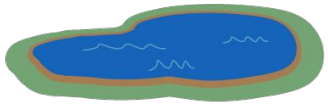
Ashland Daily Photo
http://travelJapanblog.com/ashland/2012/08/portland-oregon-from-the-air/

Data Repository

All data as we receive it stored in one place.



Databases

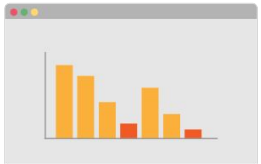
Structured views of the data we receive.

APIs

Web accessible ways for developers to interface with our databases

Web

Public knowledge accessible for all



Django Rest Framework, API Development, and Tooling

geodjango

GDAL

PROJ

django-rest-framework-gis

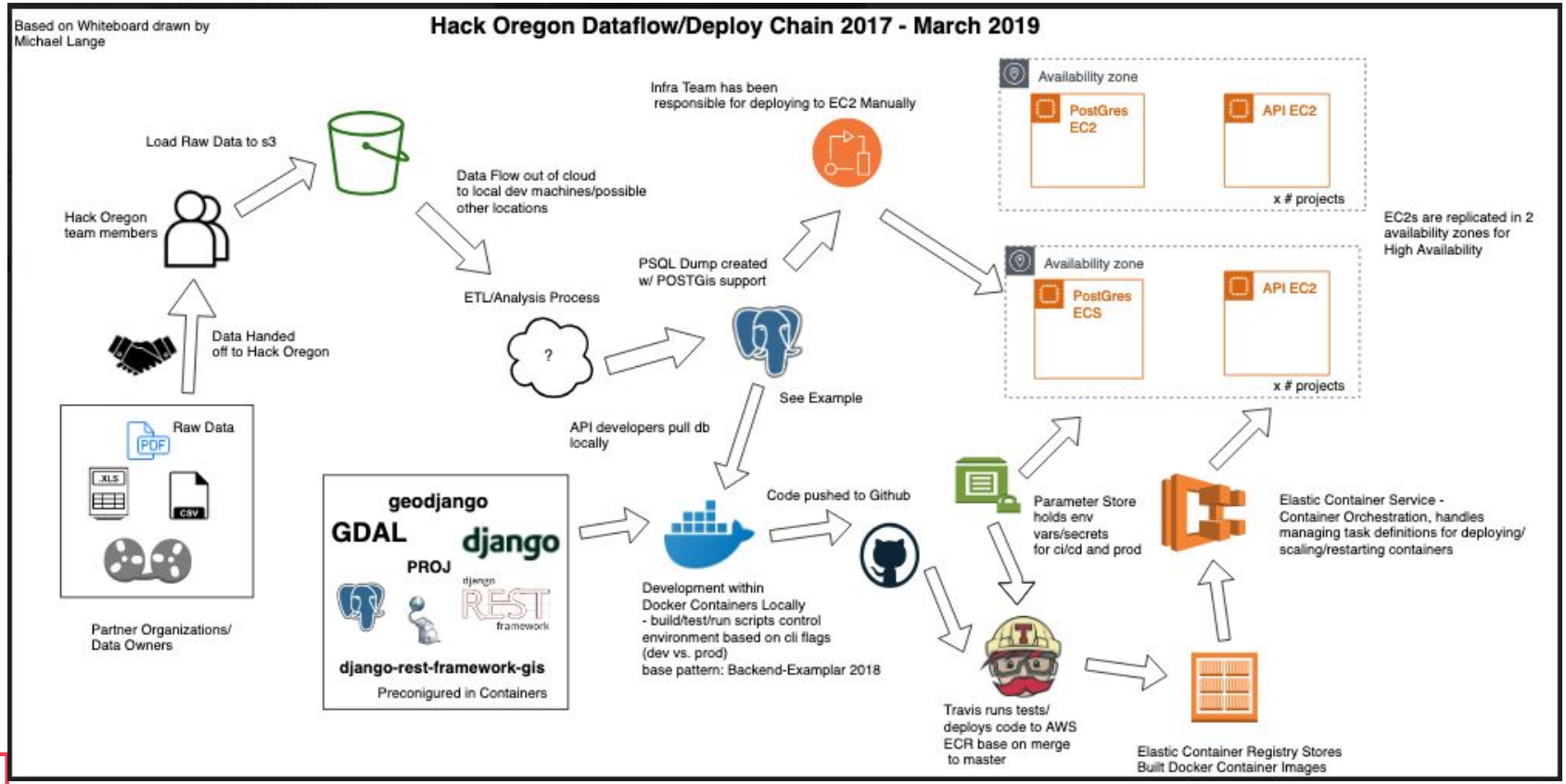
High-level takeaways

- Development pattern/what are we building?
- Workflow overview
- Project creation with cookiecutter
- API development with Django Rest Framework
- GIS Support (GeoDjango, django-rest-framework-gis, PostGIS)
- Tools/Deployment

