Great Moonstone Oxen of the First and Forsaken Oceans

Taylor McKinney Matt Stringer Cesar Cantu
Spencer Reynoso Ben Bowley-Bryant David Denton

July 11, 2013

Introduction

What is the problem?

It's often easy to forget the many crises occurring throughout the world. This site is designed to gather the data on every crisis, and present in a simple format. We gather details about victims, and information on organizations and people that want to help.

What are the use cases?

With the increasing availability of telecommunication access and this site makes use of this to share news and help victims. This site can be a good source for anyone looking to get informed and help. We can provide contact information for people and organizations involved in helping the victims of crises around the world. People who want to help can look to our site for the best ways to get involved, whether it be donating to the right charities or volunteering at a local site.

Implementation

Source Code

Most of the code, including the *import* and *export* scripts, are done in Python. The website is delivered using Django and the site is designed using Twitter's Bootstrap. The data is stored to a MySQL server on campus. Most of the separate groups have already shared data. Our database already holds data from all of the shared groups, as well as our own.

Folder Structure

The model of the data structure can be seen below.

```
wcdb/
    crises/
    scripts/
    static/
        css/
        html/
        img/
        js/
        templates/
        xml/
```

Figure 1. The Folder Layout

The root folder *must* be named *wcdb*. If downloading from Github, rename *wcdb*. Inside the *wcdb* folder, there are three folders *crises*, *scripts*, and *static*.

The *crisis* folder contains the *crisis* application and the standard Django files, including *models.py*, which has details about the table structure, and

tests.py, which contains most of our unit tests.

The *scripts* folder contains the scripts *importScript* and *export*. The *importScript* is named to avoid conflicts with the *Python* keyword.

Inside the *static* folder, there are the folders *css*, *html*, *img*, *js*, *templates*, and *xml*. The *css* folder contains the CSS files associated with Twitter's Bootstrap.

The *js* folder holds javascript files from Twitter's Bootstrap. We currently use this javascript to animate our navigation sidebar. When you hover your mouse over one of the links, its row will highlight. The *templates* folder holds just a single HTML file for now that is not in use. Later, this folder will be used to hold templated HTML data that will be used recurringly throughout the site. The *xml* folder currently holds several XML that were temporarily used during development. The only important file is *WorldCrisis.xsd.xml*. This file is the schema we test our XML data against in our *importScript* and *export*.

The *html* folder contains all of our content pages. The file *base.html* is similar to a template. It contains most of the HTML data that can be seen on the site, with several blocks filled in with content from other pages. The other pages, excluding *export.html* and *export.xml* all extend *base.html*. When Django is rendering a page, it will start by rendering data from the page it extends. Since all of our pages extend *base.html*, all of our pages use the html found there. When Django is filling in content from *base.html* and encouters a block, it will fill in this block with data from the page it was first passed. The *crisis*, *organization*, and *person* pages are our static pages. They hold information that will be filled into the template by Django. The static pages are still being delivered by Django.

Data Model

The core of our data model starts with the tables *Crisis*, *Organizations*, *Person*. These three tables all have a many to many relation between each other. We have a *Common* table to hold data these three models can all have. *Crisis*, emphOrganizations, and *Person* can have 0 or 1 *Common* objects. We have an abstract model, *AbstractListType*. The database will never write an *AbstractListType*, but abstract types are useful when scripting. The types *CommonListType* and *CrisisListType* both inherit from *AbstractListType*. *CrisisListType* will hold data for *Crisis* objects, and *CommonListType* will hold data for *Common*.

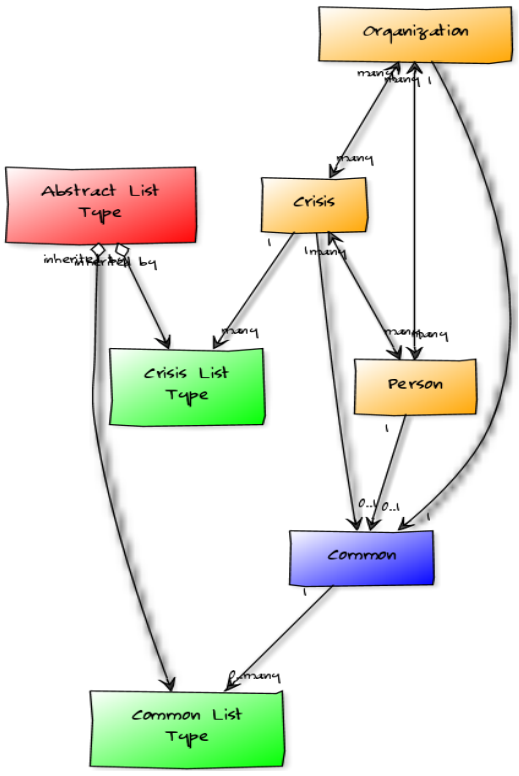

Figure 2. A diagram of our data model

Import

Import is implemented as a Python script, in the *scripts* folder. It is named *importScript.py* to avoid naming conflits with the *Python* keyword. It reads in an XML file as input, parse the information, and stores the applicable data into the database for future viewing. The XML file must conform to the schema in Figure 1. Otherwise, no data will be imported *Import* is password protected, and only site administrators may run this script. Running import is done on the website. There is a link in the navbar that reads *Import*. From this page, an administrator may select a file and upload the information.

Export

Export is implemented as a Python script, in the *scripts* folder. It is named *export.py*, though may likely be renamed during a later stage to match *importScript.py*. It takes the data stored in the database and writes into a new XML file conforming to the schema in Figure 1. *Export* does not require administrator access. Any user may export the data in this database by clicking the *Export* link from the navigation bar.

Design

XML Schema

Our class has been divided into separate groups for this project, but most groups have agreed upon a shared schema for XML. The schema as of the latest version is printed here, but the latest version can be found with this link.

Updated schema on Github

Testing

Testing is done using tools provided by Django and unittest. To run the tests, run the command **python manage.py test**. This command will run the unit tests using SQLite3. SQLite3 was used for faster unit testing. MySQL should be used for testing in later phases of development to ensure the code is runing properly on our production environment. The ideal solution is to create two testing environments, the first of which runs unit tests on SQLite3 for performance purposes, and another to run tests on MySQL before pushing to ensure there are no differences between the two that could cause errors.

Django extends the standard unittesting framework, and provides several hundred tests against the back end. In addition, we've added a number of unit tests to test our scripts *importScript* and *export*. There are plans to implement further testing for each view, but for now we are only showing static pages, and the views are easily checked manually.

Future Goals