World Crisis Project: Technical Report

Team: Byte<Me>
Members: Calvin Szeto, Kevin Jacoby, Connor Bowman, Kevin Tang and Daniel Finan

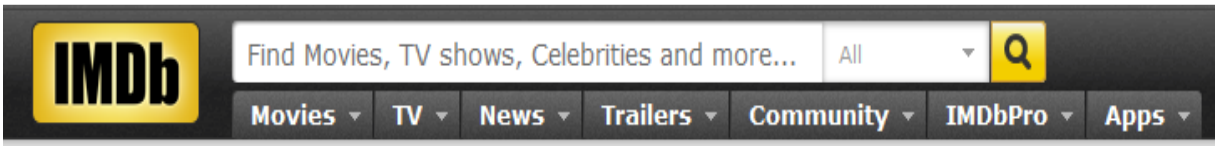
ABSTRACT

Online databases are like museums of information about a particular subject. For anything film related there is the Internet Movie Database (IMDB). There's even a database for board games called BoardGameGeek. One database that remains absent on the internet is a world crisis database. As a result, we were tasked with creating an online database specifically for world crises. We used the Google App Engine (GAE) as our framework and Django templates for our HTML pages. Our database is able to reference organizations and people who are associated with certain crises. It also contains a page for each referenced person and organization. We've implemented an import feature allowing the database to update new content in a matter of seconds. Import merge will allow future updates and additional information added to our database with an xml file. If time permits, we may add top 10 page viewed, current crisis news, and other improvements.

1 Introduction

1.1 OBJECTIVE

Our goal for this project was to build an online database about recent world crisis. Emulating the online movie database, IMDB, our website contains information and statistics on many different crises, along with the history and biographies on the organizations and people directly involved with these crises. The website contains three major kinds of pages: crisis, organization and people; which again will link to one another similar to how IMDB links between movies/shows and its actors. By using the Google App Engine we were able to build this website using Python, a language most of us had never worked with until this class.



1.2 DIFFICULTIES

With most of us being new to using Python for web design, we found this project to be very instructional. Our general unfamiliarity with Python was not the only issue. We were limited in this project in that we were required to use the Google App Engine to create our app. Some difficulties arose when trying to learn how to correctly use the Google App Engine to create models and implement the import and export utilities.

1.3 ADVANTAGES

An interesting aspect of this project is that it involves some real world application. Whenever we are able to make something that is relevant, like our World Crisis website, the project becomes more than some boring assignment. It became a lesson of how to understand, adapt, and utilize the new technology that generations past has to offer in which our generation has built and improved upon.

Furthermore, we got to work in an exciting team environment which allowed us to accomplish many things in this project in a few weeks that would, otherwise, have taken much longer to complete individually. We also gained some real world work experience by working in groups. Not only did we finish faster than we would have individually, but we were also able to apply some of the extreme programming strategies that we have been learning in class. It was a great experience to learn how practical and helpful the strategies we have been studying are.

2 Research

2.1 DATA

For our database, we chose to research The Deepwater Horizon Oil Spill with BP as the organization and Kenneth Feinberg as the person involved. We also did polio where the Global Polio Eradication Initiative and Jonas Salk contributed to the continued prevention of outbreaks. We found childhood obesity to be an interesting crisis to research since it is a consistent problem in need of a solution. Michelle Obama and the Academy of Pediatrics are taking the initiative to keep children healthy. Lastly, we decided on researching about the Greece Debt Crisis. We found that the International Monetary Fund has a stake in the issue since Greece is one of the largest borrower of the fund. As the Prime Minister of Greece, George Papandreou is the person we chose for the debt crisis.

For each crisis, organization, and person, we need to provide basic infos, a summary, and references to videos, images, and related external links. Our data had to follow the schema. There were extra info that the schema allow but not required.

2.2 TOOLS

Not only did we research crises, organizations, and people involved, we had to look up ways to make the website possible. This means that we needed to find a data storage model that meets our needs and tools that will make our implementation easier. First we see if datastore from the GAE fit the job but it gave us complication during implementation to our database. We decided to use blobstore python API from the GAE instead. Blobstore allow our data objects to be larger than those allowed in datastore. So we can now upload large files such as video and image files. Since we must create an import export mechanism, blobstore will allow other users to upload more videos and images.

