SERP Connect

Release

Feb 21, 2018

Developer Documentation

This is documentation for SERP Connect, here you can find docs detailing the inner and outer workings of the SERP Connect backend and frontend.

CHAPTER 1

Getting Started

This file is dedicated to getting you ready to start developing. If you have questions, head over to our slack channel and fire away!

1.1 Backend

We start with the backend, simply because it is required for you to have any functionality on the frontend. A high-level checklist is:

- install java 8 (sdk & jre)
- install maven version 3
- install neo4j version 2.X.Y
- logon `localhost:7474` and change password
- update password in `application.properties` in repo.
- start with `mvn exec:java -Dpippo.mode=dev`

1.1.1 Setting up the backend

java 8

Java 8 (SDK & JRE) is required. Almost all os have a standard way of installing and upgrading java. These guides may work for you, but ideally you should look it up.

Test if you already have java 8 by running (any os):

- JRE: `java -version`
- SDK: `javac -version`

ubuntu/debian

- JRE: `sudo apt-get install default-jre`
- SDK: `sudo apt-get install default-jdk`

windows: Download the JRE and SDK from oracle.

os x/mac os: install homebrew, then `brew install java`

other linux

- try the default package manager
- otherwise check this out

neo4j

Install the **community edition**, **version 2.3.X** where X is the highest you can find. Again, the installation process depends on your os/environment. Here is the official documentation. Below are summaries:

ubuntu/debian

- install neo4j via apt-get (instructions)
- run `system neo4j start` to start
- run `system neo4j stop` to stop

windows

- download the installer (.exe) from legacy
- run it and install neo4j
- start & stop neo4j using the `neo4j-ce.exe` program

mac

- download the neo4j dmg from legacy
- drag neo4j to your applications folder
- use that program to start & stop a neo4j server

other linux

- download a binary from legacy
- untar it `tar -xzf <neo4j-download.tar.gz>`
- run `pwd` to get current working directory
- run `echo 'export $PATH=/full/path/to/neo4j-download/bin/:$PATH' >> ~/.profile`
- run `source ~/.profile`
- now your current and all new shells will be able to run the neo4j script
- run `neo4j start` and `neo4j stop` to start & stop, respectively

After installation you should start a neo4j server and navigate to `http://localhost:7474` and login using neo4j/neo4j. Choose a new password, **and remember it**. You *will* need it later on.

maven

Maven is a java package manager, amongst other things. We use version 3.

mac: install homebrew, then `brew install maven30`

general

Try your luck with the package manager, otherwise these links are handy:

- download
- install

1.1.2 First steps

After all required software has been installed you are ready to proceed.

- Strap up! `cd ~`
- Organise `mkdir connect && cd ~/connect`
- Clone `git clone git@github.com:emenlu/connect.git backend`
- Charge in `cd backend`
- Open `src/main/resources/conf/application.properties` in your editor of choice
- Change `neo4j.password` to what you entered previously in the web ui
- Then run `mvn compile exec:java`
- The backend will now create a superuser and initialize the database. **(Make sure neo4j is running!)**

Et voilà, you are ready!

In the future, run `mvn compile exec:java -Dpippo.mode=dev`. It executaes both commands sequentially and launches the server in dev mode.

1.1.3 Eclipse

To use Eclipse, simply import the backend files as a github repository. We recommend using a maven plugin to facilitate running the server.

1.2 Frontend

The frontend project is often much simpler to install since it only depends on nodejs, thus this section is mainly on how to install nodejs. The high-level checklist is:

- install nodejs (v5 or v6)
- run `npm install` in repo.
- run `make dev`
- browse to `localhost:8181`

1.2.1 Setting up the frontend

installing nodejs

The frontend relies on nodejs to compile page templates and style files. Node.js has its own package manager, called `npm`, which lists dependencies in a `package.json` file. Thus the only programs you need to manually install are `npm` and `node`. Thankfully, `npm` is bundled with `node` so installing `node` is sufficient.

Type `node -v` in a terminal to check version:

- v5 and v6 are confirmed to work with the connect frontend
- v7 and v8 are unknown

If you already have node v7 or v8 then you should install a version manager to switch between multiple version. Here are a few, though some only support specific operating systems:

- nvm
- n
- nvs

The actual steps for installing Node.js vary depending on your operating system.

windows: Download the installer and run it. It will install `node` and `npm` and put them in your `%PATH%`.

mac os: Install with homebrew. `brew install node@6`

linux

Most popular linux distros have up-to-date packages of `node` and this is the easiest way to install nodejs. There is a guide here on doing this.

If this fails you must download a tarball and put `node` and `npm` into `/usr/bin` or similar. Some linuxes have a program called `alternative` to symlink files into `/usr/bin`.

installing packages

Type `npm install` in the repository to install all dependencies. Then try to run the dev. server using `make dev`. If this fails, report to Axel. Otherwise you are good to go!

CHAPTER 2

Contributing

This file is mirrored here.

2.1 Licensing

connect is licensed under the BSD 2-clause license.

2.2 Documentation

All documentation is avaiable at readthedocs for developer guides.

Join our slack channel for questions and other stuff!

2.3 Depedencies

Connect itself is mainly built on pippo and jcypher but requires some tools to actually compile/test:

- maven 3.X.Y (website)
- java 1.8
- neo4j 2.X.Y (website)

For more information about getting started, see the documentation.

2.4 Versioning strategy

The `master` branch receives all development work via merges from feature branches. New releases are tagged according to semantic versioning.

