Car Ramrod:

- **Diane Lee**: Request Handlers, Django templates
- **Sogol Moshtaghi**: ExportXml, Unit Tests.
- **Tony Santi**: Static and Dynamic HTML, Website design, Being Awesome.
- **Wonjun Lee**: Model definitions, ImportXml and its Unit Tests, XML instance
- **Daniel Moreno**: UML, Technical Report Formation, XML Instance Data

In an ever growing information based world, the availability to knowledge on a world basis has become abundant. Specifically the information of crises, people, and even organizations from around the world can be recorded and shared with other beings across the globe in a matter of seconds. In such an intuitive world, access to such information can be better found through the formation of a database. unlike scouring the web for bits and pieces of lost information, a database is an accumulative source that allows its users to get info on a particular subject in its complete form, some even allowing one to contribute their own information for a subject.

The difficulty of formatting such a database is troublesome in the complexity of the problem itself. With issues such as design, availability, presentation, and intractability of a database, not to mention the complexity with the design of the coding for such a site and site functions, to take this project on requires methodical thinking and organization to accomplish anything on such a grand scale. It is this approach that our group has taken for such a large project, to break it down into smaller, more manageable bits of work to gain overall progress

on the database.

What one can do regarding a database is limited to that of what one has time to create; along with the time to create preventions for all issue that may arise for creating a new function. This limitation has been viewed with the development of the design of the website along with all of its functions, along with the case of taking in the XML instance and procuring it into the website format. No matter what preventions are made, other peoples' instances will not be the exact as our own in format, as each has their own web design which their own required data. For example, the embedding of videos for not all groups will have the same format of embeddable links within their instances, as such at merge conflicts with site formats are bound to occur in some cases.

Data Modeling

In order to create and store data models, WC2.py defines classes that represent the elements in the XML schema. These classes have the hierarchy as the corresponding XML types. Child models are first pushed to the data store, and then their parent model is created with db.ReferenceProperty attributes that refer to these children. Every model and their instance variables and children inherit from db.Model. In the following list of models, the name of each model is followed by the corresponding type in the schema. Each attribute is followed by its type in parenthesis.

- **Link** (extType)
  - site (db.StringProperty): The name of the site.
  - title (db.StringProperty): The title of the page.
  - url (db.LinkProperty): The URL address.
  - description (db.StringProperty): A brief description of the page.
- **FullAddress** (fulladdrType)

- address (db.StringProperty): The street address.
- city (db.StringProperty): The city.
- state (db.StringProperty): The state.
- country (db.StringProperty): The country.
- zip_ (db.StringProperty): The postal code. The underbar was added to avoid Python syntax error.

- **ContactInfo** (contactsType)
  - phone (db.PhoneNumberProperty): The phone number.
  - email (db.EmailProperty): The email address.
  - mail (db.ReferenceProperty(FullAddress)): The postal address.
- **HumanImpact** (humanImpType)
  - deaths (db.IntegerProperty): The number of human lives lost in the crisis.
  - displaced (db.IntegerProperty): The number of people who lost their home.
  - injured (db.IntegerProperty): The number of injured.
  - missing (db.IntegerProperty): The number of missing.
  - misc (db.StringProperty): Other interesting facts.
- **EconomicImpact** (econImpType)
  - amount (db.IntegerProperty): The amount of wealth lost or gained during the crisis.
  - currency (db.IntegerProperty): The currency.
  - misc (db.StringProperty): Other interesting facts.
- **Impact** (impactType)
  - human (db.ReferenceProperty(HumanImpact)): Human impacts.
  - economic (db.ReferenceProperty(EconomicImpact)): Economic impacts.
- **Location** (locationType)
  - city (db.StringProperty): The city.
  - region (db.StringProperty): The region.
  - country (db.StringProperty): The country.
- **Date** (dateType)
  - time (db.StringProperty): The time of day.
  - day (db.IntegerProperty): The exact date.
  - month (db.IntegerProperty): The month, expressed in an integer.
  - year (db.IntegerProperty): The year.
  - misc (db.StringProperty): Other interesting facts.
- **CrisisInfo** (crisisInfoType)
  - history (db.TextProperty): A brief history of the crisis.
  - help (db.StringProperty): Ways improve the situation.

