## Contents:

<table>
<thead>
<tr>
<th>1 Open States v3 API</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 API Basics</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Methods</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Concepts</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Further Details</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Open States v2 GraphQL API</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Basics</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Full-fledged Example</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Further Details</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Contributing to Open States</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Getting Started</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Overview &amp; Roadmap</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Communication</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Contributing to Scrapers</td>
<td>41</td>
</tr>
<tr>
<td>3.5 State-Specific Scraper Info</td>
<td>44</td>
</tr>
<tr>
<td>3.6 Testing Scrapers</td>
<td>45</td>
</tr>
<tr>
<td>3.7 Working On openstates.org</td>
<td>47</td>
</tr>
<tr>
<td>3.8 Contributing People Data</td>
<td>49</td>
</tr>
<tr>
<td>3.9 Text Extraction</td>
<td>52</td>
</tr>
<tr>
<td>3.10 Contributing Documentation</td>
<td>56</td>
</tr>
<tr>
<td>3.11 Code of Conduct</td>
<td>56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Understanding the Data</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Main Concepts</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Categorization</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Session Naming</td>
<td>61</td>
</tr>
</tbody>
</table>
Documentation for Open States, its API, and how to contribute.
Open States provides a JSON API that can be used to programatically access state legislative information.

1.1 API Basics

The root URL for the API is https://v3.openstates.org/.

API keys are required. You can register for an API key and once activated, you’ll pass your API key via the X-API-KEY header or ?apikey query parameter.

Auto-generated interactive documentation is available at either:

- https://v3.openstates.org/docs/
- https://v3.openstates.org/redoc/ (whichever you prefer)

Issues should be filed at our issue tracker.

You can also check out our introductory blog post for more details.

1.2 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Interactive Docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>/jurisdictions</td>
<td>Get list of available jurisdictions.</td>
<td>[1]</td>
</tr>
<tr>
<td>/jurisdictions/{jurisdiction_id}</td>
<td>Get detailed metadata for a particular jurisdiction.</td>
<td>[2]</td>
</tr>
<tr>
<td>/people</td>
<td>List or search people (legislators, governors, etc.)</td>
<td>[3]</td>
</tr>
<tr>
<td>/people.geo</td>
<td>Get legislators for a given location.</td>
<td>[4]</td>
</tr>
<tr>
<td>/bills</td>
<td>Search bills by various criteria.</td>
<td>[5]</td>
</tr>
<tr>
<td>/bills/ocd-bill/{uuid}</td>
<td>Get bill by internal ID.</td>
<td>[6]</td>
</tr>
<tr>
<td>/bills/{jurisdiction}/[session]/[id]</td>
<td>Get bill by jurisdiction, session, and ID.</td>
<td>[7]</td>
</tr>
</tbody>
</table>
1.3 Concepts

Jurisdiction  The fundamental unit by which data is partitioned is the ‘jurisdiction.’ If you are just interested in states you can consider the words synonymous for the most part. Jurisdictions include states, DC & Puerto Rico, and municipal governments for which we have limited support.

Person  A legislator, governor, mayor, etc.

   Each person possibly has a number of roles, at most one of which is considered ‘current.’

Bill  A proposed piece of legislation, encompasses bills, resolutions, constitutional amendments, etc.

   A given bill may have any number of votes, sponsorships, actions, etc.

1.4 Further Details

1.4.1 API v3 Changelog

2021.04.19

   • experimental support for Jurisdiction.latest_runs include parameter

2021.03.24

   • Jurisdiction classification can now be ‘country’
   • experimental US federal support

2021.02.23

   • people district= parameter added

2020.12.21

   • people name= parameter is now fuzzy-searched, matching API v1 and v2

2020.10.30

   • move email to a top-level field on Person responses
   • add consistent post ordering

2020.10.21

   • fix datetime handling for updated_since and created_since filters
   • add list of divisions when including Jurisdiction.organizations
2020.10.13

- add updated_asc sort option
- add rate limiting
- bugfix for New York jurisdiction lookup (openstates/issues#136)

2020.09.28

- set permissive CORS settings
- bills endpoint updates:
  - added created_since filter, thanks to Donald Wasserman!
  - added sponsor and sponsor_classification filters
  - added sort parameter
  - added useful error message when searching /bills by session without jurisdiction
- restored missing Bill.from_organization field
- introduced new fields: Person.openstates_url, Bill.openstates_url

2020.09.14

- removed some unused fields from responses
- removed deprecated government classification from Jurisdiction

2020.09.10

- added Jurisdiction.legislative_sessions
- corrected initial pagination limits for release

2020.09.09

- Initial beta release.
Open States v2 GraphQL API

- API keys are required. You can register for an API key and once activated, you’ll pass your API key via the X-API-KEY header.
- You can also check out our introductory blog post for more details.

2.1 Basics

This is a GraphQL API, and some of the concepts may seem unfamiliar at first.

There is in essence, only one endpoint: https://openstates.org/graphql.

This endpoint, when accessed in a browser, will provide an interface that allows you to experiment with queries in the browser, it features autocomplete and a way to browse the full graph (click the ‘Docs’ link in the upper right corner).

A GraphQL query mirrors the structure of the data that you’d like to obtain. For example, to obtain a list of legislators you’d pass something like:

```json
{
  people {
    edges {
      node {
        name
      }
    }
  }
}
```

**Note:** If you are using the API programatically it is recommended you send the data as part of the POST body, e.g.:

```bash
curl -X POST https://openstates.org/graphql -d "\"query=\{people\{edges\{node\{name\}\}\}\}\""
```
Of course, if you try this you’ll see it doesn’t work since there are some basic limits on how much data you can request at once. We paginate with the `first`, `last`, `before` and `after` parameters to a root node. So let’s try that again:

```json
{
  people(first: 3) {
    edges {
      node {
        name
      }
    }
  }
}
```

And you’d get back JSON like:

```json
{
  "data": {
    "people": {
      "edges": [
        {
          "node": {
            "name": "Lydia Brasch"
          }
        },
        {
          "node": {
            "name": "Matt Williams"
          }
        },
        {
          "node": {
            "name": "Merv Riepe"
          }
        }
      ]
    }
  }
}
```

Ah, much better. Nodes also can take other parameters to filter the returned content. Let’s try the “name” filter which restricts our search to people named Lydia:

```json
{
  people(first: 3, name: "Lydia") {
    edges {
      node {
        name
      }
    }
  }
}
```

Results in:

```json
{
  "data": {
    "people": {
      "edges": [
      ]
    }
  }
}
```
(continues on next page)
It is also possible to request data from multiple root nodes at once, for example:

```json
{
  people(first: 1) {
    edges {
      node { name }
    }
  }
  bills(first: 1) {
    edges {
      node { title }
    }
  }
}
```

Would give back something like:

```json
{
  "data": {
    "people": {
      "edges": [
        {
          "node": {
            "name": "Lydia Brasch"
          }
        }
      ]
    },
    "bills": {
      "edges": [
        {
          "node": {
            "title": "Criminal Law - Animal Abuse Emergency Compensation Fund Establishment"
          }
        }
      ]
    }
  }
}
```

(continues on next page)
You may notice something here, that you get back just the data you need. This is extremely powerful, and lets you do the equivalent of many traditional API calls in a single query.

## 2.2 Full-fledged Example

Let’s take a look at a more useful example:

```graphql
{  bill(jurisdiction: "New York", session: "2017-2018", identifier: "S 5772") {    title    actions {      description      date    }    votes {      edges {        node {          counts {            value            option          }          votes {            voterName            voter {              id              contactDetails {                value                note                type              }            }            option          }        }      }    }    sources {      url    }    createdAt    updatedAt  }
}
```

There’s a lot going on there, let’s break it down:

```graphql
```

We’re hitting the `bill` root node, which takes 3 parameters. This should get us to a single bill from New York.
This is going to give us the title, just like we saw before.

Here we're going into a child node, in this case all of the actions taken on the bill. For each action we're requesting a the date & description.

Here too we’re going into a child node, but note that this time we use that “edges” and “node” pattern that we see on root level nodes. Certain child nodes in the API have the ability to be paginated or further limited, and votes happen to be one of them. In this case however we’re not making use of that so we’ll just ignore this.

(A full discussion of this pattern is out of scope but check out the Relay pagination specification for more detail for more.)

Here we grab a few more fields, including child nodes of each vote on our Bill.

First, we get a list of counts (essentially pairs of outcomes + numbers e.g. (yes, 31), (no, 5))

We also get individual legislator votes by name, and we traverse into another object to get the Open States ID and contact details for the voter. (Don’t sweat the exact data model here, there will be more on the structure once we get to the actual graph documentation.)

And back up at the top level, we grab a few more pieces of information about the Bill.

And now you’ve seen a glimpse of the power of this API. We were able to get back the exact fields we wanted on a bill, contact information on the legislators that have voted on the bill, and more.
Our result looks like this:

```json
{
  "data": {
    "bill": {
      "title": "Relates to bureaus of administrative adjudication",
      "actions": [
        {
          "description": "REFERRED TO LOCAL GOVERNMENT",
          "date": "2017-04-28"
        },
        {
          "description": "COMMITTEE DISCHARGED AND COMMITTED TO RULES",
          "date": "2017-06-19"
        },
        {
          "description": "ORDERED TO THIRD READING CAL.1896",
          "date": "2017-06-19"
        },
        {
          "description": "RECOMMITTED TO RULES",
          "date": "2017-06-21"
        }
      ],
      "votes": {
        "edges": [
          {
            "node": {
              "counts": [
                {
                  "value": 25,
                  "option": "yes"
                },
                {
                  "value": 0,
                  "option": "no"
                },
                {
                  "value": 0,
                  "option": "other"
                }
              ],
              "votes": [
                {
                  "voterName": "John J. Bonacic",
                  "votes": {
                    "id": "ocd-person/da013cd5-dc67-4e65-a310-73aa32ad1f7c"
                    "contactDetails": [
                      {
                        "value": "bonacic@nysenate.gov",
                        "note": "Capitol Office",
                        "type": "email"
                      },
                      {
                        "value": "Room 503\nAlbany, NY 12247",
                        "note": "District Office",
                        "type": "address"
                      }
                    ]
                  }
                }
              ]
            }
          }
        ]
      }
    }
  }
}
```
{  
  "value": "518-455-3181",
  "note": "District Office",
  "type": "voice"
},
  
  ...etc...
  
