politico-elections Documentation

Release 0.0.1

Jon McClure, Tyler Fisher, Sarah Frostenson, Jeremy C.F. Lin, Lily

Dec 13, 2017

Contents:


POLITICO

CHAPTER 1

Why this?

Beginning with the 2018 election cycle, the POLITICO Interactives Team will be handling POLITICO's coverage of election results. Our plans include covering everything from statehouses to presidential elections. Thus, we needed a centralized, reusable system to collect, manage and publish election results. This system also needs to integrate with a larger campaign database managed by all of POLITICO.

1.1 Principles

1.1.1 Election Results Done Quick

A primary function of this rig is to publish election results from the AP Elections API. We want to do that as quickly and efficiently as possible, so this system gets as much data as possible before the election begins into our database. That includes candidates, parties, and geographical information. We bake all of this data out before the election begins.

Once we gather this information, all we have to do for results is gather data from the AP Elections API (using elex, the best tool for working with AP election results), do minimal filtering in bash, and publish JSON to Amazon S3.

1.1.2 Relational election data

This system contains a fully relational model for election data. We borrowed a lot of inspiration from the fine folks making OpenCivicData, but we also account for modeling vote totals. This model will end up becoming a part of a larger campaign database that will include FEC data and campaign staff information, so the highly relational model is important to us.

See models.

1.1.3 Modular data visualization

The front-end code within this system does not handle any data visualization. Instead, we do all of that work in separate repos that we install as node modules and use across the application. That means we can use, for example, a county results map, on multiple types of pages with ease.

1.2 What's in it

1.2.1 The backend

- Django as the overarching backend framework
- PostgreSQL as the database of choice
- Django REST Framework to serialize our models to JSON
- Elex to process AP election results
- Fabric to handle server management and data processing

And lots more…

1.2.2 The frontend

- redux orm to take serialized JSON and build the relationships between live results and contextual data
- Preact to manage front-end views

CHAPTER 2

Page structure

Most of this project is organized around the structure of the pages it's built to produce.

We produce pages that aggregate elections for offices in legislative bodies, individual pages of election results for executive offices, i.e., the president and state governors, and pages that show all races in a particular state.

Here is a non-exhaustive sample of the pages we build for:

Path	Page type
Home	
`/election-results/2020/`	Homepage for a national general election
President	
`/election-results/2020/president/`	National presidential election results
`/election-results/2020/president/texas/`	State presidential election results
U.S. House and Senate	
`/election-results/2020/senate/`	U.S. Senate election results
`/election-results/2020/senate/texas/`	U.S. Senate election results in Texas
`/election-results/2020/house/`	U.S. House election results
`/election-results/2020/house/texas/`	U.S. House election results in Texas
State races	
`/election-results/2020/texas/`	All elections in Texas
`/election-results/2020/texas/governor/`	Texas governor
`/election-results/2020/texas/senate/`	Texas state senate
`/election-results/2020/texas/house/`	Texas state house
Special elections	
`/election-results/2017/alabama/special-election/dec-12/`	Special elections in Alabama on Dec. 12

Page types correspond to our models: **ElectionCycle**, **Office**, **Body** and **Division**.

- Election cycle page
- Federal executive office page (president)
- State federal executive office page (state results)
- Federal legislative body page (U.S. Senate or House)
- State federal legislative body page (state results)
- State page (all races)
- State executive office page (governor)
- State legislative body page (state house or senate)

Note: Special Elections are a bit of a special case that we host at the state level regardless of the office(s) contested.

See *election models*.

CHAPTER 3

Data structure

Data to build pages is represented by two types: election **results** from the Associated Press and **context** that helps interpret and display those results.

3.1 Results

To process our results as fast as possible, we do the bare minimum data manipulation on election night results from the AP API. The structure contains just the basic keys needed to marry results to the wider context provided by our app. This data is baked out as `./results.json` when publishing.

Listing 3.1: **results.json**

```
{
  "fipscode": "51001",
  "level": "county",
  "polid": "63126",
  "polnum": "49283",
  "precinctsreporting": 17,
  "precinctsreportingpct": 1,
  "precinctstotal": 17,
  "raceid": "47225",
  "statepostal": "VA",
  "votecount": 4879,
  "votepct": 0.457306,
  "winner": true
}
```

3.2 Context

Context is provided by our app and includes all the information needed to display results on our election night pages. The data itself is created through a number of bootstrap procedures. See also *Serialization*. This data is baked out as

`./context.json` when publishing.

