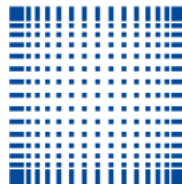


NTT Data



hochschule mannheim



kananana

Architecture Documentation

Knownana

Authors

Vladislav Chumak
Alexander Schramm
Jochen Schwander

Table of Contents

| | |
|---|-----------|
| Table of Contents | 2 |
| Preface..... | 4 |
| 1 Introduction and Goals | 5 |
| 1.1 Key Functionality..... | 5 |
| 1.2 Quality Goals | 6 |
| 1.3 Stakeholders..... | 6 |
| 1.4 Architecture Drivers | 7 |
| 1.5 Non Drivers | 11 |
| 2 Architecture Constraints..... | 13 |
| 2.1 Technical Constraints | 13 |
| 2.2 Organizational Constraints | 13 |
| 3 System Scope and Context | 15 |
| 4 Building Block View..... | 16 |
| 4.1 Level 1..... | 16 |
| 4.2 Level 2..... | 17 |
| 4.3 Level 3..... | 22 |
| 5 Runtime View..... | 28 |
| 5.1 Create Article | 28 |
| 5.2 Read Article | 30 |
| 5.3 Search Article | 32 |
| 6 Deployment View | 34 |
| 7 Concepts..... | 35 |
| 7.1 Persistency | 35 |
| 7.2 Testability | 37 |
| 7.3 Data Model | 37 |
| 7.4 Remote API | 38 |
| 7.5 Configuration | 38 |
| 8 Design Decisions | 39 |
| 8.1 Search Engine | 39 |
| 8.2 Persistence | 42 |
| 8.3 Backend Technology | 45 |

| | | |
|-----------|---|-----------|
| 8.4 | Frontend Technology | 46 |
| 8.5 | Folder Structure | 47 |
| 8.6 | WYSIWYG Editor | 49 |
| 8.7 | File Upload | 50 |
| 9 | Quality Scenarios | 52 |
| 9.1 | Quality Tree | 52 |
| 9.2 | Scenarios | 52 |
| 10 | Technical Risks..... | 58 |
| 10.1 | Risk: High Effort for Search-Engine Configuration | 58 |
| 10.2 | Risk: High Memory Consumption of Search-Engine | 58 |
| 11 | Glossary..... | 60 |
| | List of Figures | 61 |
| | List of Tables..... | 61 |
| | List of Abbreviations..... | 62 |
| | Version History | 63 |

Preface

This document contains the architecture description of the knowledge base system Knownana. It's structured after the arc42 template¹ with some changes. arc42 was chosen because it's an industry proven way to document an architecture and it's the customer's preferred documentation template.

In contrast to arc42 architecture drivers are explicitly listed in chapter *1.4* and usual features that are not wanted by the customer are shown in chapter *1.5*.

The arc42 chapter "Solution Concept" is not found in this document because there are no major concepts that are important for the whole system. All of the important concepts are described in the chapters where they are needed. So to avoid redundancy the chapter is not in this document.

The chapter arc42 "Concepts" contains a lot of subchapters for a variety of different concerns trying to match all possible uses. In this document only to this system applicable subchapters are present, contrary to arc42's recommendation to keep all subchapters and note why they are not relevant. This is done to not unnecessarily bloat the document's size.

¹ <http://arc42.org/>

1 Introduction and Goals

This document specifies the software architecture of the knowledge base platform which is being implemented at Mannheim University of Applied Sciences as part of MSP program for a customer from NTT DATA Company.

Before the completion of this project NTT DATA Company has managed its knowledge by means of Microsoft SharePoint. SharePoint does not satisfy the requirements of the company regarding simplicity, generic usage, ease of use and global accessibility. The goal of the knowledge base platform is to address the shortcomings of SharePoint and enhance the corporate infrastructure in NTT DATA Company. The knowledge base has to provide an easy way to collect and share knowledge within NTT DATA Company. It should be a web application that allows users to populate their knowledge in arbitrary formats quickly on a central server, and make it easy to find for other users.

1.1 Key Functionality

The following use case diagram illustrates the key functionalities of the knowledge base platform.

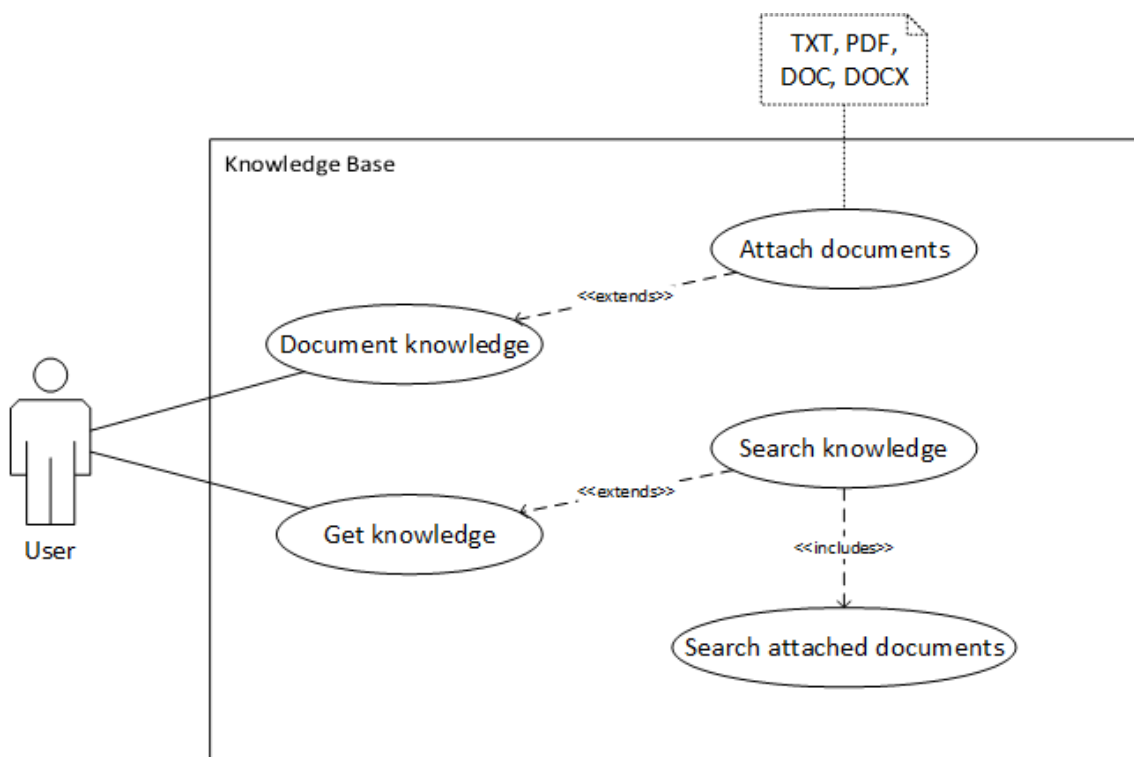


Figure 1: key functions of the knowledge base platform

1.2 Quality Goals

The following quality goals have top priority for the customer.

1.2.1 Usability

The system must be intuitive and easy in use. A new user should be able to get familiar with the knowledge base platform within 15 minutes. A user should be able to create a new knowledge article within 5 minutes.

1.2.2 Testability

The backend code should be tested in an automated way reaching a branch coverage of 85%.

1.2.3 Documentation

The code must be documented in a manner similar to JavaDoc.

1.3 Stakeholders

The following table represents the stakeholders who are related to the architecture.

1.3.1 Software Architects

- Understand the requirements of the customer
- Design and document the architecture
- Understand the existing architecture
- Guide developers and testers

1.3.2 Developers

- Implement architecture components

1.3.3 Testers

- Design and implement tests for the architecture components

1.3.4 Customer

- Mr. Andreas Maier, NTT DATA Deutschland GmbH
- Expresses and prioritizes requirements which eventually affect the set of architecture drivers
- Wants to be informed about the chosen technologies early

1.3.5 Trainer

- Dr. Jens Knodel, Mannheim University of Applied Sciences
- Supports architecture team with architecture know-how

1.4 Architecture Drivers

This section contains future and realized architecture drivers. Realized architecture drivers are drivers inside the scope of the current requirements, while future drivers are anticipated drivers likely to be driving the architecture in the future.

1.4.1 Realized

This section contains all current drivers. Each driver has his own table.

| Categorization | | |
|----------------|--|---|
| Driver Name | Full text search | |
| Driver ID | AD-01-Search | |
| Status | Realized | |
| Priority | High | |
| Description | | Quantification |
| Environment | The system contains articles. | At least one article |
| Stimulus | The user wants to find information about a specific problem. | Enters at least one keyword |
| Response | The system presents relevant articles. | All articles containing respective keywords are presented |

Table 1: AD-01-Search

| Categorization | | |
|--------------------|---|---|
| Driver Name | Usability | |
| Driver ID | AD-02-Usability | |
| Status | Under Design | |
| Priority | Medium | |
| Description | | Quantification |
| Environment | The system is running. | - |
| Stimulus | The user interacts with the system. | The user didn't use the system. |
| Response | The user can use the system intuitively and productively. | The user understands the system and can create his first article within 15 minutes. |

Table 2: AD-02-Usability

| Categorization | | |
|--------------------|--|---|
| Driver Name | Generic Usage | |
| Driver ID | AD-03-Generic | |
| Status | Under Design | |
| Priority | High | |
| Description | | Quantification |
| Environment | The system is running. | - |
| Stimulus | The user wants to create an article containing documents. | User uploads at least one file. |
| Response | The system supports PDF, Microsoft Office and text files regardless of their format. | The system accepts and indexes an unlimited number docx, doc, pdf and txt files. As well as a text of any length, containing tables, bullet points, text formatting and images. |

Table 3: AD-03-Generic

| Categorization | | |
|--------------------|---|--------------------------------------|
| Driver Name | Testability | |
| Driver ID | AD-04-Testability | |
| Status | Under Design | |
| Priority | Medium | |
| Description | | Quantification |
| Environment | The system implementation started. | Source code is available. |
| Stimulus | The tester wants to test the system. | - |
| Response | The code is testable in an automated way. | 85% branch coverage can be achieved. |

Table 4: AD-04-Testability

| Categorization | | |
|--------------------|---|---|
| Driver Name | Persistence of articles | |
| Driver ID | AD-05-Persistence | |
| Status | Under Design | |
| Priority | High | |
| Description | | Quantification |
| Environment | The system contains articles with attached documents. | At least one article with one document. |
| Stimulus | The user wants to read an article including its documents. | The user has found the article. |
| Response | The system displays the articles content and offers the documents for download. | The retrieved documents are in their original format. |

Table 5: AD-05-Persistence

1.4.2 Future

This section contains all future drivers. Each driver has his own table.

| Categorization | | |
|--------------------|---|---------------------------------------|
| Driver Name | Mobile Usage | |
| Driver ID | AD-06-Mobile | |
| Status | Conceptual | |
| Priority | Low | |
| Description | | Quantification |
| Environment | The system is running. The user is not in his office and only has his smart-phone with him. | At least one article. |
| Stimulus | The user wants to read an article. | The user has found the article. |
| Response | The system adapts to the users device and displays the article. | The systems usability is still given. |

Table 6: AD-06-Mobile

| Categorization | | |
|--------------------|---|---|
| Driver Name | Scalability | |
| Driver ID | AD-07-Scalability | |
| Status | Conceptual | |
| Priority | Low | |
| Description | | Quantification |
| Environment | The system outgrew its initial user scope. | The system has more than one hundred users. |
| Stimulus | The administrator wants to increase the maximum load the system can handle. | - |
| Response | The system scales well horizontally. | All components are clusterable. |

Table 7: AD-07-Scalability

| Categorization | | |
|-----------------------|--|---|
| Driver Name | Integration with other systems | |
| Driver ID | AD-08-Integration | |
| Status | Conceptual | |
| Priority | Low | |
| Description | | Quantification |
| Environment | The system is running in an environment with other systems. | At least one other system is available. |
| Stimulus | The user wants the system to automatically import information from another system. | - |
| Response | The system offers interfaces to integrate other systems. | At least one interface. |

Table 8: AD-08-Integration

1.5 Non Drivers

The following sub chapters contain typical drivers one would assume to find in a project which are explicitly not relevant for the context of the knowledge base. For each so called non-driver a rationale is given why it is not relevant for this project.

