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 isn't 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

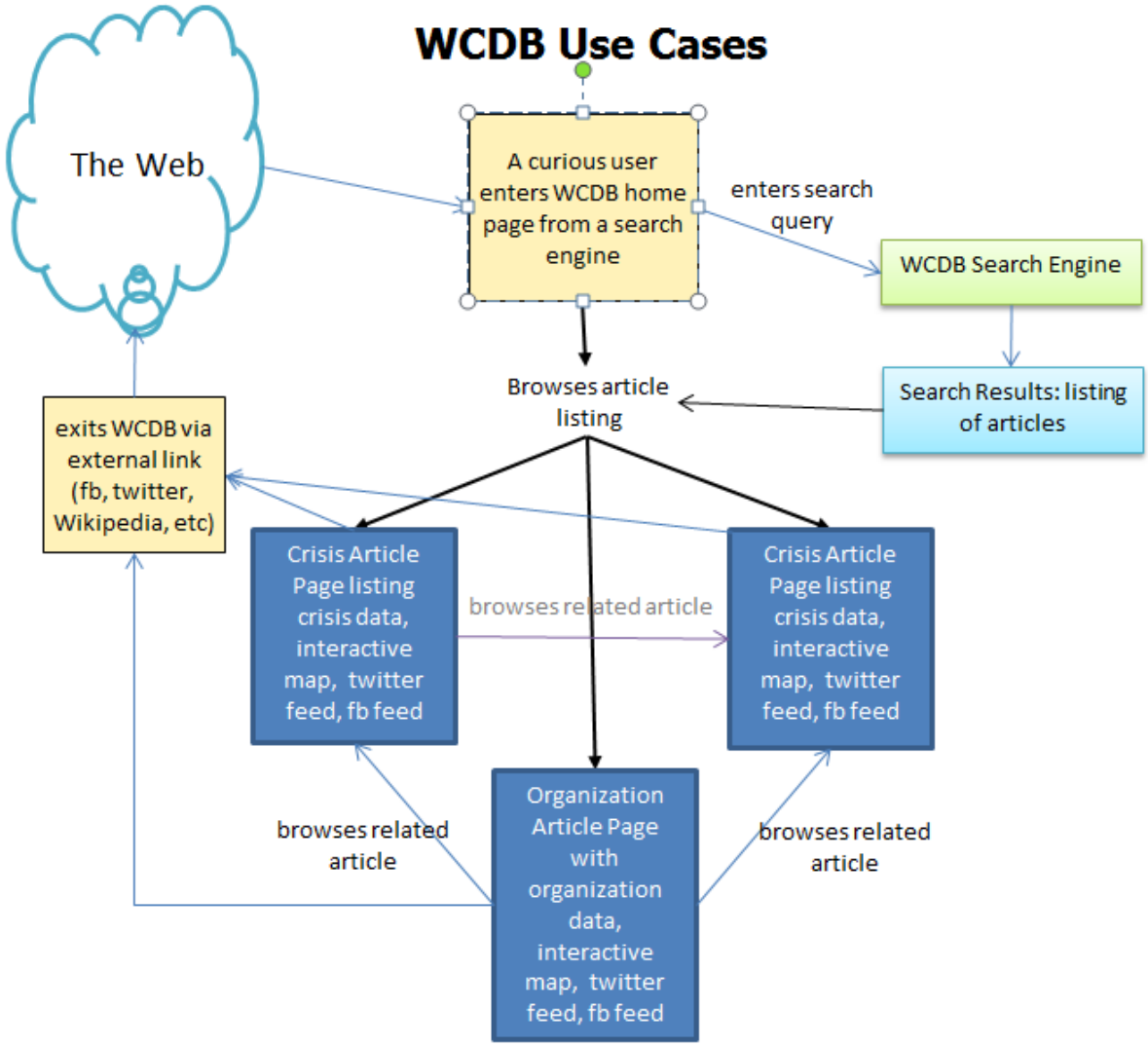

Fig. 1: a flowchart representing the UI interactions for each use case

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 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 instance of a type has its own 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}*

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.

