INDRA Database Documentation

Release 1.0.0

P. A. Greene, J. A. Bachman, B. M. Gyori

Feb 18, 2020

Contents

1 Knowledge sources **3**
- 1.1 Daily Readers . . . 3
- 1.2 Other Readers . . . 3
- 1.3 Other Databases . . . 3

2 Knowledge Assembly **5**

3 Access **7**

4 Further INDRA Database documentation **9**
- 4.1 License and funding . . . 9
- 4.2 INDRA Database modules . . . 9
  - 4.2.1 The Client . . . 9
  - 4.2.2 Utilities . . . 11
  - 4.2.3 Pipeline Managers . . . 11
  - 4.2.4 Database Integrated Reading Tools . . . 12
  - 4.2.5 Database Schemas . . . 13
  - 4.2.6 Some Miscellaneous Modules . . . 13

5 INDRA Database REST API **15**
- 5.1 INDRA Database REST API . . . 15
  - 5.1.1 The Statement Endpoints . . . 16
  - 5.1.2 Curation . . . 18
  - 5.1.3 Usage examples . . . 19

6 Indices and tables **23**

The INDRA (Integrated Network and Dynamical Reasoning Assembler) Database is a framework for creating, maintaining, and accessing a database of content, readings, and statements. This implementation is currently designed to work primarily with Amazon Web Services RDS running Postrgres 9+. Used as a backend to INDRA, the INDRA Database provides a systematic way of scaling the knowledge acquired from other databases, reading, and manual input, and puts that knowledge at your fingertips through a direct Python client and a REST api.

CHAPTER 1

Knowledge sources

The INDRA Database currently integrates and distills knowledge from several different sources, both biology-focused natural language processing systems and other pre-existing databases

1.1 Daily Readers

We have read all available content, and every day we run the following readers:

- REACH
- Sparser

on the latest content drawn from:

- PubMed - ~19 million abstracts and ~29 million titles
- PubMed Central - ~2.7 million fulltext
- Elsevier - ~0.7 million fulltext (requires special access)

1.2 Other Readers

We also include more or less static content extracted from the following readers:

- TRIPS
- RLIMS-P

1.3 Other Databases

We include the information from these pre-existing databases:

- Pathway Commons database

- BEL Large Corpus
- SIGNOR
- BioGRID
- TAS
- TRRUST
- PhosphoSitePlus
- Causal Biological Networks Database

These databases are retrieved primarily using the tools in `indra.sources`. The statements extracted from all of these sources are stored and updated in the database.

CHAPTER 2

Knowledge Assembly

The INDRA Database uses the powerful internal assembly tools available in INDRA but implemented for large-scale incremental assembly. The resulting corpus of cleaned and de-duplicated statements, each with fully maintained provenance, is the primary product of the database.

For more details on the internal assembly process of INDRA, see the INDRA documentation.

CHAPTER 3

Access

The content in the database can be accessed by those that created it using the `indra_db.client` submodule. This repo also implements a REST API which can be used by those without direct acccess to the database. For access to our REST API, please contact the authors.

The INDRA database only works for Python 3 (tested in 3.5 and 3.6).

First, install INDRA, then simply clone this repo, and make sure that it is visible in your `PYTHONPATH`.

The development of INDRA DB is funded under the DARPA Communicating with Computers program (ARO grant W911NF-15-1-0544).

CHAPTER 4

Further INDRA Database documentation

4.1 License and funding

The INDRA was developed with funding from ARO grant W911NF-14-1-0397, "Programmatic modelling for reasoning across complex mechanisms" under the DARPA Big Mechanism program, and the INDRA database was developed as an extention of that core project. Work has continued under W911NF-14-1-0391, "Active context" under the DARPA Communicating with Computers program, and the DARPA Automated Scientific Discovery Framework project.

4.2 INDRA Database modules

4.2.1 The Client

The purpose of the client is to be the gateway for external access to the content of the databases. Here we define high level access functions for getting data out of the database in a natural way. This is where the queries used by the REST API are defined, and most users looking to access knowledge on the database should use the client if they can, as it is heavily optimized.

Our system utilizes 2 databases, one which represents the "ground truth", as we know it, and is structured naturally for performing updates on our knowledge; it will always be the most up to date. We also have a "readonly" database that we used for our outward facing services. This database is optimized for fast queries and the content in it is updated weekly. Each database has its own set of access tools.

