TCP Connections Phase 1 Technical Report

Team Members

Maxx Boehme, Alek Anwar Merani, Scott Enriquez, Paul Glass, Charles Tang

Table of Contents

I. The Problem

There currently is not a consolidated source for information on world crises, making it difficult to understand and follow the relationships among the various people and organizations involved in such crises. In addition, information found throughout the web isn't always kept up-to-date and fails to address their present day impact using social media. Encyclopedic sources like Wikipedia are comprehensive but lack focus, making it difficult for someone specifically looking for *crisis-related* information on a given subject. WCDB also structures information in terms of shared attributes among crises, making it easy to draw comparisons and contextualize the impact of various crises.

II. The Use Cases

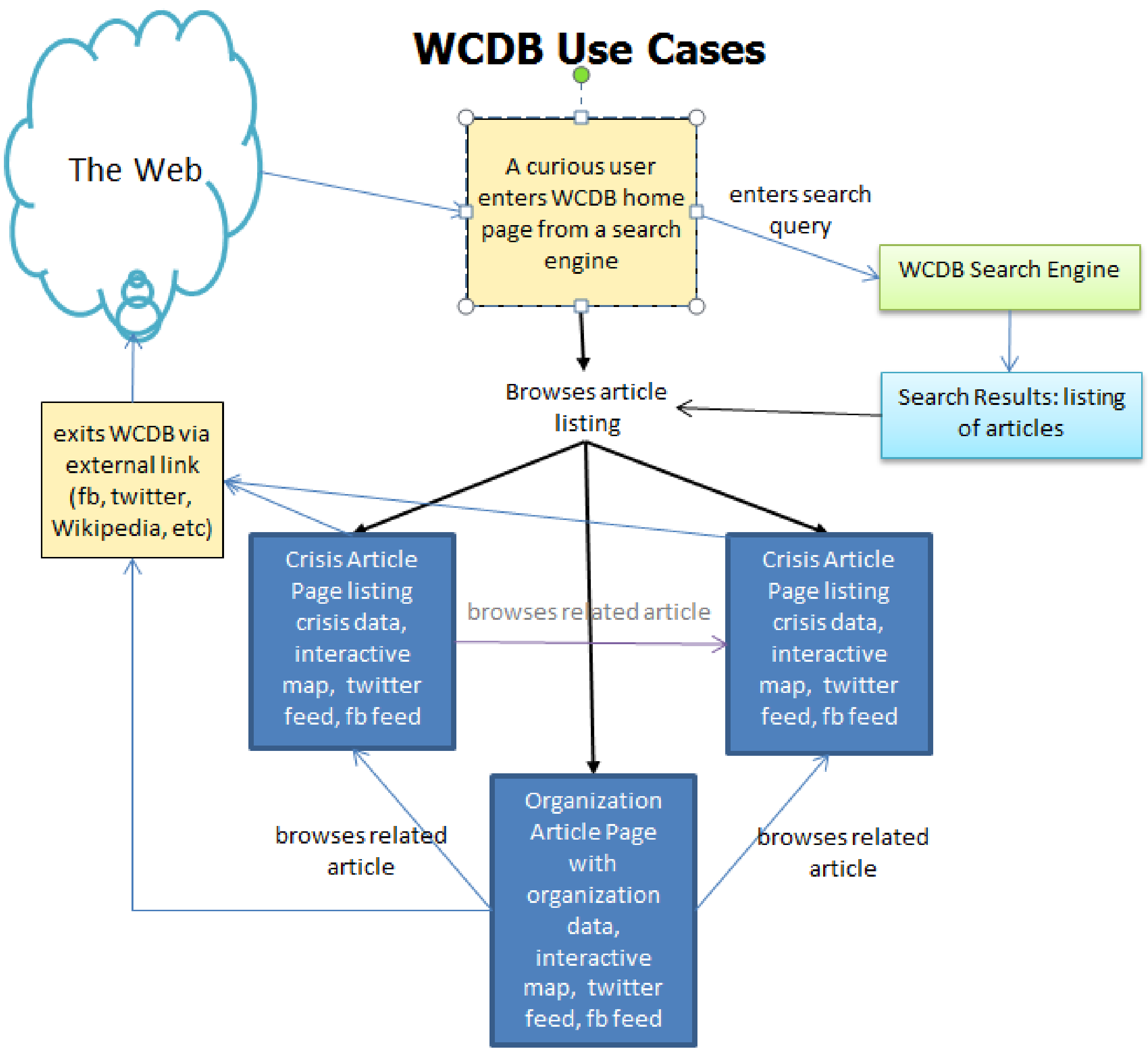

WCDB Use Cases
The Web
A curious user enters WCDB home page from a search engine
enters search query
WCDB Search Engine
Search Results: listing of articles
Browses article listing
exits WCDB via external link (fb, twitter, Wikipedia, etc)
Crisis Article Page listing crisis data, interactive map, twitter feed, fb feed
browses related article
Crisis Article Page listing crisis data, interactive map, twitter feed, fb feed
browses related article
Organization Article Page with organization data, interactive map, twitter feed, fb feed
browses related article

Fig. 1: a flowchart representing the UI interactions for each use case

III. REST API

The REST API is a definition of all of the resources our application offers, including what attributes define each resource and what modifications are allowed to each resource. The API will accept and return JSON representations of our resources. The resources are stored in a database, so the design of the API should keep in mind that all of the data presented is converted to JSON from database tables. Each request is made with a particular HTTP method. The usage of HTTP methods is as follows:

- GET - retrieve a resource
- POST - create a resource
- PUT - update a resource
- DELETE - delete a resource
- HEAD - check that a resource exists, but do not retrieve it

Each response should return a particular HTTP status code. For example, response code 200 indicates a successful request, while 201 indicates the successful creation of a resource.

URLs

The URL of the API needs to be different than the URL of the main application. Our application is located at tcp-connections.herokuapp.com. This will serve HTML pages, like the user expects. The API will reside at *tcp-connections.herokuapp.com*. In the following documentation, “<base-url>” is equivalent to “tcp-connections.herokuapp.com”.

Resource types

Our API defines three different resources types, each of which has a unique identifier:

1. Crisis -- Each crisis has its URL at *<base-url>/crises/{crisis-id}*
2. Organization -- Each organization has its URL at *<base-url>/orgs/{org-id}*
3. Person -- Each person has its URL at *<base-url>/people/{person-id}*

These identifiers are defined to be the primary key of the entity in the database. As such, the user should not have to keep track of which primary keys are used or not. When a new resource is created, a primary key is returned to the user.

Each of these three entities may be associated with an arbitrary number of different resources. A Crisis can be associated with any number of Persons and any number of Organizations. A Person can be associated with any number of Crises and any number of

Organizations. And so forth.

Getting an instance

We can get instances by performing a GET request to the URL identifying the instance. To retrieve a Person with ID 123, perform an HTTP GET request to *<base-url>/people/123*. If successful, this will return status code 200 along with a JSON representation of the data for that particular Person. Retrieving a Crisis or an Organization is handled similarly.

Creating an instance

We can create an instance by performing a POST request to the resource type URL. The POST request should contain JSON containing the attributes that define the instance. To create a Crisis, we make an HTTP POST request to *<base-url>/crises* with JSON that contains the name of the crises, where the crisis occurred, and so forth. A successful response will have status code 201 and will contain the URL with the unique id of the newly-created Crises. Creation of Organizations and Persons is handled similarly. This design prevents the user from having to specify unique IDs upon creation of the instance. Furthermore, associations between different instances cannot be made upon creation.

Updating an instance

We can update an instance by performing a PUT request to the instance URL. To update a Crisis with unique id 123, we make an HTTP PUT request to *<base-url>/crises/123* with the JSON representation of the attributes we want to be updated. The response will have status code 204 (successful, but the response is empty). The Organizations and Persons associated with the Crisis cannot be updated via this method. Unique IDs cannot be updated. This ensures every method is clear about what instances are altered or not altered with each request. Updating Organizations and Persons is similar.

Deleting an instance

Deleting an instance is similar to getting an instance. We make a DELETE request to the URL specifying the instance. To delete an organization with unique id 234, we make an HTTP DELETE request to *<base-url>/orgs/234*. A successful response will be empty and have status code 200. Deleting a Crisis or a Person is similar.