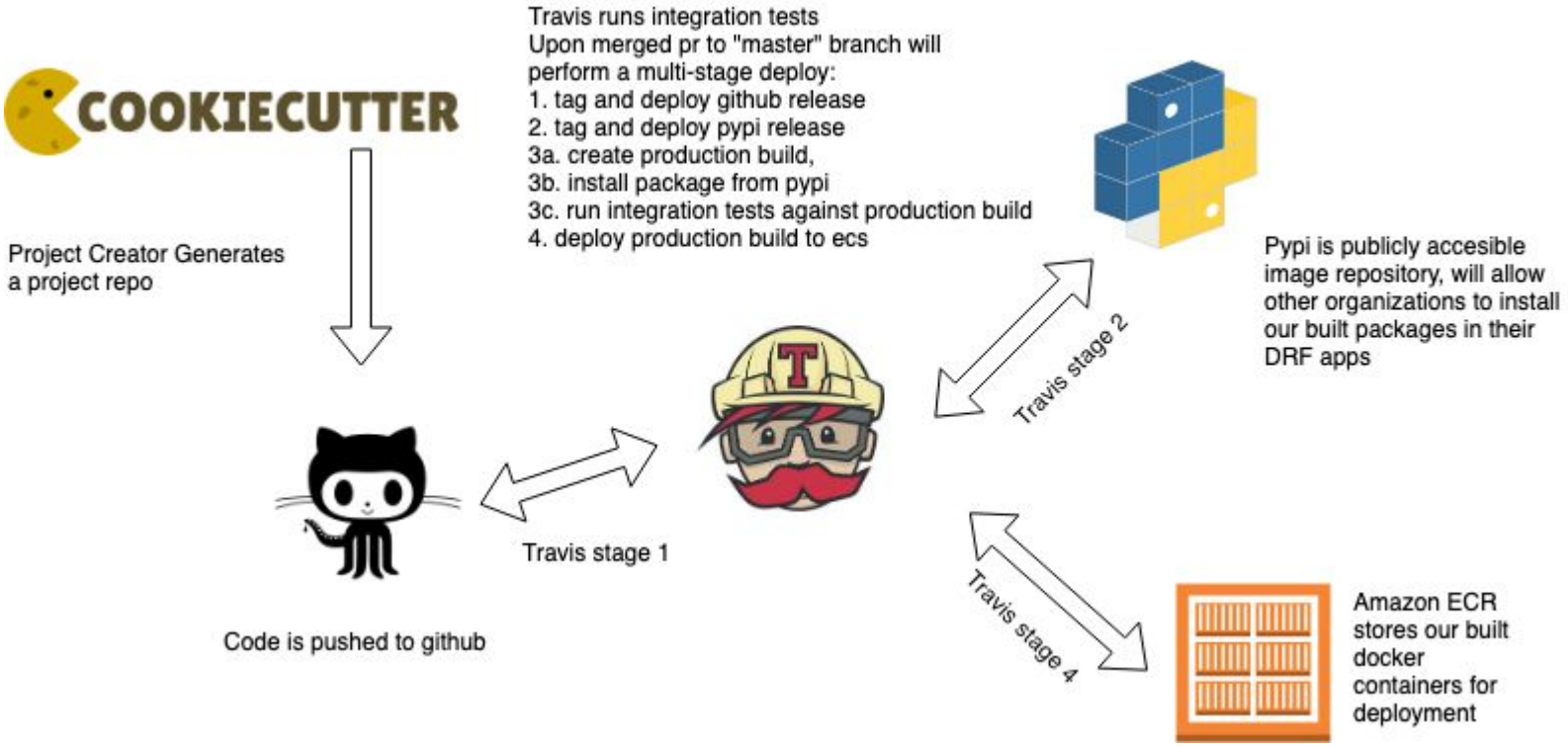
What are we doing?