},
"option": "yes"
},
{
  "voterName": "Neil D. Breslin",
  "voter": {
    "id": "ocd-person/4b710aee-1b99-42e0-90e2-d41338e8c5df"
    "contactDetails": [ ...etc... ]
  },
"option": "yes"
},
{
  "voterName": "David Carlucci",
  "voter": {
    "id": "ocd-person/1b0feab9-02a7-4bcc-b089-3ab23286da68"
    "contactDetails": [ ...etc... ]
  },
"option": "yes"
},
  
  ...etc...
  
},
"sources": [
  
  
  "url": "http://www.nysenate.gov/legislation/bills/2017/S5772"
  
  "url": "http://assembly.state.ny.us/leg/?default_fld=&bn=S5772&Summary=Y&Actions=Y&Text=Y"
  
  
  "createdAt": "2017-07-15 05:08:15.848526+00:00",
  "updatedAt": "2017-07-15 05:08:15.848541+00:00"
]}
}
2.3 Further Details

2.3.1 Root Nodes

As seen in the introduction, when constructing a query you will start your query at one (or more) root nodes. The following root nodes are available:

jurisdictions

Get a list of all jurisdictions.
This will return a list of JurisdictionNode objects, one for each state (plus Puerto Rico and DC).
Pagination: This endpoint accepts the usual Pagination parameters, but pagination is not required.

people

Get a list of all people matching certain criteria.
This will return a list of PersonNode objects, one for each person matching your query.
Pagination: This endpoint accepts the usual Pagination parameters, and you must limit your results to no more than 100 using either the “first” or “last” parameter.

Parameters:

name Limit response to people who’s name contains the provided string.
Includes partial matches & case-insensitive matches.

memberOf Limit response to people that have a currently active membership record for an organization. The value
passed to memberOf can be an ocd-organization ID or a name (e.g. ‘Republican’ or ‘Nebraska Legislature’).

everMemberOf Limit response to people that have any recorded membership record for an organization. Operates
as a superset of memberOf.

Specifying memberOf and everMemberOf in the same query is invalid.

district When specifying either memberOf or everMemberOf, limits to people who’s membership represented
the district with a given label. (e.g. memberOf: “Nebraska Legislature”, district: “7”)

Specifying district without memberOf or everMemberOf is invalid.

latitude and longitude Limit to people that are currently representing the district(s) containing the point spec-
fied by the provided coordinates.
Must be specified together.

bills

Get a list of all bills matching certain criteria.
This will return a list of BillNode objects, one for each person matching your query.
Pagination: This endpoint accepts the usual Pagination parameters, and you must limit your results to no more than 100 using either the “first” or “last” parameter.
Parameters:

**jurisdiction** Limit to bills associated with given jurisdiction, parameter can either be a human-readable jurisdiction name or an ocd-jurisdiction ID.

**chamber** Limit to bills originating in a given chamber. (e.g. upper, lower, legislature)

**session** Limit to bills originating in a given legislative session. This parameter should be the desired session’s identifier. (See LegislativeSessionNode).

**classification** Limit to bills with a given classification (e.g. “bill” or “resolution”)

**subject** Limit to bills with a given subject (e.g. “Agriculture”)

**searchQuery** Limit to bills that contain a given term. (Experimental until 2020!)

**updatedSince** Limit to bills that have had data updated since a given time (UTC).

  Time should be in the format YYYY-MM-DD[THH:MM:SS].

**actionsSince** Limit to bills that have had actions since a given time (UTC).

  Time should be in the format YYYY-MM-DD.

**jurisdiction**

Look up a single jurisdiction by name or ID.

This will return a single JurisdictionNode object with the provided name or ID parameter.

Parameters:

**name** The human-readable name of the jurisdiction, such as ‘New Hampshire’.

**id** The ocd-jurisdiction ID of the desired jurisdiction, such as ‘ocd-jurisdiction/country:us/state:nh’.

You are required to provide one of the two available parameters.

**person**

Look up a single person by ocd-person ID.

This will return a single PersonNode by ID.

Parameters:

**id** ocd-person ID for the desired individual.

**organization**

Look up a single organization by ocd-organization ID.

This will return a single OrganizationNode by ID.
Parameters:

**id**  ocd-organization ID for the desired individual.

**bill**

Look up a single bill by ID, URL, or (jurisdiction, session, identifier) combo.
This will return a single `BillNode` object with the specified bill.

Parameters:

**id**  The ocd-bill ID of the desired bill, such as ‘ocd-jurisdiction/country:us/state:nh’.

**openstatesUrl**  The URL of the desired bill, such as ‘https://openstates.org/nc/bills/2019/HB760/’.

**jurisdiction, session, identifier**  Must be specified together to fully identify a bill.

As is true elsewhere, jurisdiction may be specified by name (New Hampshire) or ocd-jurisdiction ID (ocd-jurisdiction/country:us/state:nh).

Session is specified by legislative session identifier (e.g. 2018 or 49).

Identifier is the exact identifier of the desired bill, such as “HB 327”.

You are required to provide one either **id** or the other parameters to fully specify a bill. Use **bills** if you are looking for something more broad.

### 2.3.2 Data Types

Starting at the base nodes, data in the API is represented as interconnected nodes of various types. This page provides an overview of the nodes.

Another good way to get acquainted with the layout is to use the [GraphQL browser](#) (click Docs in the upper right corner).

#### Jurisdictions & Sessions

**JurisdictionNode**

A Jurisdiction is the Open Civic Data term for the top level divisions of the US. Open States is comprised of 52 jurisdictions, one for each state, and two more for D.C. and Puerto Rico.

Each JurisdictionNode has the following attributes & nodes available:

- **id** - ocd-jurisdiction identifier, these are permanent identifiers assigned to each Jurisdiction
- **name** - human-readable name for the jurisdiction (e.g. Kansas)
- **url** - URL of official website for jurisdiction
- **featureFlags** - reserved for future use
- **legislativeSessions** - Paginated list (see [Pagination](#)) of LegislativeSessionNode belonging to this jurisdiction’s legislature.
- **organizations** - Paginated list of OrganizationNode belonging to this jurisdiction.
– it is also possible to filter the list of children using the classification parameter

- lastScrapedAt - Time when last scrape finished.

See also: Open Civic Data Jurisdiction reference

**LegislativeSessionNode**

A legislative session is a convening of the legislature, either a primary or special session.

Each LegislativeSessionNode has the following attributes and nodes available:

- jurisdiction - JurisdictionNode which this session belongs to.
- identifier - short identifier by which this session is referred to (e.g. 2017s1 or 121)
- name - formal name of session (e.g. “2017 Special Session #1” or “121st Session”
- classification - “primary” or “special”
- startDate - start date of session if known
- endDate - end date of session if known

**DivisionNode**

Divisions represent particular geopolitical boundaries. Divisions exist for states as well as their component districts and are tied closely to political geographies.

- id - Open Civic Data Division ID
- name - human-readable name for the division
- redirect - link to another DivisionNode, only present if division has been replaced
- country - country code (will be “us”) for all Open States divisions
- createdAt - date at which this object was created in our system
- updatedAt - date at which this object was last updated in our system
- extras - JSON string with optional additional information about the object

**People & Organizations**

**PersonNode**

People, typically legislators and their associated metadata.

Note that most fields are optional beyond name as often we don’t have a reliable given/family name or birthDate for instance.

- id - Open Civic Data Person ID
- name - primary name for the person
- sortName - alternate name to sort by (if known)
- familyName - hereditary name, essentially a “last name” (if known)
- givenName - essentially a “first name” (if known)
• **image** - full URL to official image of legislator

• **birthDate** - see *Fuzzy Date Format*

• **deathDate** - see *Fuzzy Date Format*

• **identifiers** - list of other known identifiers, *IdentifierNode*

• **otherNames** - list of other known names, *NameNode*

• **links** - official URLs relating to this person, *LinkNode*

• **contactDetails** - ways to contact this person (via email, phone, etc.), *contactdetailnode*

• **currentMemberships** - currently active memberships *MembershipNode*
  
  -- can be filtered with the **classification** parameter to only get memberships to certain types of *OrganizationNode*

• **oldMemberships** - inactive memberships *MembershipNode*
  
  -- can be filtered with the **classification** parameter to only get memberships to certain types of *OrganizationNode*

• **sources** - URLs which were used in compiling Open States’ information on this subject, *LinkNode*

• **createdAt** - date at which this object was created in our system

• **updatedAt** - date at which this object was last updated in our system

• **extras** - JSON string with optional additional information about the object

See also:

• Popolo’s person

• Open Civic Data OCDEP 5

---

**OrganizationNode**

Organizations that comprise the state legislatures and their associated metadata.

A typical bicameral legislature is comprised of a top-level organization (classification=legislature), two chambers (classification=upper & lower), and any number of committees (classification=committee).

