## Contents

1 Introduction 3

2 Installation 5
   2.1 Detailed instructions 5

3 Usage example 7
   3.1 1) RDBMS 7
   3.2 2) Database 7
   3.3 3) Module 7

4 API 11
   4.1 pg_jts 11
   4.2 pg_database 13
   4.3 pg_query 15

Python Module Index 17
pg_jts extracts JSON table schemas from a live PostgreSQL database.
For now please look at these slides: 20150927_talk.pdf

**TL;DR** Describing a PostgreSQL database as a JSON-table-schema allows to use tools supporting JSON-table-schema, in particular jts_erd for visualizing the database in an entity-relationship diagram.
CHAPTER 2

Installation

Beware: This software is in alpha state.
Currently there is no python package; you have to install from source.
It works with python3.4 and PostgreSQL 9.4; other versions are untested, but other minor versions of python3 and
PostgreSQL 9 are expected to work.
You need psycopg2 on your PYTHONPATH.

2.1 Detailed instructions

Prepare a virtualenv with python3:

```bash
mkdir pg_jts
cd pg_jts
virtualenv -p python3
source bin/activate
```

Install package libpq-dev and then:

```bash
pip3 install psycopg2
```

In the virtualenv root dir:

```bash
git clone https://github.com/iburadempa/pg_jts.git
```
CHAPTER 3

Usage example

3.1 1) RDBMS

Install PostgreSQL 9.4

3.2 2) Database

Either choose an existing database or create a new one like this:

```
createuser testuser -P
createdb -E utf-8 -O testuser testdb
```

Check that you can access it like this:

```
psql -W -U testuser -h 127.0.0.1 testdb
```

Create some SQL structures:

```
COMMENT ON database testdb IS 'test';
CREATE TYPE chan AS ENUM('email', 'xmpp', 'sip');
CREATE TABLE channel (id SERIAL PRIMARY KEY, channel_type CHAN, channel_attrs JSONB);
COMMENT ON TABLE channel IS 'communication channel';
COMMENT ON COLUMN channel.channel_attrs IS 'Channel attributes (specific to channel_type)';
CREATE TABLE person (id SERIAL PRIMARY KEY, name VARCHAR(100) NOT NULL, channel_id INT NULL REFERENCES channel(id));
CREATE INDEX person__name ON person (name);
COMMENT ON COLUMN person.channel_id IS 'references channel(id) 1--1..N';
CREATE TABLE software_release (id SERIAL PRIMARY KEY, software_name VARCHAR(100) NOT NULL, release_name VARCHAR(100), major INT NOT NULL, minor INT NOT NULL, patch INT NOT NULL, revision VARCHAR(50));
ALTER TABLE software_release ADD CONSTRAINT software_release__version UNIQUE(software_name, major, minor, patch);
CREATE INDEX software_release__versions2 ON software_release (major, minor);
CREATE INDEX software_release__versions3 ON software_release (major, minor, patch);
CREATE TABLE feature_change (id SERIAL PRIMARY KEY, description TEXT NOT NULL, major INT NOT NULL, minor INT NOT NULL);
COMMENT ON TABLE feature_change IS 'changes of features for software releases; (major, minor) references software_release (major, minor) 1..N--1';
```

