Municipal Scrapers Contributor's Guide Documentation

Release

Open Civic Data

June 14, 2017

Contents

The opencivicdata organization on Github is the home to a collaborative effort to define common schemas and provide tools for gathering information on government organizations, people, legislation, and events.

If you're looking to contribute to the project or learn how to use the data, you're in the right place.

CHAPTER 1

Writing Scrapers

Getting Started Writing Scrapers

While we strive to make writing scrapers as simple as possible, there are a few prerequisites:

- Python (or Ruby using pupa-ruby)
- Understanding GitHub
- Scraping Basics

If you're already well-versed in Python, GitHub, and basics of web scraping you can skip to Getting Started.

Note: These instructions are intended for Linux or OS X. If you're using Windows you'll probably benefit from using something like MinGW or a VM running Linux. If you're using OS X you may also find the excellent OS X-specific docs published by Open North useful.

Python

If you aren't already familiar with Python you might want to start with Python on Codecademy.

Note: Make sure you are using Python 3.3 or newer.

Having a local development environment is recommended, virtualenv & virtualenvwrapper are optional tools that will help you keep your Python environment clean if you work on multiple projects.

Understanding GitHub

Contributing code requires a free GitHub account, if you haven't use Git before there's a Git tutorial to get you started.

Scraping Basics

It is useful to understand the basic concept of web scraping before beginning, which is somewhat beyond the scope of this documentation. We recommend this source.

We recommend the lxml.html library. If you work with jQuery but haven't used XPath you may also find lxml.cssselect useful, though it is a bit more limited.

In our experience spending a few minutes brushing up on the basics of XPath is well worth it as it makes scrapers easier to write and more maintainable in the long run.

Getting Started

The first thing to do is to choose a repository to work with, or create a new one.

Most likely you'll be creating a fork of one of the existing scraper repositories:

- scrapers-us-municipal - US municipal governments
- scrapers-us-state - US state-level governments
- scrapers-us-federal - US federal government
- scrapers-ca - Canadian legislative
- influence-usa/scrapers-us-state - US state influence data

If your scraper falls into one of those categories you should fork it and create a new directory within that repository. We'd also suggest you work on a branch to make merging changes as easy as possible.

If you're hoping to create a scraper for something not yet covered please email the Open Civic Data list and we can work with you to decide the best way to proceed.

Once you've chosen a repository you'll need to install the *pupa* library (the first syllable of pupa is pronounced 'pew' as in 'pew pew pew pew pew'). Also install any other dependencies (like *lxml*) that you'll be using to do your scraping. If you're using an existing repo, you should be able to get all necessary libraries by installing the requirements listed in that repository's requirements.txt file.

An example of how you might configure your setup:

```
# using a virtualenv highly recommended
$ mkvirtualenv --python `which python3` opencivicdata
# Install pupa
$ pip install --upgrade pupa
# Clone the repo that you forked on GitHub
$ git clone git@github.com:<yourusername>/scrapers-us-state.git
# Switch to a branch to make pulling your work later as easy as possible
$ cd scrapers-us-state
$ git checkout --branch <new-branch-name>
# ...do work...
$ git push --set-upstream origin <new-branch-name>
```

If you're all set up, you can move on to *Creating a New Scraper*.

Creating a New Scraper

If you've followed the directions at *Writing Scrapers* then you're ready to start a new scraper.

We'll be creating a new people scraper for Seattle, but simply subsititute your own city name for Seattle as you follow these next few steps.

To copy a skeleton project into a new scraper directory, use pupa's **init** command. It will ask you a few questions.

```
$ pupa init seattle
jurisdiction name: Seattle City Council
division id (look this up in the opencivicdata/ocd-division-ids repository):
official URL: http://seattle.gov/council/
```

```
create people scraper? [Y/n]: y
create events scraper? [y/N]: n
create bills scraper? [y/N]: n
create votes scraper? [y/N]: n
```

(For beginners we recommend starting with just a single scraper, it is easy to create more scrapers later.)

In order to prevent duplication and redundancy, standardized division-id's are available in the repository ocd-division-ids. In the identifiers subdirectory, you'll find full csvs for each of the jurisdicitions we've entered so far - open the appropriate one and find the relevant division. If you are interested in adding a new geography or a new division within an existing geography, please contact open-civic-data@googlegroups.com.

This process should have created a new directory (named for whatever argument you gave to **pupa init**, seattle in this case) which contains an __init__.py and a file for each scraper you asked pupa to create.

Your __init__.py should look something like this:

```
from pupa.scrape import Jurisdiction
from .people import SeattlePersonScraper


class Seattle(Jurisdiction):
    division_id = "ocd-division/country:us/state:wa/place:seattle"
    name = "Seattle City Council"
    url = "http://seattle.gov/council/"
    scrapers = {
        "people": SeattlePersonScraper,
    }

    def get_organizations(self):
        org = Organization(name="org_name",
            classification="legislature")

        org.add_post(
            label="position_description",
            role="position_type")

        yield org
```

Every scraper is required to provide a *Jurisdiction* subclass. **pupa init** created a working subclass but you may want to specify additional details. For a full description of all the options visit *Jurisdiction Objects*.

You'll also notice that your class defines a list of scrapers. These are used by **pupa update** when deciding which scrapers to run. By default *pupa update* will run all of your scrapers, but you can look at `../pupa/update` for further details.

In addition, every scraper needs to define at least one organization. In this case, the organiztion will likely be the Seattle City Council. Replace the text org_name with the name of the organization you're scraping. The organization also needs to have a classification. Select the most appropriate from this list, and replace "legislature" with it:

- legislature
- executive
- upper
- lower
- party
- committee

- commission

Finally, the file created by pupa init adds posts to the organization. Scrapers can run without posts, so if you won't be looking at people, feel free to delete this line. But if you will be scraping people, you should add the posts you'll be scraping. For example, for the Seattle City Council, you'll want to add a post for each of the 9 seats (called Positions in Seattle). For Position 1, we'd set the label to "Council Position 1" and the role to "Councilmember".

Once the orginazitaion is created and the positions are added, yield the organization. (If you're not familiar with yield and generators in python, we recommend this talk from PyCon 2013.)

You can create as many organizations as needed. For Seattle, you might also want an executive so you can scrape the mayor's office, and add the mayor as a position. Yield each organization after adding it. Don't worry about adding every committee - organizations such as committees can be added later when you find them with a scraper.

You're now set up to scrape data! Next up we'll discuss how to scrape events, bills and people.

Writing a Person Scraper

This document is meant to provide a tutorial-like overview of the steps toward contributing a municipal Person scraper to the Open Civic Data project.

This guide assumes you have a working pupa setup. If you don't please refer to the introduction on *Writing Scrapers*.

Special notes about People scrapers

The name is a bit misleading - so-called People scrapers actually scrape `Person`, `Organization` and `Membership` objects.

The relationship between these three types is so close that they all should be scraped at the same time.

Target Data

People scrapers pull in all sorts of information about `Organization` `Membership` and `Person` objects.

The target data commonly includes:

- People, and their posts (what bodies they represent)
  - Alternate names
  - Current photo
  - links (homepage, YouTube account, Twitter account)
  - Contact information (email, physical address, phone number)
  - Any other identifiers that might be commonly used
  - Committee memberships
- Orgs (committees, etc)
  - Other names
  - Commonly used IDs
  - Contact information for the whole body
  - Posts (sometimes called seats) on the org
  - People in each org, and in which seat they sit.

Creating a New Person scraper

Our person scraper can be located anywhere, and simply needs to be importable by the `__init__.py` so that we can reference it in the *get_scraper* method. Your scraper can even by located in the `__init__.py` file itself if you want to keep things extra simple, but scraper code can eventually get pretty lengthy, so its more scalable to break each scraper out into it's own file. The default is to put the code in a file called `people.py`. Open up that file to see the scraper stub generated by the **pupa** init program. It should look like this:

```
from pupa.scrape import Scraper, Person


class SeattlePersonScraper(Scraper):

    def scrape(self):
        # needs to be implemented
        pass
```

This is the default scraper template, which isn't very useful yet, but it helps to clarify what the intent of the scraper is. Let's take a closer look.

In order to scrape people and committees, we'll use the scrape method that's been defined in the sample scraper, yielding each *Person* object. You may also yield an iterable of *Person* objects, which helps if you are scraping both people and committees for the Jurisdiction, but want to keep the scraper logic in their own routines.

As you might have guessed by now, *Person* scrapers scrape many *People*, as well as any *Membership* objects that you might find along the way.

Let's take a look at sample working Pupa scraper:

```
from pupa.scrape import Scraper, Person
class SeattlePersonScraper(Scraper):
    def scrape(self):
        john = Person(name="John Smith",
        district="Position 1",
        role="Councilmember",
        primary_org="legislature")
        john.add_source(url="http://example.com")
        yield john
```

A person requires a name and a membership. The district, role and primary_org fields allow us to find the post to which John Smith is assigned. Recall that we added this post in `__init__`. You can go back and add more posts in `__init__` if needed. In addition, each entity that's scraped needs a source, which is added using `add_source`.

Committees and Memberships

As noted, the People scraper can also handle committees. We can use the following code to add committees:

```
from pupa.scrape import Scraper, Person, Organization
class SeattlePersonScraper(Scraper):
    def scrape(self):
        comm = Organization(name="Transportation Committee",
                            classification="committee",
                            chamber="legislature")

        comm.add_source(url="http://example.com/committtees/transit")
        yield comm
```

And we might want to add relationships between people and committees. The Person object initializer automatically creates a relationship between a person and his/her primary organization, but if we want to make John Smith a member of the Transportation Committee, we can use the Organization's `add_member` method. The full script is as follows:

```
from pupa.scrape import Scraper, Person, Organization


class SeattlePersonScraper(Scraper):

    def scrape(self):
        doc = self.get("http://www.sunlightfoundation.com")
        john = Person(name="John Smith",
            district="Position 1",
            role="Councilmember",
            primary_org="legislature")
        john.add_source(url="http://example.com")
        yield john

        comm = Organization(name="Transportation Committee",
                            classification="committee",
                            chamber="legislature")
        comm.add_source(url="http://example.com/committtees/transit")
        comm.add_member(john,role="chair")
        yield comm
```

Scraper Example

Of course, in real scrapers, you'll need to write some code to take care of getting the list of people that are in that jurisdiction, or have memberships in the Legislature. Hardcoding names, such as in the examples above doesn't do much for us, since we won't be able to capture the current state of the world.

As a slightly more fun example, here's a scraper that will scrape the Sunlight website for people's information. This is deliberately a mildly complex example (as well as being purely for fun!), to get a feel for what a working Person scraper may look like. Note that we're assuming that Sunlight is a committee of the United States. Here's the `__init__.py` contents:

```
from pupa.scrape import Jurisdiction, Organization
from .people import UsaPersonScraper

class Usa(Jurisdiction):
    division_id = "ocd-division/country:us"
    classification = "committee"
    name = "United States"
    url = "www.sunlightfoundation.com"
    scrapers = {
        "people": UsaPersonScraper,
    }

    def get_organizations(self):
        org = Organization(name="Sunlight Foundation", classification="committee")

        org.add_post(label="president", role="president")
        org.add_post(label="co-founder", role="co-founder")
        org.add_post(label="staff", role="staff")
        org.add_post(label="fellow", role="fellow")
        org.add_post(label="consultant", role="consultant")
        org.add_post(label="intern", role="intern")
```

```
        org.add_source("www.sunlightfoundation.com")

        yield org
```

And here's our people scraper:

```
from pupa.scrape import Scraper, Person
import lxml.html

class UsaPersonScraper(Scraper):

    def scrape(self):
        url = "http://sunlightfoundation.com/team/"
        entry = self.get(url).text
        page = lxml.html.fromstring(entry)
        page.make_links_absolute(url)

        for position in page.xpath("//ul[contains(@class,'sunlightStaff')]/li"):
            position_name = position.xpath('.//h3')[0].text
            position_name = position_name.replace("Sunlight","").strip()
            position_name = position_name.rstrip("s")

            for person in position.xpath(".//li"):
                name = person.xpath(".//span")[0].text.strip()
                homepage = person.xpath("..//a/@href")[0]
                member = Person(name=name,
                    role=position_name,
                    primary_org="committee")
                member.add_link(homepage)
                member.add_source(url)
                yield member
```

Special notes regarding Posts, Memberships and Districts

The keen observer will note that we're using role, district and primary_org to note the person's primary position.

Looking at the Popolo spec, you might be confused on why this isn't an opaque ID, or some sort of slug.

We use full strings to help avoid having to search through all available organizations at scrape-time. The resolution is done at import-time.

Writing an Events Scraper

Events listings are one of the more compelling datasets that we are able to collect, since it allows for near real-time updating of upcoming events. Events include hearings, meetings, or bascially anything with a date and time listed by the organization you're scraping.

Target Data

Event scrapers pull down information regarding upcoming (or past) Events and associated metadata, such as who was there, what was talked about, and any supporting material.

Some of the commonly scraped data includes:

- Name of the event
- When the event starts and ends
- Items on the Agenda
  - Related entities (people, orgs, bills)
  - Subject of the agenda item
  - Related media
- Where the event is to take place
  - Lat / lon (if it exists)
  - Description of location (such address or building)
  - Venue link
- Associated documents
- Associated people, orgs, participants
- Any video or audio of the event

Creating a new Events scraper

Let's take a look at a sample Pupa event scraper:

```
from pupa.scrape import Scraper
from pupa.scrape import Event
import datetime as dt
import pytz

class SeattleEventScraper(Scraper):
    def scrape(self):
        when = dt.datetime(1776,7,4,9,15)
        tz = pytz.timezone("US/Pacific") #set the timezone for this location
        when = tz.localize(when)
        e = Event(name="Hearing",  # Event Name
                  start_time=when,  # When the event will take place
                  timezone=tz.zone, #the local timezone for the event
                  location_name='Town Hall')  # Where the event will be
        e.add_source("http://example.com")
        yield e
```

The events scraper looks a lot like a person scraper - the same stuff is going on here - the magic `scrape` method, returns an iterable of objects. Unlike people, where we often found committees or other organizations, it's not common to come across other objects while scraping events, so this scraper will usually just return `Event` objects.

