World Crises Final Technical Report

Team Members

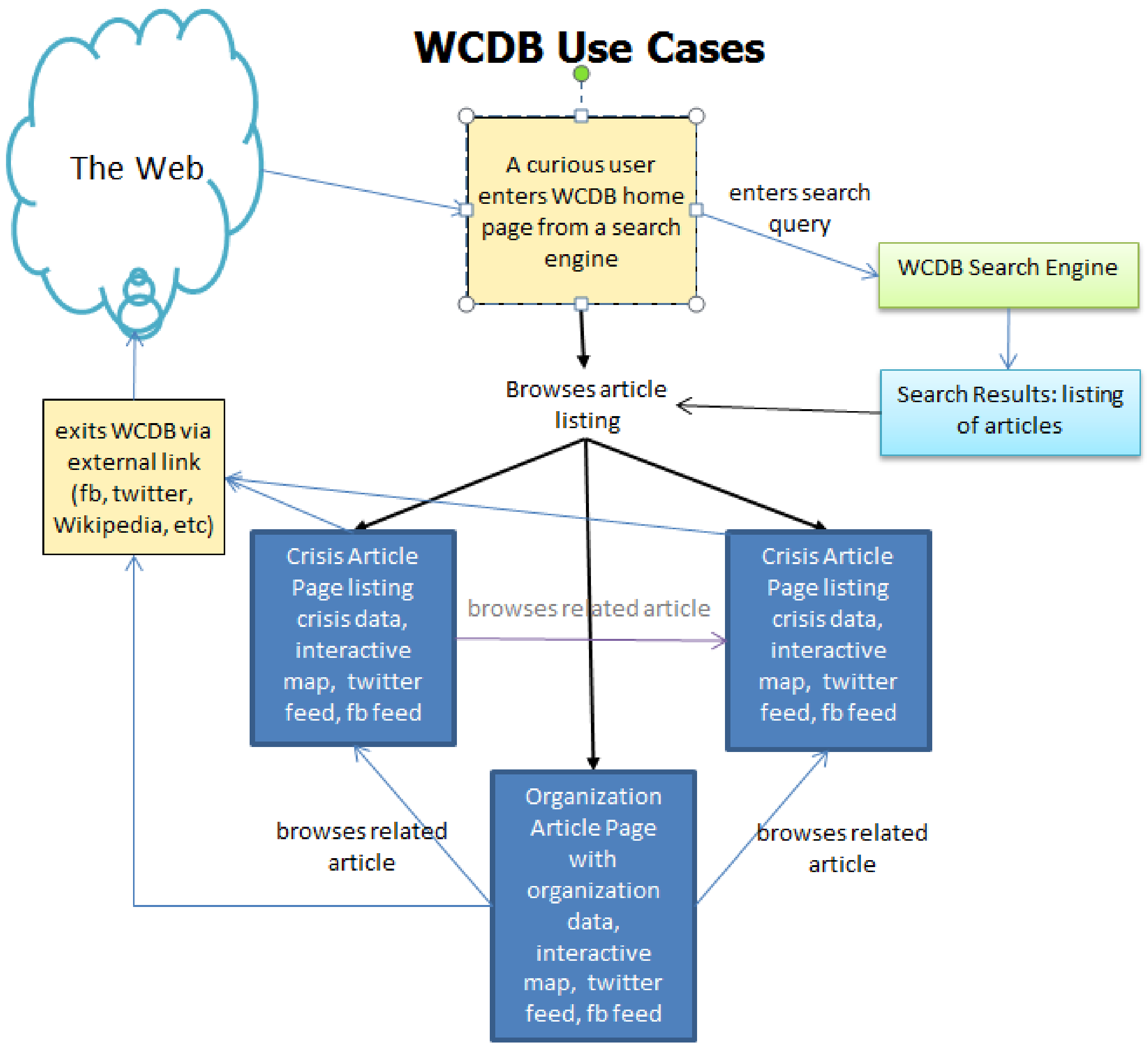
Maxx Boehme, Alek Anwar Merani, Scott Enriquez, Paul Glass, Charles Tang

Table of Contents

1. The Problem

There currently is not a consolidated source for information on world crises, which makes 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 seeking *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. Our implementation also offers a JSON API and search functionality to allow the information we've collected to contribute to other projects.