The Principal Database Client

This is the set of client tools to access the most-nearly ground truth knowledge stored on the principal database.

Access Readings and Text Content (`indra_db.client.principal.content`)

This defines a simple API to access the content that we store on the database for external purposes.

Submit and Retrieve Curations (`indra_db.client.principal.curation`)

On our services, users have the ability to curate the results we present, indicating whether they are correct or not, and how they may be incorrect. The API for adding and retrieving that input is defined here.

Get Raw Statements (`indra_db.client.principal.raw_statements`)

Get the raw, uncleaned and un-merged Statements based on agent and type or by paper(s) of origin.

The Readonly Client

Here are our primary tools intended for retrieving Statements, in particular Pre-Assembled (PA) Statements, from the readonly database. This is some of the most heavily optimized access code in the repo, and is the backbone of most external or outward facing applications.

The readonly database, as the name suggests, is designed to take only read requests, and is updated via dump only once a week. This allows users of our database to access it even as we perform daily updates on the principal database, without worrying about queries interfering.

Get Pre-Assembled Statements (`indra_db.client.readonly.pa_statements`)

Here are the tools used to get PA Statements from the readonly database, with the goal of retrieving at least 1,000 Statements with 10 evidence each in under 30 seconds.

Get Simple Interactions from Metadata (`indra_db.client.readonly.interactions`)

This provides an API to get somewhat less detailed data than above, using just the metadata of the database (not looking into the Statement JSONs), but is much faster. These tools can be sufficient if, for example, all that is needed is an interactome.

Miscellaneous Client APIs (Mostly Deprecated)

There are some, generally archaic, client functions which use both readonly and principal resources. I make no guarantee that these will work.

Get Datasets (`indra_db.client.datasets`)

An early attempt at something very like the `indra_db.client.readonly.interactions` approach to getting superficial data out of the database.

Get Statements (`indra_db.client.statements`)

The first round of tools written to get Statements out of the database, utilizing far too many queries and taking absurdly long to complete. Most of their functions have been outmoded, with the exception of getting PA Statements from the principal database, which (as of this writing) has yet to be implemented.

4.2.2 Utilities

Here live the more mundane and backend utilities used throughout other modules of the codebase, and potentially elsewhere, although they are not intended for external use in general. Several more-or-less bespoke scripts are also stored here.

Database Session Constructors (`indra_db.util.constructors`)

Constructors to get interfaces to the different databases, selecting among the various physical instances defined in the config file.

Scripts to Get Content (`indra_db.util.content_scripts`)

General scripts for getting content by various IDs.

Distilling Raw Statements (`indra_db.util.distill_statements`)

Do some pre-pre-assembly cleansing of the raw Statements to account for various kinds of duplicity that are artifacts of our content collection and reading pipelines rather than representing actually duplicated knowledge in the literature.

Script to Create a SIF Dump (`indra_db.util.dump_sif`)

Create an interactome from metadata in the database and dump the results as a sif file.

General Helper Functions (`indra_db.util.helpers`)

Functions with broad utility throughout the repository, but otherwise miscellaneous.

Routines for Inserting Statements and Content (`indra_db.util.insert`)

Inserting content into the database can be a rather involved process, but here are defined high-level utilities to uniformly accomplish the task.

4.2.3 Pipeline Managers

This module contains the pipelines used to update content and knowledge in the database, and move or transform that knowledge on a regular basis.

Content Manager (indra_db.managers.content_manager)

The Content Managers, as the name suggests, manage the text content that is stored in the database. A parent class is defined, and managers for different sources (e.g. PubMed) can be defined by inheriting from this parent. This file is also used as the shell command to run updates of the content.

Reading Manager (indra_db.managers.reading_manager)

The Reading Managers handle the reading of the text contend and the processing of those readings into statements. As with Content Managers, different reading pipelines can be handled by defining children of a parent class.

PreAssembly Manager (indra_db.managers.preassembly_manager)

The Preassembly Manager performs the complex process of preassembly, where after cleaning the raw extractions and fixing groundings and sites, the numerous duplicate Statements extracted from databases and literature are distilled into a corpus of unique statements, with links back to the raw Statements, and their history and provenance.