The scraper above contains the minimum elements required to create an event. But there's much more we might want to add. The following scraper adds particpants and documents that are relevant to the hearing:

```
from pupa.scrape import Scraper
from pupa.scrape import Event
import datetime as dt
import pytz

class SeattleEventScraper(Scraper):
    def scrape(self):
        when = dt.datetime(1776,7,4,9,15)
```

```
        tz = pytz.timezone("US/Pacific") #set the timezone for this location
        when = tz.localize(when)
        e = Event(name="Hearing",  # Event Name
                      start_time=when,  # When the event will take place
                      timezone=tz.zone, #the local timezone for the event
                      location_name='unknown')  # Where the event will be
        e.add_source("http://example.com")

        #add a committee
        e.add_participant(name="Transportation Committee",
                          type="committee")

        #add a person
        e.add_person(name="Joe Smith", note="Hearing Chair")

        #add an mpeg video
        e.add_media_link(note="Video of meeting",
                          url="http://example.com/hearing/video.mpg",
                          media_type="video/mpeg")

        #add a pdf of meeting minutes
        e.add_media_link(note="Meeting minutes",
                          url="http://example.com/hearing/minutes.pdf",
                          media_type="application/pdf")

        yield e
```

The event is now much more fleshed out. But we're still missing the meat of an event: the agenda! Next we'll add agenda items:

```
from pupa.scrape import Scraper
from pupa.scrape import Event
import datetime as dt
import pytz

class SeattleEventScraper(Scraper):
    def scrape(self):
        when = dt.datetime(1776,7,4,9,15)
        tz = pytz.timezone("US/Pacific") #set the timezone for this location
        when = tz.localize(when)
        e = Event(name="Hearing",  # Event Name
                      start_time=when,  # When the event will take place
                      timezone=tz.zone, #the local timezone for the event
                      location_name='unknown')  # Where the event will be
        e.add_source("http://example.com")

        #add a committee
        e.add_participant(name="Transportation Committee",
                          type="committee")

        #add a person
        e.add_person(name="Joe Smith", note="Hearing Chair")

        #add an mpeg video
        e.add_media_link(note="Video of meeting",
                          url="http://example.com/hearing/video.mpg",
                          media_type="video/mpeg")
```

```
        #add a pdf of meeting minutes
        e.add_media_link(note="Meeting minutes",
                         url="http://example.com/hearing/minutes.pdf",
                         media_type="application/pdf")


        #add an agenda item to this event
        a = e.add_agenda_item(description="Testimony from concerned citizens")

        #the testimony is about transportation and the environment
        a.add_subject("Transportation")
        a.add_subject("Environment")

        #and includes these two committees
        a.add_committee("Transportation")
        a.add_committee("Environment and Natural Resources")

        #these people will be present
        a.add_person("Jane Brown")
        a.add_person("Alicia Jones")
        a.add_person("Fred Green")

        #they'll be discussing this bill
        a.add_bill("HB101")

        #here's a document that is included
        a.add_media_link(note="Written version of testimony",
                         url="http://example.com/hearing/testimony.pdf",
                         media_type="application/pdf")

        yield e
```

This example shows how to use the events model exhaustively. However, we haven't done any actual web-scraping. All of the details we added are hard-coded. It is quite difficult to show an example of a functioning web-scraper for an events page, as we have found that legislative events pages or calendars tend to change formats somewhat frequently. For an example of a scraper that hits an actual webpage to find information, see *Writing a Person Scraper*.

Writing a Bill Scraper

Bill scrapers are scrapers that pull down all legislatition on a jurisdiction's legislative website (including, but not limited to things like House and Senate Resolutions, Bills or city ordinances).

Scrapers should scrape all bills from a session every single night.

Target Data

Bill scrapers are used to pull in information regarding Legislation, and basic associated metadata.

Bill scrapers should collect all the information it's able to. The most common bits of data are:

- Basic information (name, session, chamber, summary)
- Sponsorship information (primary, secondary, etc)
- Actions regarding the legislation (introduction date, committee referral, chamber crossover, etc)
- Alternate names of the Legislation

- Related documents (fiscal reports, supporting data)
- Bill versions (Introduced Version, as amended)
- Related bills (companion bills, reintroductions)
- Subjects (Technology, Transportation, Education)

Note: In addition to the data above, it's common for Bill scrapers to also scrape in Vote information as well, since it's often linked directly from the Vote page.

Overview

Bills are certainly the most complicated and varied thing we'll be scraping as part of this project. For starters, bills are the only object that needs to be attached to a legislative session. A legislative session the period during which actions can happen to a bill. After the end of a session, bills that have not been passed would need to be re-introduced. For example, the US Congress has 2-year sessions that start at the beginning of odd years. Sessions sometimes have numbers (the US congressional session starting in 2015 is the 114th session). Note that sessions are not necessarily the same thing as terms. In the US House, a Rep's term is 2 years long and coincides with the bounds of a session, but Senators' terms are 6 years long, so span multiple sessions. In some jurisdictions, members can be elected mid-session.

For municipal governments, it can be somewhat difficult to find information about legislative sessions. We recommend contacting the city or town clerk's office, or the press office for larger cities.

When you've figured out the beginning and end dates for the most recent session, add it to `__init__.py`:. The variable legislative_sessions needs to be a list of dictionaries with at least an identifier, a name, a start_date and an end_date. The identifier should be a unique description of the session (so the 2015 regular session of a legislature might have an identifier of 2015), the name should be a human-readable string, and the dates should be in format YYYY-MM-DD. For example:

```
legislative_sessions = [{"identifier":"2015",
            "name":"2015 Regular Session",
            "start_date": "2015-01-01",
            "end_date": "2016-12-31"}]
```

Now that we've got at least one session in `__init__.py`, let's start out with a simple scraper:

```
from pupa.scrape import Scraper
from pupa.scrape import Bill


class SeattleBillScraper(Scraper):

    def scrape(self):
        session = self.jurisdiction.legislative_sessions[0]
        bill = Bill(identifier="R101",
                legislative_session=session["identifier"],
                title="More cookies for children",
                classification="resolution")
        bill.add_source("http://example.com")
        yield bill
```

Only identifier, legislative_session and title are required. Classification will default to "bill" if none is given. Classification may be any of the following:

- bill
- resolution

- concurrent resolution
- joint resolution
- memorial
- commemoration
- concurrent memorial
- joint memorial
- proposed bill
- proclamation
- nomination
- contract
- claim
- appointment
- constitutional amendment
- petition
- order
- concurrent order
- appropriation
- ordinance
- motion

For a bicameral legislature, the chamber should also be included in the bill initialization (as 'upper' or 'lower').

We can add a variety of other pieces of information to a bill. All are optional and bills will successfully import without any of these extras, but please note that internal quality checks may be triggered by bills with no versions or actions as those ought to exist for every available bill. The scraper below gives examples of all additional pieces of data that can be added:

```
from pupa.scrape import Scraper
from pupa.scrape import Bill


class SeattleBillScraper(Scraper):

    def scrape(self):
        session = self.jurisdiction.legislative_sessions[0]
        bill = Bill(identifier="R101",
                legislative_session=session["identifier"],
                title="More cookies for children",
                classification="resolution")
        bill.add_source("http://example.com")

        #add a sponsor
        bill.add_sponsorship(name="Joe Smith", #name of person or org
                classification="Primary", #primary? secondary? first? co-sponsor? etc
                entity_type="person", #person or organization
                primary=True #boolean, T if primary, F otherwise
                )
```

```
        #add subject(s)
        bill.add_subject("Nutrition")
        bill.add_subject("Youth")

        #add abstract or summary
        bill.add_abstract(abstract="Provides every child with a cookie",
                        note="Abstract for introduced version")


        #add other title(s) the bill may have gone by
        #perhaps a former title or a subtitle?
        bill.add_title("Om nom nom cookies")

        #add other ID(s) the bill has previously had
        #this can be useful for bills that are
        #renamed or substituted or have an omnibus relationship
        bill.add_identifier("R095")

        #add versions of the bill text
        bill.add_version_link(note="Introduced",
                            url="http://example.com/R101.pdf",
                            date="2015-05-05", #optional, YYYY-MM-DD
                            media_type="application/pdf" #optional but useful!
                            )

        #add other documents (not versions)
        #such as fiscal analysis, committee report,
        #testimony, etc
        bill.add_document_link(note="Fiscal Note",
                            url="http://example.com/R101/FiscalNote.pdf",
                            date="2015-05-05", #optional, YYYY-MM-DD
                            media_type="application/pdf" #optional but useful!
                            )

        #add related bill, useful for bills that were replaced,
        #substituted, in an omnibus relationship, continued
        #from a previous session, etc.
        bill.add_related_bill(identifier="R105",
                            legislative_session=session["identifier"],
                            relation_type="companion" #companion, prior-session,
                                                    #replaced-by, replaces
                            )


        #add actions. an action can also take a chamber
        #('upper' or 'lower') if this is a bicameral legislature
        act = bill.add_action(description="Bill Introduced",
                        date="2015-05-05",
                        classification="introduction", #see note about allowed classifications
                        )

        #add entities to the action. This is how you'd add
        #committees or people who participated
        act.add_related_entity(name="Transportation Committee",
                               entity_type="organization")

        yield bill
```

Bill actions should be one of the following:

- filing
- introduction
- reading-1
- reading-2
- reading-3
- passage
- failure
- withdrawal
- substitution
- amendment-introduction
- amendment-passage
- amendment-withdrawal
- amendment-failure
- amendment-amendment
- committee-passage
- committee-passage-favorable
- committee-passage-unfavorable
- committee-failure
- executive-receipt
- executive-signature
- executive-veto
- executive-veto-line-item
- veto-override-passage
- veto-override-failure
- deferral
- receipt
- referral
- referral-committee

Note that when we actually scrape the site, we'd like to limit the bills we ingest to the current legislative session. Depending on the site, this can be done by navigating to a page that only contains information from the current session, or by limiting a search by the date range related to a session.

Scraping Votes

In almost every case, votes are found on the same page as bills, so we tend to scrape them from the bill scraper. Below is an example (we've removed all but the required features of a bill to keep things shorter.)

Now, let's take a look at how we can add Vote information to a bill:

```
from pupa.scrape import Scraper
from pupa.scrape import Bill, Vote


class SeattleBillScraper(Scraper):

    def scrape(self):
        session = self.jurisdiction.legislative_sessions[0]
        bill = Bill(identifier="R101",
                legislative_session=session["identifier"],
                title="More cookies for children",
                classification="resolution")
        bill.add_source("http://example.com")

        #create a vote
        v = Vote(legislative_session=session["identifier"],
                    motion_text = 'Shall the bill pass the first reading?',
                    start_date = '2015-05-06', #date of the vote
                    classification = 'bill-passage', #or 'amendment-passage' or 'veto-override'
                    result = 'pass', #or 'fail'
                    bill = bill
                    )

        #we'll add the legislators' votes below.
        #note that sometimes only the counts are available,
        #not how individuals vote. So skip to the counts if
        #that's the case.

        #add yes and no votes
        v.yes("John Smith")
        v.no("Susan Jones")
        v.yes("Jessica Brown")

        #add votes with other classifications
        #option can be 'yes', 'no', 'absent',
        #'abstain', 'not voting', 'paired', 'excused'
        v.vote(option="absent",
                voter="Angela Cruz")


        #when possible it is best to set the vote
        #counts separately from the way individuals voted
        #this is important because vote documents can often
        #be the hardest thing to parse and the most liekly to contain errors
        #so if we can get good, reliable data on the vote count,
        #we should use it.
        v.set_count(option="yes", value=2)
        v.set_count(option="no", value=1)
        v.set_count(option="absent", value=1)

        v.add_source("https://example.com/R101/votes")

        yield bill
        yield v
```

If you're unable to scrape the `Vote` at the same time as you're scraping that particular `Bill`, you can attempt to match by using the alternate signature of the `set_bill` method:

```
v.set_bill("R101", chamber="upper")
```

This call will dispatch based on the type of the first argument. For more information, check out the `pupa.models.vote.Vote.set_bill()` documentation.

Running the Scraper

As you develop it will be a good idea to run the scraper to ensure that the output JSON is in good shape.

Run the scraper:

```
$ pupa update seattle
```

Where `seattle` is simply a Python-importable path to your scraper directory. From there, the `jurisdiction` object will be able to tell `pupa` where to find the scrapers.

In addition, there are some useful arguments to know about.

Firstly, when doing local testing, `--fast` disables Pupa's scrape throttling, and uses the `scrape_cache` to prevent fetching pages over the line. This is useful when doing prototyping, but shouldn't be used regularly, since it puts more load on these websites, and will read stale data (if your cache stays around).

Secondly, if don't have an opencivicdata postgres database set up, it's useful to pass `--scrape` to `pupa`, to prevent the `--import` and `--report` stages from running.

Lastly, being able to restrict which scraper gets run by indicating `people`, `bills`, `events` or `votes` after the jurisdiction.

At any point, you can run:

```
$ pupa update -h
```

To get most up-to-date information regarding the invocation of Pupa.

Usually, during rapid development, the invocation would look something like:

```
$ pupa update seattle people --fast
```

Validating Data

After this completes, the data will be in the `scraped_data` folder. Each OpenCivic object that gets saved will be written to `scraped_data/<jurisdiction_id>/<type>_<tmp_id>.json`.

This object will be a JSON-encoded OpenCivic object, which is a well-documented and defined format for Government data.

By spot-checking a few of the entries, you can check to see if data looks funny, or if things aren't being categorized properly.

If you want to spot-check some data, using a modern POSIX system should allow you to run something similar to:

```
$ python -m json.tool $(ls | shuf -n 1) | vim -
```

Feel free to change `vim` to whatever editor you prefer for such tasks.

If you do use vim, there's a helpful JSON Plugin

Here is an example JSON file you'd get if you run the events scraper we created in *Writing an Events Scraper*, although note that your IDs will be different:

```
{
    "_id": "efa7ccee-f4d6-11e4-b1eb-843a4bcaaa18",
    "agenda": [
        {
            "description": "Testimony from concerned citizens",
            "media": [
                {
                    "date": "",
                    "links": [
                        {
                            "media_type": "application/pdf",
                            "url": "http://example.com/hearing/testimony.pdf"
                        }
                    ],
                    "note": "Written version of testimony"
                }
            ],
            "notes": [],
            "order": "0",
            "related_entities": [
                {
                    "entity_type": "committee",
                    "name": "Transportation",
                    "note": "participant"
                },
                {
                    "entity_type": "committee",
                    "name": "Environment and Natural Resources",
                    "note": "participant"
                },
                {
                    "entity_type": "person",
                    "name": "Jane Brown",
                    "note": "participant"
                },
                {
                    "entity_type": "person",
                    "name": "Alicia Jones",
                    "note": "participant"
                },
                {
                    "entity_type": "person",
                    "name": "Fred Green",
                    "note": "participant"
                },
                {
                    "entity_type": "bill",
                    "name": "HB101",
                    "note": "consideration"
                }
            ],
            "subjects": [
                "Transportation",
                "Environment"
            ]
        }
    ],
    "all_day": false,
```

```
    "classification": "event",
    "description": "",
    "documents": [],
    "end_time": null,
    "extras": {},
    "links": [],
    "location": {
        "coordinates": null,
        "name": "unknown",
        "note": ""
    },
    "media": [
        {
            "date": "",
            "links": [
                {
                    "media_type": "video/mpeg",
                    "url": "http://example.com/hearing/video.mpg"
                }
            ],
            "note": "Video of meeting"
        },
        {
            "date": "",
            "links": [
                {
                    "media_type": "application/pdf",
                    "url": "http://example.com/hearing/minutes.pdf"
                }
            ],
            "note": "Meeting minutes"
        }
    ],
    "name": "Hearing",
    "participants": [
        {
            "entity_type": "committee",
            "name": "Transportation Committee",
            "note": "participant"
        },
        {
            "entity_type": "person",
            "name": "Joe Smith",
            "note": "Hearing Chair"
        }
    ],
    "sources": [
        {
            "note": "",
            "url": "http://example.com"
        }
    ],
    "start_time": "1776-07-04T17:08:00+00:00",
    "status": "confirmed",
    "timezone": "US/Pacific"
}
```

Submitting a Pull Request

The municipal scraping effort we're working on is extremely friendly to contributors of all backgrounds, and we accept code contributions via GitHub Pull Requests.

