TECHNICAL REPORT

World Crisis project

University of Texas at Austin

Computer Science 373

Summer 2012

All the Single Ladies

Fayz Rahman: Front End of the Website

Ian Buitrago: Source Controller and Merge Import

Robert Reed: Import and Export

Ross Shwarts: Search and Research

Jake Wilke: GAE Handler

EXECUTIVE SUMMARY

Every year the lives of millions of people across the globe are significantly affected by various widespread crises. The effects of these tragedies are almost always long term, and there are usually organizations that continue to provide aid for as long as it is needed. Yet, the media will cover a crisis extensively for only a few weeks before another topic comes into vogue and the crisis is no longer mentioned. Unfortunately, the public consciousness in the developed parts of the world has a short attention span and the people affected by these crises are quickly forgotten by the individuals most capable of providing the resources needed. As a consequence, these organizations begin to lose funding once news sources move on to the next "hip" topic, while the effects of the crisis continue without the support needed.

The goal of this project is to build a website to track and aggregate information about recent and ongoing crises, the people affected, along with information about the organizations involved. This is an effort to keep these events and the people affected in the peripheral vision of the public consciousness. Accordingly, the site does not only contain a compilation of data on the events that occurred, but also provides research on the current status of the people, places, and organizations involved. More importantly, it provides links and contact information for those organizations involved in providing aid. In this way, the site goes beyond just raising awareness and is extended to providing concrete information that every visitor to the website can use to get involved. We have also provided information on key public figures that are involved. This is to provide role-model-like examples that users are already familiar with (chiefly for motivation) or, in the case of militant and terrorist crises, to give information on the people behind the atrocities that have been committed.

In keeping with our aims, the site emulates the IMDB website design in that each page has information about a single crisis, organization, or person including statistics, summarized data, pictures, videos, and links to external sites that can assist users in getting involved. Every page also links to related pages on our website. So, if a user is looking at data on a particular crisis, say the earthquakes near Port-Au-Prince, Haiti, the page contains a link to every organization contained in the database that is involved in the rebuilding of these communities and, accordingly, each of these pages contains information on the work that organization does and its contact information and, of course, a link leading back to the earthquakes.

To import data into the site, we used an XML instance, which must conform to a precisely defined XSD schema. This includes support for updating and merging data for entities already stored in the Datastore. After being imported, the new data is then parsed using the ElementTree interface, and then stored into Google App Engine (via Datastore models). The export feature displays an XML instance in the browser that allows the user to get access to all of the data and links stored in our website. The data in any instance file that does not conform to our schema will not be imported into the website. There is currently no device to insure the correctness of imported data.

Of course, the site needs to be both attractive and easy to use. Navigation must be intuitive and each page should be compact and free of clutter. We did not want any part of our presentation to become a barrier to the information we are providing. To do this we used page templates to display the information contained in the Datastore in a consistent fashion. This allowed us to make the site update itself dynamically as data is imported, but it also made things cleaner, which is allows for a more attractive website. We implemented a search feature, which is an essential component of any website, but it is especially important for our goals. What good is a database of crises if a user cannot easily find the particular crisis in which they are interested? Additionally, we provide browsing tools for each of the page types.

To implement this website in the most efficient and economical manner possible, we used a number of free tools, chief of which is the Google App Engine, which served as the host for our site. The majority of the code is written in python (via GAE's Python API). Schema validation was accomplished using Minixsv. Python's ElementTree was used to parse and write XML files, allowing for relatively easy import/export of the data into the GAE Datastore, which in turn provides the website with easy access to our data. The data models we used and the relationships between them were carefully designed in UML with the help of Gliffy to display a readable class diagram for quick reference throughout production. To put the information on the website, Django templates were used to dynamically generate the HTML pages from the data in the Datastore, which can then be displayed by the Google App Engine. GAE also provides a search API for implementing our in-site search. Finally, we used GitHub to coordinate our efforts and keep track of all project issues.

Finally, we needed to guarantee that our site worked as intended. The import/export feature was the primary target of our automated unit tests. For these we used snippets of an XML document that we could paste together in different combinations that would pick out the various corner cases involved in importing schema-verified data. These tests were then embedded into the GAEUnit framework. Acceptance tests, mostly done by hand were used for almost every other part of the site. The most difficult part of the project to test have been the Django templates, which have required us to walk through the various pages generated to make sure that all links and videos display correctly and that photos are being embedded and resized correctly, which did not always display similarly on different browsers. Also, we had to make sure all design decisions were implemented.

