**Dokumentation der Eigenleistung – Backend & Systemintegration**

Im Rahmen des Karteikarten-Projekts war ich für einen großteil für die Entwicklung des Backends und die technische Systemintegration verantwortlich, ebenfalls für die Logik hinter des Logins und Regestrieren, die sicherheit dahinter und die anzeige der Topüics, subtopics und flashcards. Dabei habe ich das Backend von Grund auf konzipiert, entwickelt und getestet. Darüber hinaus habe ich die Anbindung an das Frontend übernommen und einen stabilen Entwicklungs- und Produktions-Workflow mit Docker realisiert sowie eine Sichere verbindung zwischen frotend und backend sowie sicheres einlioggen gewährlistet und diue datenanzeige unf verwaltung..

**1. Motivation und Grundentscheidungen**

Motivation und Grundentscheidungen

Zu Beginn des Projekts war es mir besonders wichtig, dass wir die Modularität der Anwendung sowohl im Frontend als auch im Backend von Anfang an in den Mittelpunkt stellen. Schon im ersten Semester haben wir viel Wert darauf gelegt, unseren Code so aufzubauen, dass die einzelnen Komponenten und Funktionen klar voneinander getrennt und gut wiederverwendbar sind. Dies erleichtert nicht nur die Entwicklung im Team, sondern sorgt auch für eine bessere Wartbarkeit und Nachvollziehbarkeit des Projekts.

Im zweiten Semester, als der Fokus auf die Entwicklung des Backends gelegt wurde, standen wir vor der grundlegenden Entscheidung, wie wir unseren Server und die technische Infrastruktur gestalten möchten. Wir haben uns bewusst für die Nutzung von Docker entschieden. Das lag einerseits daran, dass wir bereits erste Erfahrungen mit Docker gesammelt hatten, andererseits aber auch daran, dass sich Anwendungen mit Docker am Ende besonders leicht und zuverlässig auf verschiedenen Systemen deployen lassen – gerade für ein Projekt, das später auf einem Server produktiv laufen soll, ist das ein großer Vorteil.

Im Frontend haben wir großen Wert darauf gelegt, moderne Angular-Best-Practices anzuwenden. Dazu zählt insbesondere der Einsatz von Observables für asynchrone Datenflüsse. Das sorgt dafür, dass die Anwendung sehr flexibel auf Änderungen reagieren kann und die Datenhaltung stets aktuell ist. Ein weiteres Ziel war es, den Code so übersichtlich und verständlich wie möglich zu gestalten – sowohl durch saubere Struktur als auch durch ausführliche Kommentare.

Die Entscheidung für Angular als Framework fiel uns vor allem deshalb leicht, weil wir während unserer Praxisphase bereits praktische Erfahrungen mit Angular sammeln konnten. Zwar ist der Einstieg in Angular nicht ganz einfach, da die Architektur zunächst komplex und ungewohnt wirken kann. Sobald man sich jedoch in das Framework eingearbeitet hat und die Grundprinzipien verstanden sind, zeigt sich, dass die Modularität und die klare Struktur von Angular enorme Vorteile bieten. Anfangs wirkt die Struktur zwar verschachtelt und vielleicht sogar unübersichtlich, aber gerade bei größeren Projekten wie diesem bewährt sich die saubere Trennung von Komponenten, Services und Modulen.

**Funktion: Anzeige, Bearbeitung und Verwaltung der Themen, Unterthemen und Karteikarten**

Im Rahmen des Projekts habe ich die Kernfunktion entwickelt, die es ermöglicht, **Themen**, **Unterthemen** (Subtopics) und **Karteikarten** (Flashcards) in der Anwendung anzuzeigen, zu erstellen, zu bearbeiten und zu löschen. Diese Funktion ist das Herzstück der gesamten Anwendung und bildet die Grundlage für die strukturierte Organisation und effiziente Nutzung der digitalen Karteikarten.

**Idee und Zielsetzung**

Meine Grundidee dabei war, die Daten nicht einfach nur als „flache Liste“ zu speichern, sondern klar **gestaffelt** und hierarchisch zu strukturieren:

* **Themengebiete** (z.B. „Mathematik“ oder „Informatik“) bilden die oberste Ebene.
* Jedes Thema enthält beliebig viele **Unterthemen** (z.B. „Algebra“, „Geometrie“).
* Jeder Unterpunkt kann wiederum eine eigene Sammlung von **Karteikarten** mit Frage und Antwort enthalten.

Dadurch lässt sich die Lernumgebung individuell und übersichtlich gestalten. Für mich war besonders wichtig, dass die Nutzer ihre Inhalte einfach verwalten, anpassen und gegebenenfalls auch wieder löschen können.