Each Organization is comprised of the following attributes and nodes:

• **id** - Open Civic Data Organization ID

• **name** - primary name for the person

• **image** - full URL to official image for organization

• **classification** - the type of organization as described above

• **foundingDate** - see *Fuzzy Date Format*

• **dissolutionDate** - see *Fuzzy Date Format*

• **parent** - parent OrganizationNode if one exists

• **children** - paginated list of child OrganizationNode objects
  
  -- it is also possible to filter the list of children using the **classification** parameter

• **currentMemberships** - list of all current members of this Organization

• **identifiers** - list of other known identifiers for this organization, *IdentifierNode*
- `otherNames` - list of other known names for this organization, NameNode
- `links` - official URLs relating to this person, LinkNode
- `sources` - URLs which were used in compiling Open States’ information on this subject, LinkNode
- `createdAt` - date at which this object was created in our system
- `updatedAt` - date at which this object was last updated in our system
- `extras` - JSON string with optional additional information about the object

See also:
- Popolo’s organization
- Open Civic Data OCDEP 5

### MembershipNode

A MembershipNode represents a connection between a personnode and an organizationnode. A membership may optionally also reference a particular postnode, such as a particular seat within a given chamber.

Each membership has the following attributes and nodes:

- `id` - Open Civic Data Membership ID
- `personName` - the raw name of the person that the membership describes (see Name Matching)
- `person` - personnode
- `organization` - organizationnode
- `post` - postnode
- `label` - label assigned to this membership
- `role` - role fulfilled by this membership
- `startDate` - start date of membership if known
- `endDate` - end date of membership if known
- `createdAt` - date at which this object was created in our system
- `updatedAt` - date at which this object was last updated in our system
- `extras` - JSON string with optional additional information about the object

See also:
- Popolo’s membership
- Open Civic Data OCDEP 5

### PostNode

A PostNode represents a given position within an organization. The most common example would be a seat such as Maryland’s 4th House Seat.

It is worth noting that some seats can have multiple active memberships at once, as noted in maximumMemberships.

Each post has the following attributes and nodes:

- `id` - Open Civic Data Post ID
- **label** - label assigned to this post (e.g. 3)
- **role** - role fulfilled by this membership (e.g. 'member')
- **division** - related `divisionnode` if this role has a relevant division
- **startDate** - start date of membership if known
- **endDate** - end date of membership if known
- **maximumMemberships** - typically 1, but set higher in the case of multi-member districts
- **createdAt** - date at which this object was created in our system
- **updatedAt** - date at which this object was last updated in our system
- **extras** - JSON string with optional additional information about the object

See also:

- Popolo’s post
- Open Civic Data OCDEP 5

### Bills & Votes

#### BillNode

A BillNode represents any legislative instrument such as a bill or resolution.

Each node has the following attributes and nodes available:

- **id** - Internal ocd-bill identifier for this bill.
- **legislativeSession** - link to `LegislativeSessionNode` this bill is from
- **identifier** - primary identifier for this bill (e.g. HB 264)
- **title** - primary title for this bill
- **fromOrganization** - organization (typically upper or lower chamber) primarily associated with this bill
- **classification** - list of one or more bill types such as “bill” or “resolution”
- **subject** - list of zero or more subjects assigned by the state
- **abstracts** - list of abstracts provided by the state, `BillAbstractNode`
- **otherTitles** - list of other titles provided by the state, `BillTitleNode`
- **otherIdentifiers** - list of other identifiers provided by the state, `BillIdentifierNode`
- **actions** - list of actions (such as introduction, amendment, passage, etc.) that have been taken on the bill, `BillActionNode`
- **sponsorships** - list of bill sponsors, `BillSponsorshipNode`
- **relatedBills** - list of related bills as provided by the state, `RelatedBillNode`
- **versions** - list of bill versions as provided by the state, `BillDocumentNode`
- **documents** - list of related documents (e.g. legal analysis, fiscal notes, etc.) as provided by the state, `BillDocumentNode`
- **votes** - paginated list of `VoteEventNode` related to the bill
- **sources** - URLs which were used in compiling Open States’ information on this subject, `linknode`
• **openstatesUrl** - URL to bill page on OpenStates.org
• **createdAt** - date at which this object was created in our system
• **updatedAt** - date at which this object was last updated in our system
• **extras** - JSON string with optional additional information about the object

**BillAbstractNode**

Represents an official abstract for a bill, each BillAbstractNode has the following attributes:

• **abstract** - the abstract itself
• **note** - optional note about origin/purpose of abstract
• **date** - optional date associated with abstract

**BillTitleNode**

Represents an alternate title for a bill, each BillTitleNode has the following attributes:

• **title** - the alternate title
• **note** - optional note about origin/purpose of this title

**BillIdentifierNode**

Represents an alternate identifier for a bill, each BillIdentifierNode has the following attributes:

• **identifier** - the alternate identifier
• **scheme** - a name for the identifier scheme
• **note** - optional note about origin/purpose of this identifier

**BillActionNode**

Represents an action taken on a bill, each BillActionNode has the following attributes and nodes:

• **organization** - *OrganizationNode* where this action originated, will typically be either upper or lower chamber, or perhaps legislature as a whole.
• **description** - text describing the action as provided by the jurisdiction.
• **date** - date action took place (see *Fuzzy Date Format*)
• **classification** - list of zero or more normalized action types (see *Action Types*)
• **order** - integer by which actions can be sorted, not intended for display purposes
• **extras** - JSON string providing extra information about this action
• **vote** - if there is a known associated vote, pointer to the relevant *VoteEventNode*
• **relatedEntities** - a list of *RelatedEntityNode* with known entities referenced in this action

2.3. **Further Details**
RelatedEntityNode

Represents an entity that is related to a BillActionNode.

- **name** - raw (source-provided) name of entity
- **entityType** - either organization or person
  - **organization** - if entityType is ‘organization’, the resolved OrganizationNode
  - **person** - if entityType is ‘person’, the resolved PersonNode

See Name Matching for details on how name relates to organization and person.

BillSponsorshipNode

Represents a sponsor of a bill.

- **name** - raw (source-provided) name of sponsoring person or organization
- **entityType** - either organization or person
  - **organization** - if entityType is ‘organization’, the resolved OrganizationNode
  - **person** - if entityType is ‘person’, the resolved PersonNode
- **primary** - boolean, true if sponsorship is considered by the jurisdiction to be “primary” (note: in many states multiple primary sponsors may exist)
- **classification** - jurisdiction-provided type of sponsorship, such as “author” or “cosponsor”. These meanings typically vary across states, which is why we provide primary as a sort of indicator of the degree of sponsorship indicated.

See Name Matching for details on how name relates to organization and person.

RelatedBillNode

Represents relationships between bills.

- **identifier** - identifier of related bill (e.g. SB 401)
- **legislativeSession** - identifier of related session (in same jurisdiction)
- **relationType** - type of relationship such as “companion”, “prior-session”, “replaced-by”, or “replaces”
- **relatedBill** - if the related bill is found to exist in our data, link to the BillNode

BillDocumentNode

Representation of documents and versions on bills. A given document can have multiple links representing different manifestations (e.g. HTML, PDF, DOC) of the same content.

- **note** - note describing the purpose of the document or version (e.g. Final Printing)
- **date** - optional date associated with the document
- **links** - list of one or more MimetypeLinkNode with actual URLs to bills.
MimetypeLinkNode

Represents a single manifestation of a particular document.

- **mediaType** - media type (aka MIME type) such as application/pdf or text/html
- **url** - URL to official copy of the bill
- **text** - text describing this particular manifestation (e.g. PDF)

VoteEventNode

Represents a vote taken on a bill.

- **id** - Internal ocd-vote identifier for this bill.
- **identifier** - Identifier used by jurisdiction to uniquely identify the vote.
- **motionText** - Text of the motion being voted upon, such as “motion to pass the bill as amended.”
- **motionClassification** - List with zero or more classifications for this motion, such as “passage” or “veto-override”
- **startDate** - Date on which the vote took place. (see Fuzzy Date Format)
- **result** - Outcome of the vote, ‘pass’ or ‘fail’.
- **organization** - Related OrganizationNode where vote took place.
- **billAction** - Optional linked BillActionNode.
- **votes** - List of PersonVoteNode for each individual’s recorded vote. (May not be present depending on jurisdiction.)
- **counts** - List of VoteCountNode with sums of each outcome (e.g. yea/nay/abstain).
- **sources** - URLs which were used in compiling Open States’ information on this subject, LinkNode
- **createdAt** - date at which this object was created in our system
- **updatedAt** - date at which this object was last updated in our system
- **extras** - JSON string with optional additional information about the object

See also: Open Civic Data vote format.

PersonVoteNode

Represents an individual person’s vote (e.g. yea or nay) on a given bill.

- **option** - Option chosen by this individual. (yea, nay, abstain, other, etc.)
- **voterName** - Raw name of voter as provided by jurisdiction.
- **voter** - Resolved PersonNode representing voter. (See Name Matching)
- **note** - Note attached to this vote, sometimes used for explaining an “other” vote.
VoteCountNode

Represents the sum of votes for a given option.

- **option** - Option in question. (yea, nay, abstain, other, etc.)
- **value** - Number of individuals voting this way.

Other Nodes

IdentifierNode

Represents an alternate identifier, each with the following attributes:

- **identifier** - the alternate identifier
- **scheme** - a name for the identifier scheme

NameNode

Represents an alternate name, each with the following attributes:

- **name** - the alternate name
- **note** - note about usage/origin of this alternate name
- **startDate** - date at which this name began being valid (blank if unknown)
- **endDate** - date at which this name stopped being valid (blank if unknown or still active)

LinkNode

Represents a single link associated with a person or used as a source.

