Technical Report for WCDB Phase III

Team CorgiShmorgishborg

November 21, 2013

Contents

Chapter 1

Our Group

Our group name is *CorgiShmorgishborg*. And our group members are:

- Ashley Ng
- Clay Smalley
- Kasey Reed
- Patrick Fennelly
- Yutian Li

Figure 1.1: Not our Corgi

Chapter 2

Introduction

2.1 Problem

We intend to gather information of events of world crises, and make accessing them easier than ever. We hope by doing this, we can raise awareness and attract attention around people. It offers a new approach of spreading help, love and humanity.

2.2 Use Cases

We link crises with people and organizations. It makes finding relevant information about a specific crisis, or some crises fitting a condition easier.

So people can use our system to raise awareness and attract attention, and use this system to analyze new ways to help alleviate the impact of crises.

Chapter 3

Design

We use Django as our backend server. It handles requests of JSON objects, database access, and one and only one template, i.e., the overall framework. View manipulation, templates, and redirection are handled by AngularJS, which is conventionnaly done on server side. For our page design, we used Bootstrap.

3.1 RESTful API

The design for our RESTful API was centered around what kind of data a user would of a World Crisis Database wish to see. We have three main categories:

- Crises
- People
- Organizations

Each of these categories have several API endpoints associated with them corresponding to the various HTTP request methods, namely, `GET`, `POST`, `PUT`, and `DELETE`. An example we consider crisis:

- `GET /api/crisis 200` - Returns a JSON object containing a summary list of the crises documented in the database.
- `GET /api/crisis/{id} 200` - Returns a JSON object containing the full details on a particular crisis.
- `POST /api/crisis 204` - Creates a new crisis in the database. Takes as input a JSON object.
- `PUT /api/crisis/{id} 204` - Updates an exisiting crisis in the database. Takes as input a JSON object.
- `DELETE /api/crisis/{id} 204` - Delete an exisiting crisis.

For a particular crisis we have the example listing 3.1 (in JSON format).

```
{
  "id":1,
  "name":"Hurricane Katrina",
  "kind":"Natural disaster",
  "location":"Florida",
  "date and time":"08/30/2005",
  "human impact":"Making people homeless",
  "economic impact":"$81.2 billion",
  "resources needed":[
    "Electricity",
    "Purified water",
    "Shelter"
  ],
  "ways to help":[
    "Donation",
    "Telling other people about it",
    "Get involved in donation campaigns"
  ],
  "citations":[
    "http://katrinaresearchhub.ssrc.org/KatrinaBibliography.pdf"
  ],
  "links":[
    "http://en.wikipedia.org/wiki/Hurricane_Katrina",
    "http://www.katrina.noaa.gov/",
    "http://www.history.com/topics/hurricane-katrina"
  ],
  "images":[
    "http://upload.wikimedia.org/wikipedia/commons/a/a4/Hurricane_Katrina_August_28_2005_NASA.jpg",
    "http://upload.wikimedia.org/wikipedia/commons/0/06/Damage_and_destruction_to_houses_in_Biloxi%2C_Mississippi_by_hurricane_Katrina_14605.jpg",
    "http://newshour-tc.pbs.org/newshour/local/gulfcoast/slideshow_viewsfromground/media/slide1floodedhomes.jpg"
  ],
  "videos":[
    "http://www.youtube.com/watch?v=gWMfZA3sY8Y",
    "http://video.nationalgeographic.com/video/environment/environment-natural-disasters/hurricanes/katrina/",
    "http://www.youtube.com/watch?v=-Kou0HBpX4A"
  ],
  "maps":[
    "https://maps.google.com/maps?q=florida&hl=en&sll=30.307761,-97.753401&sspn=0.723213,1.451569&hnear=Florida&t=m&z=6",
    "https://maps.google.com/maps?q=hurricane+katrina&hl=en&sll=27.664827,-81.515754&sspn=11.855722,23.225098&hq=hurricane+katrina&t=m&z=9"
  ],
  "feeds":[
    "https://twitter.com/twc_hurricane"
  ],
  "organizations":[
    "United Methodist Church",
    "The Salvation Army",
    "Red Cross"
  ],
  "people":[
    "George W. Bush",
    "Angelina Jolie",
    "John Travolta",
    "Oprah Winfrey"
  ]
}
```

Listing 3.1: Crisis JSON Object

We can see that of much of our data consists of lists. We chose this design, because it is simple, yet informative.