As for the effectiveness of our solution, that is difficult to measure. We hope that our users will get involved. Yet, even if we add visitor counters, it would not inform us as to how many of these visitors went on to contribute to organizations or in what fashion. All we can do is verify that our site works correctly and identify ways to improve it.

INTRODUCTION

Problem

There are many recent and ongoing crises that have fallen out of the awareness of the developed world. A tsunami might hit vast areas in numerous countries, even more than one continent all at the same time, killing thousands and causing billions of dollars in damages to infrastructure. Information about the disaster and its immediate effects will be in the media continuously for weeks on end. During this time millions of people from around the world will recognize the dire need for aid and send aid in the form of food and water and/or monetary donations to organizations working on site. But shortly after, as quickly as changing the channel on a television, the media switches attention to the next big story or crisis, leaving the tsunami and its victims to fade from global consciousness.

Importance

Clearly, the fallout from these world crises far outlasts the media coverage afforded them in almost every case. Yet, the support and funding to the organizations providing relief and aid to those most affected by these disasters often reduces dramatically once public attention shifts to another topic, and these organizations become severely crippled in their capacity to help. Thus, the effects of a particular crisis last longer than necessary, people groups remain displaced or without food and medicine, and infrastructures remain desolate. There must be a way to keep those affected by the various disasters every year in the hearts and minds of the people of the developed world, those who are most able to help.

Our Solution (Contribution)

We are unable to stop disasters from happening, change the nature of the media, or force individuals to get involved. What we can do is make it easy for these individuals to become or remain involved by providing easily accessible information on these disasters, including updated statuses on the after-effects and the resource requirements of those organizations that are providing aid. The most obvious way to do this, within the confines of our particular skills and available funds, is to build a website that can scale indefinitely and remain stable and navigable while accepting a constant stream of new data and updates. This type of site can help to bring more awareness for those forgotten after these disasters cease to appear in the media, while providing concrete ways for visitors to get involved and make a difference in the lives of the survivors.

Difficulty and Key Components

At the beginning of this process we knew that we would need to acquire and use a number of existing technologies, without which our project would be infeasible. Unfortunately, only one or two people from our group had experience with building and/or designing websites or databases and this little bit of experience was isolated to only a small section of the overall project. Fortunately, though, there are many tools that would be suitable for this type of project that are available free on the Internet, along with the necessary documentation and user blogs to point out special cases and idiosyncrasies regarding their use. Therefore, we were able to start off with a complete battery of free tools that would meet all of our needs and allow us to build the entire site without spending any money that could otherwise go to the aid of those in need.

Of course, this approach involved its own difficulties. First, there was quite a learning curve for everyone involved and much of our time was spent researching how to use the chosen tools. One way we tackled this issue was that we separated group members into pairs that would be assigned specific tasks that required only a subset of these tools. In that way, each member of the group became expert in only 1 or 2 of these tools, requiring only a limited working knowledge in the others. This reduced the time spent researching and allowed us to begin making tangible progress very quickly.

Another difficulty was that the majority of the website features that we wanted to implement were reliant on one or more others. Thus, any attempts to build the entire site at once would require us to leave testing until the very end, since many of the features would not work correctly without the others. This was unacceptable, both because it left much uncertainty in the outcome of our product (especially with our team being so inexperienced) and because we use Extreme Programming (XP) methodology. So, we built the website over several iterations (in accordance with standard XP practices), each iteration ending with a working, usable website that was hosted on line. In this manner, we were able to guarantee that dependent features were always built on top of completed features, allowing us to fully test every feature being worked on during any phase of development.

Ultimately, the most difficult task has been the gathering of information for our site. The manner that we have chosen for importing information to our site requires the compilation of specific information provided in a very specific format. These have largely been completed by hand, although the automation of this process is a future goal. This takes extensive amounts of time and research for every crisis included in the site. One way we have dealt with this is to make a public import feature that any user of the site has access to. This feature automatically updates previously stored data with the current information via a “merge” setting. This gives other people the opportunity to contribute to our site and for those involved in the crises to provide real-time updates.

Results

In the end, we were able to produce a functional site that displays all the imported information correctly in an attractive way. We are pleased with the look of the site, especially with the lack of clutter and advertising that plagues so many free sites. Also, finding and navigating through to a particular page is very simple. A user can go to a list of just the crises, organizations, or people, from which they can choose a particular entity of that type. Additionally, we have implemented a simple search tool that is sufficient for the purposes of our site. Finally, there is a system in place for adding information that is accessible to users around the world, so the site has the capacity to grow and continually be updated over time. Thus, we feel that the site we have produced thus far is an important start toward achieving our goal of maintaining communication and resource flow between the victims of major crises and the world at large.

