Structuring Educational Media into a Searchable Digital Database with API Access

A prototype for educational media management

Bachelor's thesis report within the Computer Engineering programme

HAMPUS RAMSTEN
JOAKIM TUOVINEN

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg 2026
www.chalmers.se

Bachelor's Thesis Report 2026

Structuring Educational Media into a Searchable Digital Database with API Access

A prototype for educational media management

Hampus Ramsten
Joakim Tuovinen

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg 2026

Structuring Educational Media into a Searchable Digital Database with API Access
A prototype for educational media management
Hampus Ramsten
Joakim Tuovinen

Supervisor: John J. Camilleri, Chalmers University of Technology
Examiner: Thierry Coquand, Chalmers University of Technology

Bachelor's thesis report 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden

Cover image: A picture of the PostgreSQL logo.

Written in LaTeX
Gothenburg 2026

Abstract

Digimar is an educational platform for maritime communication that uses several types of learning material, such as videos, transcripts, presentations, documents, images and chatbot content. Before this project, the material lacked a centralized technical interface, which made it harder to search, organize and reuse content across different frontend applications.

This thesis presents the design and implementation of a searchable digital database with API access for Digimar. The system consists of a Go based HTTP REST API, a PostgreSQL database, a static frontend console and a local Docker Compose setup. The API supports several resource types, including chapters, sections, content items, videos, transcripts, PowerPoint files, documents, images and chatbots. It also includes search across metadata and text fields, role based API keys, health and readiness endpoints, and documentation for the API and data model.

The result is a working prototype that provides one shared backend for managing Digimar learning material. Most of the planned goals were completed, including storage of the planned content types, REST API access, search, access control, Docker based local development, documentation and a frontend tool for manual operation. The planned PDF to markdown conversion automation was not completed and remains future work. Other future improvements include more formal database migrations, improved file storage, search ranking based on user preferences and production concerns such as backups, caching and monitoring.

Keywords: Digimar, PostgreSQL, Go, REST API, searchable database, educational media.

Sammanfattning

Digimar är en utbildningsplattform för maritim kommunikation som använder flera typer av läromaterial, till exempel videor, transkriptioner, presentationer, dokument, bilder och chatbot-innehåll. Innan detta projekt saknade materialet ett centraliserat tekniskt gränssnitt, vilket gjorde det svårare att söka, organisera och återanvända innehåll i olika frontend-applikationer.

Detta examensarbete presenterar designen och implementationen av en sökbar digital databas med API-åtkomst för Digimar. Systemet består av ett HTTP REST API byggt i Go, en PostgreSQL-databas, en statisk frontend-konsol och en lokal Docker Compose-miljö. API:t stödjer flera resurstyper, bland annat kapitel, sektioner, innehållsposter, videor, transkriptioner, PowerPoint-filer, dokument, bilder och chatbots. Systemet innehåller även sökning i metadata och textfält, rollbaserade API-nycklar, hälso- och redo-kontroller samt dokumentation för API och datamodell.

Resultatet är en fungerande prototyp som ger ett gemensamt backend-system för hantering av Digimars läromaterial. De flesta planerade mål uppnåddes, inklusive lagring av planerade innehållstyper, REST API-åtkomst, sökning, åtkomstkontroll, lokal utveckling med Docker, dokumentation och ett frontend-verktyg för manuell hantering. Den planerade automatiseringen för att konvertera PDF-filer till markdown slutfördes inte och kvarstår som framtida arbete. Andra möjliga förbättringar är mer formella databasmigreringar, förbättrad filhantering, sökrankning baserad på användarpreferenser samt produktionsrelaterade delar som säkerhetskopiering, caching och övervakning.

Nyckelord: Digimar, PostgreSQL, Go, REST API, sökbar databas, utbildningsmedia.

Acronyms

The following acronyms are used throughout the report.

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSONB	Binary representation of JSON in PostgreSQL
POC	Proof of Concept
REST	Representational State Transfer
SQL	Structured Query Language
UUID	Universally Unique Identifier

1
Introduction

1.1 Background

Digimar, short for Digital Education for Maritime Communication, is an educational platform focused on maritime communication. It is part of a larger European Erasmus+ project that aims to improve navigational safety by developing digital learning tools and material for routine maritime communication. The project involves several European partners from higher education and the maritime sector. Chalmers is one of the participating partners, and D-Smart is a separate Chalmers project that builds on the material developed within Digimar.

The Digimar platform has developed several types of learning content, mainly videos, video transcripts, PowerPoint presentations, chatbots and interactive exercises. These materials are intended to support maritime communication training for students, shore service operators and other maritime stakeholders. The aim of D-Smart is to reuse and further develop this material as part of a continued digital learning environment for maritime communication.

The existing Digimar content is currently stored in a shared OneDrive folder, which includes the different material. This leads to difficulties in searching, categorizing and reusing current learning materials. Not having a standardized way of accessing the content also makes it difficult to develop and integrate new learning applications, such as web applications, mobile applications, chatbots or other tools that need structured access to the same material.

To manage and organize these resources effectively, D-Smart needs a digital database with API access. Because the existing material is stored in different formats and locations, some content will need to be imported, converted or structured before it can be stored and accessed through the new database.

This thesis project addresses these problems by designing and building an HTTP API. The API is intended to support D-Smart by providing a standardized way to store, retrieve and reuse the educational resources originally developed by Digimar. In parallel, two other D-Smart project groups are working on a new website and a chatbot. The API developed in this thesis is intended to support these projects by giving them structured access to the same learning material through one backend service.

1.2 Purpose

The purpose of this thesis is to design and implement a searchable digital database with API access for D-Smart. The implementation should make it possible to store and retrieve educational resources through one backend service. The API should provide a standardized way of accessing the learning material, so that it can be reused by different learning applications.

1.3 Goals

At the start of the project, the intended system was defined as a searchable database with API access. It should:

1. Store and manage several content types, including videos, presentation slides, transcripts, chatbots, documents, chapters, sections and images.
2. Expose these resources through a REST-oriented HTTP API.
3. Support search across metadata and text fields.
4. Protect read, write, and administrative API access with role-based API keys.
5. Support local, reproducible development with Docker Compose.
6. Provide technical documentation and a frontend tool for manual operation.
7. Script or automation to convert PDF files to markdown format

1.4 Limitations

The project will focus on the functionality of the HTTP REST API and the local database setup. It will not design new course content and will not include advanced user identification beyond API-key based access.

The project will not consider production scale system architecture. This means that load balancing, failover handling and automated backups are outside the scope. Caching will also not be considered in the implementation.

The project will not define final production limits for file sizes. Very large files and more advanced storage solutions are therefore treated as future work.

The project will support a fixed set of content types. Handling new content types and migrating existing data to future versions will not be included in the project scope.