The URL of corresponding API is chosen to have `api/` leading the URL for the HTML page. We decided not to use header info like `Accept: application/json` to distinguish between JSON requests and HTML requests. This facilitates debugging and development. We can switch to the more succinct design in no time.

3.2 RESTful API Python Implementation

To implement the above RESTful API specification we created a callable API-Entity class which manages the retrieval of data for a particular request. This APIEntity class uses the request context and the url path to decide what method should be called. Rather than write the API code for each entity (Crisis, Person, Organization) we abstracted away the notion of a particular entity and focused on the fact that each was represented by a django model. Then by defining the representation of data for each model we could abstract away the details of each entity from the APIEntity class. With the use of python decorators we were able to apply a functional programming methodology which allows for the composition of functions. With this we could abstract away the notion of errors and let the APIEntity focus on retrieving data. Let us consider a fragment of the APIEntity class. Note that the class is callable since it implements the call function. Once it decides which API endpoint should be called it makes that call. As an example if a call were made to /api/crises then callable would become the function getEntitySummaryList which would return the data as indicated.

```python
class APIEntity(object):
    def __init__(self, request, model):
        self.request = request
        self.request_method = request.method
        self.model = model

    @handle_error
    def __call__(self, idNumber=None, model=None):
        temp = {
            'GET': self.getEntitySummaryList if idNumber is None else
                self.getParticularEntity if model is None else self.getParticularEntitysRelatedEntities,
            'POST': self.createEntity,
            'PUT' : self.updateParticularEntity,
            'DELETE': self.deleteParticularEntity
        }
        if self.request not in temp:
            return {'status_code': 405, 'response_body': None}
        callable = temp[self.request_method]
        return callable(self, **{'idNumber': idNumber, 'model': model, 'post_data': self.request.body})

    def getEntitySummaryList(self, *args, **kwargs):
        data = [i.getSummary() for i in self.model.objects.all()]
```

```
            return { 'status_code': 200,
                     'response_body': data }
```

3.3 Django Models

Our Django Models consist of 8 different classes. Three of which are our three main categories: `Crisis`, `Organization`, and `Person`. Each of our main categories have zero or a few associations with each of its linked classes, representing its extensible attributes.

As seen in figure 3.1 our `Crisis` class has zero or more associations with our `Resource` class, our `WayToHelp` class, our `CrisisMedium` class. While our `Resource` class, `WayToHelp` class, `CrisisMedium` class can only be related to one and only one `Crisis`. Our `Organization` and `Person` class is structured in a familiar way. The `CrisisMedium`, `OrganizationMedium`, and `PersonMedium` each have 6 different media types: citation, link, image, video, map, and feed.

`Crisis`, `Organization`, and `Person` relate to each other in a many-to-many relationship.

Note that although `Crisis`, `Organization`, `Person` share some attributes, like `Medium` which store the links, images and so on, we separated these into different databases. On one hand, there is no possibility that a person will share a same image with a crisis. On the other hand, this simplifies database design. We do not have to uniform `Crisis`, `Organization`, and `Person` to be able to share attributes.

Though we have been talking about Python classes the whole time, Django is going to generate the corresponding SQL syntaxes for us. So as long as we get our classes right, the actual database will work as intended.