Listing 3.2: **context.json**

```
{
"uid": "<{Body|State|Office} UID>",
"content": { /* ... */ },
"elections": [
  {
    "uid": "<Election UID>",
    "date": "2017-11-07",
    "office": { /* ... */},
    "primary_party": null,
    "division": { /* ... */ },
    "candidates": [
      { /* ... */ },
      // ...
    ],
    "override_votes": false,
    "ap_election_id": "47225",
    "called": false,
    "tabulated": false,
    "override_ap_call": true,
    "override_ap_votes": false
  },
  // ...
],
"parties": [
  { /* ... */ },
  // ...
],
"division": {
  /* ... */
  "children": [
    { /* ... */ },
    // ...
  ]
}
}
```

3.3 Other data

There are also supplemental data files we bake out. For example, GeoJSON created by commands-export-geography and county-level census data from commands-run-census.

CHAPTER 4

Models

Models for elections.

4.1 Core models

4.1.1 geography models

class geography.models.**Division**(**args*, ***kwargs*)
A political or administrative geography.

For example, a particular state, county, district, precinct or municipality.

add_intersecting(*division*, *intersection=None*, *symm=True*)
Adds paired relationships between intersecting divisions.

Optional intersection represents the portion of the area of the related division intersecting this division. You can only specify an intersection on one side of the relationship when adding a peer.

get_intersection(*division*)
Get intersection percentage of intersecting divisions.

remove_intersecting(*division*, *symm=True*)
Removes paired relationships between intersecting divisions

save(**args*, ***kwargs*)
uid: `division:{parentuid}_{levelcode}-{code}`

set_intersection(*division*, *intersection*)
Set intersection percentage of intersecting divisions.

class geography.models.**DivisionLevel**(**args*, ***kwargs*)
Level of government or administration at which a division exists.

For example, federal, state, district, county, precinct, municipal.

save(**args*, ***kwargs*)
uid: `{levelcode}`

class geography.models.**IntersectRelationship**(**args*, ***kwargs*)
Each IntersectRelationship instance represents one side of a paired relationship between intersecting divisions.

The intersection field represents the decimal proportion of the to_division that intersects with the from_division. It's useful for apportioning counts between the areas, for example, population statistics from census data.

class geography.models.**Geography**(**args*, ***kwargs*)
The spatial representation (in GeoJSON) of a Division.

4.1.2 entity models

class entity.models.**Person**(**args*, ***kwargs*)
A real human being.

save(**args*, ***kwargs*)
uid: `person:{slug}`

class entity.models.**Jurisdiction**(**args*, ***kwargs*)
A Jurisdiction represents a logical unit of governance, comprised of a collection of legislative bodies, administrative offices or public services.

For example: the United States Federal Government, the Government of the District of Columbia, Columbia Missouri City Government, etc.

save(**args*, ***kwargs*)
uid: `{division.uid}_jurisdiction:{slug}`

class entity.models.**Body**(**args*, ***kwargs*)
A body represents a collection of offices or individuals organized around a common government or public service function.

For example: the U.S. Senate, Florida House of Representatives, Columbia City Council, etc.

Note: Duplicate slugs are allowed on this model to accomodate states, for example:

- florida/senate/
- michigan/senate/

---

save(**args*, ***kwargs*)
uid: `{jurisdiction.uid}_body:{slug}`

class entity.models.**Office**(**args*, ***kwargs*)
An office represents a post, seat or position occuppied by an individual as a result of an election.

For example: Senator, Governor, President, Representative.

In the case of executive positions, like governor or president, the office is tied directlty to a jurisdiction. Otherwise, the office ties to a body tied to a jurisdiction.

Note: Duplicate slugs are allowed on this model to accomodate states, for example:

- florida/house/seat-2/
- michigan/house/seat-2/

---

is_executive()
Is this an executive office?

save(**args*, ***kwargs*)
uid: `{body.uid | jurisdiction.uid}_office:{slug}`

4.1.3 election models

class election.models.**Candidate**(**args*, ***kwargs*)
A person who runs in a race for an office.

get_candidate_election(*election*)
Get a CandidateElection.

get_delegates()
Get all pledged delegates for this candidate.

get_election_delegates(*election*)
Get all pledged delegates for this candidate in an election.

get_election_electoral_votes(*election*)
Get all electoral votes for this candidate in an election.

get_election_votes(*election*)
Get all votes for this candidate in an election.

get_elections()
Get all elections a candidate is in.

save(**args*, ***kwargs*)
uid: `{person.uid}_candidate:{party.uid}-{cycle.ap_code}`

class election.models.**Party**(**args*, ***kwargs*)
A political party.

