Group Name: The A Team
CS373 Summer 2013
Glenn Downing

Administrative Password: ateam

Group Members:

- Brandon Lee
- Paul Carroll
- Shaelyn Watson
- Shan R. Gupta
- Olga Saprycheva
- Eduardo Saenz

Introduction

World crises require quick distribution of information if they hope to be alleviated by collective human effort. This can be accomplished using efficient website design to relay relevant information, gather more information, and rally support. The problem modern society faces is attaining knowledge about disastrous events that do not impact us personally or are outside the comfortable line of sight. Websites such as the one designed by The A Team put such information in range by presenting relevant information about world crises in a clear, usable, and eventually dynamic manner.

When a user visits the WCDB website, a use case is defined. Someone would visit our site to learn about world crises such as: NSA Wiretapping, the Mexican Drug Violence, and the Bee Colony Collapse, etc. An administrator can also update the data presented on the website by upload a well-formed and valid XML file. The underlying XML Schema allows for maximum data entry about crises, people, and organizations. The implementation of an import option adds dynamic abilities to our site. An export option on the website is added to provide proof-of-concept when an import is regurgitated back as XML, allowing data retention and integrity validation.

Instead of presenting data through a set of static pages, as in the initial phase of the project, the Django templates were rewritten to be dynamic. This is realized by the HTML drawing information from the database. These features make for an improved website because they allow the public to add world crises that deserve attention.

Design

XML Schema :

The XML schema design specifies an organized structure for the data about crises, related organizations, and people at the lowest level of the WCDB webpage. The schema defines the attributes that are acceptable in the XML documents to be uploaded into the site, and the schema for this project had to be accepted by the entire class. There is data that is common across all three entities: citations, external links, videos, images, maps, social feeds, and a summary. These data have been abstracted into a 'Common' tag in the XML schema. All elements in the Common tag except the Summary tag, are 'ListType' meaning they are composed of list items which are able to hold hyperlink references as well as text data. This list structure is excellent for presenting data as bulleted points on a webpage, but it did require an extra table to be constructed which is discussed in the next section. Example:

```
<Crisis ID="CRI_NSAWRT" Name="NSA Wiretapping">
        ...
        <Common>
                <Citations>
                        <li href="http://www.guardian.co.uk/...">Traynor, Ian. "E ...</li>
                        …
                </Citations>
                ...
        </Common>
</Crisis>
```

The person, org, and crisis tags are "interconnected." They know about one another because of the ID/IDREF system in the schema. There is a standardized identification system for crises, people, and organizations set by the class. For example the identification for Edward Snowden is PER_ESNWDN. This identification system is seen throughout the other two tags as well. For example:

```
<Crisis ID="CRI_NSAWRT" Name="NSA Wiretapping">
        <People>
                <Person ID="PER_ESNWDN" />
        </People>
        …
</Crisis>
```

Local Directory Structure:

The directory structure adheres to the django standard. We have a main project directory called ‘cs373_ATeam’ and then within our project we have our app directory ‘wcdb’.

```
cs373_ATeam - project directory
        ++-- minixsv
        |       ++--files for minixsv module
        ++-- genxmlIf
        |       ++-- files for genxmlIf module
        ++-- wcdb - application directory
        |       ++-- static - static files directory
        |               ++-- Bootstrap framework files, CSS, and JavaScript
        |       ++-- templates - template html directory
        |               ++-- wcdb
        |                       ++-- *.html - html template files
        |       ++-- models.py - models file
        |       ++-- views.py - views file
        |       ++-- loadModels.py - logic to load models from xml
        |       ++-- unloadModels.py - logic to export xml from models
        |       ++-- getDbModel.py - export of crisis, orgs, and people in dict form from db
        |       ++-- tests.py - unit tests
        ++-- urls.py - controllers file
```

Heroku Directory Structure:

```
cs373_ATeam - project directory
|       ++-- static - static files directory
|               ++-- Bootstrap framework files, CSS, and JavaScript
|       ++-- urls.py - controller file
|       ++-- settings.py - settings file
|       ++-- test_suite_runner.py - custom test suite runner
|       ++-- wsgi.py
wcdb - application directory
```

| ++-- templates - template html directory
| ++-- wcdb
| ++-- *.html html template files
| ++-- models.py - models file
| ++-- views.py - views file
| ++-- loadModels.py - logic to load models from xml
| ++-- unloadModels.py - logic to export xml from models
| ++-- getDbModel.py - export of crisis, orgs, and people in dict form from db
| ++-- tests.py - unit tests
minixsv
| ++-- files for minxsv module
genxmlIf
| ++-- files for genxmlIf module
venv - virtual environment directory

