|  |  |
| --- | --- |
|  | **kan** |

**Architecture Documentation  
Knownana**

**Authors**Vladislav Chumak  
Alexander Schramm  
Jochen Schwander

# Table of Contents

Table of Contents 2

List of Figures 4

List of Tables 4

List of Abbreviations 5

Version History 6

Preface 7

1 Introduction and Goals 8

2 Architecture Constraints 11

3 System Scope and Context 12

4 Solution Strategy 13

5 Building Block View 14

5.1 Level 1 14

5.2 Level 2 14

6 Runtime View 15

6.1 Create Article 15

6.2 Read Article 17

6.3 Search Article 18

7 Deployment View 20

8 Concepts 21

9 Design Decisions 22

9.1 Search Engine 22

9.1.1 Influencing Factors 22

9.1.2 Constraints 22

9.1.3 Considered Alternatives 22

9.1.4 Comparison 24

9.1.5 Decision 25

9.2 Persistence 25

9.3 Backend Technology 25

9.3.1 Influencing Factors 25

9.3.2 Constraints 25

9.3.3 Considered Alternatives 26

9.3.4 Decision 26

9.4 Frontend 26

9.5 Deployment 26

10 Quality Scenarios 27

11 Technical Risks 28

12 Glossary 29

# List of Figures

Figure 1: context diagram 12

Figure 2: level 1 building block view 14

Figure 3: create article sequence diagram 16

Figure 4: read article sequence diagram 17

Figure 5: search article sequence diagram 19

# List of Tables

Table 1: AD-01-Search 8

Table 2: AD-02-Usability 8

Table 3: AD-03-Generic 9

Table 4: AD-04-Testability 9

Table 5: AD-05-Persistence 10

Table 6: search engine comparison 24

Table 7: search engine benchmark 25

# List of Abbreviations

API Application Programming Interface

FTP File Transfer Protocol

HTML HyperText Markup Language

JAR Java Archive

JDBC Java Database Connectivity

JSON JavaScript Object Notation

MVC Model View Controller

PDF Portable Document Format

REST Representational State Transfer

UML Unified Modeling Language

URL Uniform Resource Locator

WAR Web Archive

XML Extensible Markup Language

# Version History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 0.1 | 24.05.2016 | Arc42 structure & architecture drivers & backend technology design decision |
| 0.2 |  |  |

# Preface

TODO arc42

# Introduction and Goals

TODO Drivers

|  |  |  |
| --- | --- | --- |
| **Categorization** | | |
| **Driver Name** | Full text search | |
| **Driver ID** | AD-01-Search | |
| **Status** | Under Design | |
| **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

# Architecture Constraints

TODO hardware constraints?

# System Scope and Context

Figure 1: 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. The system does not interact with any other system.

# Solution Strategy

TODO Client Server

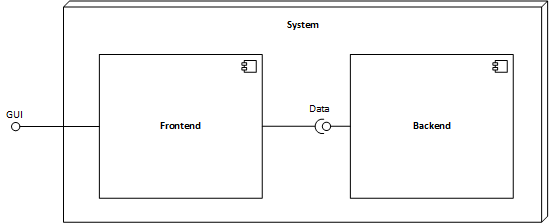
# Building Block View

TODO

## Level 1

TODO

Figure 2: level 1 building block view



## Level 2

TODO

# Runtime View

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

## 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 3* 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 loads the new article page from the static webserver and offers the user the possibility to enter a formatted article with pictures and attached documents.

As his second action, the user writes the article and decides to use pictures in it, as well as to attach a document. The frontend forwards the picture and document to the dynamic webserver. The dynamic webserver saves the document and picture into the file system. It also saves metadata about which document and picture belong to the article in the database. When the picture upload is done, the dynamic webserver returns an URL pointing to the picture to the frontend. After this, the frontend loads the picture from the static webserver.

The user thirdly clicks on the button to save the article after he is content with it. The frontend sends the article to the dynamic webserver. The dynamic webserver saves the article on the file system and updates the metadata in the database. After the article is saved, the dynamic webserver sends an URL to the article back to the frontend. The frontend then allows to read the article (see *6.2 Read Article*).

