Kseniia Koshkina

INTERNAL WIKI FOR DOCUMENTATION MANAGEMENT

Bachelor's thesis

Bachelor of Engineering

Information Technology

2025

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor or Engineering
Author	Kseniia Koshkina
Thesis title	Internal Wiki for documentation management
Commissioned by	Metatavu Oy
Year	2025
Pages	44 pages
Supervisor	Matti Juutilainen

ABSTRACT

The purpose of the thesis was to develop a documentation management platform aiming to enhance knowledge sharing within the commissioner company and serve as a central point of access to documentation. The objective was to organize documentation into wiki-style articles including a recommendation system, comprehensive search functionality, and management capabilities that would be different for users and administrators.

The backend of the application was presented by way of several AWS Lambda functions, interacting with DynamoDB, where structured article records were stored, and with a S3 bucket for managing media assets. The frontend was implemented with ReactJS and featuring an integrated rich text editor created with Lexical. This is a widely adopted stack that complies with the commissioner's technology preferences.

The system was designed with optimization and future development in mind. Therefore, the platform architecture supports continuous improvement, aiming to ensure that it can adapt to the commissioner's changing needs.

Keywords: web development, serverless architecture, ReactJS, AWS Lambda, DynamoDB

CONTENTS

1 INTRODUCTION

In today's rapidly changing world, the ability to constantly learn and adapt to changes is crucial to remain competitive in any position. The growth and expansion of a company highly depend on the ability of employees to adjust to new challenges, and without proper training tools progress can be hindered. In order to support this need for constant learning and development, companies require an internal digital platform serving as a comprehensive learning resource and centralized knowledge repository for their employees.

1.1 Overview and objectives

Inspired by the collaborative and accessible nature of Wikipedia, this thesis aims to address the aforementioned need by proposing an article-based documentation storage, designed to meet the specific needs of the commissioner, Metatavu Oy. The objective was to create a Wiki-like component for documentation management integrated into the Metatavu Oy's internal web application used by employees. This documentation management system would allow users to create, update and delete articles and facilitate navigation between topics and recommendation mechanisms.

As part of the company's application, the component aims to improve knowledge sharing and accessibility of company's documentation and support continuous learning through collaboration within the organization. For new team members, it serves as a comprehensive onboarding tool, providing learning guides to support their integration into the organization. For experienced employees, it serves as a reliable reference point, assisting in daily work. The Wiki documentation module is designed to become a reliable assistant for employees.

1.2 Technologies

Since this Wiki documentation component is developed as part of the Metatavu Oy platform, the technology stack must be predefined to ensure compatibility and maintainability across the company platform. By using a standardized stack, the

development process remains efficient, and the integration of new features is streamlined.

Both frontend and backend components are run in the AWS cloud. Backend is presented by way of several Lambda functions that act as a mediator between the article storage system and the web application. For data storage, the solution relies on DynamoDB for structured metadata and content, while an S3 bucket is used for storing article media assets. This serverless architecture enables a cost-efficient and scalable on-demand solution automating resource allocation. For automatic deployment and fluent development process, the study utilizes a Serverless framework.

The frontend part is implemented as a React application written on TypeScript and built with Vite. In order to implement rich-text editor, a Lexical library is used. This is a modern and widely adopted stack of technologies that complies with the commissioner’s technology preferences.

1.3 Thesis structure

The first part of the thesis, theoretical framework, is focused on the technologies used. The system design section describes the application architecture and the basis for implemented solutions, and the implementation section examines the development process. Finally, the thesis concludes with an evaluation of the documentation component and suggestions for future improvements.

2 THEORETICAL BACKGROUND

This section will focus on the theoretical background of the technologies used in the thesis. Various technologies will be explored, beginning with the AWS cloud infrastructure - specifically data storage and Lambdas that form the core of the backend solution - and continuing with modern web development frameworks.

2.1 AWS

In the modern world, setting up and managing on-premises servers for running company applications is often not cost-effective. Instead, businesses of all sizes from startups to large enterprises prefer to rely on cloud service providers for scalable, reliable, and cost-effective solutions. Among these providers, Amazon Web Services (AWS) stands as the largest and the most popular cloud platform, surpassing other major providers like Microsoft Azure. (Richter 2025.)

AWS manage the data center infrastructure and provides manageable and scalable resources. AWS clients can create a variety of cloud services with different levels of control over the underlying hardware (What is cloud computing? 2025). For example, AWS Lambda abstracts all hardware and infrastructure management while more customizable virtual machine instances with AWS Elastic Compute Cloud (EC2) allow users to define the resources, operating systems, and network configurations. (What is Amazon EC2? 2025.)

Also, AWS data centers are globally distributed, forming multiple Availability Zones within various AWS Regions – geographic areas. Clients can choose the Region where the resources would be deployed, and the latency would be the smallest for the closest corresponding geographic area. This structure enables companies to deploy applications closer to their end-users, reducing latency and improving performance. (AWS Regions 2025.)

The benefits of the AWS make it a flexible and suitable solution for a variety of organizations and tasks.

2.1.1 DynamoDB

DynamoDB is a NoSQL database solution offered by AWS designed for high performance and scalability. It is designed for high-traffic applications such as financial services, gaming and even streaming platforms. (What is Amazon DynamoDB? 2025.)

Compared to relational databases, DynamoDB does not have a predefined schema, and instead of tables, data is stored in a key-value or a document format. These data models, generally, consist of a primary key - a unique identifier of an instance and a value that is represented by the collection of attributes. These attributes can be strings, numbers, binaries and data structures such as lists and maps. The primary key consists of either a single partition key (simple primary key) or a combination of a partition key and a sort key (composite primary key). The sort key enables flexible sorting of items based on numerical values, string lengths or date-time string values formatted in ISO 8601 (DynamoDB does not have a dedicated datetime type). Combined composite key must be unique; however, items can share the same partition key. The sorting, then, applies when requesting the items within the same partition key. (Core components of Amazon DynamoDB 2025.)

Table 1 illustrates the DynamoDB structure. Each item includes a composite primary key (a partition key – **pk** and a sort key – **sk**) along with attributes. As shown in the table, sorting order is determined by a datetime string and applied within groups of items that share the same partition key.

Table 1. DynamoDB components and sorting

pk (String)	sk (String)	attributes
2	2021-02-24T...	some data
2	2023-02-24T...	some data
1	2022-02-24T...	some data
1	2024-02-24T...	some data
1	2025-02-24T...	some data

The key-value nature of DynamoDB provides limited access patterns, requiring developers to carefully design and define database access patterns or the questions that DynamoDB should answer.

DynamoDB supports two primary types of access operations: scan and queries. A scan operation reads an entire database and examines each item and related attributes, making it resource-heavy and slow for large datasets. In contrast, query operation retrieves items efficiently by primary keys or second indexes. Developers are generally recommended to utilize the query which supports them by data access patterns. (Best practices for querying and scanning… 2025.)

Given that DynamoDB is a NoSQL database and does not support traditional relation database operations such as joins, developers are encouraged to denormalize data as much as possible and duplicate full records instead of referencing to them (Best practices for modeling relational data… 2025). This approach reduces query complexity by preventing multiple-table relationships and ensures faster and more efficient access.

Furthermore, to increase query flexibility, developers can add second indexes of two types: local secondary indexes (LSIs) which share the same partition key as the primary key but use a different sort key, and global secondary indexes (GSIs) which allow entirely new partition and sort keys for broader querying options. While LSIs are useful for refining queries under the same partition, GSIs offer greater versatility and are more commonly used. However, creating a GSI results in DynamoDB maintaining a separate index table that includes a copy of selected attributes from a base table, a process known as projection. While the GSIs support new query patterns, they also increase storage and write costs. (DeBrie 2020, 38-42.)

By strategically using features such as denormalization and secondary indexes, developers can maximize DynamoDB’s performance while ensuring efficient data retrieval patterns.