Limitations

Unfortunately, the current amount of data that has been added to the site is rather small and not at all in keeping with our ideals. Thus, the site’s effectiveness is mostly limited by how much properly formatted information that has been submitted to the site. Another limitation is the amount of updated information readily available on the Internet. While the Internet is full of freely available information from a seemingly unlimited amount of sources, we have found it quite difficult, sometimes impossible, to find all the desired information on a particular entity. Often this is a direct result of the problem that we are trying to solve: the media, and thus the general population, has ceased to talk about a crisis that has occurred more than a month past. This is why direct contributions from the organizations and people involved in these crises are so critical. Our current model for adding and updating data requires technical knowledge before it can be used. Future iterations of the website should add other methods of adding information that any layman can use to contribute to the information contained in our site.

This leads to another problem: reliability of information. We are very limited in our capacity to verify the information being added by users, which can significantly mar the usefulness and reputation of our website. We do, however, want to provide users with concrete ways to help, so any future efforts to address this issue will first be concentrated on contact information for the organizations involved, followed by verifying that the correct organizations are linked to each crisis.

Additionally, there are some technical limitations to our project, mostly resulting from our lack of funding and experience. For example, our entire website is hosted on the Google App Engine which puts a limit to how much data can be stored for a particular app before charging money. Likewise, we use GitHub extensively for source control and communication with each other about our code. GitHub will eventually charge us for use of private repositories on their site. Similarly, we lack the funds to buy tools for our website, which is, therefore, limited by the capabilities of the free tools that we have used. As for experience, we have very little, but we are finding that our knowledge of how to build and improve the website is continuously growing and that each iteration seems to be progressing more smoothly and effectively. We expect that this will continue to be the case over time as our expertise in the tools becomes more developed and our team finds a rhythm and work flow that we all are comfortable with.

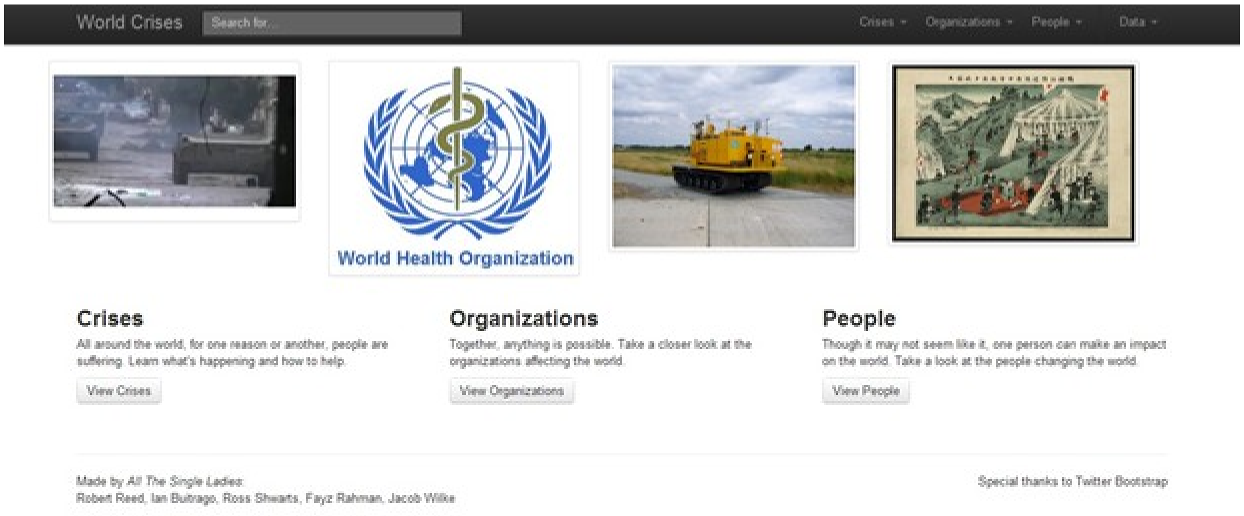
DESIGN

Design Goals

The primary idea that drove our website design was simplicity. By eliminating unnecessary site flourishes the user is able to focus on what is important – the data. Thus, every aspect of the user interface and data handling is treated in a minimalist way as much as possible. That being said, we also wanted to garner an emotional response from our audience in the hopes of engaging users in the plight of the victims that are very much in need of their help. Yet, we wanted to achieve this without compromising the truthfulness and validity of our data. In the end, this was mostly done by making the images a major portion of most of our pages.

