# otvorenezmluvy.sk

*Technical documentation*

# Introduction

CrowdCloud is an application that helps to collect, search, analyze vast amounts of documents and through power of crowdsourcing find the interesting ones. We use powerful features that help users to slice-and-dice documents using faceted search and notify users when documents matching their criteria are added. Users can interact with documents by commenting, marking them as suspicious and even highlighting parts of documents. Documents are automatically downloaded from public sources and processed through OCR that extracts text even from non-text documents.

The pilot application running on www.otvorenezmluvy.sk collects Slovak government contracts (documents) that are published at various official government sites.

# Installation

CrowdCloud is a classic Ruby on Rails 3.2 application, gems are managed using Bundler, application is deployed via Capistrano. Database schema is managed using raw SQL schema dumper (not Rails default).

Full source code is available at https://github.com/otvorenezmluvy

## Prerequisites

- Linux environment (debian-based is prefered)
- MRI Ruby 1.9.3+
- PostgreSQL database 9.1+
- ElasticSearch
- Redis
- Tesseract OCR
- GraphicMagick
- MemCache (optional)

## Installation guide

1. Install and start prerequisites if needed (e.g. elasticsearch, redis).
2. Add eulang analyzer (for handling accented characters) to elasticsearch.yml config

```
index:
  analysis:
    analyzer:
      eulang:
        type: custom
```

```
tokenizer: standard
filter: [standard, lowercase, asciifolding, stop]
```

3. Application settings for database and elasticsearch can be found in `database.yml` and `crowdcloud.yml`
4. Run default Rails application bootstrap (`bundle install, rake db:create, rake db:schema:load, rake db:seed`)
5. Create elasticsearch indexes & index data (if available) with `rake crowdcloud:index:rebuild`
6. *Optional.* Start downloading data from CRZ source by running `rake rowdcloud:crz:download` and `rake resque:work`

# Project structure

## Top level architecture

Project consists of these three main parts:

- Wrappers/parsers
- Public web interface
- Administration interface

## Data model

The core model works with arbitrary documents and attachments as defined in the Documents Core module, the pilot implementation (www.otvorenezmluvy.sk) deals with contracts (a specialization of document) from two sources with corresponding modules CRZ and Egov.

### Documents Core

- *Document* - represents a single published document
  - has_many *Attachments*
  - has_many *Comments*
  - has_many *Heuristics*
- *Attachment* - represents an attachment (pdf/text) file related to a *Document*
  - has_many *Pages*
- *Page* - represents a single page of an *Attachment*
  - has_many *Comments*

### CRZ Specific Documents

- *Crz::Document* < *Document* - abstract class represents a document from CRZ portal

    - has_one *Crz::DocumentDetail*
- *Crz::DocumentDetail* - represents various specific fields for a *Crz::Document*
- *Crz::Contract* < *Crz::Document* - represents a contract from CRZ portal
    - has_many *Crz::AppendixConnections*
- *Crz::Appendix* < *Crz::Document* - represents an appendix from CRZ portal
- *Crz::AppendixConnection* - represents a connection to an appendix from a *CRZ::Contract*

### EGov Specific Documents

- *Egovsk::Document* < *Document* - abstract class represents a document from Egov.sk portal
    - has_one *Egovsk::DocumentDetail*
- *Egovsk::DocumentDetail* - represents various specific fields for a *Egovsk::Document*
- *Egovsk::Contract* < *Egovsk::Document* - represents a contract from Egov.sk portal
- *Egovsk::Appendix* < *Egovsk::Document* - represents an appendix from EGov.sk portal

### Intelligence

- *Heuristic* - represents an automatic heuristic for scoring document suspiciousness.
    - has_many *Documents*

### Questionnaire

- *Question* - represents a question from questionnaire.
    - has_many *QuestionChoices*
- *QuestionChoice* - *r*epresents a question choice for a particular *Question*
    - has_many *QuestionAnswers*

### User activities

- *Comment* - represents a user comment on a *Document* and/or particular *Page* area of *Document*.
    - has_many *Votings*
    - has_many *CommentReports*
- *CommentReport* - represents a user reporting of a *Comment* (e.g. malicious, abusive...)
- *Voting* - represents users voting for/against a particular *Comment*
- *Watchlist* - represents a *Document* watched by a *User*
- *DocumentOpening* - represents a *Document* opened by a *User*
- *QuestionAnswer* - represents an user answer to a *Question*

### Stream

- *Event* - represents an event in user's activity stream and has following subclasses
    - *UserRegisteredEvent* - represents an event when user has registered
    - *QuestionAnswerEvent* - represents an event when a question is answered

- *DocumentEvent* - abstract class representing event related to a document
  - *DocumentOpenedEvent* - represents an opening of document by user
  - WatchingStartedEvent/WatchingStoppedEvent- represents an event by user that starts/stops watching a document
  - *ControversyReportedEvent* - represents an event when users marks a document as controversial
- *CommentEvent* - abstract class representing event related to a comment
  - *MyCommentEvent* - represents an event related to users comment
  - *OthersCommentEvent* - represents an event related to comments by other users (e.g. on a watched document)

# Processes

### Downloading and preprocessing contracts

New contracts are downloaded periodically, from a cron job. The cron job is defined in `config/schedule.rb`. This file is automatically picked up by the *whenever* gem which translates it into a cron configuration and updates local crontab on each deploy. Jobs are then managed by unix daemon *cron* and run at the specified intervals. The cron job invokes *rake* tasks defined in `lib/tasks/crowdcloud.rake` which start the download process. The rake tasks check the remote websites (currently CRZ and egov) for new updates and schedule downloads for each new contract. The rake tasks themselves do not download data, they just parse remote websites/invoke third-party APIs and schedule *Resque* jobs.