Before you begin, you should create a GitHub account if you don't already have one, and learn the basics of using Git (slightly out of scope for this document). This guide assumes basic proficiency with Git.

Please test your scraper locally, but if you have questions we're all quite happy to go back and forth in the github comments to work on changes, if they're needed.

In general, if you just keep the Pull Request short and self-contained. If you'll be modifying multiple jurisdictions, please submit separate pull requests. It will make life much easier for the reviewers!

Fork the repo you want to contribute to

First navigate to the repo you want to contribute to and create a fork. If you're contributing a municipal scraper within the United States, for example, view that repo's page on Github and click the `fork` button.

This creates a repo (called `scrapers-us-municipal`, if you've followed the link just up above) on your personal account that you can commit to.

You can `clone` this repo down to your local machine by using *git clone* (do read up on the GitHub guide if you're having trouble with this step - or git!)

After pulling the repo down, we'll set a new remote called `upstream` to help interact with the `opencivicdata` repo later on.

```
$ git remote add upstream git@github.com:opencivicdata/municipal-scrapers-us.git
```

If you cloned the `opencivicdata` repo before you forked the repo on GitHub, don't worry - you can adjust this fairly quickly!

```
$ git remote rm origin
$ git remote add origin git@github.com:yourbadself/municipal-scrapers-us.git
$ git fetch origin
```

See also:

Github's docs on forking a repo: https://help.github.com/articles/fork-a-repo

Submit a pull request

Before you submit your Pull Request, it's quite handy to run through a quick checklist of common (and easy to catch) gotchas:

- Have you added yourself to the `AUTHORS` file? If not, please do.
- Is your Pull Request up-to-date with the `opencivicdata` repo? If it's not, it might be helpful to jump down to the `Keeping your branch up to date` section below.

Finally, navigate to the commit you made to your forked repo, and click the button to submit a pull request.

See also:

Github's docs on using pull request: https://help.github.com/articles/fork-a-repo

Best Practice

Note: This guide won't get into a generic `git` tutorial, and assumes basic proficiency with `git` and some knowledge of GitHub.

It's good practice to use a branch when working on the scrapers, this helps continue to integrate changes into your branch, and helps you compare changes without much effort. With many people working on the codebase at the same time, it's likely we'll end up with changes that impact others sometimes. By using a branch, it's much easier to fix these conflicts.

Warning: Please do make sure you always create a branch off the *master* branch, unless you've got lots of git experience and are doing this for a very specific reason.

To create a branch, you can checkout a new branch (this operation creates the branch, so don't worry about using `git branch` just yet.)

```
$ git checkout -b bugfix/fix-this-broken-jurisdiction
```

It's common to prefix a branch with one of `bugfix`, or `feature` (or anything else that's short and desriptive). After the prefix, you should add a descriptive slug related to the change, so that it's easy to remember which branch is which. These are sometimes called "Topic branches".

After this, you can check which branch you're working on by running `git branch`, and looking for the marked branch.

```
$ git branch
* bugfix/fix-this-broken-jurisdiction
  master
```

To switch back to the master branch (for any reason), you can `checkout` the branch again.

```
$ git checkout master
$ git branch
  bugfix/fix-this-broken-jurisdiction
* master
```

Keeping your branch up to date

It saves quite a bit of time if you can ensure that all changes have been incorporated in your branch when sending in a Pull Request. Often times this is not an issue for short-lived branches, however, sometimes people have changed code in the `opencivicdata` repo, and you need to merge code from "`upstream`" into your working branch.

Let's go over how to do this.

Warning: The following assumes you have a setup similar to above. Make sure that you have the *upstream* remote set up, and are working on a topic branch.

Firstly, be sure that you've commited all your code, and you're up to date.

```
$ git branch
* bugfix/fix-this-broken-jurisdiction
  master
$ git checkout master
$ git pull upstream master
$ git checkout bugfix/fix-this-broken-jurisdiction
$ git merge master
```

Please do remember to change `bugfix/fix-this-broken-jurisdiction` with the name of your topic branch that you're working on (as seen in the output of the first command run).

Checking what you've changed

You can check how much has changed at any point very simply, by using `git diff`. Something like:

```
$ git diff master --color
```

Can come in quite handy when reviewing changes before sending in a Pull Request.

Common tips for writing scrapers

The following doc contains a list of useful recipies to help scrape data down from legislative websites. These are by no means the only way to do these things, but it's a description of some of the things we've found to work well.

Fetching a page and setting URLs to absolute paths

It's handy to be able to set all the relative URL paths to absolute paths. `lxml` has a pretty neat facility for doing this.

It's not uncommon to see a method such as:

```
def lxmlize(self, url):
    entry = self.get(url).text
    page = lxml.html.fromstring(entry)
    page.make_links_absolute(url)
    return page
```

Getting the current session

We might want to know what the current legislative session is. A legislative session is required for a bill, and can be helpful in limiting the duration of a scrape (for legislatures that have persistent pages, we probably don't want to scrape all bills/legislators/events back to when they started keeping track!) Sessions are created in `__init__.py` as a list of dictionaries. Jurisdictions can do all kinds of weird things with sessions (we've seen them create sessions inside sessions) so keeping track based on date won't work. Instead, you'll need to order sessions chronologically, with the current one on top. For example:

```
legislative_sessions = [{"identifier":"2015",
                         "name":"2015 Regular Session",
                         "start_date": "2015-01-01",
                         "end_date": "2016-12-31"},
                         {"identifier":"2013",
                         "name":"2013 Regular Session",
                         "start_date": "2013-01-01",
                         "end_date": "2014-12-31"}]
```

Then to get the current session from any scraper, you can call:

```
self.jurisdiction.legislative_sessions[0]
```

Common XPath tricks

The following is a small list of very common tricks hackers use in `xpath` expressions.

Quick text grabs

Getting text values of HTML elements:

```
//some-tag/ul/li/text()
```

Which would be roughly similar to the following pseudo-code:

```
[x.text for x in page.xpath("//some-tag/ul/li")]

# or, more abstractly:

for el in page.xpath("//some-tag/ul/li"):
    deal_with(el.text)
```

This is helpful for quickly getting the text values of a bunch of nodes at once without having to call `.text` on all of them. It's worth noting that this is *different* behavior than `.text_content()`.

Class limiting / ID limiting

Sometimes it's helpful to get particular nodes of a given class or ID:

```
//some-tag[@class='foo']//div[@id='joe']
```

This expression will find all `div` objects with an `id` of `joe` (I know, you *should* only use an `id` once, but alas sometimes these things happen) that are sub-nodes of a `some-tag` with a `class` of `foo`.

In addition, you can also limit by other things, too, such as `text()`:

```
//some-other-tag[text()="FULL TEXT"]/*
```

This will find any `some-other-tag` tags that contain `FULL TEXT` as their `text()` entry. As you can guess, most XPath expressions (etc)

Contains queries

With the above, it's sometimes needed to search for all `class` attributes that *contain* a given string (sometimes sites have quite a bit of autogenerated stuff around an ID or class name, but a substring stays in place)

Let's take a look at limiting queries:

```
//some-tag[contains(@class, 'MainDiv')]
```

This will find any instance of `some-tag` who's `class` contains the substring `MainDiv`. For example, this *will* match an element such as `<some-tag class='FooBar12394MainDiv333' ></some-div>`, but it will *not* match `<some-tag class='FooBarMain123Divsf'></some-div>` or a `some-tag` without a class.

Keep in mind that the `@foo` can be any attribute of the HTML element, such as `@src` for an `img` tag or an `@href` for an `a` tag.

Array Access

> **Warning:** Be careful with this one!

You can access indexes of returned lists using square brackets (just like in Python itself), although this tends to not be advised (since the counts can often change, and you may end up scraping in bad data).

However, this is sometimes needed:

```
//foobar/baz[1]/*
```

to get all entries under the 1st `baz` under a `foobar`. It's also worth noting that **xpath indexes are 1-based not 0-based**. Start your counts from `1` not `0` and you'll have a much better day!

Axis Overview

XPath also features what are known as the "Axis". The "axis" is a way of selecting other nodes via a given node (which is usually defined by an xpath)

The most useful one is `following-sibling` or `parent`

Let's take a look at `following-sibling`:

```
//th[contains(text(), "foo")]/following-sibling::td
```

This will find any `th` elements that contain `foo` in the `text()`, and search for any `td` elements which *follow* the `th` element.

Or, if we look at a `parent` relation:

```
//img[@id='foo']/parent::div[@class='bar']/text()
```

will fetch the text of a `div` with a `class` set to `bar` who has a sub-node, which is an `img` with an `id` set to `foo`. This expression will continue all the way back up to the root node.

Writing "defensive" scrapers

We tend to write very fragile scrapers - prone to break very loudly (and as soon as we can) when/if the site changes.

As a general rule, if the site has changed, we have a strong chance of pulling in bad data. As a result, we don't want the scraper to continue on without throwing an error, so that we can be sure bad data never gets imported into the database. We do this by hard-coding very fragile xpaths, which use full names (rather than contains, unless there's a reason to), and always double-check the incoming data looks sane (or raise an `Exception`).

One way that's common to help trigger breakage when table rows get moved around is to unpack the list into variables - this also has an added bonus of being more descriptive in what is where in the row, which aids in debugging a broken scraper. Usually, you'd see something like:

```
for row in page.xpath("//table[@id='foo']/tr"):
    name, district, email = row.xpath("./*")
```

Which will trigger breakage if the number of rows change. It still helps to sill assert that you have sane values in such a table, since the order of the entries may change, and you'll end up changing everyone's name to "District 5".

Another common way of doing this is by blindly using an index off an xpath, forcing an `IndexError` if the index isn't present. This helps avoid queries where nothing is returned, or too little is returned. You should also be careful to check the `len()` of the values to ensure too much wasn't returned as well.

Commonly, scrapers need to normalize and transform bad data into good data (in edge-cases, such as setting `party` data), and this can be a good place to add a quick check that no data we didn't expect made it into the database.

Using a dict to index the scraped data is a good way of doing this:

```
party = {"democrat": "Democratic",
         "republican": "Republican",
         "independent": "Independent"}[scraped_party.lower().strip()]
```

You can be sure that if the data wasn't one of the expected 3 that it will raise a `KeyError` and force someone to ensure the scraped data is (in fact) correct (or if a new party needs to be added).

Since this is infrequent enough, this is a pretty good tradeoff for data quality (and is slightly easier to maintain than a big `if`/`elif`/`else` block).

The end goal here is to make sure that *no scraper ever allows bad data into the database*. So long as your scraper is doing this, you've written a defensive scraper!

CHAPTER 2

Open Civic Data Formats

Adopting the OCD Specification

Warning: Parts of Open Civic Data underwent a large refactor as of mid-2014, some information on this page may be out of date. We're working on updating this documentation as soon as possible.
We'll remove these messages from pages as they're updated and vetted.

If you're a city or vendor looking to adopt the OCD specification, this section serves as an implementation guide, including working examples from the OCD API. When you've implemented any one of the OCD elements, please contact us so we can begin collecting your data into Open Civic Data.

To begin, you'll want to find the *division id* for your locality.

Finding Your Division ID

If you're within the United States (or a selection of other countries), your geographic division should already be included in the database. To look up your division id, visit the Open Civic Data editor and lookup tool. Start by typing in your state, and then locality name.

For instance, by typing in 'OH' for Ohio and then 'Cleveland' for the City of Cleveland, the division id displayed should look like this:

> ocd-division/country:us/state:oh/place:cleveland

This data is pulled from the US Census and should include every geographic division listed in the Census. If your division is too new to be in the Census or you otherwise need to add it, please take a look at the OCD repository for the division ids on Github. If you're outside the United States, chances are your division has not been added yet. You can clone the aforementioned repository to see if your division exists. If it does not, review the *requirements for new ids* and then send an email to our google group. New ids for divisions and jurisdictions are created and agreed upon by consensus via the mailing list, to prevent collisions.

Finding or Creating Organizations

Now that you've found your geopolitical division, you need to find (or create) your *organization*. To see if your organization already exists, you can use the same editor and lookup tool that you used to look up the division. Following the example above, if you click on the division id for Cleveland, you should see a list of all the organizations inside that division.

If you see your organization, or a parent of your organization, take note of the organization id and the jurisdiction id. If not, you'll need to create these. Jurisdiction ids are used to help identify top level parents, such as a city council or state legislature. These bodies usually have multiple children, and sometimes multiple levels of children, such as committees or the upper and lower chambers of a state legislature. The jurisdiction id helps to identify the top level governing body for all of the organizations underneath it. You can read more about creating a jurisdiction and organization ids *here*.