3.3 3) Module

In the virtualenv root go to subdir pg_jts and run python3:
You will obtain a JSON representation of the database and a list of notifications. The data structure encoded as JSON looks like this:

```json
{
  'database_description': 'test',
  'database_name': 'testdb',
  'datapackages': [{
    'datapackage': 'public',
    'resources': [{
      'fields': [
        {'constraints': {'required': False}, 'default_value': 'channel_id_seq()', 'name': 'id', 'type': 'int4'},
        {'constraints': {'required': True}, 'description': 'Channel attributes (specific to channel_type)', 'name': 'channel_attrs', 'type': 'jsonb'}
      ],
      'foreignKeys': [],
      'indexes': [{
        'creation': 'CREATE UNIQUE INDEX channel_pkey ON channel USING btree (id)',
        'fields': ['id'],
        'name': 'channel_pkey',
        'primary': True,
        'unique': True
      }],
      'name': 'channel',
      'primaryKey': ['id']
    }],
    'fields': [{
      'constraints': {'required': False}, 'default_value': 'person_id_seq()', 'name': 'id', 'type': 'int4'},
      {'constraints': {'required': False}, 'name': 'name', 'type': 'varchar(100)'},
      {'constraints': {'required': True}, 'description': 'references channel(id) 1--1..N', 'name': 'channel_id', 'type': 'int4'}
    ],
    'foreignKeys': [{
      'enforced': True,
      'fields': ['channel_id'],
      'reference': {'datapackage': 'public', 'fields': ['id'], 'name': 'person_channel_id_fkey', 'resource': 'channel'}
    }],
    'indexes': [{
      'creation': 'CREATE UNIQUE INDEX person_pkey ON person USING btree (id)',
      'definition': 'btree (id)',
      'fields': ['id'],
      'name': 'person_pkey',
      'primary': True,
      'unique': True},
      {'creation': 'CREATE INDEX person__name ON person USING btree (name)',
      'definition': 'btree (name)',
      'fields': ['name'],
      'name': 'person__name',
      'primary': False,
      'unique': False}
    ],
    'name': 'person',
    'primaryKey': ['id']
  }],
  'fields': [{
    'constraints': {'required': False}, 'default_value': 'software_release_id_seq()', 'name': 'id', 'type': 'int4'},
    {'constraints': {'required': False}, 'name': 'software_name', 'type': 'varchar(100)'},
    {'constraints': {'required': True}, 'name': 'release_name', 'type': 'varchar(100)'},
    {'constraints': {'required': False}, 'name': 'major', 'type': 'int4'},
    {'constraints': {'required': False}, 'name': 'minor', 'type': 'int4'},
    {'constraints': {'required': False}, 'name': 'patch', 'type': 'int4'},
    {'constraints': {'required': True}, 'name': 'revision', 'type': 'varchar(50)'}
  ],
  'foreignKeys': [],
  'indexes': [{
      'creation': 'CREATE UNIQUE INDEX software_release__version ON software_release USING btree (software_name, major, minor, patch)',
      'definition': 'btree (software_name, major, minor, patch)',
      'fields': ['software_name', 'major', 'minor', 'patch'],
      'name': 'software_release__version',
      'primary': False,
      'unique': True},
      {'creation': 'CREATE INDEX software_release__versions2 ON software_release USING btree (major, minor)',
      'definition': 'btree (major, minor)'
    ]
}
```
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>An auto-incrementing integer primary key for each feature change.</td>
</tr>
<tr>
<td>description</td>
<td>Textual description of the feature change.</td>
</tr>
<tr>
<td>major</td>
<td>Major version number for the feature change.</td>
</tr>
<tr>
<td>minor</td>
<td>Minor version number for the feature change.</td>
</tr>
<tr>
<td>patch</td>
<td>Patch version number for the feature change.</td>
</tr>
</tbody>
</table>

**Indexes**

- `CREATE UNIQUE INDEX feature_change_pkey ON feature_change USING btree (id)`
- `CREATE UNIQUE INDEX software_release_pkey ON software_release USING btree (id)`

**Generation Information**

- Generation begin time: 2015-10-18 13:30:20.086386+02
- Generation end time: 2015-10-18 13:30:20.086386+02
- Source: PostgreSQL
- Source version: 9.4.4
4.1 pg_jts

Create a generalized JSON-table-schema structure from a live postgres database.

The JSON data structure returned from `get_database()` is a generalization of the JSON-table-schema: The resources in our structure comply with the table definition there (we extend it in allowed ways). Our structure comprises the whole database. It is the JSON-encoded form of a dictionary with these keys (values being strings, if not otherwise indicated):

- **source**: the string ‘PostgreSQL’
- **source_version**: the PostgreSQL version returned by the server
- **database_name**: the database name
- **database_description**: the comment on the database
- **generation_begin_time**: begin datetime as returned from PostgreSQL
- **generation_end_time**: end datetime as returned from PostgreSQL
- **datapackages**: a list of dictionaries, one for each PostgreSQL schema, with these keys:
  - **datapackage**: the name of the PostgreSQL schema
  - **resources**: a list of dictionaries, each describing a table within the current PostgreSQL schema and having these keys:
    - **name**: the name of the table
    - **description**: the table comment (only those components not part of a weak foreign key definition)
    - **primaryKey**: the primary key of the table, which is a list of column names
    - **fields**: a list of dictionaries describing the table columns and having these keys:
      - **name**: the column name
      - **description**: the column comment
      - **position**:
      - **type**: the PostgreSQL data type, e.g., ‘varchar(100)’ or ‘int4’
      - **defaultValue**: the default value of the column, e.g., ‘0’, or ‘person_id_seq()’ in case of a sequence
      - **constraints**: a dictionary describing constraints on the current column, with these keys:
        - **required**: boolean telling whether the column has a ‘NOT NULL’ constraint
* **indexes:** a list of dictionaries, one per index and column, having these keys:
  