The actual downloading and preprocessing happens in Resque jobs. The actual logic and execution depends on the scraped website, but these steps are common among all scraping jobs:

- download the HTML with contract metadata
- parse the HTML, extract metadata and links to attachments
- download the pdf attachments
- split pdfs into images (one image per page) using a tool *graphicsmagick*
- extract text, either using *pdftotext* tool or *Tesseract OCR*

The metadata is saved in database and the images and texts on filesystem in `public/documents`.

Note that in order to process the queued jobs, separate resque workers must be running. Resque workers can be started by invoking `rake resque:work`.

# Components

### Document viewer

Document viewer is a components responsible for displaying the contracts. It can display scanned contract, its extracted plain text and provides tools for interactive annotating. It is implemented as a standard *Backbone.js* application in `app/assets/javascripts/document_viewer`. The same viewer is used both on portal to show contracts and in the widget, although with different configuration. Document viewer is instantiated as shown in the example and accepts several options.

```
var dv = DV.DocumentViewer.init({
  container: '.document',
```

```
  attachments: <%== @attachments.to_json %>,
  currentUser: "<%= current_user.label %>",
  onReply: CrowdCloud.Comments.replyTo,
  onShowReplies: CrowdCloud.Comments.showComment,
  width: '673px',
  height: '990px',
  annotationsAllowed: true,
  zoom: false,
  commentList: false
});
```

| **Option name** | **Type** | **Description** |
|---|---|---|
| `container` | `string` | CSS selector for the wrapping DOM element where the document viewer shall be rendered. |
| `attachments` | `array` | JSON-encoded array of attachments. Each attachment should provide: name, number and array of pages. Each page should provide: scanUrl, textUrl and optionally an array of annotations. |
| `comments` | `array` | JSON-encoded array of comments related to the document. |
| `currentUser` | `string` | Name of the user. It is displayed when adding new annotation. |
| `onReply` | `function` | Callback invoked when user chooses to reply to annotation. |
| `onShowReplies` | `function` | Callback invoked when user chooses to see replies to an annotation. |
| `width` | `integer` | Width of the document viewer in pixels. `parseInt`'d before usage. |
| `height` | `integer` | Height of the document viewer in pixels. `parseInt`'d before usage. |
| `annotationsAllowed` | `boolean` | Should we allow to annotate parts of the contract? |
| `zoom` | `boolean` | Can we zoom? |
| `commentList` | `boolean` | Should we display a toggleable tab which shows all comments? If set to true, the comments are pulled from the `comments` option. |

# Extending the project

## Adding a new source parser / wrapper

Before adding a parser/wrapper for a new source you will probably need to add a model for holding custom fields for your documents. The best way to start is to look how existing models work and extend *Document* class (e.g. *Crz::Document*, *Crz::DocumentDetail* or *Egovsk::Document*, *Egovsk::DocumentDetail*).

*A word of warning: These models **don't** use classic single table inheritance that is common for RubyOnRails, but for the sake of extensibility pull custom fields to separate models *DocumentDetail.*

If you want your documents custom fields to be indexed (to fully leverage the power of faceted fulltext searching) make sure you add them to *Document::Indexable* mixin and refer to the next section of how to add a new facet to the interface for such fields.

There are no restrictions on how to download, parse and process your own documents. Look at existing parsers for Crz and Egovsk for example production ready parsers.

**Warning: Make sure documents are saved using *Configuration.document_repository .save(document)*** method instead of the default *ActiveRecord::Base.save*. Using a repository object decouples model from various callbacks (indexing, heuristics calculations, etc.) and is considered a best practice for easier testing.

## Extending faceted search

### Adding a facet

Facets for search GUI are defined in *Settings.facets* method and there are multiple ready-to-use types of facets:

- *FulltextFacet* - fulltext search in document (_all field in ElasticSearch) you probably need only one of this facet.
- *SearchableTermsFacet* - facet containing strings with autocomplete feature
- *DateFacet* - facet for fields containing date, default bucketing by month.
- *RangeFacet* - facet for arbitrary ranges
- *StatisticalFacet* - facet for calculating statistical properties on fields (e.g. sum)

After adding new facet/s to this definition you might need to supply some translations and templates for customizing the look&feel to make it fully working, but the process is standard and

should be straightforward for any Rails developer.

### Adding a sort field

Adding a new field that should be used for sorting is done by adding it to definition defined in Configuration.factic[:sort_fields]. Again, some translations need to be added.

# Contact information

**Authors:** minio, s.r.o. (kontakt@minio.sk), Aliancia Fair-Play (www.fair-play.sk), Transparency International Slovensko (www.transparency.sk)

Feel free to contact us anytime.