For example, here's how the data looks for the Ohio state senate. The division id references the state of Ohio, and the jurisdiction id references the overall Ohio state legislature (including the house and senate). The organization id for the Ohio state senate is *ocd-organization/b87d2136-3b43-11e3-9ac3-1231391cd4ec*.

```
{
    division_id: "ocd-division/country:us/state:oh",
    classification: "legislature",
    founding_date: null,
    chamber: "upper",
    identifiers: [

    ],
    posts: [
        {
            role: "member",
            label: "Member",
            num_seats: 1,
            id: "1"
        },
        {
            role: "member",
            label: "Member",
            num_seats: 1,
            id: "10"
        },
        {
            role: "member",
            label: "Member",
            num_seats: 1,
            id: "11"
        }
        ...
    ],
    other_names: [

    ],
    contact_details: [

    ],
    id: "ocd-organization/b87d2136-3b43-11e3-9ac3-1231391cd4ec",
    links: [

    ],
    name: "Ohio General Assembly, Senate",
    dissolution_date: null,
    sources: [
        {
            url: "http://www.legislature.state.oh.us/",
            note: null
        }
    ],
    memberships: [
```

```
        ...
    ],
    parent_id: null,
    extras: {

    },
    abbreviation: "oh",
    jurisdiction_id: "ocd-jurisdiction/country:us/state:oh/legislature"
}
```

Publishing Your Local Representatives

Representatives can be expressed using the *Person Object* format. You can read more about the explicit elements on the *person page* but for a quick start, here's an example in JSON:

```
{
    "_type": "person",
    "contact_details": [
        {
            "note": "",
            "type": "email",
            "value": "roswellmayor@roswell-nm.gov"
        },
        {
            "note": "",
            "type": "voice",
            "value": "575-637-6202"
        }
    ],
    "name": "Del Jurney",
    "links": [],
    "gender": "m",
    "image": "http://www.roswell-nm.gov/images/library/Image/del-jurney.jpg,
    "other_names": [],
    "sources": [
        {
            "url": "http://www.roswell-nm.gov/staticpages/index.php/city-mayor,
            "note": ""
        }
    ],
    "extras": {},
    "_id": "ocd-person/bff59848-b1c4-11e2-b819-12313d2facc4",
    "biography": "Roswell City Mayor Del Jurney. The Mayor is elected at-large and represents all ne
}
```

In person objects, the only absolutely required field is the name attribute. If the type (person, in this case) cannot be inferred from the endpoint, then a type attribute with the value "person" is also necessary. The more information you add, the better. Person objects can also be linked to organizations. For instance, in the OCD API, each organization object has a 'memberships' attribute, which is an array of people holding office. Here's an example of the memberships from the organization object displayed above:

```
memberships: [
        {
            person: {
                contact_details: [

                ],
```

```
    birth_date: null,
    biography: null,
    chamber: "upper",
    identifiers: [

    ],
    name: "Nina Turner",
    image: "http://www.ohiosenate.gov/senate/Assets/Headshots/Small/25.jpg",
    updated_at: "2014-04-16T00:18:58.287",
    other_names: [

    ],
    death_date: null,
    id: "ocd-person/ba595e34-3b43-11e3-9ac3-1231391cd4ec",
    links: [
        {
            url: "http://www.ohiosenate.gov/senate/turner",
            note: "Homepage"
        }
    ],
    summary: null,
    district: "25",
    extras: {
        first_name: "Nina",
        last_name: "Turner",
        +biography: "Representing Ohio's 25",
        office_phone: "(614) 466-4583"
    },
    gender: null,
    sources: [
        {
            url: "http://www.ohiosenate.gov/senate/members/senate-directory"
        }
    ],
    created_at: "2011-02-22T21:25:58.284"
},
contact_details: [
    {
        value: "Senate Building 1 Capitol Square, 2nd Floor Columbus, OH 43215",
        note: "Capitol Office",
        type: "address"
    },
    {
        value: "614-466-4583",
        note: "Capitol Office",
        type: "phone"
    }
],
end_date: null,
sources: [

],
role: null,
chamber: "upper",
organization_id: "ocd-organization/b87d2136-3b43-11e3-9ac3-1231391cd4ec",
post_id: "25",
extras: {
    term: "2013-2014"
```

```
            },
            start_date: "2013",
            unmatched_legislator: null,
            person_id: "ocd-person/ba595e34-3b43-11e3-9ac3-1231391cd4ec"
        },

        ...
    ]
```

The object includes lots of information about the legislature seat generally, and then contains a person attribute that contains information about the legislator filling this seat specifically. The generic information about the seat is important because it can exist and describe the seat even if it isn't presently occupied.

And More!

These are the basics of what any API or data store that adopts the OCD standard should contain. You can read more about other objects, like *events*, *bills* and *votes* on their respective pages. OCD is a new effort and improvements to the standard are being made all the time. If you have suggestions, questions, or want to participate in shaping the OCD standard, please join our google group.

Data Types

The Open Civic Data specifications define the following core types:

division A political geography such as a state, county, or congressional district. May have multiple boundaries over their lifetime.

Division IDs take the form `ocd-division/country:<country_code>[<type>:type_id>]+`. The canonical repository of division IDs is opencivicdata/ocd-division-ids You can also look up a division id using the Open Civic Data editor and lookup tool.

See *Division Objects* for details.

jurisdiction A governing body that exists within a division. While 'Florida' would be a division, the Florida State Legislature would be a jurisdiction.

Jurisdictions IDs take the form `ocd-jurisdiction/<jurisdiction_id>/<jurisdiction_type>` where `jurisdiction_id` is the ID for the related division without the *ocd-division/* prefix and `jurisdiction_type` is *council*, *legislature*, etc.

See *Jurisdiction Objects* for details.

person A person, typically a politician or government official.

The Popolo person schema is used to represent person data.

See *Person Objects* for details.

organization A group of people, such as a city council, state senate, or committee.

The Popolo organization schema is used to represent organization data.

See *Organization Objects* for details.

bill A legislative document and its history, may technically be a resolution, appointment, or contract so long as it has a name and would be considered to have a legislative history.

See *Bill Objects* for details.

vote The record of a vote taken on a motion, such as a confirmation or passage of a bill. May contain individual legislator's yay/nay votes or just an outcome.

See *Vote Objects* for details.

event A legislative event, such as a meeting or hearing.

See *Event Objects* for details.

Jurisdiction Objects

Jurisdiction objects have the following fields:

Basic Details

name (*string*) Name of jurisdiction (e.g. North Carolina General Assembly) **(required)**

url (*string*) URL pointing to jurisdiction's website. **(required)**

classification (*string*) A jurisdiction category. **(required)**

Allowed values:

- government
- legislature
- executive
- school_system
- transit_authority

legislative_sessions (*object*) Dictionary describing sessions, each key is a session slug that must also appear in one `sessions` list in `terms`. Values consist of several fields giving more detail about the session. **(required)**

Each element in session_details is an object with the following keys:

name (*string*) Name of session, typically a year span like 2011-2012. **(required)**

identifier (*string*) Identifier of session. **(required)**

classification (*string*) Type of session: primary or special.

start_date (*datetime*) Start date of session.

end_date (*datetime*) End date of session.

Additional Metadata

feature_flags (*array*) A way to mark certain features as available on a per-jurisdiction basis. **(required, minItems: 0)**

Each element in feature_flags is of type (string)

Division Objects

Warning: Parts of Open Civic Data underwent a large refactor as of mid-2014, some information on this page may be out of date. We're working on updating this documentation as soon as possible.
We'll remove these messages from pages as they're updated and vetted.

Basic Details

id Open Civic Data division ID.

country Two-letter ISO-3166 alpha-2 country code. (e.g. 'us', 'ca')

display_name Human-readable name for division.

Additional Fields

geometries A list of associated geometries, each of which has the following fields:

start Best approximation of date boundary became effective.

end Best approximation of date boundary was replaced or made obsolete (null for current boundaries).

boundary Boundary object- fields are determined from underlying data source, but always provides:

centroid

Object containing the centroid, not guaranteed to be within the object.

Example:

```
{ "type": "Point", "coordinates": [-176.59989528409687, 51.88215100813731] }
```

extent

Object describing the extents. [left-most, lower-most, right-most, upper-most]

Example:

```
[ -176.71309799999997, 51.80080899999999, -176.46673599999997, 51.95761899999999 ]
```

children A list of child jurisdiction ids.

Person Objects

Person objects have the following fields:

Basics

name (*string*) A person's preferred full name **(required)**

image (*string, null*) A URL of a head shot **(required)**

contact_details (*array*) Contact information for this entity. **(required, minItems: 0)**

Each element in contact_details is an object with the following keys:

note (*string, null*) for grouping data by location/etc. **(required)**

type (*string*) type of contact (e.g. phone, email, address) **(required)**

value (*string*) actual phone number/email address/etc. **(required)**

label (*string, null*) human-readable label **(required)**

links (*array*) URLs for documents about the person **(required, minItems: 0)**

Each element in links is an object with the following keys:

note (*string, null*) A note, e.g. 'Wikipedia page' **(required)**

url (*string*) A URL for a document about the person **(required)**

Extended Details

sort_name (*string, null*) A name to use in a lexicographically ordered list **(required)**

family_name (*string, null*) One or more family names **(required)**

given_name (*string, null*) One or more primary given names **(required)**

gender (*string, null*) A gender **(required)**

summary (*string, null*) A one-line account of a person's life **(required)**

national_identity (*string, null*) A national identity **(required)**

biography (*string, null*) An extended account of a person's life **(required)**

birth_date (*string, null*) A date of birth **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

death_date (*string, null*) A date of death **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

Alternate Names/Identifiers

identifiers (*array*) IDs other than the primary ID that the object may be known by. **(required, minItems: 0)**

Each element in identifiers is an object with the following keys:

scheme (*string, null*) What service this identifier is used by. **(required)**

identifier (*string*) The 3rd-party identifier, such as OKL0001000. **(required)**

other_names (*array*) Alternate or former names for this object. **(required, minItems: 0)**

Each element in other_names is an object with the following keys:

note (*string, null*) An optional note describing where this alternate name came from or its relationship to the entity. **(required)**

name (*string*) An alternate name this object is sometimes known by. **(required)**

end_date (*string, null*) The date at which this name was no longer valid. (null if still valid/valid indefinitely) **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

start_date (*string, null*) The date at which this name became valid.(null if unknown/valid indefinitely) **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

Common Fields

updated_at (*string, datetime, null*) The time at which the resource was last modified **(required)**

created_at (*string, datetime, null*) The time at which the resource was created **(required)**

sources (*array*) URLs for sources relating to the object **(required, minItems: 1)**

Each element in sources is an object with the following keys:

url (*string*) URL of resource used to collect information **(required)**

note (*null, string*) note about what information this URL was used for **(required)**

Organization Objects

Organization objects have the following fields:

Basics

name (*string*) A primary name, e.g. a legally recognized name **(required)**

classification (*string, null*) An organization category, e.g. committee **(required)**

Allowed values:

- legislature
- executive
- upper
- lower
- party
- committee
- commission
- corporation
- agency
- department

parent_id (*string, null*) The ID of the organization that contains this organization **(required)**

contact_details (*array*) Contact information for this entity. **(required, minItems: 0)**

Each element in contact_details is an object with the following keys:

note (*string, null*) for grouping data by location/etc. **(required)**

type (*string*) type of contact (e.g. phone, email, address) **(required)**

value (*string*) actual phone number/email address/etc. **(required)**

label (*string, null*) human-readable label **(required)**

links (*array*) URLs for documents about the person **(required, minItems: 0)**

Each element in links is an object with the following keys:

note (*string, null*) A note, e.g. 'Wikipedia page' **(required)**

url (*string*) A URL for a document about the person **(required)**

Posts

posts (*array*) Posts within the organization **(required, minItems: 0)**

Each element in posts is an object with the following keys:

id (*string, null*) The post's unique identifier **(required)**

label (*string*) A label describing the post **(required)**

role (*string, null*) The function that the holder of the post fulfills **(required)**

start_date (*string, null*) The date on which the post was created **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

end_date (*string, null*) The date on which the post was eliminated **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

contact_details (*array*) Contact information for this entity. **(required, minItems: 0)**

Each element in contact_details is an object with the following keys:

note (*string, null*) for grouping data by location/etc. **(required)**

type (*string*) type of contact (e.g. phone, email, address) **(required)**

value (*string*) actual phone number/email address/etc. **(required)**

label (*string, null*) human-readable label **(required)**

links (*array*) URLs for documents about the person **(required, minItems: 0)**

Each element in links is an object with the following keys:

note (*string, null*) A note, e.g. 'Wikipedia page' **(required)**

url (*string*) A URL for a document about the person **(required)**

Extended Details

image (*string, null*) A URL of an image **(required)**

founding_date (*string, null*) A date of founding **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

dissolution_date (*string, null*) A date of dissolution **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

Alternate Names/Identifiers

identifiers (*array*) IDs other than the primary ID that the object may be known by. **(required, minItems: 0)**

Each element in identifiers is an object with the following keys:

scheme (*string, null*) What service this identifier is used by. **(required)**

identifier (*string*) The 3rd-party identifier, such as OKL0001000. **(required)**

other_names (*array*) Alternate or former names for this object. **(required, minItems: 0)**

Each element in other_names is an object with the following keys:

note (*string, null*) An optional note describing where this alternate name came from or its relationship to the entity. **(required)**

name (*string*) An alternate name this object is sometimes known by. **(required)**

end_date (*string, null*) The date at which this name was no longer valid. (null if still valid/valid indefinitely) **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

start_date (*string, null*) The date at which this name became valid.(null if unknown/valid indefinitely) **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

Common Fields

updated_at (*string, datetime, null*) The time at which the resource was last modified **(required)**

created_at (*string, datetime, null*) The time at which the resource was created **(required)**

sources (*array*) URLs for sources relating to the object **(required, minItems: 1)**

Each element in sources is an object with the following keys:

url (*string*) URL of resource used to collect information **(required)**

note (*null, string*) note about what information this URL was used for **(required)**

Bill Objects

Warning: Parts of Open Civic Data underwent a large refactor as of mid-2014, some information on this page may be out of date. We're working on updating this documentation as soon as possible.
We'll remove these messages from pages as they're updated and vetted.