Adding associations

An instance of one type may be associated with any number of instances of different types. A Crisis can be associated with zero or more Persons, for example. In the database, we need to be able to create a Crisis without requiring the foreign key of a Person (since we allow for a Crisis to be associated with zero people), so associations are not made until *after* the creation of the instances. (In the database, we imagine having a table of associations which can be updated independently of the instances themselves.)

To make an association, we use a PUT request. To associate a Crisis with unique id “crisis123” with a Person with unique id “person234”, we make an HTTP PUT request to *<base-url>/crises/crisis123/people/person123*. Equivalently, we could make the same request to *<base-url>/people/person123/crises/crisis123* (that is, you can use the first url or the second, but you do not need to make a request to both urls. Associations are two-way by default.) That is, associated instances appear as a subdirectory of each instance URL. The response will have status code 201, and will contain JSON with the URLs of the updated instances.

Deleting Associations

An association can be deleted in a manner similar to how it is created. We make a DELETE request to the URL representing the association. To remove an association between a Crisis with unique ID “crisis123” and an Organization “org123”, we make an HTTP DELETE request to the URL *<base-url>/crises/crisis123/orgs/org123*. Equivalently, we can use the URL *<base-url>/orgs/org123/crises/crisis123* (that is, use the first URL or the second URL, but you do not need to make a request to both URLs).

IV. Front End Design

Twitter Bootstrap

Twitter Bootstrap is an extremely powerful web development framework that emerged recently and in some ways is revolutionizing web development. The design is very simply and consists folders of CSS and JavaScript files to include at the root of your web server. For someone with little to no web design experience, Twitter Bootstrap is a godsend. Bootstrap itself provides enough functionality to where anyone can use the examples on the Bootstrap site to build their own website. Using only a few examples, we were able to hack out a simple website template in about an hour for our implementation of WCDB.

With this being said, there are some glaring issues with using Bootstrap. First of all is the immense overhead. While the Bootstrap site offers the ability to customize your download, it proved somewhat difficult to strip the Bootstrap code, due to it being so intertwined. We notice significantly longer load times as a result of using Bootstrap for our site. A second issue is the lack of flexibility. Bootstrap effectively makes the develop conform to a 960px-wide screen with 12 slots on each row to create with. Understandably, this design is meant to keep things packaged and responsive for mobile viewing, but on today’s larger desktop screens, 960px is not very much real estate. Any attempts to extend this length are quickly thwarted when items such as menu bars stop functioning. The 12-slot grid layout makes it difficult to create uniquely looking and functioning web pages. Because of this and the myriad of icons and items that Bootstrap provides, it really takes the creativity out of web design. Ultimately, without going to hackish lengths, Bootstrap websites all generally look the same.

Another glaring flaw of Bootstrap is that it is difficult to add the framework to existing code. In many ways, this framework feels immutable and so you must make your design conform to the scheme of Bootstrap instead of Bootstrap conforming to your design. Again, this framework relies heavily on its grid system and size requirements, so any attempts to change these will result in a drastic loss of functionality. This was a major factor in phase two when we were improving our site design and adding our own CSS, which often conflicted with the framework.

As developers, Twitter Bootstrap effectively takes the learning aspect out of web design. Since so much of the code to effectively utilize the myriad of features is widely publicized and easily repeatable, developing with Bootstrap mainly consists of coding by example. Someone who has only used this framework for web development likely does not adequately understand the basic mechanics of web design and best practices.

As far as when to use Bootstrap, it makes sense to use this framework if you know little to no development, do not already have an existing design, and are willing to conform to the uncompromising grid design. If you are proficient in HTML and CSS, you will very likely feel constricted and will very likely have to change the design of your website in order to get the basic functionality out of this framework.

Django

The Django web framework is a variation of the Model-View-Controller (MVC) architectural design pattern to separate the models of the data and the actual interface. The controller is actually provided by the request urls mapped to your views, which then use HTML templates to generate a response. Thus, it’s often called the Model-View-Template (MVT) framework.

Why Use Django?

One reason to use Django instead of other web framework tools is because it is build on Python which is open source meaning it is low cost and can easily be tailored to any platform. Python is a modern architecture with compliant standards and is highly modular. Also being built on standard language, Django is able to take advantage of libraries developed for other purposes, imaging, graphics, scientific calculations and more.

Django is also helpful by already including a build data dictionary, database interface, authoring tools, templates, and data flow, which with a traditional approach would all have to be built manually and would be very time-consuming and costly.