User Interface: Look and Feel

In keeping with the idea of simplicity, the site was designed with a clean, focused look. The site's usage of typography emphasizes the text. Color is used only sparingly, with most text being displayed in black and the background and buttons white and pattern-less. We originally played around with using colorized or patterned backgrounds that were more "interesting," but we found that it threatened our ideals for this site. Our choice of black and white gave any use of color far more impact and meaning. Thus, viewer's eyes will be drawn to the most dominating elements of color on the screen --the pictures of crises, organizations, and people.



Crises
All around the world, for one reason or another, people are suffering. Learn what's happening and how to help.
View Crises

Organizations
Together, anything is possible. Take a closer look at the organizations affecting the world.
View Organizations

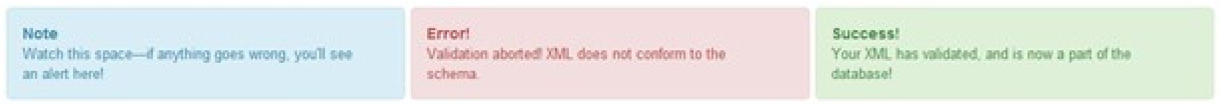
People
Though it may not seem like it, one person can make an impact on the world. Take a look at the people changing the world.
View People

Made by *All The Single Ladies*:
Robert Reed, Ian Buitrago, Ross Shwarts, Fayz Rahman, Jacob Wilke

Special thanks to Twitter Bootstrap

Example 1: Front page of the Phase II iteration of the site's home page. Note how all of the color is in the pictures, and none of it is in the user interface.

When color is used it always conveys a meaning. Links, whether external or to other pages in our site, are shown in blue to make them stand out. Status bars in both the import feature and the GAEUnit test results involve color to clearly convey the status. Green means "Success!" and red means that an error or other failure has occurred. The navigation bar at the top of most screens is a combination of grays, mostly to distinguish it from the pages themselves in a visually consistent way.



Note
Watch this space—if anything goes wrong, you'll see an alert here!

Error!
Validation aborted! XML does not conform to the schema.

Success!
Your XML has validated, and is now a part of the database!

Example 2: Examples of when color is used in the user interface. Here, we're coloring status messages on the site's import page, to give users visual feedback.

In keeping with our design mentality of simplicity, we decided that each page should hold only a minimum of information. Thus, the site is divided into several page types. Each entity contained in the website is displayed on its own page, with all pages of a single type having the same layout. There is also a sort of master page for each of the three page types that displays all entries of that type. These master pages also have a consistent layout. Finally, there are 3 tools pages: an Import page, an Export page, and GAEUnit testing.

User Interface: Navigation

There are two primary ways to navigate through the site. First, it is possible to navigate to any crisis, organization, or person page via primary links from the home page. Each of these links takes the user to the master page for that type. This page contains four images at the top, and a list of entries below it. Specifically, the list at the bottom of each page contains links to 4 of the entries of that type, and the images are taken from these 4 pages. There are then next/prev links at the bottom to navigate back and forth so that all entries are reachable in this way.

Then there is the navigation bar with 4 drop-down menus that is accessible from almost every page and is the primary way to navigate through the site. The first three drop-down menus display links to Crises, People, and Organizations. The first four links on these drop-down menus lead to the four most recent entries of that type, with a 5th link to the master page for that type. The last drop-down menu displays links to the three tool pages (Import, Export, GAEUnit) and a link to an “About” page that includes information about the site, the group, and our goals. The navigation bar also includes a search tool, which is described below.

XML

The XML instance was our tool for importing data into the site. The XSD language was our tool for specifying exactly what information can be included and how that information must be displayed. As shown in Example 3, the schema is essentially a listing of data fields, called elements, which are used as a way to transport individual pieces of the overall data to our website in an easily accessible way. Each of these elements has a specific type, whether native or user defined, that places demands and restrictions on the content of that element in an XML instance file. The elements are also nested in ways that specify the location and ordering of these elements in the file.

```
<xsd:complexType name="extType">
      <xsd:sequence>
       <xsd:element name="site" type="xsd:normalizedString"/>
       <xsd:element name="title" type="xsd:normalizedString"/>
       <xsd:element name="URL" type="xsd:token"/>
       <xsd:element name="description" type="xsd:string"
minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
</xsd:complexType>
```

Example 3: This is a snippet of code from the XSD schema that we used. This example defines an external link for use in the rest of the schema.

