World Crises Database

Stage 1 (WCDB1)

Team CrisCS

Ambareesha Nittala

Brandon Fairchild

Chris Coney

Roberto Weller

Rogelio Sanchez

Vineet Keshari

Table of Contents

Repository: https://github.com/vineetkeshari/cs373-wcdb

1. Introduction

1.1 Problem

We are setting up a website of world crises much in the same manner as IMDb that will access a searchable database with crises and associated people and organizations.

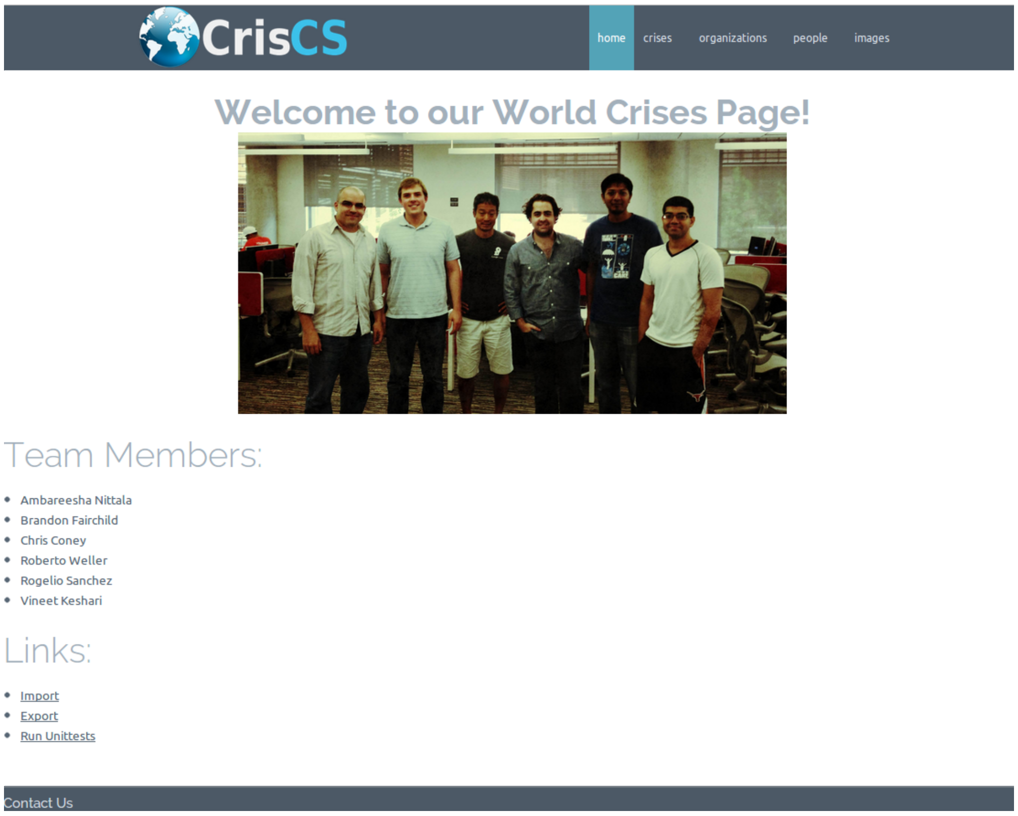
1.2 Use Cases

End users can perform a search by category (crises, person or organization) that will come back with pertinent information on the searched element and links to associated elements. For example, if West, Texas explosion is queried, not only will the information about the explosion come up, but also links to the people and organizations associated with the explosion. Say, Rick Perry is a person associated with the explosion, there will be a link to Rick Perry's page with will bring up information on him as well as associated crises and organizations, etc…

1.3 Directory Structure

- **WCDB1.pdf** : self
- **WCDB1.log** : git commit log
- **doc/*.html** : pydoc documentation/
- **criscs/** : Site directory
  - **urls.py** : Contains all valid URLs
  - **ModelFactory.py** : Creates new models defined in models.py
  - **XMLUtility.py** : Reads, parses and validates XML
  - **WCDB1.xsd.xml** : XML schema
  - **WCDB1.xml** : XML file
  - crises/ : App directory
    - **tests.py** : Runs unittest, **!!REPLACES TestWCDB1.py!!**
    - **views.py** : Reads models from database and renders views
    - **models.py** : Defines models

- forms.py : Defines import form
- admin.py : Defines admin calls
- templates/ : All HTML templates live here
- *.html



Welcome to our World Crises Page!