save(**args*, ***kwargs*)
uid: `party:{apcode}`

class election.models.**ElectionCycle**(*uid*, *slug*, *name*)

save(**args*, ***kwargs*)
uid: `cycle:{year}`

class election.models.**ElectionType**(**args*, ***kwargs*)
e.g., “General”, “Primary”

save(**args*, ***kwargs*)
uid: `electiontype:{name}`

class election.models.**ElectionDay**(**args*, ***kwargs*)
A day on which one or many elections can be held.

save(**args*, ***kwargs*)
uid: `{cycle.uid}_date:{date}`

special_election_datestring()
Formatted date string used in URL for special elections.

class election.models.**Race**(**args*, ***kwargs*)
A race for an office comprised of one or many elections.

save(**args*, ***kwargs*)
uid: `{office.uid}_{cycle.uid}_race`

class election.models.**Election**(**args*, ***kwargs*)
A specific contest in a race held on a specific day.

delete_candidate(*candidate*)
Delete a CandidateElection.

get_candidate_delegates(*candidate*)
Get all pledged delegates for a candidate in this election.

get_candidate_election(*candidate*)
Get CandidateElection for a Candidate in this election.

get_candidate_electoral_votes(*candidate*)
Get all electoral votes for a candidate in this election.

get_candidate_votes(*candidate*)
Get all votes attached to a CandidateElection for a Candidate in this election.

get_candidates()
Get all CandidateElections for this election.

get_candidates_by_party()
Get CandidateElections serialized into an object with party-slug keys.

get_delegates()
Get all pledged delegates for any candidate in this election.

get_electoral_votes()
Get all electoral votes for all candidates in this election.

get_votes()
Get all votes for this election.

save(**args*, ***kwargs*)
uid: `{race.uid}_election:{election_day}-{party}`

update_or_create_candidate(*candidate*, *aggregable=True*, *uncontested=False*)
Create a CandidateElection.

class election.models.**CandidateElection**(**args*, ***kwargs*)
A CandidateEection represents the abstract relationship between a candidate and an election and carries properties like whether the candidate is uncontested or whether we aggregate their vote totals.

class election.models.**BallotMeasure**(**args*, ***kwargs*)
A ballot measure.

save(**args*, ***kwargs*)
uid: `division_cycle_ballotmeasure:{number}`

class election.models.**BallotAnswer**(**args*, ***kwargs*)
An answer to a ballot question.

4.1.4 Vote models

class vote.models.**Votes**(**args*, ***kwargs*)
Popular vote results.

class vote.models.**ElectoralVotes**(**args*, ***kwargs*)
Electoral votes.

class vote.models.**Delegates**(**args*, ***kwargs*)
Pledged delegates.

class vote.models.**APElectionMeta**(**args*, ***kwargs*)
Election information corresponding to AP election night.

class vote.models.**ResultRun**(**args*, ***kwargs*)
A time we hit the AP election API.

class vote.models.**ElexResult**(**args*, ***kwargs*)
Bulk store of AP election API response.

4.1.5 Record identifier

Conventions for creating our `UID` field TK.

4.2 Display models

4.2.1 The Show models

class theshow.models.**PageContent**(**args*, ***kwargs*)
A specific page that content can attach to.

page_location()
Returns the published URL for page.

class theshow.models.**PageType**(**args*, ***kwargs*)
A type of page that content can attach to.

page_location_template()
Returns the published URL template for a page type.

class theshow.models.**PageContentType**(**args*, ***kwargs*)
The kind of content contained in a content block. Used to serialize content blocks.

class theshow.models.**PageContentBlock**(**args*, ***kwargs*)
A block of content for an individual page.

4.3 Other models

4.3.1 Abstract models

Abstract models are used to fill in common fields on other models.

class core.models.**UUIDBase**(**args*, ***kwargs*)
A unique id, self-constructed from a UUID.

class core.models.**UIDBase**(**args*, ***kwargs*)
A unique id conforming to our record identifier conventions.

class core.models.**SlugBase**(**args*, ***kwargs*)
Adds a unique slug.

class core.models.**PrimaryKeySlugBase**(**args*, ***kwargs*)
Adds a primary key slug.

class core.models.**NameBase**(**args*, ***kwargs*)
Adds a name field.

class core.models.**LabelBase**(**args*, ***kwargs*)
Adds a label and short label that can be derived from name.

class core.models.**SelfRelatedBase**(**args*, ***kwargs*)
Adds a self-referrencing foreign key.

class core.models.**EffectiveDateBase**(**args*, ***kwargs*)
Adds effective date fields.

class core.models.**AuditTrackBase**(**args*, ***kwargs*)
Adds auto-generated created and updated fields.