**Architektur und technische Umsetzung**

**Frontend:**  
Die Umsetzung im Frontend erfolgte modular nach Angular-Best-Practices, um die Wiederverwendbarkeit und Wartbarkeit sicherzustellen. Die Anzeige und Bearbeitung der Daten erfolgt jeweils in eigenen Komponenten:

* TopicsComponent für Themen
* SubtopicsComponent für Unterthemen
* FlashcardsListComponent für Karteikarten

Die Kommunikation mit dem Backend geschieht über einen zentralen Service (GetDataService), der alle notwendigen HTTP-Requests kapselt (abrufen, anlegen, bearbeiten, löschen).  
Die Komponenten nutzen **Observables**, um reaktiv auf Datenänderungen zu reagieren – z.B. wird beim Hinzufügen oder Bearbeiten sofort die Anzeige aktualisiert.

**Backend:**  
Das Backend ist in Node.js (TypeScript) mit Express und TypeORM realisiert. Die Datenbankstruktur bildet die Hierarchie der Themen, Unterthemen und Karteikarten 1:1 ab (Relationen). Es gibt für jede Ebene eigene Endpunkte (REST-API), z.B.:

* /api/topics – Themen verwalten
* /api/topics/:topicId/subtopics – Unterthemen zu einem Thema verwalten
* /api/topics/:topicId/subtopics/:subtopicId/flashcards – Karteikarten eines Unterthemas verwalten

Die API ist so konzipiert, dass alle CRUD-Operationen (Create, Read, Update, Delete) sowohl für Themen, Unterthemen als auch für Karteikarten sauber abgebildet und abgesichert sind.

**Wichtige Design-Entscheidungen**

* **Hierarchische Struktur:** Die Unterteilung in Themen, Unterthemen und Karteikarten war von Anfang an gesetzt, um das Lernen und Verwalten großer Mengen an Karteikarten übersichtlich zu gestalten.
* **Modularität:** Ich habe Wert darauf gelegt, dass jede Funktion (z.B. Hinzufügen, Bearbeiten, Löschen) in einer eigenen kleinen Datei bzw. Komponente gekapselt ist, statt große, schwer wartbare Dateien zu haben.
* **Direkte, reaktive Anzeige:** Über Observables werden Änderungen (z.B. neue Karte angelegt oder gelöscht) direkt in der UI sichtbar, ohne dass man die Seite neu laden muss.
* **Klare API-Struktur:** Die Endpunkte spiegeln die Datenstruktur und die „Besitzverhältnisse“ exakt wider – z.B. kann man keine Karteikarte ohne Bezug zu einem Subtopic anlegen.

**Zusammenspiel von Frontend und Backend**

Sobald ein Nutzer eine Aktion im Frontend ausführt (z.B. ein Thema anlegt, eine Karte bearbeitet oder eine Unterthema löscht), wird ein passender API-Call abgesetzt. Das Backend verarbeitet die Anfrage, nimmt die Änderungen in der Datenbank vor und gibt das aktualisierte Objekt oder einen Status zurück.  
Die Daten werden anschließend direkt im Frontend angezeigt bzw. aktualisiert. Dank Angulars reaktiver Programmierung werden auch parallele Änderungen nahtlos sichtbar.

**Herausforderungen und Lösungen**

Gerade beim Wechsel vom rein frontend-basierten Ansatz (im ersten Semester) auf die echte Backend-Anbindung (im zweiten Semester) gab es einige größere Herausforderungen:

* Die Datenmodelle und Typen mussten zum Teil grundlegend angepasst werden, weil das Backend andere Datenstrukturen zurückliefert als das vorherige JSON-Server-Mockup.
* Die Services und Komponenten mussten entsprechend überarbeitet werden, um die neuen Datentypen korrekt zu verarbeiten und darzustellen.
* Das Error-Handling und das Nachladen von Daten wurden verbessert, damit die UI robust auf alle Eventualitäten reagiert.

**Login und Registrierung**

Ein zentrales Feature unseres Karteikarten-Projekts ist die Möglichkeit für Nutzer, sich zu registrieren, einzuloggen und ihre persönlichen Daten geschützt zu verwalten. Diese Funktion wurde in enger Zusammenarbeit umgesetzt und ist ein gutes Beispiel für die Verbindung von Frontend- und Backend-Entwicklung.