1.5.1 User Management

The knowledge base does not have any form of user management. One of the key drivers is the usability which contains fast usage. Therefore, users do not have to login to the application. All users have the same user roles and filling out the author tag of an article is optional.

1.5.2 Integration

The customer explicitly demands the knowledge base to not require any remote systems. This includes the SharePoints which store project specific documents at the customer's facility, as well as the corporate LDAP server.

1.5.3 Security

As the knowledge base is designed to be an internal tool only accessible within the customer's intranet, there are no security constraints for the project.

1.5.4 Safety

The knowledge base context does not cover safety relevant issues.

1.5.5 Privacy

The customer is responsible for compliance towards legal and corporate privacy regulations. The knowledge base stores all files and data in plain text and users are responsible to not upload classified documents.

2 Architecture Constraints

The following section lists all constraints influencing the development of this project.

2.1 Technical Constraints

2.1.1 Hardware Constraints

There are no specific hardware constraints. The customer wishes the system to require as little hardware resources as possible.

2.1.2 Software Constraints

There are no restrictions in software which may be used in the context of this project. The development team is free to choose the technologies and development tools in order to complete this project. The system may rely on third party libraries which require making the source code of the system available for public consumption or rather open-source.

2.1.3 Operating System Constraints

The customer wants the system to run on Linux or Unix and Windows.

2.1.4 Programming Constraints

The software code should be documented in the manner similar to JavaDoc. Clean Code Conventions should be used.

2.2 Organizational Constraints

2.2.1 Organization and Structure

The development team Kanbanana of Mannheim University of Applied Sciences is responsible for system realization and in time delivery of system artefacts including the installation guide document to the customer. The customer is responsible for deployment of the system in the infrastructure of his company, as well as for the further maintenance and system evolutions.

2.2.2 Resources

The system should be realized and delivered to the customer until 22.06.2016. There are no reserved financial resources for this project. The Kanbanana project team consists of 15 members who are students of Mannheim University of Applied Sciences.

2.2.3 Organizational Standards

There are no organizational Standards which need to be followed in the context of this project.

2.2.4 Legal Factors

There are no legal aspects which need to be considered in the context of this project.

3 System Scope and Context

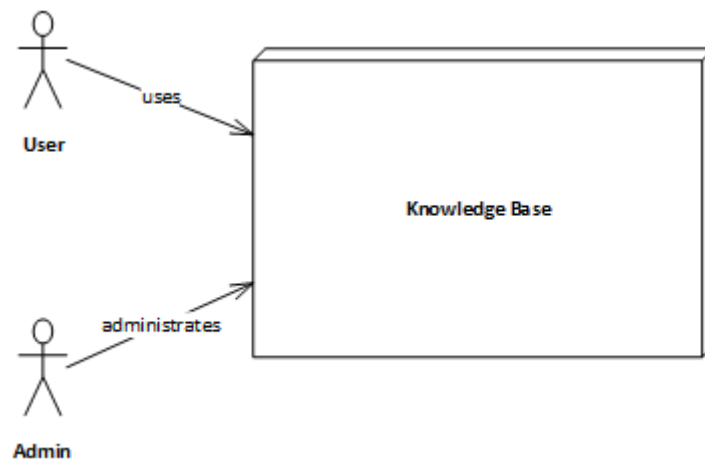


Figure 2: context diagram

The system has two roles interacting with it. The first role is the user. He can create, read, update, delete and search articles. This includes attaching documents to an article and downloading them. The second role is the admin. He can trigger the system to re-index all articles. Roles are transparent within the system, as there is no user management. The admin has direct access to the platform the knowledge base is running on, though. The system does not interact with any other system.

The system is initially developed by the Kanbanana project team. Maintenance and system evolution will be done by the customer. Additionally the customer wants to install the system himself with the help of written installation instructions.

4 Building Block View

This chapter describes the components of the knowledge base system and its relations.

4.1 Level 1

The system is a web application. It consists of two basic components – Frontend and Backend.

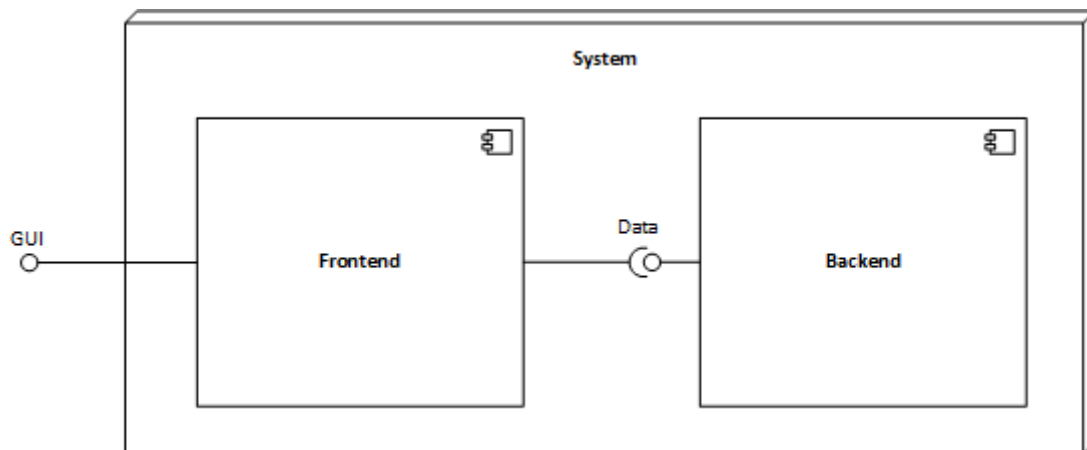


Figure 3: building block view level 1

4.1.1 Components

4.1.1.1 Frontend

The Frontend is a client side part which provides a GUI for interactions with the end user. It runs by means of a web browser installed on the end users computer. Thus, it is based on HTML/CSS and JavaScript. The GUI makes an intensive use of JavaScript which makes it very fluid and dynamic. The GUI allows the end user to access the functionalities of the Knowledge Base, such as CRUD-Operations on articles and article search.

4.1.1.2 Backend

The Backend is a server side part which manages the whole data of the knowledge base system on a central server. It supplies the Frontend with the article data it needs. It also accepts new article data.

4.1.2 Interfaces

4.1.2.1 GUI

The Frontend provides a GUI on top of HTML/CSS and JavaScript. The end user must use a web browser to access the GUI.

4.1.2.2 Data

The Frontend communicates with the Backend over AJAX.

4.1.3 Protocol

HTTPS is used as transport protocol.

4.2 Level 2

The building block view level 2 lists the internal components of level 1 components, explains them, and describes their relations.

At first a component diagram illustrates the white box view of both basic level 1 components in order to present the big picture of the main parts of the system. Afterwards the white box view for each level 1 component is explained. Furthermore the complete technology stack is presented and related to level 1 and level 2 components.

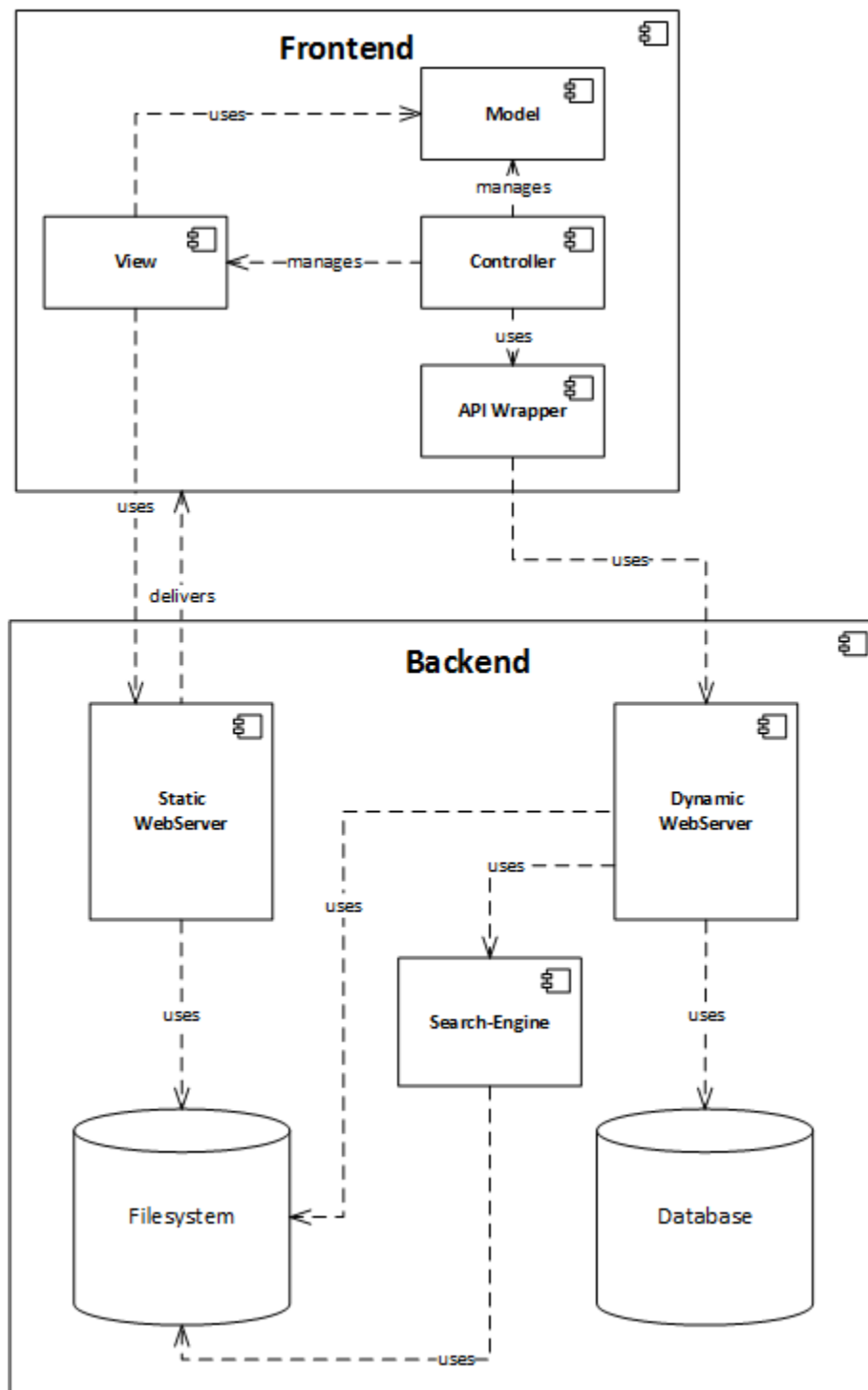


Figure 4: building block view level 2

4.2.1 Frontend

The Frontend implements the MVC-Pattern. MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns.

4.2.1.1 Model

The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

4.2.1.2 View

A presentation of data in a particular format, triggered by the controller's decision to present the data.

4.2.1.3 Controller

The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

4.2.1.4 API Wrapper

The API wrapper encapsulates all API calls to the backend. The API wrapper is the only component in the frontend which may send requests to the backend concerning the article data.

4.2.2 Backend

The backend consists of three components and two data storages.

4.2.2.1 Dynamic Webserver

The dynamic webserver component is responsible for the following tasks:

- Serving requests from the Frontend component
 - CRUD-Operations on articles and its contents
 - Document Upload
 - Search-Requests
- Search-Engine Coordination
 - Generation of search queries, delegation of search queries to search engine, and parsing of search results
 - Index Updates
- Persistence of article metadata in database
- Persistence of article content incl. attached documents on file system
- Relation management between article contents, documents, article metadata, and search engine metadata

4.2.2.2 Static Webserver

This component allows the Frontend to access static resources. Such static resources are:

- Static HTML, CSS, JavaScript, and Image files which the Frontend component consists of
- Document files such as PDF which are attached to articles

4.2.2.3 Search Engine

This component has following responsibilities:

- Processing of search queries and returning of matching results
- Search Index maintenance

In order to deliver matching articles for incoming search queries, the search engine component maintains an index which is built using article contents and document contents. This index is used during the matching process between the search query and the article contents. The article contents that need to be considered during the matching process must be communicated to the search engine by dynamic webserver component when some article content or document file gets created, updated, or deleted.