    - resources (db.StringProperty): The resources needed to improve the situation.
    - type_ (db.StringProperty): The type of the crisis.
    - time (db.ReferenceProperty(Date)): When the crisis happened.
    - loc (db.ReferenceProperty(Location)): Where the crisis happened.
    - impact (db.ReferenceProperty(Impact)): The aftermath.
- **OrgInfo** (orgInfoType)
    - type_ (db.StringProperty): The type of the organization.
    - history (db.TextProperty): A brief history of the organization.
    - contact (db.ReferenceProperty(ContactInfo)): The contact information.
    - loc (db.ReferenceProperty(Location)): Where the lodge is.
- **PersonInfo** (personInfoType)
    - type_ (db.StringProperty): The type of the person. Usually his/her job is provided.
    - birthdate (db.ReferenceProperty(Date)): The birth date of the person.
    - nationality (db.StringProperty): The nationality of the person.
    - biography (db.TextProperty): A paragraph about his/her life.
- **Reference** (extLinkType)
    - primaryImage (db.ReferenceProperty(Link)):
    - images (db.ListProperty(db.Key)): A list of Link references to image links.
    - videos (db.ListProperty(db.Key)): A list of Link references to video links.
    - socials (db.ListProperty(db.Key)): A list of Link references to social network links.
    - exts (db.ListProperty(db.Key)): A list of Link references to external links.
- **Crisis** (crisisType)
    - idref (db.StringProperty): A unique id that is used to retrieve a specific Crisis from the datastore.
    - name (db.StringProperty): The name of the crisis.
    - info (db.ReferenceProperty(CrisisInfo)): Reference to a CrisisInfo object.
    - ref (db.ReferenceProperty(Reference)): List of various links.
    - misc (db.StringProperty): Other interesting facts.
    - relatedOrgs (db.StringListProperty): List of idref strings of the related organizations.
    - relatedPeople (db.StringListProperty): List of idref strings of the related people.
- **Organization** (organizationType)
    - idref (db.StringProperty): A unique id that is used to retrieve a specific Organization from the datastore.
    - name (db.StringProperty): The name of the organization.
    - info (db.ReferenceProperty(OrgInfo)): Reference to an OrgInfo object.
    - ref (db.ReferenceProperty(Reference)): List of various links.

    - misc (db.StringProperty): Other interesting facts.
    - relatedCrises (db.StringListProperty): List of idref strings of the related crises.
    - relatedPeople (db.StringListProperty): List of idref strings of the related people.
- **Person** (personType)
    - idref (db.StringProperty): A unique id that is used to retrieve a specific Person from the datastore.
    - name (db.StringProperty): The name of the person.
    - info (db.ReferenceProperty(PersonInfo)): Reference to an PersonInfo object
    - ref (db.ReferenceProperty(Reference)): List of various links.
    - misc (db.StringProperty): Other interesting facts.
    - relatedCrises (db.StringListProperty): List of idref strings of the related crises.
    - relatedPeople (db.StringListProperty): List of idref strings of the related people.

Frameworks and Tools

- **ElementTree** library provides functions to break the XML instance into elements.
- **Google App Engine** package provides db.Model and methods.
- **Gedit** was used to write Python codes.
- **CoreFiling XML Schema Validator** was used to validate the XML schema and instance.
- **Ubuntu Konsole** displays the debug results.
- **Notepad++** was used to write HTML codes.
- **Creately** was used to create the UML diagram.
- **Pydoc** was used to create an HTML documentation of the Python codes.
- **GAEUnit 2.0** was used to carry out unit tests.
- **Django Template** was used to embed the dynamic content from the GAE datastore into the HTML.

How a request is handled

When a user makes a request from their browser to see the website, Google App Engine invokes the main function in *WC2.py*. The main function then turns the request over to one of the handler functions depending on the path prefix (The part of the URL that comes after the domain).

There are 3 path prefixes that are handled specially: */import*, */import_upload*, and */export*. All other paths are handled by *MainHandler*.

Import

When the user clicks the import button, they are redirected to */import*. This gets sent to *ImportFormHandler*. *ImportFormHandler* outputs a POST form that allows a user to select a file for upload. The form is wired to store the uploaded file to the Blobstore, then go to */import_upload*, which goes to the *ImportUploadHandler*, which extends GAE's *BlobstoreUploadHandler* and can access the uploaded file. *ImportUploadHandler* calls the import function import_file to convert XML to models and puts them in the datastore, then displays a success message and refreshes the page so that newly imported pages appear in the side menu.