  - **name:** name of the index
  - **columns:** a list with the names of the columns used in the index and ordered by priority
  - **creation:** the SQL statement for creating the index
  - **definition:** the index definition, e.g., ‘btree (id1, id2)’
  - **primary:** boolean telling whether the indexed columns form a primary key
  - **unique:** boolean telling whether the indexed columns are constrained to be unique

* **foreignKeys:** a list of foreign keys used by the current table:
  
  - **columns:** the names of the columns in the current table which are referencing a remote relation
  - **enforced:** a boolean telling whether the foreign key constraint is being enforced in PostgreSQL (True), or if it is a weak reference and the constraint is kept only by the application software (False)
  - **reference:** a dict for specifying the reference target, having these keys:
    - **datapackage:** the name of the PostgreSQL schema in which the referenced table resides
    - **resource:** the name of the referenced table
    - **name:** the name of the foreign key constraint
    - **columns:** a list of the names of the referenced columns
    - **cardinalitySelf:** (optional) the cardinality of the foreign key relation (as obtained from a column or table comment) on the side of the current table
    - **cardinalityRef:** (optional) the cardinality of the foreign key relation (as obtained from a column or table comment) on the side of the remote table
    - **label:** (optional) a label describing the foreign key relation (as obtained from a column or table comment)

### 4.1.1 Foreign key syntax

Foreign keys will be recognized where either a (hard) foreign key constraint is present in PostgreSQL, or a table or column comment describes a foreign key relation according to these syntax rules (we call this **weak reference**):

- the comment is split at 1) `;` followed by a space character or 2) `
`, and results in what we call **components**
- if a component matches one of the **relation_regexps**, we try to find a column name, a table name and an optional schema name in it; we match existing names in one of these four formats:
  - schema.table.column
  - table.column
  - schema.table(column1, column2, ..., columnN)
  - table(column1, column2, ..., columnN)
- if a relation is valid, we also extract both cardinalities on the side of the table (card1) and on the foreign side (card2); the syntax is card1 link card2, where card1 and card2 are values in **cardinalities** and link is one of `--`, `-` with an optional space character on both sides (independently).
- if a relation is valid, we also extract a label for the relation: when the component contains a string like `label="<LABEL>"`, `<LABEL>` will be extracted. (On both sides of ‘=’ an arbitrary number of white spaces may appear.)
In cases where both a foreign key constraint and a weak reference are present, the weak reference information supplements the constraint, in particular by adding cardinalities (if present).

```python
pg_jts.pg_jts.cardinalities = ['0..1', '1', '0..N', '1..N']
```

Cardinalities.

These values are allowed in weak references.

```python
pg_jts.pg_jts.get_database(db_conn_str, relation_regexps=None, exclude_tables_regexps=None)
```

Return a JSON data structure representing the PostgreSQL database.

Returns a JSON string and a list of notifications. The notifications inform about invalid or possibly unwanted syntax of the weak references (contained in the comments).

A valid PostgreSQL connection string (`db_conn_str`) is required for connecting to a live PostgreSQL database with read permissions.

The resulting data structure is missing some details. Currently mainly these structures are extracted from the database:

- tables
- foreign key relations (both constraints and weak references)
- indexes

The optional arguments have these meanings:

- `exclude_tables_regexps` is a list of regular expression strings; if a table name matches any of them, the table and all its relations to other tables are omitted from the result
- `relation_regexps` is a list of regular expression strings; if a table comment or a column comment matches any of them, it is parsed for a 'weak' foreign key relation (cf. *Foreign key syntax*)

```python
pg_jts.pg_jts.get_schema_table_column_triples(database)
```

Return a list of all (schema_name, table_name, column_name)-combinations.

`database` must have the same structure as obtained from `get_database()`.

### 4.2 pg_database

Query structure information from a PostgreSQL database.

**Extract information on these structures from a database:**

- schemas (non-system only)
- tables
- columns
- indexes
- views

**Extraction of these structures has not been implemented yet:**

- table inheritance
- sequences
- triggers
- functions

**Note:** You have to call `pg_query.db_init()` with a PostgreSQL connection string in advance.
pg_jts.pg_database.get_columns(schema_name, table_name)
Return the column properties for given table_name and schema_name.

Return a list of dictionaries with these keys:
• column_name:
• datatype:
• ordinal_pos:
• null:
• column_default:
• column_comment:

pg_jts.pg_database.get_constraints(schema_name, table_name)
Return constraints for a table, one per constraint and per column.