Team Members:

- Ambareesha Nittala
- Brandon Fairchild
- Chris Coney
- Roberto Weller
- Rogelio Sanchez
- Vineet Keshari

Links:

- Import
- Export
- Run Unittests

Contact Us

Illustration 1: Splash page

2. Design

2.1 XML Schema

Fun Fact: The XML schema (**WCDB1.xsd.xml**) used by all teams was written and gratuitously shared with everyone else by Vineet Keshari from our team.

The XML schema has the following properties:

- WorldCrises contains elements Crisis, Organization and Person that contain all data about the three types of pages.

- Each of the three types has mandatory ID and name fields. Everything else is optional.
- Keys are defined and validated using regular expressions as strings of length 10 in the format “TTT_NNNNNN”, where TTT is in [‘CRI’, ‘ORG’, ‘PER’] and NNNNNN is an identifier of six uppercase alphabets for the element.
- Many-to-many relations between all three pairs of elements are defined by using keyrefs from an element to the keys of its related elements as defined above.
- All common content {external links, citations, maps, images, videos, feeds, summary} is defined in a ‘Common’ sub-element.
- All content that is more complicated than plain-text is defined as a ListType. A ListType is a generic container that can have one or more ‘LI’ elements. The fields in ‘LI’ elements {content, href, embed, text} have been chosen in such a way so as to accommodate representations for all types of data.
- Tight checks on content, min and max occur for each element to ensure that all data collected by various teams will be consistent.

2.2 Models

The models defined in **models.py** follow the same hierarchy as the types defined in the XSD. Figure 1 shows how they are related. The models are:

- Each of Crisis, Organization and Person models extend the base model WCDBElement, which contains the name, ID (primary key) and other fields common to all three models.
- ListType is a model corresponding to one ListType in the XML. Each ListType references its parent WCDBElement. There is a one-to-many mapping from WCDBElement to ListType, as each element may contain many ListTypes.
- Each sub-element of a ListType (LI) contains the following fields: content, text, href and embed. These are stored in the LI model and reference the parent ListType model. Since a ListType contains one or more models of type LI, ListType to LI is also a one-to-many mapping.
- Relations between crises, people and organizations are stored in models R_*, which store references to the two respective elements they relate.