Export

When the user clicks the export button, they are redirected to */export*. main sends this request to *ExportHandler*, which simply calls the export function *ExportXml* and returns the output. The content type is set to *text/xml* so that the XML is displayed correctly.

MainHandler

MainHandler handles all other paths, including */crises/**, */organizations/**, */people/**, and the root. *MainHandler* contains a series of conditionals that checks the path prefix for the above cases, queries the datastore for the required data, and renders the page using the appropriate django template. To do this, *MainHandler* invokes the function *template·render* on a django template file and a dictionary containing the necessary data.

In the case of the crisis, organization and people pages, (located at */crises/**, */organizations/**, */people/**), *MainHandler* treats the remainder of the path as an ID and queries the datastore for a model of the type designated by the path prefix that has a matching *idref* value. This model is added to the dictionary with the key *me*. Most of the attributes that we will want to display on the profile page will be accessible through *me* from the django

template. However, a few of *me*'s attributes are references to other models and we need to query the datastore for their models and add them to the dictionary separately. This includes all of the Link types (*images*, *videos*, *socials*, *exts*), and the related page references (*relatedCrises*, *relatedPeople*, *relatedCrises*).

In the case that we are at the root page, MainHandler will render all of the main page elements, including the navigation buttons, the menu bar on the left side, and the iframe in which profile pages are to be loaded. In order to draw the dynamic page links for the left menu, django needs access to the names and IDs of the people, organizations, and crises in the datastore. *MainHandler* queries for a finite number of each of these and simply adds the three resultant lists to the dictionary to be provided to *template·render*.

Finally if it is none of the above cases, a "Page not found" message is displayed.

Import Solution

The import part of the project is handled by the function import_file along with some helper functions. The significance of importing is that data needs to be imported from a valid xml instance and be stored in a data store and therefore it has to import in the GAE models. import_file is called in ImportXml, which takes an .xml file as an argument, reads it, and calls the import_file. As such import_file takes the .xml file as its argument.

As a requirement for this project we used ElementTree which is a container object that stores hierarchical data in the memory. The data read from the xml file in the function

import_file is stored in a list (called data) consecutively in the order that they appear in the xml file (as objects). ElementTree is used to go over the xml file and parse it, as seen by: root.findall("crisis") which finds matching sub-elements of the xml file with the tag "crisis", puts them in a list, and returns the list. Using ElementTree for the purpose of parsing xml files is significant in that it has a tree like structure which makes it relatively easy to search or iterate through the data.

We go through the list which contains all the elements with a specific tag (person, organization or crisis), for each element of which, an object is built using the GAE models and constructors. The objects are built in a hierarchical order as the xml file is read. The elements are found and the objects along with the sub-elements form the objects based on the respective GAE model. These models are then put in the data store.

All the elements and the sub-elements form a single object (crisis, person or organization) with its attributes based on the xml file. The object is then appended to the list (as well as the database). This process is repeated for crisis, person, and organization and their respective objects are built, which is this is how the list gets populated. Note that the list is used only for testing purposes; as such it does not have anything to do with the database or the actual app.

We originally used a dictionary as our container instead of a list. The elements in that dictionary were accessed by their name which was also the key to the dictionary. The values to the keys were the objects' attributes. We thought that this dictionary was complicated to populate and to test, so instead we decided to use a list as our container and store all the actual objects in it consecutively.

Export Solution

The export found in ExportXml is needed to export data from the GAE models to create an xml file. All the GAE models are stored in the list (data) in the import phase. As such the function ExportXml takes the list “data” and returns a string in xml format. It does this by iterating through the list and checking the type of the object (thing) in the list, all within a for-loop. The attributes of the objects are derived from the object themselves and are each appended to a list which is later converted to a string. The order in which we append to the list should be consistent with the xml instance.

We originally used a string and concatenated all the attributes to that string. However, this appeared to be really expensive, so we decided to append to a list which is an amortized constant action. After the list is populated completely, we convert it to a string using the join operator.