2
Method

2.1 Approach

The thesis is carried out as an iterative project. The work is structured in four stages, beginning with defining the requirements and datamodel, backend API development, frontend development, and evaluation criteria. The implementation is planned to proceed through database design, backend API development, access control, search, frontend support, documentation, and verification.

2.2 Requirements and Data Model

The initial phase focuses on identifying the core requirements of the API. This includes what type of material should be handled, what metadata is required for each type and which operations the system should support. The requirments also include support for search functionality, tags, links between types and access control. Based on these requirements a relational data model is planned in PostgreSQL.

2.3 Backend API

The backend is planned to be implemented as an HTTP REST API in Go, using PostgreSQL as persistent storage. The API is intended to provide CRUD style operations for each content type, allowing content to be created, retrieved, updated and deleted. The backend work also includes API key based access control with read only and read write access. Additionally it should support search, filtering and file streaming for media.

2.4 Collaboration With Other Project Groups

Since two other D-Smart project groups will work in parallel on a website and a chatbot, it will be necessary to communicate with these groups during the project. This communication will be used to understand what type of learning material they need to access, how they need to retrieve it, and what requirements this creates for the API. Feedback from these groups will be considered when defining and adjusting the requirements, data model and API endpoints.

2.5 Search and Fuzzy Search

Search functionality will be included as part of the backend API. The purpose of the search functionality is to make it possible to find educational resources based on relevant fields. The project will also investigate fuzzy search. Fuzzy search means that the system should be able to return relevant results even when the search query does not exactly match the stored content.

2.6 Frontend Support

Frontend development is considered an optional part of the project if time is available. The frontend is planned to be an administrative interface in React for managing the content. The planned functionality for the frontend includes listing, searching, creating, updating content. Additionally it should support a JSON editor view for chatbots and sections.

2.7 Evaluation and Documentation

The final stage focuses on verifying that the system supports the intended Digimar content. This includes checking that related resources, such as a video, transcript, powerpoints, sections and content, can be represented, stored and retrieved through the API.

Technical documentation is also produced as part of this stage. The documentation includes an API specification, a description of the data model, and instructions for running the system locally using Docker Compose.

3
Technical Background

3.1 HTTP REST API

An Application Programming Interface, or API, defines how software components can interact with each other. APIs can hide implementation details and provide a simpler interface for using functionality without requiring the client to know how it is implemented [1]. So for example instead of the client accessing the database directly, the client sends a request to the API which handles the operation and returns a response.

The Hypertext Transfer Protocol, HTTP, is a stateless application-level protocol used for communication between clients and servers [2]. HTTP follows a request-response model, where the client sends a request and the server returns a response [3]. HTTP request methods describe the intended operation of a request, such as retrieving, creating, updating, or deleting resources [4]. This model can be used when designing HTTP-based APIs, where clients interact with serverside resources through request methods and URL paths.

REST, Representational State Transfer, is an architectural style for network-based software systems. It is based on constraints such as client-server separation, stateless communication, cacheability, and a uniform interface [5]. In a REST-oriented API, the system is structured around resources, where resources are identified through paths and accessed through representations rather than by exposing internal implementation details.

3.2 Relational Databases

A relational database organizes data in tables, where data is stored in rows and columns. This structure is intended to make stored information easier to search, retrieve, and relate across tables [6]. In a relational database, primary keys are used to identify rows in a table. Foreign keys are used to reference rows in another table, which maintains relationships and referential integrity between tables [7].

PostgreSQL is an open-source object-relational database system [8]. PostgreSQL supports more flexible data types, such as `JSONB` and `BYTEA`. `JSONB` stores JSON data in a decomposed binary format, while `BYTEA` is used for storing binary strings [9, 10].

PostgreSQL was chosen over a NoSQL database because the project data has clear relationships between resources such as documents, chapters, sections and related content. A relational model makes these relationships explicit through primary keys and foreign keys, which helps keep the stored data consistent. A NoSQL database could store flexible document-like data, but it would make relational constraints and joins less direct. PostgreSQL also still provides flexibility through `JSONB`, so the project can store structured relational data while keeping support for fields that are better represented as JSON.

3.3 Text Search and Trigram Matching

Text search is used to find stored text values that match a search term. A simple search can be based on exact or partial matching, but this only returns results where the search term appears directly in the stored text. In some systems like this API, it is also useful to support fuzzy matching, where similar words or partial matches can return a relevant result.

PostgreSQL provides the `pg_trgm` extension for trigram based text matching. A trigram is a group of three consecutive characters from a string. By comparing the trigrams of two strings, PostgreSQL can estimate how similar the strings are. The extension provides functions and operators such as `similarity()`, `word_similarity()`, and the `%` operator for similarity based matching [11].

PostgreSQL also supports GIN indexes, which are designed for cases where values contain multiple component values, such as arrays or text search data [12]. The `pg_trgm` extension can use GIN indexes to make trigram based searches faster, since the database can use the index instead of comparing the search term with every stored text value row by row [11].

3.4 Go and Backend Libraries

Go is chosen for the backend because the project needs a small HTTP service that is simple to run, test and deploy. Go provides built-in standard library support for HTTP servers and clients through the `net/http` package [13, 14]. This makes it possible to build the API without depending on a large web framework.

The backend will use Go modules to manage dependencies and version information for the project [15]. The main external library used by the backend is `pgx`, which is a PostgreSQL driver and toolkit for Go. In this project it will be used through Go's `database/sql` interface to connect the API to PostgreSQL [16]. These choices support a backend that is focused on HTTP request handling, JSON responses and database access.

3.5 Containerization with Docker

A container is an isolated process with all of the files and dependencies it needs to run. They are also portable which means that they can run anywhere [17]. Docker is used to build and run containers from images, and an image contains the application and the runtime environment needed to execute it [18]. Docker Compose helps with running multiple Docker containers by providing a way to define, configure and run all needed containers [19].
Docker is chosen because the project will need a reproducible local environment for the backend, database and frontend. Instead of requiring each developer to install and configure every service manually, the services can be started in containers with the same configuration each time. Docker Compose is used because it can define and run multi-container applications, which fits a system that needs a Go API, a PostgreSQL database and a frontend console to run together [19, 20]. This makes the development setup easier to share and test across different machines.

4

Implementation

4.1 System Overview

The implemented system consists of a Go based backend API, a PostgreSQL database, a frontend and a containerized setup using Docker Compose.

4.2 Database Schema

A PostgreSQL database was implemented to store the educational content managed by the API. The schema was firstly designed around the main type of resources already used by Digimar, such as videos, transcripts, powerpoints, documents, chatbots and exercises. During development, standalone exercises were replaced with chapters and sections as the primary structure for oranizing the learning content, as this better aligned with the requirements of the group developing the webiste.
The final schema is centered around chapters and sections, content with relationships to videos, transcripts and powerpoints, documents and chatbots. In addition with other support tables such as images, tags, negative key words and API keys.

