Group Name: The A Team

cs373 Summer 2013

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 internet 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. To organize all of this data we designed an XML Schema which was centered around the crisis, person, and organization tags. The focus on these tags is reflected in the design of the Django models, where the tags each receive their own class to be represented as a model. Because our data was originally in an XML format we are able to accept other well-formed and valid XML instances that conform to our schema, thus our 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. We present our crisis data through a set of static pages with rich media content, and linking to relevant pages within our website.

Future iterations of our website may include persistent imported data, data additions through our import function, and dynamic rendering of all content. The import facility will allow for persistent data and data additions as it will probably be linked to a database. From there, the displayed pages will have to be reworked to accommodate newly added information. These features will make for an improved site because it will allow the public to add world crises that they believe deserve attention.

Design

XML Schema:

The final XML Schema we decided upon was inspired from the shared crisis spreadsheet used by the class to enter data. This spreadsheet includes data for crises, people, organizations, and even places. A class wide schema was created by Vineet where the required tags were crisis, person, and org. There are common tags among the three major tags mentioned which are: Citations, External Links, Images, Videos, Maps, Feeds, and Summary. These common tags are all optional; this is preferred because at times it is difficult to obtain certain information such a media feed for example.

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 Hildebrando Pascoal is PER_HPASSS. This identification system is seen throughout the other two tags as well.

Our contributions to the class schema include making the place tag unbounded per crisis tag. This was intended for crises that affect multiple parts over the world, deforestation for example. Unfortunately though, the place tag was removed because it was discovered place was not a required data. Our second major contribution was adding a description tag. This tag allowed adding human written text about the crisis, organization, or person that the tag was nested under. This later evolved into the many different common type tags that are present in the schema today.

We did not update to the latest shared class schema because of the change that was added: making the interconnected objects not required, i.e.: a crisis does not need at least one person and organization. We did not accept this change because we believe our crises should take advantage of this deep interconnected relationship. We hope this schema will be adopted class wide, but if not our adaptable code allows for the described change.

The schema is excellent because it provides the maximum amount of data to display. With tags for images, videos, text, etc. We provide the front-end of our website the most content possible. The ID/IDREF system of the schema will also benefit us in later iterations of this website because it provides the backbone for correct url linking in between the pages of crises, organizations, and people.

Directory Structure:

Our 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’.