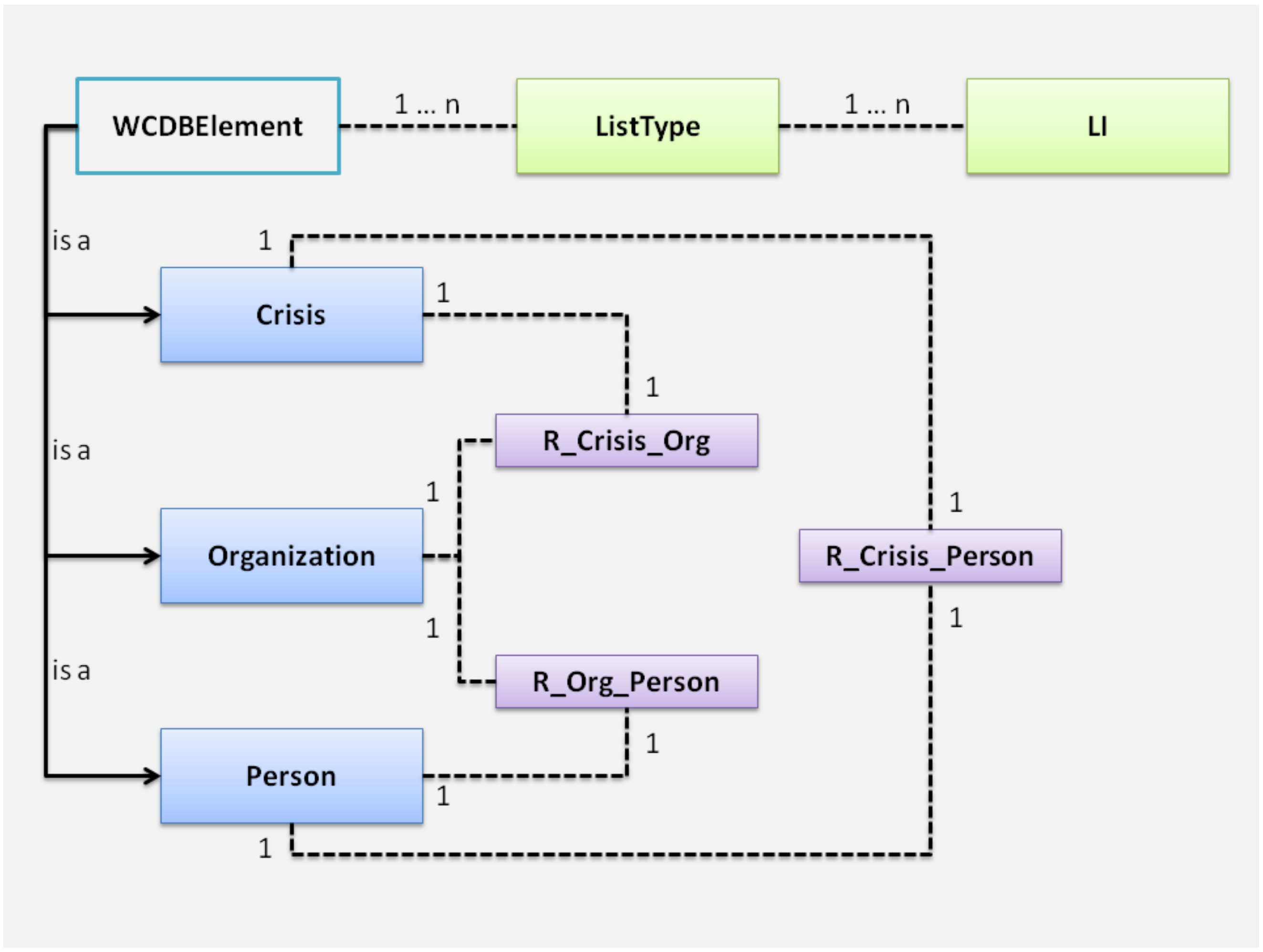


Figure 1: Django Model Types and Relations

3. Implementation

Figure 2 illustrates the data and control flow pipeline for the application. The various components are defined in detail in the subsequent subsections.