2.5 Contributions

There are primarily two ways you can contribute to connect:

- **Issues:** Post bug reports, feature requests and code changes as issues to the github repo.
- **Pull requests:** Well-defined (fixes one issue) pull requests are happily accepted as long as they pass the test suite. If you fix bugs please provide a test case to combat regression.

2.6 The procedure

- Pick the issue: It is recommended to pick existing issues but we accept contributions not related to any issue. After picking an issue, please comment and state something like "hi, i'm calling dibs on this one". This will allow people with access to update status flags and what not to reduce risk of collisions.
- Begin by preparing for work:
- Make sure your master is in sync: `git checkout master && git pull upstream master`
- Make a feature branch: `git checkout -b feature-branch`
- Work work and make commits
- After that sweet sweet code we suggest you test: `mvn clean verify`
- If, for some reason, the test doesn't work don't worry too much - the CI will test for you
- It's time to prepare for pull request:
- Again, make sure master is up-to-date: `git checkout master && git pull upstream master`
- Forward your branch: `git checkout feature-branch && git rebase master`
- Push to your repo: `git push origin feature-branch`
- Make pull request from github.

2.7 Landing the PR

- Please refer to the relevant issue and provide a clean description of your solution.
- It is recommended (but not *required*) to rebase your commits into a clean flow (to simplify review)
- Mention @emenlu when your PR is ready and she will assign someone to poke at it

2.8 Students @ LTH

For students at LTH, Lunds University there exists a public Trello board here. Tasks from this board can give you some $$$ when completed.

CHAPTER 3

Development

A quick intro on what you might want to know when starting backend or frontend development.

3.1 Backend

The core application logic resides in the backend. We strive to keep the backend logic as simple as possible.

3.1.1 Backend dependencies

- maven version ~3
- java sdk version ~1.8
- neo4j version ~2.6

3.1.2 Backend design

Here is an overview of the different components that make up the backend. Each box corresponds to a java package with a similar name.

- `Connect` initializes the server modules (`se.lth.cs.connect.modules.*`) and routes (`se.lth.cs.connect.routes.*`)
- General database queries are executed by `Database`. Queries are built using jcypher helpers.
- Specialized database queries should be hidden by an interface, e.g. login/authetication logic is implemented in `AccountSystem`.
- The two exceptions (`DatabaseException.java` and `RequestException.java`) are thrown by handlers and handled by a function in `Connect.java`, like a bubble-style event.

- Event cascading is explicitly specified in the `se.lth.cs.connect.events` package and related classes. Example: `when an entry is deleted orphaned facets should also be deleted`. Not all user-initiated actions have an associated event and events should be added thoughtfully and sparingly as they increase overall systematic complexity.

3.1.3 Backend tips

Here are some tips new contributors might appreciate:

- We use a graph-database called Neo4j. The quickest way to get up to speed of what it is and how it works is to visit their website.
- Neo4j runs a graphical interface located at `http://localhost:7474/browser/`. This very helpful to try and prototype commands and later to see if correct connections and data was added to the database.
- jcypher is used for querying the database from the backend much like SQL but with different syntax. The documentation is somewhat limited but there are examples on their github wiki. You should probably look at how things are done in the connect.routes package before trying your luck at the wiki, though. Note that jcypher can't always translate a Neo4j query directly.
- The `application.properties` file has to be updated with the correct username and password for neo4j database.

3.2 Frontend

3.2.1 Frontend dependencies

- node js version 5, 6 or 7 (confirmed working)

3.2.2 Frontend overview

Here is an overview of the different components that make up the frontend.

The structure is quite straightforward:

- We use jade, less and js to make up the webpages. The less and js files both have a base file which they are dependent on, then each page sub-levels down to have it's unique properties which the rest of each pages sub-levels depend on.
- Jquery v3 is used and is imported via the base class so this is standard across all pages.
- For styling, LESS is used. Structurally we have a working files folder which divides the pages up and then it is imported into an `all.less` file which is converted into one CSS file.
- There are two api files: `api.js` and `api-dev.js`. The `api-dev.js` file only changes the server to which the ajax queries from `api.js` are made.

3.2.3 Frontend tips

- It can be important to run `make clean` every now and then to be sure nothing is cached and the changes implemented are what you have made. **`make clean` is run automatically when running any make command.**
- There are a few utilities which are used across some of the pages. They can be found in `src/js/util`.

- One example is el.js which is used to efficiently create elements. It is encouraged to use these utilities where possible to keep the coding consistent.
- There is no need to (re)build after *modifying* any files. Simply save the file and reload localhost website to see the changes.
- If you add new views, then add them to `app.js`.
- If you add new LESS files, add them to `src/less/all.less`.

CHAPTER 4

Testing

Connect relies on a number of test steps:

- maven integration tests (unit tests are supported)
- circleci continuous integration link on commits and pull requests
- codecov coverage reports link generated with jacoco

In addition to the integration testing and coverage reporting done by circleci developers are also expected to test their changes locally. The maven commands are:

- run unit tests only: `mvn test`
- clean `target/`, run unit tests and then integration tests: `mvn clean verify`

During integration testing an included neo4j server will automatically be started. **An error will be thrown if a unit test case tries to access a route that queries the database.** The database is *not* ephemeral, but lives in the `target/` directory. Most tests will create stuff in the database and will run into problems if entries, collections or users already exists. This is why it is recommended to run `clean` before `verify`.

4.1 Writing test cases

The connect backend is tested using the pippo module `pippo-test` (link).