Django Views

These are modules with view functions that accept requests passed by URLs and generate responses by rendering the appropriate templates. The view function for the bare website is the homepage or splash page and is invoked by the root URL.

The view functions for apps are highly specialized. Along with the request, they can accept a context tuple, which is captured by the regular expression that parses the URL. They are then passed to the template as a dictionary where it's replaced by placeholders in the templates, before being passed to the render function.

Django Controllers

Although Django doesn't have an explicit controller module, the URL mappings and the template system serve as this part of the design scheme.

URL Mapping

The urls.py file uses a regular expression to parse the request URL and call the corresponding view function along with any context. The project's urls.py includes urls.py of all the apps within the project.

Templates

The templates folder in the projects consists of all the HTML template pages used by our view functions. The template pages are text documents marked up with Django's template language containing tags, { % this_is_a_tag %} or variables {{ variable }}. These are placeholder and basic logic that determine how the pages are displayed, and the tags are effectively "filled in" when the page is rendered.

For our templates, we created a base.html which contains the components of our website that appear on every page and also tags that are implemented in other specialized pages as needed. We can extend the base.html by simply adding {% extend base.html %} at the beginning of other templates. This removed a great deal of repeated code, such as the static navigation bar. We also implemented certain tags in our home.py and others in crisis_index.html, org_index.html etc.

The URL: *tcp-connections.herokuapp.com/crisis/1/*
is encountered by the urls.py for crises, which invokes the crises_index.html view function for the crises app and passes a context {cid: 1}.

The view function renders the response using the crises_index.html template along with the passed context. Inside of the tags of our *_index.html files are references to the data models that are translated by Django into SQL statements, and the data is then loaded into the page. Once the proper data is selected from our tables, the page is then properly rendered.

Static Files (Phase 1)

In phase 1, the static folder contained all the files that are used in our HTML templates such as our custom CSS stylesheets and JavaScript as well as the Bootstrap files. They can be referenced by /static/<file-path>.

Dynamic Pages (Phase 2)

One of the most significant changes in phase two of the project was replacing the static HTML files with dynamically generated pages. We developed new templates similar to the ones that we originally used to plug in raw data via the static HTML files. Now, however, instead of raw data being stored in manually created files, we used Django's database interface to replace the original nine unique pages with three templates: one for each type of article. This is a much more scalable and sustainable solution than using our original templates alone. In phase two, if we continued our approach, we would have needed thirty unique HTML files.

During the course of the project, we did not write any SQL or make an XML schema. Instead, we combined our Django models, which specified the structure of the tables, with corresponding JSON objects to represent our data and load into the database. The fields and attributes of the JSONs matched with our Django models we created to help input data into the database. We personally found that this powerful combination allowed for an easy blend of web development and data retrieval.

Settings Folder

It’s often the case that you need to develop your Django app in more than one environment and the settings.py file needs to be configured appropraitely. We encountered the need to create separate settings for local development and Heroku because our database configuration are different. Also the Heroku version of our app uses Gunicorn to interact with our Django application.