When designing the XML schema, the driving forces behind our decisions were the same as if we were writing software. We wanted types that made sense for the data the elements contain, we wanted to group snippets of “code” that are used multiple times into reusable structures(complex types, groups), and we wanted to use good names for the elements that make the expected contents of each element clear. Also, given the nature of the project and the expectation that some of the data would either be incomplete or would simply not be available or applicable for a specific entry, we wanted to allow for the omission of as much of the less critical data as possible.

We carefully designed our schema in this way. However, in the end we had to use a schema that we did not write ourselves due to circumstances beyond our control. While this schema made use of the programming principles outlined above, it did not allow for the omission of data as readily. Additionally, elements that contain essential information could be left blank. This has ultimately required that conditional logic be added to several places in our source code, thus complicating our python files unnecessarily. Fortunately, this other schema does contain elements for all of the essential data that we require, as well as several that we had not thought to include.

UML and GAE Datastore

Once the information is imported into the website, we needed to store that information in an easily accessible way. Simply storing the XML instance files would greatly prohibit accessibility and make our algorithms excruciatingly slow, likely causing our users to move on to other sites. The GAE provides a database like tool called the Datastore, which allows us to distribute the information contained in each instance across several models (database tables) that contain specific pieces of the information that we need to store so that it can be accessed very quickly. This solves our problem; the pages of our website can be generated in real time seemingly instantly, at the cost of a slight overhead only when the information is imported or exported.

To design how this information should be stored, we used UML before developing the models. The XML schema that we used is structured in a way that we were able to do a 1 to 1 relationship between the schema structures and our models, transferring each element directly to a field in model. Each class in the UML has its own box which includes all the data fields that it will contain. The actual models, of course, include reference keys for the elements they point to. Because of the nature of the information being stored, most of relationships in our design are 1 to 1. The exceptions to this involve the Link class, which contains information for both external links and embedded content, and the Reference class, which contains links to other pages on our site.