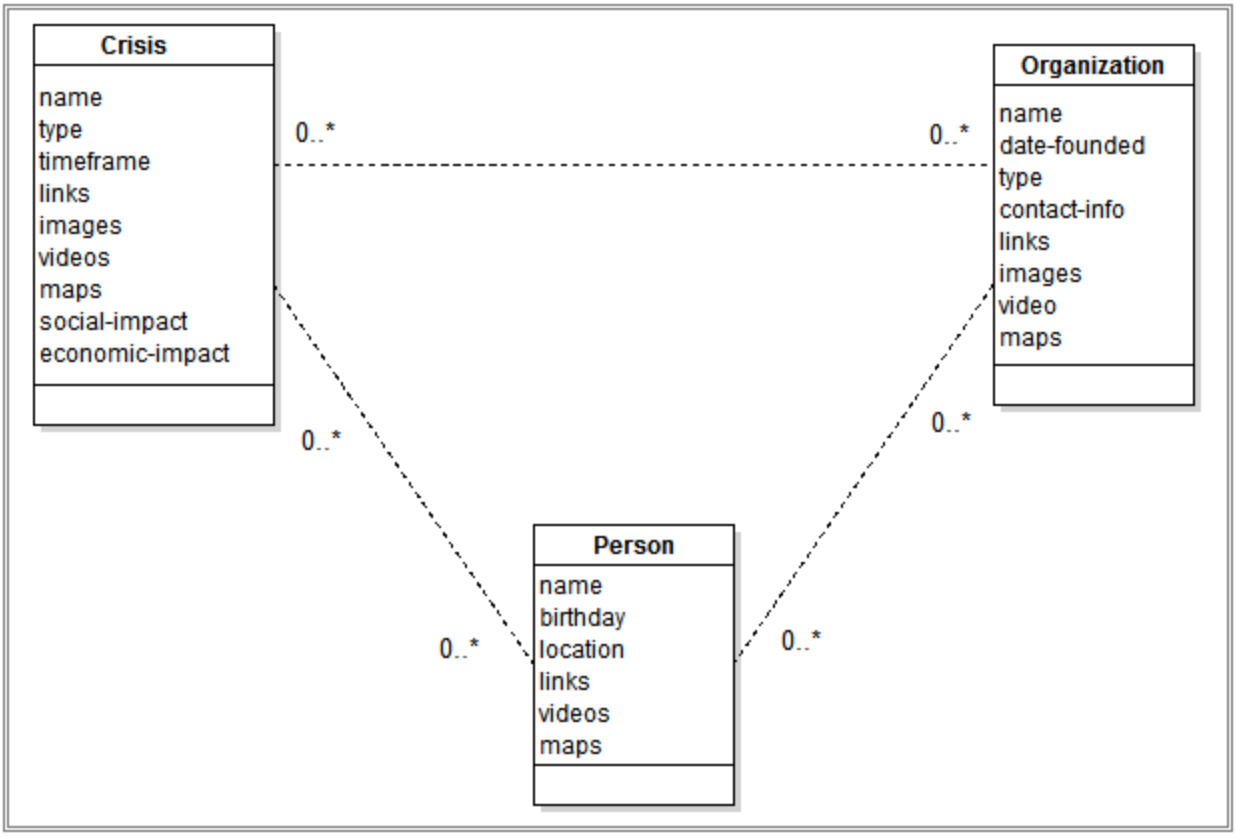

Fig. 2: A basic diagram of resource attributes and relationships

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 that particular Person. Retrieving a Crisis or an Organization is similar.

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 similar. 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 with 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) 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 simply Google code examples and plug in their own data. Using only a few examples, we were able to hack out a simple website template in about an hour for 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. 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.

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, 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 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 and do not already have an existing design and are willing to conform to the 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 full functionality out of this framework.