There are some additional modules that are used to facilitate easier testing of expected behaviour.

- Some API actions will send an email. It is possible to use a mock `MailClient`, such as `Mailbox` to capture and verify mails.
- A `URLParser` class helps with link extraction.

4.2 Detecting test files

A test file is run during unit testing if it matches `*Test.java` or `Test*.java`.

Integration tests are matched with `IT*.java` and `*IT.java`.

CHAPTER 5

Graph

5.1 User

Accounts are stored in the neo4j graph, labeled with `user`. The account identity is the email used to sign up (and later verify account). Passwords are hashed with scrypt @ standard settings (N = 16384, R = 8, P = 1) and transmitted in plain-text. (Use HTTPS!)

5.2 Project

A project is represented as a graph node with a `project` label. Projects are the basic premise on which collections expand. Each project has a taxonomy.

5.3 Collection

A collection is represented as a graph node with a `collection` label. Collections are the main and only way of organizing entries and are mainly identified by a combination of their unique id and non-unique (user provided) name. Each collection has a (potentially nil) set of taxonomy extensions of the base taxonomy (provided by the project).

5.4 Entry

An entry is represented as a graph node with a `entry` label. Each entry has an additional entry type label: `research` or `challenge`. Entry nodes contain the properties specific to the entry type. They also have relations to their entities. The relationship type is the taxonomy classification, e.g `:ASSESSING`.

5.5 Entity / Facet

An entity is represented as a graph node with a `facet` label and is the free text sample of at least one entry. The relation type between an `entry` and `facet` determines the classification of that entity (text sample) for that specific entry. Many entries can classify the same entity, even with different classifications.

5.6 Token

A token is represented as a graph node with a `token` label. Tokens are in fact a special type of authentication nodes. Tokens are given meaning by their relation to the user node. At the moment only two relation types exist: `EMAIL_TOKEN` and `RESET_TOKEN` for email verification and password reset, respectively. At the moment tokens don't expire (but they probably should).

CHAPTER 6

API

This is the specification of the public API.

Status codes are generally:

- 200: ok
- 400: something wrong with the request
- 401: authentication error
- 404: not found
- 500: server or database error

If an endpoint has parameters they are required for the request to success (otherwise a 400 is thrown). A parameter not found in the URL should be sent in the request body as `application/x-www-form-urlencoded`. Some endpoints require input as JSON. The endpoint description will include a special JSON Request object if JSON is required.

6.1 Project

A project specifies the default taxonomy which collections can extend. Usually each project is hosted on a different website.

```
{
   "name": "serp",
   "link": "http://serpconnect.cs.lth.se"
}
```

Where `name` is a unique (across backends) project name and `link` a url to the website of the project.

6.1.1 Query projects

GET /v1/project
Get a list of all known projects.

Example response:

```
{
    "projects": [PROJECT]
}
```

Response JSON Object

- **projects** (*array*) – An array of *Project* objects.

Status Codes

- 200 OK – ok, return taxonomy

6.1.2 Create new project

POST /v1/project
Create a new project listing.

Parameters

- **name** (*string*) – unique, alphanumeric name ([a-zA-Z0-9])
- **link** (*string*) – url to website of the project

Response Headers

- Content-Type – application/json

Example response:

```
{
    "name": "serp-test",
    "link": "http://test.serpconnect.cs.lth.se"
}
```

Response JSON Object

- **name** (*string*) – the name you provided
- **link** (*string*) – the link you provided

Status Codes

- 200 OK – ok, echo back the project details
- 400 Bad Request – name/link missing or incorrect name/already taken
- 401 Unauthorized – must be logged in
- 403 Forbidden – only verified users can create projects

6.1.3 Query project taxonomy

GET /v1/project/(string: *name*)/taxonomy

Get a flattened version of the project taxonomy. The flattened graph assumes an implicit "ROOT" node object as the top parent.

Parameters

- **name** (*String*) – unique, alphanumeric project name

```
{
    "version": 0,
    "taxonomy": [FACETS]
}
```

Response JSON Object

- **version** (*integer*) – A version identifier.
- **taxonomy** (*array*) – An array of *Facet* objects. The flattened taxonomy.

Status Codes

- 200 OK – ok, return taxonomy
- 404 Not Found – project not found

6.1.4 Update project taxonomy

PUT /v1/project/(string: *name*)/taxonomy

Update the extended taxonomy. The request will only pass if the version is >= (greater than or equal to) the currently stored version.

Parameters

- **name** (*string*) – project name

```
{
   "version": 0,
   "taxonomy": [FACETS]
}
```

Request JSON Object

- **version** (*integer*) – Reference to the version this extension is based on.
- **taxonomy** (*array*) – The *Facet* nodes of the extended taxonomy.

Status Codes

- 400 Bad Request – illegal json, out of date version
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a admin or creator of project project
- 404 Not Found – no project with that name exists

6.2 Graph

A graph consists of entries and edges.

GET /v1/entry
Fetch all entries and edges in the database.

```
{
      "nodes": [ENTRIES],
      "edges": [EDGES]
}
```

Response JSON Object

- **nodes** (*array*) – An array of *Entry* objects
- **edges** (*array*) – An array of *Edge* objects

Status Codes

- 200 OK – ok, return graph

6.2.1 Graph Taxonomy

GET /v1/entry/taxonomy
Get a flattened version of the standard SERP taxonomy. The flattened graph assumes an implicit "ROOT" node object as the top parent.

```
{
    "version": 0,
    "taxonomy": [FACETS]
}
```

Response JSON Object

- **version** (*integer*) – A version identifier.
- **taxonomy** (*array*) – An array of *Facet* objects. The flattened taxonomy.