2.1.2 Simple Storage Service

Simple Storage Service (Amazon S3) is an object storage service provided by AWS. Object storage is a way of storing data designed to process large amounts of unstructured information. Compared to traditional file storage systems which

organize data in a hierarchical structure of folders and files, object storage is relatively simple and is represented only by a collection of independent objects. (What is Amazon S3 2025.)

The object can be an image, a video or a text document. Each object in S3 consists of the data itself and metadata which includes a unique identifier for the object. Since objects are identified by a unique key, searching and accessing files is quick and efficient. This structure allows easy scalability and replication across multiple servers and data centers, enabling users to store virtually unlimited amounts of data.

2.1.3 Lambda

AWS Lamda is a serverless computing service offered by AWS. Generally, it is a code running in a container that replies to some trigger making it an event driven service. Such triggers could be HTTP requests, file uploads to S3 or preconfigured time-base events. (What is AWS Lambda? 2025.)

Lambda functions are executed inside containerized environments. Each container is allocated RAM, and CPU resources are automatically provisioned in proportion to the memory configuration. Users can define the amount of initially allocated memory, while AWS scales resources dynamically on-demand based on the load. AWS Lambda supports a variety of runtime environments including Node.js, Python and .NET. (Lambda runtimes 2025.)

After function execution, the container enters a frozen state for a period of time before being terminated. If new requests arrive after the container’s termination, AWS Lambda must reinitialize a fresh environment called a “cold start”. In order to improve performance and reduce the latency from initialization phase, running environments can be reused for subsequent requests. This practice is known as a “warm start”, and it is more common in production workloads. (Understanding the Lambda execution… 2025.)

AWS Lambda is a cost-effective solution where clients pay only for the execution time consumed by their functions. With proper system design, execution time can be optimized, and the number of requests minimized to further reduce costs.

By combining this cost-effective pricing model with Lambda's flexibility and integration with other AWS services, developers can build powerful, scalable applications that comply with a wide range of tasks. These include a backend application to handle HTTP requests, manager of scheduled jobs or applications integrated with services, for instance, Amazon S3 or DynamoDB.

2.2 Serverless Framework

Although AWS Cloud provides an abstracted infrastructure setup, the Serverless Framework further simplifies the process of creating and deploying serverless applications by automatically managing AWS services.

The serverless configurations file serves as the central point of specifications for serverless application, containing definitions that specify Lambda functions, API Gateway routes, permissions, and resources such as S3 or DynamoDB. These configurations are automatically translated into an AWS CloudFormation template which AWS then uses to manage the creation and configuration of infrastructure resources. (Serverless Framework Concepts 2024.)

AWS CloudFormation manages resources in a stack and process updates in a sequential order across deployments. Each deployment through the Serverless Framework triggers a CloudFormation stack update that applies only the necessary changes. (How CloudFormation works 2025.) This ensures that infrastructure changes are processed in a consistent and controlled order, contributing to the reliability and stability of application development with Serverless Framework.

2.3 REST API

Application Programming Interface (API) is a set of rules for communicating with a piece of software. REST API is a standardized web interface needed for data exchange over the Internet. REST, representation state transfer, is a client-server architectural pattern which defines rules for the representation and interaction of server resources. (What is a RESTful API? 2025.)

REST services identify server resources mainly by Uniform Resource Locator (URL). This URL is commonly referred to as an endpoint, defining the path to the resource. The method of the HTTP request (e.g., GET, POST, PUT, DELETE) is taken to detect the type of action over the resources. (What is a RESTful API? 2025.)

The REST fundamental constraints are presented below. Only APIs that fully implement these constraints can be properly characterized as RESTful (What is a RESTful API? 2025):

- Uniform interface – a standardized way to interact with resources.
- Statelessness – each request is independent and does not share the state or context of previous requests.
- Client-server – separation of these two concerns, enabling independent development.
- Layered system – the server can rely on other resources' APIs without clients needing to know the underlying details of any intermediaries.
- Cacheable – responses can be reused to improve performance.

2.4 Swagger and Open API generator tool

In order to ensure proper implementation and adoption of REST APIs, comprehensive documentation is essential. This is where API specification standards like Swagger play a crucial role.

Swagger, also known as the Open API Specification, is a standard for defining the structure and functionality of HTTP-based APIs using YAML or JSON. The specification stores available endpoints, supported operations for each endpoint

(HTTP methods), operation parameters and authorization requirements. (Swagger Documentation 2025.)

The Open API generator is a tool that helps to create client libraries by using an API specification as input. It automatically generates code libraries to interact and retrieve data from backend applications. (OpenAPI Generator Documentation 2025.)

The Open API Generator automates the creation of client libraries and reduces manual work associated with managing API definitions in frontend development. This methodology contributes to increased precision and transparency in backend development processes, while also increasing the maintainability of the overall software system.

2.5 JSON Web Token

JSON Web Token (JWT) is a user authentication token that contains user session and authentication data. JWT is a reliable and secure open-source standard for JSON transmission. JSON wraps JWT data in a compact way, so that it can be efficiently transmitted in HTTP headers, POST bodies, or even URLs, making it suitable for high-performance applications. (Krause M 2024.)

A JWT consists of a header, a payload and a signature. The header specifies which signing algorithm was used. The payload contains entity information such as user data including the name of the user and provided access roles to the resources. The JWT signature is a guarantee of the integrity of the stored data since any change to the token will invalidate the signature. Thus, by integrating the JWT token as an authorization method, the application remains secure and can fully trust the contents of the token. (Introduction to JSON Web Tokens 2025.)

JWT is a lightweight, secure, and flexible solution for handling authentication in modern web applications. By using signed or encrypted tokens, developers can

ensure both the integrity and confidentiality of transmitted data, while reducing the need for repeated database lookups or session storage.

2.6 React

React is JavaScript library for the development of user interfaces on different platforms. React embraces both functional and OOP principles and focuses on reusable components. Once a component is defined, it can be used in a hierarchy of independent components for creating an entire application. The components use states and properties for UI rendering, with React hooks managing state and side effects. (Krause M 2024.)

States in React are dynamic data that can change over time. Each React component manages its own states, updating HTML automatically whenever the state changes. This reactivity ensures the user interface always reflects up-to-date data. (Krause M 2024.)

In order to define what a component should render, React uses JSX syntax extension to describe the appearance of UI. It allows developers to write HTML-like code directly in JS files. JSX makes it easier to visualize UI structure and integrate dynamic behavior using JS expressions.

React uses a virtual DOM (Document Object Model) for real DOM imitation, in-memory representation. React updates the virtual DOM based on changes in the state of components and then calculates the difference between the previous and current virtual DOM. This approach allows React to only update changed elements in the real DOM automatically, which makes it possible for developers to make changes without directly manipulating the on-screen elements. (Difference Between Virtual DOM… 2025.)

2.7 Jotai

Jotai is a state management library characterized by its simplicity and ease of use. While it is less popular than Redux, it is significantly simpler and often more

suitable for small projects due to its minimalistic design. Jotai allows developers to create globally accessed states, referred to as atoms. These atoms are manipulated using the useAtom hook, an abstraction that extends React's useState, enabling read and write operations for atoms. (Kato, D. n.d.)

When the atom is assigned to a new value, only the components subscribed to that atom are re-rendered (Kato, D. n.d.). Since atoms are globally shared and accessed, developers can eliminate the number of unnecessary complex re-computation and re-fetching operations across components or globally manage themes and language settings for the application.

2.8 TypeScript

TypeScript (TS) is a programming language represented as a superset of JavaScript (JS). TS solves common JS problems such as type errors, lack of type safety and late error detection. Unlike JS, TS is a statically typed language that compiles into JS. It includes JS while offering additional features that make it more robust and predictable. (Krause M. 2024.)

The advantages of TS are especially evident when working with large applications as it helps to solve problems with variable types. For example, because TS defines data types, it integrates autocompletions more correctly, thus helping the developer with code writing. Its strict typing system provides reliable code and helps catch bugs early during development. This reduces debugging time and minimizes runtime issues.

TypeScript empowers developers to write more maintainable, scalable and error-resistant code. It provides a good balance of flexibility and safety, which makes it a beneficial solution for software development.