- **url** - URL
- **text** - text describing the use of this particular URL

ContactDetailNode

Used to represent a contact method for a given person.

- **type** - type of contact detail (e.g. voice, email, address, etc.)
- **value** - actual phone number, email address, etc.
- **note** - used to group contact data by location (e.g. Home Address, Office Address)
- **label** - human-readable label for this contact detail

2.3.3 Other Notes

There are a few other things to be aware of while using the API:
Explore the Graph

GraphQL is still quite new, so we figured it might be good to provide some helpful tips on how to think about the data and how you’ll use the API.

First, it is probably well worth your time to play around in GraphiQL to explore the API and data. It was heavily used when developing the API and writing tests, and is a very powerful tool, particularly when you make use of the self-documenting nature of the graph.

When you’re thinking about how to query don’t necessarily try to replicate your old API calls exactly. For example, perhaps you were grabbing all bills that met a given criteria and then grabbing all sponsors contact details. This can now be done in one call by traversing from the `bills` root node into the `BillSponsorshipNode` and then up to the `PersonNode` and finally to the `ContactDetailNode`. This may sound complex at first, but once you get the hang of it, it really does unlock a ton of power and will make your apps more powerful and efficient.

Pagination

In several places (such as the `bills` and `BillNode’s votes`) we mention that nodes are paginated.

What this means in practice is that instead of getting back the underlying node type, say `BillNode`, directly, you’ll get back `BillConnectionNode` or similar. (In practice there are connection node types for each paginated type, but all work the same way in our case.)

Arguments

Each paginated endpoint accepts any of four parameters:

- `first` - given an integer N, only return the first N nodes
- `last` - given an integer N, only return the last N nodes
- `after` - combined with `first`, will return first N nodes after a given “cursor”
- `before` - combined with `last`, will return last N nodes before a given “cursor”

So typically you’d paginate using `first`, obtaining a cursor, and then calling the API again with a combination of `first` and `after`.

The same process could be carried out with `last` and `before` to paginate in reverse.

Responses

Let’s take a look at everything that pagination makes available:

```json
{
  bills(first:20) {
    edges {
      node {
        title
      }
      cursor
    }
  }
  pageInfo {
    hasNextPage
    hasPreviousPage
    endCursor
  }
}
```

(continues on next page)
You’ll see that the connection node has three nodes: `edges`, `pageInfo`, and `totalCount`:

- **edges** - a list of objects that each have a `node` and `cursor` attribute:
  - `node` - the underlying node type, in our case `BillNode`
  - `cursor` - an opaque cursor for this particular item, it can be used with the `before` and `after` parameters each paginated node accepts as arguments.

- **pageInfo** - a list of helpful pieces of information about this page:
  - `hasNextPage` - boolean that is true if there is another page after this
  - `hasPreviousPage` - boolean that is true if there is a page before this
  - `endCursor` - last cursor in the set of edges, can be used with `after` to paginate forward
  - `startCursor` - first cursor in the set of edges, can be used with `before` to paginate backwards

- **totalCount** - total number of objects available from this connection

### In Practice

Let’s say you want to get all of the people matching a given criteria:

You’d start with a query for all people matching your criteria, ensuring to set the page size to no greater than the maximum:

```graphql
{
  people(memberOf: "Some Organization", first: 100) {
    edges {
      node {
        name
      }
    }
    pageInfo {
      hasNextPage
      endCursor
    }
  }
}
```

Let’s say we got back a list of 100 edges and our `pageInfo` object looked like:

```json
{
  "hasNextPage": true,
  "endCursor": "ZXJyYXlxb20uZWN0aW9uOjA="
}
```

So you’d make another call:
{  
  people(memberOf: "Some Organization", first: 100, after:"ZXJyYXlxb20uZWN0aW9uOjA=  
  " ) {  
    edges {  
      node {  
        name  
      }  
    }  
    pageInfo {  
      hasNextPage  
      endCursor  
    }  
  }  
}

And let’s say in this case you got back only 75 edges, and our pageInfo object looks like:

{  
  "hasNextPage": false,  
  "endCursor": "AXjYylxX2bu1wxa9uunnb="  
}

We’d stop iteration at this point, of course, if hasNextPage had been true, we’d continue on until it wasn’t.

**Renaming fields**

A really useful trick that is often overlooked is that you can rename fields when retrieving them, for example:

{  
  republicans: people(memberOf: "Republican", first: 5) {  
    edges {  
      node {  
        full_name: name  
      }  
    }  
  }  
}

Would give back:

{  
  "data": {  
    "republicans": {  
      "edges": [  
        {  
          "node": {  
            "full_name": "Michelle Udall"  
          }  
        },  
        {  
          "node": {  
            "full_name": "Kimberly Yee"  
          }  
        },  
        {  
          "node": {  
            "full_name": "John Doe"  
          }  
        },  
        {  
          "node": {  
            "full_name": "Jane Smith"  
          }  
        },  
        {  
          "node": {  
            "full_name": "Susan Johnson"  
          }  
        }  
      ]  
    }  
  }  
}

(continues on next page)
Note that we’re both renaming a top-level node here as well as a piece of data within the query. You can also use this to query the same root node twice (doing so without renaming isn’t allowed since it results in a name conflict).

For example:

```json
{
  republicans: people(memberOf: "Republican", first: 5) {
    edges {
      node {
        full_name: name
      }
    }
  }
  democrats: people(memberOf: "Democratic", first: 5) {
    edges {
      node {
        full_name: name
      }
    }
  }
}
```

**Fuzzy Date Format**

Unless otherwise noted (most notably `createdAt` and `updatedAt`) all date objects are “fuzzy”. Instead of being expressed as an exact date, it is possible a given date takes any of the following formats:

- YYYY
- YYYY-MM
- YYYY-MM-DD
- YYYY-MM-DD HH:MM:SS (if times are allowed)

Action/Vote times are all assumed to be in the state capitol’s time zone.

Times related to our updates such as `updatedAt` and `createdAt` are in UTC.
Name Matching

In several places such as bill sponsorships and votes you’ll notice that we have a raw string representing a person or organization as well as a place for a link to the appropriate `OrganizationNode` or `PersonNode`.

Because of the way we collect the data from states, we always collect the raw data and later make an attempt to (via a mix of automated matching and manual fixes) connect the reference with data we’ve already collected.

In many cases these linkages will not be provided, but with some upcoming new tools to help us improve this matching we’ll be able to dramatically improve the number of matched entities in the near future.

2.3.4 Examples

Get basic information for all legislatures

See in GraphiQL

```graphql
{
  jurisdictions {
    edges {
      node {
        name
        legislativeSessions {
          edges {
            node {
              name
            }
          }
        }
        legislature: organizations(classification: "legislature", first: 1) {
          edges {
            node {
              name
              classification
              children(first: 5) {
                edges {
                  node {
                    name
                    classification
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Get overview of a legislature’s structure

See in GraphiQL

2.3. Further Details
Search for bills that match a given condition

See in GraphiQL

```graphql
{
    edges {
      node {
        id
        identifier
        title
        createdAt
        legislativeSession {
          identifier
          jurisdiction {
            name
          }
        }
        actions {
          date
          description
          classification
        }
      }
    }
  }
}
```
Get all information on a particular bill

See in GraphiQL

```json
{  b1: bill(jurisdiction: "Hawaii", session: "2017 Regular Session", identifier: "HB-475") {  id  identifier  title  classification  updatedAt  createdAt  legislativeSession {  identifier  jurisdiction {  name  }  }  actions {  date  description  classification  }  documents {  date  note  links {  url  }  }  }
}
```
Get information about a specific legislator

See in GraphiQL
Get legislators for a given state/chamber

See in GraphiQL

`ocd-organization/ddf820b5-5246-46b3-a807-99b5914ad39f` is the id of the Alabama Senate chamber.

```graphql
{ people(memberOf:"ocd-organization/ddf820b5-5246-46b3-a807-99b5914ad39f", first:-100) { edges { node { name party: currentMemberships(classification:"party") { organization { name } } } links { url } sources { url } chamber: currentMemberships(classification:["upper", "lower"]) { post { label } organization { name classification parent { name } } } } } } (continues on next page)```
Search for legislators that represent a given area

See in GraphQL

```graphql
{
  people(latitude: 40.7460022, longitude: -73.9584642, first: 100) {
    edges {
      node {
        name
        chamber: currentMemberships(classification: ["upper", "lower"]){
          post {
            label
          }
          organization {
            name
            classification
            parent {
              name
            }
          }
        }
      }
    }
  }
}
```

### 2.3.5 Changelog

Changelog for Open States GraphQL API:

**v2.6 (March 2021)**

- added preliminary support for federal jurisdiction
- added Jurisdiction.classification node
- Jurisdictions are now filterable by classification (municipal, state, country)

**v2.5 (July 2020)**

- added Jurisdiction.lastScrapedAt, openstates/issues#32
v2.4 (April 2020)

• removed unused fields from graph (organization.links, organization.other_names)

v2.3 (August 2019)

• add experimental full text search via searchQuery parameter to bills node

v2.2 (June 2019)

• add openstatesUrl to bills query
• speed improvements

v2.1 (Feb 2019)

• fix lat-lon behavior to limit to active memberships
• improve handling of retired legislators
• fix type of maximum_memberships
• bill version ordering is now consistent

v2.0 (January 2019)

• bugfix for maximum_memberships type
• bugfix for versions field
• improve tests

Beta Release (November 2018)

• API Keys are now required
• consider classification when using current_memberships
• fix geo filtering
• add openstatesUrl to Bill node for ease of linkage to OpenStates.org
• add Person.oldMemberships as analog to currentMemberships
• add actionSince filter to bills node
• fix 500 errors/optimization when using GraphQL fragments
• addition of basic protection for excessive queries
• add totalCount to assist in pagination
• add Organization.currentMemberships
Preview Release 1 (May 2018)

- fix for people pagination
- add updatedSince for people
- add sponsor argument for bills node
- allow votes to take pagination parameters
- allow traversing to votes from person

Preview Release 0 (Dec 2017)

Initial draft release of the API, no backwards-compatibility guarantee made.
Open States’ origin is as a community-driven project, and is only possible because of the sustained effort of dozens of volunteers.

