<u>Technical Report</u>

MEMBERS: Dak Erwin, David Turet, Jung Yeol Kim, Keten Joshi, Raul Mireles, Syed Danish Irfan

In this project we created an IMDB-like website that focused its information on world crises along with key organizations and people involved. This website implements several different utilities, some of which are very practical from a user's point of view, the others are more useful from the developer's standpoint. However, to understand all of the functions, it is first key to know that the way in which the website's information was stored was in a relationship model database hosted by the Google App Engine (GAE). As a quick reminder, a relationship model database is one in which a main subject, called an entity, is made as its own table and has several attributes. For example, a world crisis has several attributes such as the name, kind, and location, all of which are unique to a crisis and as such are stored directly into the Crisis table. Now, say there were several organizations involved in helping this crisis. These organizations fit in another table that has the crisis to which they pertain to as a key name in what we call a one to many relationship; one crisis can have many organizations that helped it. However, this table would not be quite accurate. Sure, many organizations can help one crisis, but one organization can also help many other crises. In situations like this, we require a linking table called a many to many table in which the entries link each crisis that is helped by an organization and vice versa. More, in-depth reading can be found online, but now that our database structure is apparent, it is possible to briefly describe what the main functions of the project were.

First, we had the task of entering data into the Google Datastore (the official term for our database) which was done in two ways. One way to insert data was through a merge function which first tested to make sure the xml schema a user gave us conformed with the schema of our datastore, and if it did then it would store the union of the two pieces of data. The other option for entry was called replace, in which a valid xml schema was provided, the contents of the database were wiped clean, and then the information in the schema was stored, making it the only contents saved. While this function seems pretty silly to release to the public, it can be useful for developers in case harmful data was somehow put into the database so that it could be overwritten. The more user-friendly functions we implemented were export and search functions. In export, the contents of our database were printed out to the browser in an appealing xml tree. One could argue that this is, in fact, something also for developers, and while there is some truth to that, we have access to the database directly, so there is less need to actually see the schema. Then, in the search function, we searched through our database using the string provided by a user and printed out the matches in a sortable format, something rather handy when trying to navigate through pages on information. A more in-depth approach of these features are described later in the design section.

Due to the disheartening nature of this information, many people may ask why we chose to provide information on this topic. However, it is that very thought process that gave us our purpose. This topic was chosen to address the media blackout of any serious global turmoils after a few weeks. The fact of the matter is, even though there is no coverage of it on the television, the people of Japan are still seriously struggling from all kinds of issues, the environment in the Gulf of Mexico is still struggling from oil damage, and even the victims of the 2004 Indian Ocean tsunami are still facing daily troubles. It is our objective to bring these conflicts into a better light and to hopefully be able to raise some much needed support. Once we decided on the subject, it was pretty clear how to proceed from there. The internet carries information worldwide in a matter of seconds, and of several of the popular sites that are used, IMDB portrays a very excellent form mainly in how well it connect pieces of information with each other. From there it was clear to see how we needed to proceed, but that doesn't mean it was easy to do so. Just at step one: gathering information, the conflicts of this project were defined as an almost perfect metaphor. When gathering information of a crisis, we would eventually find data on the organizations that help and are helping to fix the issue. This meant gathering info on the organization, which in tern had more crises which it has helped and several key people involved in them. Thus, we had to research the people involved who also had other world crises and organizations they have helped. This format of unfolding issues carried on to building the database, all the way across the project to designing a pleasing website design. Still, once everything was put into perspective, that did not make it easy to do. All of us were working with the GAE for the first time, which took a lot of getting used to in the sense of how to morph normal python code to satisfy the design. Then, dealing with the actual datastore was a large issue. Between figure out how to implement the data models to learning how to actually insert the correct data, swinging all the way around to outputting that information on a dynamic web page, and then figure out how to make all of this more efficient, we had a lot to do in a relatively small amount of time, incidentally the largest limitation of this project and the hardest part to deal with. Since this was a school project, we had but two weeks to finish a very large slice of this project. At first we might have had a few schedule conflicts that prevented us from meeting, but by the end, our schedules revolved around this project. Even with work being delegated across six people, both the volume and complexity of the work took the entire time allotted to come up with a finish project. Of course, finish it we did, and the interesting part is how we did.

At the start of this project was an XML schema which allowed for a unified structure that all the groups' projects followed. By validating an XML instance with the schema itself, we knew the given instance could be incorporated into separate projects. Establishing this base structure also enabled easier additions to the project through features such as the merge. The agreed upon XML schema for this project called for separate instances for crises, organizations, and people. Within each of these tags, multiple <crisis>, <org>, or <person> instances could be added to <crises>, <orgs> and <people>, respectively.