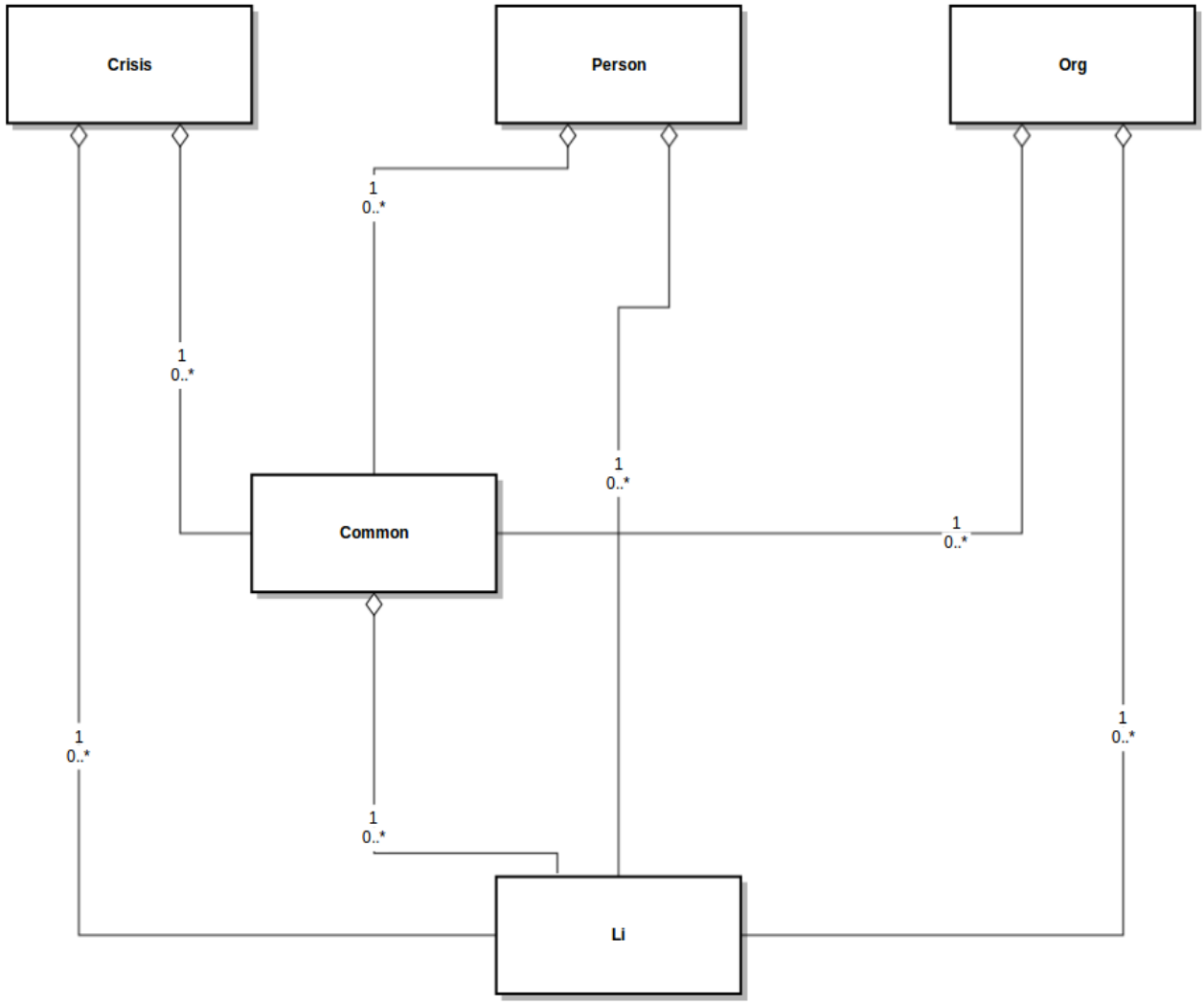
```
cs373_ATeam - our project directory
        ++-- minixsv
        |       ++--files for minixsv module
        ++-- genxmlIf
        |       ++-- files for genxmlIf module
        ++-- wcdb - our application directory
        |       ++-- static - our static files directory
        |               ++-- bootstrap framework files, css, and javascript
        |       ++-- templates - our template html directory
        |               ++-- wcdb
        |                       ++-- *.html - here live our html template files
        |       ++-- models.py - our models file
        |       ++-- views.py - our views file
        |       ++-- loadModels.py - logic to instantiate our models from xml
        |       ++-- unloadModels.py - logic to export xml from our models
        |       ++-- tests.py - where our unittests live
        ++-- urls.py - our controllers file
```

Django Models:

A model is a representation of a single logic unit which can contain data members, functions, etc. which is saved somewhere for example a database. The purpose of our models is to represent the crises, people, and organizations and the data like 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 we want to capture as much data from the schema instance as possible, so our models contain member variables that correspond to the data associated with the three main tags.

Our models do take advantage of two helper classes, Li and and Common. Li represents the list items that are seen throughout the schema and Common represents the ‘Common’ tag which is seen in the three required tags. We used this abstraction because they were common elements through the models and we believed this abstraction really helped in writing simple easy to change code.

If we had to support a schema change our models would update to add or remove appropriate data members in the model. In future iterations of our website our models provide a way to update a crisis, organization, or person “in place”, i.e.: an entire rewrite of the model is not required but the model can stay in the database and only be updated with appropriate information while the other information stays put.



