entnehmen:

```
router.get('/', async (req, res) => {
  console.log("User opening browse subpage");
  try {
    // Destructuring of query parameters
    const { searchTerm, tags } = req.query;
    console.log("Searchterm:", searchTerm);
    console.log("Tags:", tags);

    // Return HTML page if no search term was given
    if (!searchTerm) {
      return res.sendFile(path.join(__dirname,
"../../static/browse.html"));
    }
```

Sollte der Query keinen Suchbegriff enthalten, handelt es sich nicht um eine valide Anfrage und dem Client wird die statische Browse-Seite gesendet.

Gibt es einen `searchTerm`, wird der passende Filter `query` formuliert, welcher mittels eines regulären Ausdrucks überprüfen soll, welche Dokumente den `searchTerm` in deren `title` oder `decription` enthalten. `$options: "i"` spezifiziert hierbei, dass die sich RegEx hierbei case-insensitive verhalten soll. Da der Suchbegriff des Users generell verlangt wird, Tags jedoch optional sind, werden diese dem Filter nur hinzugefügt, wenn sie auch übergeben wurden

```
    // Base query for search term (document is a match if titel or
description contains search term)
    let query = {
      $or: [
        {
          title: {
            $regex: ".*" + searchTerm + ".*",
            $options: "i"
          }
        },
        {
          description: {
            $regex: ".*" + searchTerm + ".*",
            $options: "i"
          }
        }
      ]
    };

    // Only checking if tag matches if tag is given
    if (tags && tags.trim() !== "") {
      query.tag = { $in: tags.split(",").map(t => t.trim()) };
    }
```

Anschließend wird die Suche in der Datenbank mit dem Filter durchgeführt und die Ergebnisse auf die Attribute `userID`, `title`, `uploadDate`, `description` und `tag` projiziert. Diese Projektion findet statt, da client-seitig keine anderen Informationen über die Dokumente benötigt werden.

Außerdem werden mithilfe des `User`-Modells die `userID`s um die `firstName`-Attribute der User ergänzt, und ein Mapping der aktuellen Attribut-Werte zu den im Frontend erwarteten Keys vollzogen:

```
    // Populate with user info
    const populatedMatches = await Doc.populate(matches, {
      path: "userID",
      model: User,
      select: "firstName"
    });

    // Mapping onto format expected by frontend
    const documents = populatedMatches.map(doc => ({
      docID: doc._id,
      docTitle: doc.title,
      docDescription: doc.description,
      docTag: doc.tag,
      docAuthor: doc.userID.firstName,
      docDate: doc.uploadDate
    }));
```

Sollten all diese Schritte fehlerfrei funktioniert haben, wird die entsprechende Response mit der Anzahl der Dokumente und dem Dokumenten-Array mit Status 200 an das Frontend gesendet, anderenfalls ist die Response der Status 400 für eine invalide Anfrage und ein leeres Dokumenten-Array:

```
    console.log("Matching documents:", populatedMatches);
    // sending response back to client
    res.status(200).json({ numDocs: documents.length, documents });
  } catch (err) {
    // error handling
    console.log(err);
    res.status(400).json({ numDocs: 0, documents: [] });
  }
});
```

Da die GET-Request (außer in dem Fall eines reinen Aufrufs von `/browse` ohne `searchTerm`) nun jedoch bereits eine Response mit JSON-Body sendet und der Body einer HTTP-Response nur ein Format (HTML oder JSON, nicht jedoch beides gleichzeitig) unterstützt, muss die HTML-Datei der Seite anderweitig versendet werden.

Hierzu wird die POST-Methode gewissermaßen zweckentfremdet. Stellt der Client eine POST-Request, wird diese ungeachtet ihres Inhalts mit der HTML-Datei beantwortet:

```
router.post("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../../static/browse.html"));
});
```

Der Dokumenten-Download erfolgt ebenfalls per GET-Request, über die Route `"/browse/download"`, mit der ID des herunterzuladenden Dokumemts im Query.

Abbildung 10 stellt schematisch die verkürzte HTTP-Request und -Response bei fehlerfreiem Ablauf der Download-GET-Request dar.

Sollte die angefragte Datei nicht gefunden werden, wird dem Client der Status 400 wegen der invaliden Anfrage sowie eine entsprechende Fehlernachricht gesendet:

```
router.get("/download", async (req, res) => {
  try {
    // Use of the docID as a query parameter
    const docID = req.query.docID;
    console.log("User requested " + docID);

    // Find the document with the given ID
    const file = await Doc.findById(docID).exec();

    // Checking if the document exists
    if (!file) {
      // If the document does not exist, return a status code of 400
      return res.status(400).send("The document you requested does not
seem to exist");
    }
```

Existiert ein Dokument mit der angefragten ID in der Datenbank, so wird ein passender Response-Header formuliert und die Datei mit Status 200 an den Client zurückgesendet:

```
      // If the document exists, send it to the client
      console.log(file);
      // Configuring the response headers
      res.set({
        "Content-Type": "file.fileType", // "application/octet-stream"
becomes "file.fileType" to get the file type
        "Content-Disposition": `attachment;
filename="${file.originalName}"` // original filename
      });

      // Sending the file to the client
      res.status(200).send(file.file);
    }
```

Gibt es dennoch einen Fehler, so wird davon ausgegangen, dass es sich um eine invalide Anfrage handelt und der passende Status wird mit entsprechender Meldung an den Client zurückgegeben:

```
} catch (err) {
    // Error handling
    console.log(err);
    res.status(400).send("Invalid request");
  }
});

module.exports = router;
```

Dokumenten-Upload mit `uploadRoute.js`

`uploadRoutes.js` ist das Skript, welchen den Upload der Dokumente über die Route `"/api/upload` ermöglicht. Hierzu werden auch hier wieder die Dokumenten- und User-Modelle importiert und ein Express-Router instanziiert.

Ziel ist es, dass der Client per POST-Request eine Datei an den Server senden kann und dieser folgende Antwort gibt:

- Status 200 im Falle eines erfolgreichen Uploads
- Status 400 bei einer inavliden User-ID
- Status 500 ansonsten

Abbildung 11 stellt schematisch die verkürzte HTTP-Request und -Response bei fehlerfreiem Ablauf der Upload-POST-Request dar.

Hierzu muss zuerst die Validität der POST-Request sichergestellt werden. Diese sollte im Body den Dokumententitel, dessen Beschreibung, einen Tag (Exercise, Summary oder Scribbled Notes, quasi die Art des Lerninhalts) und die ID des Users enthalten, welcher den Upload tätigt.

`req.file` sollte zudem einen `data` (die eigentlichen Inhalte der Datei), einen `mimetype` (Dateitypen) und `name` (Dateinamen) haben:

```
router.post("/", async (req, res) => {
    try {
        const body = req.body;

        const fileObj = req.files.uploadFile; // express-fileupload for
file handling
        // Making sure file was sent
        if (!fileObj) {
            return res.status(400).json({ message: "File is missing" });
        }
        // Destructuring into file contents and metadata
        const fileBuffer = fileObj.data;
        const fileType = fileObj.mimetype;
        const originalName = fileObj.name;
```

```
        // Destructuring request body
        const title = body.docTitle;
        const description = body.description;
        const tag = body.tag;
        const userID = body.userID;
```

Wurde die Request entsprechend destrukturiert, kann überprüft werden, ob es einen entsprechenden User mit dieser ID in der Datenbank gibt und wenn ja, ein neues `Doc`-Objekt mit den passenden Attributen erzeugt, gespeichert und die Response über den erfolgreichen Upload an den Client gesendet werden:

```
        // Checking if user exists
        const user = await User.findById(userID).exec();
        if (!user) {
            return res.status(400).json({ message: "Your User ID does not
exist" });
        }

        // Creating new document
        const uploadDate = new Date();
        const doc = new Doc({
            userID,
            title,
            uploadDate,
            description,
            file: fileBuffer,
            fileType,
            originalName,
            tag,
        });

        // Saving document to database
        await doc.save();
        return res.status(200).json({ message: "Doc saved successfully"
});
```

Gibt es bei einem dieser Schritte einen Fehler, wird dem Client ein Status 500 gesendet, um zu signalisieren, dass es serverseitig ein Problem gegeben haben muss (es könnte auch das Format der Request falsch sein, allerdings lassen sich auch Fehler durch serverseitige Schwächen wie die Inkompatibilität des aktuellen Datenbank-Setups für Dateien mit mehr als 16 MB hervorrufen):

```
    } catch (err) {
        // Error handling
        console.log(err);
        res.status(500).json({ message: "Server Error" });
    }
});

module.exports = router;
```

Signup, Login und Signup durch `userRoutes.js`

Aufgabe von `userRoutes.js` ist es, drei Routen - `api/users/signup`, `api/users/login` und `api/users/logout` - zu erstellen, welche dem Client POST-Requests für die Registrierung, den Login oder den Logout eines Users ermöglichen.