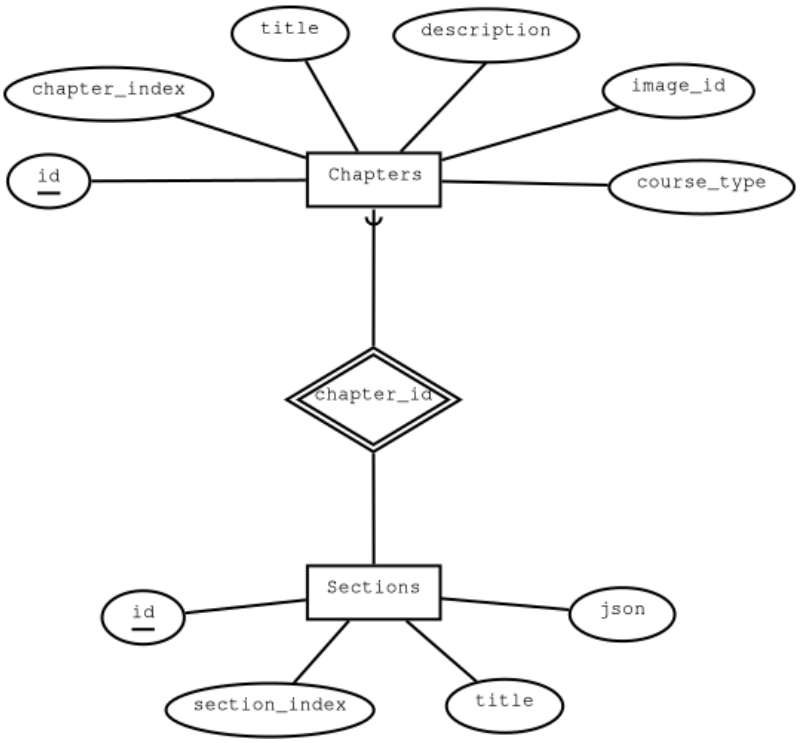

Figure 4.1: ER diagram of the content and chapter schema.

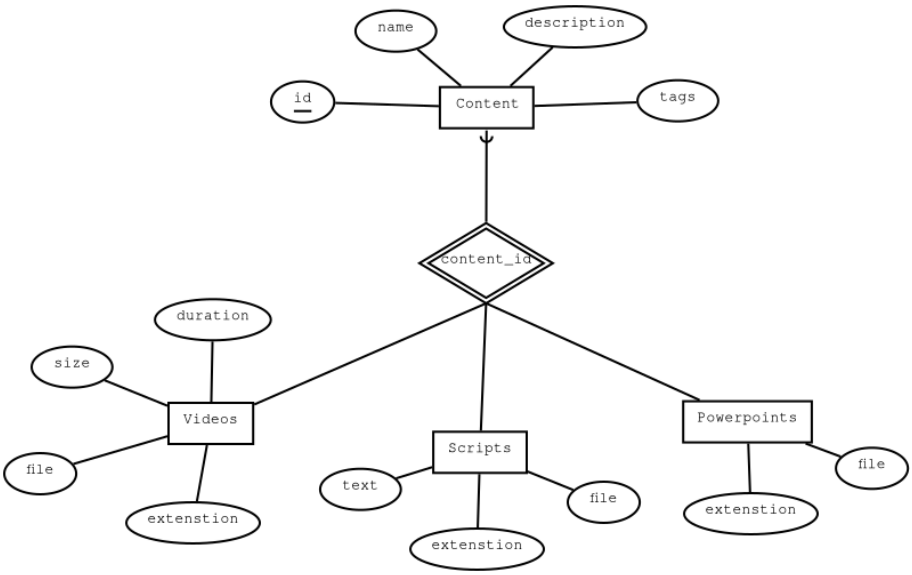

Figure 4.2: ER diagram of the content schema.

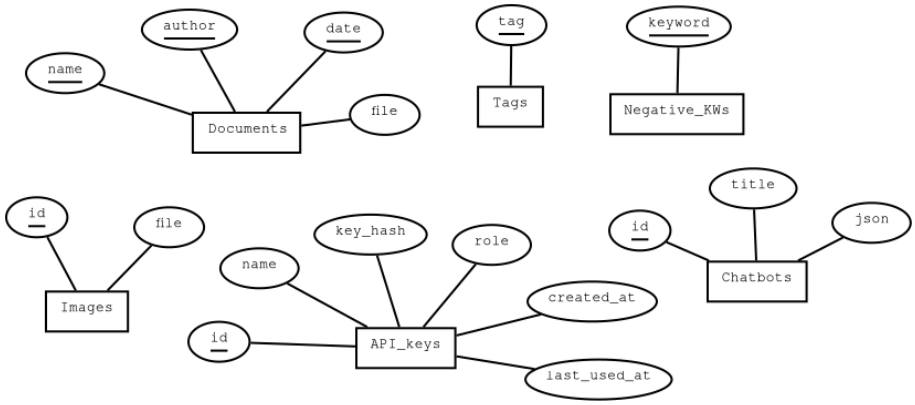

Figure 4.3: ER diagram of the chatbots and support tables.

4.2.1 Core Tables

The most central part of the schema is the chapters and sections, and the content, videos, transcript and powerpoints.

A chapter represents a larger unit of learning material where a chapter can have several sections. The chapter table stores a UUID, index, title, description, an optional reference to an image and the course type. A section represents a smaller part of a chapter and contains the learning material with exercises. Sections stores a UUID, a reference id to a chapter, index, title and a JSONB. The image table stores a UUID and the BYTEA of the image, and it is used to store images for the sections.

Content table was designed to support the original content structure used by Digimar. In the database schema, a content item represent a learning unit. Each content item can have a video, transcript and powerpoint. Content stores a UUID, name, description and tags, for each content type they store extenstion and file, videos has two extra fields with size and duration, and transcripts stores the raw text.

4.2.2 JSON based content

Sections and chatbots both store their main content as a JSONB in the database. This was done because the structure of these resources can vary depending on the type of content being stored. For example, different sections may contain different combinations of text, exercises, images, or other type of material. Chatbots already had a clear structured table, but instead of having it stored as those fields, JSON was used to give developers more flexibility and to support other structures in the future.

4.2.3 Relationships and Constraints

As mentioned earlier, each section belongs to a chapter. This means that a section must reference an existing chapter before it can be created. This relationship is enforced using a foreign key from the sections table to the chapters table. Both

chapter and section indexes are unique to prevent duplicate positions. Triggers are also used to maintain valid ordering of chapters and sections. They prevent a new chapter or section from being inserted with an index higher than the current maximum index plus one. If a chapter or section is inserted at an existing index, the trigger shifts the existing indexes to make room for the new entry. The content relationship makes it possible to group resources that belongs together. So that one content item can represent a lecture while the related video, transcript and powerpoint store the different format of that lecture. This avoids duplicating shared metadata and makes it possible to retrieve related resources through a common content reference. The video, script and powerpoint table references the content table id using foreign key.