4.3.2 demographic models

Demographic models are used to create serialized data from the U.S. Census Bureau.

See also commands-run-census.

class demographic.models.**CensusTable**(**args*, ***kwargs*)
A census series.

class demographic.models.**CensusVariable**(**args*, ***kwargs*)
Individual variables on census series to pull, e.g., "001E" on ACS table 19001, the total for household income.

class demographic.models.**CensusEstimate**(**args*, ***kwargs*)
Individual census series estimates.

class demographic.models.**CensusLabel**(**args*, ***kwargs*)
Custom labels for census variables that allow us to aggregate variables.

CHAPTER 5

Serialization

Our front-end app is built to consume all the necessary context to interpret AP election results through JSON. We use Django REST to serialize our models to provide that context.

Each page we publish has a matching serialization endpoint. In preview we hit that endpoint directly. During publishing we bake out the endpoint as a `context.json` file that is published with the page alongside a `results.json` created by our daemonized results gathering process.

Note: A second serialization step occurs on the front end when we marry context and results. That data is what is actually fed to our front-end components to display the state of any election. See *Front-end Models*.

5.1 Page serialization

Page serialization matches our *Page structure*. Pages endpoints are always parameterized using an election day.

5.1.1 Cycle pages

Serializers

class theshow.serializers.**ElectionDaySerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_bodies(*obj*)
Bodies with offices up for election on election day.

get_executive_offices(*obj*)
Executive offices up for election on election day.

get_special_elections(*obj*)
States holding a special election on election day.

get_states(*obj*)
States holding a non-special election on election day.

Viewsets

class theshow.viewsets.**ElectionDayList**(***kwargs*)

class theshow.viewsets.**ElectionDayDetail**(***kwargs*)

5.1.2 State pages

Serializers

class theshow.serializers.**StateListSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class theshow.serializers.**StateSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_content(*obj*)
All content for a state's page on an election day.

get_division(*obj*)
Division.

get_elections(*obj*)
All elections in division.

get_parties(*obj*)
All parties.

Viewsets

class theshow.viewsets.**StateMixin**

get_queryset()
Returns a queryset of all states holding a non-special election on a date.

get_serializer_context()
Adds `election_day` to serializer context.

class theshow.viewsets.**StateList**(***kwargs*)

class theshow.viewsets.**StateDetail**(***kwargs*)

5.1.3 Body pages

Serializers

class theshow.serializers.**BodyListSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class theshow.serializers.**BodySerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_content(*obj*)
All content for body's page on an election day.

get_division(*obj*)
Division.

get_elections(*obj*)
All elections held on an election day.

get_parties(*obj*)
All parties.

Viewsets

class theshow.viewsets.**BodyMixin**

get_queryset()
Returns a queryset of all bodies holding an election on a date.

get_serializer_context()
Adds `election_day` to serializer context.

class theshow.viewsets.**BodyList**(***kwargs*)

class theshow.viewsets.**BodyDetail**(***kwargs*)

5.1.4 Office pages

Serializers

class theshow.serializers.**OfficeListSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class theshow.serializers.**OfficeSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_content(*obj*)
All content for office's page on an election day.

get_division(*obj*)
Division.

get_elections(*obj*)
All elections on an election day.

get_parties(*obj*)
All parties.

Viewsets

class theshow.viewsets.**OfficeMixin**

get_queryset()
Returns a queryset of all executive offices holding an election on a date.

get_serializer_context()
Adds `election_day` to serializer context.

class theshow.viewsets.**OfficeList**(***kwargs*)

class theshow.viewsets.**OfficeDetail**(***kwargs*)

5.2 Serializers

class election.serializers.**DivisionSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_level(*obj*)
DivisionLevel slug

class election.serializers.**PersonSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_images(*obj*)
Object of images serialized by tag name.

class election.serializers.**CandidateSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_party(*obj*)
Party AP code.

class election.serializers.**CandidateElectionSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_override_winner(*obj*)
Winner marked in backend.

class election.serializers.**OfficeSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class election.serializers.**APElectionMetaSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class election.serializers.**ElectionSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_candidates(*obj*)
CandidateElections.

get_date(*obj*)
Election date.

get_office(*obj*)
Office candidates are running for.

get_override_votes(*obj*)
Votes entered into backend. Only used if `override_ap_votes = True`.