V. Back End Design

Our Django Structure

The directory of our Django project structures looks as follows:

```
.
├── apiary.apib
├── makefile
├── Procfile
├── README.md
├── requirements.txt
├── wcdb
│   ├── crises
│   │   ├── __init__.py
│   │   ├── models.py
│   │   ├── templatetags
│   │   │   ├── __init__.py
│   │   │   ├── model_extras.py
│   │   ├── tests.py
│   │   ├── urls.py
│   │   ├── views.py
│   ├── fixtures
│   │   ├── crises-citations.json
│   │   ├── crises-data.json
│   │   ├── crises-help.json
│   │   ├── crises-image.json
│   │   ├── crises-links.json
│   │   ├── crises-list.json
│   │   ├── crises-maps.json
│   │   ├── crises-resourses.json
│   │   ├── crises-twitter.json
│   │   ├── crises-videos.json
│   │   ├── OrganizationCitations.json
│   │   ├── OrganizationContactInfo.json
│   │   ├── OrganizationContactInfo.json~
│   │   ├── OrganizationData.json
│   │   ├── OrganizationImages.json
│   │   ├── OrganizationLinks.json
│   │   ├── OrganizationList.json
│   │   ├── OrganizationMaps.json
│   │   ├── OrganizationTwitter.json
│   │   ├── OrganizationVideos.json
│   │   ├── PeopleCitations.json
│   │   ├── PeopleData.json
│   │   ├── PeopleImages.json
│   │   ├── People.json
│   │   ├── PeopleLinks.json
│   │   ├── PeopleMaps.json
│   │   ├── PeopleTwitter.json
│   │   └── PeopleVideos.json
│   ├── __init__.py
│   ├── manage.py
│   ├── mydb.db
│   ├── Procfile
│   ├── requirements.txt
│   ├── settings
│   │   ├── heroku.py
│   │   ├── __init__.py
│   │   ├── localpg.py
│   │   ├── local.py
│   ├── static
│   │   ├── css
│   │   ├── fonts
│   │   └── js
│   ├── templates
│   │   ├── about.html
│   │   ├── base.html
│   │   ├── crises
│   │   │   └── crises_index.html
│   │   ├── crises_index.html
│   │   ├── home.html
│   │   ├── links_page.html
│   │   ├── organizations
│   │   │   └── org_index.html
│   │   ├── org_index.html
│   │   ├── people
│   │   │   └── people_index.html
│   │   └── people_index.html
│   ├── urls.py
│   ├── views.py
│   └── wsgi.py
└── WCDB1.pdf
```

You'll notice that unlike other MVC frameworks there are multiple view files. Since Django was made for rapid development, modularization is given priority. The idea is you have a Django project or website, WCDB for example, and within it you would use various apps that have specialized functions, like plugins in other frameworks. Here it's called *crises*. The root views, URLs, and templates provide the bare essentials to get the website running and then modular apps are added and removed as needed.

Django Models

Relevant file: *models.py*

These are basically modules defined for each app in the project by classes that represent data, which maps to our relational database.

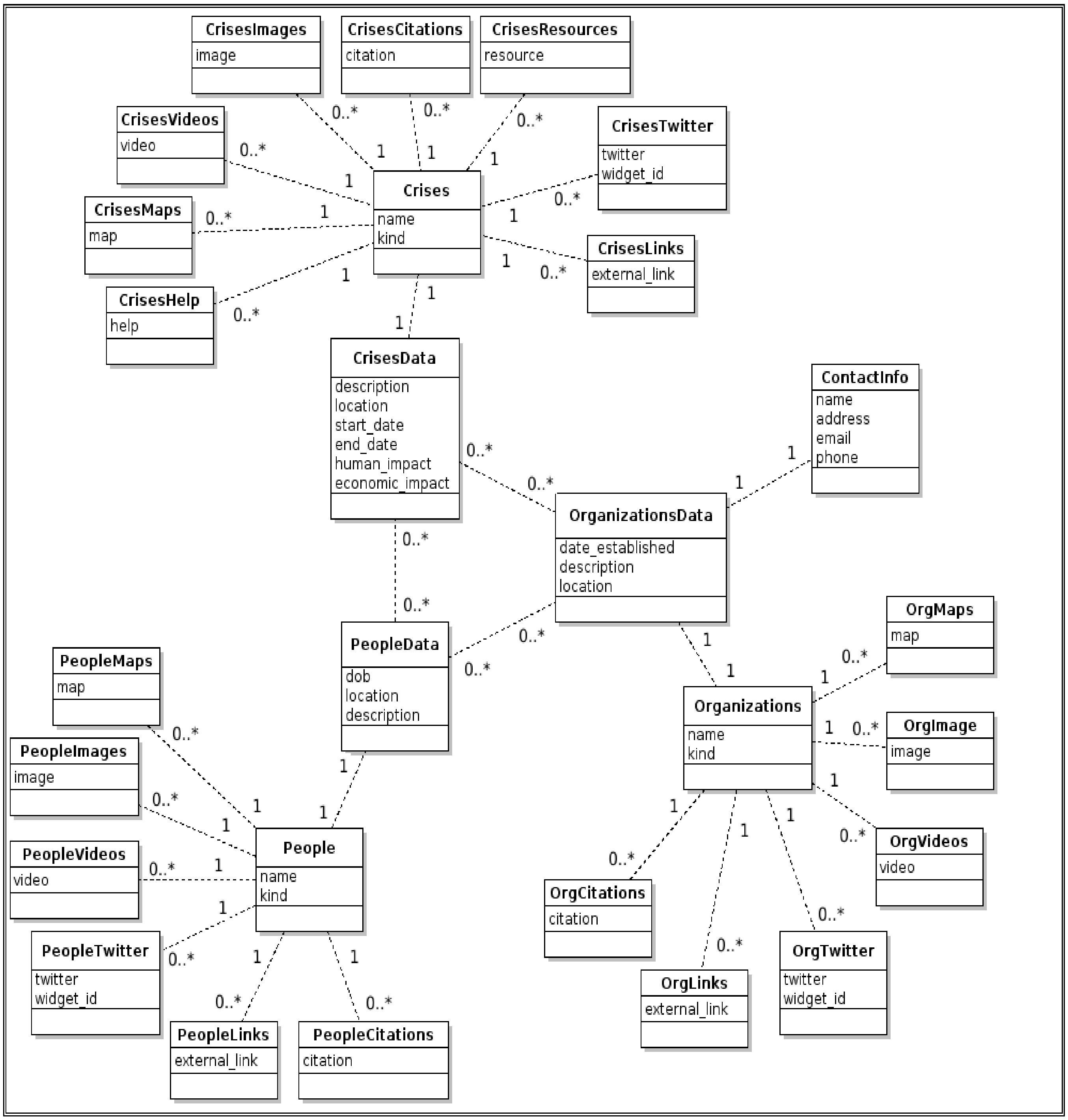
Models contain attributes that will represent a database field. With this Django gives you an automatically-generated database-access API. This provides a simpler interface for creating database objects and requires far less boilerplate code than writing the corresponding SQL from scratch.