Bill objects have the following fields:

Basics

_type (*string*) All bills have a _type field set to bill. **(required)**

Allowed values:

- bill

organization (*string, null*) name of the legislative body that this bill belongs to **(required)**

organization_id (*string, null*) ID of legislative body that this bill belongs to **(required)**

session (*string*) associated with one of the jurisdiction's sessions **(required)**

name (*string*) jurisdiction-assigned permanent name. Must be unique within a given session (e.g. HB 3). Note: not to be confused with `title`. **(required)**

chamber (*string, null*) chamber vote took place in (if legislature is bicameral, otherwise null) **(required)**

Allowed values:

- upper
- lower
- joint

title (*string*) primary display title for the bill **(required)**

type (*array*) array of types (e.g. bill, resolution) **(required, minItems: 0)**

Each element in type is of type (string)

subject (*array*) List of related subjects. **(required, minItems: 0)**

Each element in subject is of type (string)

summaries (*array*) List of summaries of bill, each item in list has a note and text attribute. **(required, minItems: 0)**

Each element in summaries is an object with the following keys:

note (*string, null*) note describing source of summary **(required)**

text (*string*) Summary of bill. **(required)**

Common Fields

updated_at (*string, datetime*) the time that the object was last updated

created_at (*string, datetime*) the time that this object was first created

sources (*array*) URLs for sources relating to the object **(required, minItems: 1)**

Each element in sources is an object with the following keys:

url (*string*) URL of resource used to collect information **(required)**

note (*null, string*) note about what information this URL was used for **(required)**

Other/Related Bills

other_titles (*array*) list of other titles this bill is known by.A common use is when a state provides a common title and a long or technical title as well. It is also acceptable to include popular but unofficial titles of the bill as well, such as'Obamacare' for the 'Patient Protection and Affordable Care Act.'*note* can be used to describe the relationship this has to the bill, for example Obamacare might be noted as a colloquial name. Each item in the list has a title and a note. **(required, minItems: 0)**

Each element in other_titles is an object with the following keys:

note (*string, null*) Note describing source. **(required)**

title (*string*) Alternate title. **(required)**

other_names (*array*) list of other names this bill is known by in the current session, for example if HB 33 and SB 17 refer to the same bill this prevents having to have identical entries for each. **(required, minItems: 0)**

Each element in other_names is an object with the following keys:

note (*string, null*) note describing why this name is attached **(required)**

name (*string*) name (e.g. HB 22) **(required)**

related_bills (*array*) Links to related bills. Currently only used for companion bills, but extensible for other uses. **(required, minItems: 0)**

Each element in related_bills is an object with the following keys:

session (*string*) Session of related bill. **(required)**

name (*string*) Name of related bill. **(required)**

relation_type (*string*) **(required)**

Allowed values:

- companion
- prior-session
- replaced-by
- replaces

Sponsors and Actions

sponsors (*array*) List of entities responsible for sponsoring/authoring the bill. **(required, minItems: 0)**

Each element in sponsors is an object with the following keys:

_type (*string, null*) Type of entity if the sponsor has been resolved to another entity in the database. **(required)**

Allowed values:

- organization
- person

name (*string*) Name of sponsor, as given by source. **(required)**

sponsorship_type (*string*) Type of sponsorship, via upstream source. **(required)**

primary (*boolean*) Indicates if sponsor is considered primary by source **(required)**

chamber (*string, null*) Chamber of sponsor. **(required)**

Allowed values:

- upper
- lower

id (*string, null*) ID of entity if the sponsor has been resolved to another entity in the database. **(required)**

actions (*array*) List of actions taken on the bill. **(required, minItems: 0)**

Each element in actions is an object with the following keys:

date (*string*) date of action **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

type (*array*) array of normalized action types **(required, minItems: 0)**

Each element in type is of type (string)

description (*string*) description of the action taken as given by source **(required)**

actor (*string, null*) name for the actor (e.g. 'upper', 'lower', etc.) **(required)**

related_entities (*array*) list of related entities for the action, such as related committee for a referral or a person for a sponsorship. **(required, minItems: 0)**

Each element in related_entities is an object with the following keys:

_type (*string, null*) Type of entity if the sponsor has been resolved to another entity in the database. **(required)**

Allowed values:

- organization
- person

name (*string*) Name of entity given by source data **(required)**

id (*string, null*) ID of entity if the sponsor has been resolved to another entity in the database. **(required)**

Documents and Versions

documents (*array*) Any non-version related documents, elements are identical to versions. **(required, minItems: 0)**

Each element in documents is an object with the following keys:

date (*string, null*) Document posting date **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

type (*string, null*) Type of document **(required)**

name (*string*) Name of document **(required)**

links (*array*) List of links to text for this document (pdf, html, etc.). **(required, minItems: 0)**

Each element in links is an object with the following keys:

media_type (*string*) IANA Media Type of document **(required)**

url (*string*) URL to document **(required)**

text (*string*) Text of the document

versions (*array*) Versions of a bill's text (First Printing, As Amended, etc.) **(required, minItems: 0)**

Each element in versions is an object with the following keys:

date (*string, null*) Version posting date **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

type (*string, null*) Type of version **(required)**

name (*string*) Name of version **(required)**

links (*array*) List of links for this version (pdf, html, etc.). **(required, minItems: 0)**

Each element in links is an object with the following keys:

media_type (*string*) IANA Media Type of document **(required)**

url (*string*) URL to document **(required)**

text (*string*) Text of the document

Vote Objects

> **Warning:** Parts of Open Civic Data underwent a large refactor as of mid-2014, some information on this page may be out of date. We're working on updating this documentation as soon as possible.
> We'll remove these messages from pages as they're updated and vetted.

Vote objects have the following fields:

Basic Fields

organization (*string, null*) name of the voting organization **(required)**

organization_id (*string, null*) id of the voting organization **(required)**

_type (*string*) All vote objects must have a _type field set to vote. **(required)**

Allowed values:

- vote

session (*string*) Associated with one of the jurisdiction's sessions **(required)**

chamber (*string, null*) chamber vote took place in (if legislature is bicameral, otherwise null) **(required)**

Allowed values:

- upper
- lower
- joint

date (*string*) date of the action **(required)**

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

motion (*string*) description of motion (from upstream source) **(required)**

type (*array*) array of types **(required, minItems: 0)**

Each element in type is of type (string)

Allowed values:

- bill-passage
- amendment-passage
- veto-override

passed (*boolean*) boolean indicating if vote passed **(required)**

Common Fields

updated_at (*string, datetime*) the time that the object was last updated

created_at (*string, datetime*) the time that this object was first created

sources (*array*) URLs for sources relating to the object **(required, minItems: 1)**

Each element in sources is an object with the following keys:

url (*string*) URL of resource used to collect information **(required)**

note (*null, string*) note about what information this URL was used for **(required)**

Relationship to Bill

bill (*object, null*) Related bill, votes will have a non-null bill object ifthey are related to a bill. Bills will have the following fields: **(required)**

chamber (*string, null*) bill's chamber if vote was on a bill (and legislature is bicameral, otherwise null) **(required)**

Allowed values:

- upper
- lower

id (*string, null*) bill's internal id if bill was matched with an object in the database **(required)**

name (*string*) bill name (e.g. HB 21) **(required)**

Vote Counts

vote_counts (**) list of objects with vote_type and count properties **(required)**

Each element in vote_counts is an object with the following keys:

count (*integer*) number of people voting this way **(required)**

(minimum value: 0)

vote_type (*string*) (e.g. yes, no, not-voting) **(required)**

Allowed values:

- yes
- no
- absent
- abstain
- not voting
- paired
- excused
- other

roll_call (**) list of individual legislator votes **(required)**

Each element in roll_call is an object with the following keys:

person (*object*) person object representing the voter **(required)**

name (*string*) person's name as provided by the source **(required)**

id (*string, null*) person's internal id if they've been matched to an entity in the database **(required)**

vote_type (*string*) (e.g. yes, no, not-voting) **(required)**

Allowed values:

- yes
- no
- absent
- abstain
- not voting
- paired
- excused
- other

Event Objects

Event objects have the following fields:

Basics

_type (*string*) All events must have a _type field set to one of the entries in the enum below. **(required)**

Allowed values:

- event

name (*string*) A simple name of the event, such as "Fiscal subcommittee hearing on pudding cups" **(required)**

description (*string*) A longer description describing the event. As an example, "Topics for discussion include this that and the other thing. In addition, lunch will be served". **(required)**

classification (*string*) An event category, e.g. town hall. **(required)**

start_time (*datetime*) Starting date / time of the event. This should be fully timezone qualified. **(required)**

timezone (*string*) Time zone of the event. **(required)**

end_time (*datetime, null*) Ending date / time of the event. This should be fully timezone qualified. **(required)**

all_day (*boolean*) Whether the event is a full-day event. **(required)**

status (*string*) String that denotes the status of the meeting. This is useful for showing the meeting is cancelled in a machine-readable way. **(required)**

Allowed values:

- cancelled
- tentative
- confirmed
- passed

location (*object, null*) Where the event will take place. **(required)**

url (*string*) URL of the location, if applicable.

name (*string*) name of the location, such as "City Hall, Boston, MA, USA", or "Room E201, Dolan Science Center, 20700 North Park Blvd University Heights Ohio, 44118" **(required)**

coordinates (*object, null*) coordinates where this event will take place. If the location hasn't (or isn't) geolocated or geocodable, than this should be set to null. **(required)**

latitude (*string*) latitude of the location, if any **(required)**

longitude (*string*) longitude of the location, if any **(required)**

Linked Entities

media (*array*) This "special" schema is used in two places in the Event schema, on the top level and inside the agenda item. This is an optional component that may be omited entirely from a document. **(required, minItems: 0)**

Each element in media is an object with the following keys:

note (*string*) Human-readable name of the media link, such as "Recording of the meeting" or "Discussion of construction near the watershed" **(required)**

date (*string*) Date of the recording

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

offset (*number, null*) Offset where the related part starts. This is optional and may be ommited entirely. **(required)**

links (*array*) List of links to the same media item, each with a different MIME type. **(required, minItems: 0)**

Each element in links is an object with the following keys:

media_type (*string*) IANA Media Type of the media, such as video/mp4 or audio/webm **(required)**

url (*string*) URL where this media may be accessed **(required)**

text (*string*) Text of the media

links (*array*) Links related to the event that are not documents or items in the Agenda. This is filled with helpful links for the event, such as a committee's homepage, reference material or links to learn more about subjects related to the event. **(required, minItems: 0)**

Each element in links is an object with the following keys:

note (*string*) Human-readable name of the link. Something like "Historical precedent for popsicle procurement"

url (*string*) A URL for a link about the event **(required)**

participants (*array*) List of participants in the event. This includes committees invited, legislators chairing the event or people who are attending. **(required, minItems: 0)**

Each element in participants is an object with the following keys:

note (*string*) Note regarding the relationship, such as *chair* for the chair of a meeting. **(required)**

name (*string*) Human-readable name of the participant. **(required)**

entity_type (*string*) What type of entity is this? *person* may be used if the person is not a legislator, but attending the event, such as an invited speaker or one who is offering testimony.

Allowed values:

- organization
- person

entity_name (*string*) Reconciled name of the participant. **(required)**

entity_id (*string, null*) Reconciled ID of the participant. **(required)**

agenda (*array*) Agenda of the event, if any. This contains information about the meeting's agenda, such as bills to discuss or people to present. **(required, minItems: 0)**

Each element in agenda is an object with the following keys:

description (*string*) Human-readable string that represents this agenda item. A good example would be something like The Committee will consider SB 2339, HB 100 **(required)**

classification (*string*) An agenda item category.

order (*string*) order of this item, useful for re-creating meeting minutes. This may be ommited entirely. It may also optionally contains "dots" to denote nested agenda items, such as "1.1.2.1" or "2", which may go on as needed.

subjects (*array*) List of related topics of this agenda item relates to. **(minItems: 0)**

Each element in subjects is of type (string)

notes (*array*) List of notes taken during this agenda item, may be used to construct meeting minutes. **(minItems: 0)**

Each element in notes is of type (string)

‘

related_entities (*array*) Entities that relate to this agenda item, such as presenters, legislative instruments, or committees. **(required, minItems: 0)**

Each element in related_entities is an object with the following keys:

note (*string*) Human-readable string (if any) noting the relationship between the entity and the agenda item, such as "Jeff will be presenting on the effects of too much cookie dough" **(required)**

name (*string*) Human-readable string representing the entity, such as *John Q. Smith*. **(required)**

entity_type (*string*) Type of the related object.

Allowed values:

- bill
- organization
- person
- vote_event

entity_name (*string*) Reconciled name of the entity. **(required)**

entity_id (*string, null*) Reconciled ID of the entity. **(required)**

media (*array*) This "special" schema is used in two places in the Event schema, on the top level and inside the agenda item. This is an optional component that may be omited entirely from a document. **(required, minItems: 0)**

Each element in media is an object with the following keys:

note (*string*) Human-readable name of the media link, such as "Recording of the meeting" or "Discussion of construction near the watershed" **(required)**

date (*string*) Date of the recording

(must match format: `^[0-9]{4}(-[0-9]{2}){0,2}$`)

offset (*number, null*) Offset where the related part starts. This is optional and may be ommited entirely. **(required)**

links (*array*) List of links to the same media item, each with a different MIME type. **(required, minItems: 0)**

Each element in links is an object with the following keys:

media_type (*string*) IANA Media Type of the media, such as video/mp4 or audio/webm **(required)**

url (*string*) URL where this media may be accessed **(required)**

text (*string*) Text of the media

documents (*array*) Links to related documents for the event. Usually, this includes things like pre-written testimony, spreadsheets or a slide deck that a presenter will use. **(required, minItems: 0)**

Each element in documents is an object with the following keys:

note (*string*) Human-readable name of the document. Something like "Fiscal Report" or "John Smith's Slides". **(required)**

date (*string*) Document posting date **(required)**

media_type (*string*) IANA Media Type of document **(required)**

url (*string*) URL to document **(required)**

text (*string*) Text of the document

links (*array*) List of links to text for this document (pdf, html, etc.). **(required, minItems: 0)**

Each element in links is an object with the following keys:

media_type (*string*) IANA Media Type of document **(required)**

url (*string*) URL to document **(required)**

text (*string*) Text of the document

Common Fields

updated_at (*string, datetime*) the time that this object was last updated.

created_at (*string, datetime*) the time that this object was first created.

sources (*array*) URLs for sources relating to the object **(required, minItems: 1)**

Each element in sources is an object with the following keys:

url (*string*) URL of resource used to collect information **(required)**

note (*null, string*) note about what information this URL was used for **(required)**

OCD Identifiers

Open Civic Data Identifiers (or OCD IDs) are a common Identifier format used in the Open Civic Data projects, in a defined format, ripe for reuse with any legislative dataset.

Creating a new OCD ID

Consensus on IDs is needed for a few of the types, but other IDs may be issued without any concern at all. The following is a helpful table of when it's OK (and not OK) to create new IDs without reaching rough consensus.

OCD ID Type	Can issue new ID
person	Yes (UUID1)
organization	Yes (UUID1)
division	No (Needs to undergo a review and survey of entries at that geopolitical level)
jurisdiction	No (needs to undergo a review to ensure we have consistent names for legislative bodies)

If you need to create a new ID that requires rough consensus, emailing the Open Civic Data mailing list with as much detail regarding the situation as you can generally proves to be the best way to solicit feedback.

General Format

OCD IDs have the general format of: `ocd-${type}/${data}`. Some valid types are `division`, `jurisdiction`, and `person`. Each type has its own format (for the data half of the ID), and a brief overview can be found below.

Division IDs

Division IDs are one of the more common OpenCivic identifiers. Division IDs denote a particular geopolitical division. Information regarding valid Division IDs can be found in OCDEP 2: Open Civic Data Divisions.