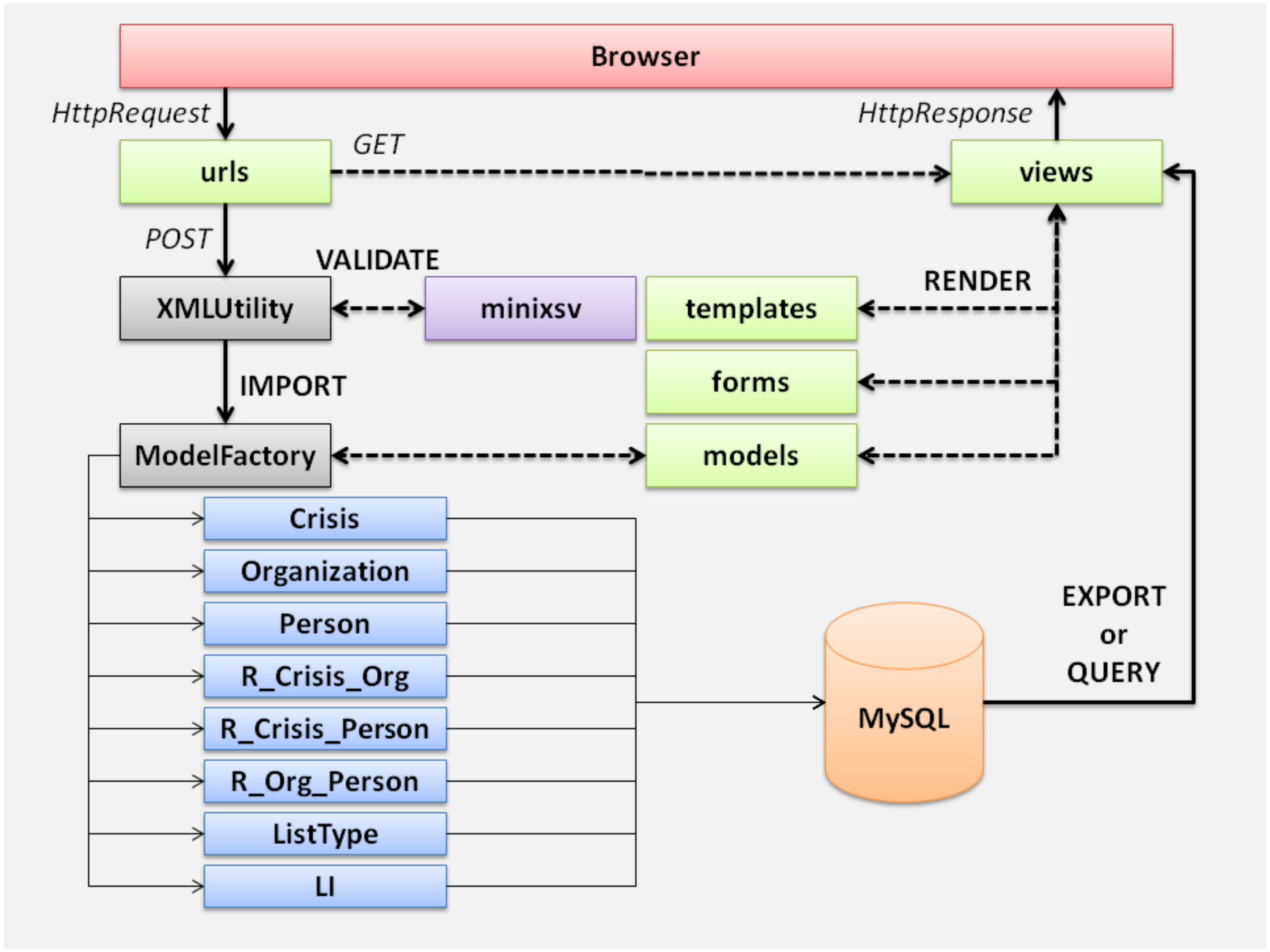


Figure 2: Application pipeline for data (solid) and control (dotted) flows

3.1 Import/Export Facility

XML files can be imported using the 'Import' link on the homepage. The import pipeline has the following elements:

- **forms.py** defines a form containing fields for file upload and password.
- **views.py** checks the password, reads the XML file from the POST request and stores it in a temporary location under <appname>/xml/*/*
- **XMLUtility.py** uses **minixsv** to validate the stored XML against **WCDB1.xsd.xml**
- It then reads the XML into an ElementTree and calls **ModelFactory.py** for each element to generate the corresponding models.
- It then passes the generated models to **views.py** for database persistence and response generation.

The models stored in the database can be exported into an XML using the 'Export' link on the homepage. This will generate a human-readable XML of all the models and display it in the browser. The generated XML validates against our schema (**WCDB1.xsd.xml**).

3.2 Template and Views

We defined functions under the **views.py** file to access and format data from the mysql database to a django instance, which is then passed to a bootstrap instance. There are three main "view" functions named "'X_view" that call on several helper functions to arrange the data needed to pass onto our defined template file. The "X" in "X_view" stands for either person, org (organization), or crisis. Also defined are functions that collect important information from the values within the database that is needed to help create the data structures we used to then breakdown this data further. Below is a better breakdown of the functions.

The function *get_indices* takes in parameters that are used to find the type of list types that are needed. We then create a dictionary that stores these list types. For this *view_id* being passed through, the locations of every single list type are collected. We store the indices to the corresponding keys in the dictionary and then return it to the calling function. The function *get_media* is a good example of where *get_indices* is used. As the function is named, *get_media* used the index information from *get_indices* to collect the data /media stored at those indices. This data is then appended into a dictionary that is returned to the calling function. The function *get_crises_details* also makes use of *get_indices*.

3.3 Frontend

3.3.1 Front End Design

For the styling of the front end we decided to integrate Bootstrap to our project. Bootstrap is an open-source frameworks that integrates css, javascript, and html. Bootstrap was created by twitter and allowed us to make our webapp look exceptional without having towrite all the css and javascript.

3.3.2 Front End Structure

We really did not want to have to rewrite code for every template. Using django we were able to use a hierarchical structure in which all of our templates inherited from a base template. this template contains the entire header and footer for all our pages. The header's tabs are dynamically filled with the names of our crises, people, and organizations.

In order to achieve the hierarchy we used blocks.

The base html (parent) file contains the following constructs:

```
{% block content %}
{% endblock %}
```

and its child html files contain:

```
{% extends "parent.html" %}
{% block content %}
{% endblock %}
```

4. Testing

4.1 unittest

Unit tests are performed by the server. We have the server exec the test.py file directly by having it execute the "python manage.py tests.py" command. We pipe the output to our unit tests view. Our unit tests are comprised of a variety of positive and negative tests, ensuring that the our methods operate as intended with valid and invalid inputs, respectively. Right now, our tests are primarily whitebox. We plan on implementing blackbox tests in the future using some kind of testing framework. We subclassed the test runner and removed the database setup and teardown to bypass the test_db creation for Project 1. For project 2 we will move to using the test_db so that we may have better test coverage.