- Hack Oregon doesn't just build apps, we build systems.
  - APIs provide a public facing gateway to our data
  - Our APIs will power our front end applications as well as future data science
- We work with Django as our web framework
  - "Batteries included model"
- 3rd party Django Rest Framework applications
  - Third Party Packages allow developers to share code that extends the functionality of Django REST framework, in order to support additional use-cases
  - Installable through a package repository

A look at our existing deployment chain ...



API Deployment Quickview



Cookiecutter

- Cookiecutter is a templating engine
- Simplifies our process of building APIs and distributable packages
  - Provides boilerplate and development tooling w/minimal configuration
  - Help developers get up and coding faster

Create a Project

- Install cookiecutter

```
pip install cookiecutter
```

- Use cookiecutter to create repo from github template

```
cookiecutter
gh:hackoregon/2019-backend-cookiecutter-django
```

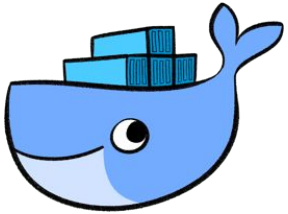
- Follow prompts to fill in templates

- Create git repo

```
cd <cookiecutter-repo-directory>
git init
```

- Create a remote repo in Hack Oregon organization (matching the “github_repo” template value
- Make initial commit

```
git remote add origin <repo url>
git add .
git commit -m "initial commit"
git push origin master
```



- Make new branch

```
git checkout -b
cool-new-feature
```

- Create .env and set

```
cp env.sample .env
```

- Build container

```
./bin/build.sh -d
```

- Start container

```
./bin/start.sh -d
```

- Open url in browser

```
api_1  | [2019-05-18
16:59:44 +0000] [26] [INFO]
Listening at:
http://0.0.0.0:8000 (26)
```

- Start coding

Project Naming Conventions

hack_oregon_team - what project/theme team you are working on

Ex: Transportation Systems

python_package_namespace - this should generally be: hackoregon_< your team>
Ex: hackoregon_transportation_systems

python_subpackage - name of the more specific application/module you are developing. Should generally be able to stand alone as individual/reusable package of code, though can pull in other packages as dependencies

Ex: passenger_census, toad, biketown

*you may have more than one subpackage, but this cookiecutter will spin up first one automagically. A pattern for adding additional subpackages will be provided

Project Naming Conventions

version - you will generally start with 0.1.0 as a first version, with version 1.X being the production/demo day release (we will provide more info on versioning)

gis - whether this project should include django-rest-framework-gis package as a dependency (can be added later manually by updating setup.py)
Ex: True or False

"data_science_repo_url" - URL of repo where team data science work is contained "data_science_repo_name" - Name of repo where data science work is contained

Create a Project (Demo)

- Install cookiecutter

```
pip install cookiecutter
```

- Use cookiecutter to create repo from github template

```
cookiecutter
gh:hackoregon/2019-backend-cookiecutter-django
```

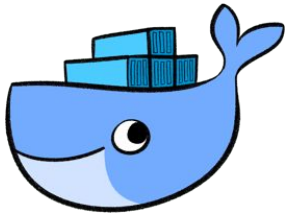
- Follow prompts to fill in templates

- Create git repo

```
cd <cookiecutter-repo-directory>
git init
```

- Create a remote repo in Hack Oregon organization (matching the “github_repo” template value
- Make initial commit

```
git remote add origin <repo url>
git add .
git commit -m "initial commit"
git push origin master
```



- Make new branch

```
git checkout -b
cool-new-feature
```

- Create .env and set

```
cp env.sample .env
```

- Build container

```
./bin/build.sh -d
```

- Start container

```
./bin/start.sh -d
```

- Open url in browser

```
api_1  | [2019-05-18
16:59:44 +0000] [26] [INFO]
Listening at:
http://0.0.0.0:8000 (26)
```

- Start coding

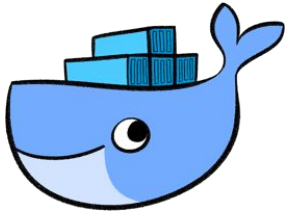
Join a Project

- Clone the repo

```
git clone <github-repo-url>
```

- Make new branch

```
git checkout -b cool-new-feature
```



- Create .env and set

```
cp env.sample .env
```

- Build container

```
./bin/build.sh -d
```

- Start container

```
./bin/start.sh -d
```

- Open url in browser

```
api_1  | [2019-05-18 16:59:44 +0000] [26] [INFO] Listening at: http://0.0.0.0:8000 (26)
```