The general format is: `ocd-division/country:<country_code>[/<type>:<type_id>]+`. `country_code` must be a valid ISO 3166-1 alpha-2 code for the country. `type` shall be the type of boundary (such as `country`, `state`, `city`), while `type_id` shall be the unique ID for the entity at this level.

For more information on what exactly is correct in this format, please do take a look at OCDEP 2: Open Civic Data Divisions.

Jurisdiction IDs

Jurisdiction IDs are based on the Division IDs, but have a slightly adjusted format. The `type` shall be set to `jurisdiction`, and the data half of the ID shall have a trailing `type`, which matches the jurisdiction type.

The ID looks something like `ocd-jurisdiction/country:us/state:ex/place:example/legislature`.

This format isn't fully formalized yet, so please take care when using these.

Person IDs, Org IDs

The valid types are `person` for a Person, and `organization` for an Organization.

Person and Org IDs contain a UUID for the data-part, created by pupa using `uuid.uuid1`.

An example of a valid OCD Person ID is `ocd-person/ebaff054-05df-11e3-a53b-f0def1bd7298`.

CHAPTER 3

Style Guidelines

General

Version Control

Code is managed in git. Changes should contain clear, descriptive English text describing the thought that went into why you're making the change, rather than describing what you changed.

In two weeks, It's a lot more helpful to know *why* you changed *foo* to *foo_with_bar*, than read a commit message that says *change foo to be foo_with_bar*.

The first line of a Git commit should be 50 chars or less, followed by a blank line, followed by a longer description of the changeset (if required). The long description should contains lines that are all under 72 chars.

Line Length

Please try to keep line length under 80 chars wide, 100 characters should be considered the hard limit.

Open Civic Data Workflow

Submitting Changes

All changes should be submitted in the form of a Pull Request. Small changes, even ones that appear to be quite simple, can often prove to cause issues down the line.

Suggested Git Branching Model

It's strongly encouraged to use a sane Git branching model, one such model is:

Maintain two remotes:

```
upstream: Open Civic Data repo
origin:   Fork of the repo
```

Always keep *upstream/master*, *origin/master* and *refs/heads/master* 100% ABSOLUTELY in sync. Before making a new branch, or sending in a PR, give master a pull, and make sure things are all sync'd nicely.

Here's an example of creating a new branch:

```
git checkout master
git checkout -b paultag/bugfix/fix-typo-in-readme
git push -u origin paultag/bugfix/fix-typo-in-readme
```

It should go without saying that both *paultag* and *bugfix* should be changed to match your username, and the flavor of branch is usually something like *feature*, or *bugfix*.

Python Code Guidelines

Python Version

Please use an up to date Python. All new development is Python 3.4+ only. All efforts to support older versions of Python 3, or even Python 2 are on a purely best-effort basis, and large refactors of code to make it Python 2 compatable will likely be rejected.

Code Standards

All code must follow PEP 8. You may check compliance with PEP8 by using the *flake8* tool.

```
pip install flake8
flake8 .
```

Please address outstanding *flake8* issues. Any test suites should also test code style.

Comments

Please add comments and descriptive docstrings to your code. Clearly, if the code doesn't require them, that's OK, but comments can be quite helpful later on.

Sprinkle them around like sriracha.

Trailing Spaces

Please ensure that we don't have any trailing spaces on any code lines or commit lines.

CHAPTER 4

Open Civic Data Enhancement Proposals

OCDEP 1: Purpose and Guidelines

Created 2014-06-05

Author James Turk

Status Accepted

What is an OCDEP?

An OCDEP (Open Civic Data Enhancement Proposal) is a design document providing information to the Open Civic Data community, or describing a new feature or process for Open Civic Data. OCDEPs provide concise technical specifications of features, along with rationales.

We intend the proposal process to be the primary mechanism for proposing major new features, for collecting community input on issues, and for documenting design decisions that have gone into Open Civic Data.

The concept of OCDEPs is a blatant copy of Python's PEPs (http://www.python.org/dev/peps/) and Django's use of DEPs.

Rationale

We're introducing the concept of OCDEPs in Open Civic Data in order to formalize the process of making significant changes to the project.

Now that Open Civic Data is beginning to gain traction, we want to use this to avoid duplicate effort (by announcing intentions) and to avoid developers from feeling like they're building atop unstable ground.

While there will still be discussions that take place on the Google Group (https://groups.google.com/forum/#!forum/open-civic-data) and in IRC, etc. it is expected that all major decisions will take place via the approval of an OCDEP.

When to write a proposal

Taking inspriration from Django's DEP-0001 (https://github.com/django/deps/blob/master/deps/0001.rst) we too wish to avoid introducing "bureaucracy for bureaucracy's sake" and so have a relatively simple process reflecting the small size of the project.

Proposals should be for significant additions or changes, especially those that would introduce some sort of compatibility issue for downstream users.

Examples of things that will fall under the proposal process:

- addition of a new type to Open Civic Data
- backwards incompatible changes to the API response format
- significant new functionality to the API (such as a new method, probably not a single new field in a response)

And things that will *not* require a proposal:

- purely technical changes, such as a backend refactor
- bugfixes

Submitting a Proposal

Our process is intentionally lean at the moment, as we try to get it up and running.

To submit a proposal, write a text document in the and submit it as a pull request to this GitHub repository (https://github.com/opencivicdata/docs.opencivicdata.org). Put it in the "proposals/drafts" directory of the repository and give it a short name that describes the feature, e.g. "formal-xml-serialization.rst".

The proposal format is reStructuredText, with "Created," "Author" and "Status" fields near the top. For example, look at the top of this document. "Status" should be "Draft" to start.

Beyond that, a proposal should include the following sections:

- Overview. A sentence or short paragraph describing the feature.
- Rationale. A few paragraphs describing why this feature is needed and what specific problem(s) it solves.
- Implementation. A technical description of how this feature will be implemented. This may or may not include code snippets.
- Copyright. A statement placing the document in the public domain via the CC0 1.0 Universal License. For example, see the bottom of this document.

Once you've written a proposal and submitted the pull request, post a message about it to the open-civic-data mailing list (https://groups.google.com/forum/#!forum/open-civic-data).

At that point, the core Open Civic Data team will make sure it's technically feasible, not spam, etc., assign it a number and commit it to the repository. This doesn't mean the feature will be implemented; it merely means the proposal is officially an OCDEP.

Discussion and Acceptance of a Proposal

After something has been proposed, the merits will be discussed, the proposal edited, with a goal of reaching a general consensus.

As a guiding principle these discussions should be as open as possible, suitable places include:

- Issues or pull requests on docs.opencivicdata.org issue tracker (tag these with the *proposal* tag).
- The Open Civic Data mailing list.

Note that final acceptance (or rejection) must be announced via the mailing list, and to include as many people as possible it is recommended that questions needing broad consensus take place there, while GitHub is more suitable for more isolated concerns and revisions.

At this point the proposal status will be changed (from Draft) to Accepted, Rejected, or Withdrawn.

If an agreement cannot be reached the core Open Civic Data team reserves the right to accept or reject the proposal. Additionally, the core team may choose to set the status to Deferred, indicating that no decision has been made and the issue can be revisited at some future point.

Implementation of a Proposal

Once a proposal has been marked as Accepted, implementation can begin (if it hasn't already) and upon reasonable completion the proposal will be marked as Final, indicating that future changes to this feature or component will require an additional proposal.

Copyright

OCDEP 2: Division Identifiers

Created 2014-06-06

Author James Turk

Status Accepted

Overview

Definition and procedures around defining Open Civic Data Divisions and their identifiers.

Definitions

Division A political geography such as a state, county, or congressional district, which may have multiple boundaries over its lifetime. Types of divisions include, among others:

- Governmental jurisdiction - A division that a government has jurisdiction over. (e.g. North Carolina)
- Political district - A division that elects a representative to a legislature. (e.g. North Carolina Congressional District 4)
- Service zone - An area to which a government provides a service. (e.g. Washington DC Police District 105)

Boundary A geographical boundary, defined by a shapefile or a sequence of address ranges. (e.g. NC Congressional District 10 for the 113th Congress)

This document describes an identifier scheme for assigning globally unique identifiers to divisions. It does **not** intend to describe any scheme for boundaries.

Rationale

Divisions can be seen as the smallest building block in the Open Civic Data ecosystem, Jurisdictions and Organizations will exist within a Division and People are elected to represent a Division. As such, providing unique identifiers enables collaboration across groups dealing with any of these types.

This proposal in fact predates the formal proposal process by a full year, originally part of the ocd-division-ids repository, the ids are already in use by Sunlight, Google, Granicus, Open North, Open Elections, and several other projects. (This document simply exists to formalize what was already decided.)

Implementation

Identifier Scheme

Identifiers are in the format `ocd-division/country:<country_code>(/<type>:<type_id>)*`

country_code An ISO-3166-1 alpha-2 code.

type The type of boundary. (e.g. *country*, *state*, *town*, *city*, *cd*, *sldl*, *sldu*)

- Valid characters are lowercase UTF-8 letters, hyphen (-), and underscore (_).
- Use existing types where possible.

type_id An identifier that is locally unique to its scope.

- Valid characters are lowercase UTF-8 letters, numerals (0-9), period (.), hyphen (-), underscore (_), and tilde (~). These characters match the unreserved characters in a URI RFC 3986 section 2.3.
- Characters must be converted to UTF-8.
- Uppercase characters must be converted to lowercase.
- Spaces must be converted to underscores.
- All invalid characters must be converted to tildes (~).
- Leading zeros should be removed unless doing so changes the meaning of the identifier.

Assignment

- If possible, all divisions of the same type should be defined at the same time; for example, all state divisions should be defined at once. Similarly, all cities in North Carolina should be defined at once, to avoid adopting a scheme that produces collisions.
- When selecting a `type_id`, preference should be given to existing, common identifiers, like postal abbreviations for US states. Numeric identifiers (such as US county FIPS codes) should be avoided if textual names are clear and unambiguous; however, numeric identifiers may be appended to disambiguate a `type_id`.
- The set of types within each country should not grow unnecessarily. Each country maintainer should publish a list of types for easy reference. The addition of a new type must be justified.
  - For example: In the US, there are no clear-cut differences between cities, towns, villages, etc. Therefore, the Census-recommended term `place` is used as the type of cities, etc.

Repository layout

The `identifiers` directory contains CSV files assigning all OCD identifiers:

- A single CSV file per country, in the format `country-<country_code>.csv`.
  - The URLs of these files are stable.
- An optional directory per country, in the format `country-<country_code>`:
  - A file hierarchy, in which CSV files describe parts of the top-level country CSV file.

  - The URLs of these files are **not** stable.

The `corrections` directory contains CSV files that map incorrect OCD identifiers to correct OCD identifiers. Common errors include missing diacritics, differences in hyphenation and word order, use of Roman numerals, etc.

Identifiers

If a CSV file has no header row, the CSV is assumed to have two columns with the headers *id* and *name*.

If a CSV file has a header row, the first column name must be *id*.

Column names with special meaning are:

name The name of the division.

sameAs An OCD identifier which identifies the same division as this identifier. The row corresponding to the identifier in this column must have a blank value in its *sameAs* column, i.e. there must be no daisy-chaining or circular references.

sameAsNote A note describing how or why the division has multiple identifiers.

validThrough The date on which the division is no longer valid, in the format `YYYY`, `YYYY-MM` or `YYYY-MM-DD`. A division may become invalid if, for example, a political district is abolished.

validFrom The date on which a division becomes valid, in the format `YYYY`, `YYYY-MM` or `YYYY-MM-DD`. A division may become valid if, for example, a political district is created.

- There are no restrictions on other columns.
- An effort should be made to use descriptive CSV filenames.

Corrections

A correction CSV file must contain:

incorrectId An incorrect OCD identifier, i.e. an OCD identifier that was never valid.

id The corrected OCD identifier.

note Free-text describing the error, e.g. "missing diacritics".

Semantics

- All OCD identifiers are first-class. However, if it is necessary for a system for choose a "primary" or "preferred" identifier for a division, it should use those identifiers with an empty `sameAs` column.
- The `sameAs` relationship is symmetric and transitive. The `sameAs` relationship is not true for all time; it is only true in the present.

Governance

This project has an informal governance structure, led by the project's early contributors and informed by the Open Civic Data Google Group. Responsibility for a country's identifiers may be assigned to country-specific organizations.

Examples

United States ocd-division/country:us

North Carolina ocd-division/country:us/state:nc

North Carolina 2nd Congressional District ocd-division/country:us/state:nc/cd:2

North Carolina State Lower Legislative District 1 ocd-division/country:us/state:nc/sldl:1

Wake County, North Carolina ocd-division/country:us/state:nc/county:wake

Cary, North Carolina (*note that despite being within Wake County this is not indicated due to not being an identifying feature*)
ocd-division/country:us/state:nc/place:cary

Kildaire Farms Homeowners Association, Cary, North Carolina ocd-division/country:us/state:nc/place:cary/hoa:kildaire_farms

Washington DC, Ward 8 ocd-division/country:us/district:dc/ward:8

Washington DC, ANC 4A ocd-division/country:us/district:dc/anc:4a

Washington DC, ANC 4A, section 08 (*note: this is a strict subset of the ANC for purposes of representation*)
ocd-division/country:us/district:dc/anc:4a/section:8

New York City, City Council District 36 (happens to be in Brooklyn- but not significant to include in id)
ocd-division/country:us/state:ny/place:new_york/council_district:36

Canadian Federal Electoral District 13004 aka Fundy Royal (known as Royal from 1914-1966, Fundy-Royal from 1966-2003, ar
ocd-division/country:ca/ed:13004

Copyright

OCDEP 3: Jurisdictions

Created 2014-06-12

Author James Turk

Status Accepted

Overview

Definition of the Open Civic Data Jurisdiction type.

Rationale

A Jurisdiction represents a logical unit of governance.

Examples would include: the United States Federal Government, the Government of the District of Columbia, the Lexington-Fayette Urban County Government, or the Wake County Public School System.

The following would not be considered Jurisdictions:

- Bethesda, MD - Bethesda is a Census Designated Place and as such has no formal government.

- North Carolina's General Assembly - The General Assembly is part of the state government, and would be an `Organization` within North Carolina's government.

Additionally, Open Civic Data Jurisdictions should not be confused with the concept of judicial jurisdiction, which is an altogether different issue that we do not attempt to address.

All entities within the Open Civic Data ecosystem are related (directly or indirectly) to a Jurisdiction, and so, along with Divisions (which Jurisdictions themselves depend upon), Jurisdictions can be viewed as one of the foundational pieces of Open Civic Data.

Implementation

The Jurisdiction type has the following properties:

id An ID in the format `ocd-jurisdiction/country:<country_code>(/<type>:<type_id>)*/<classification` (where the first part of the ID is identical to the related `Division`.)