Django Models :

A Django model is a class that inherits from models.Model. These models interact with the database behind the scenes in such a fashion that when the database is synced with the model definitions, the model names become the tables in the database, the class variables that are defined as one of the “models” types within the model class become the columns for that table, and instances of that object that are saved using instance.save() become the records in the table, filling in their corresponding database values based on the values given to them for the attributes defined in the model definition.

The purpose of our models is to represent the crises, people, and organizations and the data (such as images, videos, text, etc.) associated with them along with the connections to other models as well. The models are influenced by the schema in that the information stored in the model reflects the information received through import as defined by the XML schema for the website. Thus, the models contain member variables that correspond to the data associated with the three main tags.

The models definition for this website contains five models: Crisis, Person, Org, Relations, and Li. There is also a helper class called Common which is not a model as it does not inherit from models.Model. The Crisis, Person, and Org objects are meant to hold the information about crises, people, and organizations that are detailed in an imported XML file. The Relations model is meant to store the relations between various organizations, people, and crises, and thus has the 3 ID fields, along with an auto-generated id field which serves as the primary key for the corresponding table (Django does not allow for multiple column primary keys at this time). The reason for having this model is so that a separate table will be created in the database to store these

relations, removing the need to store multiple entries in, say, the Crisis table in order to reflect the relation between a single crisis and multiple organizations and people. The Li model represents the list items that are seen throughout the schema. The reason for the existence of this model is the same as that of the Relations model, with the only difference being that the Li model stores information about these list items and associates them with their corresponding person, organization, or crisis via the model_id field.

The Common helper class reflects the presence of the common node in the XML schema that was defined for the website. However, there is no need for a Common table in the database because all of the elements of a given Common node in the XMl are elements associated with a crisis, person, or organization. Thus, the common helper class simply allows for an easy way to make the models.py file imitate the XML schema and make processing of the other models easier.

These models are populated with information via the import function that is described later in this report. The information that they are populated is what is then used to store information into the database backing the website. In addition to allowing new entries to be added to the database, Django models also allow updates to add more information to an existing crisis, person, org, or list item or alter existing information using the same method used to add new entries: model_instance.save(). Thus, it is possible to use this Django model system to quickly merge new data into the database, and then have it reflected in the website itself.

Implementation

Import : <Brandon, Shan>

Given that the goal of this website is to spread as much information about the crises affecting our world as possible so that people know what they are dealing with and that the amount of information to collect is near limitless, it simply would not be feasible to first collect our information and then post the website. The site must be dynamic; it needs to have the ability to incorporate new information as it comes in. This is accomplished via the import facility.

To access the import facility, import_view() is called from views.py. The view provides a password-protection feature to prevent people who are not authorized to be altering the information behind the website from altering the database from which the website is constructed. In order to alter the database, the import facility relies on XML files to handle the information that is to be added. However, this information could easily contain a formatting error of some sort which could then result in incorrect information being added

to the database. To handle this possibility, the import_view uses the pyxsval function from the minixsv module to first validate the incoming XML against a pre-defined XML schema ("WorldCrises.xsd.xml") before allowing the XML to be imported into the database. The validation function is stored in loadModels.py, is called "validate()" and handles the parsing of the XML file into a tree using ElementTree, returning the tree as its result.

Once the file has been validated, the tree it produces is traversed to produce objects of the 5 different django models types, and stores within these instances the information associated with them before saving those instances to the database as records in the corresponding table. It begins by breaking the XML down according to people, crises, and organizations, creating objects for each one using the populate_models() method from loadModels.py. These objects are then populated with the corresponding information through the populate_crisis(), populate_person, and populate_org() methods. While the information about each of these instances is being added to the models, the import facility also checks the fields that relate a given person/crisis/organization instance to other people/crises/organizations and creates the appropriate instances of the Relations model based on the XML data and then saves them to the database. Li instances are created and saved in a similar fashion, although the work is outsourced to the populate_li() method. In the case of Li's stored in the Common node in the XML file, this call to populate_li() is made via a populate_common() method, whose presence is the result of having to access the Li information through a Common node on the tree rather than just through a person, crisis, or organization node.