Hierzu wird das `User`-Schema der Datenbank benötigt, sowie ein Modul `bcrypt` zum Hashing des vom User vergebenen Passworts mithilfe eines ebenfalls durch `bcrypt` generierten Salts. Dieser `passwordHash` wird initial bei der Registrierung eines neuen Nutzers berechnet und als dessen Passwort in der Datenbank hinterlegt. Zuvor werden die Validität der Signup-POST-Request überprüft und sichergestellt, dass es nicht bereits einen Nutzer mit derselben E-Mail-Adresse gibt. Diese Einmaligkeit ist wichtig, da die E-Mail-Adresse später auch beim Login vom Nutzer verwendet wird. Zeitliche Kontinuität hingegen ist keine Anforderung an die E-Mail-Adresse, da alle mit dem User verknüpften Daten anderer Collections in der Datenbank hierfür den Primärschlüssel `_id` des Users für die Zuordnung nutzen.

Die oben beschriebenen Abläufe sind folgendermaßen implementiert:

```
// Signup
router.post('/signup', async (req, res) => {
  try {
    // Destructuring request body
    const { firstname, email, password } = req.body;

    // Making sure body is valid
    if (!firstname || !email || !password) {
      return res.status(400).json({ msg: "All fields are required" });
    }

    // Checking whether an account with this email already exists
    let user = await User.findOne({ email });
    if (user) {
      return res.status(400).json({ msg: "User already exists" });
    }

    // Hashing the password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Creating a new user based on the model defined in
src/models/userModel.js
    user = new User({
      firstName: firstname,
      email,
      password: hashedPassword
    });
    // Saving new user to the database
    await user.save();
```

War der Signup-Vorgang erfolgreich, lautet die Antwort an den Client Status 201, sonst Status 500:

```
    // Sending response to the client
    res.status(201).json({ msg: "User registered successfully" });
  } catch (err) {
    // Error handling
    console.error("Signup Error:", err);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Für den Login wird nun, wie beschrieben, die E-Mail-Adresse des Users genutzt, um sicherzustellen, dass es einen entsprechenden Account gibt. Ist dies der Fall, vergleicht `bcrypt.compare()` die Passwörter. Ein inkorrektes Passwort führt zu einer Fehler-Response an den Client; ein korrektes Passwort zur Erstellung einer Session:

```
// Login
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(400).json({ message: "E-Mail nicht gefunden" });
    }
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(400).json({ message: "Falsches Passwort" });
    }
    // Saved logged in user in session
    req.session.user = {
      id: user._id,
      firstName: user.firstName,
      email: user.email
    };
```

Die Session-Informationen werden im Request-Objekt gespeichert und eine passende Response, welche auch den Vornamen des Users beinhaltet, wird an den Client gesendet. Serverseitig fehlerhafte Login-Versuche - abweichend von den bereits oben beschriebenen Fehlern durch inkorrekte User-Daten - resultieren in Status 500:

```
    return res.status(200).json({
      message: "Login erfolgreich",
      firstName: user.firstName
    });
  } catch (error) {
    res.status(500).json({ message: "Serverfehler" });
```

```
  }
});
```

Loggt der User sich aus, geschieht dies über eine POST-Request an `/api/users/logout`. Hierzu wird die hinterlegte Session zerstört und der Client erhält mti `res.clearCookie()` die Instruktion, den hinterlegten Session-Cookie zu entfernen:

```
// Logout
router.post('/logout', (req, res) => {
  req.session.destroy(err => {
    if (err) {
      console.error("Logout Error:", err);
      return res.status(500).json({ message: "Logout Error" });
    }
    res.clearCookie('connect.sid'); // standard name for session cookie
    res.status(200).json({ message: "Logout erfolgreich" });
  });
});
```

Eine letzte Route `/api/users/status` gibt es außerdem zur Abfrage des Session-Status - ob der Client aktuell mit einem User eingeloggt ist oder nicht. Hier wird eine entsprechende GET-Request mit einer Response mit JSON-Body beantwortet, der einen mit Booleschem Wert belegten Key `loggedIn` enthält und auch die User-Informationen zurücksendet, sollte dieser eingeloggt sein (`id`, `firstName` und `email`):

```
// request session status (logged in or not)
router.get('/status', (req, res) => {
  if (req.session && req.session.user) {
    return res.status(200).json({
      loggedIn: true,
      user: req.session.user
    });
  } else {
    return res.status(200).json({ loggedIn: false });
  }
});
```

Datenzugriff und Modellinteraktion

Bei der verwendeten Datenbank handelt es sich um eine MongoDB-Datenbank, welche alle CRUD-Operationen unterstützt. Vorteile dieser NoSQL-Datenbank sind die BSON-Datenstruktur, welche JSON stark ähnelt, und die allgemein sehr einfache Integration von MongoDB und JavaScript miteinander durch das `mongoose`-Modul.

Für das Prototyping und die Nutzung der Datenbank zur Entwicklung dieses Projektes mit verhältnismäßig knappem Zeitrahmen hat sich MongoDB insbesondere auch angeboten, da die Collections einer MongoDB-Datenbank keinem festen Schema folgen müssen. Dokumente derselben Collection dürfen sich in den

hinterlegten Feldern und den Datentypen dieser Felder voneinander unterscheiden. Das hat es einfach gemacht, die Datenbank schnell um neue Daten zu erweitern und die hinterlegten Daten im Entwicklungsprozess anzupassen.

Auch die Möglichkeit, beispielsweise PDF-Dateien in Buffern direkt in Dokumenten der Datenbank zu speichern - nicht nur deren Pfade im Dateiverzeichnis - und die Datenbank selbst per `mongodump`-Befehl als JSON-Datei zu exportieren, hat in Kombination mit Git zum Versionsmanagement die synchrone Projektarbeit über mehrere Geräte hinweg erleichtert.

Weitere Vorteile von MongoDB sind gute Skalierbarkeit durch die verteilte Speicherung sowie die hohe Performance bei Lese- und Schreiboperationen einzelner Dokumente.

User-Modell: `userModel.js`

```
// import mongoose
const mongoose = require("mongoose");

// schema definition
const userSchema = new mongoose.Schema({
  firstName: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});

// module export
module.exports = mongoose.model("User", userSchema);
```

Bei `userSchema` handelt es sich um `mongoose.Schema`-Objekt. In diesem konkreten Schema werden drei Felder definiert: `firstName`, `email` und `password`. Alle davon sind vom Typ `String` und müssen bei Instanziierung eines `User`-Objektes für dieses definiert werden. Die E-Mail muss zudem einmalig sein.

Dieser "Bauplan" wird mittels des `mongoose.Schema`-Konstruktors formuliert und mit `module.exports = mongoose.model("User", userSchema)` das Modell hierzu, welches die Interaktion mit der passenden Collection ermöglicht.

Die Collection wird bei der ersten Speicherung eines mit dem `User`-Modell instanziierten Objekts automatisch erstellt. Ihr Name entspricht dann dem Namen des Modells - hier `User` - in lowercase-Buchstaben und mit einem angehängten "s". In diesem Beispiel wird die MongoDB-Collection, die alle gespeicherten `User`-Objekte enthält, also mit "users" bezeichnet.

Dokumenten-Modell: `docModel.js`