The following pattern should be used create a `jurisdiction_id` given a `division_id` and `classification`:

```
jurisdiction_id = (division_id.replace('ocd-division', 'ocd-jurisdiction') +
                   '/' + classification)
```

name The common name of the Jurisdiction, such as 'Wyoming' or 'Hope County School System'

url The primary website of the Jurisdiction.

classification The type of jurisdiction being defined, current options are:

- *government* - A combined government for a city, county, state, or country where the legislature and executive (and possibly judicial) branches form a cohesive whole that should be considered as one. (The United States is one such example.)
- *legislature* - A legislature in a region in which there is no unified legislative-executive government. An example would be a Parliament in Westminster systems.
- *executive* - An executive branch in a region in which there is no unified legislative-executive government. An example would be the cabinet in Westminster systems.
- *school* - A school system that is independent from a city/county government.
- *park* - An independent park district.
- *sewer* - An independent sewer district.
- *forest* - An independent forest preserve district.
- *transit_authority* - An independent transit district or authority.

division, division_id A link to an Open Civic Data division (or the object itself embedded within the Jurisdiction).

legislative_sessions A list of sub-objects representing times when the jurisdiction's legislature has met (if one exists).

identifier An identifier that uniquely identifies the session within the context of the Jurisdiction. (e.g. 2009)

name The canonical name of the session. (e.g. 2009 Regular Session)

classification The type of session, choices are:

- primary - A regularly scheduled session.
- special - Any session that is not regularly scheduled.

start_date Start date of session in YYYY-MM-DD format (may be approximate by leaving off -MM-DD or -DD portion).

end_date End date of session in YYYY-MM-DD format (may be approximate by leaving off -MM-DD or -DD portion).

feature_flags A list of features that are present for data in this jurisdiction. The definition of these flags is currently left up to individual implementors.

Copyright

OCDEP 4: Events

Created 2014-06-09

Author Paul Tagliamonte

Status Accepted

Overview

Definition of Open Civic Data Events.

Definitions

Event Something happening at a given time and location, past, present or future.

Media Audio and/or Video recordings.

Agenda Points to be discussed at the event, as defined in the Agenda section below.

Rationale

A core part of the democratic process is ensuring that citizens have representation and a say in how the government operates. One important way in which this happens is through hearings and opportunities for public testimony. As such, Open Civic Data Events need to be able to store data such as hearings, and provide structure to represent this aspect of government.

Implementation

Core Fields

name Name of the event, examples include "Fiscal committee meeting on April 10th" or "Appropriations - S/C on Article II"

classification Classification of the event. Current values (which may be expanded) are `committee-meeting` and `hearing`.

start_time Starting time of the event. This field is serialized as an ISO 8601 datetime string, normalized to UTC, and must be fully datetime aware. ISO 8601 defines times expressed in UTC with a special UTC designator ("Z") as the timezone.

timezone This must always be set to the timezone in which the event is held. This also aids in display, since you may always know what the local time of the event is.

Optional Fields

agenda List of agenda items as defined in the `Agenda Items` section below.

description Description of the Event.

location Location object, as defined in the Location section below.

all_day Boolean value set to boolean `True` if the event is an all-day event, otherwise it must be set to the boolean value `False`.

end_time Ending time of the event. This field is serialized as an ISO 8601 datetime string, normalized to UTC, and must be fully datetime aware. ISO 8601 defines times expressed in UTC with a special UTC designator ("Z") as the timezone.

status

Status of the event, values are currently:

- `cancelled` for a cancelled event
- `tentative` for an event which is scheduled but not confirmed.
- `confirmed` for an event which is confirmed as happening.
- `passed` for a past event.

links Links associated with the event, as defined in the Links section below.

participants Participants associated with the event, as defined in the Participants section below.

documents List of documents associated with the Event, as described in the `Documents` section below.

media List of media objects, as defined in the `Media` section below.

Location

name Name of the location, such as "City Hall, Boston, MA, USA", or "221B Baker Street".

note Optional human-readable note to help with getting to the meeting place, such as "The meeting will take place at the Minority Whip's desk on the floor"

url Optional URI of the venue, such as a webpage.

coordinates Object to store the exact coordinates of the venue. This object has two keys, `latitude` and `longitude`.

Participants

name Name of the participant

id Open Civic Data ID of the participants

type Type of the participant, either `person` or `organization`.

note Human-readable note regarding the relationship

Agenda Items

Agenda items are the list of topics to be discussed at the meeting, as well as metadata related to the topic or proceeding.

Required Fields

description Description of the agenda item. Examples include "Consideration of SB 2339, HB 100" or "John Q. Public will give testimony".

Optional Fields

related_entities List of objects as defined in the `Related Entities` section below.

media List of media objects, as defined in the `Media` section below.

documents List of documents associated with the Agenda Item, as described in the `Documents` section below.

notes List of strings, which store any notes taken during the event while discussing this item.

subjects List of subjects (stored as strings) of this item.

Related Entities

entity_type Type of the related entity, such as `bill`, `person`, or `organization`.

id Open Civic Data ID of the entity.

name Human-readable name of this entity, such as "John Q. Smith", or "HB 101".

note Optional note regarding the relation between this entity and the agenda item, such as "Jeff will be presenting on the effects of the watershed contstruction".

Documents

Related documents to the event, things like pre-written testimony, spreadsheets or a slide deck should be related here.

note Relationship of the document to the event. Examples include "Fiscal Report" or "John Smith's slides"

url URI where the content may be found.

media_type Full IANA media type of the remote content.

Links

Links are a list of related URLs which are not documents, media or agenda items. Examples include a committee's homepage, reference material or links to learn more about subjects related to the event.

note Human-readable name of the link.

url URI pointing to the resource referenced.

Media

Media, most often recordings of the event, are extremely useful for those who wish to review what was dicsussed at a meeting, and provides a record of resolutions to issues discussed.

Required Fields

name A required string that describes the particular media link. Examples include "Discussion of the contstruction near the watershed" or "Fiscal committee meeting on April 9th."

type What kind of media this is, as defined below in the section `Defined Types`.

links List of objects which contain two keys, `url` and `media_type`. `url` is a fully qualified URI to the media. `media_type` is the full IANA media type of the content at the URI.

Optional Fields

date A string in `YYYY-MM-DD` format which stores the date of the recording.

offset Integer offset into the media to use as a starting point. This is defined as a nonzero positive integer, the value of which is the number of seconds into the recording to skip to. Implementations should use this (if present) as the starting point when playing back video.

Defined Types

recording A recording of the event

testimony recorded testimony, either pre-recorded and submitted, or a recording made at the event.

Examples

Example:

```
{
    "_id": "ocd-event/146e36d8-d243-11e3-ad6e-f0def1bd7298",
    "_type": "event",
    "agenda": [
        {
            "description": "Opening remarks from Speaker Andy Tobin",
            "order": 1,
            "subjects": [],
            "media": [],
            "notes": [],
            "related_entities": [
                {
                    "id": "ocd-person/072c3c11-cd8c-4544-9ab1-b60486e1832e",
                    "name": "Andy Tobin",
                    "note": "speaker"
                }
            ]
        },
        {
            "description": "Presentation by Director Henry Darwin, Arizona Department of Environmenta
            "order": 2,
            "subjects": ["epa", "green energy", "environmental issues"],
            "media": [],
            "notes": [],
            "related_entities": [
                {
                    "id": "ocd-person/a284a515-32b2-4338-a38d-1938a4ac9f8f",
                    "name": "Henry Darwin",
```

```
                "note": null
            },
            {
                "id": "ocd-organization/684139f7-b5a5-4702-9a92-2be976b29eef",
                "name": "Environmental Protection Agency (EPA)",
                "note": null
            }
        ]
    },
    {
        "description": "Public Testimony",
        "order": 3,
        "subjects": [],
        "media": [],
        "notes": [],
        "related_entities": []
    },
],
"all_day": false,
"description": null,
"documents": [
    {
        "note": "Agenda",
        "url": "http://committee.example.com/agenda.pdf",
        "media_type": "application/pdf",
    }
],
"end": null,
"extras": {},
"links": [
    {
        "name": "EPA Website",
        "url": "http://www.epa.gov/"
    }
],
"location": {
    "coordinates": {
        "latitude": "33.448040",
        "longitude": "-112.097379"
    },
    "name": "State Legislative Building",
    "note": null
},
"media": [
    {
        "date": "2014-04-12",
        "links": [
            {
                "media_type": "video/mp4",
                "url": "http://example.com/video.mp4"
            },
            {
                "media_type": "video/webm",
                "url": "http://example.com/video.webm"
            }
        ],
        "name": "Recording of the meeting",
        "offset": 19,
```

```
            "type": "recording"
        }
    ],
    "name": "Meeting of the Committee on Energy",
    "participants": [
        {
            "id": "ocd-organization/487b972c-4aa6-40e7-b355-0d73580e06e8",
            "name": "Committee on Energy",
            "note": "Host Committee"
        },
        {
            "id": "ocd-person/072c3c11-cd8c-4544-9ab1-b60486e1832e",
            "name": "Andy Tobin",
            "note": "Speaker"
        }
    ],
    "sources": [
        {
            "note": "scraped source",
            "url": "http://example.com/events"
        }
    ],
    "status": "passed",
    "type": "event",
    "start_date": 1408932805.0
}
```

Defined Schema

Schema:

```
media_schema = {
    "items": {
        "properties": {
            "name": { "type": "string" },
            "type": { "type": "string" },
            "date": fuzzy_date_blank,
            "offset": { "type": ["number", "null"] },
            "links": {
                "items": {
                    "properties": {
                        "media_type": { "type": "string", "blank": True },
                        "url": { "type": "string" },
                    },
                    "type": "object"
                },
                "type": "array"
            },
        },
        "type": "object"
    },
    "type": "array"
}

schema = {
    "properties": {
        "name": { "type": "string" },
```

```
"start_time": { "type": "datetime", },
"timezone": { "type": "string" },
"all_day": { "type": "boolean" },
"end_time": { "type": ["datetime", "null"] },
"status": {
    "type": "string", "blank": True,
    "enum": ["cancelled", "tentative", "confirmed", "passed"],
},
"classification": { "type": "string" }, # TODO: enum
"description": { "type": "string", "blank": True, },

"location": {
    "type": "object",
    "properties": {

        "name": { "type": "string", },

        "note": {
            "type": "string", "blank": True,
        },

        "url": {
            "required": False,
            "type": "string",
        },

        "coordinates": {
            "type": ["object", "null"],
            "properties": {
                "latitude": {
                    "type": "string",
                },

                "longitude": {
                    "type": "string",
                }
            }
        },
    },
},

"media": media_schema,

"documents": {
    "items": {
        "properties": {
            "note": { "type": "string", },
            "url": { "type": "string", },
            "media_type": { "type": "string", },
        },
        "type": "object"
    },
    "type": "array"
},

"links": {
    "items": {
        "properties": {
```

```
                "note": {
                    "type": "string",
                    "blank": True,
                },

                "url": {
                    "format": "uri",
                    "type": "string"
                }
            },
            "type": "object"
        },
        "type": "array"
    },

    "participants": {
        "items": {
            "properties": {

                "name": {
                    "type": "string",
                },

                "id": {
                    "type": ["string", "null"],
                },

                "type": {
                    "enum": ["organization", "person"],
                    "type": "string",
                },

                "note": {
                    "type": "string",
                },

            },
            "type": "object"
        },
        "type": "array"
    },

    "agenda": {
        "items": {
            "properties": {
                "description": { "type": "string", },

                "order": {
                    "type": ["string", "null"],
                },

                "subjects": {
                    "items": {"type": "string"},
                    "type": "array"
                },

                "media": media_schema,
```

```
                    "notes": {
                        "items": {
                            "type": "string",
                        },
                        "type": "array",
                        "minItems": 0,
                    },

                    "related_entities": {
                        "items": {
                            "properties": {
                                "entity_type": {
                                    "type": "string",
                                },

                                "id": {
                                    "type": ["string", "null"],
                                },

                                "name": {
                                    "type": "string",
                                },

                                "note": {
                                    "type": ["string", "null"],
                                },
                            },
                            "type": "object",
                        },
                        "minItems": 0,
                        "type": "array",
                    },
                },
                "type": "object"
            },
            "minItems": 0,
            "type": "array"
        },
        "sources": sources,
        "extras": extras,
    },
    "type": "object"
}
```

Further Reading

Many ideas here were based on the work in Open States Schema.

Copyright

OCDEP 5: People, Organizations, Posts, and Memberships

Created 2014-06-12

Author James Turk

Status Accepted

Overview

Adoption of Popolo types for the Open Civic Data Person, Organization, Post, and Membership types.

Definitions

Person An individual that has served in a political office.

Organization A group of people, typically in a legislative or rule-making context.

Post A position in an organization that exists independently of the person holding it.

Membership A relationship between a Person and an Organization, possibly including a Post.

Rationale

At the core of the Open Civic Data mission is helping people discover who represents them and the structure of their government. People, Posts, Organizations, and Memberships are the means to describing that structure.

Furthermore, we have adopted (with minor differences) the Popolo schemas for these objects in the aim of being interoperable with a wide range of civic software that is being built, and due to the technical merits and completeness of the specification.

Implementation

To avoid duplicating the entire Popolo specification, this proposal only aims to highlight the differences (omitted or added fields) between Popolo and the Open Civic Data implementation.

Person

The basis for the Open Civic Data `Person` object is the Popolo Person.

Omitted Fields

Several fields have been omitted:

additional_name, honorific_prefix, honorific_suffix, patronymic_name Due to the inherent fragile nature of trying to segment names, all name fields other than `name`, `sort_name`, `family_name`, `given_name`, and `other_names` are omitted to avoid implying that they can be relied upon.

email Considered redundant with using an entry in `Person.contact_details` with type `email`.

Additional Fields

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Organization

The basis for the Open Civic Data `Organization` object is the Popolo Organization.

Omitted Fields

area, area_id Open Civic Data does not make use of the Popolo `Area` type here, instead favoring the concept of `Jurisdiction` as described in *OCDEP 3: Jurisdictions*. (See *Areas, Divisions, and Jurisdictions*)

Additional Fields

jurisdiction, jurisdiction_id An organization exists as part of a `Jurisdiction` as described in *OCDEP 3: Jurisdictions*. (See *Areas, Divisions, and Jurisdictions*)

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Post

The basis for the Open Civic Data `Post` object is the Popolo Post.

Omitted Fields

other_label This field was added to Popolo after our adoption, and we have not yet found a need to add it.

Additional Fields

division, division_id An post may have an associated `Division`, which is a synonym of Popolo's `Area`. (See *Areas, Divisions, and Jurisdictions*)

These fields are synonymous with `area` and `area_id`. It is strongly suggested that both exist for compatibility reasons.