Evaluation and Testing

Our solution is tested with unit tests for each function. The main functions to unit test are import and export functions since most of the magic happens in those functions.

As an example, import is tested by the function test_ImportXml_1 which takes the xml file as an argument like import does. The test then populates the list “data" by calling the ImportXml function. For testing purposes we assert that the list in a particular index contains the right element with the right attributes. As an example:

self.assert_(data[0].name == "Libyan Civil War")

self.assert_(data[0].info.type_ == "Civil War")

Asserts that the name attribute of the first element of the list is "Libyan Civil War" and the "type_" of the "info" attribute of that element is "Civil War".

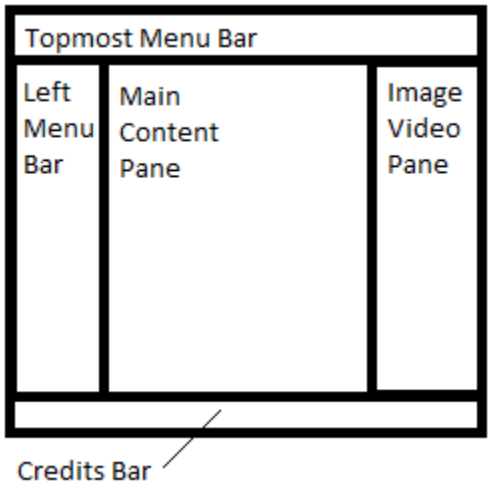
Testing the export facility is easy. In test_ExportXml_1, first we call import to form the data list and then we call the export with the data list as its argument. As explained earlier, Export returns a string. We assert that the string contains the right things and is identical to our xml instance.

The rest of the unit tests are for helper functions and their operations are trivial. Note that the reason that we switched to a list instead of dictionary was for unit-test purposes. It was hard to access the data in the dictionary and assert that the dictionary contains the right values.

HTML Design

The Front-end HTML of the project was mainly handled by Tony Santi, but Diane Lee and Daniel Moreno pitched in more than their fair share of legwork. The main goal of the HTML pages was to present a low-clutter, easy to navigate interface to the end user. A sketch by Wonjun Lee dictated the main framework of the site. It would be compartmentalized into several sections.

- The Topmost Menu Bar
- The Leftmost Menu Bar
- The Main Content Pane
- The Rightmost Image/Video Pane
- The Bottom Credits Bar



The main skeleton of the site is done with a messy combination of divs, tables, and iframes. The entire page is set to take up 90% of the browser width, and should be exactly 600px tall as to fit comfortably on any reasonable computer monitor when full screened. Originally the idea was to center the content vertically on the screen as well to support insanely large monitors, but the code to implement that assumes several things, the least of which is that no user will be using internet explorer. Suffice to say, that functionality was left out. Instead, there is a drop-shadow implemented in CSS. The Color Scheme was the result of no one having an opinion, but there haven't been any complaints yet.

"The Topmost Menu Bar" contains a link to the Homepage, a link to the XML importer, a link to the XML exporter, and a link to the GEA unit tests, and finally a search bar. The Home button's functionality is handled in HTML, while the import, export, and test are handled in the background working python code. The Search bar is merely cosmetic at this stage and will be implemented in phase 3 using regular expressions. Another feature worth noting is that none of the links mentioned change the main page of the site; they merely update an iframe that handles the main content pane and image/video pane.

"The Leftmost Menu Bar" contains three dynamic buttons, implemented with a

combination of JavaScript and CSS. The words "People:, Crises:, and Organizations:" are listed and they underline themselves when they are hovered over with a user's cursor. This is to imply that they are clickable. Should a user click any of them, they will find that each expands into a list of people, crises, and organizations respectively. Clicking on any other of the options will minimize the currently open "tab" and maximize the "tab" just clicked. The "On-Hover" functionality was handled in CSS, and the ability to toggle visibility of the sub-members of every option was handled in JavaScript. Some browsers like internet explorer do not like this very much. Each of the sub-links is also formatted using CSS to look a little nicer than the standard. Each link will change only the content pane and image pane as they are stored in an iframe to better compartmentalize the site as a whole.