```
// import mongoose
const mongoose = require("mongoose");

// schema definition
const docSchema = new mongoose.Schema({
  userID: { type: mongoose.Types.ObjectId, required: true },
```

```
  title: { type: String, required: true },
  uploadDate: { type: Date, required: true },
  description: { type: String, required: true },
  file: {type: Buffer, required: true },
  fileType: { type: String, required: true },
  originalName: { type: String, required: true },
  tag: { type: String, required: false },
});

// module export
module.exports = mongoose.model("Doc", docSchema);
```

Identisch verfahren wird für das Modell der Dokumente. Eine Besonderheit ist hier, dass neben den anderen Attributen - auf diese wird nicht näher eingegangen, da sie auch bereits dem ER-Modell entnommen werden können - auch die zu speichernde Datei selbst in einem Feld namens `file` vom Typ `Buffer` hinterlegt wird.

Das heißt, die zu speichernde Datei ist hier in binärer Form direkt in der Datenbank gespeichert. Dabei ist zu beachten, dass `mongoose` die Größe des Buffers auf 16 MB beschränkt (konkreter wird die Größe jedes BSON-Dokuments in der Datenbank auf 16 MB limitiert). Größere Dokumente werden vorerst nicht unterstützt, wobei die GridFS-Spezifizierung womöglich einen Weg bieten könnte, durch Aufteilung großer Dokumente in mehrere kleinere Einheiten auch Dateien über 16 MB zu unterstützen [^3]. Die Praktikabilität dessen im Vergleich zum einfachen Speichern der Dateipfade müsste separat weiter evaluiert werden.

Praxisbezogene Optimierungen

Perspektivisch könnte es jedoch sinnvoller Sein, auf eine relationale Datenbank umzusteigen, welche höhere formelle Standards erfordert. Damit würde ein Maß an Flexibilität verloren gehen. Mit dem Hintergedanken, dass das Projekt in der Realität den Zweck hätte, große Mengen an Dokumenten, Usern und weiteren Daten zu speichern, während potenziell mehrere Tausend Clients simultane Suchanfragen durchführen, könnte StudyBuddy von den schnelleren Lookup-Times einer relationen Datenbank profitieren.

Denn das Bottleneck in Sachen Performance wird in diesem Fall vermutlich nicht in der Lese- und Schreibgeschwindigkeit auf einzelnen Dokumenten zu finden sein, sondern eher in der Anwendung vieler Queries auf die gesamte Datenbank bei der Dokumentensuche durch User.

Zudem bietet die Flexibilität, welche uneinheitliche Datenbankeinträge in der Entwicklung erlauben, im Production-Kontext keinen Vorteil mehr, erhöht jedoch die Fehleranfälligkeit.

Dennoch wird MongoDB aktuell noch genutzt, sodass die beiden in `/src/models/userModel.js` und `/src/models/docModel.js` definierten Modelle hier erläutert werden sollen.

Auch die Speicherung der PDF-Dokumente außerhalb der Datenbank und stattdessen ein schlichter Verweis auf deren Pfad könnte hinsichtlich der Performance empfehlenswert sein.

[^3]: MongoDB, Inc. (2024). GridFS for Self-Managed Deployments. MongoDB Manual.

Hilfsmittel

Literatur

Wolf, Jürgen: HTML und CSS: Das umfassende Handbuch,
5., aktualisierte und überarbeitete Auflage 2023, Rheinwerk Verlag GmbH, Bonn 2023.

Dieses Buch wurde zur Bildung eines grundsätzlichen Verständnisses sowie als Nachschlagewerk für die verschiedenen HTML- und CSS-Elemente genutzt. Anfangs wurde es zur Vermittlung von Grundkenntnissen zu CSS-Gestaltungsmöglichkeiten und deren Anwendung verwendet. Dazu gehörten beispielsweise die Verwendung von margin, padding, color, border-Design und font-Design. Weiterhin wurden die verschiedenen Maßeinheiten in CSS und die Codierung von Farben erlernt.

Im späteren Lauf diente dieses Werk zur Vertiefung des Wissens über CSS-Flexboxen, CSS-Grid und responsives Design.

Schneider, Jürgen: Vorlesungsskript Web Engineering Kap 3 und Kap 4
Die Vorlesungsskript Kapitel 3 und Kapitel 4 wurden als Nachschlagewerk für den Aufbau und die Gestaltung der Webapplikation genutzt. Die beiden Kapitel behandeln die Themen HTML und CSS.

Artificial Intelligence

Microsoft Designer
https://designer.microsoft.com/home

Die Bildgenerierung Software Microsoft Designer wurde zur Erstellung der Grafiken verwendet. Mittels des Prompts *"Logo für eine App mit Namen "StudyBuddy" in Blau und Lila. Mit dem text "studybuddy"."* wurden mehrere Varianten des gewählten Logos bzw. Icons erzeugt. Eine Schwierigkeit stellte dabei dar, dass die Bilder meist den Schriftzug "Studdybuddy" erhielten, der einen Tippfehler hatte. Trotz des Hinweisens der KI auf diesen Fehler konnte keine Korrektur erfolgen. Deshalb wurde der Schriftzug manuell erstellt.

Weiterhin wurden die beiden Bilder auf der Homepage mit Microsoft Designer erstellt. Der hier verwendete Prompt ist *"In Blau und Lila. Gezeichnete Figur, die am PC etwas sucht. "studybuddy""*. Dort wurden aus vier generierten Bildern die zwei Grafiken ausgewählt, die auch auf der Homepage zu sehen sind.

Webseiten und -applikationen

Can I Use
https://caniuse.com/

Diese Webseite bietet die Möglichkeit, CSS-Selektoren oder -Eigenschaften auf ihre Kompatibilität mit Browsern zu prüfen.

Dementsprechend würde diese Anwendung dafür genutzt, um die Kompatibilität des erstellen CSS-Codes mit den Browsern Mozilla Firefox, Google Chrome, Microsoft Edge und Safari abzufragen.

Coloors Image Picker
https://coolors.co/image-picker/

Diese Anwendung wurde zur Erstellung der Farbauswahl für die Webseite genutzt. Nach dem Hochladen des Logos konnten verschiedene Farben aus dem Logo extrahiert und die HEX-Werte für diese Farben

ausgelesen werden. Die hier erstellte Farbauswahl findet sich in den "UI"-Farben wieder, welche im :root-Element des CSS-Codes definiert wurden.

Figma
https://www.figma.com/de-de/

Figma ist ein Design-Tool, mit dem der erste Webseitenentwurf erstellt wurde. Dieser wird in der Einleitung dieses Kapitels beschrieben.

Google Font Icons
https://fonts.google.com/icons

Google Font bietet eine große Bibliothek von u. A. Icons an. Die Symbole im Login-Formular und Signup-Formular wurden als svg-Elemente über Google Font bezogen. Dabei kann z. B. die Farbe und das Filling der Icons über Google Font definiert werden. Im Anschluss kann der Code zum svg-Element kopiert und in der HTML-Datei eingefügt werden.

IANA Media Types
https://www.iana.org/assignments/media-types/media-types.xhtml#image

Hier kanne eine gesammelte Auskunft über Medientypen eingesehen werden. Dies wurde für die Bestimmung der erlaubten Upload-Dateien auf der Seite "Share" benötigt.
Codeauszug:

```
<input type="file" id="upload" name="uploadFile" accept="image/*,
application/pdf" />
```

Hier wurde ein Upload für alle Image Datentypen sowie für PDF festgelegt. Denn StudyBuddy möchte den Austausch von z. B. Fotos von Lernunterlagen, aber auch von PDFs ermöglichen.

mdn web docs
https://developer.mozilla.org/en-US/docs/Web/CSS

Diese Webseite bietet ausführliche Erklärungen zu den Funktionsweisen, dem Syntax und der Browser-Kompatibilität von HTML- und CSS-Elementen. Deshalb wurde diese Webseite zur Schaffung eines tieferen Verständnisses für die Anwendung und die korrekte Implementierung einiger CSS-Elemente verwendet. Beispielhaft können hier das aside-Element oder der Border-Style genannt werden.

https://developer.mozilla.org/en-US/docs/Web/HTTP

Hier wurden Erklärungen und Hilfestellungen zum Umgang mit HTTP entnommen.

RealFaviconGenerator
https://realfavicongenerator.net/

Diese Seite kann zur Erstellung von eines Favicon in verschiedenen Dateiformaten verwendet werden. Für dieses Projekt wurden die Dateien favicon.svg und favicon-96x96.png mithilfe der Applikation erstellt. Grundlage dafür war das zuvor generierte Logo-Design.

svg repo
https://www.svgrepo.com/svg/408497/arrow-03

Svg repo ist eine Bibliothek für svg-Dateien. Von dort wurde der pinke Pfeil der Homepage kopiert und im HTML-Code eingebunden.