4.2.4 Schema initialization

The database schema is initialized automatically when the backend starts. The SQL schema is embedded into the application. During startup the embedded schema gets read, splits into seperate SQL statements and executes each statement againt the PostgreSQL database. The schema file is divided using a custom statement seperator. This was done because the schema contains more complex SQL blocks, such as triggers and functions, which would make splitting on semicolon unreliable. The initialization process creates required PostgreSQL extensions, tables, constraints and triggers if they do not already exist. It also includes update statements and ALTER TABLE statements to make the schema compatible with earlier versions of the database structure.

4.3 Backend API

The backend was implemented in Go as an HTTP REST API and acts as the main interface between clients and the database. It exposes HTTP endpoints for managing the diffrent resources and is responsible for request handling, input validation, access control, database communication, and response formatting.

4.3.1 Server Structure

The entry point for the backend is located at `cmd/server/main.go`. When the server starts it loads the server configuration and the server requires an `ADMIN_API_KEY` to be set. After the configuration has been loaded, the server opens a connection to the PostgreSQL database. The database schema is then intialized and then the router is created. Finally the HTTP server is created using Go's `net_http` package. The server is configured with read, write and idle timeouts to avoid leaving connections open indefinitely.

4.3.2 Routing

Routing is implemented in `internal/httpapi/router.go`. The router is responsible for managing all the API paths and connecting each path and HTTP method

to the corresponding handler function. The API follows a REST style structure where resources are exposed through endpoint paths and standard HTTP methods are used to describe the operation being performed. In general, GET requests are used to retrieve data, POST requests are used to create new resources, PATCH requests are used to update existing resources, and DELETE requests are used to remove resources. The routes are organized under `/api` path. Resource collections suchs as `/api/chapters`, `/api/sections`, `/api/content`, `/api/chatbots` and `/api/images` are used for listing or creating resources. Item specific routes like `/api/chapters/{id}` and `/api/sections/{id}` are used as operations on a single resource.

Not all route groups implement the full set of CRUD operations. The available methods depend on how each resource is expected to be used. For example, the content routes only support GET, POST and DELETE. Update operations were not implemented for content, since this part of the schema was intended to support existing Digimar material that was not expected to be modified through the API.

4.3.3 Health and Readiness Endpoints

The backend also exposes two public status endpoints: `/healthz` and `/readyz`. These routes do not require an API key, because they are intended to be used for operational checks and simple troubleshooting. The `/healthz` endpoint checks that the HTTP server process is running. It returns a status value and the time when the server started.

The `/readyz` endpoint checks whether the backend is ready to handle requests that depend on the database. It does this by pinging the PostgreSQL database with a short timeout. If the database can be reached, the endpoint returns `ready`. If the database is unavailable, it returns a service unavailable response. This makes it possible to distinguish between a server that is running and a server that is fully ready to serve API requests.

4.3.4 Handlers

The handlers are responsible for processing the incoming HTTP request after they have been matched by the router. Each handler extracts the required input from the request, such as path parameters, query parameters, or JSON data from the request body. The input is then validated before the handler performs any database operation.

For create and update operations, the handlers decode the request body into structs and check that required fields are present. If the input is invalid, the API returns an error response with an appropriate HTTP status code. For retrieval and delete operations, the handlers usually read identifiers from the request path or query parameters and use them to locate the correct database entry.

After validation, the handlers call the database-related functions needed to complete the request. The result is then written back to the client as a JSON response. Successful requests return status codes such as 200 OK or 201 Created, while errors are returned using status codes such as 400 Bad Request, 404 Not Found, or 500 Internal Server Error.

Table 4.1: Overview of the API paths.

Route	Methods
/healthz	GET
/readyz	GET
/api	GET, POST, DELETE
/api/chapters	GET, POST, DELETE
/api/chapters/{id}	GET, PATCH, DELETE
/api/sections	GET, POST, DELETE
/api/sections/{id}	GET, PATCH, DELETE
/api/content	GET, POST, DELETE
/api/content/media	GET, POST
/api/content/script	GET, POST
/api/content/pptx	GET, POST
/api/content/{id}	GET
/api/content/{id}/meta	GET
/api/content/{id}/media	GET, POST, DELETE
/api/content/{id}/script	GET, POST, DELETE
/api/content/{id}/pptx	GET, POST, DELETE
/api/documents	GET, POST, DELETE
/api/documents/{author}/{name}/{date}	GET, DELETE
/api/images	GET, POST, DELETE
/api/images/{id}	GET, DELETE
/api/chatbots	GET, POST, DELETE
/api/chatbots/{id}	GET, PATCH, DELETE
/api/search	GET, POST
/api/admin	GET
/api/admin/keys	GET, POST
/api/admin/keys/{id}	DELETE
/api/admin/clear	DELETE

4.4 Access Control

Access control is implemented as middleware in `internal/httpapi/auth.go`. The middleware is applied to the whole router, but `/healthz` and `/readyz` are public so that the server and database status can be checked without an API key. All other routes require an API key. The key can either be sent in the `X-API-Key` header or as a bearer token in the `Authorization` header.

The server has one bootstrap admin key, which is read from the `ADMIN_API_KEY` environment variable when the server starts. This key is compared using Go's constant time comparison function. The admin key can be used to access the admin routes and to create other API keys. Generated API keys are prefixed with `d-smart_`, returned to the client only once, hashed with SHA-256 and then stored in the `API_Keys` table. The raw key is therefore not stored in the database.

The issued keys have three roles: `read`, `read_write` and `admin`. A read key can only use non-admin GET requests. A read/write key can use non-admin read and write routes. Admin routes under `/api/admin` require either the bootstrap admin key or an issued admin key. When an issued key is used, the backend also updates the `last_used_at` column in the database.

4.5 Content and File Handling

Content and files are stored directly in PostgreSQL. A content row stores the shared metadata, such as id, name, description and tag. The actual asset files are stored in separate tables that use the same UUID as the content row. Videos are stored in `Videos`, scripts in `Scripts`, and powerpoints in `Powerpoints`. The foreign keys use `ON DELETE CASCADE`, so when a content item is deleted the related assets are also removed.

There are two ways to upload content assets. A client can first create a content item through `/api/content` and then upload an asset to `/api/content/{id}/media`, `/api/content/{id}/script` or `/api/content/{id}/pptx`. The other way is to use the auto upload routes `/api/content/media`, `/api/content/script` and `/api/content/pptx`. These routes accept a file name and a base64 encoded file. The backend takes the content name from the file name and creates the content row if it does not already exist. Script auto upload only accepts `.txt` and `.md` files, while powerpoint auto upload only accepts `.pptx` files.

When media is retrieved, the backend serves it inline using `http.ServeContent`. It sets `Accept-Ranges: bytes`, which makes browser seeking possible for media files. The content type is taken from the file extension when possible, otherwise it is detected from the file bytes. Scripts and powerpoints are returned as attachments, so the browser downloads them instead of trying to display them inline.