get_primary_party(*obj*)
If primary, party AP code.

class election.serializers.**PartySerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

class vote.serializers.**VotesSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

get_fipscode(*obj*)
County FIPS code

get_level(*obj*)
DivisionLevel.

get_polid(*obj*)
AP polid minus 'polid' prefix if polid else `None`.

get_polnum(*obj*)
AP polnum minus 'polnum' prefix if polnum else `None`.

get_precinctsreporting(*obj*)
Precincts reporting if vote is top level result else `None`.

get_precinctsreportingpct(*obj*)
Precincts reporting percent if vote is top level result else `None`.

get_precinctstotal(*obj*)
Precincts total if vote is top level result else `None`.

get_raceid(*obj*)
AP election ID.

get_statepostal(*obj*)
State postal abbreviation if county or state else `None`.

CHAPTER 6

Page views

Page views are constructed to support both a live preview and an export view that will be rendered to S3 when publishing an election.

6.1 View classes

6.1.1 Cycle pages

class theshow.views.cycles.**CyclePage**(***kwargs*)
Preview URL: `/election-results/cycle/{YEAR}/`

model
alias of `ElectionCycle`

class theshow.views.cycles.**CyclePageExport**(***kwargs*)
Publish URL: `/election-results/{YEAR}/`

6.1.2 State pages

class theshow.views.states.**StatePage**(***kwargs*)
Preview URL: `/election-results/state/{YEAR}/{STATE}/`

model
alias of `Division`

class theshow.views.states.**StatePageExport**(***kwargs*)
Publish URL: `/election-results/{YEAR}/{STATE}/`

class theshow.views.states.**StateFedPage**(***kwargs*)
Preview URL: `/election-results/state/{YEAR}/{BRANCH}/{STATE}/`

(`BRANCH` is either a congressional body or the office of the presidency.)

class theshow.views.states.**StateFedPageExport**(***kwargs*)
Publish URL: /election-results/{YEAR}/{BRANCH}/{STATE}/

(BRANCH is either a congressional body or the office of the presidency.)

6.1.3 Body pages

class theshow.views.bodies.**FederalBodyPage**(***kwargs*)
Preview URL: /election-results/body/{YEAR}/{BODY}/

model
alias of Body

class theshow.views.bodies.**FederalBodyPageExport**(***kwargs*)
Publish URL: /election-results/{YEAR}/{BODY}/

class theshow.views.bodies.**StateBodyPage**(***kwargs*)
Preview URL: /election-results/body/{YEAR}/{STATE}/{BODY}/

model
alias of Body

class theshow.views.bodies.**StateBodyPageExport**(***kwargs*)
Publish URL: /election-results/{YEAR}/{STATE}/{BODY}/

6.1.4 Race pages

class theshow.views.races.**FederalExecutiveRacePage**(***kwargs*)
Preview URL: /election-results/race/{YEAR}/{OFFICE}/

model
alias of Race

class theshow.views.races.**FederalExecutiveRacePageExport**(***kwargs*)
Publish URL: /election-results/{YEAR}/{OFFICE}/

class theshow.views.races.**StateExecutiveRacePage**(***kwargs*)
Preview URL: /election-results/race/{YEAR}/{STATE}/{OFFICE}/{ELECTION DAY}/

static build_context(*election_datestring*, *state_slug*, *office_slug*, *context={}*)
Build context through a staticmethod so that we can call it without an HTTPRequest when baking to AWS.

model
alias of Race

class theshow.views.races.**StateExecutiveRacePageExport**(***kwargs*)
Publish URL: /election-results/{YEAR}/{STATE}/{OFFICE}/

6.1.5 Special Elections

class theshow.views.specials.**SpecialElectionPage**(***kwargs*)
Preview URL: /election-results/special/{YEAR}/{STATE}/special-election/{MMM}-{DD}/

static build_context(*election_datestring*, *state_slug*, *context={}*)
Build context through a staticmethod so that we can call it without an HTTPRequest when baking to AWS.

cf. utils.bake.election.special_election

model
alias of `Division`

class theshow.views.specials.**SpecialElectionPageExport**(***kwargs*)
Publish URL: `/election-results/{YEAR}/{STATE}/special-election/{MMM}-{DD}/`

6.2 Export Inheritance

To swap out key script references on the server once we bake out pages, each view class generally has a child that inherits from it. For example, if there is a `RacePage` view, there will also be a `RacePageExport` view.

The child view uses a template that also inherits from the parent's template, but overrides any resources that will be baked out.