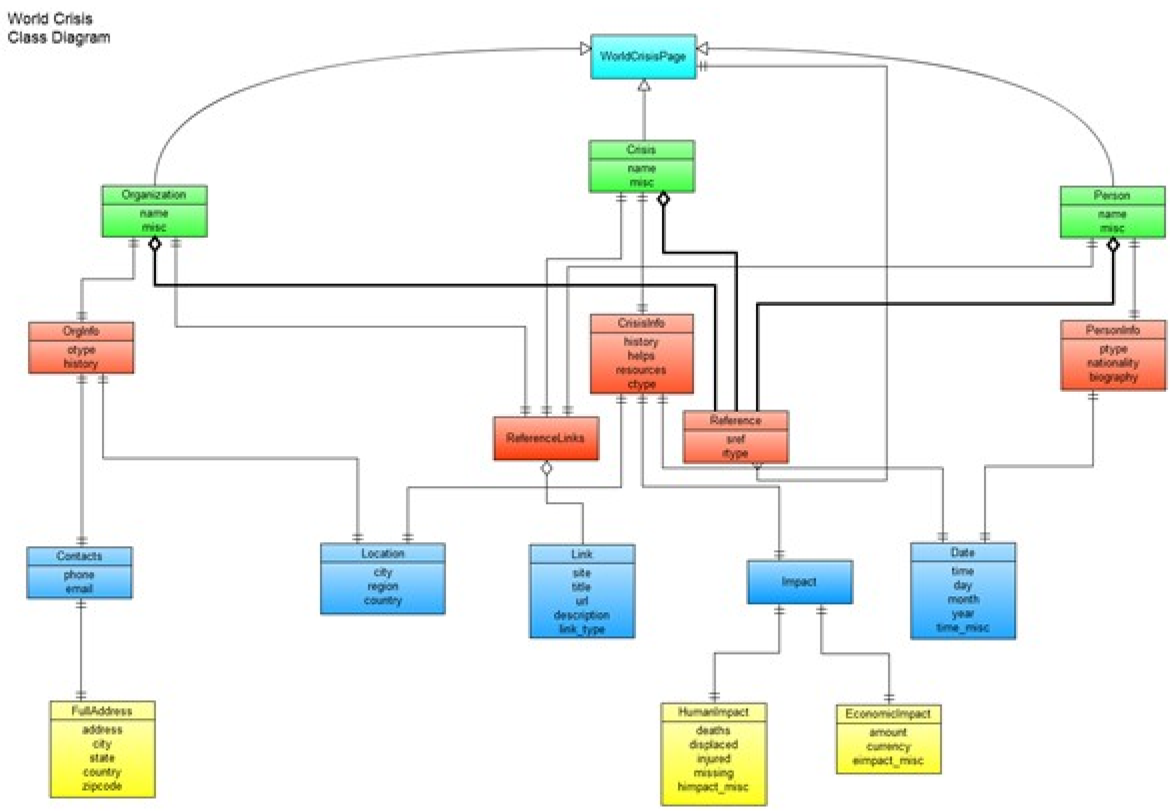


Diagram 1: This UML diagram illustrates the relationship between classes in the Datastore

In the UML diagram, the higher level models contain references to the lower level models they have a relationship with. These reference fields are not included in the diagram, as they are meta-data and have nothing to do with the data we are trying to store. If the relationship between two models is an aggregation, then the higher leveled model has a list of references to the lower level model.

At the top of the diagram is an abstract model: WorldCrisisPage. This model contains the time and date that the model was last updated. As shown in Diagram 1, there is a model for each of the three page types that inherit from this model. Each of the entries of these types is the "head" of a single data page on our website. Thus, all of the models below it are required, including the aggregations, in order to build that page on the website. So, when the data is transferred to the models from the schema, each model has to be created from the top down, with the reference key being passed back to the previous one. Thus, each model is only stored after all the lower level models have been stored. This link has the latest version of the diagram: http://www.gliffy.com/pubdoc/3716424/L.png.

Special Feature: Merge Import

Merging is now an option whenever the user wants to import XML files. Whenever this option is selected, the new XML instance is added to the existing data on the data store. Prior to this option, the import would delete everything out of the data store to perform a clean import with the XML instance being the only data visible to the website. This allows the website to be constructed in pieces of XML files as opposed to requiring one huge XML file containing the data for all of the entities.

Special Feature: Search

A search bar prominently displayed on the site--held within the navigation bar, it is effectively on every page. Initially, we had implemented an auto-complete feature to match typed text to the names of the entries in the Datastore. However, with usability testing, it was determined to be highly when searching for terms that were not a part of an entry's name.

When searching for results, the website brings users to a search results page that displays only the results, prominently. With just the text, and highlighted with a bright blue color, the focus is brought onto the results.

IMPLEMENTATION

Execution

In order to build our website, we knew that we would need to familiarize ourselves with a number of features and tools, all of which we had little to no to experience in using them. Specifically, we needed a database to store our information, a method of importing and exporting data into the website, and a dynamic way of generating separate pages for each of the entities in our database, and a way to display these pages to the screen in an attractive and intuitive format. Also, the implementation of our website was completed over 3 iterations, each of these leading to a functional version of the site.

Framework and Tools

Research and training in using these tools took up a large portion of the time that we spent working on this project. Yet, we , designing ways to implement all of the features that we desired from scratch would have made the project prohibitive. Because the work of the project was divided up between various pairs from within the group, only 2 or 3 people needed to be trained for most of the tools. We found this was a great way to divide up our resources. Furthermore, while each person did not get to become "expert" in more than 1 or 2 parts, we traded partners and positions enough that we were all exposed to the various pieces involved. This knowledge and experience is ripe to be expanded on in future assignments.

The tools that we used are as follows:

1. Google App Engine
2. Django web framework (GAE version)
3. GitHub/git
4. Minixsv
5. CoreFiling XML Schema Validator
6. Twitter Bootstrap

7. Pydoc
8. Gliffy
9. GAEUnit
10. XML

What follows is a description of how we used each of these tools, in order, how we used them, and how these tools impacted our project.

We chose to use the Google App Engine to build and host our website. Google App Engine provides a free and relatively simple way of running this site and was highly suitable for our needs. It provides a database(Datastore) that is easy to manage and that doesn't require the use of any complicate query language, while being remarkably scalable. It also provides us with a domain that we can reliably use indefinitely. GAE also provides a python SDK so that everything can be written in a language that we were already familiar with. Additionally, the Python SDK allowed us to simulate the site locally without having to update or change the live site every time we needed to locate a bug or add a new feature. In short, we found GAE to be the perfect development environment for the needs of our project.