The import facility also checks when it is adding Li instances and Relations instances to make certain that the entry it is adding is not a duplicate of an already existing value. This is crucial because without this check, something as simple as importing the same file twice would result in all the Li instances and Relations instances being duplicated, a problem which would show very clearly on the website. This check is not necessary for the People, Crisis, and Organization instances because these three models have pre-defined IDs that are used as primary keys. These primary keys are used by Django to check for the presence of a pre-existing record with the same ID, and if one is found, the records are merely updated rather than inserted. This is not possible for Li and Relations instances because they use auto-generated and incremented ID's for their primary key since Django does not allow for multi-column primary keys at this point in time. It should be noted that the check used for Li's isn't perfect as it will fail if there is even the smallest difference, resulting in a new entry in the database.

Export :

The website comes with an export facility that allows users to access the information stored in the website. By making the information available to users in a standardized

format, it becomes possible for them to not only spread the information already available but also to add to that information and even send the updated data back to this site's administrators so that new information can be added to the website.

In addition, the export facility can serve two side purposes: it allows teams working on the site to verify that the information they wish to store on the site is being retained by the database, and that the information is being stored in the manner desired. In addition, it can serve as an acceptance test of sorts by comparing the XML that was imported to the site with the XML that is exported by the website.

The export facility for the site is spread over two different files: getDbModel.py and unloadModels.py. The reason for this split is that the functions used in getDbModel.py are used not only in the export facility, but also to aid with the construction of the front-end portion of the site (the actual HTML files that display the information that is being stored). getDbModel.py contains six functions that work in conjunction to allow the export facility and the team working on the front-end part of the site to retrieve the data stored in the database. These six functions are getCrisisIDs, getPeopleIDs, getOrgIDs, getCrisis, getPerson, and getOrg. The ID functions construct a dictionary for crises, people, and organizations, which store a key:value pair in which the key is the object's ID and the value is the associated name. The other three get functions take an ID and use that ID to query the database, obtain the information associated with that ID, and pack that information into a dictionary to be provided to the caller of the function. Together, these two groups of functions allow the retrieval of any information that might be needed from the database.

It should be noted that all the fields that can have multiple values and are considered list items, and as a result are stored in a sub-dictionary called common_dict. Common_dict also stores the summary field because it is a member of the Common node in the XML schema. In addition, it should be noted that in the event that there are, for example, no people stored in a database, an empty dict is what will be returned, not a None object. For a person with a blank value for one of its singular data fields, the return value will simply be an empty string while for a field with potentially multiple values, the return value will be an empty list. The format for the dictionaries that are returned is simple: The key is the name of the attribute (such as "History" or "Locations") and the value is either the singular value associated with that attribute, or a list of the values that are associated with that attribute.

Due to the nature of the data organization within the database (having a separate table for list items and for relations between crises, orgs, and people), it is also necessary to search through the Relations and Li tables in the database when obtaining the information associated with a given Person, Crisis, or Org ID. For example, if a specific person ID is being searched, in the dictionary that's returned, the values corresponding to organizations and crises are lists because one person can be involved in more that one crisis or organization.

The part of the export facility that is entirely unique is stored within the unloadModels.py file. Here is where the data obtained using the functions in getDbModels.py is actually exported to an XML file. The functions here are branched with several helper functions to handle various items. The initial export function grabs the dictionary of person, crisis, and organization ID's and for each of those ID's it branches to the export_crisis, export_person, and export_organization functions which construct a string with for the person, crisis, or organization associated with the ID.