2. 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
Organization Article Page with organization data, interactive map, twitter feed, fb feed
browses related article
browses related article

A flowchart representing the UI interactions for each use case

3. 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 SQL 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 these methods is as follows:

- GET - retrieves a resource
- POST - creates a resource
- PUT - updates a resource
- DELETE - deletes a resource

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

3.1 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 for the browser, like the user expects. The API resides at *<base-url>/api/*. In the following documentation, “<base-url>” is equivalent to “tcp-connections.herokuapp.com”.

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

These identifiers are defined to be the primary key of the corresponding entity in the database. As such, the user should not have to keep track of which primary keys are used or not, so 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.

3.3 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.

3.4 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>/api/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.

3.5 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>/api/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 handled similarly.

3.6 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 handled similarly.

3.7 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. Associations can be specified in the payload of a PUT or POST request when creating or updating an instance. To associate a crisis with unique ID “123” with a person with unique ID “234”, we can make an HTTP PUT request to *<base-url>/crises* as usual, with “234” in the list of people in the JSON payload. Equivalently, we could make a similar request to *<base-url>/people* with “123” in the list of crises in the JSON payload. Once the association is created, the associated people of the crisis can be listed at *<base-url>/api/crises/123/people,* and the associated crises of the person can be listed at *<base-url>/api/people/234/crises.* All of this works similarly for any other pairing of entity types.

3.8 Deleting Associations

An association can be deleted in the same manner it is created. After an HTTP PUT request is performed, the updated entity will be associated with exactly the entities specified in the JSON payload. So to delete an association, make a PUT request ensuring that the ID of the entity to be disassociated is *excluded* from the JSON payload.

3.9 Deleting Associations

In the underlying implementation, we use Django’s object-oriented database API to fetch or updated the data specified by the request. A Django view receives the request. The view then grabs the data from the database or updates the database with data from the request. The appropriate response data is then packaged and converted to JSON before it is returned along with an appropriate HTTP status code.

The translation from JSON data to database queries and back proved to be a little troublesome. There was some amount of code duplication that was hard to avoid since we had to use the specific names and fields of our Django models. Each resource type has some unique information that has to be handled specifically. We also had to be careful to get the correct JSON field names as well as the correct database field names which was a bit of tedious process. The REST API also took the most work to test, and we found some slight differences using PostreSQL instead of SQLite as the backing database, which produced some additional errors we had to fix.

4. Front End Design

4.1 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 simple 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 first our implementation of WCDB.