W3C Font-Families
https://www.w3.org/Style/Examples/007/fonts.en.html

Die Unterseite zu Font-Families gibt eine Übersicht über das Aussehen verschiedener Font-Families in CSS. Damit konnte das Design der Schriften verglichen und die hier bevorzugte Font-Family "monospace" ausgewählt werden.

W3 Schools
https://www.w3schools.com/

Die Lernplattform W3 Schools wurde zum Lernen und Testen genutzt. Auf der einen Seite konnten, mittels der ausführlichen und gut strukturierten Beispiele, neue Kenntnisse zur Funktionsweise von HTML und CSS gewonnen werden. Auf der anderen Seite wurden mithilfe des integrierten "Try-it-Yourself"-Editors die Auswirkungen von unterschiedlichen Programmbestandteilen getestet.

OpenJS Foundation, Express.js
https://expressjs.com/en/guide/routing.html

Hier wurde auf die Dokumentation des verwendeten Express-Moduls zurückgegriffen.

MongoDB
https://www.mongodb.com/docs/manual/introduction/

Hier wurde auf die Dokumentation der verwendeten MongoDB-Datenbank zurückgegriffen.

MongoPlayground https://mongoplayground.net/

Diese Webseite wurde zum schnellen Austesten von MongoDB-Methoden genutzt.

Weitere Anwendungen

Microsoft PowerPoint
PowerPoint wurde zur Erstellung der Kompositionsdiagramme und zur Skalierung von Grafiken verwendet.