2.9 Lexical

Lexical is a text editor framework developed by Meta to offer a high degree of customization for developers. Developers can build simple plain-text editors as well as real-time collaborative rich-text editors. (Lexical Documentation 2025.)

Lexical, like most React-based rich text editors such as DraftJS or SlateJS, operates on top of the *contentEditable* HTML element and acts as an extended plugin or controlled version of the component. This wrapped element is referred to Editor and it controls all interactions and content updates. At the core of the system is the Editor State which maintains the internal state representation of the content or a tree of nodes similar to React Virtual DOM. The second primary component of Editor State is a Selection object which tracks user cursor and selected range. (Lexical Documentation 2025.)

Figure 1 illustrates Lexical Editor State showing a node tree which represents the document structure using a hierarchy of nodes. Each node corresponds to a block or inline element in the editor, such as headings, paragraphs, lists and links. At the bottom, there is a Selection object that indicates the position of the cursor defined by the node key and the offset (character index inside the node). The selection is defined by a range, starting from the anchor position and ending at the focus position.

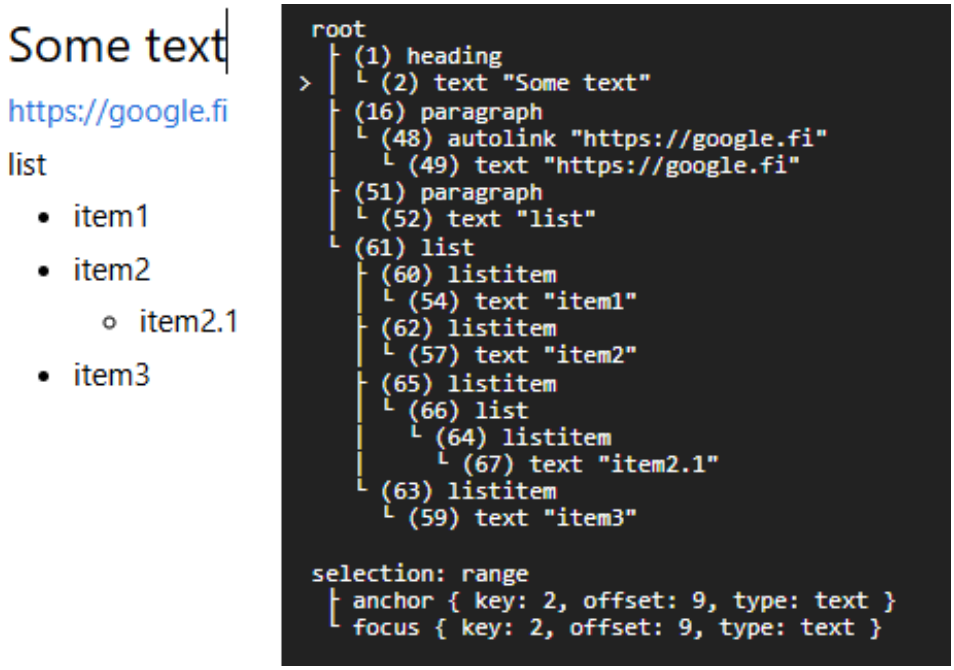

Figure 1. Lexical Editor State

When the user inputs or edits text, an *onChange* event is triggered. Based on this change, Lexical creates a new editor state that reflects the updated content, including the newly added text and changes in content type. Additionally, Lexical tracks cursor positioning (selection), ensuring the new state includes the exact position where the user is typing or where the selection is currently located. Afterward, Lexical processes the new editor state and compares it to the current state (executes similar to React diff) and updates or re-renders the visible content accordingly. This rendering occurs only for affected DOM elements, not the entire editor, improving the performance of the application. (Lexical Documentation 2025.)

In order to store formatted text, Lexical requires a transformer to serialize and deserialize the content. Lexical provides build-in support for different serialization options. The node tree content could be translated to JSON, HTML, or Markdown. (Lexical Documentation 2025.)

3 SYSTEM DESIGN

This section will describe the key solution requirements and application design decisions.

3.1 Requirements

Every large development project begins with a definition of its requirements, aims, expected functionalities and potential limitations. In web development, these specifications serve as the basis for making decisions about the storage, processing and display of data.

The main aims of this thesis are to develop a robust backend component capable of handling complex CRUD operations for articles (where an article refers to a documentation entry) and to design a well-structured frontend that facilitates easy article access.

The frontend should integrate an intuitive navigation and search system, enabling users to efficiently search for and manage content. Users should also be able to track their progress, marking articles as read, while the homepage card should display and recommend the last read, created or updated article. Additionally, a frontend application should support rich-text editing functionality while maintaining a user-friendly design.

Articles should be mainly accessed based on a defined path. These paths should be structured into a graph where related topics are linked together to enhance navigation and recommendations. For example, a playbook (common/playbook) article might reference relevant security policies which could further connect to technical guides on disk encryption (common/security/disk-encryption) or documentation on access control restrictions. In order to facilitate search functionality, articles are categorized with specific tags that help users quickly locate relevant content.

In addition to the basic functionality, the documentation management system must meet key non-functional requirements to ensure high performance, security, and scalability. Performance is a crucial aspect, requiring short load times and optimized database queries to support large datasets. Therefore, the article

search technology must be optimized to process an increasing number of articles efficiently without a decrease in performance.

Security is another critical factor, with strong authentication and authorization mechanisms to protect sensitive documentation and enforce access control. Only authorized users can access, edit or delete content, with update and deletion functionalities restricted exclusively to admin users.

Responsiveness is also an important aspect and is considered during design to ensure that websites are operational on various devices including desktop and mobile versions.

3.2 Key design decisions

The application is expected to manage between 100 and 1,000 articles, which influences decisions regarding the location of the key processing logic. Given this scale, most of the logic, including search functionality and content filtering, will be conducted on the frontend to ensure fast and responsive interactions for users.

The application will fetch only article metadata such as title, tags, creation date, path and author rather than full article content. This article metadata is expected to remain under 1 MB in size (even if 1000 of articles are requested) which ensures quick loading even in circumstances of slow network connection. By keeping all necessary metadata available on the client-side, searching and filtering can be performed without additional backend requests, thus simplifying backend server logic.

However, as the number of articles grows, relying only on front-end-based processing may become inefficient. In this case, more advanced solutions will be required, shifting search and data retrieval mechanisms to the backend. This would involve implementing a more scalable search system, pagination and optimized queries to maintain performance despite increasing data volumes.

The application features a rich-text editor that operates with Markdown formatting, finding a balance between advanced text editing capabilities and simplicity. This approach allows users to apply formatting such as headings, lists, bold or italic text, links and images. The stored Markdown content can be easily rendered into HTML and displayed, which ensures the compatibility of the solution with the main web application.

In order to enforce access control and restrict resources to authorized users, this study utilizes Keycloack solution which is already used within the commissioning organization.

3.3 System architecture overview

The high-level architecture system is illustrated in Figure 2. The frontend application for authentication and authorization uses the Keycloak service that responds with the JSON Web Token (JWT) upon successful login. The frontend (Metatavu-home) includes the JWT in requests to the backend through an API Gateway. Requests received by API gateway invoke corresponding registered lambdas to the endpoint.

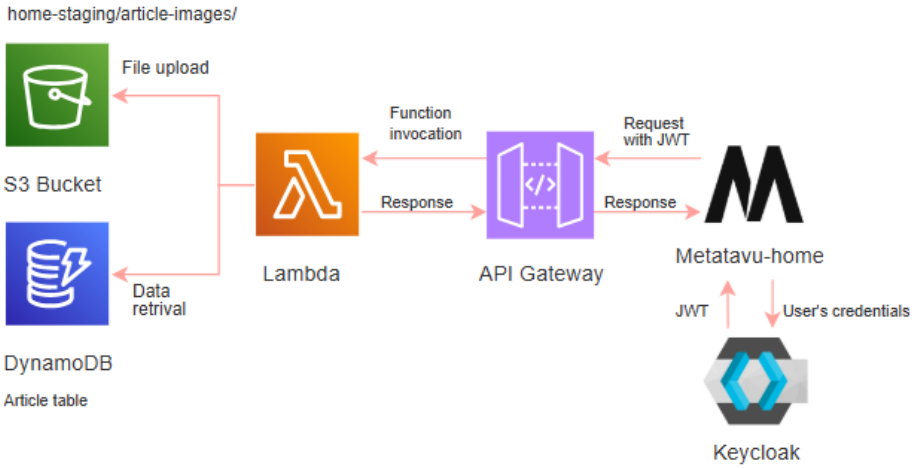

Figure 2. Overall system architecture

Article media assets are publicly available and stored in the folder in S3 bucket storage. Public read operations are permitted only for the */article-images* folder, while write access is restricted exclusively to Lambda functions.