Knowledge Base Manager (indra_db.managers.knowledgebase_manager)

The INDRA Databases also derives much of its knowledge from external databases and other resources not extracted from plain text, referred to in this repo as "knowledge bases", so as to avoid the ambiguity of "database". This manager handles the updates of those knowledge bases, each of which requires different handling.

Readonly Manager (indra_db.managers.readonly_manager)

This handles the generation of the content for the readonly database from the principal database.

4.2.4 Database Integrated Reading Tools

Here are defined the procedures for reading content on the database, stashing the reading outputs, and producing statements from the readings, and inserting those raw statements into the database.

The Database Readers (indra_db.reading.read_db)

A reader is defined as a python class which implements the machinery needed to process the text content we store, read it, and extract Statements from the reading results, storing the readings along the way. The reader must conform to a standard interface, which then allows readers to be run in a plug-and-play manner.

The Database Script for Running on AWS (indra_db.reading.read_db_aws)

This is the script used to run reading on AWS Batch, generally run from an AWS Lambda function.

The Database Reporter (indra_db.reading.report_db_aws)

Create an object that is used to aggregate and report on the reading process, allowing for effective monitoring.

A Class to Manage and Monitor AWS Batch Jobs (`indra_db.reading.submit_reading_pipeline`)

Allow a manager to monitor the Batch jobs to prevent runaway jobs, and smooth out job runs and submissions.

4.2.5 Database Schemas

Here are defined the schemas for the principal and readonly databases, as well as some useful mixin classes.

Principal Database Schema (`indra_db.schemas.principal_schema`)

Defines the *get_schema* function for the principal database, which represents the "ground truth" of the knowledge we aggregate.

Readonly Database Schema (`indra_db.schemas.readonly_schema`)

Defines the *get_schema* function for the readonly database, which is used by external services to access the Statement knowledge we acquire.

Class Mix-ins (`indra_db.schemas.mixins`)

This defines class mixins that are used to add general features to SQLAlchemy table objects via multiple inheritance.

Indexes (`indra_db.schemas.indexes`)

This defines the classes needed to create and maintain indices in the database, the other part of the infrastructure of which is included in the *IndraDBTable* class mixin definition.

4.2.6 Some Miscellaneous Modules

Here are some modules and files that live on their own, and don't fit neatly into other categories.

Low Level Database Interface (`indra_db.databases`)

The Database Manager classes are the lowest level interface with the database, implemented with SQLAlchemy, providing useful short-cuts but also allowing full access to SQLAlchemy's API.

Belief Calculator (`indra_db.belief`)

The belief in the knowledge of a Statement is a measure of our confidence that the Statement is an accurate representation of the text, _NOT_ our confidence in the validity of what was in that text. Given the size of the content in the database, some special care is needed when calculating this value, which depends heavily on the support relations between pre-assembled Statements.

CHAPTER 5

INDRA Database REST API

5.1 INDRA Database REST API

The INDRA Database software has been developed to create and maintain a database of text references, content, reading results, and ultimately INDRA Statements extracted from those reading results. The software also manages the generation and update process of cleaning, deduplicating, and finding relations between the raw Statement extractions, into what are called pre-assembled Statements. All INDRA Statements can be represented as JSON, which is the format returned by the API.

This web API provides the code necessary to support a REST service which allows access to the pre-assembled Statements in a database. The system is still under heavy development so capabilities are always expanding, but as of this writing, the API supports:

- *''statements/from_agents'* <#from-agents>'_, getting Statements by agents, using various ids or names, by statement type (e.g. Phosphorylation), or
- *''statements/from_hash'* <#from-hash>'_ and *''statements/from_hashes'* <#from-hashes>'_, getting Statements by Statement hash, either singly or in batches, and
- *''statements/from_papers'* <#from-papers>'_, getting Statements using the paper ids from which they were extracted, and
- *''curation/submit/<hash>'* <#curation>'_ you can also curate Statements, helping us improve the quality and accuracy of our content.

As mentioned, the service is changing rapidly, and this documentation may at times be out of date. For the latest, check github or contact us.

You need the following information to access a running web service:

- The address of the web service (below shown with the placeholder `api.host`)
- An API key which needs to be sent in the header of each request to the service, or any other credentials that are implemented.

If you want to use our implementation of the web API, you can contact us for the path and an API key.