Another tool that we had to learn was gliffy. Gliffy is an online tool that allow the user to create many different types of diagram. For our case, we used gliffy to create our UML class diagram. With gliffy, we were able to show the relationship between each of our classes of our model. One thing that gliffy did not provide us is the ability to show the multiplicity between each relationship. For that, we needed an image editor what allow word input on the image. A program called GraphicsGale did the trick. GraphicsGale is mainly designed for sprite artists but we've adapted the program for our purpose to add the multiplicity between each relationship in our model.

For the framework itself, we used the Google App Engine. It's free and it allow us to store our data with the engine's datastore allowing us to create the import and export function.

The Django template language for python allow use to make our design a reality. It let us change our static html pages into dynamic html pages. One feature that Django has is automatic HTML escaping which help protect our code from malicious imports. Django allow our html implementation safe and less of a hassle.

2.3 DEVELOPMENT

Our project was broken down into three phases:

In phase one, we began our research on our four main crises, and the organizations and people that were involved. With all the research we've gathered, we then built our XML instance that validates our XML schema we were using. Phase one also required us to begin designing a set of GAE models to represent the data we would parse out of our XML file. The bulk of this phase consists of building and testing the import and export function which will be used to read in different XML files to our database models. Lastly, phase one required us to set-up static HTML pages, which will become dynamic in phase 2.

In phase two, we first needed to revise our GAE models and our import/export functions to fit the new class proposed schema. Additionally we needed to import data into the GAE datastore. After that we needed to convert our static HTML pages to dynamic pages to make it user friendly. Then, with the use of Django templates, we would be able to embed our dynamic content from our GAE models directly into our HTML code. Along with writing more tests for our code, phase two required us to build a UML model for our DB classes and to begin to write a technical report of the entire project.

In phase three, along with the final touches to the website's appearance. We have to include a search bar that can search the crisis, organization and people that have been stored in our database. We also are required to implement a merge option when a user imports a new XML file. This merge will avoid duplicate elements being added to our database.

3 Design

3.1 XML

The first step of the project was organizing the content which would later populate the website. Since the website needed to be able to import content via an XML file, the logical tactic was to create an XML schema which would be used to validate

imported XML files. Our goal while creating this was to build a generic schema that could then be adapted to any similar XML schema, in case our design wasn't picked to be used as the class schema for phases two and three of the project; we accomplished this by basing our schema heavily off of another group's schema. The final schema was fairly straightforward: the data was split among crises, organizations, and persons, with some attributes further grouped by similarity, such as location attributes or date attributes.