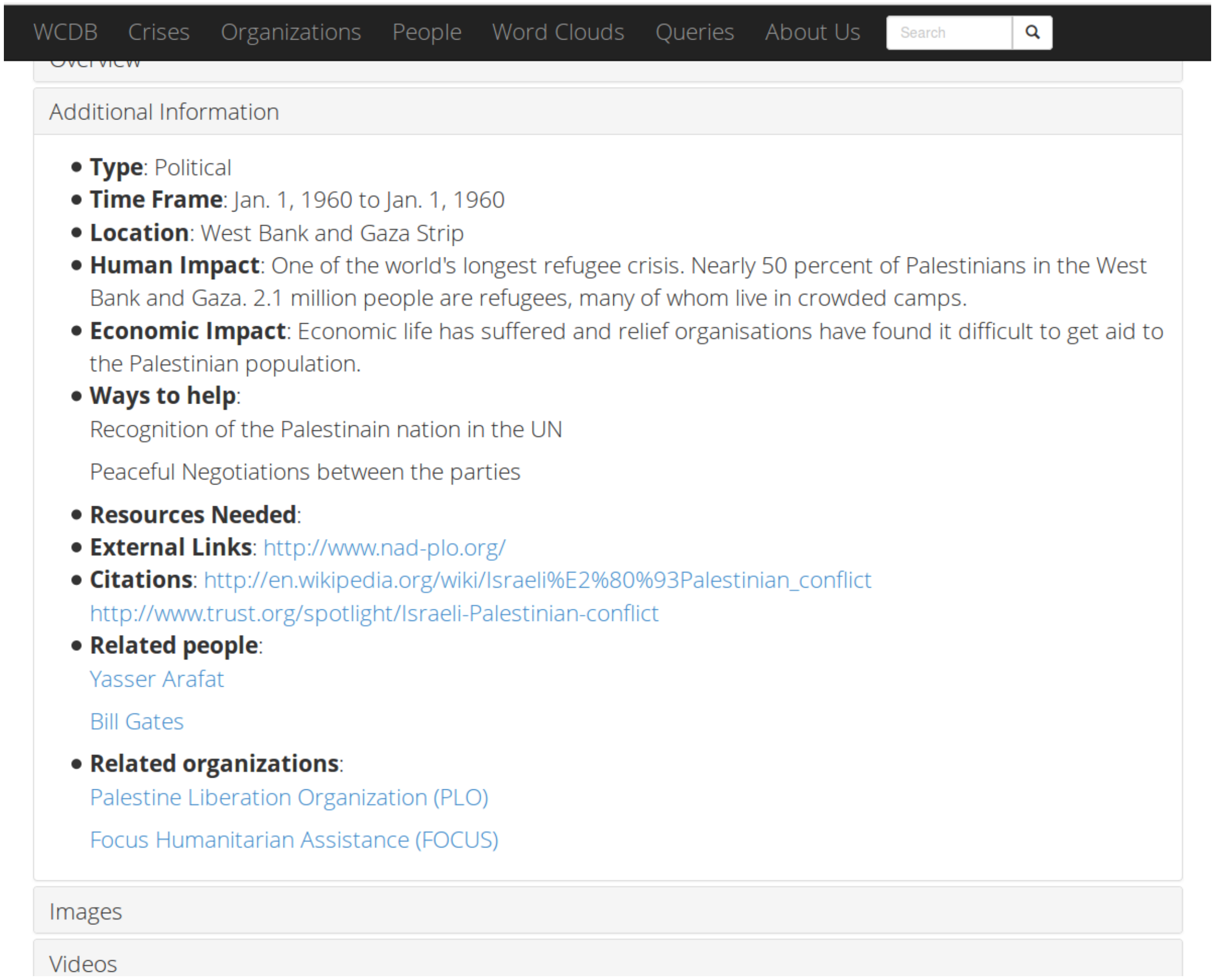
With this being said, there are some glaring issues with using Bootstrap. First of all is the immense overhead. When loading our Heroku application, we dealt with lengthy initial load times before the CSS was cached. While the Bootstrap site offers the ability to customize your download, it proved somewhat difficult to strip the Bootstrap code, due to it being so intertwined. A second issue is the lack of flexibility. Bootstrap 3.0 forces the developer 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 unique 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 have the same layout.

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.

4.2 Page Layout



Additional Information

- **Type**: Political
- **Time Frame**: Jan. 1, 1960 to Jan. 1, 1960
- **Location**: West Bank and Gaza Strip
- **Human Impact**: One of the world's longest refugee crisis. Nearly 50 percent of Palestinians in the West Bank and Gaza. 2.1 million people are refugees, many of whom live in crowded camps.
- **Economic Impact**: Economic life has suffered and relief organisations have found it difficult to get aid to the Palestinian population.
- **Ways to help**:
  Recognition of the Palestinain nation in the UN

  Peaceful Negotiations between the parties
- **Resources Needed**:
- **External Links**: http://www.nad-plo.org/
- **Citations**: http://en.wikipedia.org/wiki/Israeli%E2%80%93Palestinian_conflict
  http://www.trust.org/spotlight/Israeli-Palestinian-conflict
- **Related people**:
  Yasser Arafat

  Bill Gates
- **Related organizations**:
  Palestine Liberation Organization (PLO)

  Focus Humanitarian Assistance (FOCUS)

Images

Videos

An example article from the website

For our article pages, we opted to use an accordion design to give the pages consistency. Our original layout attempted to display all pertinent information on the screen at one time, but proved difficult to maintain with varied length information and sized images. The accordion design not only provides a cleaner design but also allows the user to focus on one section at a time by only showing what is requested, which is particularly beneficial when viewing on a mobile device with little real estate. The static navigation bar at the top of each page allows for easy traversal of the site.

5. 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 the views which then use HTML templates to generate a page. Thus, it’s often called the “Model-View-Template” (MVT) framework.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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, such as the navbar, 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. We also implemented certain tags in our home.html and others in crisis_index.html, org_index.html etc.

