StudyBuddy

Inhalts- und Dateiverzeichnis

▶ Dateiverzeichnis

```
/StudyBuddy
|-- /config
|   |-- config.js
|   |-- nginx.conf
|
|-- /src
|   |-- /controllers
|   |   |-- userController.js
|   |   |-- productController.js
|   |
|   |-- /models
|   |   |-- userModel.js
|   |   |-- productModel.js
|   |
|   |-- /routes
|   |   |-- userRoutes.js
|   |   |-- productRoutes.js
|   |
|   |-- /middleware
|   |   |-- authMiddleware.js
|   |   |-- errorHandling.js
|   |
|   |-- /utils
|   |   |-- helper.js
|   |
|   |-- app.js
|
```

```
|-- /public
|   |-- /css
|   |   |-- main.css
|   |
|   |-- /js
|   |   |-- main.js
|   |
|   |-- /images
|   |   |-- logo.png
|
|-- /scripts
|   |-- migrate.js
|   |-- seed.js
|
|-- /static
|   |-- /html
|   |   |-- index.html
|   |   |-- user.html
|   |   |-- product.html
|
|-- .env
|-- .gitignore
|-- package.json
|-- package-lock.json
|-- README.md
```

- Aufteilung server- und clientseitig?
- Routing
- Aufbau verschiedener subpages
- Redirecting User zu URL mit Tags und searchTerm?

Serverseitige Implementierung

Grundsätzliche Anforderungen

Das Backend soll einen schnellen und effizienten Umgang mit den Requests der Users 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:

`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.

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 mit Hilfe des `User`-Modells die `userID`s um die `firstName`-Attribute der User ergänzt, und ein Mapping der aktuellen Attribut-Werte zu den im Frontend erwartetetn 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.

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 angfragten 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;
```

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

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 gegegben 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 16MB hervorrufen):

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

module.exports = router;
```

`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 Hashig des vom User vergebenen Passworts mit Hilfe 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 sichgerstellt, 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.

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.

`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.

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 16MB beschränkt (konkreter wird die Größe jedes BSON-Dokuments in der Datenbank auf 16MB 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 16MB zu unterstützen [^1]. Die Praktikabilität dessen im Vergleich zum einfachen Speichern der Dateipfade müsste separat weiter evaluiert werden.

References

[^1]: MongoDB, Inc. (2024). GridFS for Self-Managed Deployments. MongoDB Manual.