**Ausgangslage und erste Iteration**

Im ersten Semester wurde zunächst eine Mock-Authentifizierung direkt im Frontend mit einem lokalen JSON-Server von Luis umgesetzt. Diese Lösung war für die schnelle Entwicklung und erste Tests zwar ausreichend, bot jedoch keinerlei Sicherheit und war für die spätere Produktivnutzung nicht geeignet. Es gab keine echte Nutzerverwaltung, keine Passworthashes, keine sicheren Sessions – im Prinzip war alles "Fake Login".

**Ziel: Sichere, skalierbare Authentifizierung**

Mit Beginn der Backend-Implementierung im zweiten Semester habe ich die Authentifizierung von Grund auf neu konzipiert – mit Fokus auf **Sicherheit, Skalierbarkeit und Best Practices**. Die wichtigsten Ziele dabei:

* **Echte Nutzerverwaltung** mit Persistenz in einer echten MySQL-Datenbank (via TypeORM)
* **Sicheres Passwort-Handling**: Passwörter niemals im Klartext speichern, sondern mit einer modernen Hash-Funktion
* **Session Management** ohne klassische Cookies, sondern über stateless JWT-Token für mehr Sicherheit und bessere Skalierbarkeit (z.B. für spätere mobile Apps)
* **Zentrale Validierung** aller Eingaben, um Angriffsvektoren wie SQL-Injection oder schwache Passwörter zu verhindern

**Entscheidung für JWT und Argon2**

* **JWT (JSON Web Token):**  
  Ich habe mich bewusst für JWT entschieden, weil dieses Verfahren moderne Authentifizierung "stateless" und skalierbar ermöglicht. Das bedeutet, der Server muss keine Sessions speichern, sondern jeder Request enthält ein Token, das von der Backend-API verifiziert werden kann. Dies ist ideal für REST-APIs, wie wir sie nutzen, und vereinfacht das Hosting und die spätere Erweiterung.
* **Argon2 für Passwort-Hashing:**  
  Sicherheit stand im Vordergrund. Deshalb nutze ich **Argon2**. Jedes Passwort wird vor dem Speichern automatisch mit einem einzigartigen Salt gehasht, sodass selbst bei Datenbank-Leak keine Klartext-Passwörter kompromittiert werden.

**Technische Umsetzung und Ablauf**

**Frontend (Angular):**

* Luis hat das **Aussehen, die UX und die Komponenten für Login und Registrierung** entwickelt (bis Commit 94e5fa9). Hier wurde bereits auf eine saubere und intuitive Benutzerführung geachtet.
* Nach Fertigstellung des Backends habe ich die Authentifizierungslogik nahtlos integriert:
  + Entwicklung eines **Auth-Services**, der die Anmeldedaten an die REST-API sendet und die erhaltenen JWT-Tokens sicher im LocalStorage speichert.
  + Implementierung eines **HTTP-Interceptors**, der bei allen geschützten Requests automatisch das Token im Header mitsendet („Bearer Token“).
  + Einführung eines **AuthGuards** zum Schutz der Routen, sodass nicht eingeloggte Nutzer keinen Zugriff auf geschützte Bereiche haben.
  + Zentrale Logout-Logik, die alle Authentifizierungsdaten löscht.

**Backend (Node.js/Express):**

* Entwicklung des gesamten Authentifizierungs- und User-Management-Stacks:
  + **Register-Endpoint:** Nimmt neue Nutzerdaten entgegen, prüft sie (mit Zod auf Mindestlänge, Email-Format etc.), hasht das Passwort mit Argon2 und legt den neuen Nutzer sicher in der Datenbank an.
  + **Login-Endpoint:** Prüft die Kombination aus Username/E-Mail und Passwort, verifiziert den Hash und gibt im Erfolgsfall ein frisch signiertes JWT-Token aus.
  + **Middleware** für die Prüfung aller geschützten Endpunkte: Jedes Token wird auf Gültigkeit und Unverfälschtheit geprüft, bevor Zugriff auf Daten gewährt wird.
  + **Sichere Fehlerbehandlung** und Logging aller Anmelde- und Registrierungsversuche.

**2. Backend-Architektur & Sicherheit**

* **Struktur & Modularität:**  
  Ich habe das Backend nach klaren Schichten aufgebaut:
  + config/ für Umgebungsvariablen und Settings
  + controllers/ für die einzelnen Routen-Handler
  + entities/ als Abbild der Datenbanktabellen (User, Topic, Subtopic, Flashcard)
  + middleware/ für wiederverwendbare Funktionen wie Authentifizierung, Fehlerbehandlung etc.
  + services/ für Business-Logik und Wiederverwendung
  + routes/ für die eigentliche API-Struktur  
    Jede Komponente ist in einer eigenen Datei ausgelagert.
* **Sicherheit:**  
  Das System verwendet **JWT-Token** für Authentifizierung und Sessions, und Passwörter werden mit Argon2 gehasht. Damit erfüllen wir aktuelle Sicherheitsstandards. Alle Requests werden serverseitig auf gültige Daten (über Zod) geprüft.