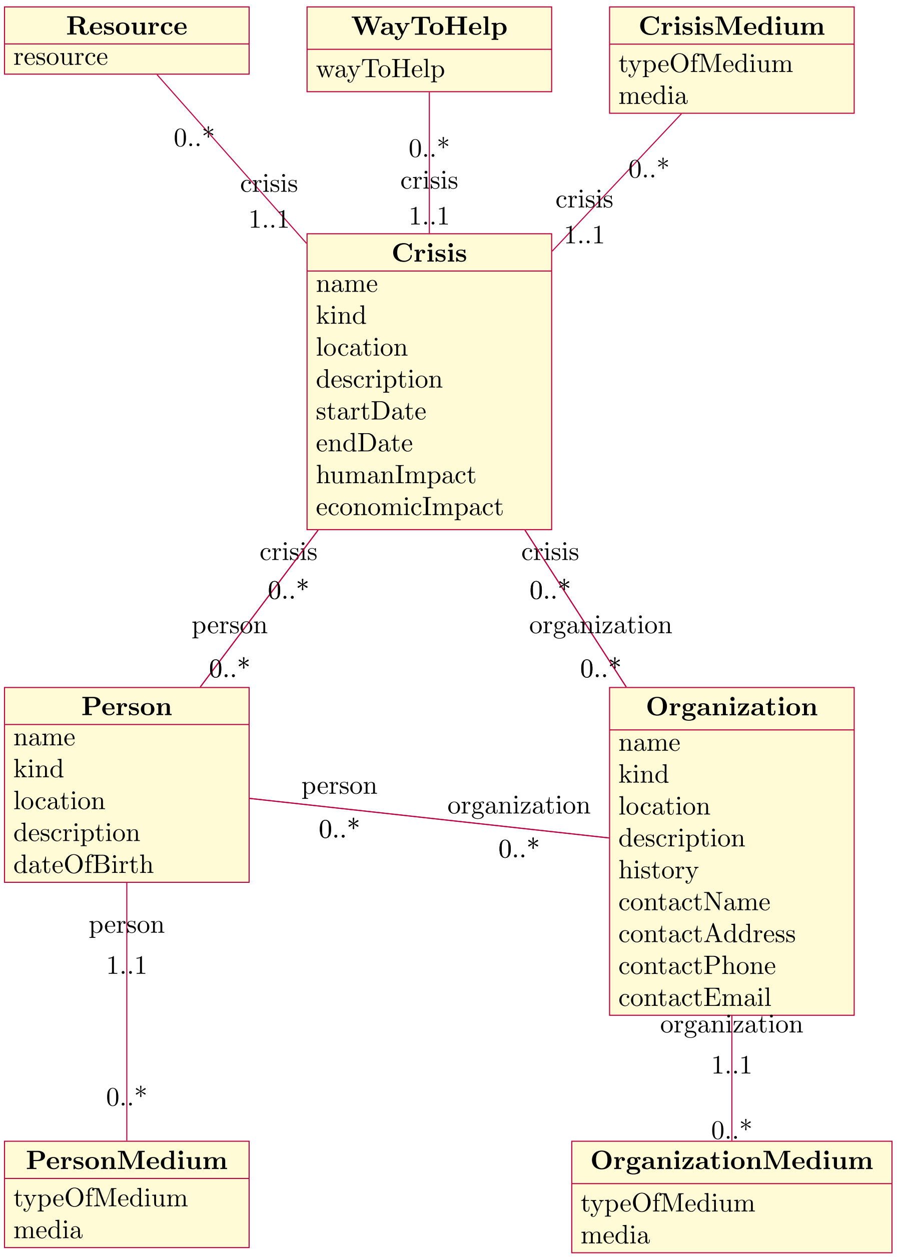

Figure 3.1: Crisis Diagram

3.4 Search Functionality

In order to add search functionality, each one of the models has a static method that utilizes the database search functionality to do the search.

It first creates a delicate Q object to combine the predicates. It does this both for the and operation and or operation, and makes sure that the two do not overlap.

Then it uses the Django and database API to do the actual search. It uses regular expression to match only whole words in a case-insensitive fashion. This is much faster than getting everything into Python and doing string matching.

The search results are parsed into the correct format. It has a title, a link, a random picture, and a description that combines all other fields.

Then the client side is responsible for displaying the result and show only relevant parts, displaying ellipsis for others. Then it bolds the keywords.

3.5 Unit Testing

3.5.1 SQL/Model Database Testing

In order to test our models, we used the Django ORM (Object Relational Mapping). This enabled us to test the SQL database by testing our Django model classes and their inner functions. For each Crisis, Organization, and Person, we wrote a test class whose member functions tested the functionality of the Django model. We wrote tests which both passed and failed. For the purposes of this report, we will only consider the tests for the Django model Crisis.