Status Codes

- 200 OK – ok, return taxonomy

6.3 Facet

A node in the taxonomy tree is called a facet.

```
{
   "id": "PLANNING",
   "name": "Test planning",
   "parent": "SCOPE"
}
```

Where `id` is a (per-taxonomy) unique identifier of this facet, `name` is a descriptive name and `parent` is the `id` of the parent node (since a taxonomy is a tree).

6.4 Edge

An edge looks like this:

```
{
    "source": 9,
    "target": 13,
    "type": "PLANNING"
}
```

Where `source` is the origin entry node id, `target` is the targeted entity node id and `type` the (SERP) classification of this relation.

6.5 Entry

An entry is either a classified challenge or research result that a user submitted to the database. Each entry consists of entry-specific information and a classification. These two pieces of data must be queried separately. See *Find entry by id* and *Get entry taxonomy*.

6.5.1 Find entry by id

GET /v1/entry/(**int:** *entry_id*)
Retrieve information of an entry specified by *entry_id*.

Parameters

- **entry_id** (*int*) – entry's unique id

Response Headers

- Content-Type – application/json

```
{
    "id": 55,
    "hash": "YOnPVli1utklw1a3LXiw9pBl6gmpsd4BUabV9I1UyhA=",
    "type": "research",
    "contact": "space_monkey@planet.zoo",
    "reference": "An In-Depth study of the Space Monkey Phenomenon",
    "doi": "doi:xyz",
    "description": null,
    "date": null,
    "pending": false
}
```

Response JSON Object

- **id** (*integer*) – a (recycled) unique id
- **hash** (*string*) – unique hash of this information
- **type** (*string*) – challenge or research
- **contact** (*string*) – not used
- **reference** (*string*) – only valid for research type entries, lists relevant references
- **doi** (*string*) – only valid for research type entries, optional, the DOI of a related paper

- **date** (*string*) – currently broken, a standard javascript date
- **pending** (*boolean*) – is entry pending admin approval

Status Codes

- 200 OK – ok, return information
- 400 Bad Request – entry_id must be an int
- 404 Not Found – no entry with that id exists at the moment (it might have existed but was deleted)

6.5.2 Get entry taxonomy

GET /v1/entry/(**int:** *entry_id*)**/taxonomy**

Retrieve the taxonomy of a specific entry.

Parameters

- **entry_id** (*int*) – entry's unique id

Response Headers

- Content-Type – application/json

```
{
    "INFORMATION": [
        "No data currently collected"
    ],
    "SOLVING": [
        "unspecified"
    ],
    "PLANNING": [
        "testing environment trade-off (simulated, real system production)",
        "testing phase trade-off",
        "testing-level trade-off (function, interaction)",
        "automation trade-off"
    ]
}
```

Response JSON Object

- **<key>** (*array*) – each key corresponds to a classification with entities

Status Codes

- 200 OK – ok, return entry taxonomy
- 400 Bad Request – entry_id must be an int
- 404 Not Found – no entry with that id exists at the moment (it might have existed but was deleted)

6.5.3 Submit new entry

POST /v1/entry/new

Submit a new entry.

Request JSON Object

- **entryType** (*string*) – either `challenge` or `research`
- **collection** (*int*) – unique id of collection to add entry to
- **reference** (*string*) – only required for research entries, a list of references
- **doi** (*string*) – optional for research entries, a DOI of this publication
- **description** (*string*) – only required for challenge entries, describing the challenge
- **serpClassification** (*json*) – the SERP classification
- **date** (*string*) – javascript date text representation

Example request json:

```
{
    "entryType": "challenge",
    "collection": 2,
    "description": "how to do software dev without cookies?",
    "date": "Mon Sep 28 1998 14:36:22 GMT-0700 (PDT)",
    "serpClassification": {
        "IMPROVING": ["cookies for software dev"],
        "INFORMATION": ["hungry hungry devs"]
    }
}
```

Example response:

```
{
    "message": "ok"
}
```

Status Codes

- 400 Bad Request – bad request
- 401 Unauthorized – must be logged in to submit new entries
- 403 Forbidden – must have verified email addr before submitting entries, must be member of collection

6.5.4 Edit existing entry

PUT /v1/entry/(**int:** *entry_id*)

Edit taxonomy and/or fields of an existing entry. Request is same as *Submit new entry*, but without a `collection` field.

param entry_id unique id of entry

type entry_id int

Example request:

```
{
    entryType: "challenge",
    description: "how to do software dev without cookies?",
    date: "Mon Sep 28 1998 14:36:22 GMT-0700 (PDT)",
    serpClassification: {
        "IMPROVING": ["cookies for software dev"],
        "INFORMATION": ["hungry hungry devs"]
```

```
    }
}
```

Status Codes

- 400 Bad Request – entry_id must be an int
- 403 Forbidden – must be member of at least one of the collections that own the entry

6.6 Account

6.6.1 Authenticate

POST /v1/account/login
Authenticate user.

Status Codes

- 200 OK – ok, user is logged in on the returned session token
- 400 Bad Request – email/passw combination is invalid

6.6.2 Register an account

POST /v1/account/register
Register new user.