The URL: *tcp-connections.herokuapp.com/crisis/1/*
is encountered by the url router (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.

5.6 Static Files

The static folder contains 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>*.

5.7 Dynamic Pages

One of the most significant changes in the second phase 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 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, one for each unique entity.

We did not write any SQL or make an XML schema for our database. 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. This proved to be a very powerful combination allowed for an easy blend of web development and data retrieval.

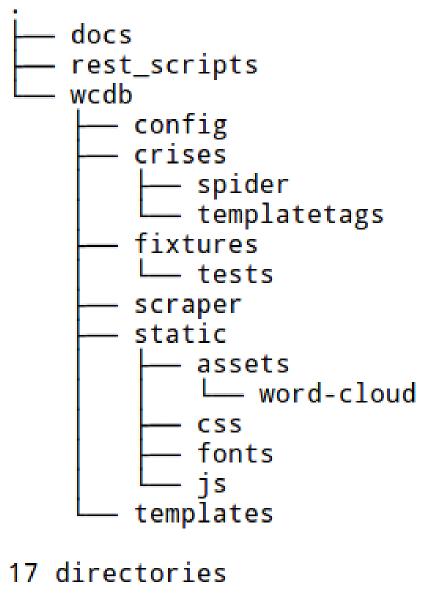
5.8 Settings Folder

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.

6. Back End Design

6.1 Our Django Structure

The directories of our Django project appear as follows:



```
.
├── docs
├── rest_scripts
└── wcdb
    ├── config
    ├── crises
    │   ├── spider
    │   └── templatetags
    ├── fixtures
    │   └── tests
    ├── scraper
    ├── static
    │   ├── assets
    │   │   └── word-cloud
    │   ├── css
    │   ├── fonts
    │   └── js
    └── templates

17 directories
```

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.

6.2 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.

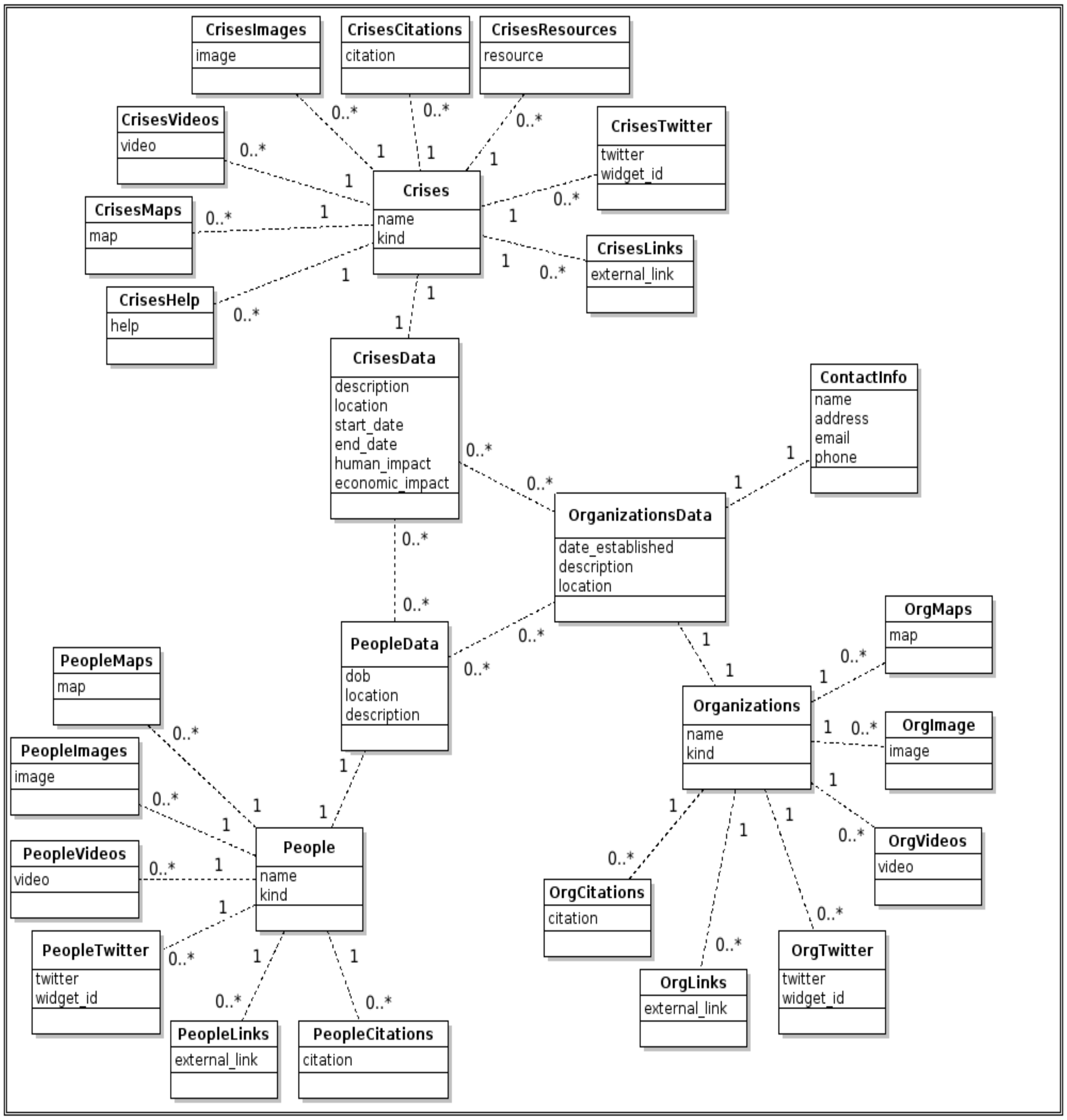
Models contain attributes that will represent a database field. With this Django gives you an automatically-generated database-access API. This provides a simpler interface for creating database objects and requires far less boilerplate code than writing the corresponding SQL from scratch.

