Great Moonstone Oxen of the First and Forsaken Oceans

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

July 24, 2013

Introduction

What is the problem?

It is often easy to forget the many crises occurring throughout the world. World Crisis Database is designed as a framework to effectively gather data related to recent crises. WCD then presents the crisis data, including victims, organizations, and political players involved, in a digitally logged format in an user friendly format.

What are the use cases?

This site makes use of worldwide internet access by bringing news regarding crises and by promoting help for victims. This site can be a effective resource for anyone who wants to be informed on current events. Furthermore, individuals who has a desire to help can easily find contact information for people and organizations involved and from there find best ways to get involved, including infomration on how to donate to the charities or volunteer at a local site.

Implementation

Source Code

Most of the code, including the *importScript* and *export* scripts, is written in Python using Django web framework. The base level CSS and javascript are built off of Twitter's Bootstrap front end framework (found at the following website http://twitter.github.io/bootstrap/). The framework provides templates for buttons, navbars, and icons along with some javascript functionality. The data is stored to a MySQL server on campus and hosted on Z at the following URL http://zweb.cs.utexas.edu/users/cs373/benbb/.

Since most of the groups have already shared data, the production database already holds data most of the data including 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, named *wcdb*, must be renamed to wcdb when downloading from Github. Within the root folder, there are three subdirectories: *crises*, *scripts*, and *static*.

The crisis folder contains the crisis application and the standard Django files:

- *models.py* - a detail of the table structure
- *tests.py* - the unit tests and single acceptance as a catch for any bugs in the program.

The scripts folder contains the scripts *importScript* and *export*. The deviation in convention to importScript is to avoid conflicts with the Python keyword import. In phase 3, both scripts *export* will be renamed to importXML and exportXML in order to have a similar convention and more appropriately describe their function.

Inside the static folder, there are the following 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. Twitter's Bootstrap has provided the following functionality to the website:

- animating the navbar down along with drop menu for each item when clicked
- capturing text for searches
- highlighting the row when a link is hovered over
- providing a image carousel for each crisis

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 our XML data tests 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.

Data Model

The core data model starts with the tables following tables: *Crisis*, *Organization*, *Person*. All three tables are connected via a many-to-many relationship between with a *Common* table to hold data that all three models share. In the common table, Crisis, Organization, and Person can have 0 or 1 Common objects. There is 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.

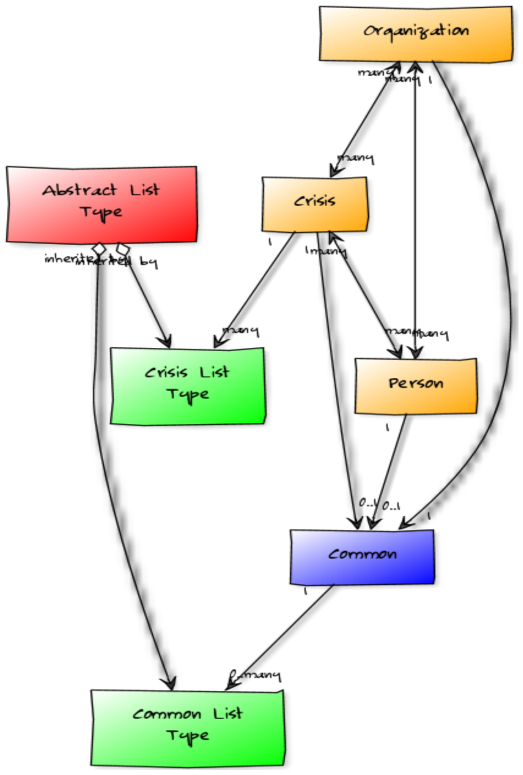

Figure 2. A diagram of our data model

XML Importation

The XML importation functionality is implemented as a Python script named *importScript.py* and is located in the *scripts* folder. The file name was chosen specifically to avoid conflicts with Python's import keyword. It reads in an XML file as input, parses the information, and stores the applicable data into the database using Django's model system.

The XML importation functionality is on a page on the website. This page is only accessible to site adminstraitors and those with sufficient privileges. A link to the page itself is found under the Adminstrative drop down menu, and is labeled "Import" From the page, an administrator may select a file and upload the information. This file must conform to the schema as described in *WorldCrisis.xsd.xml*; importation will fail otherwise. Any XML containing *Document Type Definitions* (DTD) is rejected as a security risk.

For testing purposes the password to gain access to the XML import functionality is currently **downing**. Users retain permissions to access the page through a cookie. This cookie will expire five minutes after logging in or immediately after closing the browser.