Images and documents are handled as their own resources. Images are uploaded as base64, stored as BYTEA values and given a generated UUID. The image list returns id, size and detected content type, and single images are served inline. Documents store name, author, date and file bytes. The document list route returns metadata and route information, while the single document GET route also returns the stored file as a base64 encoded value in `file_base64`.

4.6 Search

The search functionality was implemented through the `/api/search` endpoint. It supports both GET and POST requests. For GET requests, the search term is provided as a query parameter using `?q=`, while POST requests accept the search term in a JSON body.

The search implementation queries several parts of the database at the same time, which is content, documents, chapters, sections and chatbots. It returns the results in a shared response format where each result contains a type, identifier, name, optional extra information and route links that can be used to retrieve the matched resource.

Table 4.2: Fields included in the search query.

Resource type	Table	Searched fields
Content	`content`, `scripts`	`name`, `description`, `tag`, `text`
Document	`documents`	`name`, `author`, `date`
Chapter	`chapters`	`title`, `description`, `course type`
Section	`sections`	`title`, `json`
Chatbot	`chatbots`	`title`, `scenarios`

The SQL query combines the result from these resource types using a `UNION ALL`. Each part of the query returns the same fields: type, id, name, extra and an internal score, so that results from different tables can be combined into one response.

Table 4.3: Common fields produced by the search query.

SQL field	Purpose
`type`	Identifies the resource type of the result
`id`	Identifier used to build a route to the matching resource
`name`	Main display name of the result
`extra`	Additional context about the result
`score`	Internal ranking value used for ordering results

In each part of the query the `WHERE` clause determines wheter a row should be included or not. It checks several searchable fields using both partial matching and trigram similarity matching. For example, content items are included if the search term matches the content name, description, tag, or transcript text. This is done

using the expression `ILIKE '%' || $1 || '%'` for substring matching and the trigram `%` operator for fuzzy matching.

The score field is used to rank the match rows. It is calculated with `GREATEST()`, which returns the largest value from several comparison expressions. This means that each searchable field contributes a possible score, and the best match becomes the final score for that row. A direct match in a title or name recieve a high fixed score while fuzzy matches recieves scores from functions such as `similarity()` and `word_similarity()`. This score is then used to prioritize results where the search term matches fields more strongly.

The extra field is used to provide context that depends on the resource type. For content, it describes which related media types are available such as videos, transcripts and powerpoints. For document, it contains the author, while chapter result use it for course type. Section results use the field to include the related chapter and chatbots result leaves it empty.

PostgreSQL's `pg_trgm` extenstion is used to support fuzzy search. The extenstion compares strings using trigrams. It provides the functions `similarity()` and `word_similarity()` and also the `%` operator. GIN trigram indexes are also created on the searchable fields so that PostgreSQL can locate possible matches more efficiently instead of scanning all text values row by row.

4.7 Frontend

The frontend in `db-server/frontend` is a static API console. It is built with plain HTML, CSS and JavaScript. The JavaScript code handles view switching, form submissions, API requests, response rendering and local state in the browser. It is served as a separate Docker Compose service using an nginx container and is meant for manual operation of the backend API.

The frontend reads the API base URL from `config.js`. In the Docker setup this file is written by the frontend container entrypoint from the `API_BASE_URL` environment variable. The console also stores the selected base URL and API key in the browser local storage and sends the key with requests using the `X-API-Key` header. The top bar includes buttons for `/healthz`, `/readyz`, `/api` and `/api/admin`, which makes it possible to quickly check whether the backend is running, whether the database is reachable, and whether the provided admin key works.

The API console has separate views for overview, content, chapters and sections, documents, images, chatbots, API keys and search. The overview view calls `/api`, `/api/content`, `/api/chapters`, `/api/sections`, `/api/documents`, `/api/images`, `/api/chatbots` and `/api/admin/keys` to show live resource counts. The other views use the same routes as the backend API and provide forms for creating, updating, inspecting and deleting the supported resources. File uploads are converted to base64 in the browser before they are sent to the backend.

Images of the frontend interface can be found in appendix B.

4.8 Local Setup and Deployment

The local setup uses Dockerfiles for the backend API and the frontend. The backend Dockerfile builds the GO API server and the frontend Dockerfile builds and serve the frontend. Docker Compose is then used to run the backend, frontend and the PostgreSQL database togheter as seperate services. Configuration values are provided in the `.env` file. The backend reads the environment variables HTTP address, database connection string, admin API key, allowed CORS origins and PostgreSQL password. The frontend reads api base url, which defines where API requests should be sent.

Table 4.4: Environment variables used by the local setup.

Component	Environment variable	Required
Backend	`HTTP_ADDR`	No
Backend	`DATABASE_URL`	No
Backend	`ADMIN_API_KEY`	Yes
Backend	`CORS_ALLOWED_ORIGINS`	No
Backend and Database	`POSTGRES_PASSWORD`	No
Frontend	`API_BASE_URL`	No

4.9 Testing

Testing is mainly implemented through Go tests in the backend. The backend contains 13 `_test.go` files and 53 top level test functions. The tests cover configuration loading, schema initialization, PostgreSQL connection setup, authentication, router behaviour, helper functions, content handlers, image handlers, resource handlers and integration style flows.

The authentication tests check the role rules, key extraction, invalid keys, admin access and API key creation. The router tests check that the routes accept the intended HTTP methods and reject unsupported methods. Handler tests use test requests and test databases to verify that resources can be created, listed, retrieved, updated and deleted. The schema tests also check that the embedded PostgreSQL schema can be split and executed correctly.

A full `go test ./...` run depends on Docker Desktop, because several tests start PostgreSQL containers through the test support package. This makes the tests closer to the real runtime environment, since database specific behaviour such as constraints, triggers and PostgreSQL extensions are tested against PostgreSQL instead of being replaced with an in-memory database.

5
Results

5.1 Final system overview

The final system provides a centralized way of managing multiple types of Digimar learning material. The Go HTTP REST API acts as the main interface to the system, while the PostgreSQL database stores the resources. A frontend is also provided to support manual interaction with the API.

It supports several content types, including chapters, sections, videos, transcripts, PowerPoint presentations, documents, images, and chatbots. Chapters and sections are used as the main structure for organizing learning content, while the content related resources make it possible to group videos, transcripts, and presentation slides that belong to the same learning material.