The code to support the REST service can be found in `api.py`, implemented using the Flask Python package. The means of hosting this api are left to the user. We have had success using Zappa and AWS Lambda, and recommend it for a quick and efficient way to get the API up and running.

5.1.1 The Statement Endpoints

For all queries, an API key is required, which is passed as a parameter `api_key` to any/all queries. Below is detailed documentation for the different endpoints of the API that return statements (i.e. those with the root `/statements`). All endpoints that return statements have the following options to control the size and order of the response:

- **`format`**: The endpoint is capable of returning both HTML and JSON content by setting the format parameter to "html" or "json", respectively. See the *section on output formats* below.
- **`max_stmts`**: Set the maximum number of statements you wish to receive. The REST API maximum is 1000, which cannot be overridden by this argument (to prevent request timeouts).
- **`ev_limit`**: The default varies, but in general the amount of Evidence returned for each statement is limited. A single statement can have upwards of 10,000 pieces of evidence, so this allows queries to be run reliably. There is no limitation on this value, so use with caution. Setting too high a value may cause a request to time out or be too large to return.
- **`best_first`**: This is set to "true" by default, so statements with the most evidence are returned first. These are generally the most reliable, however they are also generally the most canonical. Set this parameter to "false" to get statements in an arbitrary order. This can also speed up a query. You may however find you get a lot of low-quality content.

The output formats

The output format is controlled by the **`format`** option described above, with options to return JSON or HTML.

JSON: The default value, intended for programmatic use, is "json". The JSON that is returned is of the following form (with many made-up but reasonable numbers filled in):

```
{
  "statements": {  # Dict of statement JSONs keyed by hash
    "12345234234": {...},  # Statement JSON 1
    "-246809323482": {...},  # Statement JSON 2
    ...},
  "offset": 2000,  # offset of SQL query
  "evidence_limit": 10,  # evidence limit used
  "statement_limit": 1000,  # REST API Limit
  "evidence_totals": {  # dict of available evidence for each statement keyed by hash
    "12345234234": 7657,
    "-246809323482": 870,
    ...},
  "total_evidence": 163708, # The total amount of evidence available
  "evidence_returned": 10000  # The total amount of evidence returned
}
```

where the `"statements"` element contains a dictionary of INDRA Statement JSONs keyed by a shallow statement hash (see *here* for more details on these hashes). You can look at the JSON schema on github for details on the Statement JSON. To learn more about INDRA Statements, you can read the documentation.

HTML: The other `format` parameter option, designed for easier manual usage, is "html". The service will then return an HTML document that, when opened in a web browser and if logged in, provides a graphical user interface for viewing and curating statements at the evidence level. The web page also allows you to easily query for more

evidence for a given statement. Documentation for the html output (produced by INDRA's HTML assembler) can be found here.

Get Statements by agents (and type): `GET api.host/statements/from_agents`

This endpoint allows you to get statements filtering by their agents and the type of Statement. The query parameters are as follows:

- **subject**, **object**: The HGNC gene symbol of the subject or object of the Statement. **Note**: only one of each of `subject` and `object` will be accepted per query.
  - Example 1: if looking for Statements where MAP2K1 is a subject (e.g. "What does MAP2K1 phosphorylate?"), specify `subject=MAP2K1` as a query parameter
  - Example 2: if looking for Statements where MAP2K1 is the subject and MAPK1 is the object, add both `subject=MAP2K1` and `object=MAPK1` as query parameters.
  - Example 3: you can specify the agent id namespace by appending `@<namespace>` to the agent id in the parameter, e.g. `subject=6871@HGNC`.
- **agent***: This parameter is used if the specific role of the agent (subject or object) is irrelevant, or the distinction doesn't apply to the type of Statement of interest (e.g. Complex, Translocation, ActiveForm). **Note**: You can include as many `agent*` queries as you like, however you will only get Statements that include all agents you query, in addition to those queried for `subject` and `object`. Furthermore, to include multiple agents on our particular implementation, which uses the AWS API Gateway, you must include a suffix to each agent key, such as `agent0` and `agent1`, or else all but one agent will be stripped out. Note that you need not use integers, you can add any suffix you like, e.g. `agentOfDestruction=TP53` would be entirely valid.
  - Example 1: To obtain Statements that involve SMAD2 in any role, add `agent=SMAD2` to the query.
  - Example 2: As with `subject` and `object`, you can specify the namespace for an agent by appending `@<namespace>` to the agent's id, e.g. `agent=ERK@TEXT`.
  - Example 3: If you wanted to query multiple statements, you could include `agent0=MEK@FPLX` and `agent1=ERK@FPLX`. Note that the value of the integers has no real bearing on the ordering, and only serves to make the agents uniquely keyed. Thus `agent1=MEK@FPLX` and `agent0=ERK@FPLX` will give exactly the same result.
- **type**: This parameter can be used to specify what type of Statement of interest (e.g. Phosphorylation, Activation, Complex).
  - Example: To answer the question "Does MAP2K1 phosphorylate MAPK1?" the parameter `type=Phosphorylation` can be included in your query. Note that this field is not case sensitive, so `type=phosphorylation` would give the same result.