Detailed Look

validateXML

This function is what parseXML uses to validate the XML. It uses minixsv to validate the XML given in the parameter and returns false if an error is thrown. The XML Schema used **must** be in the same folder as the script, and must be named “WorldCrisis.xsd.xml”. This function will not work otherwise.

parseXML

Parses the file given in the parameter and returns an *xml.etree.ElementTree* Object.

xmlToModels

Converts an *xml.etree.ElementTree* to Django database models, as well as stores them in the database. The database must be properly configured for this function to properly store and create the models.

link_up_dict - The keys for this dictionary are always strings whose contents are the IDs of either crises or organizations. The corresponding value for any given key is a list of IDs from crises, persons or organizations that reference the key.

reference_dict - The keys for this dictionary are the IDs of crises, persons or organizations stored as strings. The value for the key is the model the ID references.

addReferences

Fills *link_up_dict* as outlined in **xmlToModels**.

linkUpModels

Associates the models stored in *reference_dict* with the models in *link_up_dict* and writes them to the database.

XML Exportation

Export is implemented as a Python script in the scripts folder. (It is named export.py, although it will be renamed during a later stage to be consistent with importScript.py.) Export the data stored in the database and writes it into a new XML file conforming to the schema. The user is prompted to download the XML file automatically. As export does not require administrator access, any user can export this database by clicking the Export link from the navigation bar.

Design

XML Schema

Although our class has been divided into separate groups for this project, our group agreed to share an XML schema between groups. The latest version of the schema can be found at this url.

https://github.com/aaronj1335/cs373-wcdb1-schema/blob/master/WorldCrises.xsd.xml

Theentire XML file is wrapped in a *WorldCrisis* object. Within the WorldCrisis object, there are one or many *Crisis*, *Organization*, and *Person* elements. Each of those elements have type key types, named *CrisisKey*, *OrgKey*, and *PersonKey*. There are wrappers for containers of these objects that can contain a list, i.e., a Crises container for multiple Crisis objects, etc. Furthermore, there are type definitions for each type: *CrisisType*, *PersonType*, *OrgType*.

CommonType defines data related to Crisis, Person, and Organization, such as links, images, and videos. There is a *ListType* definition that contains a list of XML tokens to facilitate objects with lists of data.

Testing

Testing is done using tools provided by Django and unittest. To run the tests, run the command **python manage.py test**, which will run the unit tests using MySQL. Initial test used SQLite. Since MySQL is used in production, all tests have been shifted to this database platform in order to ensure the code is running properly on our production environment. The ideal solution is to create two testing environments: one that runs unit tests on SQLite3 for performance purposes, the second runs tests on MySQL before pushing. This allows us to run quick unit tests while working and ensure the site works on our production environment.

Django extends the standard unittesting framework, providing several hundred tests against the back end. In addition, a number of unit tests are added for the scripts importScript and export.

Page Design

User Accounts

Features

User Accounts

To gain access to the WCD website without admin access, there is a create user page where potential users sign up to the WCD website. Those that signup to webpage via this portal will be given access to view all crises, organizations, and people enterred in database. Admin privileges are defaulted to false, which gives all site functionality except for allowing non admins to import XML in to the database. For those who may need admin privileges, accounts can be manually updated outside of the create users page.

Search

This uses the

Random Button

Added as the F̈eeling Symphatethic Button;̧ this button randomnly shows an existing crisis, not organization or people.

Phase 2 Goals

The primary future goal for phase 2 was to get the site presenting data from the database dynamically. This has been completed.

Also at the end of Phase 1, the site has been modeled after IMBD website with emphasis on adding the specific functionality found in the site. The following list below details the the specifications and their status into into the WCD website functionality:

- Each page should have a gallery for its related images (completed)
- A search page, where users can search for any key (completed)
- Organize pages by filters, like date and location (phase 3 issue)
- A short list of recent events on the home page based on how k̈indẗhey are (a measure for total impact) (phase 3 issue)
- Improve the display of data, including charts and graphs (phase 3 issue)

Some other desired features:

- A Kindness Slider, where users can vote and rate organizations and people
- A proper logo for our team and site
- A random crisis action, to show people random crises
- A sympathy button for crises
- User profiles

Phase 3 Goals

Under Phase 3, we plan to add the following goals:

- Construct SQL queries to submit to Piazza
- Improve randomize button to pull from non-crises
- Coordinate with other teams to get XML data into a single db
- Generate cd