Figma
Figma wurde zur Erstellung der schematischen Veranschaulichungen der HTTP-Kommunikation genutzt.

Abbildungsverzeichnis

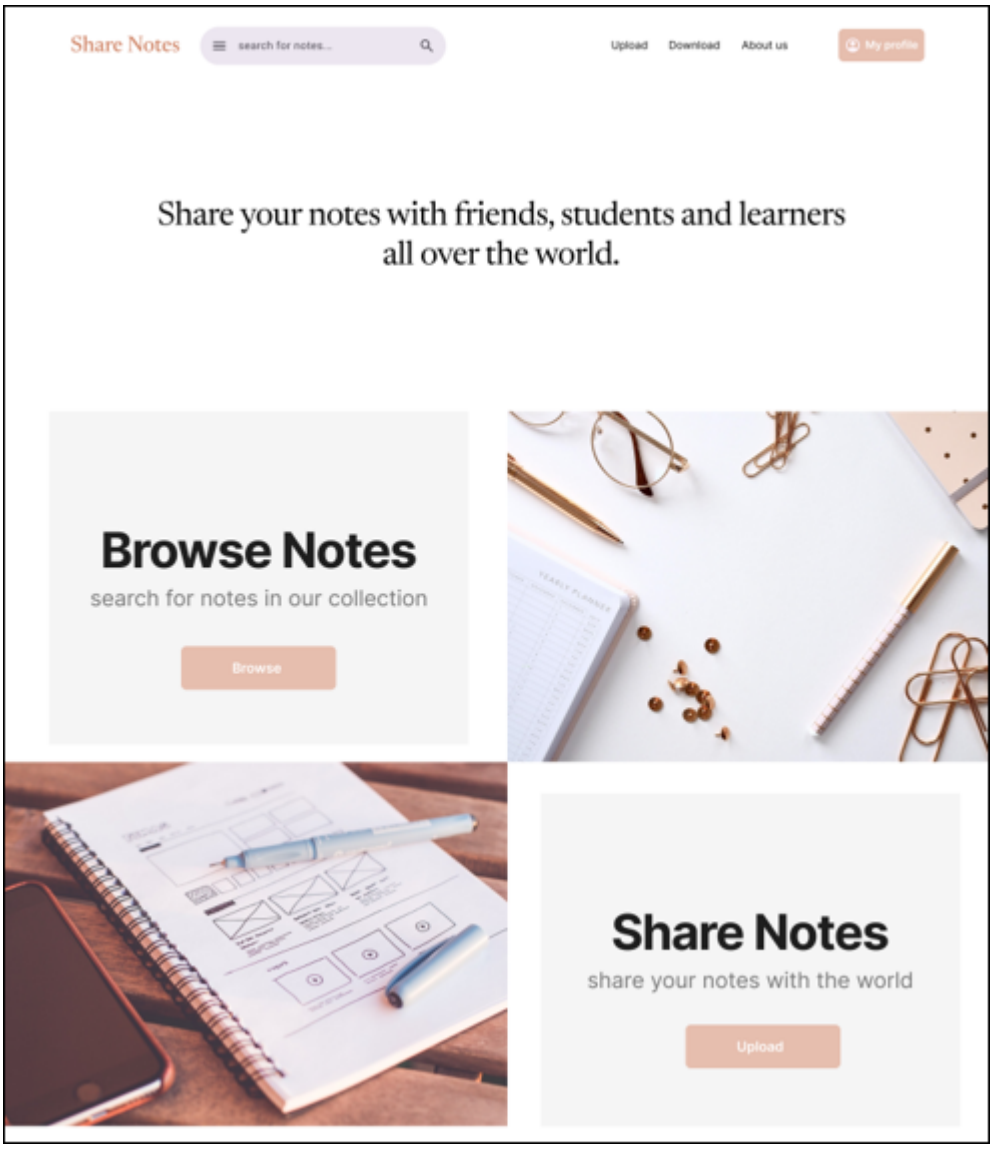

Abb. 2: Sitemap

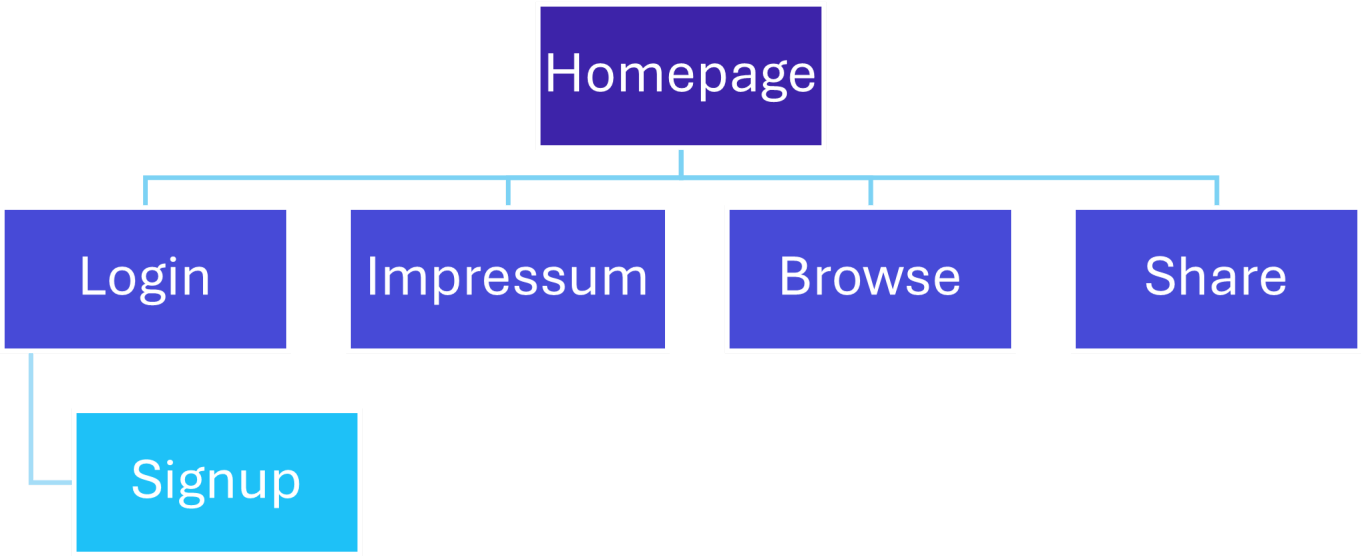

Abb. 3: Kompositionsdiagramm Homepage

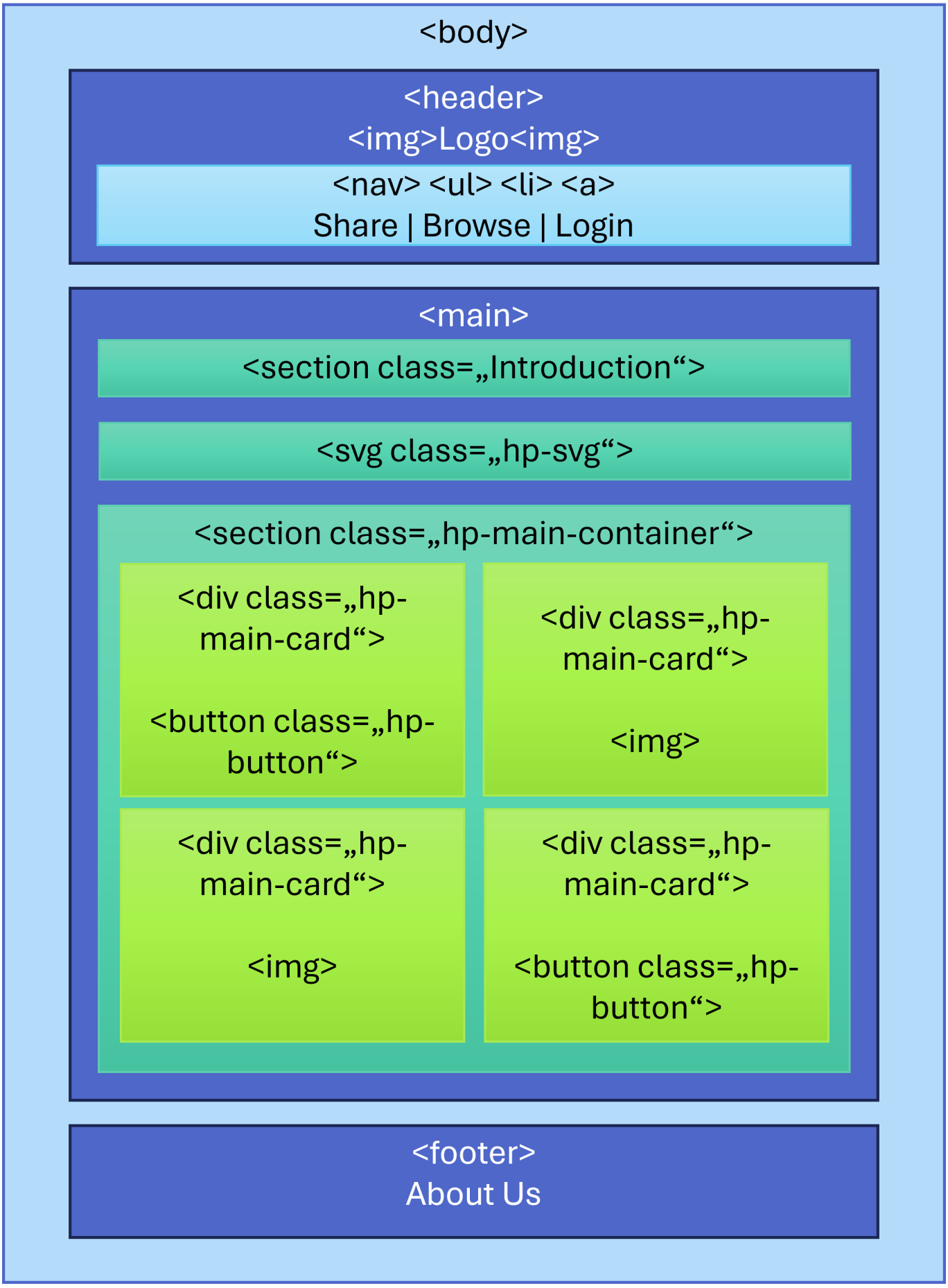

Abb. 4: Kompositionsdiagramm Login

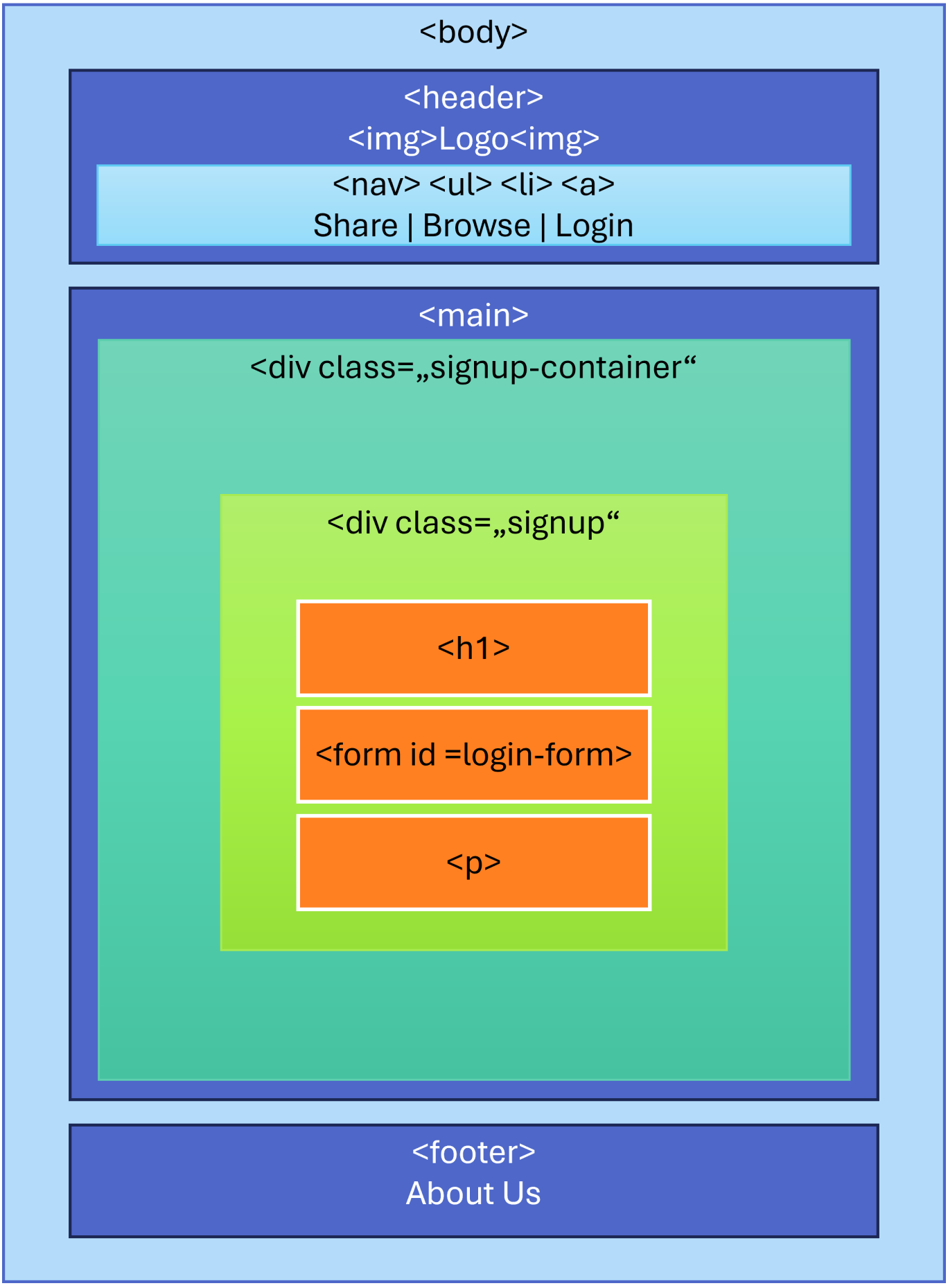

Abb. 5: Kompositionsdiagramm Signup

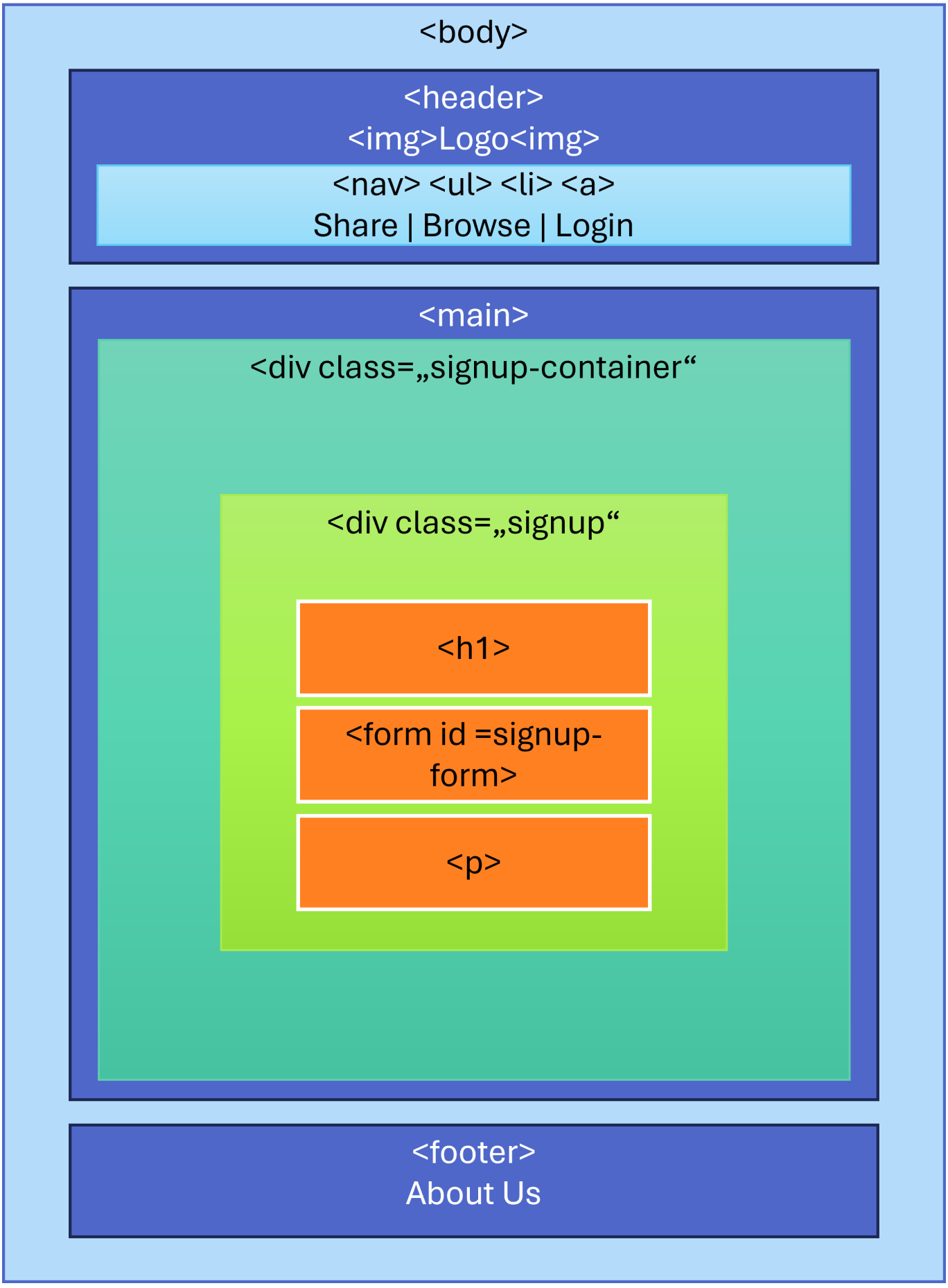

Abb. 6: Kompositionsdiagramm Impressum

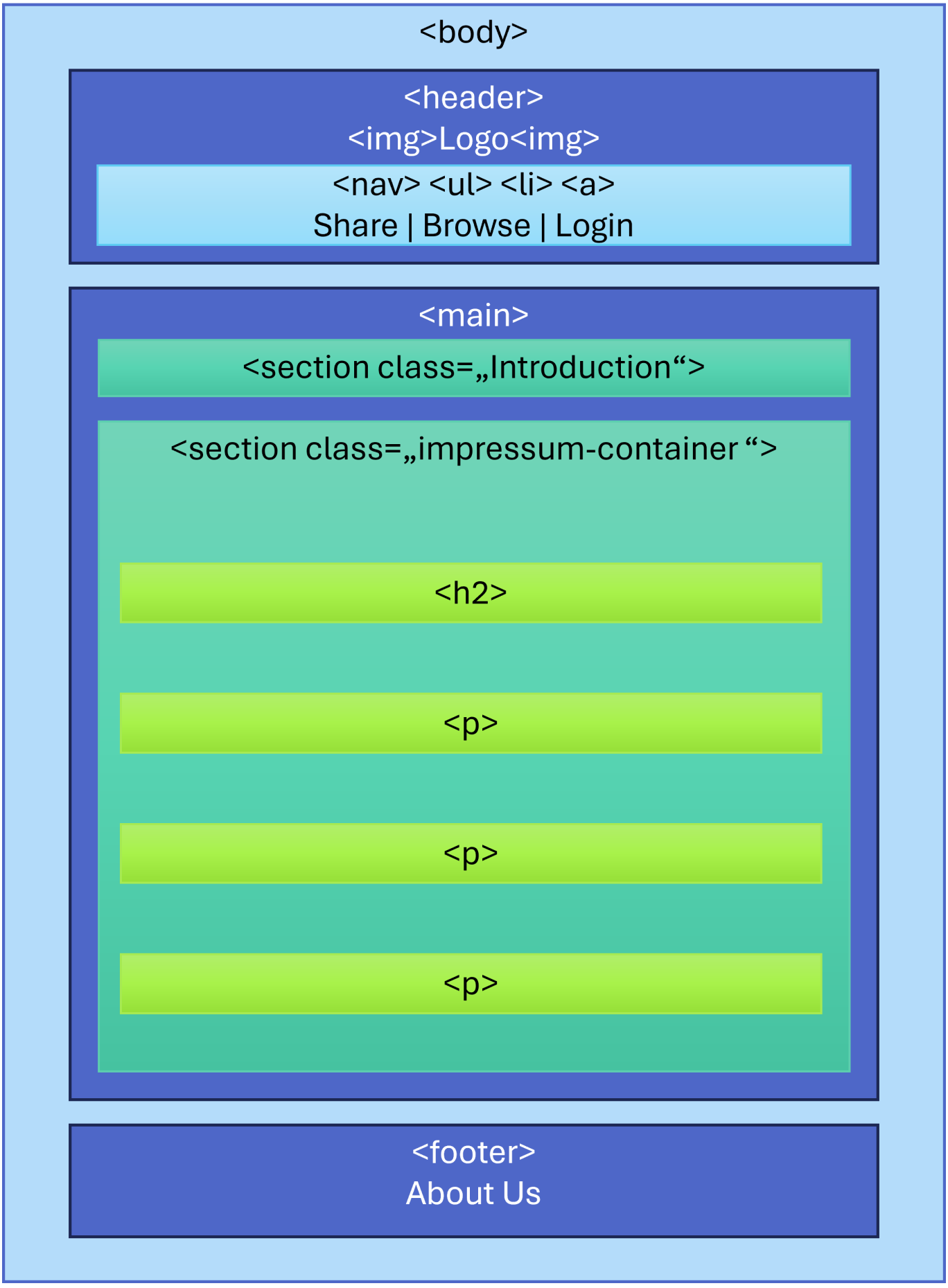

Abb. 7: Kompositionsdiagramm Browse

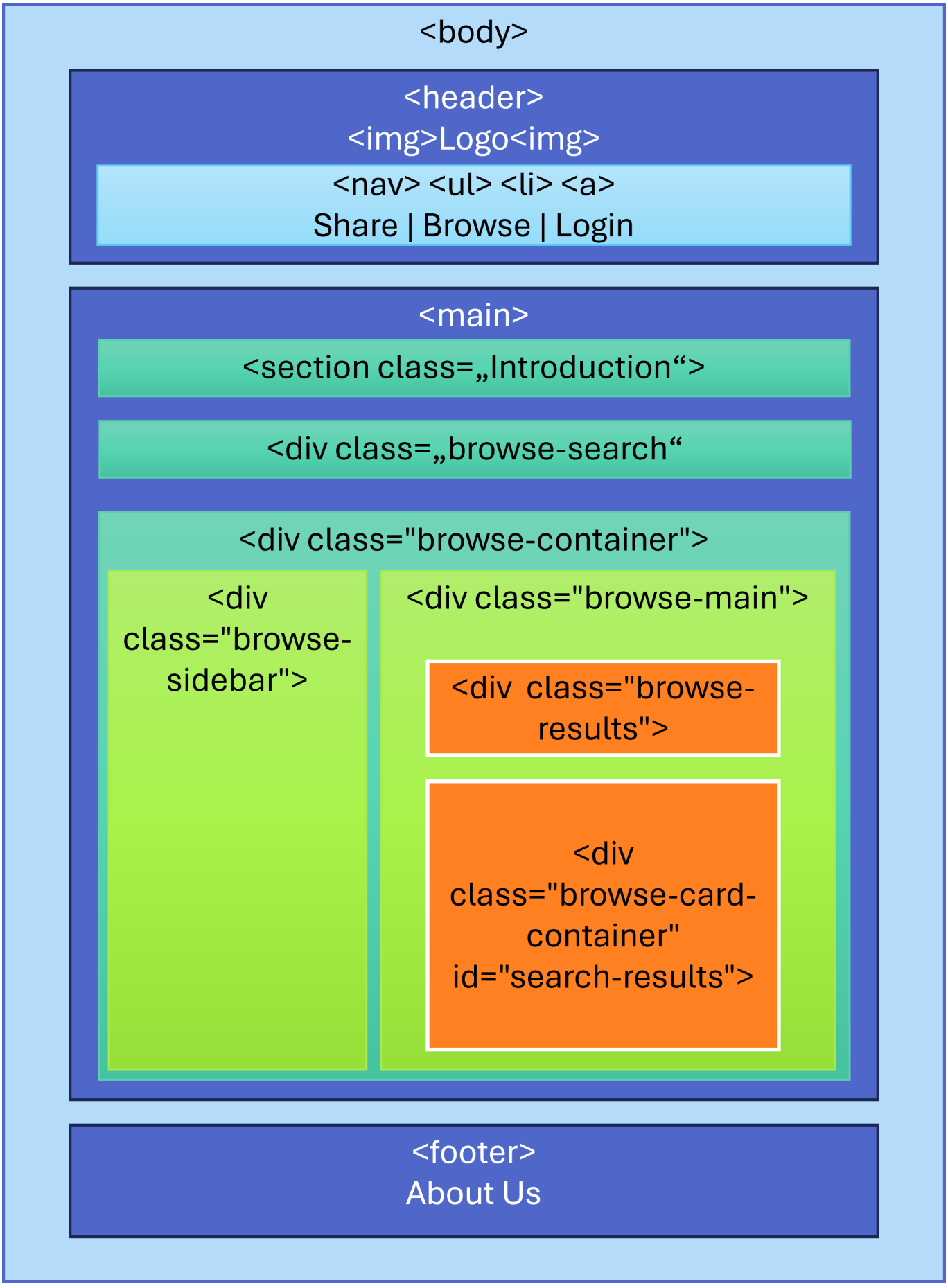

Abb. 8: Kompositionsdiagramm Share

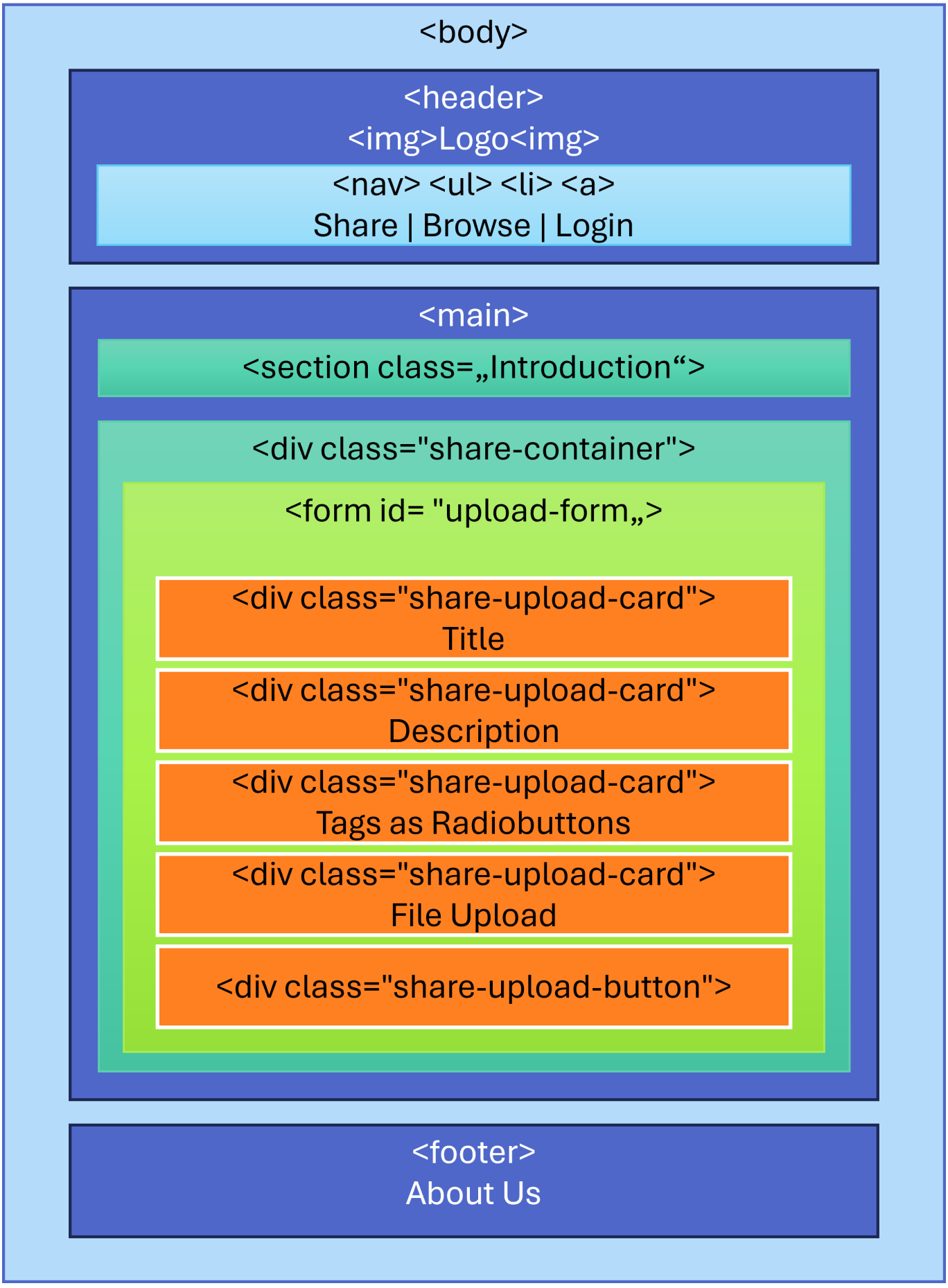

Abb. 9: Browse-Schema

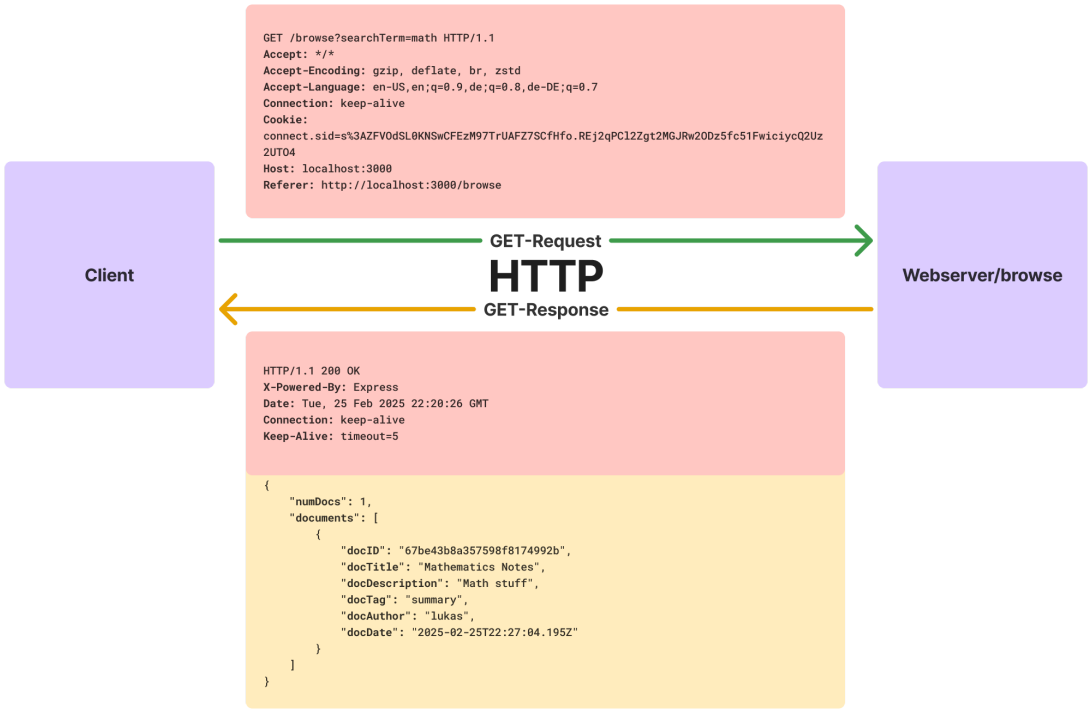

Abb. 10: Download-Schema

Abb. 11: Upload-Schema

Stichwortverzeichnis

API: Application Programming Interface. Eine Schnittstelle, die es ermöglichte, verschiedene Software-Komoponenten miteinder kommunizieren zu lassen.

Alert: Browserfunktion zur Anzeige von Nachrichten an den Benutzer.

Asynchron: Hier: async/wait. eine Programmiertechnik, um auf asynchrone Operationen wie Netzwerkaufrufe (z.B. `fetch()`) zu warten, ohne den Ablauf des Programms zu blockieren