5.1.1 Project Folder Structure

```
db-server/
|-- .dockerignore
|-- .env
|-- Dockerfile
|-- README.md
|-- docker-compose.yml
|-- cmd/
|   |-- server/
|       |-- main.go
|-- db/
|   |-- schema.go
|   |-- schema.postgres.sql
|-- docs/
|   |-- api-design.md
|   |-- data-model.md
|-- frontend/
|   |-- Dockerfile
|   |-- app.js
|   |-- config.js
|   |-- docker-entrypoint.sh
|   |-- index.html
|   |-- styles.css
|-- insert_script/
|   |-- README.md
|   |-- import_digimar.py
|-- internal/
    |-- config/
    |   |-- config.go
    |-- httpapi/
    |   |-- auth.go
    |   |-- chapters.go
    |   |-- chatbots.go
    |   |-- content.go
    |   |-- discovery.go
    |   |-- documents.go
    |   |-- images.go
    |   |-- resource_helpers.go
    |   |-- router.go
    |   |-- search.go
    |   |-- sections.go
    |-- store/
    |   |-- postgres.go
    |-- testsupport/
        |-- postgres.go
```

5.2 API functionality

The API exposes the resource types through REST oriented HTTP endpoints. It uses standard methods such as `GET`, `POST`, `PATCH` and `DELETE` for managing the content. The supported operations vary depending on the resource type. A complete overview of the API routes is provided in the Table 4.1.

Chapters, sections, and chatbots support creation, retrieval, updating, and deletion. These resources represent data that may need to be changed after creation. File based resources, such as videos, transcripts, PowerPoint presentations, documents and images, support operations for storing, retrieving, and deleting data.

5.3 Search functionality

The API has a search endpoint that allows several resource types to be searched. The search functionality covers content, documents, chapters, sections and chatbots. It covers both meta data and text based fields, such as names, descriptions, tags, transcript text, documents information, section content and chatbot scenario data. The search also support fuzzy matching, which allows similar or partial search terms to return relevant results. Each search results also returns a common format which is the resource type, name, identifier, extra context and route links.

The search endpoint is supported by GIN trigram indexes on all searchable text fields. Table 5.1 shows the storage cost of these indexes.

Table 5.1: Search index sizes

Search index	Table	Size
`scripts_text_trgm_idx`	scripts	592 kB
`chatbots_scenarios_text_trgm_idx`	chatbots	456 kB
`sections_json_text_trgm_idx`	sections	184 kB
`content_description_trgm_idx`	content	112 kB
`chatbots_title_trgm_idx`	chatbots	64 kB
`chapters_description_trgm_idx`	chapters	48 kB
`content_name_trgm_idx`	content	40 kB
`content_tag_trgm_idx`	content	32 kB
`chapters_coursetype_trgm_idx`	chapters	24 kB
`sections_title_trgm_idx`	sections	24 kB
`chapters_title_trgm_idx`	chapters	24 kB
`sections_description_trgm_idx`	sections	24 kB
`documents_author_trgm_idx`	documents	16 kB
`documents_name_trgm_idx`	documents	16 kB
`documents_date_trgm_idx`	documents	16 kB
Total	**All search indexes**	**1672 kB**

The total storage cost of the search indexes is 1672 kB. The largest indexes are on the longest text based fields, such as script text, chatbot scenario data and section JSON content. Currently there are no documents in the database and therefore the document indexes are small.

To evaluate the behavior of the search endpoint, several search terms were tested using the real search query. The tests compare correctly spelled queries, partial queries and misspelled queries. The tests in Table 5.2 were conducted by running the full API search query with `EXPLAIN ANALYZE` and recording the number of returned rows and the execution time.

Table 5.2: Fuzzy and partial search behavior

Query	Query type	Results	Execution time
`introduction`	Correct spelling	6	72.171 ms
`introdution`	Missing character typo	3	45.891 ms
`introducton`	Missing character typo	3	63.505 ms
`digimar`	Correct spelling	32	53.174 ms
`digima`	Partial term	32	54.340 ms
`digmar`	Missing character typo	0	42.505 ms
`scenario`	Correct spelling	63	80.614 ms
`scena`	Partial term	63	58.001 ms
`scenrio`	Missing character typo	0	70.273 ms
`maritime communication`	Correct phrase	42	64.286 ms
`maritime comunication`	Typo phrase	4	54.939 ms
`maritime communicaton`	Typo phrase	4	70.574 ms
`Vessel Traffic Services`	Correct phrase	4	71.880 ms
`Vessel Traffic Servces`	Typo phrase	0	70.849 ms
`Vessel Trafik Services`	Typo phrase	0	48.066 ms
`standard phraseology`	Correct phrase	4	72.299 ms
`standard phraseolgy`	Typo phrase	3	45.208 ms
`standrd phraseology`	Typo phrase	3	45.264 ms
`anchoring position coordinates`	Correct phrase	1	42.970 ms
`anchoring position cordinates`	Typo phrase	1	47.463 ms
`anchoring positon coordinates`	Typo phrase	1	45.217 ms

The results show that partial search works well for several terms. For example, `digima` returned the same number of results as `digimar`, and `scena` returned the same number of results as `scenario`. This shows that the search endpoint can return relevant results even when the user only enters part of a word.

The results also show that fuzzy matching can handle some spelling mistakes. The misspelled queries `introdution`, `introducton`, `standard phraseolgy`, `standrd phraseology`, `anchoring position cordinates` and `anchoring positon coordinates` all returned results. However, fuzzy matching did not work for every typo. Queries such as `digmar`, `scenrio`, `Vessel Traffic Servces` and `Vessel Trafik Services` returned no results.

It is also important to distinguish between search behavior and index performance. The results in Table 5.2 were produced using the full API search query. In the inspected query plans, PostgreSQL mainly used sequential scans for this combined query instead of the GIN trigram indexes. Because the database is small, PostgreSQL opted for scanning all rows instead of using the trigram index. Therefore, Table 5.2 mainly shows the behavior of the search functionality, while the next table isolates individual indexed fields to measure the trigram index performance directly.

For the index performance tests, individual indexed fields were queried directly. Sequential scans were disabled to force PostgreSQL to use the trigram index, and index scans were disabled in a separate run to measure the equivalent sequential scan.

Table 5.3: Search index performance on script text queries

Query	Condition	Plan	Results	Time	Effect
`Vessel Traffic Services`	`ILIKE`	Trigram index	2	0.755 ms	7.39x
`Vessel Traffic Services`	`ILIKE`	Sequential scan	2	5.582 ms	–
`Vessel Traffic Services`	`%`	Trigram index	0	39.266 ms	0.58x
`Vessel Traffic Services`	`%`	Sequential scan	0	22.904 ms	–
`maritime communication`	`ILIKE`	Trigram index	29	3.689 ms	0.62x
`maritime communication`	`ILIKE`	Sequential scan	29	2.279 ms	–
`maritime communication`	`%`	Trigram index	0	33.343 ms	1.37x
`maritime communication`	`%`	Sequential scan	0	45.623 ms	–
`welcome to digimar`	`ILIKE`	Trigram index	28	1.683 ms	2.98x
`welcome to digimar`	`ILIKE`	Sequential scan	28	5.016 ms	–
`welcome to digimar`	`%`	Trigram index	0	45.881 ms	0.98x
`welcome to digimar`	`%`	Sequential scan	0	45.064 ms	–

Table 5.3 compares trigram index scans with sequential scans on the `scripts.text` field. The results show that the trigram index improved some `ILIKE` queries, especially the more selective `Vessel Traffic Services` query, which was 7.4 times faster with the index. The `welcome to digimar` query was also faster with the index for `ILIKE`, while `maritime communication` was slower. For the tested fuzzy `%` conditions, the trigram index was slower than the sequential scan in all cases.