For example, a parent template may reference a script on the server like this:

```
class RacePage(ClassView):
  template = 'page.html'
```

```
<!-- page.html -->

{%block foot%}
<script src="{% static 'theshow/js/app.js'%}"></script>
{%endblock%}
```

... while the export view's template would override that reference with the script's URL on AWS:

```
class RacePageExport(RacePage):
  template = 'page.export.html'
```

```
<!-- page.export.html -->

{% extends "page.html" %}

{%block foot%}
<script src="http://politico.com/election-results/cdn/{{election_day}}/js/app.js"></
↪script>
{%endblock%}
```

6.3 Static context builders

To support exporting views in a management command, the parent view class has a static `build_context` method that can return a context object. That context object should be enough to render the child template with the `django.template.loader.render_to_string` method when baking to AWS.

CHAPTER 7

Bootstrapping data

These commands help us import data from the Associated Press, U.S. Census and other sources used to build context and other data sources.

If bootstrapping from scratch, they should be run in the order below.

7.1 bootstrap/bootstrap_geography

```
Command.help = 'Downloads and bootstraps geographic data for states and counties from the U
```

7.2 bootstrap/bootstrap_fed

```
Command.help = 'Loads basic structure of the federal governement. Must be run AFTER bootstr
```

7.3 bootstrap/bootstrap_jurisdictions

```
Command.help = 'Loads federal and state jurisdictions. Must be run AFTER thebootstrap_geogr
```

7.4 bootstrap/bootstrap_census

```
Command.help = 'After modeling your desired census tables and estimates in Django, this com
```

7.5 bootstrap/bootstrap_election

```
Command.help = 'Bootstraps election meta models for all elections on an election date.'
```

7.6 bootstrap/bootstrap_content

```
Command.help = 'Bootstraps page content items for pages for all elections on an election da
```

7.7 bootstrap/bootstrap_results_db

```
Command.help = 'Ingests master results JSON file from Elex and updates the results models
```

7.8 bootstrap/bootstrap_results_elex

```
Command.help = 'Creates config files used by our bash process to bake out election results
```

7.9 bootstrap/bootstrap_historical_election

```
Command.help = 'This is some ugly code that bootstraps previous presidential election resul
```

CHAPTER 8

Baking data

These commands help us export data used by.

8.1 bakery/bake_context

```
Command.help = 'Bakes out context for an election.'
```

8.2 bakery/bake_election

```
Command.help = 'Publishes an election!'
```

8.3 bakery/bake_geography

```
Command.help = 'Uploads county-level topojson by state.'
```

8.4 bakery/bake_pages

```
Command.help = 'Bakes pages for an election.'
```

8.5 bakery/bake_statics

```
Command.help = 'Bakes JavaScript and CSS for election date'
```

CHAPTER 9

The Front End

Front-end applications are built in this repo using generator-politico-django in `theshow/staticapp`.

9.1 Stack

- redux
- redux-orm
- preact

9.2 Front-end Models

We let the client resolve the relationships between our slim results data and the context of any campaigns, passing only the keys needed to deserialize `results.json` and `context.json`.

To ensure we're reliably setting and querying relationships between our data, we use a front-end ORM. We also centralize in model methods all the logic needed to resolve any overrides of AP data we set in the backend and pass clean data to all our front-end components.

Model files for the front-end are in `theshow/staticapp/src/js/common/models/` and include models for:

- Division
- Office
- Election
- Party
- Candidate
- AP election status
- AP results

- Override results from our backend

See redux-orm for details on interacting with models.

Each model contains a `serialize()` method except Election, which contains a `serializeResults(divisions)` method that serializes results for an election in the given division(s) and handles all AP overrides. Most dataviz components consume the serialized results returned from an Election model.

9.3 Serialized results

Election's serialize results method returns a data structure that all of our dataviz components use.

Listing 9.1: **serialized results**

```
{
  id: <Election UID>,
  status: {
      called: <APMeta called>,
      tabulated: <APMeta tabulated>,
      overrideApCall: <APMeta override AP call>,
      overrideApVotes: <APMeta override AP votes>,
  },
  office: { /* ... */ },
  divisions: {
    <Division ID>: {
      code: <Division code>,
      codeComponents: { /* ... */ },
      id: <Division ID>,
      label: <Division label>,
      level: <Division level>,
      parent: <Division parent>,
      postalCode: <Division postal code>,
      precinctsReporting: precinctsReporting,
      precinctsReportingPct: precinctsReportingPct,
      precinctsTotal: precinctsTotal,
      results: [
        {
          candidate: { /* ... */ }
          voteCount: voteCount,
          votePct: votePct,
          winner: winner,
        },
        /* ... */
      ],
      shortLabel: <Division short label>
      topojson: { /* ... */}
    }
  }
}
```

9.4 Static files

JS, CSS and images are shared across page types. We bake them out to static paths we can reference absolutely in our baked pages suffixed with a random hash.

```
/election-results/cdn/{election day}/{js|css|images}/{filename}-{hash}.{ext}
```

See commands-bake-statics.