6.3 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 handles 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

6.4 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:

6.5 Configuring Files

Procfile: This file tells the dyno, which is a virtualized Unix container, what kind of role it's 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.

7. Testing

Django has nice testing facilities included. Django allows you to specify a fixture for each set of test cases. The fixture, in our case, is a json file used to load a temporary database with data. The tests will then be run against the temporary database, leaving the production database intact. We used this method to test our database tables, ensuring we can create, fetch, and update our database.

Django also has a built-in "client" object, that lets you make requests through

Django's infrastructure. This allowed us to make requests to our REST API locally, and verify correct status codes and correct json responses. Unfortunately though, we found additional failures on Heroku (using PostgreSQL) that did not appear using SQLite. We wrote some further tests to make requests to our REST API on Heroku to verify correct behavior.

Testing the search functionality consisted of testing each function used in the process of searching. In particular, there is a *query(...)* function which takes search terms and returns a list of results. We manually verified the results of certain queries, and then used those queries as test cases, ensuring the correct ordering of the results.

8. Search

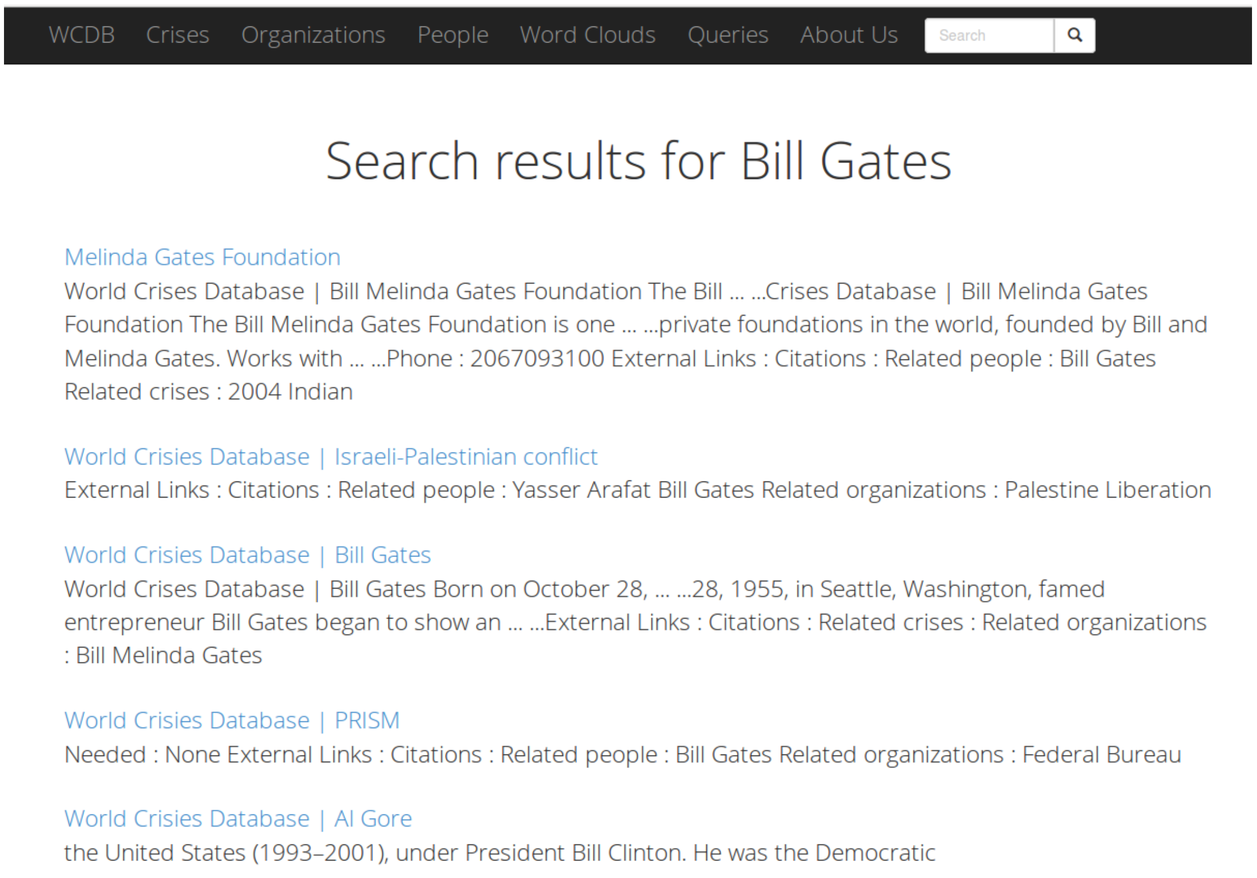
8.1 The Spider and Index

In the final phase of the project, we added search functionality to our application. We approached the problem by creating an index. This index contains a mapping of a word to an object containing a list of the URLs that it appears in, metadata about the article that it's contained in, and the context that surrounds the word. This index is generated by a spider that crawls over the data, parses it, and generates the corresponding mappings. ***For any changes to the database, the spider function must be rerun in order to create an up-to-date index.***

8.2 Query and Ranking