Table 5.4: Search index performance on chatbot title queries

Query	Condition	Plan	Results	Time	Effect
`VTS`	`ILIKE`	Trigram index	8	0.379 ms	0.75x
`VTS`	`ILIKE`	Sequential scan	8	0.284 ms	–
`VTS`	`%`	Trigram index	0	0.703 ms	2.12x
`VTS`	`%`	Sequential scan	0	1.492 ms	–
`Vessel`	`ILIKE`	Trigram index	13	0.159 ms	3.89x
`Vessel`	`ILIKE`	Sequential scan	13	0.619 ms	–
`Vessel`	`%`	Trigram index	0	0.396 ms	4.86x
`Vessel`	`%`	Sequential scan	0	1.925 ms	–
`Entering`	`ILIKE`	Trigram index	5	0.562 ms	0.97x
`Entering`	`ILIKE`	Sequential scan	5	0.547 ms	–
`Entering`	`%`	Trigram index	0	0.981 ms	1.99x
`Entering`	`%`	Sequential scan	0	1.949 ms	–
`Q1`	`%`	Trigram index	4	3.323 ms	0.59x
`Q1`	`%`	Sequential scan	4	1.956 ms	–
`Q2`	`%`	Trigram index	1	1.521 ms	2.16x
`Q2`	`%`	Sequential scan	1	3.285 ms	–

The original chatbot title used for the full title typo tests was:

`A3 Vessel enters VTS area and must drop anchor to wait for pilot (Chalmers)`

The full title typo queries in Table 5.4 are:

- `Q1`: `A3 Vessl enters VTS area and must drop anchor to wait for pilot (Chalmers)`
- `Q2`: `A3 Vessl entr VTS are and must drop ancor to wait for pilot (Chalmers)`

Table 5.4 shows the behavior of the trigram index on the shorter `chatbots.title` field. The `ILIKE` condition worked well for substring matching. For example, `Vessel` returned 13 results and was faster with the trigram index.

The fuzzy `%` condition behaved differently. Single word fuzzy queries such as `VTS`, `Vessel` and `Entering` returned no final results, even though the trigram index produced candidate rows. This is because `%` compares the query with the full title value, rather than simply checking whether the word appears inside the title.

The full title typo queries show more clearly when fuzzy matching is useful. `Q1` returned 4 results, while the more misspelled `Q2` returned 1 result. This shows that the fuzzy `%` operator works better when the query is similar to the whole indexed field, such as a misspelled title. However, the performance benefit of the index was mixed.

5.4 Access Control

Access control is handled with API keys. Health and readiness routes are public, but the rest of the API requires a key. There are different key types for different levels of access. A `read` key can be used to retrieve data with GET requests, a `read_write` key can be used to both retrieve and change normal resources, and an `admin` key is required for administrative routes such as creating or deleting API keys.

This means that frontend clients must use the correct key depending on what they need to do. A frontend that only displays content can use a read key, while a frontend or admin tool that uploads, updates or deletes content needs a read/write or admin key.

5.5 Using the system

The system can be used directly through HTTP requests or through the frontend console in `db-server/frontend`. A typical frontend workflow starts by calling collection routes such as `/api/chapters`, `/api/sections`, `/api/content`, `/api/documents`, `/api/images` or `/api/chatbots`. The returned items and route links can then be used to request specific resources or files. The request must include an API key with enough access for the operation, for example a read key for retrieving data and a read/write key for creating or changing data.

For local use, the Docker Compose setup starts PostgreSQL, the Go backend and the frontend console. The backend runs on port 8080 and the frontend console runs on port 8081. The console can be used to verify the server, create API keys, upload content, manage chapters and sections, inspect stored data and test search without writing separate client code.

5.6 Current use

The backend is already being used by two frontend teams in the project. Both teams can use the same API as a shared source of data instead of creating separate temporary data structures for each frontend. This also helped verify that the API is usable outside the backend itself, since the routes, authentication and returned JSON formats are already being integrated into real frontend work.

6
Conclusion

6.1 Project Summary

This thesis set out to design and implement a searchable digital database with API access for Digimar. The purpose was to provide a centralized backend service for storing, retrieving, searching and managing different types of educational resources.

The result is a working Go based HTTP REST API with a PostgreSQL database, a frontend console and a local Docker Compose setup. The API supports chapters, sections, videos, transcripts, presentations, documents, images and chatbots. These resources can be accessed and managed through REST oriented HTTP endpoints.

The system also includes search functionality across several resource types, role based API keys, health and readiness endpoints, technical documentation and a frontend tool for manual operation. The backend is also being used by two frontend teams, which shows that the API can be integrated into other parts of the Digimar project.

6.2 Goal Completion

The first goal was to store and manage several content types. This was fully completed for the planned resource types. Videos, presentation slides, transcripts, chatbots, documents, chapters, sections and images can all be represented in the database and accessed through the API.

The second goal was to expose the resources through a REST oriented HTTP API. This was fully completed. The API includes collection routes and item specific routes, and uses standard HTTP methods such as `GET`, `POST`, `PATCH` and `DELETE` depending on what each resource supports.

The third goal was to support search across metadata and text fields. This was fully completed. The search endpoint covers content metadata, transcript text, document metadata, chapter fields, section text fields and chatbot scenario data. It also supports fuzzy matching using PostgreSQL trigram matching. Future work could still improve how search results are ranked or optimized based on user preferences.

The fourth goal was to protect read, write and administrative API access with role

based API keys. This was fully completed through the three roles `read`, `read_write` and `admin`. These roles separate normal read access, write access and administrative access.

The fifth goal was to support local reproducible development with Docker Compose. This was fully completed. The PostgreSQL database, Go backend and frontend console can be started together through the Docker Compose setup.

The sixth goal was to provide technical documentation and a frontend tool for manual operation. This was also completed. The project includes documentation for the API and data model, and the frontend console can be used to inspect the API, manage content, upload files, create API keys and test search.

The seventh goal was to provide a script or automation to convert PDF files to markdown format. This was not completed in the final implementation. The project includes import support for existing Digimar material, but it does not include a finished PDF to markdown conversion workflow. The decision to drop this goal was made as requirements from the the users of this project was changed, and this feature was no longer needed.

6.3 Critical Discussion

One part that works well is the central API structure. The system gives different frontend teams one shared backend instead of separate temporary data structures. The resource routes, common JSON responses and route links make the API easier to explore and integrate with. The schema also works well for the current Digimar material, since chapters and sections provide a learning structure while videos, transcripts and PowerPoint files can be grouped under content items.