4.2.2.4 Database

All article metadata are stored in a database. Examples for such metadata are:

- Article name, author, last editor, date created, date updated
- Document name, relation to article, relation to the indexed entry in search engine, path on the file system.

4.2.2.5 File System

The file system stores all article contents including images and attached document files. It also contains Frontend files like HTML/CSS and JavaScript.

4.2.3 Technology Stack

This chapter lists all technologies used in knowledge base system und relates them to level 1 and level 2 components. Figure 5 shows Figure 4 with technologies.

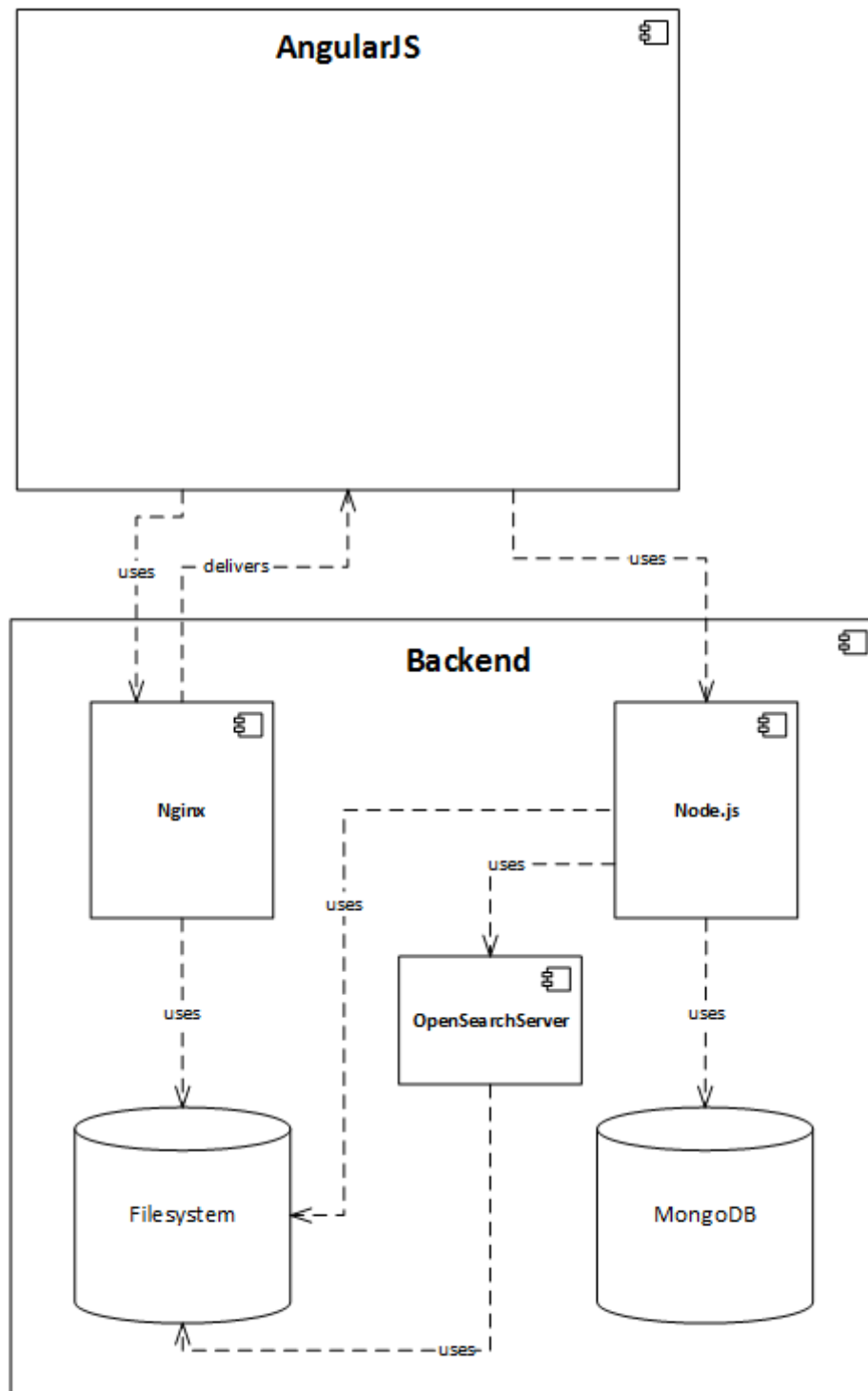


Figure 5: components with technologies

4.2.3.1 AngularJS

The complete Frontend component (level 1) is realized by means of the client side JavaScript web framework AngularJS. AngularJS implements the MVC pattern and provides a convenient API for AJAX requests.

4.2.3.2 Nginx

Static Webserver component is powered by Nginx. Nginx is a popular and lightweight HTTP-Server technology which is able to deliver static content. It also offers many other features like mail proxy server which are not used in the context of knowledge base system.

4.2.3.3 Node.js

The dynamic webserver component is powered by Node.js. Node.js provides a lightweight solution for implementation of server side logic by using JavaScript as programming language. It offers very convenient ways to realize:

- REST server implementation
- JSON handling
- MongoDB integration

4.2.3.4 OpenSearchServer

OpenSearchServer is a fully implemented search engine server which is running by means of a web container such as tomcat. It is a complete solution which implements full text search and phonetic search for many different document formats including TXT, PDF, DOC, and DOCX.

Dynamic webserver uses REST/JSON API in order to communicate with OpenSearchServer.

4.2.3.5 MongoDB

The Database component is powered by MongoDB. It's a document oriented NoSQL database. It is well supported by Node.js.

4.3 Level 3

Building block level 3 describes the internal structure of the dynamic webserver and the frontend.

4.3.1 Dynamic Webserver

The dynamic webserver consists of the five components Article Handler, Article Service, Database Connector, File System Connector and Search Engine Connector and exports a REST API for the frontend. *Figure 6* shows the internals of the dynamic webserver in a component diagram. Additionally the dynamic webserver contains the two files “www” which is the execution starting point and “server.js” which contains the initialization of the server and enables the Article Handler to get REST calls.

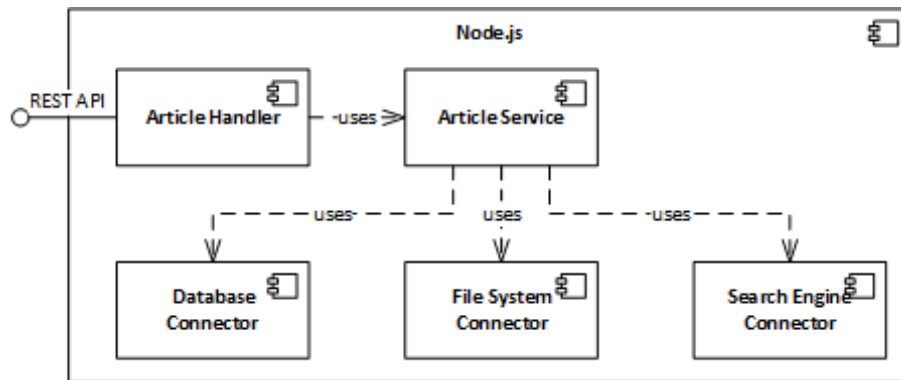


Figure 6: dynamic webserver components

4.3.1.1 Article Handler

The Article Handler provides the REST API of the dynamic webserver. It allows to create, read, update, delete and search for articles. It encapsulates the REST call handling from the business logic and delegates calls to the appropriate Article Service functions.

4.3.1.2 Article Service

The Article Service allows to create, read, update, delete and search for articles. It uses the Database Connector to access article metadata from the database and the File System Connector for persistence operations of the articles and its documents. It also uses the Search Engine Connector to search for articles.

4.3.1.3 Database Connector

The Database Connector encapsulates access to the database from the application logic.

4.3.1.4 File System Connector

The File System Connector handles all file system accesses and provides an abstraction layer away from the file system for the application logic.

4.3.1.5 Search Engine Connector

The Search Engine Connector handles all API calls to the search engine.

4.3.1.6 REST API

The operations of the REST API are described in *Table 9* grouped by their HTTP method.

| URL | Description | Parameters | Return Value |
|---|--|---|--|
| GET | | | |
| /api/articles/{id} | Returns the article with the given id. | id: unique article id | Article, see JSON object below |
| /api/articles/last | Returns the last articles the user saw. | Cookie | Array of articles |
| /api/articles?q={query} | Search for articles matching the given query. | query: search query | Array of articles matching the search query |
| POST | | | |
| /api/articles | Create initial empty new article | - | Article id |
| /api/articles/{id}/documents | Upload documents to an article | HTML Form documents: contains the documents | Array of file JSON objects like in example below |
| PUT | | | |
| /api/articles/{id} | Update the article with the given id, completely overwriting it. | id: unique article id body: article JSON object | Article |
| DELETE | | | |
| /api/articles/{id} | Deletes the article with the given id. | id: unique article id | - |
| /api/articles/{id}/documents/{filename} | Deletes the document with the given name from the article with the given id. | id: unique article id filename: name of the document | - |

Table 9: REST API description

All articles are represented like the following JSON object:

```
{
  "id": 1,
  "isTemporary": false,
  "author": {
    "name": "Max Mustermann",
    "email": "max@mustermann.com"
  },
  "lastChangedBy": {
    "name": "Martin Mustermann",
    "email": "martin@mustermann.com"
  },
  "lastChanged": 20160530,
  "title": "Knowledge",
  "text": "This is some <b>important</b> Knowledge",
  "files": [
    {
      "filetype": "pdf",
      "name": "some knowledge",
      "path": "/this/is/some knowledge.pdf"
    }
  ]
}
```

A formal description of the REST API in the RAML format can be found in the system's documentation².

² In the repository under docs/documentation/api/api.raml

4.3.2 Frontend

The frontend runs within the user's browser. It implements both the MVC pattern and SOA. *Figure 7* gives an overview of the different components of the frontend and its layers.

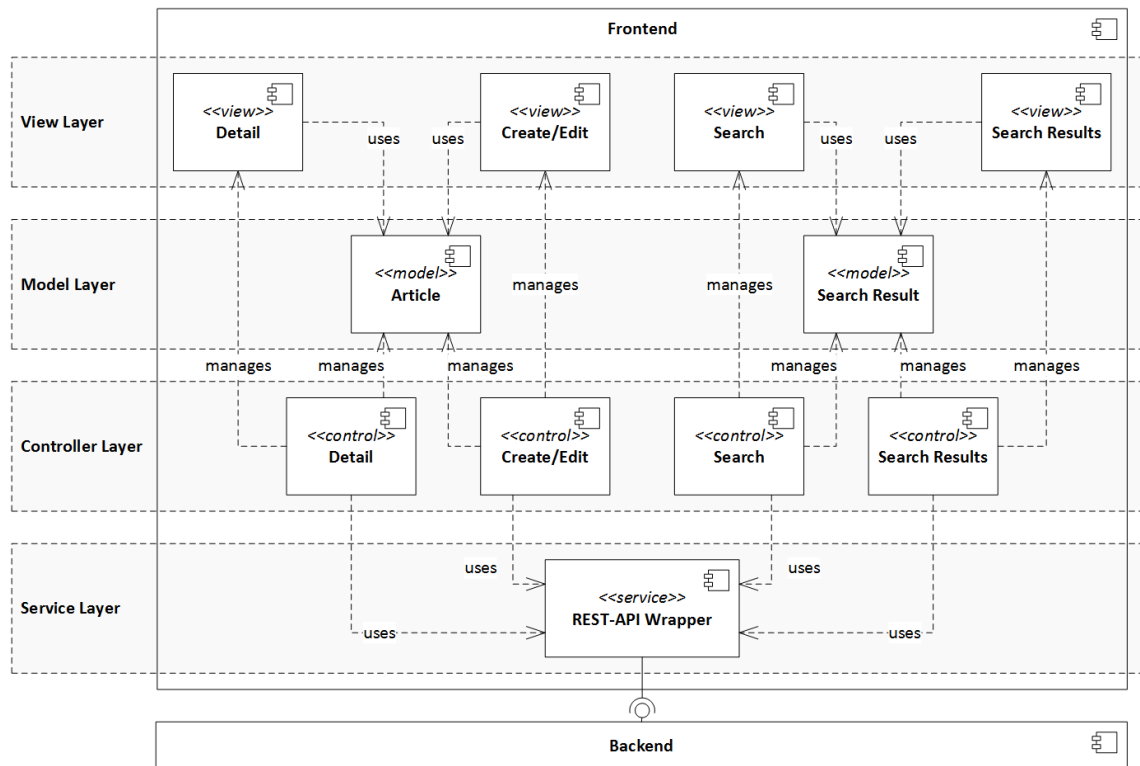


Figure 7: frontend components

4.3.2.1 Service Layer

The fundamental part of the frontend is its service layer. It connects the frontend to the backend by wrapping the backend's REST interface within a service available globally for all controllers. This way all alike calls to the backend are routed through the same method, offering the opportunity to integrate caches, interceptors and making changes in the server API transparent for all frontend components except the service layer component.