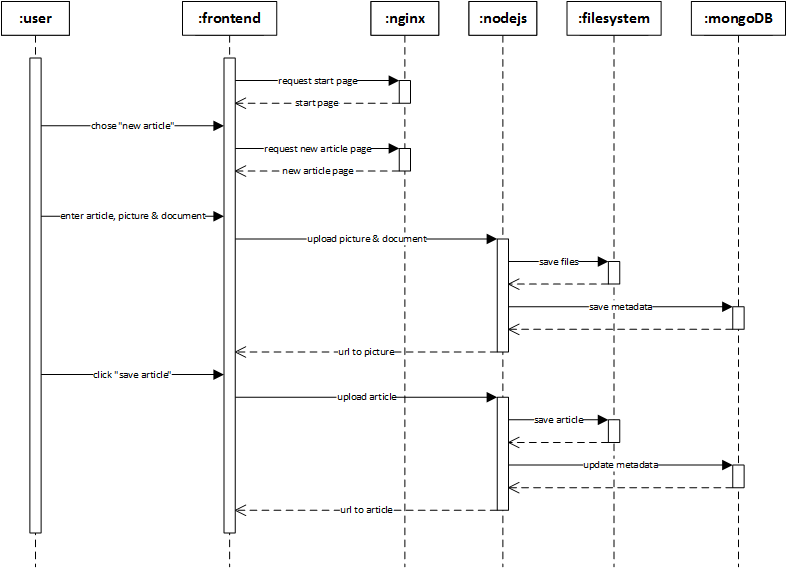


Figure 3: create article sequence diagram

## Read Article

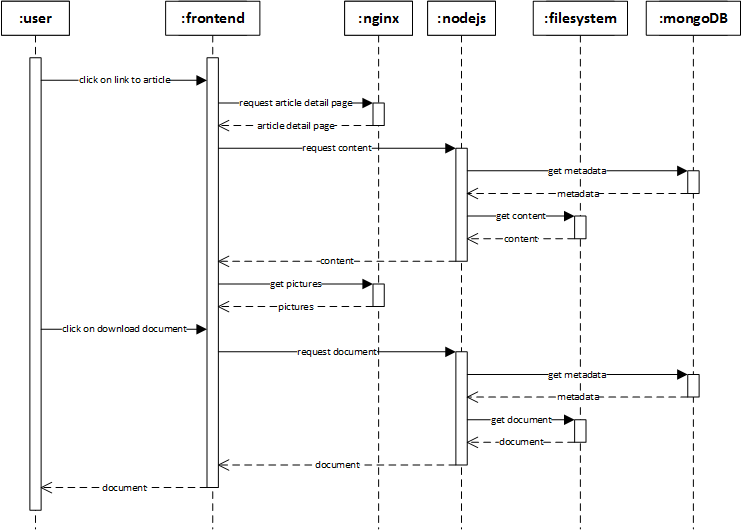
The user wants to read about a specific problem. He opens his browser and reads the according knowledge base article. *Figure 4* shows the interaction between the user and the knowledge base.

Figure 4: read article sequence diagram

First the user navigates his browser to the article page. After it is loaded from the static webserver, the frontend requests the article’s content 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 content is send to the frontend. The frontend displays it, loading pictures from the static webserver.

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 form the dynamic sever which reads the article’s metadata from the database. With the metadata the dynamic webserver reads the document from the file system and sends it to the frontend with its original name. The user can then download the document.

## Search Article

The user has a problem and needs a solution. He starts his browser and searches for solutions the in the knowledge base. *Figure 5* 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 5: search article sequence diagram

# Deployment View

TODO

# Concepts

TODO

# Design Decisions

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

## 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.

### Influencing Factors

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

### Constraints

This decision was done without any constraints.

### Considered Alternatives

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

#### Apache Lucene & Tika

Apache Lucene is a 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.

#### 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.

#### 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 OpenSearch-Server 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.

#### 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.

### 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 6* 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 6: 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 7* 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 7: search engine benchmark

### 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 OpenSearch-Server because of its ability to crawl data sources.

## Persistence

TODO fs, hybrid, db

TODO mongo

## Backend Technology

The backend has to provide the users web browser with the frontend. The backend has to supply the frontend with the articles, including attached documents and images.

### Influencing Factors

This decision is influenced by the architecture drivers *AD-04-Testability* and *AD-05-Persistence*, as well as by the design decisions *9.1 Search Engine* and *9.2 Persistence*.

### Constraints

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

### Considered Alternatives

The two following alternatives were considered as backend technologies.

#### Node.js with Nginx

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

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 for fast and stable delivery of static content with Nginx, while Node.js handles dynamic content.

#### 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.

### Decision

Both technologies are capable to fulfill the requirements. The decision was made for Node.js with Nginx because they provide easier integration with REST and the team competences favored Node.js.

## Frontend

TODO angular

## Deployment

TODO docker?

# Quality Scenarios

Todo?

# Technical Risks

TODO

# 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 build 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.