Seeker

Project Documentation

Jaroslaw Hirniak

Last updated on: *January 10, 2015*

Contents

Chapter 1

Project

1.1 Motivation

The World Wide Web is rich source of information and many excellent search engines exist that help to find interesting information in the matters of seconds. However, they only provide the link to the source of information and if user wants to compare data from many sources or gather it for analysis is left with daunting task of collecting all this information by hand.

This is repetitive and daunting task, something that humans do not like, but computers love. It would be much better to let computer prepare data for the analysis for human and let human do the creative work.

The goal of this project is to research method and develop application to generate aggregate documents composed of sections of interest extracted from many sources (e.g. PDF, DOC, HTML, etc.) with proper annotations (about sources and their version) and provide those documents for analysis by humans.

1.2 Proof of concept

The project idea originated as a solution to a problem faced by the researches in minority languages at the University of Edinburgh, who faced issue of gathering data from European Charter for Regional or Minority Languages website for comparative analysis and producing various reports based on that data.

Therefore, the proof of concept should solve the issue faced by the researches in minority languages, and provide them with a tool that enables easy aggregate document generation from source documents available on European Charter for Regional or Minority Languages website.

1.3 Data sources

The sources of data come from specified list of websites[1] website, all of which are available in PDF format. There are 651 documents linked on the page, of which 309 documents are in English (some documents are produced also in French or country producing report official language), of which 300 documents are of actual interest (rest includes charter advertisements or more general texts). There are 25 countries producing reports, each adhering to general layout, but with subtle differences coming from the tool used or typesetting conventions, which should be taken into account during parsing PDF into JSON[2].

1.4 Stakeholders

- Researchers - people who are using service often for their research;
- Consumers - people who need to access data occassionally and do not have need to register an account in the service;
- Administrators - people having rights to add sources, respond to tickets, etc.;
- Developer - person responsible for the project development and maintenance.

1.5 Project Name

The project name *Seeker* comes from "Seeker, Reaper" poem by George Campbell Hay. You can listen to the peom being recited on YouTube here.

[1] However, it could be adapted for more general approach with web crawler searching for interesting kind of data, but that would involve scalability issues of storing and parsing such large data set and automatic recognition ofatRegional or Minority Languages

[2] The best parser would have simple regular expression and extract only relevant sections without specific formatting. This has been achieved for this project with context aware parser.

Chapter 2

Requirements

2.1 Minimal requirement

The goal for 8 week summer project was set to

- carry out research and recommend appropriate solution (architecture, full software stack, and persistence layer);
- Implement a proof of concept functionality to prove the possibility of developing efficient solution for the stated problem, this is, ability to query European Committee for Minority Languages and receive combined documents composed of elements specified in the query.

2.2 Data format

The main source of the data are going to be the European Charter for Regional or Minority Languages expert reports[1].

The goal of the project is to account for the future grow and to include also documents from Galeic Plans produced by various bodies in Scotland [2].

The data needs to be assumed to do not have any common format or fields.

2.3 Use cases

During the initial meeting on 26 May 2014 the below use cases had been identified.

[1] `http://www.coe.int/t/dg4/education/minlang/Report/default_en.asp`
[2] http://www.gaidhlig.org.uk/bord/en/our-work/gaelic-language-plans.php

Chapter 3

Requirements analysis

3.1 Objectives

- Contiunable - the project development should be easy to continue
  - Well-documented
  - Popular and well-documented technologies should be used
  - Well-organized code base
- Schemaless - documents (sometimes even from the same source) will have different structure, so it is important to not restrict the database to store one kind (structure) of information
- Adaptability - it shoud be easy to adapt a project to a new kind of documents/task
- Modularity - the distinguishable project parts should be implemented in modular fashion to increase clarity and make the future development easier
- Open source - all technologies and tools used should be open source

3.2 Data layer

3.2.1 Example query

1. Show all Committee of Experts recommendations under article 7b.
2. Show all sections relating to Scottish Gaelic.

3.2.2 Options

The following database technologies were taken into consideration:

- Traditional RDMS such as PostreSQL and MySQL
- Key-value store such as Riak and Redis
- Document database such as MongoDB and CouchDB
- Graph database such as Neo4J and Infinite Graph

Traditional RDMS has drawback of dedicated to the outlined schema and every new document (not to mention every new source of documents) has a chance to have different organization than the previous ones.

Key-value store has not enough good query languages and

Document database has both flexible structure of aggregates (so we are not dedicated to one schema) and powerful query language (which can be implemented using incremental update)

Graph database has ability to perform rich queries regarding relationships, but is not as powerful when it comes to queries relating the content, can answer queries like 1 and 2 in O(1) time, but update is expensive.

3.2.3 Choice

MongoDB has been chosen for storing documents because

- it has big community of supporters and developers;
- it is well-documented;
- naturally blends with JavaScript when implementing query (JQuery notation) and data from the database (JSON);
- can store and operate on big documents of varying schema easily
- has rich query language with greater capabilities than the need outlined in the example queries subsection;
- when used with Node.js increases programmers productivity significantly (one language, centralized view)

When developing session and account management functionality it would be beneficial to implement the concept of polyglot persistence and for these two use key-value database, which creates separation of the concers as well as suits better the case.

3.3 Application layer

3.3.1 Example usage

3.3.2 Options

Three programming languages/frameworks were considered as potentially suitable for the project, namely:

- Java and JSP
- Python and Django
- JavaScript and Node.js

Java and JSP does not support well the programmer productivity as to perform even the simplies operations it is very expressive. When used the application needs to be developed and tested as a whole, so it does not support modularity and extendability well. The data layer technology of choice provides API for the language, but it is rather easier and more natural to operate on it in JavaScript than in Java.