4.3.2.2 Controller Layer

Business logic of the frontend application is implemented within the controller layer. Here a controller is defined for each view handling user interactions by manipulating respective models and forwarding calls to the backend to the service layer.

4.3.2.3 Model Layer

The model layer stores the data presented and worked on in the frontend. Different models are loaded by controllers using the backend API service from the service layer. Data from

these models are presented to the user in the view layer and is manipulated by controllers in the controller layer.

4.3.2.4 View Layer

The view layer delineates the interface of the frontend application towards the user. It is rendered by the browser and has its functionality handled by its respective controller from the control layer. Information shown within a view is based upon its respective model from the model layer.

5 Runtime View

This chapter contains the runtime view of the system, represented by multiple runtime scenarios. Each section represents one runtime scenario. These are shown with UML sequence diagrams.

5.1 Create Article

The user wants to make his knowledge available to other users. He starts his browser and creates a new article in the knowledge base. *Figure 8* shows the interaction between user and knowledge base.

- First the user navigates his browser onto the knowledge base webpage from the static webserver. After it is loaded, the user clicks on the button to create a new article. The frontend requests a new article id from the dynamic webserver. The dynamic webserver creates a new article entry with temp flag on the database and returns the database unique id to the frontend as article id. The frontend loads the create article view for the user to insert text, metadata, images and documents.
- As his second action, the user writes the article using the WYSIWYG editor.
- Then he decides to use images in the article. Images are converted to base64 by the frontend in order to embed them into the article content.
- Additionally, the user uploads a document to the article. As soon as the document file is added to the article, the frontend uploads it to the dynamic webserver. The webserver then initializes the article's directory and stores the document there. Afterwards a static link for the uploaded document is generated, stored in the article's database entry and send to the frontend.
- Finally, the user clicks on the button to save the article. The frontend sends the article content and metadata (author, email) to the dynamic webserver. The dynamic webserver saves the article content in the article's directory and the metadata in the database and removes the temp flag. It also moves the article's documents from the temporary to the final directory. After the article is saved, the dynamic webserver sends the article details to the frontend. The frontend then allows to read the article (see *5.2 Read Article*).

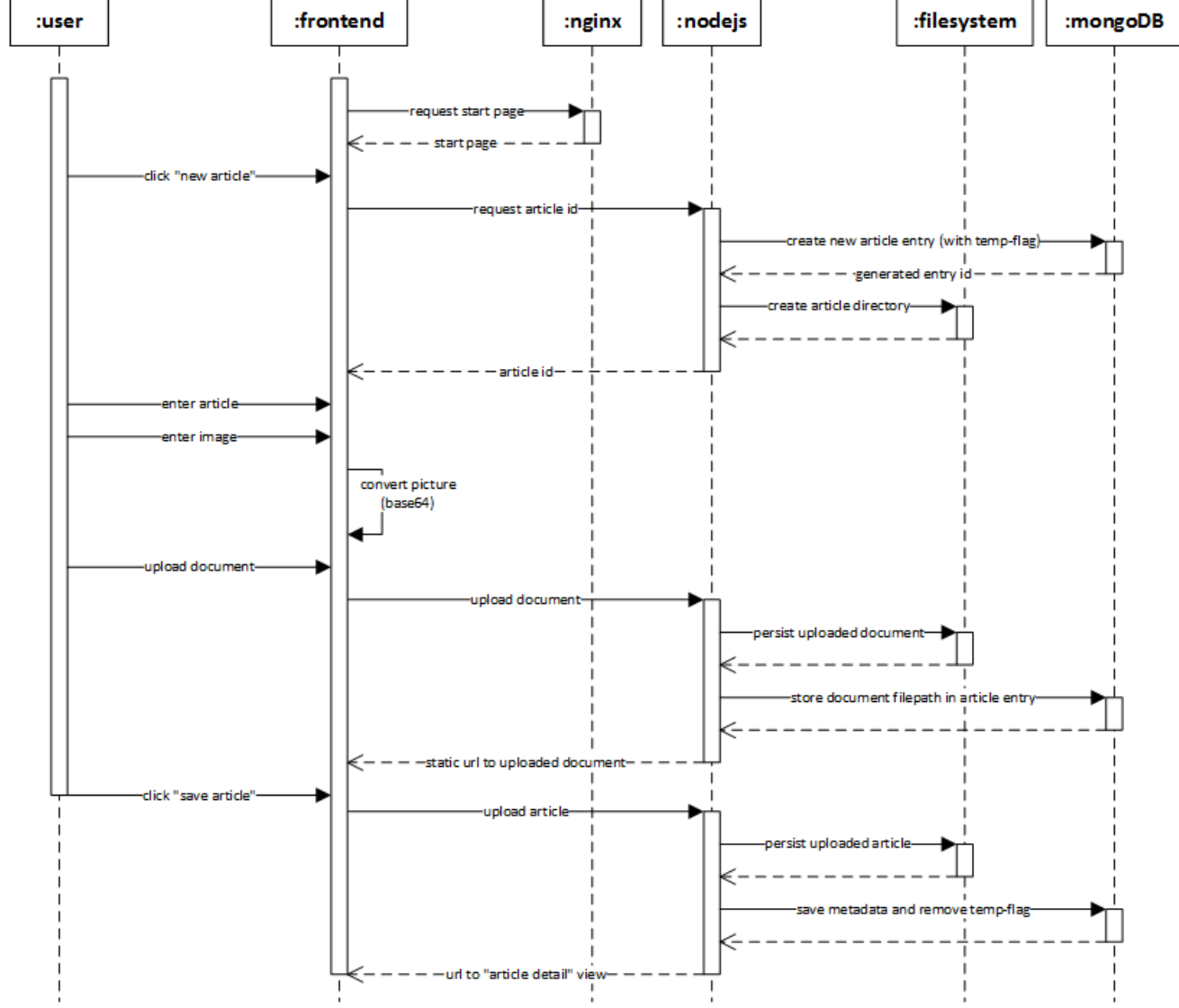


Figure 8: create article sequence diagram

5.2 Read Article

The user wants to read about a specific problem. He opens his browser and reads the according knowledge base article. *Figure 9* shows the interaction between the user and the knowledge base.

- First the user navigates his browser to the knowledge base. After it is loaded from the static webserver, the user clicks on a link to an article. The frontend then requests the article from the dynamic webserver. The dynamic webserver accesses the database to read the article's metadata. With the information in the metadata, the dynamic webserver can then load the article's content from the file system. The article is send to the frontend.
- Reading in the article the user decides to download an attached document for further information. He clicks on the document to download it. The frontend requests the document from the static webserver which reads the document from the file system and sends it to the frontend with its original name. The user can then download the document.

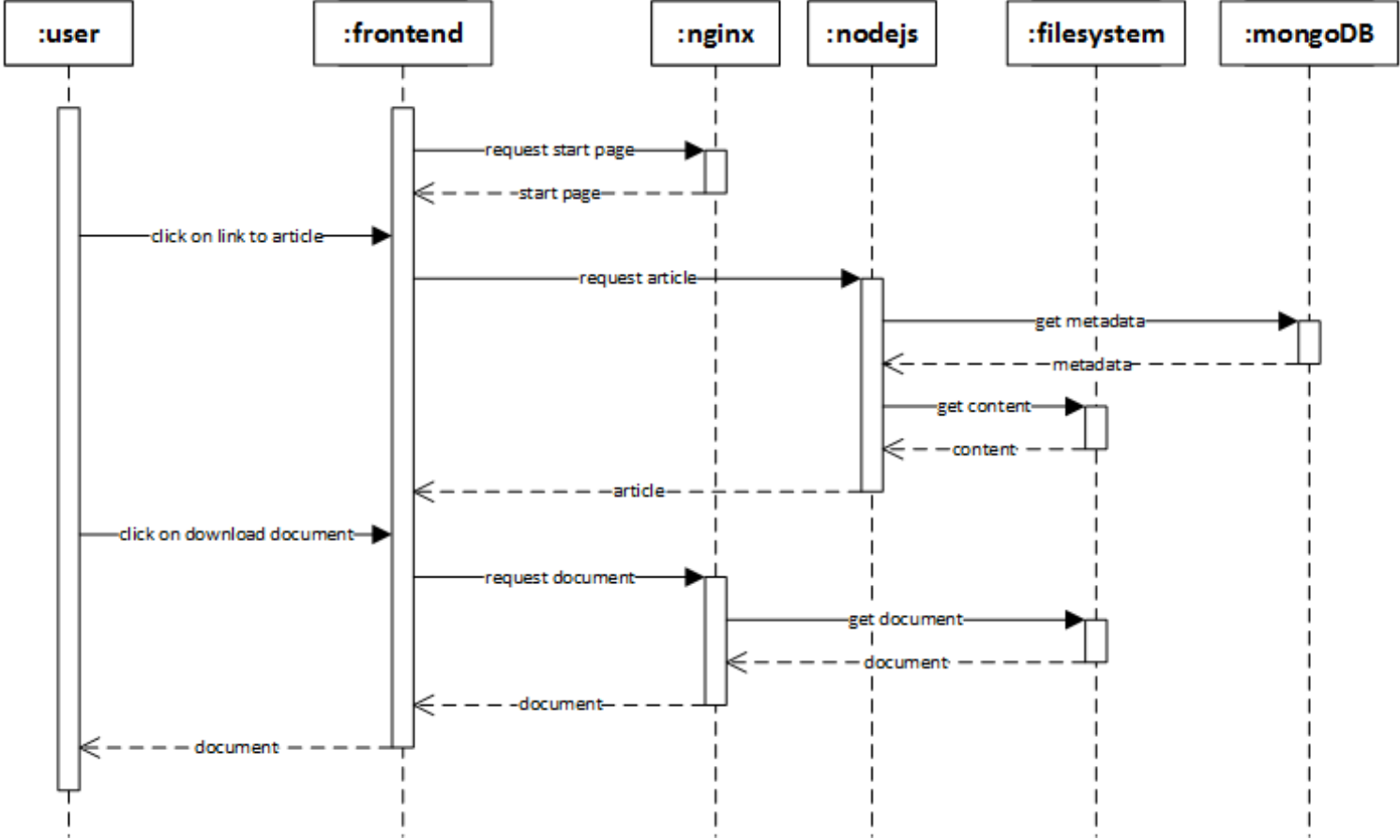


Figure 9: read article sequence diagram

5.3 Search Article

The user has a problem and needs a solution. He starts his browser and searches for solutions in the knowledge base. *Figure 10* shows the interaction between the user and the knowledge base.

- First the user navigates his browser onto the knowledge base webpage from the static webserver. After it is loaded, the user enters keywords matching his problem in the search field. The frontend sends the keywords to the dynamic webserver. The dynamic webserver then queries the search engine. With the search results the dynamic webserver accesses the database for metadata to match articles and documents together. Afterwards it returns matching articles to the frontend. The frontend displays those articles to the user.