Implementation

Import:

An import facility is a program that should be able to import a file, sent in by the user, to the Django models. Since we do not interact with the database on phase 1 on this project, we implement a workaround by implementing the models as classes. In phase 2, this design will obviously change as we will interact with the database. The user can upload a file from their computer to the website. This file is passed to the import facility, where it is first validated. If it does not have a valid xml extension, or if minixsv complains that it does not match our schema, the website will display an error message to the user to select another file. If we are passed a valid xml file, we use ElementTree to parse the file and populate our models. The main challenges of this portion were dealing with null cases, duplicate model populating, and verifying

with minixsv. Our minixsv issues are discussed later in the report. We use code similar to the following to populate fields of some model:

```
def populate_crisis(root, list) :
    """
    Function expects a node in an element tree and a list as parameters. Find instances of
    crisis in the tree and adds it to the list
    """
    for crisis in root.findall("Crisis"):
        temp_crisis              =              Crisis()
        temp_crisis.crisis_ID =        crisis.get("ID")
        temp_crisis.name     =      crisis.get("Name")
        if crisis.find("Kind") is not None :
            temp_crisis.kind     = crisis.find("Kind").text
        if crisis.find("Date") is not None :
            temp_crisis.date     = crisis.find("Date").text
        if crisis.find("Time") is not None :
            temp_crisis.time     = crisis.find("Time").text

        for human_impact in crisis.find("HumanImpact") or [] :
            temp_li = Li()
            temp_li.populate(human_impact)
            list_add(temp_crisis.human_impact, temp_li)
```

We discovered we failed to account for the optional setting of some of the models' attributes. Initially, we believed that if element's find method doesn't find a match, the loop would simply not run. This assumption was incorrect and the loop executed with a None value, causing the program to throw errors. The oversight was remedied with the addition of an "or []" before entering the loop. Thus, if the find returns a None, the loop uses the empty list and doesn't execute. When we ran our export facility, we discovered list attributes of our models were getting filled twice. Originally, we believed this was an error with our list_add function. A simple function expecting a list and some object as parameters. The actual error was the overarching import facility running twice. The duplicate lists actually ended up misleading us. The reason we didn't notice anything with attributes such as name, is that they are simply assigned. So when the value was assigned again with the same value, it wasn't noticeable for those values.

Export :

This phase of the WCDB project asked for the implementation of an export facility. This served as a proof-of-concept, asking the team to undo their work for the import and verify the output. This served two purposes. First, the team could verify the data retention of within their Django models by insuring that such information could be extracted from the models when needed. Second, the export acted as an implicit acceptance test by allowing comparison between XML in and XML out (which should be close to the same, maybe some different XML formatting).

In order to generate XML by accessing information from the Django models, the import and export facilities were first linked. Import ultimately returns a dictionary that is then picked up by export and unpacked accordingly. This dictionary had crises, orgs, and ppl keys whose values were lists of the perspective model instance. A unique export function was written to withdraw information from each model, designed around what information was needed to be extracted in each case. For data extraction within the Li() and Common() classes, a method was added within the class defined in models.py to handle the unique cases of printing from these data structures.

Ultimately, as specially designed functions withdrew information from the models, the XML was retrieved through string concatenation. If model instance contained relevant information, opening tags were concatenated to the string. This was easy since the Django models were written in Python also. Then the data found listed beneath such a tag within the models was extracted and concatenated before the run concatenated a closing tag. The export facility outlined the importance of data integrity, since it provided a full view of the exact contents of the models.

Minixsv Issues:

We at first had many issues with importing the minixsv module and using it correctly. We can not '$ sudo setup.py install' on the school machines to install minisxv, so we had to innovate. Our first approach was to just copy the appropriate directories into the Django project and use import statements where we needed the module. This did not work because Django could not find the minixsv folders even though they were in the project directory. We discovered that we could extend Django's installed apps to include minixsv, this way Django was able to find the appropriate minixsv directories.