Get a Statement by hash: `GET api.host/statements/from_hash/<hash>`

INDRA Statement objects have a method, `get_hash`, which produces hash from the content of the Statement. A shallow hash only considers the meaning of the statement (agents, type, modifications, etc.), whereas a deeper hash also considers the list of evidence available for that Statement. The shallow hash is what is used in this application, as it has the same uniqueness properties used in deduplication. As mentioned above, the Statements are returned keyed by their hash. In addition, if you construct a Statement in python, you may get its hash and quickly find any evidence for that Statement in the database.

This endpoint has no extra parameters, but rather takes an extension to the path. So, to look up the hash 123456789, you would use `statements/from_hash/123456789`.

Because this only returns one statement, the default evidence limit is extremely generous, set to 10,000. Thus you are most likely to get all the evidence for a given statement this way. As described above, the evidence limit can also be raised, at the risk of a timed out request.

Get Statements from many hashes: `POST api.host/statements/from_hashes`

Like the previous endpoint, this endpoint uses hashes to retrieve Statements, however instead of only being allowed one at a time, a bach of hashes may be sent as json data. Because data is sent, this is a POST request, even though you are in practice "getting" information. There are no special parameters for this endpoint. The json data should be formatted as:

```
{"hashes": [12345, 246810]}
```

with up to 1,000 hashes given in the list.

Get Statements from paper ids: `POST api.host/statements/from_papers`

Using this endpoint, you can pretend you have a fleet of text extraction tools that run in seconds! Specifically, you can get the INDRA Statements with evidence from a given list of papers by passing one of the ids of those papers. As with the above method, the fact that data (paper ids) is send requires this to be a POST request. The papers ids should be formatted as:

```
{"ids": [{"id": "12345", "type": "pmid"},
         {"id": "234525", "type": "tcid"},
         {"id": "PMC23423", "type": "pmcid"}]}
```

a list of dicts, each containing id type and and id value.

5.1.2 Curation

Because the mechanisms represented by our Statements come in large part from automatic extractions, there can often be errors. For this reason, we always provide the sentences from which a Statement was extracted (if we extracted it, some of our content comes from other curated databases), as well as provenance to lead back to the content (abstract, full text, etc.) that was read, which allows you to use your own judgement regarding the validity of a Statement.

If you find something wrong with a Statement, you can use this curation endpoint to record your observation. This will not necessarily have any immediate effect on the output, however, over time it will help us improve the readers we use, our methods for extracting Statements from those reader outputs, could help us filter erroneous content, and will help us improve our pre-assembly algorithms.

Further instruction on curation best practices can be found here.

Curate statements: `POST api.host/curation/submit/<hash>`

If you wish to curate a Statement, you must first decide whether you are curating the Statement as generally incorrect, or whether a particular sentence supports a given Statement. This is the "level" of your curation:

- **pa**: At this level, you are curating the knowledge in a **p**re-**a**ssembled Statement. For example, if a Statement indicates that "differentiation binds apoptosis", regardless of whether the reader(s) made a valid extraction, it is clearly wrong.
- **raw**: At this level, you are curating a particular raw extraction, in other words stating that an automatic reader made an error. Even more explicitly, you can judge whether the sentence supports the extracted Statement. For example the (hypothetical) sentence "KRAS was found to actively inhibit BRAF" does not support the

Statement "KRAS activates BRAF". As another example (here a grounding error), would be that the sentence "IR causes cell death", where IR is Ionizing Radiation does not support the extraction "'Insulin Receptor' causes cell death".