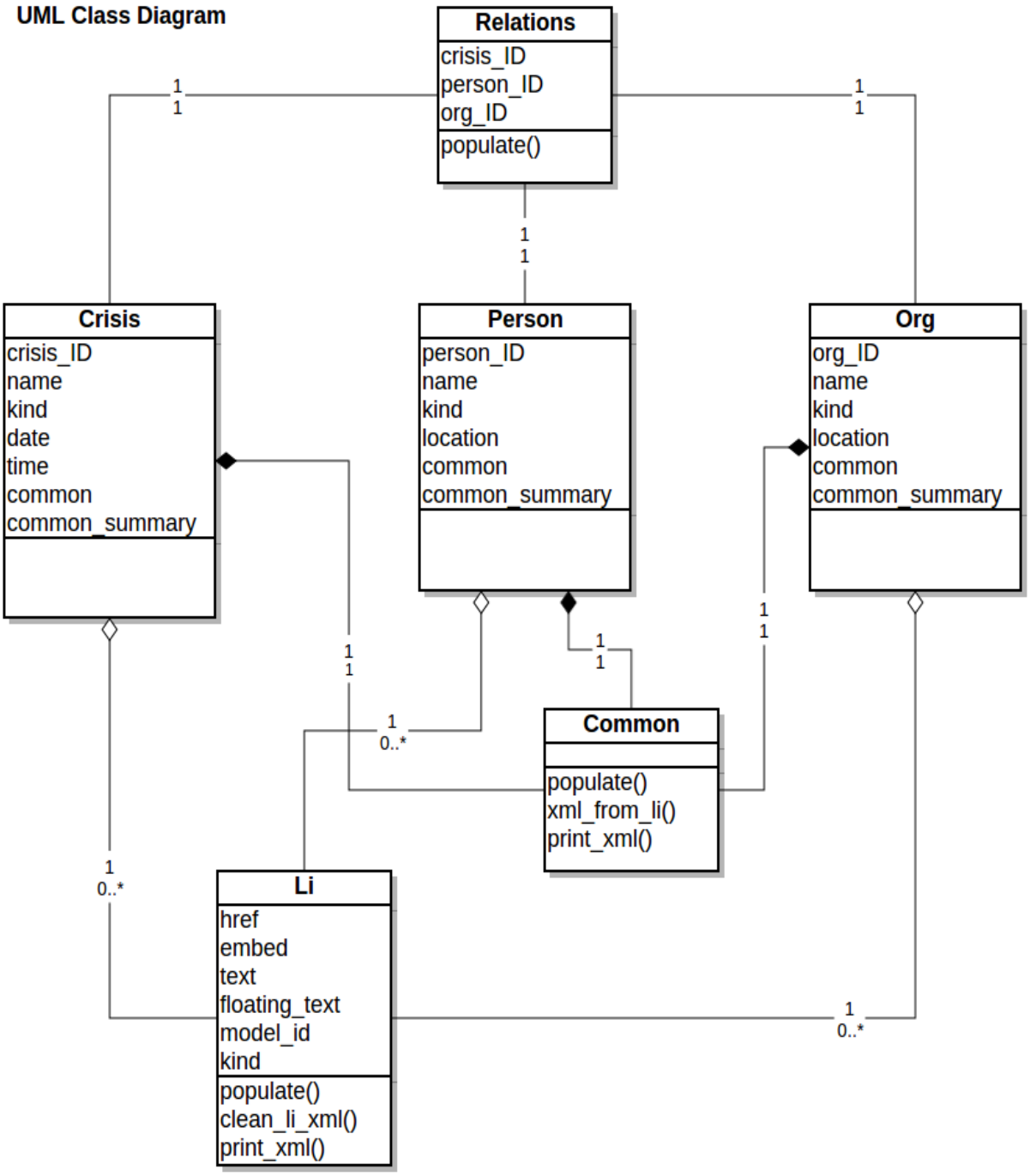
The MySQL database associated with Django does not store NULL values. Instead it simply puts in its own version of a "null" value. For example, columns of type CharField default to the following value: u'' . This has to be accounted for when reading values from the database for exporting - a check for a None object will not recognize the "null" value and this will result in the export function adding tags with no data associated with them. Similarly, it is crucial that the final string to be exported to a file gets encoded in UTF-8 format because the typical conversion to ASCII characters fails when a right quotation mark is encountered. This is because the CharFields in the database are stored and retrieved as unicode and thus, when the ASCII codec of a text editor attempts to convert, say, a right quotation mark with a unicode value of 0x2019 to ASCII, it throws an error because ASCII only handles values up to 255. This is a result of ASCII using one byte for characters, compared to unicode's four. Simply replacing the unicode value with the ASCII value for a single quotation mark is not acceptable either, because doing so results in the item in question being interpreted by the database as a new entry, should the exported file be imported, which results in duplicate entries for the same item in the database. By using UTF-8 formatting, the ASCII codecs become capable of handling the string properly, and this duplication issue is avoided.