Status Codes

- 200 OK – ok, registration email has been sent
- 400 Bad Request – email is already registered

6.6.3 Reset password

The password reset process is simple:

- User clicks 'reset my password' and enters email
- Email is sent to the email address (1)
- User clicks on link in received email
- Backend checks token in url, sets session flag and forwards to frontend
- User enters new password and submits new password
- User is now logged in and the old password has been replaced

POST /v1/account/reset-password
Send a password reset request. Matches (1) in the description above.

Status Codes

- 200 OK – ok

GET /v1/account/reset-password?(string: *token*)
Consume the reset token and return a new, flagged, session id. Forwards to frontend.

Parameters

- **token** (*string*) – a querystring value of the reset token found in the email

Status Codes

- 302 Found – ok, forwarding to frontend
- 400 Bad Request – invalid password reset token

Only requests with an attached session id that is considered authenticated (i.e. after *Authenticate*) are allowed access to routes below.

6.6.4 Check login status

GET /v1/account/login

Test if session is authenticated/user is logged in.

Status Codes

- 200 OK – ok logged in
- 401 Unauthorized – no not logged in

6.6.5 Get friends of a user

GET /v1/account/friends

Parameters

- **email** (*String*) – entry's unique email

```
["turtle@rock.gov", "zebra@afri.ca"]
```

Response JSON Object

- **emails** (*array*) – an array of emails related to the users email including the users email.

6.6.6 Get collections

GET /v1/account/collections

Query a list of collections that the currently authenticated user is a member of.

Parameters

- **project** (*String*) – include only collections in this project

Response Headers

- Content-Type – application/json

```
[ { "name": "rick's best systems", "id": 2 } ]
```

Response JSON Array of Objects

- **name** – non-unique name of the collection
- **id** – unique id of the collection

6.6.7 Query self

GET /v1/account/self
Get an at-a-glance snapshot of stats and data about the current user.

Response Headers

- Content-Type – application/json

```
{
   "email": "zoo@world.gov",
   "trust": "Admin",
   "collections": [COLLECTIONS]
   "entries": [ENTRIES]
}
```

Response JSON Object

- **email** (*string*) – user's email
- **trust** (*string*) – trust level (see *Trust*)
- **collections** (*array*) – An array of collection objects, equivalent to *Get collections*
- **entries** (*array*) – An array of approved/pending *Entry* objects this user has submitted.

6.6.8 Logout

POST /v1/account/logout
Logout this user and reset the session.

Status Codes

- 200 OK – ok

6.6.9 Delete account

POST /v1/account/delete
WARNING - Delete the currently authenticated user.

6.6.10 Change password

POST /v1/account/change-password
Change authentication password. Does not require subsequent requests to re-authenticate.

Request JSON Object

- **old** (*string*) – old password
- **new** (*string*) – new password

Status Codes

- 200 OK – ok
- 400 Bad Request – wrong old password

6.6.11 Get collection invites

GET /v1/account/invites
Query list of collections have user is invited to. Return equivalent to *Get collections*.

6.6.12 Query user by email

GET /v1/account/(**string:** *email*)
Perform *Query self* but target a specific user. Returns same output.

Parameters

- **email** (*string*) – email of user

Status Codes

- 200 OK – ok
- 400 Bad Request – invalid email

6.7 Collection

6.7.1 Create new collection

POST /v1/collection/
Create a new collection.

Parameters

- **name** (*string*) – the collection's name (doesn't have to be unique).

Status Codes

- 400 Bad Request – must provide name
- 401 Unauthorized – must be logged in to create new collections

6.7.2 Get collection graph

GET /v1/collection/(**int:** *id*)**/graph**
Query the node graph of entries and entities.

Parameters

- **id** (*int*) – collection id

```
{
   "nodes": [ENTRIES],
   "edges": [EDGES]
}
```

Response JSON Object

- **nodes** (*array*) – An array of *Entry* objects.
- **edges** (*array*) – An array of *Edge* objects.

Status Codes

- 400 Bad Request – id must be an integer
- 404 Not Found – no collection with that id exists

6.7.3 Get statistics

GET /v1/collection/(int: *id*)/stats
Query number of members and entries in this collection.

Parameters

- **id** (*int*) – collection id

```
{
    "members": 2,
    "entries": 9
}
```

Response JSON Object

- **members** (*int*) – number of users, excluding invited, that connected to this collection
- **entries** (*int*) – number of entries that are connected to this collection

Status Codes

- 400 Bad Request – id must be an integer
- 404 Not Found – no collection with that id exists

6.7.4 Get collection project

GET /v1/collection/(int: *id*)/project
Query the project this collection extends.

Parameters

- **id** (*int*) – collection id

```
{
   "name": "serp",
   "link": "http://serpconnect.cs.lth.se"
}
```

Status Codes

- 400 Bad Request – id must be an integer
- 404 Not Found – no collection with that id exists

6.7.5 Get entries

GET /v1/collection/(int: *id*)/entries
Query entries in this collection.

Parameters

- **id** (*int*) – collection id

```
[Entry, Entry, ..., Entry]
```

Response JSON Array of Objects

- **Entry** – An *Entry* object.

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 404 Not Found – no collection with that id exists

Only requests with an attached session id, where the user is directly connected to the specified collection, are allowed access to these routes.

6.7.6 Accept an invite

POST /v1/collection/(int: *id*)/accept

Accept an invitation to join a specific collection.

Parameters

- **id** (*int*) – collection id

Status Codes

- 400 Bad Request – must provide id, id must be an integer, must be invited to that exception
- 404 Not Found – no collection with that id exists

Only requests with an attached session id, where the user is directly connected to the specified collection, are allowed access to these routes.