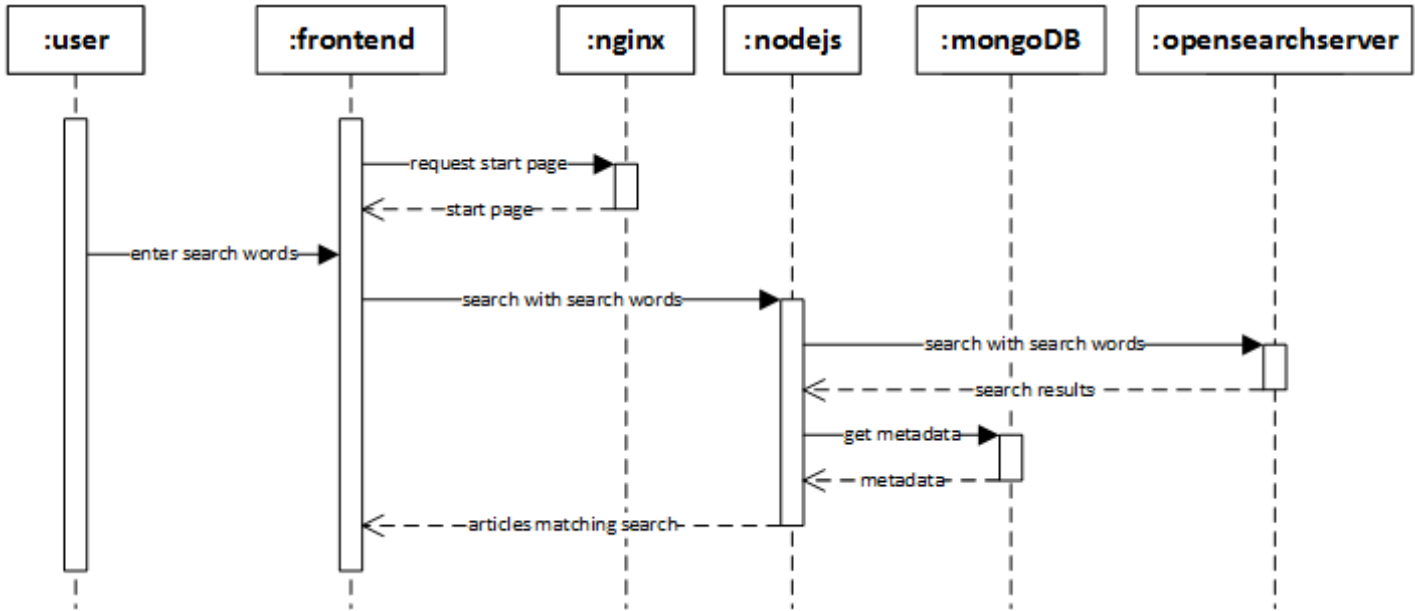


Figure 10: search article sequence diagram

6 Deployment View

The system is split into four Docker images and two Docker volumes. Each of these images contains one backend component. There are images for Nginx, Node.js, OpenSearchServer and mongoDB. The first three share one volume called file system. This volume contains the articles and the documents, as well as the frontend and backend files. The mongoDB image has its own volume containing the database. The frontend and backend communicate over the HTTPS protocol. The system is intended to run on one physical or virtual Docker host. *Figure 11* shows this deployment structure.

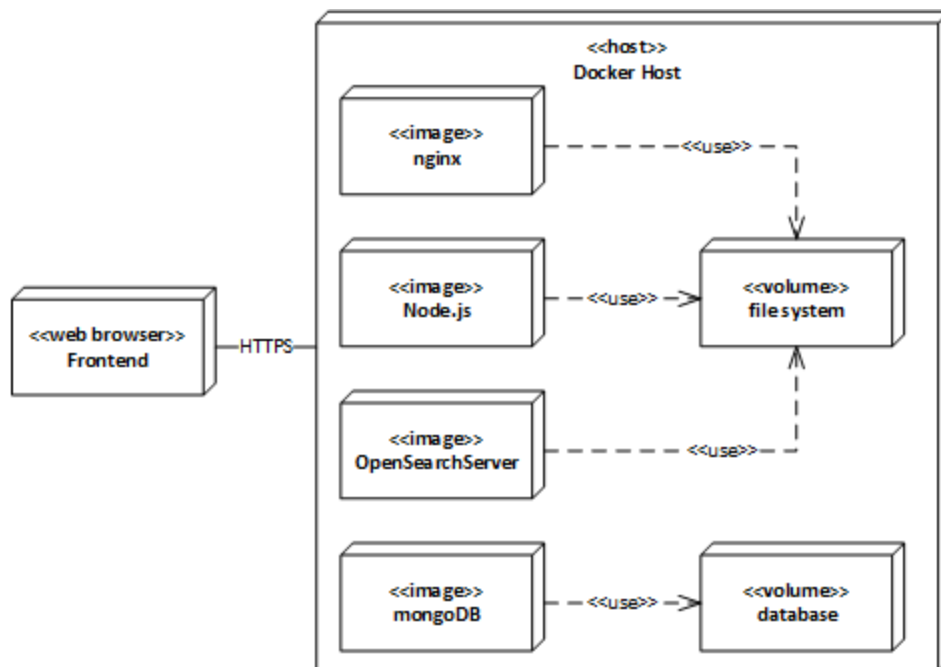


Figure 11: deployment diagram

7 Concepts

This chapter contains the important concepts of the system that are not already described in other views. Each sub chapter represents one concept.

7.1 Persistency

The knowledge base uses a hybrid approach to store articles with attached documents and their metadata. Articles and documents are stored in the file system while metadata is stored in a mongoDB.

7.1.1 File System

All articles are stored in one folder called “articles”. Each article has its own subfolder with its id as name. This subfolder holds the article and its attached documents. On the top level, next to the “articles” folder, there is a folder for the previous version of an article, called “articles_old”. Each article with an old version has its own subfolder with its id as name similar to the “articles” folder. As only the article’s content should be versioned it doesn’t contain any documents. When an article is created and files are attached during that process, they are saved into a top level folder called “articles_temp”. Each temporary article has its subfolder with a unique id there. *Figure 12* shows this structure.

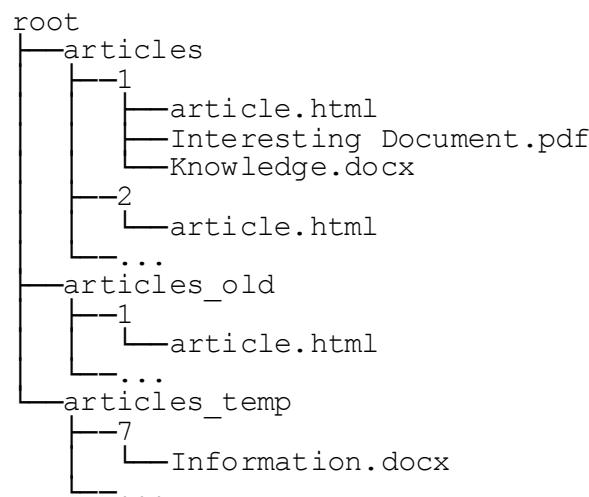


Figure 12: folder tree

7.1.2 Database

The metadata for each article is saved in one JSON document in the mongoDB. All these documents look like the following example:

```
{
  "id": 1,
  "isTemporary": false,
  "author": {
    "name": "Max Mustermann",
    "email": "max@mustermann.com"
  },
  "lastChangedBy": {
    "name": "Martin Mustermann",
    "email": "martin@mustermann.com"
  },
  "lastChanged": 20160530,
  "title": "Knowledge",
  "files": [
    {
      "filetype": "pdf",
      "name": "some knowledge",
      "path": "/this/is/some knowledge.pdf"
    }
  ]
}
```

7.1.3 Database and File System Synchronization

With a distributed persistency concept as delineated before synchronization between the different storages is mandatory. The concept has to take in mind that processes might be interrupted, detect and resolve inconsistencies. Therefore, two mechanisms are integrated into the knowledge base's persistency concept.

7.1.3.1 Temporary Flag

As soon as a user starts creating a new article in the frontend, the backend generates a new entry for it in the database. This new entry is generated with a temporary flag which indicates that the user has not saved his work yet. Files uploaded by the user respective to this new article are stored in a location different to those articles which are persistent already. This division of file storages prevents the search engine's file crawler to index files of articles which have not been persisted yet.

As soon as the user clicks the save button in the frontend, all missing information is added to the article entry in the backend and the HTML content of the WYSIWYG editor is saved. At this point the temporary flag is removed from the article entry and the article's file directory is moved to the location which is crawled by the search engine's crawler.

The file system location for non-persistent files as well as database entries with the temporary flag are deleted on a daily basis if they are older than 24 hours.

7.1.3.2 Inconsistency clean

Another scheduled script is planned to run on a daily business (preferably at night). This script compares the file links saved in the database to the content of the crawled directory of the file system. File links with missing representatives and files with missing links in the database are either deleted or moved to a maintenance directory monitored by the system's admin.

7.2 Testability

In order to fulfill architecture driver *AD-04-Testability*, the system is built in modules. Each of these modules is separately testable. Node.js and AngularJS both enable automated unit tests. These are run as part of the build process.

7.3 Data Model

The diagram in *Figure 13* illustrates the business model without relation to technology.

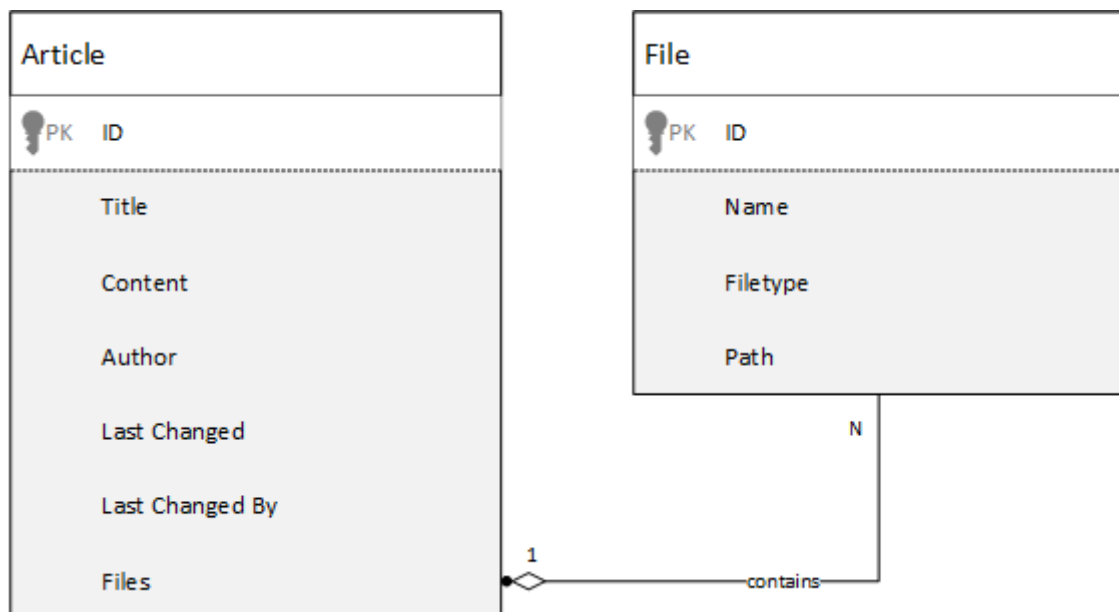


Figure 13: data model

This is the basic conceptual model which reappears in following chapters in terms of their context:

- 4.3.1.6 REST API
- 7.1 Persistency

7.3.1 Article

An article represents a basic knowledge unit of knowledge base system. It consists of the attributes shown in *Table 10*.

| Name | Description |
|-----------------|--|
| Title | A concise one line description of the knowledge unit as plain text |
| Content | A structured text with formatting such as headings, bullet lists, tables, etc. The context may also contain images right in the text flow. The content format is HTML. |
| Author | Name and E-Mail of the person who has created the article |
| Last Changed | Date of the last article change |
| Last Changed By | Name and E-Mail of the person who has made last change on the article |
| Last Changed | List of files which are attached to the article. An attachment is done by file upload of a user. |

Table 10: data model description

7.3.2 File

An article may contain attached files of various types such as PDF documents. A user can download files from an article in original format. A user can also find an article by search terms which occur in an attached file under condition that the file type is supported by search engine.