The search is executed when the page */search/{query}* is requested. The query words are delimited by the character +. For example to search for Bill Gates, the proper URL is /search/Bill+Gates. There is also a search bar anchored on every page that generates the proper search URL for the user.

The query function is invoked on the end of the URL and parsed by regular expressions. Each of the tokens is looked up in the index, and each matching URL is counted. This is the basis of our result ranking system. URLs with more matching words are displayed before URLs with less matching words.



Search results for Bill Gates

Melinda Gates Foundation
World Crises Database | Bill Melinda Gates Foundation The Bill ... ...Crises Database | Bill Melinda Gates Foundation The Bill Melinda Gates Foundation is one ... ...private foundations in the world, founded by Bill and Melinda Gates. Works with ... ...Phone : 2067093100 External Links : Citations : Related people : Bill Gates Related crises : 2004 Indian

World Crisies Database | Israeli-Palestinian conflict
External Links : Citations : Related people : Yasser Arafat Bill Gates Related organizations : Palestine Liberation

World Crisies Database | Bill Gates
World Crises Database | Bill Gates Born on October 28, ...28, 1955, in Seattle, Washington, famed entrepreneur Bill Gates began to show an ... ...External Links : Citations : Related crises : Related organizations : Bill Melinda Gates

World Crisies Database | PRISM
Needed : None External Links : Citations : Related people : Bill Gates Related organizations : Federal Bureau

World Crisies Database | Al Gore
the United States (1993–2001), under President Bill Clinton. He was the Democratic

The search page displaying ranked, hyperlinked results and contexts

9. SQL Queries

In the final phase of the project, we implemented 5 unique queries against our database, as well as 5 non-unique queries from other groups, many of which were rather complex. In order to implement these complex queries, we had to write raw SQL, rather than use the Django models to abstract away that layer of complexity. Though SQL is relatively verbose compared to using Python to interact with Django models, the SQL gave us the flexibility we needed for this situation.