The access control model is another strength. API keys with different roles provide a simple way to separate read access, write access and administrative access. This is suitable for the scope of the project and avoids exposing all operations to every client. The Docker Compose setup also makes the system easier to run locally, since PostgreSQL, the backend and the frontend console can be started together.

There are also parts that could be better. File handling is simple, but not ideal for large production use. Files are uploaded through the API and stored in the database, which works for the prototype but may not be the best long term solution for large videos or many large documents. The frontend console is helpful for testing and administration, but it is not designed as a polished user facing application.

Another weakness is database evolution. The project includes schema initialization and some update statements, but it does not use a formal migration system. This is acceptable for the current prototype, but it would become harder to manage if the database schema changes often or if new content types are added later.

The search results show that the current search implementation works well for partial text matching, but that the fuzzy matching behavior is more nuanced. The API combines `ILIKE`, fuzzy trigram matching with `%`, and ranking with `similarity` and `word_similarity`. The tests show that these do not behave equally well for all field types.

For long text fields such as `scripts.text`, the `ILIKE` condition is useful because it can find phrases inside transcripts. However, the fuzzy `%` operator is less suitable for these long fields because it compares the query with the whole text value. A short query phrase can appear inside a transcript without the entire transcript being similar enough to pass the fuzzy threshold. This explains why fuzzy only searches on script text returned no results, even when related phrases existed in the text.

For shorter fields such as `chatbots.title`, the fuzzy `%` operator behaves more as expected when the query is similar to the whole field value. The full title typo tests showed that misspelled title queries can still return relevant results. However, single word fuzzy queries such as `Vessel` or `Entering` did not return results with `%`, because the operator compares the single word against the full title rather than checking whether the word appears inside the title.

The index performance results show that GIN trigram indexes can speed up selective partial searches, but they do not automatically improve every query. When a search term matches most rows, the index cannot filter out much data and a sequential scan can be faster. This explains why PostgreSQL often chose sequential scans for the full API query on the currently small database. The indexes are still useful for scalability and for selective searches, but their benefit depends on the query condition and field length.

Overall, the current API search query is functional and supports useful partial and fuzzy search behavior, but the tests suggest one possible improvement: keep `ILIKE` and `%` for shorter metadata fields such as names and titles, but avoid using `%` for long transcript or scenario `JSON`.

6.4 Future Work

Future work should include a more formal database migration system, so that schema changes can be tracked and applied in a controlled way. File storage could also be improved by adding file size limits, object storage or streaming support for larger media files. Another area for future work is a script or automation for converting PDF files into markdown before they are imported into the system.

The search functionality could focus on tuning the existing ranking model, improving fuzzy matching and testing scalability. The search already ranks results and gives priority to name and title matches, but the score values could be evaluated further. The fuzzy matching could also be improved by tuning trigram similarity thresholds or adding spell correction support, since the tests showed that not all

typos are handled equally well. The frontend could also be developed further into a more complete administration tool or replaced by a production frontend. Finally, a production deployment would need work on backups, caching, monitoring, load balancing and other operational concerns that were outside the scope of this project.

Bibliography

[1] MDN Web Docs, "API," 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/API (accessed: 2026-05-07).

[2] HTTP Semantics, RFC 9110, Internet Engineering Task Force, 2022. Available: https://www.rfc-editor.org/rfc/rfc9110.html (accessed: 2026-05-07).

[3] MDN Web Docs, "Overview of HTTP," 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview (accessed: 2026-05-07).

[4] MDN Web Docs, "HTTP request methods," 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods (accessed: 2026-05-07).

[5] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, CA, USA, 2000. [Online]. Available: https://roy.gbiv.com/pubs/dissertation/top.htm (accessed: 2026-05-07).

[6] R. Sheposh, "Relational database," *Salem Press Encyclopedia of Science*, Research Starters, 2026. [Online]. Available: https://research.ebsco.com/c/lu54te/viewer/html/dcoscexydz (accessed: 2026-05-07).

[7] PostgreSQL Global Development Group, "Constraints," 2026. [Online]. Available: https://www.postgresql.org/docs/current/ddl-constraints.html (accessed: 2026-05-07).

[8] PostgreSQL Global Development Group, "About PostgreSQL," 2026. [Online]. Available: https://www.postgresql.org/about/ (accessed: 2026-05-07).

[9] PostgreSQL Global Development Group, "JSON Types," 2026. [Online]. Available: https://www.postgresql.org/docs/current/datatype-json.html (accessed: 2026-05-07).

[10] PostgreSQL Global Development Group, "Binary Data Types," 2026. [Online]. Available: https://www.postgresql.org/docs/current/datatype-binary.html (accessed: 2026-05-07).

[11] PostgreSQL Global Development Group, "F.35. pg_trgm – support for similarity of text using trigram matching," 2026. [Online]. Available: https://www.postgresql.org/docs/current/pgtrgm.html (accessed: 2026-05-07).

[12] PostgreSQL Global Development Group, "GIN Indexes," 2026. [Online]. Available: https://www.postgresql.org/docs/current/gin.html (accessed: 2026-05-07).

[13] The Go Authors, "The Go Programming Language," 2026. [Online]. Available: https://go.dev/ (accessed: 2026-05-12).

[14] The Go Authors, “Package http,” 2026. [Online]. Available: https://pkg.go.dev/net/http (accessed: 2026-05-12).
[15] The Go Authors, “Go Modules Reference,” 2026. [Online]. Available: https://go.dev/ref/mod (accessed: 2026-05-12).
[16] J. Carin, “pgx package,” 2026. [Online]. Available: https://pkg.go.dev/github.com/jackc/pgx/v5 (accessed: 2026-05-12).
[17] Docker, “What is a container?,” 2026. [Online]. Available: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/ (accessed: 2026-05-07).
[18] Docker, “What is Docker?,” 2026. [Online]. Available: https://docs.docker.com/get-started/docker-overview/ (accessed: 2026-05-07).
[19] Docker, “Defining and running multi-container applications with Docker Compose,” 2026. [Online]. Available: https://docs.docker.com/guides/docker-compose/ (accessed: 2026-05-07).
[20] Docker, “Multi-container applications,” 2026. [Online]. Available: https://docs.docker.com/get-started/docker-concepts/running-containers/multi-container-applications/ (accessed: 2026-05-12).

A

Timeline from the Planning Report

Table A.1: Original timeline from the planning report.

Week(s)	Planned activity
1–2	Planning phase: requirement clarification, planning report, and approvals
3–4	Data model, API design, and project setup
5–6	CRUD implementation, API keys, and validation
7–8	Search/filtering, content linking, and file streaming
9–10	Testing, bug fixing, stabilization, and initial evaluation
11–12	Final evaluation and consolidation of results
13–15 or remaining time	Thesis writing, revision, and final submission
If time remained	Optional goals: frontend work, database performance review, and PDF-to-Markdown script

B

Images from the frontend interface