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Membership

The basis for the Open Civic Data `Membership` object is the Popolo Membership.

Omitted Fields

area, area_id Memberships are not used to relate to defined areas/divisions, this must be done through a `Post`.

member, member_id We do not support Organization-Organization memberships, and therefore use the more specific `person` and `person_id` fields

Additional Fields

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Areas, Divisions, and Jurisdictions

Two main differences have emerged between our handling of political areas and that of Popolo:

The first difference is the use of `Division` instead of `Area` on `Post`. `Division` in Open Civic Data predates `Area`'s inclusion in Popolo, but the two are now essentially synonymous. For compatibility reasons Open Civic Data will continue to use the term Division, but for compatibility with other Popolo-based systems `area` and `area_id` should be present in any API.

Additionally, Open Civic Data uses the concept of Jurisdictions, which represent entities with authority over a given area. (For more detail on Jurisdictions see *OCDEP 3: Jurisdictions*.)

Due to the presence of Jurisdictions, a decision was made to not use `area & area_id` on Organizations, instead `Organization` objects are linked instead to a `Jurisdiction`. A link to `Division` (a synonym for `Area` does exist on `Jurisdiction` so it is still possible to get the `Area` represented by an `Organization`).

Copyright

OCDEP 6: Bills

Created 2014-06-16

Author James Turk

Status Accepted

Overview

Definition of Open Civic Data Bill schema.

Rationale

Bills are the primary artifact of legislatures, proposals to modify or create laws. Open Civic Data Bills are the representation of this concept, which may sometimes be known by other names such as Proposal, Resolution, or Measure.

Implementation

id A unique ID in the format `ocd-bill/{uuid}`.

legislative_session A reference to a `legislative_session` from the relevant `Jurisdiction` (see *OCDEP 3: Jurisdictions* for details). This must contain the `name` attribute at minimum, but `may` contain any valid object that is found in the `Jurisdiction.legislative_session` object.

identifier A name for the bill, such as 'HB 1' or '2117'.

title The current title of the bill, such as 'The Patient Protection and Affordable Care Act'.

from_organization, from_organization_id optional The organization that the bill was originally introduced in. In the US this would be the organization representing the House if it were a House bill or resolution, and the Senate if it were a Senate bill.

This field can be omitted if it is not known or there is not a single chamber of introduction (such as a Joint Resolution).

classification A list of classifications for this bill, suggested values would be things like 'bill', 'resolution', 'constitutional amendment', etc.

subject optional A curated list of subject areas that this bill is a part of.

abstracts A list of objects representing available abstracts (sometimes called summaries) for the bill, each with the following fields:

abstract The text of the abstract.

note optional A note about the origin of the summary, such as "Republican Caucus Summary" or "Library of Congress Summary"

other_titles A list of objects representing alternate titles for the bill.

title The text of an alternate title that someone might use to refer to the bill, an example might be "Obamacare" for the Affordable Care Act.

note optional A note describing the origin of the title.

other_identifiers A list of objects representing alternate identifiers for the bill.

Also note that this is to refer to bills that have multiple names, such as in Tennessee where bills are given a House and Senate number but have shared history. In states where there are two related bills with distinct parallel histories, a second Bill object should be created and the `related_bill` property (described below) should be used.

identifier The alternate identifier (e.g. HB 7)

note optional A note describing the reason for the alternate name.

scheme optional If the identifier belongs to a 3rd-party site (such as OpenStates.org assigned bill ids) it must provide a scheme, scheme should be omitted if it is an identifier from the primary source.

actions A list of objects representing individual actions that take place on a bill, comprising the legislative history of the proposal in question. Actions consist of the following properties:

organization, organization_id The organization that this action took place within.

description Description of the action.

date The date the action occurred in YYYY-MM-DD format. (can be partial by omitting -MM-DD or -DD component).

classification A list of classifications for this actions, suggested values would be things like 'passage', 'introduction', etc.

related_entities A list of all related entities (such as legislators mentioned by name in the action). Each entity has the following fields:

name The upstream-given name of this related entity.

entity_type 'organization' or 'person' - the type of entity that is related

organization, organization_id If the `entity_type` is 'organization' and the entity is resolved, will be the organization that is related.

person, person_id If the `entity_type` is 'person' and the entity is resolved, will be the person that is related.

related_bills List of all related (but distinct) bills. An example might be a bill that was introduced in a prior session (and thus is similar but has different legislative history).

An array of entities with the following fields:

identifier The identifier of the related bill, such as SB 22.

legislative_session A link to the legislative session the related bill is from.

relation_type Description of the relation between the two bills, can be:

- companion - A companion bill introduced in the same session in opposite chamber.
- prior-session - Same bill as introduced in a prior session.
- replaced-by - A bill that was replaced by another bill.
- replaces - A bill that supercedes another bill.

related_bill, related_bill_id If the related bill exists in the data set, a link to the complete record for the bill. (**can be null if no such link has yet been made**)

sponsorships A list of all sponsoring people and organizations.

primary A boolean that is true if the sponsor can be considered 'primary.' The exact meaning of this is jurisdiction-dependent.

Note: this can often be derived from classification, but can often be highly jurisdiction dependent (among US states for example the terms author, sponsor, cosponsor, and coauthor can mean quite different things). Because of this, and the frequent need to indicate which author(s) should be displayed in a short list of sponsors when the full list might be half of the legislature or more, we've chosen to include this field.

classification The classification of the sponsor, such as 'cosponsor' or 'author'.

name The upstream-given name of this sponsor.

entity_type 'organization' or 'person' - the type of the sponsor.

organization, organization_id If the `entity_type` is 'organization' and the entity is resolved, will be the sponsoring organization.

person, person_id If the `entity_type` is 'person' and the entity is resolved, will be the sponsoring individual.

versions All versions of the bill.

note Note describing the version (e.g. 'Introduced', 'Engrossed', etc.)

date The date the version was published in YYYY-MM-DD format (partial dates are acceptable).

links Links to 'available forms' of the version. Each version can be available in multiple forms such as PDF and HTML. (For those familiar with DCAT this is the same as the `Distribution` class.) Has the following properties:

url URL of the link.

media_type The media type of the link.

documents All documents related to the bill with the exception of versions (which are part of the above `versions`).

note Note describing the document's relation to the bill (e.g. 'Fiscal Note', 'Committee Report', etc.)

date The date the document was published in YYYY-MM-DD format (partial dates are acceptable).

links Links to 'available forms' of the document. Each document can be available in multiple forms such as PDF and HTML. (For those familiar with DCAT this is the same as the `Distribution` class.) Has the following properties:

url URL of the link.

media_type The media type of the link.

created_at Time that this object was created at in the system, not to be confused with the date of introduction.

updated_at Time that this object was last updated in the system, not to be confused with the last action.

sources List of sources used in assembling this object. Has the following properties:

url URL of the resource.

note **optional** Description of what this source was used for.

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Further Reading

Many ideas here were based on the work in Open States.

Copyright

OCDEP 7: Votes

Created 2014-06-18

Author James Turk

Status Accepted

Overview

Definition of the Open Civic Data vote types.

(Based in part on Popolo types for the Open Civic Data Vote and VoteEvent).

Rationale

Determining how legislators have voted on specific proposals is a vital component to creating legislative accountability. The `VoteEvent` type and its related subtypes exist to enable the recording of votes (meaning the outcome of a proposal as well as the individual positions of legislators).

An attempt has been made to reconcile our vote types (originally based in Open States work) with the schemas put forward by the Popolo specification.

Implementation

Because at the time of writing (June 2014) the Popolo vote specification was still somewhat in a state of flux we've included pieces of the specification here for consistency.

VoteEvent

Because the term 'vote' can mean multiple things, Popolo makes use of the term `VoteEvent` to describe the actual event of a legislative vote taking place and the term `Vote` for a legislator's vote in a given `VoteEvent`. We generally use that terminology here, with the exception that the recommended API endpoint and ID scheme for `VoteEvent` objects will use `vote` and `ocd-vote/{{uuid}}` for VoteEvent.

Each `VoteEvent` has the following fields:

id Open Civic Data-style id, in the format `ocd-vote/{{uuid}}`.

identifier **optional** Upstream identifier if one exists, such as 'Roll Call #2372'.

motion The motion being voted upon. Represented as an object with `text` and `classification` elements.

> `text` is a required string describing the motion.
>
> `classification` is a list of classifiers (e.g. 'bill-passage') and can also be an empty list.
>
> In Popolo there is a Motion class that has additional properties, only text and classification are supported at this time.

start_date Time at which the vote begins. In flexible YYYY-MM-DD format.

end_date **optional** Time at which the vote ends, should such a time exist. In flexible YYYY-MM-DD format.

result The outcome of the vote, can be 'pass' or 'fail'.

organization, organization_id The `Organization` in which the `VoteEvent` took place.

legislative_session, legislative_session_id Reference to the `Jurisdiction.legislative_session` the `VoteEvent` takes place in.

bill, bill_id **optional** Reference to the `Bill` object the `VoteEvent` is related to (if the vote in question is on a bill).

votes A list of objects representing individual legislator's votes. (Popolo refers to these as Vote).

> This list may not be present if the individual voters are not known, for example in the case of a voice vote.
>
> Each element has the following properties:
>
> **voter_name** The name of the voter as it appears on the primary source. (Useful for when it cannot yet be resolved to a specific individual.)
>
> **voter, voter_id** A reference to the `Person` responsible for the vote. (may be null if the `voter_name` has not been resolved to an individual.)
>
> **option** The opinion of the `voter` casting this `Vote`. See *Voting Options*.

note A freeform text field describing any additional information about the vote. An example might be an excuse if the option is 'excused'.

counts A list of count objects (what Popolo calls Count).

These represent the total number of individuals voting a particular way. Each has two properties:

option The voting option being tallied. See *Voting Options*.

value The total number of individuals voting this way. (Should match the sum of the `votes` with the same `option` in almost every case, assuming they are present.)

created_at Time that this object was created at in the system, not to be confused with the start_date.

updated_at Time that this object was last updated in the system.

sources List of sources used in assembling this object. Has the following properties:

url URL of the resource.

note **optional** Description of what this source was used for.

extras Common to all Open Civic Data types, the value is a key-value store suitable for storing arbitrary information not covered elsewhere.

Voting Options

The possible values of `counts.option` and `votes.option` are as follows:

- `yes`
- `no`
- `abstain`
- `absent`
- `not voting`
- `paired`

Additional values will be added in the future as needed.

Differences from Popolo

- `VoteEvent` has an `extra` attribute which is not defined in Popolo.
- `Vote` objects (within `VoteEvent.votes`) have a `voter_name` property that is used for when a voter cannot be linked to a known `Person`. Additionally, `voter` can only link to a `Person`.
- `Vote` objects also do not currently support the `group`, `role`, `weight`, or `pair` properties. There is a chance these will be adopted in the future after their use is necessary.
- `Vote` objects have a `note` property that is not present in Popolo's specification.
- `Motion` objects from Popolo are represented simply as `motion.text`, this is future-proof should we ever choose to adopt additional `Motion` properties.

Copyright

OCDEP 101: Standardize Usage of Dates & Times

Created 2017-05-26

Author James Turk

Status Accepted

Overview

There are currently three ways we handle dates & times throughout Open Civic Data. This proposal aims to evaluate them and make several changes that will increase consistency & serve as guidance for future decisions.

Rationale

The current situation:

"Fuzzy Date Field" This is implemented as a char field allowing up to 10 characters. Dates are expected to conform to the YYYY[-MM[-DD]] format.

This comes from Popolo, and allows dates to be specified with varying degrees of accuracy depending on what is known. (e.g. sometimes we only know a person's birth year, or the month something came into being)

The field is used in the following places:

- BillAbstract.date
- BillAction.date
- BillDocument.date
- BillVersion.date
- EventDocument.date
- EventMedia.date (via EventMediaBase)
- EventAgendaMedia.date (via EventMediaBase)
- LegislativeSession.{start_date,end_date}
- Membership.{start_date,end_date}
- Organization.{founding_date,dissolution_date}
- OrganizationName.{start_date,end_date} (via OtherNameBase)
- Person.{birth_date,death_date}
- PersonName.{start_date,end_date} (via OtherNameBase)
- Post.{start_date,end_date}

"Fuzzy Datetime Field" For VoteEvents sometimes the time is important too, so we extended the above field to 19 characters, allowing an additional inclusion of time in HH:MM:SS.

This field is used only in

- VoteEvent.{start_date,end_date}

Finally, we sometimes use native DateTime fields.

Notably this is used for every model's created_at/updated_at timestamp.

It is also used for

- Event.{start_time,end_time}

This has the advantage of being timezone-aware.

Issues with current approach

For the most part this is OK, and we're fairly consistent. Most uses of fuzzy date align with the goals, but in a few cases it seems like we've made some mistakes:

1. VoteEvent and Event have very similar start/end times but use different and incompatible representations. VoteEvent's special case of fuzzy time can have a time but lacks a timezone, while Event's fields are named start_time/end_time and use a DateTime object, the only place one is used (also requiring more precision than we're guaranteed to have).

And two other issues:

2. The extended format is almost ISO8601 datetime, but uses a space instead of a 'T' as the separator of date & time.
3. We need the ability to set times on BillAction.date, just like VoteEvent. We are frequently forced to truncate times.

Implementation

We'd make the following changes:

1. To address #1 and #2: add timezone to "fuzzy datetime" and bring the full format in line w/ ISO8601, changing the format from:

```
YYYY[-MM][-DD][ HH:MM:SS]
```

to

```
^
[0-9]{4}
(
  (-[0-9]{2}){0,2} |
  (-[0-9]{2}){2} T [0-9]{2}(:[0-9]{2}){0,2} (Z | [+-] [0-9]{2}(:[0-9]{2})?)
)?
$
```

Also considered:

- Convert it to a full datetime, but this would require a time on every vote. We might not have one.
- Define that time is always stored in UTC, but this would be more error prone than being explicit.

2. To further address #1, rename Event.start_time,end_time to start_date,end_date to match Event and have it adopt the fuzzy datetime. This was chosen instead of renaming VoteEvent's fields to remain consistent w/ Popolo & other standards. This also makes the separate timezone field on event redundant and confusing, so it would be removed.

Also considered:

- Leaving this be, but I think we should take this opportunity to fix as many time related issues as we can.

3. To address #3, extend BillAction.date to allow "fuzzy datetimes".

Also considered:

- It could also become a full datetime (see #1), but would mostly have to fake the time.
- Naming the field 'time' was initially recommended, but since we aren't changing other fields that has been withdrawn.

Copyright

For proposals under consideration see *drafts*.