Python and Django had the same drawbacks, except it provides greater productivity.

JavaScript and Node.js means that one language would be used for front-end and back-end, including data quering and manipulation, what highly increases the programmer productivity. Further, JavaScript as front-end language provides rich choice of libraries like jQuery, Angular, etc. that can be used to further increase productivity and presentation quality. When it comes to the back-end all operations are handled efficiently using event-loop. The aggregates are retrieved in JSON format and MongoDB can be naturally quiered using jQuery like notation. This all results in complete and efficient solutioon.

3.3.3 Choice

Node.js has been chosen for the front-end and back-end development because

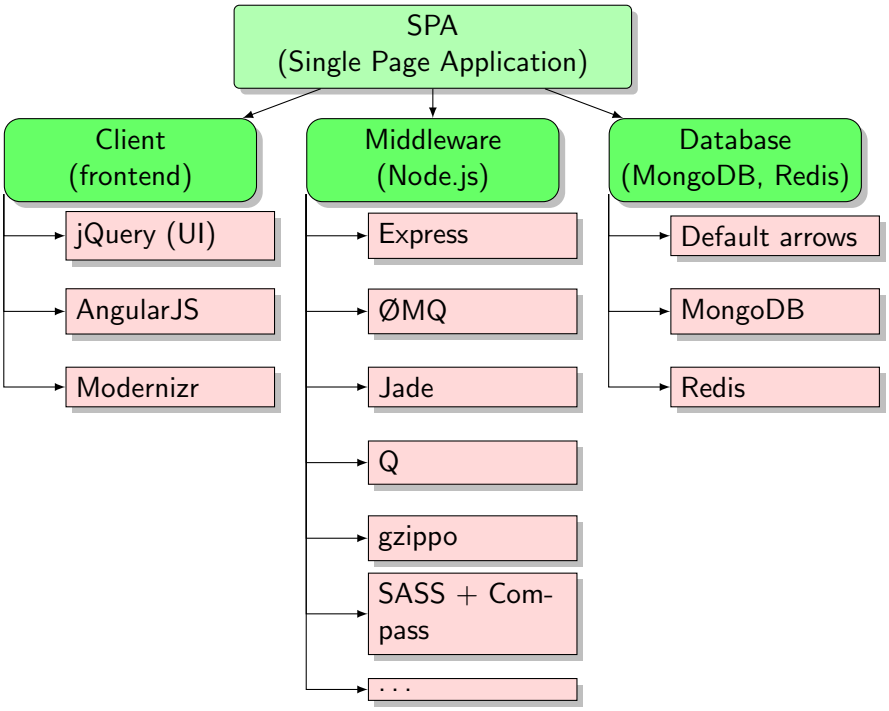
- it integrates well with MongoDB
- enables using one language for everything (front-end, back-end, database);
- is efficient

Chapter 4

Architecture

4.1 Overview

4.1.1 Application



Description

Client:

- jQuery (UI)
- AngularJS
- Modernizr

Middleware:

- Express - main framework for message passing and service the content to the clients
- ØMQ - for message passing, mainly for efficient Router/Dealer protocol implementation
- Jade - for more productivity in templating
- gzippo - for serving gzipped content
- SASS + Compass - styling, plus features like building sprites out of files

Database:

- MongoDB - document data store for document management *TODO*
- Redis - key-value data store for session and user data management

4.1.2 Full layer

Picture showing Yeoman, Grunt, Karma testing with Jasmin, etc. code: sudo npm install -g yo # installs yeoman, grunt, bower, etc. globally sudo npm install -g generator-angular-fullstack # install angular generator (btw the most popular generator) yo angular-fullstack seeker (projectname)# create full stack locally with jade tempalates [?] Would you like to use Sass (with Compass)? Yes [?] Would you like to include Twitter Bootstrap? Yes [?] Would you like to use the Sass version of Twitter Bootstrap? Yes [?] Which modules would you like to include? angular-resource.js angular-cookies.js angular-sanitize.js angular-route.js mongo with mongosee passport All yeses

4.1.3 Code management and deployment process

4.2 Full stack

4.2.1 Client facing

4.2.2 Middleware

4.2.3 Persistence data store

4.3 System design

4.4 Query language

4.4.1 Elements

- *source documents* (e.g. Committee of Experts reports)
- *document specifiers* (e.g. cycle, country producing the document)
- *selectors* (i.e., languages, sections, words)
- *organizer* specifying how to structure the output document (e.g. arrange by section numbers, in each section by countries alphabetically)
- *template* choose how to style the output document

4.4.2 Dependencies as tree

1. *Source documents* must be root.
2. *Document specifiers* can be children of each another, but can occur only once on any path from root to leaves, can appear multiple times.
3. *Selectors* (i.e. languages, sections, words) can be children of any node, and they are always terminal (leaves).
4. *Organizers*, *templates*, and other similar options which can be added in the future must be children of root, and in case of *organizers* they must be aware of all root's children except itself to display available options.

4.4.3 Query evaluation stages

1. Select relevant documents using:
    - producer - group (e.g. committee)
    - producer - origin, i.e. countries (e.g. UK)
    - cycles - time (e.g. last)

2. filter the document content using selectors (i.e. sections, languages, filters- words, phrases, etc.)

3. reorganize resulting JSON using organizers

4. render using template