7.4 Remote API

In order to allow easy integration of the frontend and backend they communicate over a REST API. This REST API also allows to integrate other systems with the knowledge base, should this ever be needed. The REST API is described in chapter 4.3.1.6 *REST API*.

7.5 Configuration

All configuration of the backend is done in a JSON file called “config.js”. It contains the server settings for file system directories, database connection and search engine connection.

The configuration of Nginx, mongoDB and OpenSearchServer are done in their respective configuration files.

8 Design Decisions

This chapter contains all major design decisions. Each section represents one of these.

8.1 Search Engine

The main purpose of a knowledge base is to share and search information gathered throughout the execution of projects within a company. Therefore, information once stored within the knowledge base has to be searchable in a convenient, fast and configurable way. Usually articles written for the knowledge base only represent an abstract of the detailed information contained in one or more documents attached to the article. As the abstract article might not cover all frequently used buzzwords or might be missing at all for some documents, it is important to also index uploaded files so that their content is not neglected in user inquiries. For this purpose, a search engine with an integrated file parser is integrated into the knowledge base. The following implementations for search engines are evaluated. All of them are open source and do not require the acquisition of licenses. Also none do handle the actual persistence of documents handed over for indexing. Therefore, various persistency options are evaluated and described later in this document.

8.1.1 Influences

This decision is influenced by the architecture drivers *AD-01-Search*, *AD-03-Generic* and *AD-05-Persistence*.

It influences the design decision 8.3 *Backend Technology* and the component 4.2.2 *Backend*.

8.1.2 Constraints

This decision was made without any constraints.

8.1.3 Considered Alternatives

The four following alternatives were considered as search engines. The evaluation was done based on prototypes.

8.1.3.1 Apache Lucene & Tika

Apache Lucene is a full text search engine library written in Java. For evaluation the library is combine with Apache Tika, a toolkit for detection and extraction of metadata and text content from various file types (e.g. DOCX, PPTX, TXT, and PDF). Both Lucene and Tika can be used as standalone applications, but can also be embedded as JAR libraries into a Java project. In order to index a file with Lucene, it as to be abstracted to a java

object instance of `org.apache.lucene.document.Document`. The conversion from various proprietary file formats into this more abstract version is handled by Tika. The requirement of this very specific document form limits the possibilities of connecting Lucene with non-Java technologies. Also the implementation of a search engine with Lucene and Tike requires an unhandy amount of glue code, which has to be implemented, updated and tested discretely.

8.1.3.2 Apache Solr

Apache Solr is an open source platform built on Lucene and Tika. It adds new features to the search engine, abstracts away from the Java-only interface and covers the document parsing formerly done by external modules like Tika. Solr can be attached to other business logic by calling its REST service with either XML- or JSON-based data. Additionally, Solr provides an API for Java wrapping the REST service for more convenient integration. Solr can even be used to index data stored in databases, as long as there is a JDBC driver for respective database. Even though Solr decreases the implementation and testing effort compared to a bare Lucene solution, configuration effort increases a lot to make the new level of abstraction work.

8.1.3.3 OpenSearchServer

OpenSearchServer is a platform containing multiple heavy weight components used in combination with a search engine. Besides the search engine itself it contains a file parser comparable to Tika and multiple crawlers able to crawl various data sources like SAMBA drives, FTP servers, JDBC-enabled databases and web pages. OpenSearchServer is a stand-alone solution delivered either with an integrated webserver or as a WAR file to be embedded in a web container. Internally OpenSearchServer uses Lucene, just as Solr does. Besides its REST interface which can handle XML- and JSON-based data, API wrappers are available for PHP, Ruby, Perl and C#. The biggest benefit of OpenSearchServer is the set of crawlers it offers. The business logic does not have to handle explicit indexing of each uploaded document. Instead a crawler can be activated on the data storage (e.g. a dedicated directory). The crawler recognizes altered and newly added files and automatically executes the indexing. The downside of this convenience is the huge configuration effort for the crawlers and other components of OpenSearchServer.

8.1.3.4 Elastic Search

Elastic Search is another open source platform built on top of Lucene. It comes as a stand-alone server with RESTful API for JSON-based data. It focuses on massively distributed data sources and optimizes for analytics performance. Therefore, its drivers do not fit the requirements of the knowledge base project. Features like data visualization are not relevant for this project and just like bare Lucene, Elastic Search does require a parser module comparable to Tika in order to index content of files.

8.1.4 Comparison

The search engines introduced above are compared for their qualities respective to following attributes:

- **Stand-alone:** Is the search engine capable of running on its own?
- **Embeddable:** Is the search engine embeddable into another project?
- **API:** How can other components communicate with the search engine?
- **Data Format:** Which data formats are accepted by the search engine?
- **Data Source:** Which data sources does the search engine accept?
- **License:** Under what license is the search engine?

Table 11 delineates the differences between evaluated search engines towards the attributes explained above.

| | Apache Lucene & Tika | Apache Solr | OpenSearch-Server | Elastic Search |
|--------------------|-----------------------------------|---|---|-------------------------------|
| Stand-alone | Yes | Yes | Yes | Yes |
| Embeddable | Yes | Yes (not recommended) | (WAR File) | No |
| API | Native Java | REST service and service wrapper for Java | REST service and service wrapper for several languages | RESTful service |
| Data Format | Java Object | XML, JSON | XML, JSON | JSON |
| Data Source | Explicit from within Java Program | Explicit over REST service or link to JDBC-enabled database | Explicit over REST service and crawlers for JDBC, FTP, SAMBA, file system | Explicit over RESTful service |
| License | Apache License 2 | Apache License 2 | GNU GPL 3 | Apache License 2 |

Table 11: search engine comparison

After comparison of search engines towards these attributes the Apache Lucene & Tika approach is rated unsuitable for the knowledge base project. The effort which has to be put into the approach in terms of implementing and testing glue code is not justifiable.

Also the Elastic Search approach is rated unsuitable, as its drivers and focus do not match with the goals of the knowledge base project. After this first decision Apache Solr and OpenSearchServer were compared with a benchmark on how fast they could index documents. As they both build on Apache Lucene their search performance wasn't evaluated.

Table 12 shows the times Apache Solr and OpenSearchServer needed to index different document types, sizes and numbers.

| | Initial Index 45 PDFs (461 MB) | Re-index 1 PDF (520 KB) | Re-index 1 PDF (12 MB) | Re-index 1 DOCX (78 KB) |
|-------------------------|---|--|---------------------------------------|--|
| OpenSearchServer | 00:05:22 | 00:00:01 | 00:00:05 | 00:00:08 |
| Apache Solr | 00:02:35 | 00:00:00 | 00:00:02 | 00:00:01 |

Table 12: search engine benchmark

8.1.5 Decision

As Elastic Search is not able to fulfill the requirements and Apache Lucene & Tika are too much overhead in regards of programming, those two are no viable choice as search engine for the knowledge base. Apache Solr and OpenSearchServer are both viable choices. They both meet the requirements, even though OpenSearchServer is slower than Apache Solr when indexing files. The final decision was made in favor of OpenSearchServer because of its ability to crawl data sources and its better out of the box configuration. OpenSearchServer also allows easier phonetic search than Apache Solr.

8.2 Persistence

The knowledge base serves as a main sharing point for information and documents. Therefore, it has to be capable of storing many different kind of information in the following formats:

- DOC, DOX, PDF files
- HTML text
- Images
- Metadata (Author, Date, etc.)

The different parts of an article have to be delivered in the same form in which they were submitted to the knowledge base. But between uploading and search all information from these components have to be indexed in order to be considered in full text search. So it has to be accessible for both the search engine and the application itself.

8.2.1 Influences

The decision is influenced by the architecture drivers *AD-01-Search*, *AD-03-Generic* and *AD-05-Persistence*.

It influences the design decision 8.3 *Backend Technology*, the component 4.2.2 *Backend* the concept 7.1 *Persistency* and the design decision 8.7 *File Upload*.

8.2.2 Constraints

Information has to be stored in a way that the chosen search engine can directly access it in order to execute the indexing. Further the solution should be capable of providing a versioning concept for at least two versions of each article.

8.2.3 Considered Alternatives

The three following alternatives were considered as persistence options. The evaluation was done based on hands-on experience and comparison of benefits and drawbacks.

8.2.3.1 Database approach

For the database approach the technologies MongoDB and CouchDB were chosen for hands-on testing. Both databases are document-oriented, which matches the data form used in the knowledge base context. A benefit of using a database for both loose data and files is the general concurrency handling. Also all information related to an article can be stored in the same logical space. A specific benefit of using CouchDB is the versioning feature available out-of-the-box. MongoDB does not provide versioning, but is faster in up and download.

The drawbacks of the database approach are the more complex maintenance of the system and the stored data compared to persistence on file system level, as well as the limitation for search engines to index files. Evaluated search engines can only access databases which provide a JDBC driver. MongoDB supports JDBC, CouchDB does not. Further MongoDB is not capable of storing files larger than 16MB out-of-the-box. An additional framework named GridFS is needed, which splits a file into chunks referenced by a head entry in the database.

8.2.3.2 File System approach

The file system approach promises easy maintenance and porting, as everything is stored in files in a directory tree. All files can be accessed by administrators over a user interface for the file system provided by any operating system. Additionally, the architectural complexity is very low. The only way to organize the files are folders which can be composed in a tree structure. Eventually the access for the search engine to stored data is guaranteed

as all search engines evaluated for the knowledge base context are able to retrieve files from the file system.

A drawback of the file system approach is the limitation regarding versioning. Versioning on file system level often means redundancy and therefore demands a lot of storage space. Further the file system does not provide any means of concurrency handling. This problem has to be tackled by the business logic. Another task to be handled by the business logic is the mapping of data onto the directory structure on the file system as there is no query language available.

8.2.3.3 Hybrid approach

With the hybrid approach benefits of both the database and the file system are combined whilst minimizing the drawbacks of both approaches as much as possible. Here a combination of MySQL and the file system was evaluated as a hands-on experience. Files are stored in the file system while loose data is stored in a strictly structured database schema. This way large files do not harm the performance of the database, while loose data is structured and can be queried without reading and blocking files.

Unfavorable however is the fact that for this approach two systems - the database and the file system – have to be configured. Also the consistency between file links in the database and actual files in the file system has to be ensured by the business logic. Finally, the effort of reading an article is increased as both the database and the file system has to be accessed in order to retrieve all parts of an article.

8.2.4 Comparison

The persistency approaches introduced above are compared for their qualities respective to following attributes:

- **Concurrency Handling:** Does the approach handle concurrent accesses?
- **Maintenance:** How convenient is the system maintenance for an administrator?
- **Complexity:** How complex would the solution emerging from the approach be?
- **Versioning:** Does the approach handle versioning?
- **Confidence:** How much confidence was gained from the hands-on experience?

Table 13 delineates the differences between evaluated persistency approaches towards the attributes explained above.

| | Database approach | File System approach | Hybrid approach |
|-------------|-------------------|----------------------|-----------------|
| Concurrency | ++ | -- | ++ |
| Maintenance | -- | ++ | + |
| Complexity | + | + | - |
| Versioning | + | - | + |
| Confidence | - | - | + |

Table 13: persistence comparison

8.2.5 Decision

The decision was made to utilize the hybrid approach for the knowledge base project. By combining the two other approaches it is able to mitigate the most critical drawbacks whilst maintaining a manageable complexity. Further most developers in the team have experience with an approach of this sort. As technical foundation a combination of files system and MongoDB is chosen, as its data structure fits best to the meta- and loose data occurring in the context.

8.3 Backend Technology

The backend has to deliver the frontend to the user's web browser. The backend has to supply the frontend with the articles, including attached documents and images.

8.3.1 Influences

This decision is influenced by the architecture drivers *AD-04-Testability* and *AD-05-Persistence*, as well as by the design decisions *8.1 Search Engine* and *8.2 Persistence*.

It influences the component *4.2.2 Backend*.

8.3.2 Constraints

All developers are capable of the two programming languages Java and JavaScript.

8.3.3 Considered Alternatives

The two following alternatives were considered as backend technologies.

8.3.3.1 Node.js with Nginx

Node.js is a lightweight and performant JavaScript server. It allows easy file handling and simple REST calls. It's extremely easy to integrate with MongoDB. As a downside, pure JavaScript does not provide type safety.