With such a solid foundation for the structure of the information coming into our google app project, an import function was very practical. The first issue regarded parsing the information in a meaningful way. In this project, we used jQuery as our "modeler" for parsing the instance. The built in find() function searched for the first occurrence of the specified tag name. In doing this find, we could search for our desired tags within the scope of the tags that the parser was located. For example, the contact instance (in the following figure) contains a nested address instance with an attribute "name" common in both.

Fig 1 (from wc.xml)

```
<contact_info>
        <name>ICG/IOTWS</name>
        <phone_number>61396694103</phone_number>
        <email>r.bailey@bom.gov.au</email>
        <website>http://www.ioc-tsunami.org/</website>
     <address>
                <name>ICG/IOTWS</name>
                <street_add>700 Collins Street Docklands GPO Box 1289</street_add>
                <city>Melbourne</city>
                <country>Australia</country>
                <zip>0</zip>
        </address>
</contact_info>
```

A naive approach would involve treating the contact instance as an entity with the attributes of the address tag taken directly. However, the presence of the name within both levels creates a conflict in the find function. The approach we used, rather, treated address as a child of contact_info, similar to how we treated contact_info a child of org. Therefore we called find('contact_info'), collected the four desired attributes, and then find('address') to collect the four attributes of address (as seen in Figure 2).

Fig 2 (from Modeler.html)

```
$($(this).find('contact_info')).each(function(index) {
        contact_name = $(this).find('name:first').text();
        contact_phone_number = $(this).find('phone_number:first').text();
        contact_email = $(this).find('email:first').text();
        contact_website = $(this).find('website:first').text();
        $($(this).find('address')).each(function(index) {
                address_name = $(this).find('name:first').text();
                address_street_add = $(this).find('street_add:first').text();
                address_city = $(this).find('city:first').text();
                address_country = $(this).find('country:first').text();
                address_zip = $(this).find('zip:first').text();
        });
});
```

After collecting this model as well as the other models for an instance, we created a JavaScript Object Notation (JSON) model for a single organization. We then used the built in "stringify()" function to cast it as a string and added it to the list of organizations. This now let us hold on to the data in the xml file to build the database models.

In our our view file, we added crises, organizations, and people objects, along with all of their child objects into the datastore before handling the relational information between them. This was necessary because we ultimately only wanted to display the crises, organizations, and people on a given page if we actually have information on them. Our views.py made a request for the list of JSON strings and dealt with the encoding on the viewer side. Since the text came in as unicode, it was immediately cast as strings. We then unpacked the information in our instantiation of a model, attached it as a child to the given entity, and provided a key name before calling the put() function, loading it into the database.

When actually building the database structure, we opted for key name as our unique identifier for entities rather than an ID. The benefit of using key names was that rows in the datastore could not be added multiple times as opposed to an ID, in which adding duplicate data could still entered in the database since it would have a different ID number, even with the same potential key name. We used the convention of replacing empty space with an underscore "_" for our key names, which uniquely identified our entries in the datastore. For crisis, we had simple attributes for date, kind, and location, and created other models for any attribute with nested attributes. As seen in the History model used for an Org instance, we created Contact which has the reference property to an organization, its simple String properties name, phone (number), email, and website, as well as a child object Address, which contains the name, street_add, city, country, and zip. By abstracting these objects to different levels, the information remains relevant to the given property parallel to the XML schema. In order to keep track of the relational qualities of our crises, organizations, and people, we implemented 3 tables that act as junction tables: PersonCrisis, PersonOrg, and OrgCrisis. The key name for these were formed by concatenating the last eight characters of each key together to provide uniqueness.

In the final stages of this project, we needed to implement a search feature. A string search feature for a page has become so common that its presence is assumed on web applications. Intuitively, one expects to encounter a conspicuous input field on a web page, type in a word or phrase, hit the nearest HTML button, and be redirected to a set of results. Afterward, minimal scrolling or reading will be left for the user before arriving at his or her most relevant result. The web application's search box is embedded into the toolbar present on all the pages of the site. Four main components are required in order to move site functionality into the backend for the python scripting to implement logic to execute a relevant search on the user's input text: Input field name and value, form action, form method. An HTML form element is adequate to initiate the interface between user interaction and web application scripts and functions.

Fig. 3

```
<form action="/search" method="post">
	Search: <input type="text" size="22" maxlength="30" name="search_string"/>
<input type="submit" value="submit">
```

Form action mapped to "/search". The app.yaml file specifies that url requests of the pattern '/search' following the host name should be directed to scripts contained in views.py in the top level of the directory. Views.py holds class definitions for web request handlers. At the end of files in which web request handling classes have been defined, there tuple pairs with names of the url request pattern, and the class invoked to handle the incoming request. In the case of the search form above, '/search' maps to SearchPage . After a request has been lead into the appropriate request handler class, there are two methods which could handle the input: get() and post().