Our second challenge with minixsv was actually using the tool correctly. It was desirable to use the 'parseAndValidate()' function because it allows you to specify what type of tree is returned. But the problem we encountered with using this function was it required file names for parameters. We could of course pass in the filename for the XSD file because it was static, but passing the filename to the uploaded file from the import page was not easy because it was in memory and not saved anywhere on the filesystem. We solved this problem by creating a temporary file that is saved to the same directory our app is in, we were then able to pass in the filename and be given back an ElementTree as a return value.

Front-End Implementation:

We took advantage of Django's template system. A template is a document that allows for HTML rendering, lightweight scripting, passing of variables to the template, etc. All of our front-end static pages use a default template which displays our navigation bar on all pages which allows you to link to home, import, export, and unit tests.

Our static pages are hardcoded to include the information about our crises, organizations, and people. Although we could have gotten this data from the imported files provided, we decided to save that feature for a future iteration of our project. In later iterations, templates will be passed a large quantity of parsed data, from the Django models, and it will be the templates' job to present that data nicely.

Testing

We had some problems initially because we did not set up Django correctly. We later made sure to ‘python manage.py syncdb’ which gave us the ability to run ‘python manage.py test wcdb’. We chose to use Django’s built in unit test framework because of the added features Django unit tests added to the testing framework. These features such as database model integration, will be very beneficial in later iterations of our website. As a result of our using of Django’s unit test framework our test file is named “tests.py”.

SQLite3:

Without the ability to create databases, we could not run unit tests. Our first approach was to use Brandon Fairchild’s solution of overwriting the method that sets up a database and running the unit tests without a database. After trying to implement this solution, we still got errors indicating our database engine settings were not correct and we did not have the credentials to create one. We then used sqlite3 to run unit tests. In the databases dictionary in settings.py, we changed the value associated with the ‘ENGINE’ key to 'django.db.backends.sqlite3' and the value associated with the 'NAME' key to 'ateam.db'. In the future, we will use the mysql test_cs373_ database. We only used sqlite because a mysql database was unavailable and we were unable to create one.

Unittests :

Unittest development was held up due to database issues involved with testing Django. Therefore, many unit tests were written with only hope that they would work in the end. Actually, when it finally came to testing most of the project had already been written sans unit tests. So going back to write the unittests was a slow and painful expose of minor kinks in our implementation. Let it be noted that becomes a project in itself to write all the unittests at once, they seem so small and innocent when they are written alongside development of the code.

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 design of this schema was based on a combination of commonalities noticed among the data collected and project requirements and is the result of a class-wide effort to reach a consensus on the schema. Our schema, however, is slightly different from the most recent class schema, since we appreciate the interconnectedness of model classes as one of the primary goals of this project.

Currently, the importing of the XML is not actually tied to the display of HTML pages, but this is a feature that is soon to be added. The static HTML pages provided were produced by

looking up information on different crises, people, and organizations, and using this information and HTML tags that we looked up on our own to construct HTML pages that display the information in an organized, and aesthetically pleasing fashion. Though in the future, we do hope to find a better way to incorporate our videos into the pages than simply embedding them vertically as they currently are. The static pages are hosted on a site called Heroku at the following link: http://immense-oasis-7538.herokuapp.com/ and can be accessed at this link without need for login credentials. We also attempted to host the website using Z at the following link: http://z.cs.utexas.edu/users/cs373/pvc95/django.wsgi/, but while it worked at one point, it no longer works correctly for reasons unknown.

We also built an import/export facility to allow information to be added and manipulated via XML documents that are regulated according to the XML Schema previously described. This facility is protected via password (the password is “ateam”) and takes in an uploaded XML document, checks to see if it is valid, returns a message if it is invalid, and adds the information in the XML to our django models. This provides some protection against invalid, incomplete, or malicious information and allows the information in our website to be altered as desired. The export part of the facility allows the retrieval of the information currently associated with our website in the form of an XML document, so that it can serve as a proof-of-concept for this phase of the project.

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.

*Documentation created using epydoc, not pydoc.