In the setup method of each test we created a Crisis object and added it to the database calling the save method.

To test our load function we created a JSON object which represented a Crisis and made the following call: crisisName.load(json_object). On completion, we then checked that the information was loaded into the appropriate field in the Crisis object. We then tested the same function with an invalid JSON object.

Next we tested the getSummary method. For the success test case we got the summary for the crisis object we created earlier (id = 1) and made sure that the fields 'name', 'id', and 'kind' where in the summary that was returned from the call to getSummary. We then used assertEqual to make sure the correct information was in the 'name' and 'kind' fields. Next for our failure test case we asked for the function to get the summary of a crisis object with the id = 2 (which does not exist). In this test case it should fail because it can not return a summary of an object that does not exist.

We next tested our getRelated method; this method will return a list of ID numbers depending on whether you pass in the model representing a 'Person' or 'Organization'. For our first test we made sure that the list was zero if the Crisis didn't have any relations. Next we created a person and associated it with our crisis by calling person.crisis.add(crisis) and vise-versa. We then did the call to get the list of ID numbers and made sure it was of length one. To test a failing case we tried to get the related crisis of a crisis.

To test our CrisesMedium class created a new CrisisMedium object which represented some particular media. We then associated the media with its correct crisis by putting it in the crisis field in the CrisisMedium class model. We tested the contents of the media fields by using assertEqual. For our failure test case we tried to set typeOfMedium to a letter that didn't exist in the dictionary of possible media (i.e something other than 'C', 'F', 'I', 'L', 'M', or 'V').

Next we tested our Resource and WayToHelp models, which was similar to the way we tested our CrisisMedium model. We created a Resource/WayToHelp object and put the crisis object in the crisis field. This crisis field represents the one to many foriegn key relationship between Crisis and Resource/WayTo-Help. We next tested this by making sure the fields were populated with the information we expected.

3.5.2 API Testing

We tested our APIs by making four methods of requests (GET, POST, PUT, and DELETE) to different URLs.

We checked that we returned the correct response code in the event of a bad request or a bad method. This occurs when we make a request to a wrong URL, or a method that is not supported. We also check that this happens when we make requests to an entity that does not exist.

For successful requests, we check that the response code is correct. For some tests, we have to set up the database by creating a few objects, and tear down by removing them. Then we can check if we actually have those objects by a GET request and test the response body.

We check DELETE by creating an object and then delete it. We verify the status code.

For testing PUT and POST, it requires more sophisticated set up. And we have to make sure that the changes really did happen.

3.6 User Interface

For our interface, we used Bootstrap, a CSS framework with many premade interface elements, such as buttons, panels, and navigation bars.

3.6.1 Bootstrap

Bootstrap is a CSS framework that provides a huge set of features, such as buttons and responsive design, to speed up the process of making a good-looking and usable website. Among the features we used were:

- Navigation bar
- Jumbotron (large title section of a page, meant to be the first thing you see)
- Buttons
- Glyphicons (monochromatic preset icons)
- Responsive columns. These columns will adapt to the size of the viewport, and avoid looking cluttered or ugly no matter what size the browser or device is

3.7 AngularJS

AngularJS is first and foremost an MVC (Model-view-controller) framework. It's goal is to allow programmers to develop highly dynamic websites fast and efficiently with the least amount of code as possible. We decided to use Angular for several reasons:

1. It greatly reduces the amount of JavaScript and HTML we would have had to write otherwise. (We didn't want to write static HTML.)
2. It fits perfectly with our API. We are able to create resources which map to APIs and then manipulate those resources as JavaScript objects.

3. Its view directives follow a declarative paradigm, which makes them conceptually easy to incorporate into HTML. View code can be kept completely free of imperative statements and separate from responsibility for control.

4. It is designed to facilitate unit testing.

5. It reduces the amount of work done on the server, having the client do most of the work.

6. It's a technology on the bleeding edge of web development.

This report does not intend to discuss how AngularJS works, but we will mention how AngularJS maps API endpoints to resources, how those resources are injected into controllers, and finally how the views are able to access that information.