- Start coding

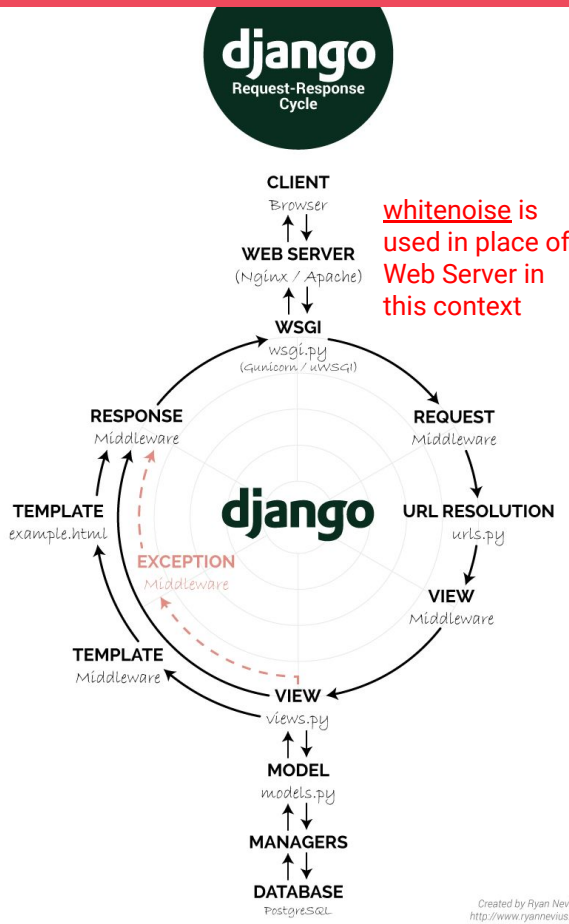
2019-backend-cookiecutter-django

- Let’s look at the template
- Base README, LICENSE, contribution, ignore files…
- Dockerfile and docker-compose
- .travis.yml
- Pytest configuration
- Pypi deployment
- Local_settings and api configuration

Building an API

- Django Rest Framework APIs
  - Configuration
  - Testing
  - Models
  - Serializers
  - Views/Viewsets
  - Routes
- Other Concepts
  - Filtering
  - Pagination
  - Status codes/exceptions

DRF uses serializers to serve json data
Does not use templates directly outside of swagger frontend



whitenoise is used in place of Web Server in this context

Building an API - Configuration

- Django Configuration
  - Configured in the settings.py file
- Docker project provides some basic configuration
  - https://github.com/hackoregon/2019-backend-docker/blob/staging/backend/hacko_settings.py
- You can override in local project
  - https://github.com/hackoregon/2019-backend-cookiecutter-django/blob/fixed/%7B%7Bcookiecutter.github_repo%7D%7D/local_settings/settings.py
- Objects need to be “set” in total, cannot append
  - Important to include all “installed apps” in you local settings
- We are configured to use whitenoise for static hosting/Gunicorn for web server
- “Debug” variable

Building an API - Testing

- Pytest-django
- Coverage
- Live Tests (against production database)
  - Configuration to prevent tear down/change of production data
  - Works locally and in travis
  - https://www.django-rest-framework.org/api-guide/testing/#live-tests
- Test your code, not Django ORM
- APIClient
  - Test api request/response
  - Status codes
  - Logic
- More resiliant code against updates/changes

Building an API - Models

- A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

- The above **`Person`** model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Building an API – Models

- Auto-generation of model from an existing database (run in docker interactive shell)

```
docker exec -it <docker-container> bash
python manage.py inspectdb > <namespace>/<subpackage>/models.py
```

- Gets you part-way there
  - Validate data types
  - Check relationships
  - If using multi-database router:
    - ```
      import django.db.models.options as options
      options.DEFAULT_NAMES = options.DEFAULT_NAMES + ('in_db',)
      ```
  - Add to meta fields:
    - ```
      in_db = 'passenger_census'
      ```

Building an API – Models (Demo)

```
class PassengerCensus(models.Model):
    summary_begin_date = models.DateField(blank=True, null=True)
    route_number = models.IntegerField(blank=True, null=True)
    direction = models.IntegerField(blank=True, null=True)
    service_key = models.TextField(blank=True, null=True)
    stop_seq = models.IntegerField(blank=True, null=True)
    location_id = models.IntegerField(blank=True, null=True)
    public_location_description = models.TextField(blank=True, null=True)
    ons = models.IntegerField(blank=True, null=True)
    offs = models.IntegerField(blank=True, null=True)
    x_coord = models.FloatField(blank=True, null=True)
    y_coord = models.FloatField(blank=True, null=True)
    geom_4326 = models.GeometryField(blank=True, null=True)
    census_block = models.CharField(max_length=255, blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'passenger_census'
        in_db = 'passenger_census'
```