3.4 DynamoDB table design

In order to meet the requirements and support defined data access patterns, primarily by path, the DynamoDB table should be structured accordingly. Table 2 illustrates the proposed structure. The *id* serves as the primary key and acts as the unique identifier for each article instance, while the *path* is configured as a secondary index representing the unique article path. Although using two unique identifiers might seem redundant, the id key is necessary to ensure that the path remains mutable. Moreover, since Global Secondary Indexes (GSIs) do not enforce uniqueness constraints, the path is used solely to enhance query performance.

Attributes such as *createdBy*, *lastUpdatedBy*, and *readBy* store users' names rather than id references, aligning with a denormalization approach. The *draft* attribute indicates the article review status: if it requires a review from admin, it is marked as draft.

Table 2. DynamoDB structure

Primary key	
id	string
GSI	
path	string
Attributes	
title	string
description	string
coverImage	string
content	string
createdBy	string
createdAt	string (Date format)
lastUpdatedBy	string
lastUpdatedAt	string (Date format)
readBy	list of string
lastReadAt	string (Date format)
tags	list of string
draft	boolean

A scan operation will be used to retrieve the complete list of articles. Although scan operations are generally slow and costly, the expected dataset is small, thus the performance is not currently a concern. Another important access pattern involves querying by article path. In order to avoid inefficient scans with filters the design incorporates a secondary index, enabling fast and efficient queries by path.

In the future, if the number of articles exceeds 10,000, the data model may need to be reconsidered, and the access patterns could be optimized further by relying more heavily on query operations. For example, a Global Secondary Index (GSI) could be created to store the publication year as a partition key and the month as a sort key. This would allow targeted queries by publication year and help more efficiently retrieve and display a limited number of articles.

In order to reduce weight and write consumption for the GSI, the KEYS_ONLY projection option was chosen. This configuration maps only the path and associated id into the GSI, omitting all other attributes.

4 IMPLEMENTATION

This section focuses on the implementation process, detailing the design and structure of both the backend and frontend components.

4.1 Backend Implementation

The backend API follows a RESTful structure as illustrated in Figure 3. The endpoints are organized around the /articles resource.

Figure 3. Backend API

The GET /articles endpoint allows optional filtering via *pathPrefix* and *draft* parameters, enabling more precise requests. In order to support the full range of CRUD operations, the API's design also includes POST on the /articles path for creating new articles and PUT, DELETE methods on the /articles/{id} path for updating and deleting accordingly. This forms the core of the RESTful structure.

Besides REST conventional CRUD endpoints, the API also provides GET /articles/path which takes the article path as a query parameter. Query parameters were chosen to avoid in path catch-all parameters (e.g. /articles/{path+}), which could lead to security issues. The POST /articles/{id}/read endpoint is registered to efficiently update not the entire table entry but specific attributes such as *readBy* and *lastReadAt* to log user interactions. Additionally, there is a dedicated endpoint for the file uploading which takes the content type and name of the file and returns a pre-signed URL for a secure file upload.

The backend codebase organization is represented in Figure 4.

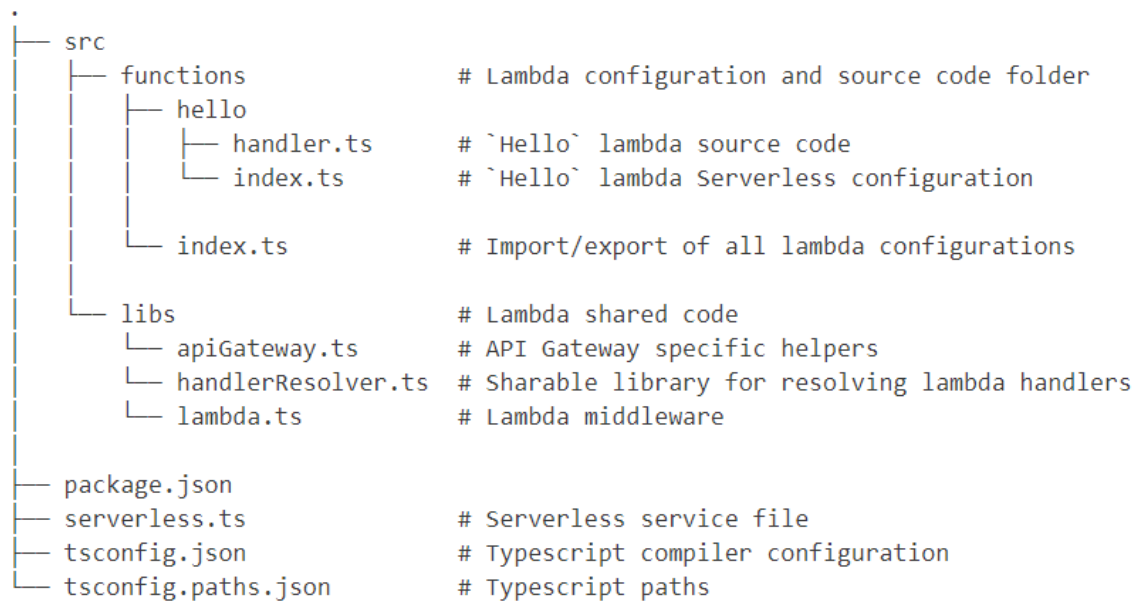

Figure 4. Codebase organization

The project codebase is mainly located in the *src* folder. It is divided into a *functions* folder which contains Lambda handlers and configurations, and a *libs* folder which holds shared code among the functions (for instance, database classes for interactions). Lambdas are stored in their own modular folders and imported/exported in the central *index.ts*. At the root level, the *serverless.ts* file registers lambdas, resources and permissions for AWS deployment.

The codebase is organized to ensure modularity and maintainability. With a clear structure that separates functions, libraries and configurations, the creation of new functions is straightforward.

4.1.1 Setting up Serverless Framework

The backend implementation began with configuring the Serverless Framework. The first step involved creating a dedicated IAM role with an AdministratorAccess policy to manage AWS resources. Once the role was created, associated user credentials were used to authenticate and log in to AWS via the Serverless Framework.