**3. Erfahrung: Probleme und Herausforderungen beim Umstieg Frontend → Backend**

Ein sehr großer Teil der Arbeit – und auch der größte Zeitfresser – war die **Umstellung vom bisherigen Mock-Server/Frontend (erstes Semester) auf das neue, echte Backend**.  
Im ersten Semester hatten wir bereits das komplette Frontend gebaut und testweise mit einem lokalen JSON-Server (ohne echtes Backend) von Luis angebunden.  
Das bedeutete allerdings, dass **Datenstrukturen, Rückgabeformate und die gesamte API im zweiten Semester komplett anders waren**.  
Ich habe die Verantwortung übernommen, die gesamte Kommunikation so umzubauen, dass alle Datentypen, Models, Services und Methoden jetzt mit dem richtigen Backend funktionieren.  
Das war aufwändiger als erwartet:

* **Fast alle Datentypen mussten geändert werden** (ID-Typen, Namen von Feldern, z.B. von id (string) auf id (number), oder von userName auf username usw.)
* Es mussten zahlreiche Services und Komponenten umgebaut und angepasst werden.
* Insbesondere die Fehlerbehandlung war mit echten Backend-Fehlermeldungen komplexer.

Im Nachhinein hätte es vielleicht geholfen, im ersten Semester schon enger am späteren Backend-Modell zu bleiben – aber diese Erfahrung gehört für mich zur echten Projektarbeit dazu und ich habe daraus sehr viel gelernt. Das war ein wichtiger Schritt, um das System wirklich produktionsreif und robust zu machen.

**4. Technische Umsetzung & Best Practices**

* **Codequalität:**  
  Mir war besonders wichtig, **keine großen „God-Classes“ oder Monsterdateien** zu bauen, sondern den Code sauber aufzuteilen:
  + **Services** für wiederverwendbare Funktionen
  + **Core/Shared/Feature**-Struktur im Frontend für Übersichtlichkeit und Wiederverwendung
  + **Alle Datenbankzugriffe** und API-Endpunkte sind sauber validiert und dokumentiert
* **Security & DevOps:**  
  Alle Secrets und Umgebungsvariablen werden in einer .env-Datei gehalten – kein Secret im Quellcode.  
  Die gesamte Anwendung ist durch Docker/Compose in wenigen Minuten startklar – und damit im Prinzip sofort bereit für den produktiven Betrieb auf jedem Server, der Docker unterstützt.

**5. Deployment & Erweiterbarkeit**

Ich habe darauf geachtet, dass die Anwendung nicht nur auf meinem Rechner läuft, sondern **auf jedem beliebigen System mit Docker**. Das Deployment ist damit maximal einfach:

* **docker-compose up** genügt, um die Datenbank und das Backend zu starten.
* Das Frontend ist unabhängig und kann z.B. auf GitHub Pages oder jedem Static-Host laufen.
* Später könnte man ohne großen Aufwand weitere Dienste (z.B. Monitoring, Backups, CI/CD) in die Docker-Umgebung integrieren.

**6. Fazit & persönliche Learnings**

Die größte Herausforderung war für mich, **die vollständige Integration des neuen Backends in ein bereits bestehendes, komplexes Frontend** – inklusive aller kleinen Anpassungen, die dadurch nötig wurden.  
Die Entscheidung für Docker und ein modulares, sicheres Backend-Design war im Nachhinein absolut richtig und hat den Grundstein für ein robustes, flexibles System gelegt.  
Ich bin sehr zufrieden damit, wie sich das Projekt entwickelt hat – und sehe darin eine solide Basis, auf der sich zukünftige Features oder eine Skalierung problemlos aufbauen lassen.

**Hinweis:**  
Im Team wurde die Aufgabenverteilung klar eingehalten:  
Luis hat z.B. das initiale Frontend-Design, das Styling, sowie die UI-Komponenten für die Authentifizierung entwickelt. Ich habe mich um das technische Backend, die Authentifizierung, Datenhaltung, sowie die Gesamtintegration gekümmert.

**Falls du einzelne Codeabschnitte (z.B. Dockerfile, Auth-Service, eine bestimmte Middleware) als Anhang dokumentieren willst, gib mir einfach Bescheid, dann formatiere ich das noch für dich.**