Originally, the import and export facilities were tied together as there was no database to ensure data retention. With a database in the middle, the website is now more powerful but required a refactoring of the export facility.

UML :

The UML diagram below depicts the six classes and the relationships between them. Relations contain model IDs used to identify and link Crises, People, and Organizations, but do not contain objects. The association line indicates a relationship but not containment. One Relations object describes one relationship between two models, so the multiplicity is one-to-one. Each Crisis, Person, and Org model contains a Common object. The Common object's life is tied to the model containing it, so there is a composition relationship between the two. Li objects hold descriptive information about the Crisis, Person, and Org objects, but Li is a model with objects in the database that can

exist independently of models.



Heroku :

Heroku is a “hosting as a service” platform. Heroku allows a project with supported platform code, such as Django, to be pushed onto its servers. Thus, the WCDB project is able to have Heroku serve its content on a unique URL dedicated to the app. Heroku is used to serve the World Crisis Database app because Heroku allocates 740 hours of free hosting per month along with providing at least two free databases with a five megabyte data max or 5,000 table row max. This meets the budget and time constraints of the WCDB team within the summer semester and therefore provides a great alternative to the z server.

Every part of the project, thus all of the pertinent code is pushed to Heroku servers with ‘git push heroku master’. This is executed after making some changes to the project (to work well on the Heroku server), and by using Django 1.5 since a wsgi file was auto-generated. This file allows the app to be hosted by Heroku. Changes to the project structure for Heroku hosting included making the ‘wcdb’ app a top-level directory. Specifically, settings.py was modified to specify the postgresql database. The team used two postgresql databases, one for production, white, and one for testing, copper. A unique test runner was created after forum advice from stackoverflow:http://stackoverflow.com/questions/13705328/how-to-run-django-tests-on-heroku. In the beginning, Eddie created a nice wiki that details beginning the process at http://eddiesaenz.net/school/2013/07/14/heroku.html.

Heroku is a simple service to use. No htaccess file needed to be written. After a few form-fitting modifications and pushing the code to a designated Heroku remote through git, Heroku took care of the rest. It’s a simple service that is used widely across the professional world.

Front-End Implementation :

The front-end is realized through Django’s template system. The templates are written using HTML, which expresses presentation instead of a programming language’s logic. A template is a document that allows for HTML rendering, lightweight scripting, and expression of variables. Each page within the site is built using an HTML template file.

In the initial phase of this project, all the static pages were hard-coded to include the information about our crises, organizations, and people. These files used the information from the XML that was being imported into the Django models, but did not complete the link. In this iteration, templates were passed a large quantity of parsed data instead of having it already filled. These data were initially received by the Django models, but the models put the data in a database in the current implementation. From here, it is the templates’ job to present that data nicely.

All of the front-end pages run through the appropriate templates to create unique page content. Universal to this content is a default navigation bar which provides ubiquitous access to the home, import, export, and unit test pages. The crisis, person, and organization types each received their own template for dynamic rendering abilities. This means that the views for this content are designed by the system itself. For example, to create a webpage for a crisis item, the following is called:

```
def crisisView(request, crisis_id):
  """
  Renders view for crises.
  """
  crisis_dict = getCrisis(crisis_id)
  return render(request, 'wcdb/cri_temp.html', crisis_dict)
```

In the above example, a method added for the second phase's implementation named "getCrisis()" is called to return a dictionary of individual crisis data. This dictionary is of the form { ['name'] : crisis.name, ['kind'] : crisis.kind, ['date'] : crisis.date, ...} and is used by the html to extract information to display. This dictionary is the link between the back-end and the front-end as it creates a useful data structure for the front-end by extracting information put in the database by the back-end.

This dictionary is passed to the html templates by the render function. The render function returns an HTML response and renders the html page provided in the second argument. The call to crisisView() is initiated from urls.py when a url request is recived from the user for 'crisis/CRI_[A-Z]{6}'. The dictionary is an optional argument that is passed so that the template will possess top-level access to all keys in the dict. In the implementation, this is realized by the dictionary extracting values by just calling the keyname of the value from the dictionary. So there is no need to use the reference {{ crisis_dict['keyname'] }}, instead just {{ keyname }} will successfully extract the lower-level information.