6.7.7 Send an invite

POST /v1/collection/(int: *id*)/invite

Invite a user to a collection.

Parameters

- **id** (*int*) – collection id

Request JSON Object

- **name** (*string*) – name of the collection

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.8 Leave a collection

POST /v1/collection/(**int:** *id*)**/leave**

Leave the collection.

Parameters

- **id** (*int*) – collection id

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.9 Remove an entry

POST /v1/collection/(**int:** *id*)**/removeEntry**

Remove an entry from the collection. If the entry isn't included in any other collections it is removed.

Parameters

- **id** (*int*) – collection id

Request JSON Object

- **entryId** (*int*) – id of entry to remove

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.10 Add an existing entry

POST /v1/collection/(**int:** *id*)**/addEntry**

Add an existing entry to the collection. This will copy the specified entry. The classifications where the facet exists in both taxonomies are copied.

Parameters

- **id** (*int*) – collection id

Request JSON Object

- **entryId** (*int*) – id of entry to add

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.11 Get members of a collection

GET /v1/collection/(**int:** *id*)**/members**
Query members in this collection.

Parameters

- **id** (*int*) – collection id

```
[User, ..., User]
```

Response JSON Array of Objects

- **User** – An *Account* object.

Status Codes

- 400 Bad Request – must provide id, id must be an integer
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.12 Get the extended taxonomy

GET /v1/collection/(**int:** *id*)**/taxonomy**
Query the extended taxonomy of this collection. *Facet* objects returned by this query will reference the standard serp taxonomy, which must be queried separately.

Parameters

- **id** (*int*) – collection id

```
{
   "version": 0,
   "taxonomy": [FACETS]
}
```

Response JSON Object

- **version** (*integer*) – Version identifier. Important for updating the taxonomy.
- **taxonomy** (*array*) – The *Facet* nodes of the extended taxonomy.

Status Codes

- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.13 Update the extended taxonomy

PUT /v1/collection/(**int:** *id*)**/taxonomy**
Update the extended taxonomy. The request will only pass if the version is >= (greater than or equal to) the currently stored version.

Parameters

- **id** (*int*) – collection id

```
{
   "version": 0,
   "taxonomy": [FACETS]
}
```

Request JSON Object

- **version** (*integer*) – Reference to the version this extension is based on.
- **taxonomy** (*array*) – The *Facet* nodes of the extended taxonomy.

Status Codes

- 400 Bad Request – illegal json, out of date version
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.14 Reclassify some entities

POST /v1/collection/(**int:** *id*)**/reclassify**

Replace old facets with new facets for some entities.

Parameters

- **id** (*int*) – collection id

```
{
   "oldFacetId": "PEOPLE",
   "newFacetId": "STRANGE-PEOPLE",
   "entities": [213, 255]
}
```

Request JSON Object

- **oldFacetId** (*string*) – The facet id that is to be replaced.
- **newFacetId** (*string*) – The replacement facet id.
- **entity** (*array*) – ids of the entities that are to be reclassified.

Status Codes

- 400 Bad Request – illegal json
- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.15 Get all the entities

GET /v1/collection/(**int:** *id*)**/entities**
Get all the entities.

Parameters

- **id** (*int*) – collection id

```
[
   {
      "id": 222,
      "text": "Regression testing"
   }
]
```

Response JSON Array of Objects

- **id** – id of the entity
- **text** – user text of the entity

Status Codes

- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.7.16 Query the classification

GET /v1/collection/(**int:** *id*)**/classification**
Get all the entities grouped by taxonomy facet.

Parameters

- **id** (*int*) – collection id

```
[
   {
      "facetId": "PEOPLE",
      "text": ["Shifty chimpanzees", "Rectangular red birds"]
   }
]
```

Response JSON Array of Objects

- **facetId** – id of the *Facet*
- **text** – text of the entities classified with this facet

Status Codes

- 401 Unauthorized – must be logged in
- 403 Forbidden – must be a member of the collection
- 404 Not Found – no collection with that id exists

6.8 Admin

Only requests with an attached session id, where user's trust level is Admin, are allowed access to these routes.

GET /v1/admin
Check if current user (via session token) is an admin.

Status Codes

- 200 OK – user is an admin
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

GET /v1/admin/pending
Get all pending entries.

```
[Entry, Entry, ..., Entry]
```

Response JSON Array of Objects

- **Entry** – An *Entry* object.

Status Codes

- 200 OK – ok, return pending entries
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

GET /v1/admin/collections
Get all collections that the admin is NOT member of

```
[Collection, Collection, ..., Collection]
```

Response JSON Array of Objects

- **Collection** – A *Collection* object.

Status Codes

- 200 OK – ok, return collections
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

POST /v1/admin/delete-collection
Delete a collection

Parameters

- **entry** (*int*) – ID of collection to delete.

Status Codes

- 200 OK – ok, collection got deleted
- 400 Bad Request – entry is not an int
- 401 Unauthorized – user is not logged in

- 403 Forbidden – user is not an admin
- 404 Not Found – no such collection exists

GET /v1/admin/collections-owned-by
Return names of all collections user is owner of

Parameters

- **email** – email of the user

Status Codes

- 200 OK – ok, return collections
- 400 Bad Request – no email was given
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

POST /v1/admin/accept-entry
Accept a pending entry.

Parameters

- **entry** (*int*) – ID of entry to accept.