Building an API – Serializers

- Serializer fields handle converting between primitive values and internal datatypes. They also deal with validating input values, as well as retrieving and setting the values from their parent objects.
- Can be very simple:
  - ```
    class AnnualRouteRidershipSerializer(serializers.ModelSerializer):
        class Meta:
            model = AnnualRouteRidership
            fields = '__all__'
    ```
- Use the “ModelSerializer” when working with serializers which meet up with models (or “GeoFeatureModelSerializer”, more on this later)
- Can specify only specific fields (via “fields”), but can also customize/add fields using the serializer fields (important when creating more custom views)
- If serializer is returning more then one object, then set `many=True`
- Relationships generally look at the primary key, but you can provide more control/also set depth


    ```

Building an API – Serializers (Demo)

```
class PassengerCensusSerializer(serializers.GeoFeatureModelSerializer):
    class Meta:
        model = PassengerCensus
        geo_field = 'geom_4326'
        id = 'id'
        fields = '__all__'
```

```
class PassengerCensusInfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = PassengerCensus
        fields = ['summary_begin_date', 'service_key']
        total_routes = IntegerField()
```

Building an API - Views and Viewsets

REST framework provides an `APIView` class, which subclasses Django's `View` class.

`APIView` classes are different from regular `View` classes in the following ways:

- Requests passed to the handler methods will be REST framework's `Request` instances, not Django's `HttpRequest` instances.
- Handler methods may return REST framework's `Response`, instead of Django's `HttpResponse`. The view will manage content negotiation and setting the correct renderer on the response.
- Any `APIException` exceptions will be caught and mediated into appropriate responses.
- Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

https://www.django-rest-framework.org/api-guide/views/

Building an API - Views and Viewsets

- Views in Django can be function based or class based
- Most likely you will stick with class based views (https://wsvincent.com/class-function-based-views/)
- Building off class-based views, you can use Generic Class based views
    - These already include some of the basic functionality needed in a view
    - http://www.cdrf.co/
- Finally, you can combine similar logic into a “viewset”
    - Django REST framework allows you to combine the logic for a set of related views in a single class, called a `ViewSet`. In other frameworks you may also find conceptually similar implementations named something like 'Resources' or 'Controllers'.
- If you only need list/retrieve access to a model, let the ReadOnlyModelViewset handle the work
    - https://www.django-rest-framework.org/api-guide/viewsets/#readonlymodelviewset

Building an API – Views and Viewsets (Demo)

```
class PassengerCensusViewSet(viewsets.ViewSetMixin, generics.ListAPIView):
    """
    This viewset will provide a list of individual Passenger Census counts by TRIMET.
    """

    queryset = PassengerCensus.objects.all()
    filter_backends = (PassengerCensusListFilter,)
    serializer_class = PassengerCensusSerializer
```

Building an API – Views and Viewsets (Demo)

```
class PassengerCensusInfoViewSet(viewsets.ViewSetMixin, generics.ListAPIView):
    """
    This viewset will provide a info about the Passenger Census by TRIMET.

    Returns:
    Distinct Census Dates
    The total number of routes for the census
    The total number of stops for the census
    """
    serializer_class = PassengerCensusInfoSerializer

    def list(self, request, *args, **kwargs):
        census = PassengerCensus.objects.all()
        census = getCensusTotals(census)
        return Response(census)
```

Building an API – Views and Viewsets (Demo)

```
class PassengerCensusInfoViewSet(viewsets.ViewSetMixin, generics.ListAPIView):
    """
    This viewset will provide a info about the Passenger Census by TRIMET.