Backend: Serverseitiger Teil der Anwendung, der Anfragen des Frontends entgegennimmt, verarbeitet und Antworten gibt.

Catch: methode in JavaScript zur Fehlerbehandlung. Wird hier verwendet, um Netzwerkfehler bei Anfragen an das Backend abzuhängen.

CSS:
Cascading Style Sheet

Credentials: Eine Einstellung in `fetch()`, die angibt, dass unter anderem Cookies bei Anfragen an das Backend gesendet werden, um Sessions zu validieren.

Cookie: Kleine Datei, die auf dem Client gespeichert wird. Enthält Session-Informationen und wird hier genutzt um Anmeldung des Benutzers zu verfolgen, damit User angemeldet bleibt

DOM: Document Object Model, eine Schnittstelle zur Manipulation und Darstellung von HTML-Dokumenten. Wird hier für den Zugriff auf HTML-Elemente verwendet.

Event Listener: Eine Funktion, die auf bestimmte Ereignisse wie Button-Clicks oder Formular-Submits reagiert.

Fetch(): Methode zum Senden von HTTP_Anfragen an das Backend.

FormData: Ein JavaScript-Objekt zum Erstellen von Key-Value-Paaren, um Formulardaten zu senden.

GET-Request: HTTP-Methode um Daten vom Server zu erhalten.

GateKeeper-Funtkion: Funktion, die überprüft, ob Benutzer eingelogg ist, bevor er auf bestimmte Seiten (hier: /share) zugreifen darf.

HTTP-Statuscodes: Codes, die die Antword des Servers auf eine Anfrage darstellen.