From inside the get() method of a request handler class, scripts can serve up a response and access GET arguments specified in the url following the '?' at the end of the url serialized by keyword var=value&var2=value format, or as input name and values from a form if the action is set to GET, though that is not as common or secure for information that changes or interacts with database contents. The post() method inside the SearchPage request handler class in views intercepts the form input coming in from the search form. The search string is the only input field; its value can be accessed in the post() method, and assigned to search_string variable with the self.request.get('option') method, the argument being the name of the input field specified in the HTML form.

Once the search string is held in a variable in the python scripts, collect_matching_models and check_match can be invoked which are top level functions defined in utils.py. These make use of search_string as an argument and can search for its presence within models in the datastore. The function, collect_matching_models, retrieves a list of models returned by db_getAllContents. Because of the limitations on the GQL queries, which cannot perform SQL-Like join queries, and only can only query one Kind(table, model) at a time db_getAllContents is a workaround which iteratively queries the database for all entires for the registered Kinds, which are models our application has defined for the datastore. After invoking the db_getAllContents function, collect_matching_models has access to a list containing all the models in the datastore. Each model in the list can then be examined for the presence of the search string in its relevant fields(strings, or string lists) with the check_match function which takes a model instance and search_string. A model instance retrieved from the data store has accessible attributes and data in model__dict__ and model_entity dictionaries which are more useful than explicit, one-at-a-time access via model.attributeName form, for dynamic and iterative processing. The check_match function will return a model instance where a regular expression match for search_string is found or None otherwise

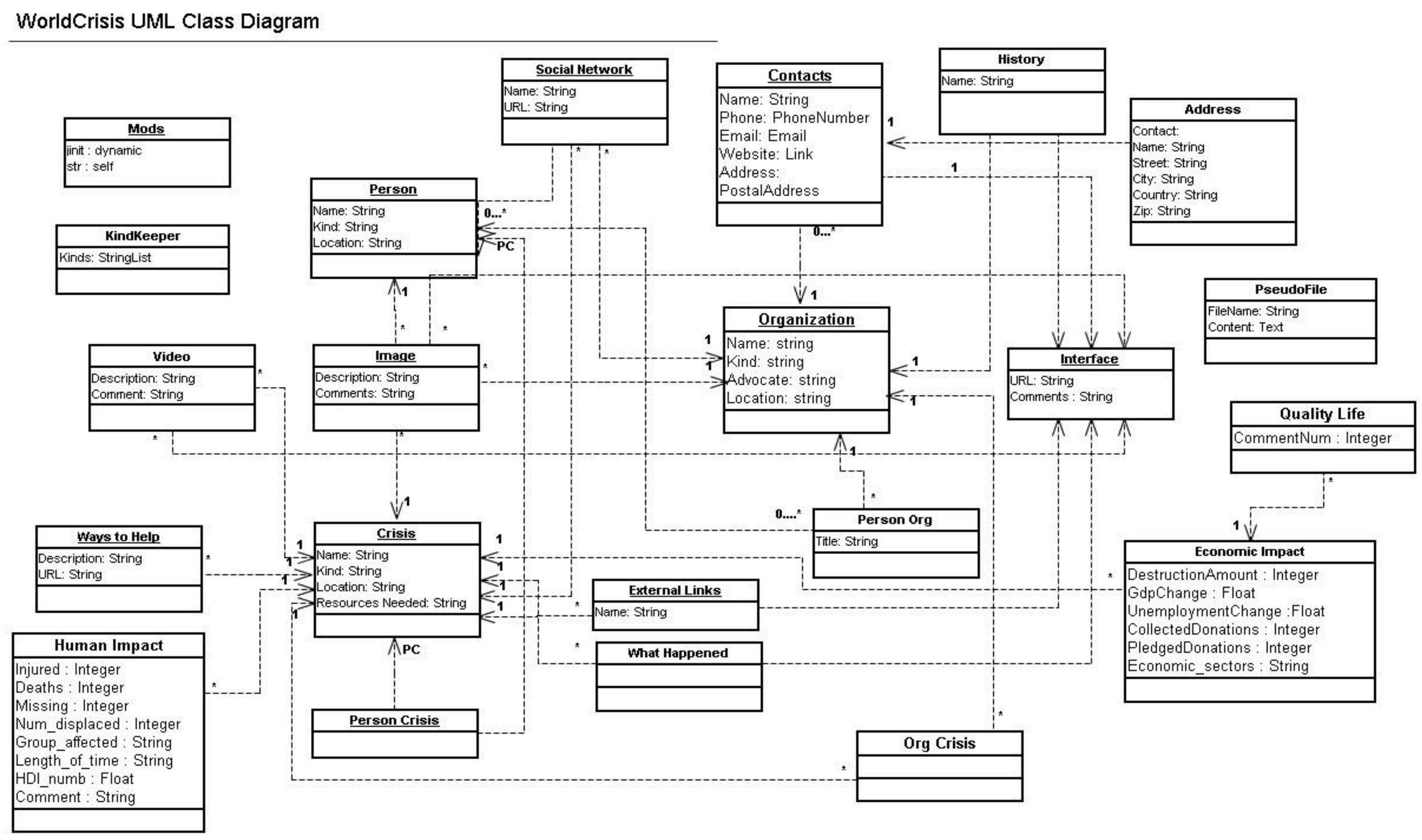
A reduced set of models which showed relevant presence of a regular expression containing the searching string remain. The reduced set of results is passed to the search.html template. A standard HTML table is built with these entries. We implement a DataTables jQuery library to render the results of our reduced relevant dataset. The DataTables library parses a table DOM element which we specify as our table of search results and enhances its appearance also adding multi-column sort on the reduced table set, and real-time text refinement of information presented in the table.