The templating system of the Django web framework was used for the generation of our web pages. We are able to dynamically generate web pages by using python-like syntax within our HTML files. For example, say we want a page with all of our crises. Statically, we'd have an unordered list, and specify each list item. Dynamically, we'd still have a unordered list, but now we can use Django to insert a for loop to iterate through a list of crises, and "write" list items out to the HTML page. Then in our python code, we can query the Datastore for crises, and pass it into the template. The Django templates were essential in making our site dynamic.

We used GitHub to host our remote repositories and track our issues. Emulating the developers of GitHub we used the fork and pull request model. We use a single main repository that we tried to keep in a stable compiling state and multiple forks of the repository, one for each person. Each team member is then free to push changes to their independent fork without affecting the main repository. Then when that fork reaches a significant milestone, the owner can submit a pull request to the main repository. The pull request creates an issue with broadcasts to all the collaborators(team members) through email. Then that code can be reviewed, discussed and then merged with the main repositroy. Using git was a huge asset in keeping track of changes.

The CoreFiling XML Schema Validator was used at first to make sure our revised XML instance validates with the new schema. Towards the end of phase 2 development it was mostly used as a sanity check whenever a new XML file wouldn't import into our site. This validator was used as the standard for any validation.

Twitter bootstrap is the user interface framework used for our front-end. It includes many built-in components that can be easily combined to create a good looking site. Plus, the default styling fits in with our design principles. These factors made Twitter Bootstrap an excellent foundation for our website's design.

Pydoc was used to generate HTML files of documentation as dictated by the project specifications. To be able to run Pydoc without errors in a GAE app, these paths should be added to PYTHONPATH variable in Linux.

```
export PYTHONPATH=~/Documents/cs373/google_appengine/:    \

~/Documents/cs373/google_appengine/lib/webob_1_1_1/
```

Example 4: Shows a useful command that can be explicitly executed or added to your .bashrc file.

The flash website Gliffy was used to design the UML. It allows shared files between registered users and also provides different URLs to the file. It can also export the file to many formats but not PDF. SVG files have infinitely better resolution though.

Validation of imported XML instance files was accomplished using minivan. Minixsv is a python library that depends on another pure python parser named genxmlif. Together, these libraries ensure that our given XML schema is well formed and that the XML file instances conform to the ground rules laid out by the schema. The interface into minixsv has a few ways to accomplish validation including file and string representations of the given schema and instances. The path we chose was to use strings. The schema is hard coded into one of our functions and the XML instance is provided at the time the user hits the “submit” button. Then minixsv and genxmlif does their magic and has 3 possible outcomes. The first is an error message that occurs if the schema and/or the XML instance is not parsable. This error occurs whenever the user forgets to use a close tag or something syntactically incorrect. The second is another error that occurs if the schema and the instance are not well-formed.   This is more of forcing that XML instance has all of the required content that is needed by the schema. Finally, the third outcome is that they are both parsable and well-formed. Then the validation is complete! The handler will then pass the information into import.

Google docs was immensely helpful in completing the report by enabling us to type in the report simultaneously. The document was easily shared between all the group members and had almost every feature of a traditional word processor including exporting the document in PDF format.

We used an XML instance to compile all of the data that we wanted to store in the Database in an organized way that is completely readable to a human, syntactically specific enough to allow for automated parsing by a machine. The XML language is remarkably flexible and accommodates this quite well. Unfortunately, the XML is actually a little too flexible and only places a minimum of restrictions on the content of such files and no restrictions on the layout or order of the information. Therefore we needed to provide a schema for what information would be expected and could be used to enforce these expectations, making them, in fact, requirements. To this end, we used the XSD language and imported minixsv to be used as a lightweight schema validator.

Datastore

The Datastore offers persistent storage for our crisis data. Without it, the data would be lost every time the browser window closes. It uses our models from Models.py to build tables and references to those tables. An issue we have run into is running a local instance of the app in the CS machines will create an Datastore file on that particular machine. This file gives permissions only to the last person who used it. So any other team member or other team cannot simulate an app on that machine that uses the Datastore. This issue is still being worked on.

Import/Export

Our import page has a file submit box that will accept a valid XML file and put that data into a presentable view on our website. Whenever the user hits the “submit” button, our handler extracts the contents of the file into a string and passes it through the minixsv and genxmlif validator. The validator ensures that the XML instance is well formed (all open tags have closing tags etc.) and also makes sure that the instance conforms to our XML schema. This validation check makes certain that we can parse the file later and that we have all the data necessary to display the given pages. After validation, our handler that runs deletes all the models in our Datastore and then proceeds to pass the XML instance file to our import function. The import function parses the file and builds an ElementTree of the XML and builds all the parts required for our models and stores all of the new models into the Datastore. At the moment, our import function requires all tags to be completely filled out otherwise there will be an import error and our website will delete all existing models.