The HTML extracts information by asking the dictionary to fill its variables. For example, a section from the crisis template (templates/wcdb/cri_temp.html) is as follows:

```
{% extends "wcdb/default.html" %}

{% block content %}
<h1>{{ name }}</h1>

<dl class="dl-horizontal">
```

```
        <dt>Date &amp;amp; Time</dt>
        <dd>{{ date }} : {{ time }}</dd>
        <dt>Location</dt>
        {% for loc in common.Locations %}
                <dd>{{ loc.floating_text }}</dd>
        {% endfor %}
        ...
    </dl>
```

Variables are written as {{ variable }} in the Django template system. This allows for dynamic capabilities since the template engine replaces variable values with the result of calling the variable. Templates can thus can be passed a range of values to create multiple instances of a formatted web page. In the example above, the dictionary entry for a crisis's name is the result of the variable {{ name }}. In the html view this is displayed as the crisis's name, demonstrating the dynamic web page generation capabilities of the WCDB website. When the template generator encounters a dot, attributes of objects can be looked up. This can be seen in the for loop over "common.Locations." In the implementation, for loops give access to lists stored in objects and create <dd> entries of the list elements.

One discrepancy that arose during development was the display of "href" and "embed" attributes as defined in the XML. The "href" attribute specifies a URL of a linked page. Images, feeds, external links, citations, and some contact information all expect urls set equal to an href in order to be displayed on the WCDB website. Videos and maps use the "embed" attribute which places external applications or interactive content into the webpage where it appears. If one of these categories is not specified with the respective href/embed attribute, the webpage does not display the content. The team had to adjust to this in phase two by re-writing XML to fit this specific scheme. If other teams do not adjust their XML in the same manner, this may pose potential problems in phase three when foreign XML is imported.

Bootstrap, CSS, and JavaScript were all used to support the front-end implementation. Bootstrap is a front-end framework that provides CSS stylings for common DOM (document object model) elements. In the WCDB website, the navigation bar and buttons are constructed with the bootstrap framework. Bootstrap is imported, the navigation bar is created, tables, background, font, links, and buttons are created by "default.html."

Bootstrap's framework rests on the framework of CSS (cascading style sheets). The website uses a detail-admin bootstrap wrapper that is made of more css. Bootstrap also relies on the JavaScript framework, though not much in the WCDB website. The team only used a script to add activating capabilities to pictures in the carousel. If all the pictures were active at the same time, they would all be displayed at the same time.

The WCDB front-end design is open for future extension. Web pages can be added by specifying a url regex in urls.py and adding a respective view function for that url. The view function can then render from newly designed templates or ones always present in the project. In the end, the HTML is filled when data is imported. By dynamically extracting the data from the database and asking the website to display all the information, the website is capable of displaying all imported data that fits the XML schema.

UI :

The User Interface (UI) is the space where human and machine interaction happens. This means that the user should be able to make sense of the feedback and behavior of the machine. This also means that the user should not be able to have enough access to break the system. The WCDB website implements a UI through the web pages generated dynamically when imported XML data fills HTML templates. Using a universal default template, the user can access the main pages of the website: home, import, export, and unit tests at any point of their interaction with the website. Links to pages within the website are displayed before any pictures and videos that the user would have to scroll through, keeping the user safely on the perfectly functional WCDB website.

The WCDB system should never crash and burn in front of a user. Instead, failures are presented to the user nicely through the user interface. Areas where the website can break are protected by case-handling tactics. The main place where a user can cause something to go wrong is in the import facility. For this reason, import rejects all invalid XML, no matter what and displays a nice rejection message to the user so they can try again. When no data has been imported, front-end presentation compensates by reminding the (admin) user to upload data on the homepage.