HTML:
Hypertext Markup Language

JSON: Ein Datenformat zur Übertragung von Daten zwischen Client und Server.

LocalStorage: Web-API, die es ermöglicht, Daten lokal im Browserdes Benutzers zu speichern. Hier wird `locasStorage.setItem() verwendet, um den firstName zu speichern.

Middleware: Eine Exüress.js-Funktion, die verwendet wird, um Anfragen zu verarbeiten, bevor sie den Endpunkt erreichen.

Model (Mongoose):
Eine Klasse, die die Instanziierung von Objekten ermöglicht, welche einem MongoDB-Schema entsprechen.

MongoDB:
Eine NoSQL-Datenbankplattform.

POST-Request: HTTP-Methode, um Daten an den Server zu senden.

Route: Eione Definition der URL-Endpunkte.

Schema (Mongoose):
Ein Muster oder eine Vorlage, die die Struktur eines MongoDB-Dokuments beschreibt. Es definiert Felder, Datentypen und andere Eigenschaften der Dokumente.

Session-Management: Methode, um Benutzer-Sitzungen zu verfolgen. Wird hier durch Cookies realisiert.

SVG:
Scalable Vector Graphics. Dieses Dateiformat stellt vektorbasierte Grafiken dar. Es eignet sich gut für den Einsatz im Webdesign, da es eine Skalierung ohne Qualitätsverlust ermöglicht. Weiterhin kann eine svg-Datei als svg-Element in HTML eingebunden und mit CSS bearbeitet werden.

Try-Catch: Ein Konstrukt zum Abfangen von Fehlern.