3.7.1 AngularJS Resources

AngularJS resources are created by creating an AngularJS factory module, for example see listing 3.2.

```
angular.module('wcdbCrisis', ['ngResource']).
  factory('Crisis', function ($resource) {
    return $resource('/api/crisis', {}, {
      query: {method: 'GET', params: {}, isArray: false},
      queryOne: {method: 'GET', params: {id: '@id'}, isArray: false
          },
      update: {method: 'PUT', params: {}, isArray: false},
      add: {method: 'POST', params: {}, isArray: false},
      delete: {method: 'DELETE', params: {}, isArray: false}
    })
  });
```

Listing 3.2: AngularJS Resource Example

This gives us several things. It gives us an object titled `Crisis` which we inject into our AngularJS app as the module `wcdbCrisis` (see /static/angular/app.js for details). We are using an AngularJS function called `$resource`. What this does is return an object which has the methods: `query`, `queryOne`, `update`, `add`, and `delete`, which we can call with any JSON object and two optional functions. We will see some example calls in the next section.

3.7.2 AngularJS Controllers

AngularJS controllers are just functions which manage the logic behind getting and manipulating data. For each of our views (`crises`, `crisis`, `people`, `person`, `organizations`, `organization`), we will have a controller which manages that view. For example, let us consider the `Crises` controller. Its purpose should be to provide the data necessary for a view which displays a list of our crises. For an example see listing 3.3.

```
function CrisesController($scope, $location, Crisis) {
  $scope.pageName = 'Crisis List Page';
  Crisis.query(function(crises) {
    $scope.crises = crises['crises'];
  });
  $scope.go = function(id) {
    $location.path('/crisis/' + id);
  };
};
```

Listing 3.3: AngularJS Controller Example

There are a few important things to note: the `$scope` variable is where we store the data we want avabilable to our view. The `$location` variable is an AngularJS variable which keeps information on the URL location. Finally, we see our `Crisis` object which has been injected by AngularJS into this controller. Note that we do not have to specify anywhere else in the app that this particular controller takes the `Crisis` resource. This is done using variable/string interpolation.

Note that we provided a function to the `Crisis` object, this is because when the call to query is executed it returns immediately an object which promises to return the data. The function we have provided is a callback which is executed once the data returns. If I wanted a particular crisis I could have executed, see listing 3.4.

```
Crisis.queryOne({id: 1}, function(crises) {
  $scope.crisis = crisis;
});
```

Listing 3.4: Example Crisis Call with JSON Object

Above we can see that we can also provide a JSON object indicating which crisis we want returned.

3.7.3 AngularJS Views

AngularJS views are simply html files which use AngularJS templating system. Let us consider the crises view, i.e., the one which displays a summary list of the crises we have stored in our database. It has an associated controller (as above). Let us assume that we want to display the list of crises, to do that we would simply do the following in HTML:

```
<tr ng-repeat="crisis in crises" ng-click="go(crisis.id)">
  <td>{{ crisis.id }}</td>
  <td>{{ crisis.name }}</td>
  <td>{{ crisis.kind }}</td>
  <td>{{ crisis.location }}</td>
</tr>
```

Note that the `crisis` variable we are using in the view must have been made available to the `$scope` variable in the controller.

3.7.4 AngularJS Testing

AngularJS was designed with easy unit testing in mind, and comes already configured to work with the Karma test runner, allowing unit tests to run automatically whenever a project file is updated. Controller code is easy to test using the Jasmine framework. The AngularJS documentation provides the example listing 3.5 of succinct testing syntax:

```
describe('PhoneCat controllers', function() {
  describe('PhoneListCtrl', function(){
    it('should create "phones" model with 3 phones', function() {
      var scope = {},
      ctrl = new PhoneListCtrl(scope);
      expect(scope.phones.length).toBe(3);
    });
  });
});
```

Listing 3.5: AngularJS Unit Test

This test deals with a global `$ctrl`. But altering it to test a scoped variable is as simple as using the built-in `$controller` function, which will cause AngularJS to look up a controller by name.