Status Codes

- 200 OK – ok, entry is approved
- 400 Bad Request – entry is not an int
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin
- 404 Not Found – no such entry exists

POST /v1/admin/reject-entry
Reject a pending entry.

Parameters

- **entry** (*int*) – ID of entry to reject.

Status Codes

- 200 OK – ok, entry is rejected
- 400 Bad Request – entry is not an int
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin
- 404 Not Found – no such entry exists

POST /v1/admin/delete-user
Delete a user with a given email

Parameters

- **email** – email of the user to be deleted

Status Codes

- 200 OK – ok, user got deleted

- 400 Bad Request – no email was given
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

POST /v1/admin/delete-entry

Delete entry with a given entry id

Parameters

- **entryId** – id of the entry

Status Codes

- 200 OK – ok, entry got deleted
- 400 Bad Request – entry is not an int
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin
- 404 Not Found – no such entry exists

PUT /v1/admin/set-trust

Set trust level of a specific user.

Parameters

- **email** (*string*) – Email of user affected user.
- **trust** (*string*) – New trust level (Admin, Verified, User, Registered, Unregistered).

Status Codes

- 200 OK – ok, user has new trust level
- 400 Bad Request – invalid trust level, must provide email, must provide trust, no such user exists
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

GET /v1/admin/users

Get all users.

```
[User, User, ..., User]
```

Response JSON Array of Objects

- **User** – An *Account* object.

Status Codes

- 200 OK – ok, return users
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

GET v1/admin/is-collection-owner

param id id of the collection

type id int

Return true if the admin is owner of the collection

Status Codes

- 200 OK – ok, return boolean
- 401 Unauthorized – user is not logged in
- 403 Forbidden – user is not an admin

CHAPTER 7

Documentation

Most documentation is written in reStructuredText but it is ok to use markdown as well. Markdown rendering uses recommonmark.

- Install pip: instructions
- `pip install -r requirements.txt`
- Use entr to watch files, rebuild and run a webserver:
  - `cd docs/ && find ./ | entr -d -r 'sphinx-build . docsbin/ && cd docsbin/ && python -SimpleHTTPServer`
  - or `cd docs/ && make watch`

CHAPTER 8

Sessions

Some notes on session management:

- A session is mainly a cookie (`JSESSIONID`) that has some data on the server
- Data on server is so far only “email” _or_ “resetemail” - used for authentication
- Server uses in-memory session store (sessions are destroyed on reboot/update)

Pippo handles all session stuff (store, creation and destruction), and we only interact with the session by invoking `rc.setSession("..", "..")` or `rc.getSession("...")`.

8.1 Domains

The connect project assumes a two-domain setup:

- A web server for serving the website itself (`domain.xyz`)
- A web server running connect (`api.domain.xyz`)

This means that all API requests are cross-domain-resource-sharing (CORS) requests.

CHAPTER 9

Trust

All accounts have a trust level that is used throughout the API to determine authorization:

- Admin (99999): Can do anything, really
- Verified (9999): Can submit entries directly, bypassing the approval/rejection phase
- User (999): Can submit entries, but they must be approved/rejected by an admin
- Registered (99): Can create collections
- Unregistered (9): ??

A user is initially considered as *registered* until they have verified their email, at which point they are automatically promoted to *user* status. Only admins can promote accounts to *verified* or *admin* status (actually, they can set trust level freely).

CHAPTER 10

Collections

A collection is a neo4j object with a `:collection` label. All users that are members of the collection has a relation of type `:MEMBER_OF` pointing to the collection. The user who created the collection is considered the owner and has an additional relationship with type `:OWNER`. It is possible to invite any email to a collection.

10.1 Inviting existing members

If the email already exists in the database an email will be sent and accept/reject links will be added to user's invitations page.

10.2 Inviting non-existing members

If the email doesn't exist in the database an email will be sent to the email asking the owner to create an account. If the person creates an account with the invited email within 1 week the user will have a pending invitation to the collection at the users pending invitations page.

This is achieved by creating a temporary user with the email-address and with trust level unregistered. This temporary user has the pending invite linked to its account. If the user ever registers with that email the users account credits will be merged with the temporary account. A periodic thread will run every 12 hours and clean up unregistered users which are older than one week old to avoid flooding the database (*CleanupUsers.java*).

10.3 Invite responses

When a user accepts or rejects an invitation the user who invited the new user will get an email of which action was taken. If a temporary user gets deleted it will send a reject email.

10.4 Leaving a collection

A collection member can leave the collection voluntarily or be kicked by the collection owner.

If a collection owner leaves the collection the collection and all related relations are destroyed.

- The collection node itself is detached and deleted.
- Pending invites (relations) to that collection are deleted.
- Entries in the collection are removed and will be deleted if they no longer are attached to any collection.

10.5 Extending the taxonomy

The purpose of extending the taxonomy is to classify entities in a more detailed way than what can be achieved with the base taxonomy. This can only be done by the owner of a collection and is done on the search page by filtering by a specific collection and then clicking on one of the nodes of the taxonomy at the top. From there, the user can add and remove leaves and change how the entities are classified.

CHAPTER 11

Import

Entries can be imported from a CSV or json file. On the submit page click on the button “Import” and select a file. A new collection will be made when importing the file and a name for that collection has to be specified.

11.1 CSV

In addition to choosing a collection name, whether Research or Challenge entries are to be imported has to be specified. Another thing that can be specified is what delimiter should be used for the CSV file. Default is comma, other choices are semi-colon, colon, tab, and many more.