Figure 5 shows the base configuration for Lambda functions. These settings define Node.js as the runtime environment, eu-north-1 (region variable) as the

deployment region and the size of allocated memory of 256 MB. Additionally, the configuration includes a Keycloak authorizer (to verify the validity of the JWT) and an environment variable which are injected into the Lambda container and used within the function logic.

```
const serverlessConfiguration: AWS = {
  service: "home-lambdas",
  frameworkVersion: "3",
  plugins: [
    "serverless-esbuild",
    "serverless-deployment-bucket",
    "serverless-offline",
    "serverless-dynamodb"
  ],
  provider: {
    name: "aws",
    runtime: "nodejs16.x",
    region: region,
    deploymentBucket: {
      name: isLocal ? "local-bucket" : `\${self:service}-\${opt:stage}-${region}-deploy`
    },
    memorySize: 256,
    timeout: 60,
    apiGateway: {
      minimumCompressionSize: 1024,
      shouldStartNameWithService: true,
    },
    httpApi: {
      cors: true,
      authorizers: {
        "homeKeycloakAuthorizer": {
          identitySource: "$request.header.Authorization",
          issuerUrl: env.AUTH_ISSUER,
          audience: ["account"]
        }
      },
    },
    environment: {
      HOME_BUCKET_NAME: "${self:custom.s3BucketName.${opt:stage}}",
      HOME_BUCKET_REGION: region
    },
```

Figure 5. Base Lambda's configurations

Figure 6 illustrates the Serverless Framework resource's definition, specifically the configurations for the Articles DynamoDB table. This includes setting up the primary key as a hash type and a named attribute of a string type. Similarly, a Global Secondary Index (GSI) is defined with its own key attributes and projection type. The table is configured with provisioned read and write capacity units - provisioned reading and writing "power" for DynamoDB.

```
resources: {
  Resources: {
    Articles: {
      Type: "AWS::DynamoDB::Table",
      DeletionPolicy: "Delete",
      Properties: {
        TableName: "Articles",
        AttributeDefinitions: [
          { AttributeName: "id", AttributeType: "S" },
          { AttributeName: "path", AttributeType: "S" }
        ],
        KeySchema: [{ AttributeName: "id", KeyType: "HASH" }],
        GlobalSecondaryIndexes: [
          {
            IndexName: "GSI_Path",
            KeySchema: [{ AttributeName: "path", KeyType: "HASH" }],
            Projection: { ProjectionType: "KEYS_ONLY" },
            ProvisionedThroughput: {
              ReadCapacityUnits: 1,
              WriteCapacityUnits: 1
            }
          }
        ],
        ProvisionedThroughput: {
          ReadCapacityUnits: 1,
          WriteCapacityUnits: 1
        }
      }
    },
  },
},
```

Figure 6. Serverless resource configurations

In order to provide Lambdas with access to created resources, the IAM permissions should be granted. Figure 7 outlines the required permissions for the Articles table, including operations such as query, scan, get or delete item. Additionally, it defines an access point for Lambdas to create objects in the s3 bucket (allowing s3:PutObject).

```
iam: {
  role: {
    statements: [
      {
        Effect: "Allow",
        Action: [
          "s3:PutObject"
        ],
        Resource: isLocal ? "*" : "arn:aws:s3:::${self:custom.s3BucketName.${opt:stage}}/*"
      },
      {
        Effect: "Allow",
        Action: [
          "dynamodb:DescribeTable",
          "dynamodb:Query",
          "dynamodb:Scan",
          "dynamodb:GetItem",
          "dynamodb:PutItem",
          "dynamodb:UpdateItem",
          "dynamodb:DeleteItem"
        ],
        Resource: isLocal ? "*" : [
          "arn:aws:dynamodb:${self:provider.region}:*:table/Articles",
          "arn:aws:dynamodb:${self:provider.region}:*:table/Articles/index/GSI_Path"
        ]
      }
    ]
  }
}
```

Figure 7. Serverless configure IAM roles

This setup simplifies the deployment process by providing a complete configuration that includes all the necessary AWS services and permissions.

4.1.2 Handle CRUD for articles

Operations on Articles are handled by the *ArticlesApiService* library, implemented as a class containing a private, read-only *docClient* attribute. This attribute is an instance of AWS SDK *DocumentClient* which facilitates interaction with DynamoDB tables, including the Article table.

One of the core methods, *listArticles* (Figure 8), is designed to scan the table with parameters including table name, filter expression, and projection expression to exclude the content attribute.

```
/**
 * Lists all article entries
 *
 * @returns list of article entries (excluding content)
 */
public listArticles = async(draft: string = "false", pathPrefix?: string): Promise<ArticleMetadataModel[]> => {
  const params: AWS.DynamoDB.DocumentClient.ScanInput = {
    TableName: TABLE_NAME,
    ExpressionAttributeNames: {
      "#path": "path",
    },
    ProjectionExpression: "id, title, description, #path, coverImage, createdBy, createdAt, lastUpdatedBy, last
    FilterExpression: "draft = :draft",
    ExpressionAttributeValues: { ":draft": draft === "true" ? true : false}
  };

  if (pathPrefix) {
    params.FilterExpression = params.FilterExpression + " AND begins_with(#path, :path)";
    params.ExpressionAttributeValues[":path"] = pathPrefix;
  }

  const articles = await this.docClient.scan(params).promise();
  return articles.Items as ArticleMetadataModel[];
};
```

Figure 8. List articles method

The *createArticle* and *updateArticle* methods (Figure 9) are implemented similarly. They both receive the new article entry and execute put operation. If the new article entry has the id that already exists in the table, the article will be overwritten.

```
/**
 * Creates an article entry
 *
 * @param article article entry
 * @returns created article entry
 */
public createArticle = async(article: ArticleModel): Promise<ArticleModel> => {
  await this.docClient
    .put({
      TableName: TABLE_NAME,
      Item: article
    }).promise();

  return article;
};
```

Figure 9. Crete/update article entry

In order to avoid frequent full-item overwrites when marking an article as read, a separate method *updateArticleReadBy* is implemented (Figure 10). This method registers updates only specific attributes of the existing item by using the update operation, rather than replacing the entire entity.

```
/**
 * Updates the readBy and related read date attributes of an article entry
 *
 * @param id id of article to be updated
 * @param userId user id to include in the read list
 */
public updateArticleReadBy = async(id: string, users: string[]) => {
  const newDate = new Date().toISOString();
  const params = {
    TableName: TABLE_NAME,
    Key: { id: id },
    UpdateExpression: "SET readBy = :users, lastReadAt = :newDate, lastUpdatedAt = :newDate",
    ExpressionAttributeValues: {
      ":users": users,
      ":newDate": newDate
    }
  };

  await this.docClient.update(params).promise();
};
```

Figure 10. Update article read attributes

Operations for deletion and retrieval articles by id execute *get* or *delete* operations of *docClient* based on the provided id. Since path GSI stores only path-to-id association, the *findArticleByPath* (Figure 11) first retrieves the article id by querying the GSI with the given path. It then performs a get operation using the retrieved id.

```
/**
 * Finds a single article entry by path
 *
 * @param path unique article path
 * @returns article entry or null if not found
 */
public findArticleByPath = async(path: string): Promise<ArticleModel | null> => {
  const params = {
    TableName: TABLE_NAME,
    IndexName: gsiPath,
    KeyConditionExpression: "#path = :path",
    ExpressionAttributeNames: { "#path": "path" },
    ExpressionAttributeValues: { ":path": path }
  };

  const result = await this.docClient.query(params).promise();
  const articleRecord = result.Items.length !== 0 ? result.Items[0] : undefined;

  if (articleRecord?.id) {
    const article = await this.findArticleById(articleRecord.id);
    return article;
  }
};
```

Figure 11. Find article by path

As previously mentioned, Lambdas are stored in isolated folders. The *index.ts* file (Figure 12) configures the functions, defining HTTP events with method, endpoint, authorizer, and the reference to the actual handler body.

```
export default {
  handler: `${handlerPath(__dirname)}/handler.main`,
  events: [
    {
      httpApi: {
        method: 'delete',
        path: '/articles/{id}',
        authorizer: {
          name: "homeKeycloakAuthorizer"
        }
      },
    },
  ],
};
```

Figure 12. Function index.ts

The handler.ts file defines a function that parses query, path parameters and body of the requests. This function handler calls appropriate methods from an instance of article API class, passing parsed data as parameters. In some cases, Lambda functions also process the data, for instance, the function associated with the GET /articles endpoint sorts the retrieved articles in descending order by the *lastUpdatedAt* attribute, ensuring that the frontend receives the most recent articles first.