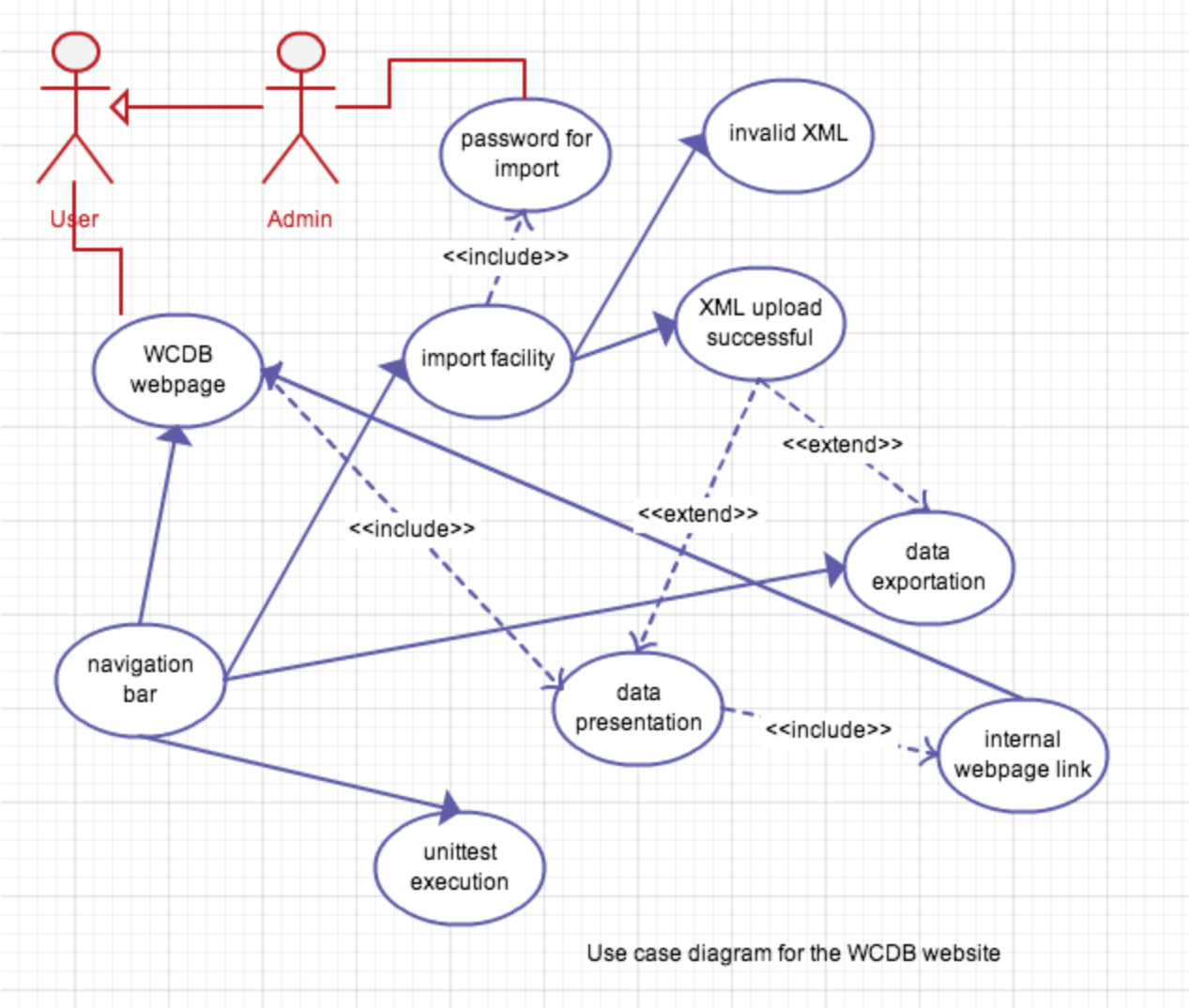
Use Cases :

There are two users to the system: administrators (admins) and regular users. Note that an administrator may also be a regular user. A use case begins when an individual visits any page of the website. By listing steps used to achieve a goal, the interaction between a user (“actor”) and the system is defined.

There is one use case that is particular to the admin user which is importing data through an XML format. The steps of this use case include entering a password (which excludes everyone but the admin), selecting an XML file to import, then receiving feedback as to whether the import was successful or not. Thus, an admin is a user with access to the site password ‘ateam.’ When a valid XML file has been accepted, the import page will

report ‘uploaded successfully.’ All other use cases are applicable to both admins and regular users. These include data discovery of crisis information, unit test execution, and data exportation. Every type of user may follow any of the embedded hyperlinks present on the website, which may include external websites as well as other web pages within the website.