A taxonomy leaf delimiter should also be specified. The delimiter is used to separate the leaves for a taxonomy node. For example, if a cell looks like (test|test2) and gets mapped to some taxonomy node and the leaf delimiter is “|”, the value of that entry will be an array consisting of [“test, “test2”].

The first row in the CSV, meaning the headers, will be mapped to the following attributes:

- Reference
- DOI
- Description
- Contact
- Date
- Intervention
- Solving
- Adapting
- Assessing
- Improving
- Planning

- Design
- Execution
- Analysis
- People
- Information
- Sut
- Other

On the left hand side of each node, there is a dropdown menu. This is used to specify how the entries should be mapped. If the alternative "only if related free-text examples are extracted" is selected, only the attributes that correspond to non-empty values in the CSV will be selected for that entry. If the alternative "for all entries" is selected, that attribute will be selected for all of the entries in the following way: If the cell is empty, the value will be "unspecified" and otherwise it will be the value that was in the cell.

On the right hand side of each node, there is a dropdown menu and a "+" icon. The dropdown is used to specify which header to map to the current node. If the "+" is pressed, the "+" will be turned into a "-" and another dropdown menu and "+" icon will appear. This means that a node can be mapped to multiple columns in the CSV. For the nodes that are not part of the taxonomy, the string values in the different columns will be concatenated. If the node is a taxonomy leaf, the string values in the different columns will be added to an array. The dropdown menus can be discarded by pressing the "-" next to a dropdown menu.

If the required attributes (e.g. Reference for research entries and Description for challenge entries) are not entered they will still get the value "unspecified" so that the entries are able to be submitted from the queue (since those attributes can't be empty). They can be edited later.

A json object will be made for every row in the CSV according to the selected mapping. Every column in each row (or, a combination of columns depending on how they were mapped) will correspond to a part of the json object. For example, if a CSV header called "Abstract" was mapped to "description", the "description" field of the json objects will get the value of the cell in the columns under the CSV header "Abstract".

The json objects will then be converted to entries and be put in the queue.

11.2 JSON

Importing the json file is a lot simpler than importing the CSV file, only the collection name has to be specified.

The file content has to be one list of json objects and in the same way as for the CSV, the json objects will then be converted to entries and queued. If there are invalid entries (e.g. no Reference for research entries or no Description for challenge entries), a message will pop up saying which entries were invalid (or how many, depending on how many there are) and the user will be asked if he/she wants to continue adding the valid entries or exiting.

CHAPTER 12

Export

A brief description on how to export entries from the system to a CSV file.

12.1 Method

For each collection on the profile page, there exists an option to export that collection. To do so, the user has to specify a filename and also a CSV delimiter and a taxonomy leaf delimiter.

The CSV delimiter is straight-forward and specifies what should separate the values in the CSV.

The taxonomy leaf delimiter is used to separate the leaves for the taxonomy nodes. For example, if some taxonomy node has the value [“test”, “test2”], and the leaf delimiter is set to “|”, the value of that cell in the CSV will be (test|test2).

The file will, as is standard, be downloaded to the users Downloads folder.

CHAPTER 13

Admin

13.1 Delete user

When logged in as an admin go to profile->users. To delete a user press the cross next to the users account level. When deleting a user a confirmation box will appear. If the user is owner of any collection a second confirmation box will appear warning which collections will be destroyed when deleting this user.

13.2 Delete collection

When logged in as an admin go to profile->all collections. To delete the collection press the delete button. A confirmation box will appear before the deletion is complete.

13.3 Delete entry

When logged in as an admin go to search. Press any entry that should be deleted. In the information box of the entry press the delete button. A confirmation box will appear.

CHAPTER 14

Import

How to add entries to the system as a user.

14.1 Submitting

Add entries via the submit page.

14.2 Importing

Bulk importing entries.

CHAPTER 15

Installing

How to setup the server.

15.1 Prerequisites

- An email account, credentials and server settings.

15.2 Compiling

- *git clone https://github.com/serpconnect/backend*
- *cd backend*
- *mvn compile package*
- The compiled server is now in *target/connect-X.Y.Z.zip*

15.3 Deploying

- Assuming *connect-X.Y.Z.zip* and *application.properties* are in current dir.
- *unzip connect-X.Y.Z.zip*
- *cp application.properties connect-X.Y.Z*
- *cd connect-X.Y.Z*
- *java -jar connect.jar*
- The server is now running using the external configuration. If no config. file is present the embedded is used instead.

CHAPTER 16

Performance

Benchmarks for performance regressions are yet to be written. The current performance is ok for a modest number of users. As the overhead incurred by jcypher is very big it is a worthwhile investment to keep the number of API requests as low as possible.

16.1 Frontend

The performance of the website is largely determined by database size. Especially home, explore and search pages are sensitive to database size. All pages that fire many small requests will benefit from a faster neo4j driver. Other pages could benefit from moving computation or filterting from the client to the server (e.g. search, graph generation).

16.2 Backend

The backend has been profiled during a number of requests to different endpoints and the results all point to the neo4j driver jcypher. Both the driver itself and its dependencies add a big (~100-600ms) overhead in request processing, i.e. time spent before the request hits the wire.

An example:

Finding a user by an email address is a common operation. The cypher query: `MATCH (u:user {email: {addr}}) RETURN u`. The test was carried out against an endpoint that only did this query. The timings below are the amount of milliseconds spent to perform the database query.

- jcypher/java: `11ms`
- node-neo4j/nodejs: `3ms`

HTTP Routing Table

/v1