The ArticlesApiService instance, interacting with the database, is created outside of the function to avoid reinitialization process in case of Lambda warm start (Figure 13).

```
const dynamoDb = new DocumentClient();
const articleService = new ArticlesApiService(dynamoDb);

/**
 * Handler for deleting an article entry in DynamoDB.
 *
 * @param event - API Gateway event.
 */
const deleteArticleHandler: APIGatewayProxyHandler = async (event: APIGatewayProxyEvent) =>
```

Figure 13. Creation of ArticlesApiService instance

Moreover, the handler validates the input data. For example, the handler associated with the DELETE /articles/{id} endpoint decodes the provided JWT, checks for administrative roles to authorize the action and ensures that the id parameter is provided and that an article with the given id exists. The logic for these checks is illustrated in the Figure 14. Each error is accompanied by a descriptive error message to help in debugging.

```
const authorizationHeader = event.headers?.Authorization;
if (!authorizationHeader)
  return {
    statusCode: 400,
    body: JSON.stringify({
      code: 400,
      message: "Missing Authorization header.",
    }),
  }

const token = authorizationHeader.split(' ')[1];
if (!token)
  return {
    statusCode: 400,
    body: JSON.stringify({
      code: 400,
      message: "Missing Bearer token.",
    }),
  }

const decodedJWT = jwt.decode(token);
if (!decodedJWT.realm_access.roles.includes("admin"))
  return {
    statusCode: 403,
    body: JSON.stringify({
      code: 403,
      message: "Access denied. Admin privileges required.",
    }),
  }

const { id } = event.pathParameters || {};
if (!id) {
  return {
    statusCode: 400,
    body: JSON.stringify({
      code: 400,
      message: "Missing or invalid 'id' path parameter.",
    }),
  };
}
```

Figure 14. Validation of request

When functions are created for deployment, they must simply be listed as functions in the *serverless.ts* configuration file.

4.1.3 Handle image upload

The Lambda function responsible for file uploads operates similarly to the functions described in Section 4.1.2. It receives the file path and content type as input parameters, verifies that the content type matches the allowed image format and then generates a pre-signed URL for uploading.

Figure 15 illustrates the process of generating a pre-signed URL which involves defining the operation type and the expiration time in seconds. This URL includes a cryptographic signature which is a hash generated using the request parameters, ensuring the URL's security.

```
const { HOME_BUCKET_NAME, HOME_BUCKET_REGION } = process.env;
const s3Client = new S3Client({ region: HOME_BUCKET_REGION });

/**
 * Function generates presigned url
 *
 * @param path  path to file in S3
 * @param contentType contentType of the file
 * @returns Buffer object
 */
export const generatePreSignedUrl = async(path: string, contentType: string) => {
  const command = new PutObjectCommand({
    Bucket: HOME_BUCKET_NAME,
    Key: path,
    ContentType: contentType
  })

  return await getSignedUrl(s3Client, command, { expiresIn: 3600 });
}
```

Figure 15. Generation of pre-signed URL

The actual file uploads are processed from the client application using the provided pre-signed URL, with the file included in the request.

4.2 Specification update

In order to streamline the development of client-side methods for retrieving data from an API and to clearly define the structure of the API, specifications are employed in the Swagger format.

The use of Open API generator tool enables the automation of client integration by generating request functions directly from the API specification. Consequently, changes in the backend are effectively synchronized in the client fetch system by the spec updates and generator tool, eliminating the need for manual adjustments on client-side code.

Following the creation of the Lambda functions, the next step involved updating the company's backend specifications. This included the definition of new endpoints, request parameters and return objects of article models in accordance with the format.

At the frontend, execution of the command (Figure 16) automatically generates the client-side functions for data retrieval. This command takes the backend spec as input and outputs the generated code to the specified directory.

```
openapi-generator-cli generate \
  -i home-lambdas-API-spec/swagger.yaml \
  -o ./src/generated/homeLambdasClient \
  -c generator-config.json \
  -g typescript-fetch
```

Figure 16. Open API generator tool command

This src/generated/homeLambdasClient directory contains code generated from the Open API tool. This folder includes data models for the API request and response, serialization/deserialization utilities for conversion between these models and JSON, and fetch-based functions interacting with the API.

This design supports a high degree of automation and reduces the impact of backend changes on the frontend implementation, therefore, improving maintainability and development efficiency.

4.3 Frontend implementation

The frontend implementation involves the creation of a card component and several screen components. The routing table and corresponding screens are as follows:

- /wiki-documentation – The base article screen with the list of articles.
- /admin/wiki-documentation – The admin version of the base article screen.
- /wiki-documentation/{article path} – The article screen displaying the content of a specific article.

Since the project did not have a predefined design, the visuals were developed and aligned with the company's existing design preferences. The Material UI library was utilized for compatibility with the metatavu-home project, providing pre-built components.

The following sections will cover the details of the article's service implementation.

4.3.1 Card component

The entry point for the service is the article card component (Figure 17), located on the main home page. This card displays the most recently updated article. The card component fetches a list of articles, excluding draft articles, and sets globally accessed states with Jotai that are referred as atoms. This is the core logic used to reduce the number of requests to the backend. The component then takes first the most recent article based on update, creation or read activity. In order to determine which operation (create, update, or read) is the most recent, the component compares the *lastUpdatedAt* property with the *createdAt* and *lastReadAt* attributes. If a match is found, the article is either created or read; otherwise, the article is simply updated.

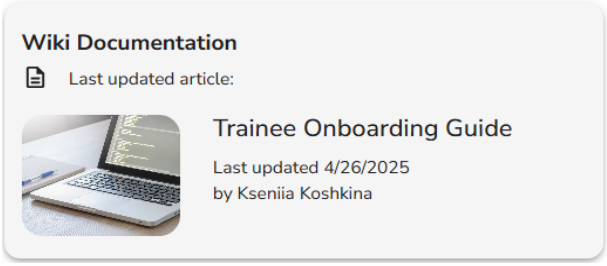

Figure 17. Home card component

The admin home screen also features an article card (Figure 18). The admin card, in addition to the article list retrieval, fetches draft articles and sets corresponding *draftArticles* atom. This card displays the total number of draft articles which are pending review.

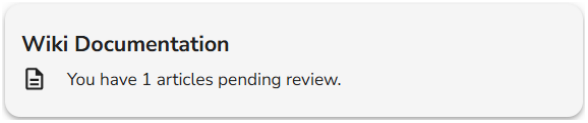

Figure 18. Admin home card

Clicking on the card will navigate to the /wiki-documentation screen or /admin/wiki-documentation.

4.3.2 Wiki documentation screen component

The wiki documentation screen component takes the article list from the atom if it is defined. When navigating from the home screen, the screen component directly takes the value and puts it inside the *displayedArticles* state. This state enables filtering articles based on the search input. If the article's atom is not defined, the component fetches the list of articles in the same way as the home card and sets the atom value accordingly.

The component (Figure 19) includes a carousel of the most recently updated articles, a search component and the actual list of articles. The article list card is similar to the one on the home screen, but with added tags. The recommendation carousel is large to attract attention and includes a brief description of the article. It displays the most recent read, updated and created articles, one per action.