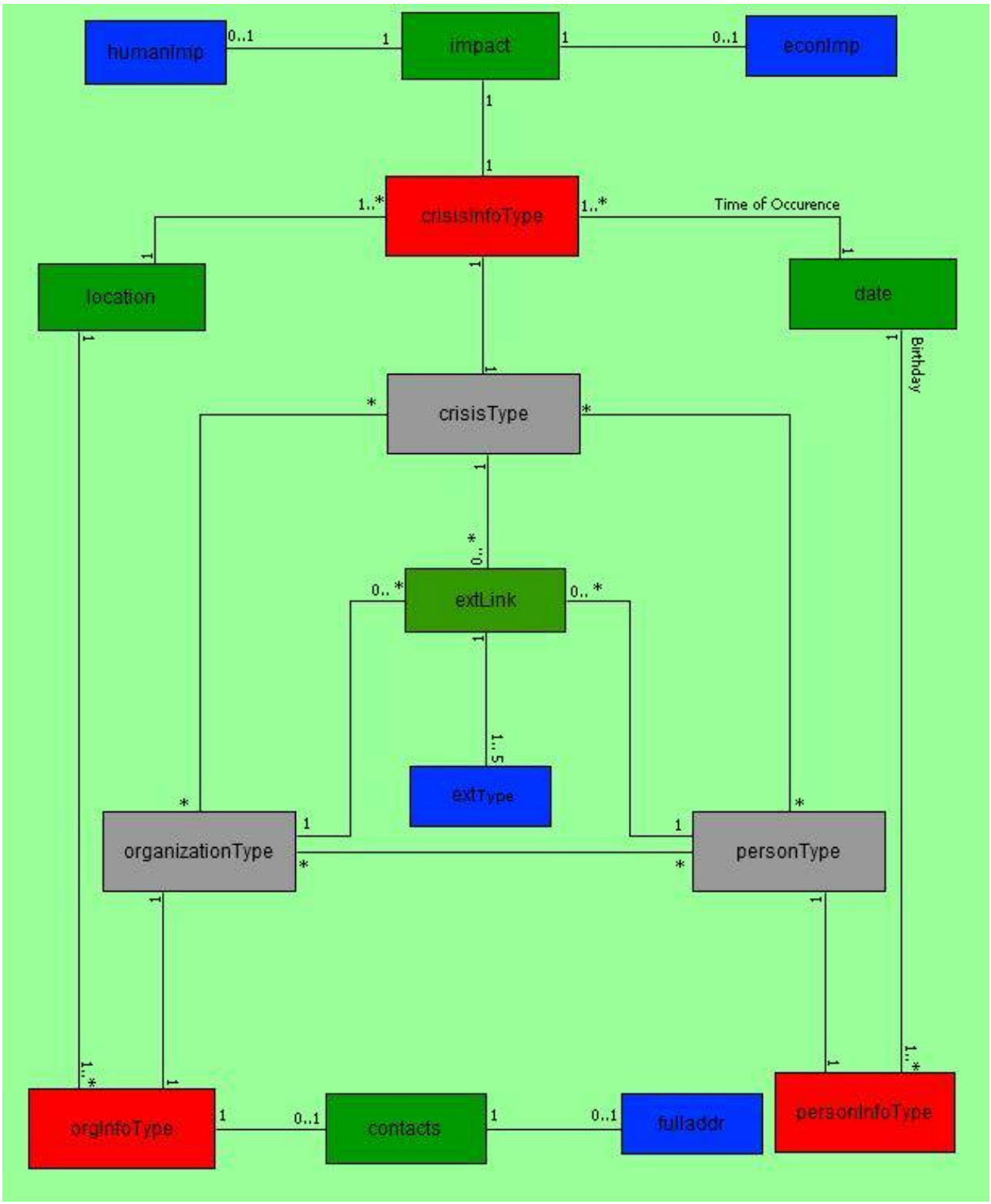


Figure 1: *xsd Class Diagram - This is based on the chosen schema*

3.2 MODELS

On the Google App Engine side of things, the data needed to be stored in Model objects, and thus our next design task was deciding how the data would be organized in the form of database models. The XML schema greatly influenced this phase. Although some of the intermediate types, such as Impact, were unnecessary, the database models for the most part reflected the types in the XML schema. Additionally, the InfoTypes in the schema were consolidated into one Info Model in the Google App Engine, since Crisis, Organization, and Person Models would only access their respective attributes. After implementation and testing, our final list of Models was:

- Crisis
- Organization
- Person
- Info
- Contact
- Location
- Time
- Mail
- Human
- Economic
- Ref

The most difficult decision in designing the Models regarded whether to have many specialized Models as opposed to three centralized Models (Crisis, Organization, Person), with most or all attributes owned by those central Models. The cost of having specialized models is that database calls are expensive, and every additional Model increased the number of required calls in the implementation. On the other hand, centralized models are complex when the number of attributes gets high. Ultimately, we decided on specialized Models because they made implementation much easier,

allowed us to consolidate repeated attributes, and there was no necessity of having a scalable website.