The guides below will walk you through how we work, and various ways you can contribute:

### 3.1 Getting Started

No matter how experienced you are, it is a good idea to read this section before diving into Open States’ code.

This guide assumes a basic familiarity with using the command line, git, and Python.

No worries if you aren’t an expert though, we’ll walk you through the steps. And as for Python, if you’ve written other languages like Javascript or Ruby you’ll probably be just fine. Don’t be afraid to ask for help either!

#### 3.1.1 Installing docker

The first thing you will need to do is get a working docker environment on your local machine. We’ll do this using Docker. No worries if you aren’t familiar with Docker, you’ll barely have to touch it beyond what this guide explains.

Install Docker and docker-compose (if not already installed on your local system):

(a) Installing Docker:

- On OSX: [Docker for Mac](#)
- On Windows: [Docker for Windows](#)
- On Linux: Use your package manager of choice or follow Docker’s instructions.

(Docker Compose is probably already installed by step 1(a) if not, proceed to step 1(b))

(b) Installing docker-compose:

- For easy installation on macOS, Windows, and 64-bit Linux.

Ensure that Docker and docker-compose are installed locally:
Of course, your versions will differ, but ensure they are relatively recent to avoid strange issues.

### 3.1.2 Installing pre-commit

To help keep the code as manageable as possible we **strongly recommend** you use pre-commit to make sure all commits adhere to our preferred style.

- See pre-commit’s installation instructions
- Within each repo you check out, run `pre-commit install` after checking out. It should look something like:

  ```
  $ pre-commit install
  pre-commit installed at .git/hooks/pre-commit
  ```

**Note:** If you’re running `flake8` and `black` yourself via your editor or similar this isn’t strictly necessary, but we find it helps ensure commits don’t fail linting. **We require all PRs to pass linting!**

### 3.1.3 Get Familiar With Our Processes

We’re glad to have you joining us, taking a few minutes to read the following pages will help you be a better member of our community:

- Our [Code of Conduct](#) is important to us, and helps us maintain a healthy community.
- We also have a [guide to help you learn where to get help](#) that you should look over.
- There’s also a [repository overview and roadmap](#) that can be helpful context for the work we’re doing here.

### 3.2 Overview & Roadmap

Open States is a fairly large & somewhat complex project comprised of many moving parts with a long history. As you look to contribute, it may be beneficial to understand a little bit about the various components.

#### 3.2.1 Repositories

These repositories make up the core of the project, if you’re looking to contribute there’s a 90% chance one of these is what you’re looking for.

- openstates-scrapers - Open States bill & vote scrapers.
- people - Open States People data, maintained as editable YAML files
- openstates-core - Open States data model & scraper backend.
- text-extraction - Text extraction powering full text search.
• openstates.org - Powers https://openstates.org/ website & API.
• openstates-metadata - Data on states and their districts, etc. that changes rarely.
• documentation - https://docs.openstates.org/ (you’re reading it now)

These repositories are other pieces of our infrastructure, but are generally not interesting to the average person.

• task-definitions - Tasks for our task runner.
• maintenance-scripts - Internal scripts used for various maintenance tasks.
• indiana-docs - A proxy for fetching Indiana’s docs without API key.
• blog - https://blog.openstates.org/
• openstates-district-maps - Source for generating Open States’ map tiles.

3.2.2 2020 Roadmap

Our current priorities:

Power User Features

• Add user logins & profiles. (launched January 2020)
• Introduce bill & issue tracking. (launched February 2020)

Data Quality

• Improve data quality and timeliness. (in progress)
• Provide publicly accessible data quality dashboard. (Q3)

API

• Improve speed of most popular graph queries. (launched June 2020)
• Provide simplified endpoints for common queries. (Q3/Q4)
• Introduce a pub/sub type mechanism for staying in sync with bill & vote updates. (Q4/2021Q1)

Bulk Data

• Add new per-state CSV data exports. (launched February 2020)
• <s>Add custom data-export creation page.</s> (replaced with per-session JSON data instead, February 2020)
• Provide bulk geographic data ahead of 2021 redistricting. (Q4)

Community

• Documentation updates (Q3)
• New Contributor Support (Q3/Q4)
• API User Dashboard (launched March 2020)
3.2.3 Recent Major Work

To give a sense of recent priorities, here are major milestones from the past few years:

- Legislation Tracking - Q1 2020
- Restoration of Historical Legislator Data - Q4 2019
- Full Text Search - Q4 2019
- 2019 Legislative Session Updates - Q1 2019
- OpenStates.org 2019 rewrite - Q1 2019
- OpenStates GraphQL API - Q4 2018
- Scraper Overhaul - Throughout much of 2017 we reworked our scrapers to be more resilient and to use an updated tech stack, replacing the one that powered the site from 2011-2016.

3.3 Communication

When joining a new community, it can be tough to figure out where to ask questions, provide feedback, or help out. Don’t worry! As long as you’re respectful and follow our Code of Conduct, we’re happy to have you!

Here are some guidelines regarding the best way to get in touch or contribute. Do note that Open States is a volunteer-powered project, and all of the core developers have day jobs; we’re excited to talk to you, but it may sometimes take a bit of time to get back to you.

3.3.1 Recommendations

Want to ask a general question, have a conversation, or keep up with the community?

The best place is our Discussion Forum. This is the main place for Open States discussion. The core team and many other contributors are present there, and we’re usually able to answer questions in a timely fashion.

Have a private or financial question, or a security concern?

Email contact@openstates.org; only the administrative team can see these.

Have you found an error or issue in the Open States data? Have a technical issue not related to the data itself?

File an issue on our bug tracker. And before you do, quickly check whether anyone else there has already reported the same bug.

Want to contribute to the project more regularly?

We also have a Slack that you can join. It is not the best place for asking questions better handled by filing an issue or opening a conversation on GitHub, but you’re welcome to join if you’re interested in being a part of the community.

3.3.2 Discouraged Methods of Communication

Please avoid using these channels to get in touch with us:

Personal email addresses of Open States developers

Please respect our boundaries & refrain from contacting any of the developers directly, unless we ask you to do so.
Twitter (or any other social media)

We mainly use the @openstates twitter account to make announcements, and don’t have the resources to provide technical support or other feedback on Twitter.

### 3.4 Contributing to Scrapers

Scrapers are at the core of what Open States does, each state requires several custom scrapers designed to extract bills, legislators, committees, and votes from the state website. All together there are around 200 scrapers, each one essentially independent, which means that there is always more work to do, but fortunately plenty of prior work to learn from.

#### 3.4.1 Checking out

Fork and clone the main scraper repository:

- Visit https://github.com/openstates/openstates-scrapers and click the ‘Fork’ button.
- Clone your fork using your tool of choice or the command line:

  ```bash
  $ git clone git@github.com:yourname/openstates-scrapers.git
  Cloning into 'openstates'...
  ```

- And remember to install pre-commit:

  ```bash
  $ pre-commit install
  pre-commit installed at .git/hooks/pre-commit
  ```

**Warning:** Before cloning on a Windows computer, you will need to disable line-ending conversion. `git config --global core.autocrlf false` After cloning and entering the repo, you’ll likely want to set global line-ending conversion back to `true`, and set local conversion to `false`.

#### 3.4.2 Repository overview

At this point you’ll have a local `openstates-scrapers` directory. Within it, you’ll find a directory called `scrapers` lets take a look at it:

```bash
$ ls scrapers
__init__.py dc in mn nj pr va
ak de ks mo nm ri vi
al fl ky ms nv sc vt
ar ga la mt ny sd wa
az hi ma nc oh tn wi
cr ia md nd ok tx wv
co id me ne or ut wy
cr il mi nh pa uts
```

This directory has 50+ python modules, one for each state.

Let’s look inside one:
Some states’ directories will differ a bit, but all will have __init__.py and bills.py. The __init__.py file for each state has basic metadata on the state including a list of sessions. Other files contain the scrapers, typically named bills, votes, etc.

### 3.4.3 Running Your First Scraper

Let’s run your state’s bills scraper (substitute your state for ‘nc’ below)

```
$ docker-compose run --rm scrape nc bills --fastmode --scrape
```

The parameters you pass after docker-compose run --rm scrape are passed to os-update. Here we’re saying that we’re running NC’s scrapers, and that we want to do it in “fast mode”. By default, os-update imports results into a postgres database; the --scrape flag skips that step.