"The Main Content Pane" contains the brunt of the data for the site. Every bit of information about the person, crisis, or organization is placed there and formatted to be as general as possible. The letters are slightly larger than average to avoid needless clutter. It is placed inside of an iframe of the main page to make it easy to swap out the data without needing to load the redundant sidebars. Each of the three center sections of the site has their own scroll bars so that navigation within a given content pane is localized. This makes hunting for data less of a chore. Internal links in the main content pane only change the internal iframe and leave the main page untouched, so the user will not see a URL change after clicking a link. External links open in a separate browser tab (unless the user's browser doesn't support tabs in which case it just pops up another window). A drop shadow is used in the main image of the Main Content Pane because they look nice.

"The Rightmost Image/Video Pane" is nothing more than a glorified list of images and videos. The width of each has been set to 220 in order to keep the look of the site consistent.

Each image has been flavored and linked together with some sort of light box derivative JavaScript code. Clicking on an item will make it pop into the foreground of the main iframe. It looks very pretty. For the static HTML pages the embeddable videos didn't seem to be a problem, but one the django templates came into the picture, YouTube just doesn't seem to want any progress made; more on that later.

"The Bottom Credits Bar" contains nothing of importance as far as content goes, but the splash of on the bottom as well as the top menu makes the site seem complete. Car Ramrod has reserved this space for inside jokes and comments than (hopefully) no one will read.

Django Design

The next phase of the project involved using django templates to dole out data based on the XML instance and the code from the python side of things. For the most part, very little changed cosmetically due to the migration the django templates. Most of the difficulty arose from schema that was chosen containing unobtainable data that is required to place in an XML instance. The most glaring example of this is the "birthdate" field. It not only required a day, month, and year, but also a time. Much of this data was unavailable, so the solution was simple; lots and lots of statements that looked like this:

```
{% if data.missingpart != None %} Missing Data: {{data.missingpart}} {% endif %}
```

Many of the required XML instance data has been left blank, which the XML parser of the Google App Engine treats as the Python value 'None.' Before writing any data, there is a

check made to see if the data is there in the first place, in which case it handles it accordingly. Lots of unimportant data is also completely ignored, such as the time of a person's birthdate. The following is the actually Django template code used to generate a person's birthdate information on the site:

```
{% if me.info.birthdate.day != None and me.info.birthdate.month != None and me.info.birthdate.year != None %}
<li><b>Birthday:</b>  {{me.info.birthdate.month}}/{{me.info.birthdate.day}}/{{me.info.birthdate.year}}</li>
{% endif %}

{% if me.info.birthdate.day == None and me.info.birthdate.month != None and me.info.birthdate.year != None %}
<li><b>Birthday:</b> Month {{me.info.birthdate.month}} of {{me.info.birthdate.year}}</li>
{% endif %}

{% if me.info.birthdate.day == None and me.info.birthdate.month == None and me.info.birthdate.year != None %}
<li><b>Birthday:</b> {{me.info.birthdate.year}}</li>
{% endif %}
```

This translates into, "if they have the day month and year listed, print them all, if they only have month and year, print those, if all they have is a year, print that. If they don't have any of that stuff, just skip it entirely."

The final hurdle with the Django templates is that they don't seem to like embeddable YouTube links. There may need to be a drastic change made to the way the videos are linked. In the static pages, the embedded films worked well, but there seems to be something unforeseen that is keeping the generated Django code from properly using the video data.

Django Templates

Currently there are 4 django templates. The template inheritance structure is shown below:

- maintemplate.html
- basetemplate.html
    - crisistemplate.html
    - organizationtemplate.html
    - persontemplate.html

maintemplate·html is responsible for the main page, which includes the navigation buttons, side menu bar, and iframe. *basetemplate·html* is the base for all the profile page templates. Our design takes advantage of django's template inheritance by placing style and layout information, and all attributes that are common to crises, organizations, and people, in *basetemplate·html*. We leave several blocks for the leaf templates, *crisistemplate·html*, *organizationtemplate·html*, and *persontemplate·html* to implement.

Implemented in base template	**Implemented in leaf templates**
• *name* • *type* • Link types - *videos* - *images* - *social* - *exts*	• *history/biography* • Unique info - *Location* - Contact info - etc. • Related pages - *relatedPeople* - *relatedOrgs*

	- *relatedCrises*

History and biography are analogous attributes, but because they have different names, they need to be implemented by the leaf templates.