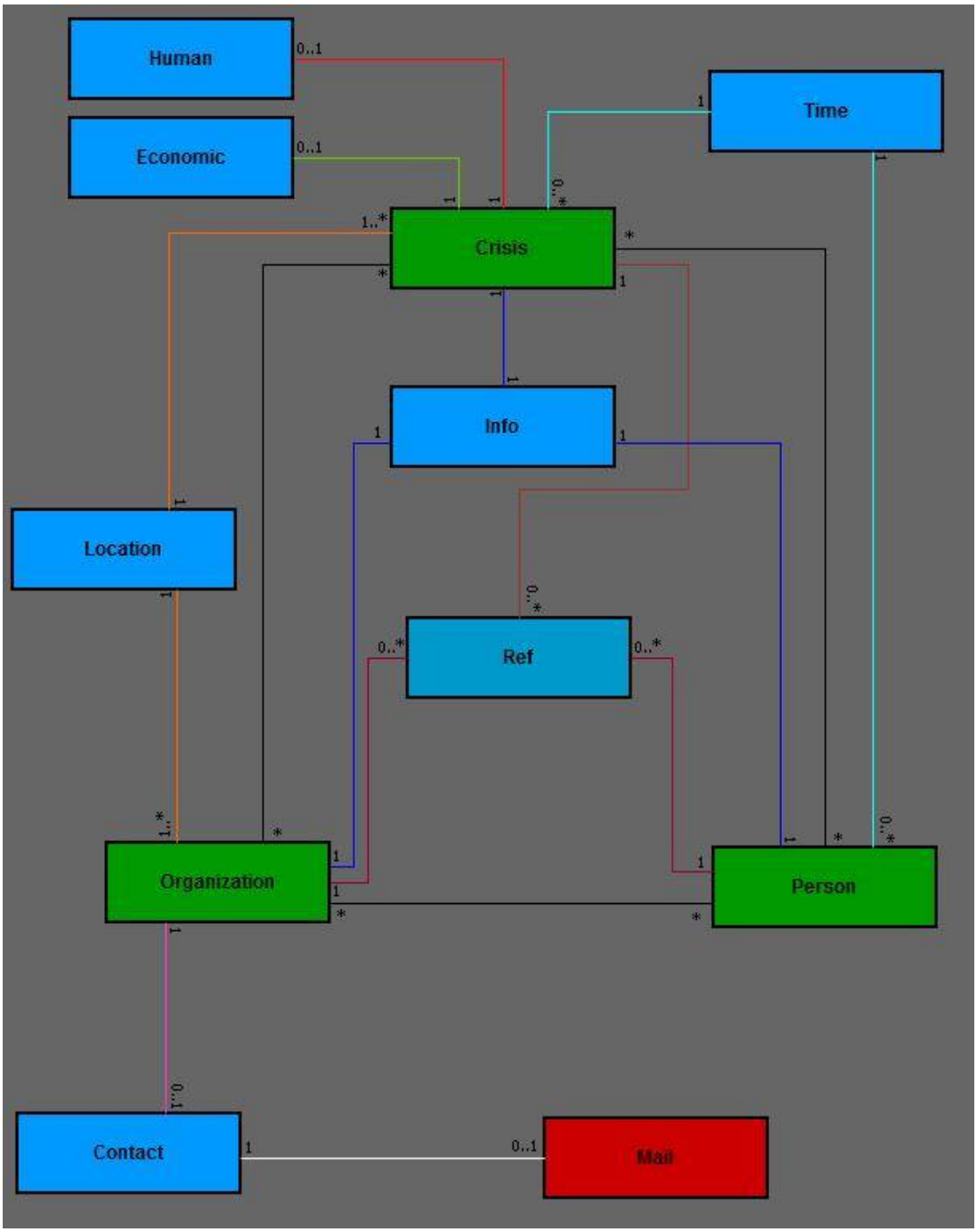


Figure 2: *UML Class Diagram of our model*

3.3 USER INTERFACE

Our goal in designing the user interface of the web pages was to create a visually stunning, intuitive website which effectively displayed the given data. Our design is based mostly off of what the group Cache Money did in the Spring 2012 semester. We really liked the slide show on the splash page. We found it to be more visually appealing than any of the other groups.

4 Implementation

4.1 GOOGLE APP ENGINE

The implementation of our project relied heavily on creative use of a handful of powerful tools. The most important tool was the Google App Engine, which provided a way to build and host our web application, but also vital to our success was the use of Django, Python, and the ElementTree module.

To begin, Google App Engine directs users using an app.yaml file, which acts like a phonebook for our files and directories. As such, our implementation files were stored in the root directory, with HTML and testing files stored in their own subdirectories.

```
application: calvins-cs373-wc
version: 1
runtime: python
api_version: 1

handlers:

- url: /validate
  script: src/validate.py

- url: /test.*
  script: src/gaeunit.py

- url: /import
  script: src/helloworld.py

- url: /importmerge
  script: src/helloworldmerge.py

- url: /export
  script: src/goodbyeworld.py

- url: /upload
  script: src/helloworld.py

- url: /mergeserve.*
  script: src/helloworldmerge.py

- url: /crisisserve
  script: src/helloworldmerge.py

- url: /orgserve
  script: src/helloworldmerge.py

- url: /personserve
  script: src/helloworldmerge.py

- url: /search
  script: src/testQuery.py
```

Figure 3: *app.yaml - A Google App Engine configuration file that specifies versions and handlers*

4.2 IMPORT

As the most implementation-heavy feature of our application, the import/export functions rely on the most tools. From the website's splash page, the import and export

links are directed by Google App Engine to an import and export Python script, which handles the acceptance of an XML file and parses it with the help of the ElementTree module. Next, the parsed data from the XML is taken apart and turned into Python objects which represent the datastore Models designed earlier. In order to make the process concise and efficient, Python dictionaries are used to retrieve model attributes in short, readable loops and unpack them into Python objects. Finally, these objects are placed into the Google App Engine's built-in datastore to be used for later.