You’ll see the run plan, which is what the update aims to capture; in this case we’re scraping the state website’s data into JSON files:

```
nc (scrape)
  bills: {}
```

Then legislative posts and organizations get created, which is mostly boilerplate:

```
08:46:35 INFO openstates: save jurisdiction North Carolina as jurisdiction_ocd-
  → jurisdiction-country:us-state:nc-government.json
08:46:35 INFO openstates: save organization North Carolina General Assembly as,
  → organization_01d6327c-72d2-11e7-8df8-0242ac130003.json
08:46:35 INFO openstates: save organization Executive Office of the Governor as
  → organization_01d63560-72d2-11e7-8df8-0242ac130003.json
08:46:35 INFO openstates: save organization Senate as organization_01d636e6-72d2-11e7-
  → 8df8-0242ac130003.json
08:46:35 INFO openstates: save post 1 as post_01d63a06-72d2-11e7-8df8-0242ac130003.
  → json
08:46:35 INFO openstates: save post 2 as post_01d63b96-72d2-11e7-8df8-0242ac130003.
  → json
08:46:35 INFO openstates: save post 3 as post_01d63cea-72d2-11e7-8df8-0242ac130003.
  → json
08:46:35 INFO openstates: save post 4 as post_01d63e34-72d2-11e7-8df8-0242ac130003.
  → json
08:46:35 INFO openstates: save post 5 as post_01d63f74-72d2-11e7-8df8-0242ac130003.
  → json
```

And then the actual data scraping begins, defaulting to the most recent legislative session:

```
08:46:36 INFO openstates: no session specified, using 2017
08:46:36 INFO scrapelib: GET - http://www.ncga.state.nc.us/gascripts/
  → SimpleBillInquiry/displaybills.pl?Session=2017&tab=Chamber&Chamber=Senate
  → BillLookUp.pl?Session=2017&BillID=S1
08:46:39 INFO openstates: save bill SR 1 in 2017 as bill_03c7edb4-72d2-11e7-8df8-
  → 0242ac130003.json
  → BillLookUp.pl?Session=2017&BillID=S2
```

(continues on next page)
Depending on the scraper you run, this part takes a while. Some scrapers can take hours to run depending on the number of bills and speed of the state’s website.

**Note:** It is often desirable to bail out of running the whole scrape (Ctrl-C) after it has gotten a bit of data, instead of letting it run the entire scrape.

To review the data you just fetched, you can browse the _data/nc/ directory and inspect the JSON files. If you’re trying to make a small fix this is often sufficient, you can confirm that the scraped data looks correct and move on.

**Note:** It is of course possible that the scrape fails. If so, there’s a good chance that isn’t your fault, especially if it starts to run and then errors out. Scrapers do break, and there’s no guarantee North Carolina didn’t change their legislator page yesterday, breaking our tutorial here.

If that’s the case and you think the issue is with the scraper, feel free to get in touch with us or file an issue.

At this point you’re ready to run scrapers and contribute fixes. Hop onto our GitHub ticket queue, pick an issue to solve, and then submit a Pull Request!

### 3.4.4 Importing Data

Optionally, if you’d like to see how your scraped data imports into the database, perhaps to diagnose an issue that is happening after the scrape, pop over to getting a working database to see how to get a local database that you can import data into.

Once that’s done, make sure that the db image from openstates.org is running:

```bash
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
27fe691ad7c5 mdillon/postgis:11-alpine "docker-entrypoint.s..." 3 hours Up 3 hours 0.0.0.0:5405->5432/tcp openstatesorg_db_1
```

Your output will vary, but if you don’t see something named openstatesorg_db running you should run this command (from the openstates.org directory, not your scraper directory):

```bash
$ docker-compose up -d db
```

Now, when you want to run imports, you can drop the `--scrape` portion of the command you’ve been running. Or if you just want to test the import of already scraped data you can replace it with `--import`.

### 3.4. Contributing to Scrapers
An import looks something like this:

```bash
$ docker-compose run --rm scrape fl bills --fast
... (truncated) ...
23:03:34 ERROR openstates: cannot resolve pseudo id to Person: ~("name": "Grant, M.")
23:03:36 ERROR openstates: cannot resolve pseudo id to Person: ~("name": "Rodrigues,
˓
→
→R.")
fl (import)
bills: {}
import:
  bill: 0 new 0 updated 2620 noop
  jurisdiction: 0 new 0 updated 1 noop
  vote_event: 21 new 12 updated 533 noop
```

The errors about unresolved pseudo-ids can safely be ignored, as long as you see the final run report the data you scraped is available in your database.

The number of objects of each type that were created & updated are available for spot checking, as well as the total number of items that were seen that already exactly matched what was in the database. These can be useful stats as you try to see if your local changes to a scraper have the impact you expect.

### 3.5 State-Specific Scraper Info

#### 3.5.1 California MySQL

California is a unique state that takes a couple of extra steps to get working locally.

California provides MySQL dumps of their data, and in order to use those we start up a local MySQL instance and read from that.

To download the data for the current session:

```bash
docker-compose run --rm ca-download
```

(You can append --year YYYY to instead select data for a given year.)

This will start a local MySQL image as well, that image will need to stay up for the next step, which is running a scrape like normal:

```bash
docker-compose run --rm ca-scrape ca bills --fast
```

#### 3.5.2 State API Keys (NY & IN)

Unfortunately, some states find it necessary to require API Keys (or other credentials) to access their best data.

Despite the difficulties this creates for contributors, in the interest of ensuring we have the best possible data we’ve made the decision that we will use this data where possible.

Our policy:

- We will maintain (when possible) two copies of credentials, one for development and one for production. (Thus minimizing the chance that a mistake made w/ a development key will jeopardize our ability to scrape.)
- We encourage developers to get an API key of their own, but if necessary we can share our testing key in limited circumstances.

Currently only a few states require API keys:
3.6 Testing Scrapers

One of the first things people new to the project tend to notice is that there aren’t a lot of tests in the scrapers. Over the years we’ve evolved a de facto policy of somewhat discouraging tests, which is definitely an unusual stance to take and warrants explanation.

3.6.1 Intentionally Fragile Scrapers

When it comes to scrapers, there are two major types of breakage:

1) the scraper collects bad information and inserts it into the database
2) the scraper encounters an error and quits without importing data

Given a choice, the second is greatly preferable. Once bad data makes it into the database, it can be difficult to detect and remove. On the other hand, the second can be triggered to alert us immediately and someone can evaluate the proper fix.

The best way to favor the second over first is to write “intentionally fragile” scrapers. That is, scrapers that raise an exception when they see unexpected input.

While it is possible to try to write a resilient scraper that recovers, by nature these scrapers are more likely to produce the first kind of error, and so we encourage scraper writers to be conservative in what errors are suppressed.

Here’s an example of an overly permissive scraper:

```python
party_abbr = doc.xpath('//span[@class="partyabbr"]')
if party_abbr == 'D':
    party = 'Democratic'
elif party_abbr == 'R':
```

(continues on next page)
The following would be preferred:

```python
party_abbr = doc.xpath('//span[@class="partyabbr"]
party = {'D': 'Democratic', 'R': 'Republican'}[party_abbr]
```

This code would raise a `KeyError` the first time a new party is found. This forces someone to take a look, fix the scraper with an entry for the new party, and then the scraper will be able to run again with correct data.

### 3.6.2 Testing Scrapers Is Hard

On most software projects a failing test means that something is broken, and passing tests should mean that things are working just fine.

In our experience however, the majority of the “breaks” that occur in scrapers are due to upstream site changes.

In the past the fragile nature of scrapers has led to people writing a lot of bad tests, which is where our stance of somewhat discouraging tests has come from. An example of a bad test:

```python
def extract_name(doc):
    return doc.xpath('//h2[@class="legislatorName"]').text_content().strip()

def test_extract_name():
    # probably a snapshot of the page at some point in time
    EXAMPLE_LEGISLATOR_HTML = '...'""

    doc = lxml.html.fromstring(EXAMPLE_LEGISLATOR_HTML)
    assert extract_name(doc) == 'Erica Example'
```

With a test like this:

- As soon as the HTML changes, the scraper will start failing, but the tests will still pass.
- The scraper will then be updated, breaking the test.
- The test HTML will be updated, fixing the test.

But since the initial scraper breakage isn’t predicted by a failing test, this type of test really doesn’t serve us any purpose and just results in extra code to maintain every time the scraper needs a slight change.

### 3.6.3 Other Strategies

Of course this isn’t to say that we just abandon the idea of testing, altogether.

If you’re more comfortable writing tests, say you’re parsing a particularly nasty PDF and want to run it against some test data: a test might make sense there as a way to be confident in your own code, by all means, write a test.

We also have some other strategies to help ensure data quality:
Validate Scraper Output

Scraper output is verified against JSON schemas that protect against common regressions (missing sources, invalid formatted districts, etc.) - most of these tests can be written effectively against scraper output across the board, and in doing so also applies universally across all 50 states.

We also aim for our underlying libraries like openstates-core to be as well tested as possible. (To be 100% clear, our lax testing philosophy only applies to site-specific scraper code, not these support libraries.)

Run Scrapers Regularly

In a sense, the scrapers are tested every night by being run. This is why the intentionally fragile approach is so important; those failures are in essence the same as integration test failures. Of course, this doesn’t tell us if the scraper is picking up bad data, etc., but combined with validation we can be fairly confident in our data.

Test Utilities

One area we can definitely improve upon is our use of (and then thorough testing of) common functions. Right now (largely because of the great variety of authors, etc.) many scrapers do similar things like conversion of party abbreviations and whitespace normalization in slightly different ways. We should be making a push to use common utility functions and thoroughly test those.

3.7 Working On openstates.org

openstates.org is the public-facing result of all the work we do. The site is built in Django and includes the web frontend and API.