CHAPTER 10

Styles

Style guides for elections.

10.1 Colors

Always use class names to target data-driven colors.

Our convention for color class names is:

```
.{palette}-{length}-{index}-{property}
```

- **palette**

  The name of the palette. For example, *gop* or *dem*.

- **length**

  If applicable, the length of the palette ramp. For example, *8* for an 8-color palette ramp.

- **index**

  If applicable, the index of the color within the palette ramp to use. For example, *1* for the first color in the palette ramp.

- **property**

  The property to target with the specified color. For example, *stroke* or *fill*. Concatenate multipart properties into camel-case, for example, *backgroundColor*.

Some example of fully specified color classes:

- `.gop-4-1-stroke`
- `.gop-fill`
- `.dem-backgroundColor`

CHAPTER 11

Graphics Modules

Candidate results bar

County results map

County swing chart

Demographic vote trend scatterplots

Demographic county maps

CHAPTER 12

Process notes

Notes on processes.

12.1 Starting up

12.1.1 Prerequisites

- Have PostgreSQL installed and a local database called `elections`
- Have python3 installed
- Have virtualenv installed
- Have jq installed (`brew install jq`)
- Have `topojson` and `topojson-simplify` installed globally

12.1.2 Steps

1. Clone this repo to a local directory
2. Start a virtualenv in the directory

```
$ virtualenv -p python3 venv
```

3. Start the virtual environment and source any environment variables.

```
$ source venv/bin/activate
$ source .env
```

4. Bootstrap the database.

```
$ fab data.bootstrap_db
```

5. Move to `theshow/staticapp` directory and yarn install node dependencies.

```
$ yarn
```

6. Run `gulp` to start developing.

```
$ gulp
```

12.2 Preparing Census data

12.2.1 Creating census data

This app contains models and commands to create county-level census data files you can use in your dataviz.

Before you begin, make sure you have these environment variables set:

```
export AWS_S3_PUBLISH_BUCKET="com.politico.interactives.politico.com"
export AWS_ACCESS_KEY_ID="<YOUR ACCESS KEY>"
export AWS_SECRET_ACCESS_KEY="<YOUR SECRET ACCESS KEY>"
```

12.2.2 Steps

1. From the root of the *elections* project, start the development server:

```
$ python manage.py runserver
```

2. Login to the backend admin at `http://localhost:8000/admin`
3. Under Demographic, click to add a new Census Table.
4. Create the table you want using Census table and variable codes. (Use Social Explorer to find the codes.) **Be sure to append an "E" to variable codes to get the census estimate, not the margin of error.** For example, *001E*.
5. Once you've created your table, you're ready to run the census command that will create your tables on the server. Specify states by FIPS codes to create county-level Census data files by state:

```
$ python manage.py run_census 51 34
```

6. You can find your new data at a URL specified with this pattern:

```
https://www.politico.com/interactives/elections/data/us-census/{series code}/
↪{table year}/{state fips}/{table code}.json
```

For example https://www.politico.com/interactives/elections/data/us-census/acs5/2015/34/B03002.json

12.2.3 Current tables used in modules

- `B03002` - Hispanic or Latino Origin by Race
- `B19001` - Household Income in the Past 12 months
- `B17020` - Poverty Status in the Past 12 Months By Age
- `B15003` - Education Attainment for the Population 25 years and Over

12.3 Bootstrapping a server

To bootstrap a new server, log into the AWS console and create a new EC2 server using the AMI for elections.

![AMI dashboard](images/ami.png)

Once the server is bootstrapped and has an IP address, use the pem file in the Private Eye repo to login to the server as such:

```
ssh ubuntu@ip-address-of-server -i path/to/private-eye/politico-east.pem
```

Once you're in the server:

12.3.1 Get Your SSH key on there

You can't rely on your pem file to get into the server; Fabric depends on your SSH key having ssh access to run its commands. So you'll need to copy your SSH key to the server.