Constraint types are:
• c: check constraint
• f: foreign key constraint
• p: primary key constraint
• u: unique constraint
• t: constraint trigger
• x: exclusion constraint
For each constraint the results are ordered by ordinal_position.

pg_jts.pg_database.get_database()
Return the name of the current database.

Returns a string.

pg_jts.pg_database.get_database_description()
Return the comment on the database.

Returns a string.

pg_jts.pg_database.get_functions(schema_name)
Return a list of triggers within a schema with given name.

NOT IMPLEMENTED; TODO:

pg_jts.pg_database.get_indexes(schema_name, table_name)
Return a list of indexes for a table within a schema.

Each index is described by a dictionary as described in pg_jts.pg_jts.

pg_jts.pg_database.get_now()
Return the current datetime from PostgreSQL.

Returns a string.

pg_jts.pg_database.get_schemas()
Return a list of all non-system schemas.

Each schema is described by a dictionary with following keys:
• schema_name: name of the schema
• schema_comment: the PostgreSQL comment characterizing the schema

pg_jts.pg_database.get_sequences(schema_name)
Return a list of sequences within a schema with given name.

NOT IMPLEMENTED; TODO:
SELECT * FROM information_schema.sequences;
pg_jts.pg_database.\texttt{get_server_version}()

Return the server version number.

Returns a string.

pg_jts.pg_database.\texttt{get_tables}(\texttt{schema_name})

Return a list of all tables within a schema.

Each table is described by a dictionary with following keys:
  • \texttt{table_name}: name of the table
  • \texttt{table_comment}: the PostgreSQL comment describing the table

pg_jts.pg_database.\texttt{get_triggers}(\texttt{schema_name})

Return a list of triggers within a schema with given name.

\textbf{NOT IMPLEMENTED; TODO:}

\texttt{SELECT * FROM information_schema.triggers;}

pg_jts.pg_database.\texttt{get_views}(\texttt{schema_name})

Return a list of views within a schema of given name.

Each view is described by a dictionary having these keys:
  • \texttt{view_name}: the name of the view (i.e. of the virtual table)
  • \texttt{view_definition}: the \texttt{SELECT} statement defining the view

\textbf{4.2.1 Developer hints}

PostgreSQL documentation:

• \url{http://www.postgresql.org/docs/current/static/catalogs.html}
• \url{http://www.postgresql.org/docs/current/static/functions-info.html}

To see the queries executed when displaying schema information with \texttt{psql}, just call \texttt{psql} with option \texttt{-E}.

\textbf{4.3 \texttt{pg_query}}

PostgreSQL access.

To use this module, \texttt{db_init()} has to be called in advance.

pg_jts.pg_query.\texttt{conn} = \texttt{None}

Database connection.

pg_jts.pg_query.\texttt{cur} = \texttt{None}

Database cursor within connection \texttt{conn}.

pg_jts.pg_query.\texttt{db_get_all}(\texttt{query}, \texttt{attrs})

Execute an SQL query and return all rows (as list of tuples).

pg_jts.pg_query.\texttt{db_init}(\texttt{db_conn_str}=\texttt{None})

Initialize a database connection using a connection string.

Source: \url{https://github.com/iburadempa/pg_jts/}
Python Module Index

p
pg_jts, 1
pg_jts.pg_database, 13
pg_jts.pg_jts, 11
pg_jts.pg_query, 15
C

cardinalities (in module pg_jts.pg_jts), 13
conn (in module pg_jts.pg_query), 15
cur (in module pg_jts.pg_query), 15

D

db_get_all() (in module pg_jts.pg_query), 15
db_init() (in module pg_jts.pg_query), 15

G

get_columns() (in module pg_jts.pg_database), 13
get_constraints() (in module pg_jts.pg_database), 14
get_database() (in module pg_jts.pg_database), 14
get_database() (in module pg_jts.pg_jts), 13
get_database_description() (in module pg_jts.pg_database), 14
get_functions() (in module pg_jts.pg_database), 14
get_indexes() (in module pg_jts.pg_database), 14
get_now() (in module pg_jts.pg_database), 14
get_schema_table_column_triples() (in module pg_jts.pg_jts), 13
get_schemas() (in module pg_jts.pg_database), 14
get_sequences() (in module pg_jts.pg_database), 14
get_server_version() (in module pg_jts.pg_database), 14
get_tables() (in module pg_jts.pg_database), 15
gt_triggers() (in module pg_jts.pg_database), 15
get_views() (in module pg_jts.pg_database), 15

P

pg_jts (module), 1
pg_jts.pg_database (module), 13
pg_jts.pg_jts (module), 11
pg_jts.pg_query (module), 15