For example the query select most related (most related people/crises) organizations required many subqueries and groupings that could not have been done using Python and models. One subquery was needed to count up all the people related to an organization. The result was a table with each organization ID paired with the number of related people. Another subquery is needed to count up the crises related to each organization to get a table with each organization ID paired with the sum of all related crises. Then from these subqueries we made another subquery to add these two into one table that contains the organization ID and total number of relations. From this we calculate the the maximum number of relations and output all organizations with the same number of relations. Python models just don’t have the flexibility to be able to do all of this or at least in a way that is as simple to understand as stated above.

In order to bypass the Django models, we used a connection.cursor() object to execute SQL directly against the database and fetch the results out to be displayed on each of our query pages.

Our Django models abstracted away several tables (namely the many-to-many association tables), so we had to determine the actual names of the tables our Django models generated in order to query them directly. To do this, we had to use the following command: *python manage.py crises sqlall --settings=config.local*

While implementing certain queries involving comparisons, we came across an issue with the “all” SQL we saw in class. For some reason the keyword was not being recognized, so we had to resort to using the min() and max() functions, which was functionally equivalent for our purposes.

To test the SQL queries, we initially tried to display the rows in a view, but that proved cumbersome, so we ended up writing a set of tests in sqltests.py that would run our queries in the shell and print the results to console using the following command: *python manage.py shell --settings=config.local < sqltests.py*

When uploading the queries to heroku we ran into some problems because when running locally our database uses SQLite while heroku uses PostgreSQL. One cause of our problems was that SQLite does not make you name your subqueries while PostgreSQL does. This meant going and adding the key word AS to all of our subqueries. Another thing we notice was that SQLite would allow the use of WHERE instead of ON when computing an INNER JOIN while PostgreSQL required the use of ON.

10. Word Cloud

One of the advantages of having a RESTful API, which exposes the data of your database, is that it’s very easy to quickly retrieve the data and visualize it. Most people can understand and appreciate data presented in an appealing way much faster than by just looking at raw blocks of data in tables. For phase 3 of our project we decided to create word clouds from the article summaries of crises, people and organizations on our website. Word cloud is a visualization tool that allows a user to see the most frequent words that appear in our database. The final visualization consists of all the words sized proportionally to their usage in the article and then displayed in an artistic way with different colors and orientations to fit in a predetermined sized “cloud”. We created a word cloud for every single article on our website, then one for each of the categories of articles namely crises, organizations and people and finally one for all the articles in our database.

10.1 How it is generated?

The word clouds are generated by analyzing the word usage in each document and then a “scalable vector graphics (SVG)” is generated using the calculated frequencies of the words passed to a javascript library called D3. SVG is an image format that allows interactivity and animation. Our word clouds are generated dynamically by using random colors and orientations from the jsons generated by the scraper.

10.2 Scraper

We wrote a python script that makes API calls via GET requests to retrieve the JSONs that contains the description field for the articles on our website. Before calculating the word frequencies we have to process the descriptions by tokenizing them and removing any stop words, which are basically common English words like the, and, of etc. These stop words are stored in a csv format so that more can be added if necessary.

The JSON is represented as {“text”:”word”, “size”:30.0} where the text field represents the word that is displayed in the visualization and the size represents the font size of the word. The font size is calculated as the term frequency / total number of words in the cloud. Thus if a word is used twice as much as another then the it will appear twice as big in the visualization. This JSON is read by using jQuery on the client side in the browser and then the word cloud is generated using the D3 library.

10.3 D3 Library

D3 allows DOM manipulation of our HTML documents by dynamically inserting an SVG element in the specified component of our page. The script can be found in wordcloud_index.html which uses the draw method from the D3 library to generate a, 1000 word by 500 word, word cloud and also randomizes the orientations of the words between 0 to 90 degrees. A feature we wish to add is where the user can input the desired size of the cloud and also the orientations of the words.

This is an example of the end result:

radioactive
material
nuclear
Soviet
accident
reactor
power
additional
near
crews
sand
boron
unit
fire
1986
Chernobyl
International
Atomic
Energy
Agency
Vienna
Austria
Ukraine
helicopters
"sarcophagus"
government
contamination
Emergency