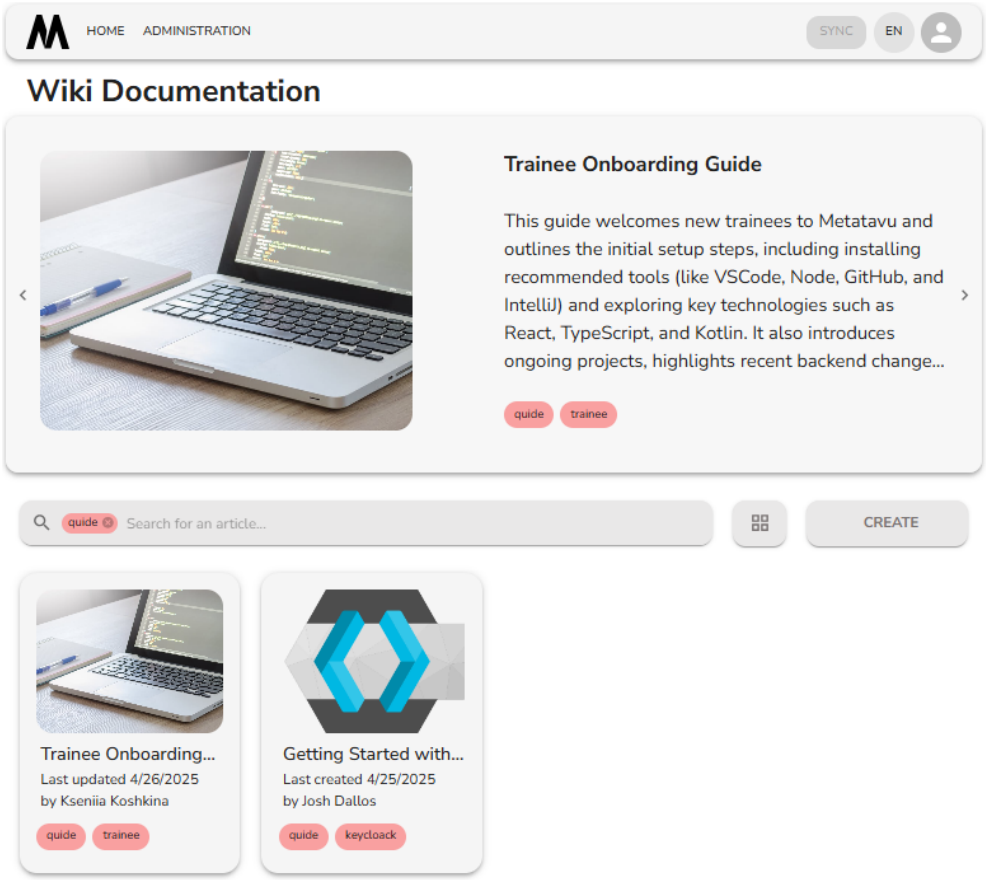

Figure 19. Wki-documentaion screen

The component also tracks the article tags, creating a set of them and placing them in the global tags atom. These tags can then be used to search for specific articles. The icon button allows users to toggle the view option between grid view (Figure 19) and list view (Figure 20).

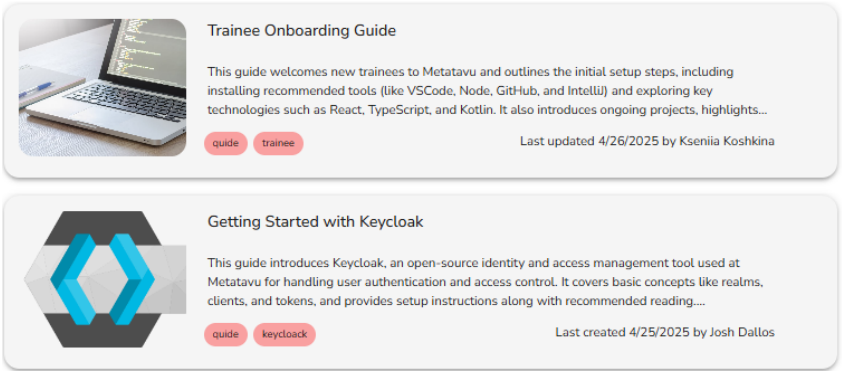

Figure 20. List view

The creation button updates the component's state, rendering the article creation form (Figure 21).

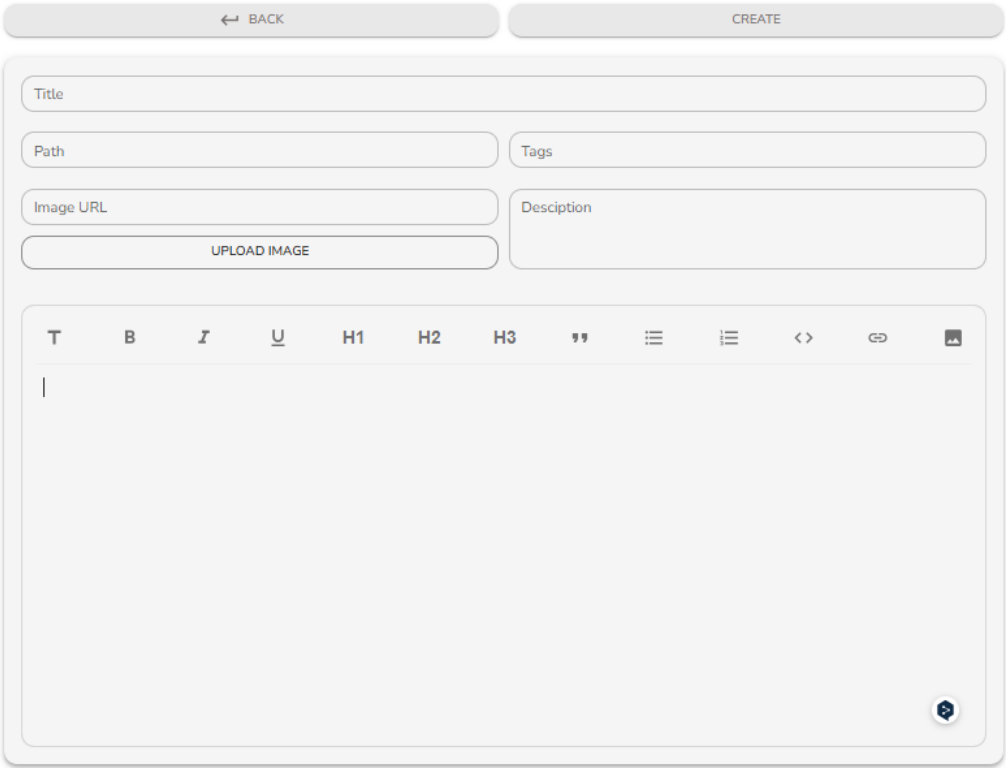

Figure 21. Form to create/update article

The form includes several input fields, such as title, path, tags, description and cover image. The tag input leverages an atom value and supports tag suggestions based on the existing variants. The file content can be typed in the rich text editor field. This form is used for both article creation and update operations. The created or updated articles by users change the draft marker and are removed from the user view, appearing in the administrator's list of articles, pending a review.

The admin screen is implemented similarly, the only difference being the ability to choose the article type based on the draft property (Figure 22).

Figure 22. Admin search filed

This switch between pending (draft) and all articles is achieved by toggling *displayedArticles* state value between *articles* and *draftArticles* atoms. Additionally, the admin screen includes functionality to review articles, publish them by changing the draft marker, and deleting them if necessary.

4.3.3 Article screen component

Clicking on the article card component navigates to the specific article screen accessible via the route /wiki-documentation/{some article path} (Figure 23). This screen component fetches the complete article entry, rendering Markdown text using the react-markdown library.