Django

The Django web framework is a variation of the Model-View-Controller (MVC) architectural 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, data base interface, authoring tools, templates, and data flow which with a traditional approach would all have to be built manually.

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 basically the home/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 regex that parse the url. They are then passed to the template as a dictionary where it's replaced by placeholders in the templates, before being rendered.

Django Controllers

Although Django doesn't have a real controller module, the url mappings and the template system handle this part of the architecture.

URL Mapping

The urls.py file uses regex to parse the request URL and call the corresponding view

function along with any context. The project 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.

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. We implement certain tags in out home.py and others in crisis.html, org.html etc.

The url: *tcp-connections.herokuapp.com/crisis/1/*
is encountered by the urls.py for crises, which invokes the crises_index 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. The crises_index.html uses this context and includes another template dynamically using {% include crisis %} where crisis is "crisis"+"cid"+".html". This was only done to demonstrate Django's capability of using contexts for generating dynamic content. Finally the corresponding page is rendered.

Static Files

The static folder contains all the files that are used in our html templates such as css stylesheets, javascripts and even images and videos can be stored here and referenced by /static/<file-path>.

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 need to be configured appropraitely. We encountered the need to create separate settings for local development and Heroku because our database configuration were different and also the Heroku uses gunicorn to interact with our Django app.

V. Back End Design

Our Django Structure

The directory of our Django project structures looks as follows:

```
.
├── apidoc.apib
├── Procfile
├── README.md
├── requirements.txt
└── wcdb
    ├── crises
    │   ├── __init__.py
    │   ├── models.py
    │   ├── tests.py
    │   ├── urls.py
    │   └── views.py
    ├── __init__.py
    ├── manage.py
    ├── Procfile
    ├── requirements.txt
    ├── settings
    │   ├── heroku.py
    │   ├── __init__.py
    │   └── local.py
    ├── static
    │   ├── css
    │   │   ├── bootstrap.css
    │   │   ├── full.css
    │   │   ├── style.css
    │   │   └── wcdb-style.css
    │   ├── fonts
    │   │   ├── glyphicons-halflings-regular.eot
    │   │   ├── glyphicons-halflings-regular.svg
    │   │   ├── glyphicons-halflings-regular.ttf
    │   │   └── glyphicons-halflings-regular.woff
    │   └── js
    │       ├── bootstrap.js
    │       └── jquery.js
    ├── templates
    │   ├── base.html
    │   ├── crises
    │   │   ├── crises_index.html
    │   │   ├── crisis1.html
    │   │   ├── crisis2.html
    │   │   └── crisis3.html
    │   ├── home.html
    │   ├── organizations
    │   │   ├── organization1.html
    │   │   ├── organization2.html
    │   │   ├── organization3.html
    │   │   └── org_index.html
    │   └── people
    │       ├── people_index.html
    │       ├── person1.html
    │       ├── person2.html
    │       └── person3.html
    ├── urls.py
    ├── views.py
    └── wsgi.py
```

You'll notice that unlike other MVC frameworks there are multiple views files. Since Django was made for rapid development, modularization is given priority. The idea is you have a Django project/website, which is wcdb here and within it you would 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

It was a natural choice to create a separate model for each of our article types: Crises, Organizations, and People. Each model enumerates the name and type of the information we expect to display on each page. In addition, we created a "WebsiteReference" model to encapsulate the two pieces of information needed for a hyperlink: the URL itself as well as the text displayed on the link.

Creating many-to-many associations between people, organizations, and crises proved somewhat tricky. Simply defining those naively in models.py resulted in a circular dependency problem. For example, People could be defined as having links to Organizations, which in turn would have links to various People. Our solution was to manually create 3 separate junction tables for each of the 3 types of associations, People to Organization, Organization to Crisis, and Crisis to People, each containing the appropriate foreign keys. We assume that associations are all symmetric, hence the need

for only 3 such tables, not 6. We were able to avoid this with our recursive many-to-many associations (i.e., People to People) by using the ‘self’ keyword.

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.