4.3 EXPORT

For export, the key tool in use is Django templating. By creating a generic XML template that matches the predefined schema, exporting becomes as easy as pulling the Google App Engine Models stored earlier and plugging them into the template. Implementing this wasn't as easy as it sounds, however, as Django only supports a limited subset of Python functionality within the template itself. The solution is once again Python dictionaries, which, after being filled out in a preprocessing step, allow the template easy access to each Model and its children objects.

4.4 IMPORT MERGE

Import merge, which was implemented in the third phase of the project, is a more powerful version of import which allows the website to handle increasing amounts of data. The first difference is that import merge does not clear the Datastore like regular

import does; this allows the Datastore to grow as more data is imported over time. Secondly, in anticipation of large amounts of data being imported in a single XML file, the import merge feature makes use of the Google App Engine's Task Queue module.

The Task Queue allows import merge to validate the XML and return to the splash page, with the actual import left running in the background. Thus, the user is not left waiting on large XML files being converted into Datastore models. Instead, the creation of each model is added to the Task Queue as an individual transaction and the new models populate the web page as they are created.

4.5 SEARCH

Implementing the search function in our code relied heavily on the Google App Engine's included Search module. To begin, each model from the Datastore needs to be converted into a searchable Document object. Although initially this was done at the end of import, we moved the process to repeat at the beginning of each search query. While this seems time-costly, it was necessary because the new import merge feature did not guarantee all models would be ready in the Datastore before the Documents could be created.

After Document construction, the actual query is run. The query string itself is pulled from a Django template, then run on the search Index twice - once for an AND search and once more for an OR search. The OR query was made simple using Python's regular expression module: this allowed us to insert the OR keyword easily into the query.

The results from the query are processed and displayed using a Django template, as usual. However, we encountered one difficulty: the appearance of duplicate results, especially results in the OR search that were already included in the AND search. The solution was to create a list of id_ref values, which is updated and checked at each result. Any duplicate results are thus not displayed.

4.6 TESTING

For testing, we use the built in gaeunit from the Google App Engine to run our unit tests. We created many tests for each function. Outside of unit testing, we also simply test our implementation by opening the web pages to see if it shows what we expected.

4.7 INTERFACE

Finally, for displaying the content, Django templates are once again the vital tool. Similar to the export function, the content is pulled from the Google App Engine Models, processed, and sent to an HTML template which plugs the data in. The final product is a consistent look and layout among all the separate web pages while also being efficient, scalable, and easily adaptable to changes.

When we first started planning out how to design our database models, we initially tried building three root models (crisis, organization and person), with each model containing multiple attributes holding the information we had parsed from our XML file. After filling out the first crisis model with respected attributes we discovered that the remaining root models used many of the same attributes that we just used in crisis. We then decided to change our approach by making these three root models become parent models to multiple child models that held these redundant attributes we continued to parse throughout these root pages.

5 Evaluation

To automate testing of the Google App Engine, we used GAEUnit. We copy the gaeunt.py file into our directory and added a test URL to app.yaml. this allowed us to run our unit tests in the GAE app server environment using a web browser.

We found it difficult to write tests for our parser before we started writing our code. Once we started manually parsing through our XML file, we ran many tests as we parsed to make sure our parser was reading what we expected it to read. Then we used the ElementTree class from the python library which made our life a lot easier. The

functions in this class made parsing the XML files a lot easier than we expected.

> xml.etree.ElementTree.fromstring(*text*)
>
> Parses an XML section from a string constant. Same as XML(). *text* is a string containing XML data. Returns an Element instance.

Using the above function made testing our parser a lot simpler. We could take a section of our XML code and run multiple tests on our import and export parser using the fromstring function the ElementTree class provides.