The two different levels also have different hashes. At the *pa* level, the hashes discussed *above* are used, as they are calculated from the knowledge contained in the statement, independent of the evidence. At the *raw* level, a different hash must be included: the `source_hash`, which identifies a specific piece of evidence, without considering the Statement extracted. Within a Statement JSON, there is a key "evidence", with a list of Evidence JSON, which includes an entry for "source_hash":

```
{"evidence": [{"source_hash": 98687578576598, ...}, ...], ...}
```

Once you know the level, and you have the correct hash(es) (the shallow pre-assembly hash and/or the source hash), you can curate a statement by POSTing a request with JSON data to the endpoint, as shown in the heading. The JSON data should contain the following fields:

- **tag**: A very short word or phrase categorizing the error, for example "grounding" for a grounding error.
- **text**: A brief description of what you think is most wrong.
- **curator**: Your name, initials, email, or other way to identify yourself. Whichever you choose, please be consistent.

Note that you can also indicate that a Statement is *correct*. In particular, if you find that a Statement has some evidence that supports the Statement and some that does not, curating examples of both is valuable. In general, flagging correct Statements can be just as valuable as flagging incorrect Statements.

5.1.3 Usage examples

The web service accepts standard HTTP requests, and any client that can send such requests can be used to interact with the service. Here we provide usage examples with the `curl` command line tool and `python` of some of the endpoints. This is by no means a comprehensive list, but rather demonstrates some of the crucial features discussed above.

In the examples, we assume the path to the web API is `https://api.host/`, and that the API key is `12345`.

`curl` is a command line tool on Linux and Mac, making it a convenient tool for making calls to this web API.

Using `curl` to query Statements about "MAP2K1 phosphorylates MAPK1":

```
curl -X GET "http://api.host/statements/from_agents?subject=MAP2K1&object=MAPK1&
↪type=phosphorylation&api_key=12345"
```


```
{
  "statements": {
    "-1072112758478440": {
      "id": "5c3dff5f-6660-4494-96d2-0142076e9b2f",
      "enz": {
        "name": "MAP2K1",
        "db_refs": {
          "UP": "Q02750",
          "HGNC": "6840"
        },
        "sbo": "http://identifiers.org/sbo/SBO:0000460"
      },
      "sbo": "http://identifiers.org/sbo/SBO:0000216",
      "evidence": [
        {
```


(continues on next page)

(continued from previous page)


```
          "source_api": "reach",
          "epistemics": {
            "section_type": null,
            "direct": true
          },
          "text": "Thus, free non visual arrestins moderately facilitate the␣
↪phosphorylation of ERK2 by MEK1.",
          "pmid": "22174878",
          "annotations": {
            "agents": {
              "raw_text": [
                "MEK1",
                "ERK2"
              ]
            },
            "content_source": "pmc_oa",
            "prior_uuids": [
              "55afb6fc-5649-4315-94bc-3ce0651fc1d3"
            ],
            "found_by": "Phosphorylation_syntax_1a_noun"
          }
        }
      ],
      "type": "Phosphorylation",
      "sub": {
        "name": "MAPK1",
        "db_refs": {
          "UP": "P28482",
          "HGNC": "6871"
        },
        "sbo": "http://identifiers.org/sbo/SBO:0000015"
      }
    }
  },
  "offset": null,
  "total_evidence": 106,
  "evidence_totals": {
    "-1072112758478440": 106
  },
  "evidence_returned": 1,
  "evidence_limit": "1",
  "statement_limit": 1000
}
```


</p> </details>

Python is another convenient way to use this web API, and has the important advantage that Statements returned from the service can directly be used directly with INDRA tools.

You can use python to get JSON Statements for the same query:

```
import requests
resp = requests.get('http://api.host/statements/from_agents',
                    params={'subject': 'MAP2K1',
                            'object': 'MAPK1',
                            'type': 'phosphorylation',
                            'api_key': 12345})
resp_json = resp.json()
```

which can now be turned into INDRA Statement objects using `stmts_from_json`:

```
from indra.statements import stmts_from_json
stmts = stmts_from_json(resp_json['statements'].values())
```

For those familiar with pre-assembled INDRA Statements, note that the `supports` and `supported_by` lists of the python Statement objects are not populated.