Design Decisions

The design of the database is heavily influenced by the REST API. In particular, the identifiers of crises, people, and organizations must be a primary key in the database. Each entity has some associated information -- maps, links, images, and so on -- and there can be zero or more of each of these attributes. Furthermore, each entity can be associated with an arbitrary number of other entities. A Person can be associated with zero or more Crises and zero or more organizations, for instance, which requires many-to-many associations in the database.

Creating these many-to-many associations between People, Organizations, and Crises was made simple by Django. Django allows us to specify which the multiplicity of our foreign keys: one-to-many or many-to-many, for example. Django also takes care creating the junction tables, and providing easy-access from one table to another with its object-oriented database api.

The requirement to support an arbitrary number of maps, images, videos, and so forth means we essentially need to store lists in our database, which is a little awkward. The solution we came up with is verbose, but flexible. We have a table of Crises, Organizations, and People, which serve as the master lists of all our entities. Each crisis is associated with an entry in CrisesData in a one-to-one relationship, and each CrisesData has a zero-to-many relationship with a table for each attribute that may have an arbitrary number of elements. So we have one table for CrisesImages, one table for CrisesCitations, one table for PeopleMaps, and so forth. This results in quite a few tables, gives us the flexibility (over using a single table for multiple attributes) to easily add other information for each attribute. For example, we required an extra *widget_id* attribute for our Twitter widgets to work, and it was easy to add this on to the CrisesTwitter, PeopleTwitter, and OrganizationTwitter without affecting other tables or data. It also avoids -- for instance compared to a single monolithic table for images, videos, and so forth -- having a bunch of null entries in our tables.



A UML diagram of our Django models

Heroku

Heroku is Platform as a Service (PaaS) on top of the AWS cloud. Deploying your website is as easy as pushing to a remote from a git directory. To create a project on Heroku, you run the heroku command ‘heroku create’ from a git directory. It gives a default domain name and the website goes live as soon as you git push your project to the heroku remote.

You need to configure your Heroku environment if you want to serve the website using a high level framework like Django. In our project the following files actually configure the Heroku environment for Django:

Configuring Files

Procfile: This file tells the dyno, which is a virtualized Unix container, what kind of role its going to play. For our project it’s a web server and we also specify that our Django app is going to communicate with the server using a Python WSGI (Web Server Gateway Interface) called gunicorn.

requirements.txt: This file contains all the dependencies of our project like the Django version, gunicorn version and other database versions. It can be generated using the command:

```
pip freeze > requirements.txt
```

wsgi.py: This file basically configures the postgress database on Heroku and also sets an environment variable that tells Django to uses the appropraite Heroku settings.