3.7.1 Checking out

Fork and clone the openstates.org repository:

• Visit https://github.com/openstates/openstates.org and click the ‘Fork’ button.
• Clone your fork using your tool of choice or the command line:

```bash
$ git clone git@github.com:yourname/openstates.org.git
Cloning into 'openstates.org'...
```

• And remember to install pre-commit:

```bash
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

3.7.2 Getting a working database

Whether you’re aiming to work on openstates.org or just want to import scraped data, you’ll need postgres server running in your docker environment.

If you haven’t set up docker yet, see Installing docker.

First, make sure that the database is running with:
Then, to initialize an empty Open States database:

```
./docker/init-db.sh
```

If you’re working on scrapers you’ll now find that this database is available to your scrape processes!

### 3.7.3 Running Tests

You can run the tests for the project via:

```
./docker/run-tests.sh
```

You can also append standard pytest arguments such as `-x` to bail on first failure.

Example of running just the v1 tests, bailing on error:

```
./docker/run-tests.sh v1 -x
```

### 3.7.4 Repository overview

The project is rather large, with quite a few django apps, here’s a quick guide:

**Django Apps:**

- `bulk/` - handles bulk downloads on the website
- `dashboards/` - dashboards for viewing various statistics
- `geo/` - geography services for legislator lookup
- `graphapi/` - powers GraphQL API
- `profiles/` - user and subscription management
- `public/` - public-facing pages (bulk of the site)
- `utils/` - utilities shared by the other applications
- `v1/` - backwards-compatibility shim implementing much of the old v1 API

**Other Stuff:**

- `ansible/` - the files used to deploy OpenStates.org are here
- `docker/` - special scripts for running tests, etc. within docker
- `openstates/` - core Django settings files
- `static/` - various static assets, including frontend code
- `templates/` - Django templates

### 3.7.5 Running openstates.org

Simply running `docker-compose up` should start django & the database, then browse to `http://localhost:8000` and you’ll be looking at your own local copy of openstates.org
3.7.6 Running outside of Docker

It might be desirable to test outside of docker sometimes to bypass caching or other issues that make development within the docker environment difficult. If so, you can install goreman (or any foreman clone) and run `goreman start`.

3.8 Contributing People Data

Person data is maintained in the `openstates/people` repository. This repository contains YAML files with all the information on given individuals, as well as scripts to work with & maintain the data.

Also, please note that this portion of the project is in the public domain in the United States with all copyright waived via a CC0 dedication. By contributing you agree to waive all copyright claims.

3.8.1 Checking out

Fork and clone the people repository:

- Visit https://github.com/openstates/people and click the ‘Fork’ button.
- Clone your fork using your tool of choice or the command line:

  ```
  $ git clone git@github.com:yourname/people.git
  Cloning into 'people'..
  ```

- And remember to *install pre-commit*:

  ```
  $ pre-commit install
  pre-commit installed at .git/hooks/pre-commit
  ```

3.8.2 Repository overview

The repository consists of a few key components:

- `settings.yml` Settings for state legislatures, including the number of seats, and current vacancies.
- `data/` Data files in YAML format on legislators, organized by state & status.
- `scripts` Various scripts used to maintain the data.
- `scrape/` Experimental new people scrapers, work-in-progress.

To run a script using `docker-compose` you can run a command like:

```
docker-compose run --rm people ./scripts/lint_yaml.py
```

3.8.3 Common tasks

Updating legislator data by hand

Let’s say you call a legislator and find out that they have a new phone number, contribute back!

See `schema.md` for details on the acceptable fields. If you’re looking to add a lot of data but unsure where it fits feel free to ask via an issue and we can either amend the schema or make a recommendation.
0. Start a new branch for this work
1. Make the edits you need in the appropriate YAML file. Please keep edits to a minimum (e.g. don’t re-order fields)
2. Submit a PR, please describe how you came across this information to expedite review.

Retiring a legislator

0. Start a new branch for this work
1. Run `./scripts/retire.py` on the appropriate legislator file(s)
2. Review the automatically edited files & submit a PR.

Updating an entire state via a scrape

Let’s say a North Carolina has had an election & it makes sense to re-scrape everything for that state.

0. Start a new branch for this work
1. Scrape data using Open States’ Scrapers
2. Run `./scripts/to_yaml.py` against the generated JSON data, this will populate the incoming/ directory
3. Check for merge candidates using `./scripts/merge.py --incoming nc`
4. Manually reconcile remaining changes, will almost certainly require some retirements as well.
5. Check that data looks clean with `./scripts/lint_yaml.py nc --summary` and prepare a PR.

Updating a single field for many people

Let’s say you want to add foobar_id to a ton of legislators from your own data set or similar.
TBD - We need to create a tool that will aid in this as it will prove a common use case & we can lower the barrier here.

3.8.4 Scripts

Several scripts are provided to help maintain/check the data.

**to_yaml.py**

```
to_yaml.py [OPTIONS] INPUT_DIR

Convert pupa scraped JSON in INPUT_DIR to YAML files for this repo.

Convert a pupa scrape directory to YAML. Will put data into incoming/ directory for usage with merge.py's --incoming option.
```

**lint_yaml.py**
lint_yaml.py [OPTIONS] [ABBREVIATIONS]

Lint YAML files, optionally also providing a summary of state's data.

<ABBREVIATIONS> can be provided to restrict linting to select states.

Options:
- -v, --verbose
--summary / --no-summary Print summary after validation errors.

merge.py

merge.py [OPTIONS]

Script to assist with merging legislator files.

Can be used in two modes: incoming or file merge.

Incoming mode analyzes incoming/ directory files (generated with to_yaml.py) and discovers identical & similar files to assist with merging.

File merge mode merges two legislator files.

Options:
- --incoming TEXT Operate in incoming mode, argument should be state abbr to scan.
- --retirement TEXT Set retirement date for all people marked retired (in incoming mode).
- --old TEXT Operate in merge mode, this is the older of two files & will be kept.
- --new TEXT In merge mode, this is the newer file that will be removed after merge.
- --help Show this message and exit.

new_person.py

new_person.py [OPTIONS]

Create a new person record.

Arguments can be passed via command line flags, omitted arguments will be prompted.

Be sure to review the file and add any additional data before committing.

Options:
- --fname TEXT First Name
- --lname TEXT Last Name
- --name TEXT Optional Name, if not provided First + Last will be used
- --state TEXT State abbreviation
- --district TEXT District
- --party TEXT Party
- --rtype TEXT Role Type

(continues on next page)
3.9 Text Extraction

The bill scrapers scrape the web and pull down metadata, including links to various versions of the bills. As a later step, we extract the actual text of the bill so that it can be indexed for search and other uses.
3.9.1 Checking out

Fork and clone the text-extraction repository:

- Visit https://github.com/openstates/text-extraction and click the ‘Fork’ button.
- Clone your fork using your tool of choice or the command line:

  ```shell
  $ git clone git@github.com:yourname/text-extraction.git
  Cloning into 'text-extraction'...
  ```

- And remember to *install pre-commit*:

  ```shell
  $ pre-commit install
  pre-commit installed at .git/hooks/pre-commit
  ```

3.9.2 Repository overview

The text extraction code itself is written as a standalone Python script `text_extract.py` that uses configuration and utility functions from within `extract/`.

You’ll also notice a directory called `raw/` – this contains a sampling of bills for each state that we can use to test text-extraction.

Typically if you’re making changes in the repository you’ll be editing files within `extract/`, we’ll come back to that later.

3.9.3 Running text_extract

Just like in other repositories, we’ll use docker-compose to run the code. In this case docker-compose is running `text_extract.py`, an all-in-one tool that has a few useful subcommands:

```shell
Usage: text_extract.py [OPTIONS] COMMAND [ARGS]...
Options:
  --help       Show this message and exit.
Commands:
  reindex-state rebuild the search index objects for a given state
  sample       obtain a sample of bills to extract text from
  status       print a status table showing the current condition of...
  test         run sample on all states, used for CI
  update       update the saved bill text in the database
```

For the purposes of development, `sample` and `update` are the only two commands that you’ll need to look at.

Let’s go ahead and run sample against NC:

```shell
$ docker-compose run --rm text-extract sample nc
```

(continues on next page)
The exact output and number of bills will vary across states, but should be pretty similar.

This command just did a lot:

- Read in the file `raw/nc.csv` to get a list of bills to sample.
- Downloaded those files (assuming this was the first run) to `raw/nc/` so future runs will be faster.
- Used the extraction function(s) defined in `extract/__init__.py` for NC to extract text from the given documents.
- Wrote that output to `text/nc/` so you can compare.

You'll also notice that it helpfully prints the number of bytes of text extracted, this is useful as a first check. Let’s go ahead and look at the shortest one, `text/nc/2017-SB 753-Edition 1.pdf.txt`. (Your run may differ, pick whichever you prefer.)

```
$ cat "text/nc/2017-SB 753 Edition 1.pdf.txt"
A BILL TO BE ENTITLED
AN ACT PROVIDING THAT THE DEPOSIT OF CURRENCY AND COINS INTO A CASH
VAULT THAT PHYSICALLY SECURES THE CASH AND ELECTRONICALLY
RECORDS THE DEPOSIT DAILY IN AN OFFICIAL DEPOSITORY BANK QUALIFIES
AS A DAILY DEPOSIT UNDER THE LOCAL GOVERNMENT BUDGET AND FISCAL
CONTROL ACT FOR FRANKLIN AND WAKE COUNTIES AND THE
MUNICIPALITIES IN THOSE COUNTIES.
The General Assembly of North Carolina enacts:
SECTION 1. Section 2 of S.L. 2011-89 reads as rewritten:
"SECTION 2. This act applies only to the City of Winston-Salem only.Winston-Salem,
Franklin County and the municipalities in Franklin County, and Wake County and the
municipalities in Wake County."
SECTION 2. This act is effective when it becomes law.
```

This looks complete, but to check, go ahead and open the equivalent source file, in this case `raw/nc/2017-SB 753-Edition 1.pdf` and confirm visually that all the text was extracted. Don’t worry about formatting, or the preamble, as we’ll often exclude that and just aim for the interesting bits of the text.

### 3.9.4 Making changes

Let’s say that we discover that a state has started publishing their bills in a new format. Perhaps Alabama switches from PDF to HTML. It’d first be good to add some of these new bills to the sample csv, which you can do manually or by invoking sample with the `--resample` flag:

```
docker-compose run --rm text-extract sample --resample al
```

Running would result in some warnings being printed and some zero byte files.

To actually handle the HTML documents we’d open up `extract/__init__.py` and find the `CONVERSION_FUNCTIONS` dictionary, you’ll see a line like:
The way extraction works is by matching a document found in a scrape to an appropriate function, in this case PDFs will be sent through the `extract_line_numbered_pdf` function.

If the new HTML was wrapped in a given element, perhaps with `<div id="billtext">` we could just update that line to look like:

```
CONVERSION_FUNCTIONS = {
    "al": {
        "application/pdf": extract_line_numbered_pdf,
        "text/html": extractor_for_element_by_id("billtext")
    },
    ...
}
```

And we’d be good to go.

### 3.9.5 Tips & Tricks

- Functions already exist for common configurations of PDF, HTML, Word Doc, and even OCR. Rarely will you need to write a custom function, always look at the options first.
- When dealing with PDFs, most are either handled by `extract_line_numbered_pdf` or `extract_sometimes_numbered_pdf`, the difference is that “sometimes numbered PDF” accounts for cases where 90% or so of bills are numbered, but a few (often resolutions) are not numbered.

### 3.9.6 Formatting Guidelines

**How far do we go? Should we strip punctuation? Newlines? Whitespace? Section headings?**

- Try not to be too aggressive with punctuation stripping, search indices/etc. can easily strip it later, but it can be handy if someone decides they want to search for things like “§ 143C-4-8.b”
- Ideally leave newlines as-is since it makes looking at changes a lot nicer for humans and stripping newlines out for final products (search/text comparison/etc.) is trivial.
- Collapsing spaces/etc. is recommended, but not required.
- Removal of section headers/etc. is fine, but if the only reason you’re writing a new function instead of using a common one is to do this, reconsider.

When in doubt, ask, you may have encountered something we haven’t considered yet and we can discuss the best practice and add it here.

**Should we include bill digests?**

There isn’t a need to, but it doesn’t hurt if separating the two is difficult.

**What about additions & deletions?**

See text-extraction issue #6.
# 3.10 Contributing Documentation

If you notice any issues on these docs, or just want to make them better the source is in the [openstates/documentation](https://github.com/openstates/documentation) repository.

This is a fairly standard [Sphinx](https://sphinx-doc.org/) project,

## 3.10.1 Checking out

Fork and clone the documentation repository:

- Visit [https://github.com/openstates/documentation](https://github.com/openstates/documentation) and click the ‘Fork’ button.
- Clone your fork using your tool of choice or the command line:

```
$ git clone git@github.com:yourname/documentation.git
Cloning into 'documentation'..
```

## 3.10.2 Building Docs Locally

Step 1) Install poetry if you haven’t already ([https://python-poetry.org/docs/](https://python-poetry.org/docs/))
Step 2) Run `poetry install` to build virtualenv
Step 3) Run `poetry run make html` to build docs.

# 3.11 Code of Conduct

## 3.11.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

## 3.11.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
• Public or private harassment
• Publishing others’ private information, such as a physical or email address, without their explicit permission
• Other conduct which could reasonably be considered inappropriate in a professional setting

3.11.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

3.11.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

3.11.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at contact@openstates.org. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

3.11.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

**Community Impact:** Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

**Consequence:** A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

**Community Impact:** A violation through a single incident or series of actions.

**Consequence:** A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.
3. Temporary Ban

**Community Impact:** A serious violation of community standards, including sustained inappropriate behavior.

**Consequence:** A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

**Community Impact:** Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

**Consequence:** A permanent ban from any sort of public interaction within the project community.

3.11.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla’s code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.
Understanding the Data

Open States data adheres to a schema that has evolved over our 11+ years of working with legislative data. Our goal is to provide as much uniformity across states as possible while still allowing for the wide diversity of legislative processes between the states.

These docs both catalog the schema and attempt to explain some of those choices, particularly where they might be surprising.

4.1 Main Concepts

The main concepts are defined below:

**Jurisdiction**  Essentially just another word for “State” in our context. (Includes DC and Puerto Rico.)

**Session (aka LegislativeSession)**  A period of time in a legislature where the same members serve together, typically punctuated by elections. All bills in a session will be uniquely numbered. (e.g. HB 1 in the 2017 session is typically not the same bill as in the 2019 session)

**Bill**  Represents all types of legislation whether it is a bill, resolution, etc.

**Vote (aka VoteEvent)**  A vote among members of the legislature, typically an entire chamber but can also be a committee vote.

**Person (aka Legislator)**  Any person that is associated with the legislature.

**Organization**  A generic term used to represent a few different concepts: legislatures, chambers, committees, and political parties.

**Post**  A particular role within an organization, typically used to represent a seat in the legislature. (e.g. the District 4 post in the North Carolina Senate Organization)

**Membership**  Ties a Person to a Post for a duration of time.

You’ll notice these concepts mostly correspond to the v2 GraphQL Root Nodes.
4.2 Categorization

One of the ways that we add value to the data we provide is by attempting to classify bills, actions, and votes across states.

This allows us to let states use their own names for these things, but for us to try to provide some mapping to a common simplified view of the legislative process.

4.2.1 Bill Types

State legislatures deal with more than bills. Despite the name of the bill objects in our data we take in all types of legislation that a state might produce. Generally looking at the bill_id will help you determine the type of legislation, but to make things easier across states we provide a type field on bills. This field is a list with one (or more) of the following values:

Common Values:
- bill
- resolution
- joint resolution
- concurrent resolution
- constitutional amendment

Some states also make use of additional types such as ‘contract’, ‘nomination’, ‘memorial’ and more.

4.2.2 Action Types

Although most states follow very similar parliamentary procedure the names that their bill status systems use for various actions almost never match up. To make analysis and the building of certain types of tools easier we attempt to classify common actions. In using our data you’ll find these values in the classification field of actions.

- filing - bill is filed (where this is a separate action from introduction)
- introduction - introduced, typically the first action
- reading-1 - first reading (often same as introduction)
- reading-2 - second reading
- reading-3 - third reading (often same as passage)
- passage - bill is passed by the chamber
- failure - bill fails to proceed from the chamber
- withdrawal - bill is withdrawn
- substitution - a substitution is made to the bill text
- deferral - consideration of the bill was deferred
- receipt - a bill was received by another chamber
- referral - a bill was sent somewhere for consideration
- referral-committee - a bill was sent to a committee for consideration
- became-law - the bill became law (through signature or inaction)
• amendment-introduction - an amendment is introduced
• amendment-passage - an amendment passes
• amendment-withdrawal - an amendment is withdrawn
• amendment-failure - an amendment fails to pass
• amendment-amendment - an amendment is amended
• amendment-deferral - consideration of an amendment is deferred
• committee-passage - the bill passes the current committee (unknown outcome, typically favorable)
• committee-passage-favorable - the bill passes the current committee favorably
• committee-passage-unfavorable - the bill passes the current committee with an unfavorable report
• committee-failure - the bill fails to advance out of committee
• executive-receipt - the bill is sent to the governor
• executive-signature - the governor signs the bill
• executive-veto - the governor vetos the bill
• executive-veto-line-item - the governor uses a line-item veto to strike part of a bill
• veto-override-passage - a veto override vote occurred and succeeded
• veto-override-failure - a veto override vote occurred and failed

4.2.3 Vote Types

Similarly to actions, we make an effort to categorize the motion being voted upon. You’ll find these values in the categorization field on VoteEvents.

Possible values:

• bill-passage - This is a vote to pass (either out of committee or a chamber)
• amendment-passage - Vote on amending a bill
• veto-override - Vote to override an executive veto

4.3 Session Naming

States name their sessions drastically differently, and sometimes inconsistently even within their own site. (49th vs 2008 Regular Session). As our goal is to help smooth these inconsistencies we put forward this guide to naming sessions within state metadata. (See https://github.com/sunlightlabs/openstates/issues/81 for discussion on the topic)

4.3.1 Default Session Names

The sessions list within terms is dangerous to change as all bill data is keyed off it. As a rule these should be short and generally useful for the scraper to make the appropriate decisions on what data to scrape.

If a state calls its 1st special session in 2010 ‘2010E1’ this is a perfectly acceptable name for the session in the metadata. Similarly 49th-regular, 2009-Special-B, etc. are fine names. Generally names with spaces should be avoided simply for ease of construction of URLs, etc. In states where spaces are already in use it is fine to continue to use them.
The one caveat is that if a state uses a unique ID that has no bearing on the session itself such as ‘7323’ for the 2011 session, this should not be used. Instead add some mapping that maps a session name that is descriptive to their internal ids.

4.3.2 Session Display Names

Because the most convenient name to refer to a session is often far from what a user might expect to see upon opening a mobile application, the session_details dict supports a display_name key.

Suitable display names are descriptive but also short and obey a given style.

General Rules

- All sessions should be in title case.
- Fewer than 20 characters is highly preferable.
- Months should be abbreviated to 3 letters (Jan., Feb., Jun., Dec.)

Ordinals

If no special sessions are present:
- [Ordinal] Legislature

If special sessions are present:
- [Ordinal] Regular Session
- [Ordinal], [Ordinal] Special Session

Examples:
- 82nd Legislature
- 82nd Regular Session
- 82nd, 3rd Special Session

Years

- [Year/Year-Range] Regular Session
- [Year/Year-Range], [Ordinal] Special Session
- [Mon. Year] Special Session

Examples:
- 2010 Regular Session
- 2011-2012, 4th Special Session
- Dec. 2011 Special Session