For all use cases, it is important that the top navigation bar be present across all pages on the website. This allows for all users to direct to the main pages of the site: home, import, export, unit tests, and the several pages of crisis data. All users are directly interacting with the UI of the project for all use cases. The admin user indirectly interfaces with the back-end models when importing valid XML data, but receive no power to manipulate any part of that process.



Use case diagram for the WCDB website

Testing

Django's built in unit test framework is used to implement unit tests because of the added features Django unit tests added to the testing framework. These features, such as database model integration, are very beneficial to website. As a result of using of Django's unit test framework the test file is named "tests.py". Before running unit tests locally in a terminal or on a locally hosted server, make sure to run the command 'python manage.py syncdb' to sync the test database with models.py. To test locally in a terminal, run 'python manage.py test wcdb'' to run unit tests. Unit tests can also be run from a local server. To do this, run 'python manage.py runserver' then click the unit tests tab to test.

Testing Databases :

SQLite3 was used to implement unit tests, locally as opposed to MySQL. Although the user has permissions to create databases, errors still arise when the unit tests are run from the local server with MySQL. When run, the user is prompted to confirm creation of a test database. However, this prompt never appears on the local server, stalling the website. This is circumvented by using SQLite3 to create and destroy test databases.

On Heroku the same problem exists: no permissions to create and destroy a database. The solution to this problem was in creating an extra database and specifying a unique test runner. This solution was also explained in the heroku section of this document.

Unittests :

The command 'python manage.py test' is used in order to run unittests. Django automatically finds all the subclasses that start with the word 'test' in all of the files. django.test.testCase must to be imported in order to run tests updating database for each test, so the results of each test case would not be affected by other tests; thus, a new database is created for each time tests are ran and flushed for each test case. For testing purposes, SQLite3 database was used. Each function is tested separately with three different inputs on average.

The loadModelsCrisisTest class tests the functions of the import facility.These unit tests mostly confirm that models are properly populated and stored in the database. The functions populate_crisis(), populate_person(), and populate_org() expect the root of an ElementTree as a parameter and populate their respective model. After population, the

model is saved to the database. These functions are tested by creating a dummy ElementTree and feeding the root of the tree to the functions. Then, the functions from getDBModels are used to check that the information has been properly stored in the database. populate_models() simply calls populate_crisis(), populate_person(), and populate_org(). It is tested by passing the dummy ElementTree to it as a parameter. The getDBModels functions are again used to check that the models are properly saved. The functions populate_common(), populate_li(), and the relations model's populate method are similar and simply populate the li model, common class, and relations model, respectively. The ModelsCrisisTest tests the methods in the Django models defined in Models.py. They are all related to populating the class they reside in and are tested similarly to the import functions.

The unloadModelsCrisisTest class tests the export facility. clean_xml(), make_non_li_string(), make_li_string(), make_common_string() are auxiliary functions used for formatting. The tests of these functions check that they form the expected string. setUp() adds Crisis, Person, and Org objects to the database for testing purposes. export_crisis(), export_person(), and export_org() each get an object from the database and return a string with the values of its attributes. Using dummy objects added in setUp(), each of the export tests check one object with values for each attribute, one object where no attribute has a value, and one object where some of the attributes have values and some do not. Objects representing each of these test cases were added to the database in setUp().

export_xml() uses export_crisis(), export_person(), and export_org() to form an xml string. Its unit tests work locally when the content of the database remains constant. However, because they query the database for all objects and their output depends on the objects of the databases, the string returned will vary with the database, so export_xml() cannot be tested reliably across databases or even as objects are added or removed from the database.

Conclusion

The WCDB website makes it easy to upload and integrate information about new crises, people, and organizations, thereby allowing the website to document more and more of the world around us and remain adaptable. This is accomplished by using a strict XML Schema which allows the website to know how to integrate the information it is given via an XML document that is uploaded through the import facility.

The modern person is just waiting to be slapped in the face by a website. Usually a website's design directly influences the number of user participants it will attract. In order to

inspire knowledge of world crises, the WCDB provides a well-rooted design to display such information while making it easy for people to both access data they wish to see and explore the connections between various people, organizations, and crises. At the same time, this website gives them the ability to find ways to help and take action should they wish to do more than just become more informed.