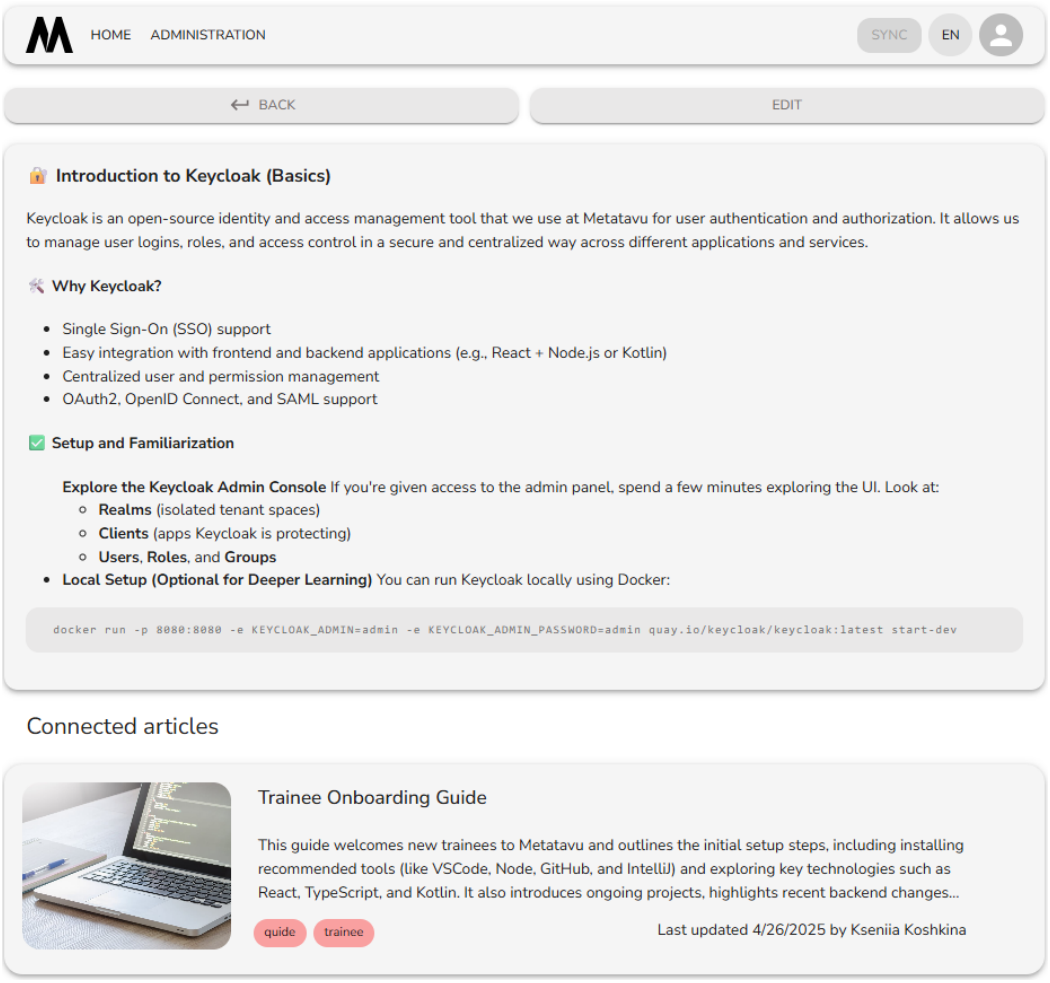

Figure 23. Article screen component

The screen also recommends related articles by fetching a list with a defined path prefix. For example, /guide/trainee_onboarding_guide and /guide/keycloak share

the same path prefix - guide. Connected articles are displayed as the list item component from the wiki documentation screen.

4.3.4 Rich text editor with Lexical

Rich text editor must feature Markdown supported text formatting. This includes text styles such as inline, bold and underline and block type formatting involving headers, quotes, code, links and images.

The creation of a Lexical editor begins with the *LexicalComposer* component (Figure 24) which initializes the editor. Inside this component, various plugins are listed to build a fully functional rich text editor. For instance, the *RichTextPlugin* provides the core editing interface, the *HistoryPlugin* enables undo/redo functionality, and a custom *ToolBar* plugin provides a toolbar component for applying text formatting.

```
const initialConfig = {
  namespace: 'MyEditor',
  theme: lexicalTheme,
  onError,
  nodes: [
    HeadingNode,
    ListNode,
    ListItemNode,
    CodeNode,
    QuoteNode,
    LinkNode,
    ImageNode
  ],
  editorState: () => $convertFromMarkdownString(markdownContent, TRANSFORMERS_WITH_IMAGE)
};

return (
  <LexicalComposer initialConfig={initialConfig}>
      <ToolBar/>
      <RichTextPlugin
        contentEditable={
          <ContentEditable className='editor'/>
        }
        ErrorBoundary={LexicalErrorBoundary}
      />
      <HistoryPlugin/>
      <AutoFocusPlugin/>
      <OnChangePlugin setEditorState={setEditorState}/>
      <LinkPlugin/>
      <ListPlugin/>
      <TabIndentationPlugin/>
  </LexicalComposer>
);
```

Figure 24. Implementation of Lexical editor

The *LexicalComposer* takes an initial configuration object as a parameter. This specification includes a theme and a list of supported nodes. Nodes represent block formatting and serve as fundamental elements for the editor state's node tree. The theme provides a mapping of nodes to CSS style classes, featuring customized visual representation of blocks.

Additionally, the configuration sets the initial editor state. When editing an article, the Markdown content is converted to supported Lexical object format. The *$convertFromMarkdownString* facilitates this conversion. It takes a transformer object as a second parameter which defines the match options for markdown syntax and maps them with the corresponding nodes. The function *$convertToMarkdownString* performs the reverse operation serializing editor state to Markdown string.

The Editor stated is global and can be accessed within plugins. The tool bar plugin applies formatting to inline text and blocks. Block formatting is applied by creating a node and inserting it into the selected part of text modifying the global editor state.

5 CONCLUSION

The thesis outcome meets the initial purpose of providing a platform for documentation management. The frontend application has an intuitive and responsive user interface. The Lexical text editor is integrated into the service, operates correctly and provides the ability to scale styles. The backend functions as expected, processing articles and reliably returning the necessary data.

However, there is room for development to scale the service further. For instance, implementing version control to save article entry before editing so that the modified version could be automatically restored by administrators. Moreover, if the number of articles grows, the system will require reconsider the access pattern to avoid the entire table scans. This could be achieved by retrieving articles based on months and years, combined with the implementation of paging.

Overall, I am satisfied with the developed solution. Throughout the implementation process, I faced challenges in design decisions, selecting the best practices and exploring the underlying technologies. This experience helped me to improve my development skills and deepen my understanding of full-stack development.

REFERENCES

AWS Lambda The Ultimate Guide. 2024. Serverless. Web page. Available at: https://www.serverless.com/aws-lambda [Accessed 21 February 2025].

AWS Regions. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/global-infrastructure/latest/regions/aws-regions.html [Accessed 8 March 2025].

Best practices for modeling relational data in DynamoDB. 2025. Amazon web Services. Web page. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-relational-modeling.html#SQLtoNoSQL.relational-modeling2 [Accessed 9 March 2025].

Best practices for querying and scanning data in DynamoDB. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-scan.html [Accessed 9 March 2025].

Core components of Amazon DynamoDB. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html [Accessed 9 March 2025].

DeBrie, A. 2020. The DynamoDB Book.

Difference Between Virtual DOM and Real DOM. 2025. GeeksforGeeks. Web page. Available at: https://www.geeksforgeeks.org/difference-between-virtual-dom-and-real-dom/ [Accessed 20 April 2025].

How CloudFormation works. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cloudformation-overview.html [Accessed 9 March 2025].

Introduction to JSON Web Tokens. 2025. Okta. Web page. Available at: https://jwt.io/introduction [Accessed 20 April 2025].

Kato, D. n.d. Jotai Documentation. Web page. Available at: https://jotai.org/ [Accessed 20 April 2025].

Krause M. 2024. The complete developer.

Lexical Documentation. 2025. Meta Platforms, Inc. Web page. Available at: https://lexical.dev/docs/intro [Accessed 20 April 2025].

Lambda runtimes. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html [Accessed 7 April 2025].

NoSQL design for DynamoDB. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html [Accessed 1 March 2025].

OpenAPI Generator Documentation. 2025. OpenAPI-Generator. Web page. Available at: https://openapi-generator.tech/ [Accessed 1 March 2025].

Richter, F. 2025. Amazon and Microsoft Stay Ahead in Global Cloud Market. Web page. Available at: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ [Accessed 10 March 2025].

Serverless Framework Concepts. 2024. Serverless. Web page. Available at: https://www.serverless.com/framework/docs/providers/aws/guide/intro [Accessed 15 March 2025].

Swagger Documentation. 2025. SmartBear Software. Web page. Available at: https://swagger.io/docs/specification/v3_0/about/ [Accessed 27 April 2025].

Understanding the Lambda execution environment lifecycle. 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html#cold-start-latency [Accessed 12 April 2025].

What is Amazon DynamoDB? 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html [Accessed 1 March 2025].

What is Amazon EC2? 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html [Accessed 1 March 2025].

What is Amazon S3? 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html [Accessed 1 March 2025].

What is AWS Lambda? 2025. Amazon Web Services. Web page. Available at: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html [Accessed 2 March 2025].

What is cloud computing? 2025. Amazon Web Services. Web page. Available at: https://aws.amazon.com/what-is-cloud-computing/ [Accessed 1 March 2025].

What is a RESTful API?. 2025. Amazon Web Services. Web page. Available at:

https://aws.amazon.com/what-is/restful-api/ [Accessed 16 April 2025].