Leveraging the python memory and time spaces can become expensive, but considering the scope of our project, limitations on database refinements available via GQL, and simplicity of implantation this iterative regular expression search will suffice. Use of Javascript for additional functionality moves more computation onto the client's browser and ameliorates demands on the server's backend which also reduces use of the CPU cycle quota tracked by the Google App Engine hosts.

Another key part of the project was the UML class diagram. UML Class diagrams give an intuitive image of the internal structure of the whole project by defining what attributes each class contains and how they are being connected to the other classes. We used 'Gliffy.com' to create a UML diagram since the file was stored online and could be accessed and modified from anywhere.

The UML diagram contains all the classes from the 'model.py' and shows how each of these database models are connect to each other. The major three 'parent' classes were 'Organization', 'Crisis' and 'Person'. All others were considered children them. The diagram also shows the multiplicity of the relations between the two classes being connected together. A broken line connecting each class represents the implementation of the class to which the arrowhead ends. For the purpose of cleaning up the diagram a little, we introduced an **'Interface'** class in this diagram which contains the

URL and comments attributes. These two attributes are common to many other classes, making this 'interface' class made the diagram less congested and easier to understand.



A very key part to our project, along with any project, was unit testing. We divided our Unit Tests into three major files which contains these functions:

1. models.py

    the models themselves

2. utils.py

    encode

    instantiate_model_list

3. views.py

    search

    castImpactValues

    replace

    merge

    export

Our group mainly focused on the test of model related module which is models.py. This file had 15 classes for models such as Crisis, Person, Org, History and so forth. The main logic was to build model objects directly in the code, load those objects into the database, and run a GQL query to check their result. It was very difficult to test many of our files since they did not take any parameters, so we often relied on the GQL results to confirm the same data that we entered was given in the database. Of course, obvious confirmation of their success or failure would be provided on the website.

When we tested encode, we compared the input list to the output list since all the encode function did was to trim the 'u', which stands for unicode string, from the front of the values in the original 'structure' list and produced a new list without 'u'. Then we compared it as usual by 'self.assertEquals'. Another function under the utils.py is the instantiate model. We created a model_list using our XML page, then we put the list into the instantiate_model_list function which creates a new list object. Further, we created the expected object and compared these two attributes to the instantiated object attributes.

In views.py, the most important function was merge. We tested the first part of our merge function by creating incomplete model objects; then we created another incomplete model object and put both of them on to the database. Now we compared the merged object with the expected attribute value to confirm equality. In the second part of the merge function, we created the Parent and Child objects and merged them together and put it in the database. Then we compared their attribute (which will now be linked to one) with their expected values. For the Replace function under views.py, we created a number of objects and then put each object into the database. Then we did an GQL query on the result and stored the result in a variable called 'ik'. This fetched the number of objects stored onto the database. Then, we tested the length of 'ik". Now we ran the db_dropcontent () command to drop all the objects from the database, which was essentially the only different pice of code between the replace and merge. Now if we checked the length of ik again, it returned 0, meaning all the objects were successfully dropped from the database. For the export function under views.py, we tested it by creating an object and putting it into the database. Then we created another query to pull the data out of the database and compared it to the expecting assertion values. We used “self.assertEquals to compare the values of the functions coming from the database. On average, we created five test cases for each class under the Export function. We also created ‘Corner Cases’ which are empty cases to verify if our function works under no values.

One final, and difficult issue we faced in this project was the unit testing for our javascript parser. After implementing a few libraries for javascript unit testing, we found qUnit, which allows for function building in the test. For our tests, we built up functions for each test in the "setup" function that matched our parser in the Modeler.html. We were able to pass in XML instances that we could parse and show (on our /testjavascript page) that all of our tests found what we were expecting.