    Returns:
    Distinct Census Dates
    The total number of routes for the census
    The total number of stops for the census
    """
    serializer_class = PassengerCensusInfoSerializer

    def list(self, request, *args, **kwargs):
        census = PassengerCensus.objects.all()
        census = getCensusTotals(census)
        return Response(census)
```

Building an API - Routing

- Routing will connect url paths with the specific view which should be returned
- Happens in the urls.py file
    - https://www.django-rest-framework.org/api-guide/routers/
- REST framework adds support for automatic URL routing to Django, and provides you with a simple, quick and consistent way of wiring your view logic to a set of URLs.
- https://www.django-rest-framework.org/api-guide/routers/#defaultrouter

Building an API - Routing (Demo)

Building an API - Other Concepts

- Filtering
- Pagination
- Status Codes

Building an API - Filtering

- The default behavior of REST framework's generic list views is to return the entire queryset for a model manager. Often you will want your API to restrict the items that are returned by the queryset.
- We will want to filter based on url query parameters
- Django filter backend is included in our template
  - https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend
  - Provides basic extensions including the “filter_fields” metafield on views
- Can create custom filters
  - https://www.django-rest-framework.org/api-guide/filtering/#custom-generic-filtering
- Other controls can be exposed to the schema through custom filters
  - https://www.django-rest-framework.org/api-guide/filtering/#pagination-schemas

Building an API - Pagination

REST framework includes support for customizable pagination styles. This allows you to modify how large result sets are split into individual pages of data.

The pagination API can support either:

- Pagination links that are provided as part of the content of the response.
- Pagination links that are included in response headers, such as `Content-Range` or `Link`.

The built-in styles currently all use links included as part of the content of the response. This style is more accessible when using the browsable API.

Pagination is only performed automatically if you're using the generic views or viewsets. If you're using a regular `APIView`, you'll need to call into the pagination API yourself to ensure you return a paginated response. See the source code for the `mixins.ListModelMixin` and `generics.GenericAPIView` classes for an example.

https://www.django-rest-framework.org/api-guide/pagination/

Building an API - Status Codes and Exceptions

Using bare status codes in your responses isn't recommended. REST framework includes a set of named constants that you can use to make your code more obvious and readable.

- DRF provides for full gamut of status codes
  - https://www.django-rest-framework.org/api-guide/status-codes/
- Use appropriately
- https://www.django-rest-framework.org/api-guide/exceptions/
- If you build custom views will need to handle expections

Building an API – A second on performance

Django Rest Framework performance relies on similar concepts to Django.

As said in our Postgres Training, “First Get it Right”.

Read this:

- https://docs.djangoproject.com/en/2.2/topics/performance/#
- https://docs.djangoproject.com/en/2.2/topics/db/optimization/#retrieve-everything-at-once-if-you-know-you-will-need-it

Test with jMeter:

- https://jmeter.apache.org/

GIS, GeoDjango, and django-rest-framework-gis

- GeoDjango intends to be a world-class geographic Web framework. Its goal is to make it as easy as possible to build GIS Web applications and harness the power of spatially enabled data.
- Requires some lower level c-libraries
  - GEOS - Geometry Engine Open Source
  - GDAL - Geospatial Data Abstraction Library
  - PROJ.4 - Cartographic Projections library
- PostGIS enabled databases support our backend, provides native support for geo datatypes
- Geodjango can be used independently of database/specific django settings if you need
- GIS support is brought to DRF via the “django-rest-framework-gis” package
  - https://github.com/djangonauts/django-rest-framework-gis
  - GeoFeatureModelSerializer
  - GeoFilters
  - “Bounding box”
  - https://github.com/djangonauts/django-rest-framework-gis#geofeaturemodelserializer

Geospatial DEMO

Versioning

- Github
  - Tagged and released versions
  - https://help.github.com/en/articles/creating-releases
- PyPi (Python Package Index)
  - PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.
  - https://pypi.org/
  - Releases are versioned
  - Think of a requirements file

Python Package Index and Deployment

- PyPi (Python Package Index) - what is it?
- Setup.py
  - Provides commands to create and publish the individual python package
- Developing locally you will do a local import
  - -e ./{{cookiecutter.python_package_namespace}}
- Travis will handle deployment to PyPi
- We are using an implicit namespace to identify our org and team
  - https://www.python.org/dev/peps/pep-0420/
  - Python 3.x convention

Documentation

- Readme
- Swagger/OpenAPI
- Docstrings
- Documentation generation

Workflow tools

-

Travis and deployment

-

More Information

- https://www.django-rest-framework.org/
- http://www.cdrf.co/
- https://www.docker.com/
- https://github.com/djangonauts/django-rest-framework-gis
- https://www.django-rest-framework.org/api-guide/schemas/
- https://swagger.io/docs/specification/about/

Thank you!

Brian Grant