1. On your local machine, run `pbcopy < ~/.ssh/id_rsa.pub`
2. SSH into the server with the pem file, as above
3. `nano ~/.ssh/authorized_keys` (or vi if that's your thing)
4. Paste your ssh key on the bottom line
5. Save the file and disconnect from the server (`ctrl+d`).
6. Try to ssh into the server with `ssh ubuntu@ipaddress`

12.3.2 Getting the app on the server

Okay, now you're ready to (attempt to) get the app on the server.

First, double check a few variables in `server_config.py`. `SERVER_PYTHON` should match the version you're running, such as `python3`. `PRODUCTION_SERVERS`/`STAGING_SERVERS` should have the IP addresses of the servers in the lists.

> For the following commands, `staging` represents the server you want to target (either `staging` or `production`) and `master` represents the branch you want to deploy.

Run `fab staging master servers.setup`

Then, install the confs with `fab staging master servers.deploy_confs`.

Check your site and see if it works! If you get an nginx error or it hangs, start checking your logs and welcome to server hell!

12.3.3 Updating the server

To update the server with changes in the repo, first ensure that all of your changes are pushed to Github. Then, run `fab staging master deploy_server`.

12.3.4 Running commands on the server

To run any fab command on the server from your local machine, use `servers.fabcast`. For example, if you want to run the `data.bootstrap_db` command on the server, you would run:

```
fab staging master servers.fabcast:data.bootstrap_db
```

Note that fabcast can only run fabric commands.

12.3.5 Upstart Services

On the server, the app basically functions through two upstart services: `app` and `deploy`. The `app` service starts uWSGI to serve Django through nginx. The `deploy` service runs the results daemon.

In the `confs` folder, you will see the upstart configuration files for those two. Hopefully, you'll never need to touch them.

You can control them through Fabric. There are three commands:

1. `fab staging master servers.start_service:name_of_service`
2. `fab staging master servers.stop_service:name_of_service`
3. `fab staging master servers.restart_service:name_of_service`

12.4 Running results on election night

You can run our results deployment for an election night on either your local computer or on a server. This will walk through both.

12.4.1 Local computer

To run results on your local computer, first ensure you are connected to the correct database and have your AP API key. You can do this by putting the correct environment variables in your `.env` file. The environment variables are:

```
export AP_API_KEY="YOURAPIKEYHERE"
export DATABASE_URL="postgresql://username:password@url:port/elections"
```

Next, check `server_config.py` and ensure the correct global variables are set. These are set per deployment target (production, staging and local). For election nights, make sure they are correct for production. For testing, make sure they are correct for staging and local.

You will want to pay special attention to the following:

- `ELEX_FLAGS`: An array of the flags that elex will run. Consult the elex docs.
- `CURRENT_ELECTION`: The election date we care about

Finally, go into `scripts/results.sh` and make sure the elex command matches the elex flags in your server config (we don't have a good way of matching these yet).

With these variables set, you can bootstrap the AP data. Do that by running `fab data.prepare_races`.

> **Warning:** Do **not** run the Fabric command that would wipe the production database.

Note: For special elections, you will need to go into your Django admin and set the Election `special` boolean to `True`. Then, run `python manage.py bootstrap_results_elex <election-date (YYYY-MM-DD)>` and `python manage.py bootstrap_content <election-date (YYYY-MM-DD)>`.

Context

For results pages, we bake out most things like candidate names, election labels and geography labels to the page in a JSON file for each page. You can bake out that information with the following management command:

```
python manage.py bake_election <election-date (YYYY-MM-DD)>
```

If we are getting results for a new state, you will also need to bake out the geography for those pages. Consult the [geography docs](./geography.md) for how to do that.

Live Results

You can publish live results from your personal computer. First, make sure you have your AP API key exported as an environment variable:

Also, make sure you are connected to the production database as demonstrated above and you have bootstrapped the current election date to the production database.

Then, you can run `fab production daemon.deploy`. This will begin deploying live results to S3.

Once the race is over and AP has finished tabulating results, you can run `python manage.py bootstrap_results_db (YYYY-MM-DD)` to update the database with the AP's tabulated results.

Replaying Tests

The results daemon process will record results automatically. Currently, they are recorded to `/tmp/ap-elex-recordings/<election-date>/national/`. You can check that folder to ensure recording is working if you run *fab staging daemon.deploy* during a live AP test.

To replay a test, run *fab staging daemon.test*, which will loop through the files in this folder and serve them locally and to your deployment target.

12.4.2 Server

TKTK

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index

H

I

J

L

M

N

O

P

R

S

U

V