Nginx is a fast and lightweight webserver well suited for static content delivery. It's a stable, mature server technology that has proven itself in the web for several years.

The combination of those two technologies allows fast and stable delivery of static content with Nginx, while Node.js handles dynamic content.

8.3.3.2 Spring Web MVC

Spring Web MVC is an industry proven Java Framework for dynamic web content. It's well documented and wide spread. On the downside, Java is relatively heavyweight and memory hungry. Doing REST calls with Java requires additional effort.

8.3.4 Decision

Both technologies are capable to fulfill the requirements. The decision was made in favor of Node.js with Nginx. They provide easier integration with REST and the team competences favored Node.js.

8.4 Frontend Technology

The knowledge base has to be usable for all members of the customer's project teams. Sharing and searching information is meant to be as fast and convenient as possible. Therefore, the frontend—the part of the knowledge base users can actually see and interact with—has to guarantee high standards towards usability.

8.4.1 Influences

The decision is influenced by the architecture drivers *AD-01-Search*, *AD-02-Usability* and *AD-03-Generic*.

The decision influences the design decision 8.6 *WYSIWYG Editor* and the component 4.2.1 *Frontend*.

8.4.2 Constraints

The frontend technology chosen has to be runnable on the following browsers:

- Mozilla Firefox Version 46.0.1
- Google Chrome Version 51.0.2704.63

Additionally, responsiveness and compliance to mobile device screen sizes would be favorable.

8.4.3 Decision

As core technology for the frontend AngularJS was chosen. AngularJS is a popular framework for creating dynamic web applications. It is the frontend technology best understood amongst Kanbanana team members. Further it is maintained by a reliable source – Google (Alphabet) – and has a better performance than its competitors EmberJS and BackboneJS.

8.5 Folder Structure

The file system has to store all articles and documents. The articles have to be stored in a versioned manner, allowing read only access, without the ability to directly restore the article, back to one version before. Documents do not have to be versioned.

8.5.1 Influences

This decision is influenced by the architecture driver *AD-05-Persistence* and the design decisions *8.2 Persistence*.

It influences the component *4.2.2 Backend* and the concept *7.1 Persistency*.

8.5.2 Constraints

This decision was made without any constraints.

8.5.3 Considered Alternatives

The following three alternatives have been considered.

8.5.3.1 One Folder for Everything

All articles and documents are saved to a single folder. Their name is replaced with a UUID. The database has to have the metadata as to which file had what name and which files belong together. *Figure 14* shows this structure.

```
files
├── 1.html
├── 2.pdf
├── 3.docx
├── 4.html
└── ...
```

Figure 14: one folder structure

This approach has the problem of heavy reliance on the database and low maintainability in the file system as no mapping of files can be done without the database. This also

means that the dynamic webserver has to deliver all documents to the frontend, which is undesired.

8.5.3.2 One Folder per Article with Nested Version Folder

All articles are stored in one folder called “articles”. Each article has its own subfolder with its id as name. This subfolder contains the article and its attached documents. Every article’s folder has a subfolder called “old”. The previous version of the article is saved in there. *Figure 15* shows this structure.

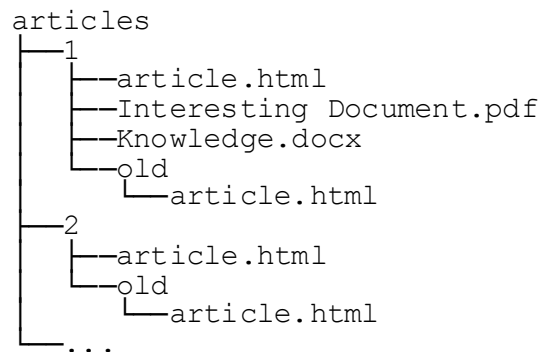


Figure 15: nested version article folders

This approach allows to map an article and its documents directly together. The “old” subfolder keeps all files belonging to an article together. The downside of this approach is that it would need additional configuration for the search engine’s file system crawler, should it be used, as the old versions should not be indexed. It also makes it possible for the static webserver to deliver the documents without any further mapping.

8.5.3.3 One Folder per Article with External Version Folder

All articles are stored in one folder called “articles”. Each article has its own subfolder with its id as name. This subfolder contains the article and its attached documents. On the top level, next to the “articles” folder, there is a folder for the previous version of an article, called “old articles”. Each article with an old version has his own subfolder with its id as name similar to the “articles” folder. As only the articles content should be versioned it doesn’t contain any documents. *Figure 16* shows this structure.

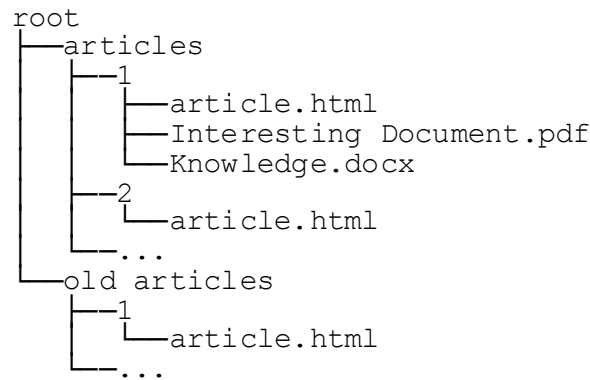


Figure 16: external versioned article folders

The downside of this approach is that the current and old article files are not directly together. They require an additional mapping step via their id folders. This approach allows using the search engine's crawler directly on the articles folder. Additionally all documents can be delivered directly from the static webserver without any further mapping.

8.5.4 Decision

The decision was made in favor of one folder per article with external version folder because it allows for the articles to be crawled by the search engine without indexing old articles. It also has the major benefit of taking load of the dynamic webserver, because the static webserver can directly deliver the documents to the frontend.

8.6 WYSIWYG Editor

The user has to write articles in the frontend. The user has to be able to format his text with headings, italic and bold writing. There have to be bullet points and tables. The editor has to be able to cope with pictures.

8.6.1 Influences

This design decision is influenced by the architecture driver *AD-02-Usability* and the design decisions *8.4 Frontend Technology*.

It influences the component *4.2.1 Frontend*.

8.6.2 Constraints

The WYSIWYG editor has to be able to be integrated into AngularJS.

8.6.3 Considered Alternatives

The four following alternatives were considered.

8.6.3.1 TinyMCE

TinyMCE is a feature rich WYSIWYG editor that allows to use text formatting, bullet points, tables and pictures. It is usable free of charge. It can be extended with plugins.

8.6.3.2 Froala Editor

Froala Editor is a feature rich WYSIWYG editor that allows to use text formatting, bullet points, tables and pictures. It is only usable after purchasing a license.

8.6.3.3 Angular WYSIWYG

Angular WYSIWYG is a basic open-source WYSIWYG editor that allows to use text formatting, bullet points and pictures. Tables are not supported. It is usable free of charge.

8.6.3.4 textAngular

textAngular is a basic open-source WYSIWYG editor that allows to use text formatting, bullet points and pictures. Tables are not supported. It is usable free of charge.

8.6.4 Decision

The decision was made in favor of TinyMCE as it has all the needed features and is usable free of charge. The ability to extend it with plugins opens it up for further extensions like the ability to link a document directly in the text.

8.7 File Upload

The documents getting attached to articles have to be saved in the system. They have to be uploaded at one point. The handling of those documents and the according article are subject of this decision.

8.7.1 Influences

This decision is influenced by the architecture drivers *AD-02-Usability*, *AD-03-Generic*, *AD-05-Persistence* and the design decision *8.2 Persistence*.

It influences the components *4.2.1 Frontend* and *4.2.2 Backend*.

8.7.2 Constraints

This decision was made without any constraints.

8.7.3 Considered Alternatives

The following two alternatives were considered.

8.7.3.1 Upload on Article Save

The user can select the documents to attach to his article while writing it. They get buffered in the browser and uploaded once he saves the article. This means no backend interaction is needed before the user decides to save the article. It reduces the complexity in the backend and obsoletes file system and database cleanup should an article not be saved. However this also means that if the user selects many or large files, the saving of the article might take quite long as everything gets uploaded at the end.

8.7.3.2 Instant Upload

The user can select the documents to attach to his article while writing it. They are uploaded immediately to the backend. This means the backend has to be aware of the temporary, unsaved article and has to manage its files. Should the user decide to not save the article, the files and the database entry for the article have to be deleted at some point. There is no certainty in knowing when the user cancels the article creation as he can simply close his browser. This means there is a higher complexity in the backend. On the upside the saving of an article is quicker as all files already have been send to the backend.

8.7.4 Decision

The decision was made in favor of instant upload even though it creates a higher complexity in the backend. The long article save operation is considered counterproductive towards the usability of the system. Many other major systems like Google Mail or Dropbox also instantly upload files when they are selected. This sets the expectancy of users for this to happen as the intuitive way.

9 Quality Scenarios

The following illustration gives an overview of the relevant quality attributes and their associated scenarios.

9.1 Quality Tree

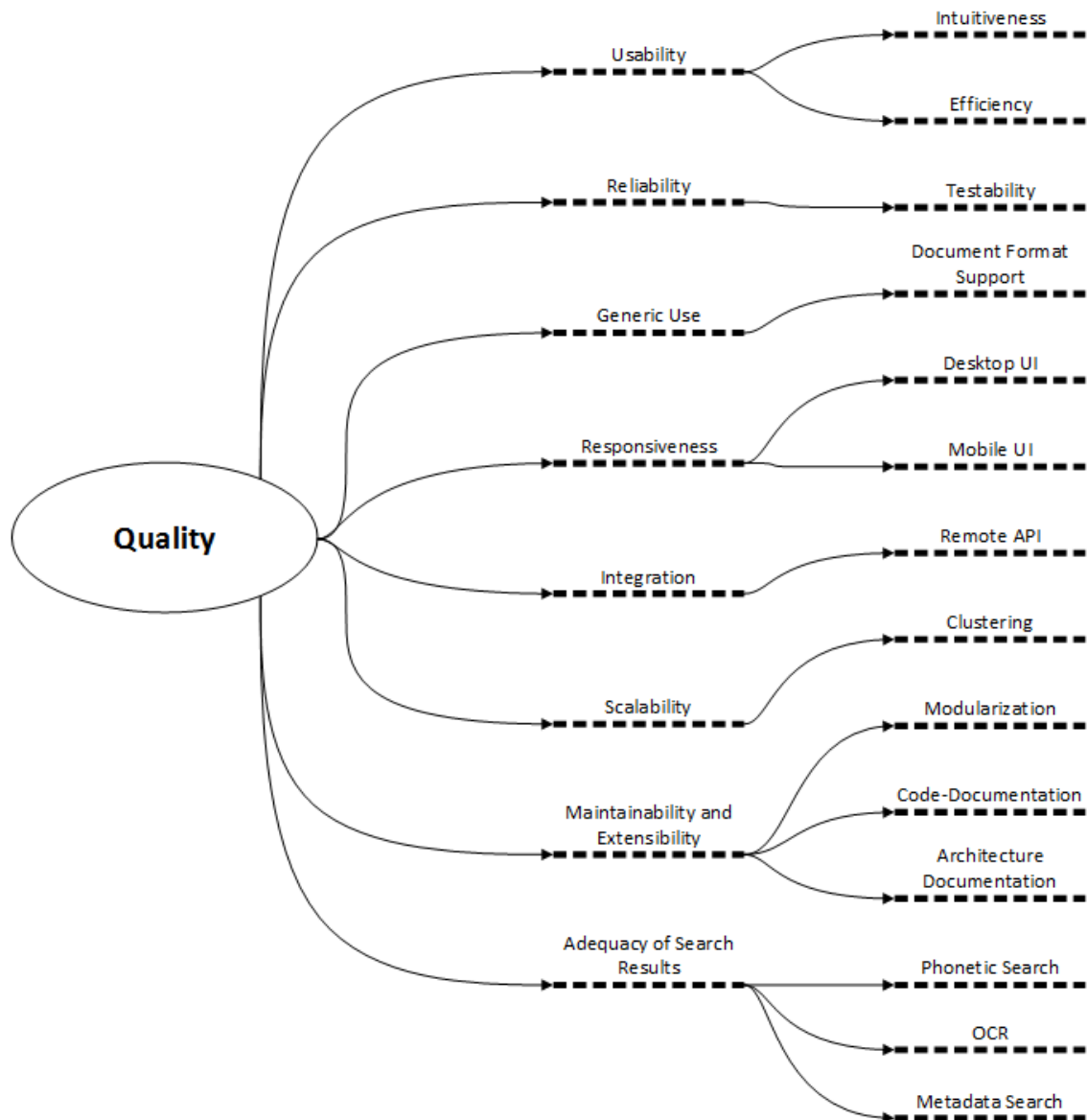


Figure 17: quality tree

9.2 Scenarios

Each quality scenario found in *Figure 17* is described in the corresponding section.