INDRA also supports a client to this API, which is documented in detail elsewhere, however using that client, the above query is simply:

```
from indra.sources import indra_db_rest as idbr
processor = idbr.get_statements(subject='MAP2K1', object='MAPK1', stmt_type=
↪'phosphorylation')
stmts = processor.statements
```

Where the URL and API key are located n a config file. A key advantage of this client is that queries that return more than 1000 statement are paged behind the scenes, so that all the statements which match the query are retrieved with a single command.

By setting the `format` option to `html` in the web API address, an HTML document that presents a graphical user interface when displayed in a web browser will be returned. The example below queries for statements where BRCA1 is subject and BRCA2 is object:

```
http://api.host/statements/from_agents?subject=BRCA1&object=BRCA2&api_key=12345&
↪format=html
```

The interface is restricted to users with login access. If you are not logged in, you will be prompted to so before you can view the loaded statements. Once logged in, the queried statements will be loaded and you will be able to curate statements on the level of individual evidences. Links to various source databases (depending on availability) are available for each piece of evidence to facilitate accurate curation. Find out more about the HTML interface in the HTML assembler documentation. For instructions on how to use it and more about the login restriction, see the manual.

Use curl to query for any kind of interaction between SMURF2 and SMAD2, returning at most 10 statements with 3 evidence each:

```
curl -X GET "http://api.host/statements/from_agents?agent0=SMURF2&agent1=SMAD2&api_
↪key=12345&max_stmts=10&ev_limit=3"
```

As above, in python this could be handled using the `requests` module, or with the client:

```
import requests
from indra.statements import stmts_from_json
from indra.sources import indra_db_rest as idbr

# With requests
resp = requests.get('http://api.host/statements/from_agents',
                    params={'agent0': 'SMURF2', 'agent1': 'SMAD',
                            'api_key': 12345, 'max_stmts': 10,
                            'ev_limit': 3})
resp_json = resp.json()
stmts = stmts_from_json(resp_json['statements'].values())

# With the client
stmts = idbr.get_statements(agents=['SMURF2', 'SMAD'], max_stmts=10,
                            ev_limit=3, simple_response=True)
```

Note the use of the `@FPLX` suffix to denote the namespace used in identifying the agent to query for things that inhibit MEK, using curl:

```
curl -X GET "http://api.host/statements/from_agents?object=MEK@FPLX&type=inhibition&
↪api_key=12345"
```

Python requests:

```
resp = requests.get('http://api.host/statements/from_agents',
                    params={'agent': 'MEK@FPLX', 'type': 'inhibition',
                            'api_key': 12345})
```

and INDRA's client:

```
stmts = idbr.get_statements(agents=['MEK@FPLX'], stmt_type='inhibition')
```

Query for a statement with the hash -1072112758478440, retrieving at most 1000 evidence, using curl:

```
curl -X GET "http://api.host/statements/from_hash/-1072112758478440?api_key=12345&ev_
↪limit=1000"
```

or INDRA's client:

```
stmts = idbr.get_statements_by_hash([-1072112758478440], ev_limit=1000)
```

Note that client does not actually use the same endpoint here, but rather uses the `/from_hashes` endpoint.

Get the statements from a paper with the pmid 22174878, and another paper with the doi 10.1259/0007-1285-34-407-693, first create the json file, call it `papers.json` with the following:

```
{
  "ids": [
    {"id": "22174878", "type": "pmid"},
    {"id": "10.1259/0007-1285-34-407-693", "type": "doi"}
  ]
}
```

and post it to the REST API with curl:

```
curl -X POST "http://api.host/statements/from_papers?api_key=12345" -d @papers.json -
↪H "Content-Type: application/json"
```

or just use INDRA's client:

```
stmts = idbr.get_statments_for_paper([('pmid', '22174878'),
                                      ('doi', '10.1259/0007-1285-34-407-693')])
```

Curate a Statement at the pre-assembled (pa) level for a Statement with hash -1072112758478440, using curl:

```
curl -X POST "http://api.host/curation/submit/-1072112758478440?api_key=12345" -d '{
↪"tag": "correct", "text": "This Statement is OK.", "curator": "Alice"}' -H "Content-
↪Type: application/json"
```

or INDRA's client:

```
idbr.submit_curation(-1072112758478440, 'correct', 'This Statement is OK.', 'Alice')
```

CHAPTER 6

Indices and tables

- genindex
- search