9.2.1 Intuitiveness

This quality scenario is associated with the architecture driver *AD-02-Usability*.

Environment: The System is up and running.

Stimulus: A user interacts with the system.

Source of Stimulus: New user who has never used the system before and wants to get used to the system.

Affected System Artefacts: *4.2.1 Frontend*

System response: The system can be easily understood by user and used intuitively.

Response measure: A new user can create an article within 15 minutes.

9.2.2 Efficiency

Environment: The System is up and running.

Stimulus: A user interacts with the system.

Source of Stimulus: Average user who wants to create a new article.

Affected System Artefacts: *4.2.1 Frontend*

System response: The system makes it for user fast and easy to create a new article.

Response measure: An average user can create a new article within 5 minutes.

9.2.3 Testability

This quality scenario is associated with the architecture driver *AD-04-Testability*.

Environment: The System is implemented.

Stimulus: A developer/tester launches automatic tests by using a Build-Management Tool script.

Source of Stimulus: Developer/Tester who wants to test the correctness of the system implementation.

Affected System Artefacts: The whole System.

System response: The Build-Management-Tool script executes all automatic tests and generates a report about the test results. The test report contains the total achieved branch coverage and the test execution results regarding whether a test finished successfully as expected, or with a failure.

Response measure: Generated test reports display the achieved branch coverage of automatic tests which must be at least 85%.

9.2.4 Document Format Support

This quality scenario is associated with the architecture driver *AD-03-Generic*.

Environment: A user is creating an article.

Stimulus: A user uploads a document file in an arbitrary format.

Source of Stimulus: A user who wants to attach document files to an article.

Affected System Artefacts: *4.2.2.1 Dynamic Webserver, 4.2.2.3 Search Engine*

System response: The system accepts the uploaded document file independently of its format and size, links it to the respective article, and tries to index it in order to make its content available for the full text search.

Response measure: The full text search considers the contents of PDF, DOC, DOCX and TXT document formats including PDF files which need OCR.

9.2.5 Desktop UI

Environment: The System is up and running.

Stimulus: A user interacts with the system using a desktop computer.

Source of Stimulus: Desktop user.

Affected System Artefacts: Frontend.

System response: The system renders a Desktop optimized version of the user interface which does not limit the functional scope of the system in any manner.

Response measure: The User Interface fits well in the screen resolution of 1600x800 pixels by using Chrome or Mozilla Firefox Web-Browsers in their latest versions.

9.2.6 Mobile UI

This quality scenario is associated with the architecture driver *AD-06-Mobile*.

Environment: The System is up and running.

Stimulus: A user interacts with the system using a mobile device.

Source of Stimulus: Mobile user.

Affected System Artefacts: *4.2.1 Frontend*

System response: The system renders an optimized version of the user interface for mobile devices. The mobile user interface allows searching and reading of articles in a reasonable manner.

Response measure: The User Interface fits well in the screen of latest iPhones and is supported by its standard browser.

9.2.7 Remote API

This quality scenario is associated with the architecture driver *AD-08-Integration*.

Environment: The System is implemented.

Stimulus: Integration of the system with other systems is planned.

Source of Stimulus: Software Architects and Developers who design and implement the integration concept.

Affected System Artefacts: *4.2.2.1 Dynamic Webserver*

System response: The system provides a well-documented REST API for article management and article search.

Response measure: Other systems can communicate with the system using the same interface as the frontend component.

9.2.8 Clustering

This quality scenario is associated with the architecture driver *AD-07-Scalability*.

Environment: The System is implemented.

Stimulus: Demand for scaling system horizontally.

Source of Stimulus: Administrator who wants to deploy the system.

Affected System Artefacts: The whole System.

System response: Each level 2 backend component is based on a technology with clustering support. The instructions for clustering can be obtained from the official documentation of technologies.

Response measure: Presence of documentation/guidance with clustering instructions for backend level 3 technologies in the internet.

9.2.9 Modularization

Environment: The System is implemented.

Stimulus: Need for Separation of Concerns.

Source of Stimulus: Software Architects/Developers/Testers.

Affected System Artefacts: The whole System.

System response: The system is modular. The modules are simple and well-documented. Therefore, a developer team can understand the system easily and divide the workload in different units in order to achieve separation of concerns.

Response measure: Architecture compliance check of building block view level 2 and 3.

9.2.10 Code-Documentation

Environment: The System is implemented.

Stimulus: A development team member needs to understand the code of the system.

Source of Stimulus: Software Architects/Developers/Testers.

Affected System Artefacts: *4.2.1 Frontend, 4.2.2.1 Dynamic Webserver*

System response: The code of the system is documented in a manner similar to Java-Doc.

Response measure: All signatures of JavaScript functions are documented.

9.2.11 Architecture Documentation

Environment: The System is implemented.

Stimulus: Development team member needs to understand the system.

Source of Stimulus: Stakeholders of architecture documentation (chapter 1.3).

Affected System Artefacts: Architecture Documentation

System response: Architecture Documentation based on arc64.

Response measure: Presence of arc64 conform Architecture Documentation document.

9.2.12 Phonetic Search

Environment: The System is up and running.

Stimulus: User uses a full text search.

Source of Stimulus: User who wants to find an article corresponding to his entered search term.

Affected System Artefacts: *4.2.2.3 Search Engine*

System response: The system performs phonetic search based on the entered search term and lists the corresponding articles.

Response measure: The search result contains also articles with synonyms of English words entered as search term.

9.2.13 OCR

Environment: The System is up and running.

Stimulus: User uploads a document file with a content which requires OCR.

Source of Stimulus: A user uploads a document file in arbitrary format.

Affected System Artefacts: *4.2.2.3 Search Engine*

System response: The Search-Engine parses the document file by using OCR, indexes its content and is able to consider the content for further full text searches.

Response measure: An uploaded PDF document file with OCR content can be found in terms of its OCR content by using the full text search.

10 Technical Risks

The following risks have been identified at the beginning of the project.

10.1 Risk: High Effort for Search-Engine Configuration

OpenSearchServer is a complete and standalone search engine solution which is used in the context of this project. The features of OpenSearchServer mentioned on the official OpenSearchServer website³ sound very promising. The risky part is the development of a suitable OpenSearchServer configuration for the knowledge base system in compliance with the system requirements. The configuration of OpenSearchServer is very voluminous. On the one hand the OpenSearchServer includes a runnable configuration out of the box. On the other hand the included configuration needs to be validated, verified, and customized if necessary in order to ensure the fulfillment of system requirements.

10.1.1 Risk mitigation

10.1.1.1 Early Start of configuration task

Early start of OpenSearchServer configuration is required in order to be able to precise the effort estimation for this task and assign development resources accordingly.

10.1.2 Contingency Planning

10.1.2.1 Assignment of many development resources

Assign as much development resources as needed to complete the extensive configuration in time.

10.2 Risk: High Memory Consumption of Search-Engine

The runnable prototype of OpenSearchServer showed that an OpenSearchServer instance consumes over 4 GB of main memory in initial state. It is not possible to gain concrete hardware constraints from the customer, even after explicit questioning. According to the customer the knowledge base system should use as less hardware resources as possible. Thus, there is a risk that the customer may not be satisfied by resource consumption of the final knowledge base system which rely on OpenSearchServer.

³ OpenSearchServer official website: <http://www.opensearchserver.com/>

10.2.1 Risk mitigation

10.2.1.1 Further Prototyping

Further research is required in order to discover the configuration options which instruct OpenSearchServer to reduce its main memory consumption.

10.2.1.2 Early customer feedback

Inform the customer about the estimated main memory consumption as soon as a precise estimation could be made, explain the benefits of OpenSearchServer, and request an agreement for fulfilment of hardware constraints respecting the estimated main memory consumption.

10.2.1.3 Exchange support of Search-Engine in architecture

The architecture should envisage a reasonable solution for an exchange of OpenSearchServer to another search engine technology which consumes less main memory.

10.2.2 Contingency Planning

10.2.2.1 Exchange of OpenSearchServer to Solr

With the support of architecture the search engine can be changed to Apache Solr. The prototypes of Solr have shown that it consumes initially much less main memory than OpenSearchServer. However, in contrast to OpenSearchServer the configuration effort for Solr is much higher than for OpenSearchServer. Therefore, it's unlikely that this change can be achieved in the context of the first system evolution cycle. Thus, the customer should be informed about this option in order to be able to achieve the desired main memory consumption in a future evolution by himself.

11 Glossary

Article An article includes structured text as its content and attached documents.

Document A document is an external file, like text, PDF or Word.

Dynamic Webserver A dynamic webserver, in contrast to a static webserver, only delivers dynamic content built from different sources like for example a database.

Static Webserver A static webserver, in contrast to a dynamic webserver, only delivers static content, like HTML pages, pictures or JavaScript files.

List of Figures

| | |
|--|----|
| Figure 1: key functions of the knowledge base platform | 5 |
| Figure 2: context diagram..... | 15 |
| Figure 3: building block view level 1 | 16 |
| Figure 4: building block view level 2 | 18 |
| Figure 5: components with technologies | 21 |
| Figure 6: dynamic webserver components | 23 |
| Figure 7: frontend components | 26 |
| Figure 8: create article sequence diagram | 29 |
| Figure 9: read article sequence diagram | 31 |
| Figure 10: search article sequence diagram | 33 |
| Figure 11: deployment diagram | 34 |
| Figure 12: folder tree..... | 35 |
| Figure 13: data model..... | 37 |
| Figure 14: one folder structure | 47 |
| Figure 15: nested version article folders | 48 |
| Figure 16: external versioned article folders | 49 |
| Figure 17: quality tree | 52 |

List of Tables

| | |
|--|----|
| Table 1: AD-01-Search | 7 |
| Table 2: AD-02-Usability | 8 |
| Table 3: AD-03- Generic | 8 |
| Table 4: AD-04-Testability | 9 |
| Table 5: AD-05-Persistence | 9 |
| Table 6: AD-06-Mobile | 10 |
| Table 7: AD-07-Scalability | 10 |
| Table 8: AD-08-Integration | 11 |
| Table 9: REST API description | 24 |
| Table 10: data model description | 38 |
| Table 11: search engine comparison | 41 |
| Table 12: search engine benchmark | 42 |
| Table 13: persistence comparison | 45 |

List of Abbreviations

| | |
|---------|---------------------------------------|
| AJAX | Asynchronous JavaScript and XML |
| API | Application Programming Interface |
| DBMS | Database Management System |
| CRUD | Create, Read, Update, Delete |
| CSS | Cascading Style Sheets |
| FTP | File Transfer Protocol |
| GUI | Graphical User Interface |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | HTTP Secure |
| ID | Identifier |
| JAR | Java Archive |
| JDBC | Java Database Connectivity |
| JSON | JavaScript Object Notation |
| LDAP | Lightweight Directory Access Protocol |
| MVC | Model View Controller |
| NoSQL | Not Only SQL |
| OCR | Optical Character Recognition |
| PDF | Portable Document Format |
| RAML | RESTful API Modeling Language |
| REST | Representational State Transfer |
| SOA | Service-oriented Architecture |
| SQL | Structured Query Language |
| UML | Unified Modeling Language |
| URL | Uniform Resource Locator |
| UUID | Universally Unique Identifier |
| WAR | Web Archive |
| WYSIWYG | What You See Is What You Get |
| XML | Extensible Markup Language |

Version History

| Version | Date | Description |
|---------|------|-------------|
| 1.0 | | |