Our export page has a button that displays the current XML instance that our website has stored in our Datastore. It displays the information attractively by being appropriately colored and correctly indented. Behind the curtain, we use the models stored in our Datastore to rebuild and re-parse the ElementTree. This XML instance can be significantly different than the actual XML instance that was imported and this is okay. For instance, if the original imported XML file has an empty tag that is optional, it will be excluded from the output of the export XML instance. Also, order of the entities within the files will also vary because of the way they are stored inside the Datastore.

The import page will have two possible settings, a replace import and a merge import. The replace import is implemented and guarantees that the information will make sure that the data is backed up prior to trying to build the models so the website isn’t empty with an invalid XML file. With the addition of the merge import, we will be able to add new models to coincide with our current models without the need to delete them. This will allow our database to expand and grow for future crises to come that need awareness.

The policy for our merge import is as follows: if the id references inside the XML instance are different from what is already in the Datastore, then no merging occurs; and if they are the same all the data gets updated to the newer instance except for text fields, which are concatenated if they are not substrings of each other. This decision to not merge based solely off of id references was mostly for cases where two entities could look similar and in reality be different. We opted to heir on the side of safety, we didn’t want to merge two instances that should really be separated. Also, if two things are the same but still have different IDREF's, they would get separate pages on the website. We felt that even though not technically appealing, this was better than the alternative. To facilitate more accurate information, we added a list of name and id references to the import page, so the user will be able to check their instances prior to importing them into our website.

Search

Our search functionality was implemented via Google's Search API for Google App Engine. The API is still considered to be experimental, and thus, the documentation was lacking and a bug prevented us from testing certain features on a local development server. Because of these factors, we were unable to implement a "snippets" feature--where each search result is accompanied by the snippet of text that matches the search terms.

Despite that, we were able to implement AND and OR searches for search phrases--where we search the data for each phrase ANDed together and each phrase ORed together. Because AND is stricter than OR, results that appear under AND will also appear under OR. To avoid redundantly displaying results, we utilized sets and set difference to remove all the results in OR that also appeared in AND.

We were also able to get an unstable version of 'exact search' working, where it searches for the exact phrase the user input. Sometimes, it works, and sometimes it doesn't.

Files and Directory Structure

All the python files are in the top level, except for our unit tests which is in a folder, "test/". Models.py contains the data models. RunExport.py calls Export.py. RunImport.py calls Import.py. Bootstrap framework is in the static/ folder. The pydoc HTML files are in the pydoc/ folder. All the template HTML files with Django are in templates/.

Team Member Responsibilities

During phase 1, Ian worked to develop primitive models for the Datastore. He also set up the GAEUnit test page for our site. Robert and Fayz developed the import/export schema and XML parsing. Jake and Ross compiled our research into both a text file and XML file. Robert was the group leader during this phase.

Phase 2 was distributed differently and a little less evenly. While everyone contributed to the design of the site, Fayz implemented the majority of that code. Robert and Jake rewrote every part of the import/export code to conform to the new schema. While Ross and Jake worked together to conform the our data to the new XML file. Ian developed an outline for the report, which everyone was able to contribute. He also helped with testing and finalization of the project. Ian and Ross co-created the initial UML diagram. Ross was the group leader during this phase.

During Phase 3 Jake, who was project 3 group leader, worked with Ian to add the Merge feature to our Import code. Jake was heavily involved with the implementation of the taskqueue and overall testing of the site. Ross and Fayz added the search feature, and Fayz continued to update and improve the design and functionality of the site with the input and feedback of the rest of the group. Ross and Robert wrote the critiques. Robert performed major revisions on the report to achieve unity, flow, and organization, from which point the rest of the group made edits and added content related to their tasks for the 3rd phase.

EVALUATION

Testing

We were adamant about using Extreme Programming (XP) methodology to guide our process of development. Guide is very much the appropriate word, as we did not hold completely true to all of the tenets put forth in the XP bible, Extreme Programming Explained, which demands failing unit tests be written for every snippet of code and that these test should be run quite often. The XP model goes further to say that no code should be added to the project source code without passing all the unit tests written for the entire project. We did not always guarantee that these were followed during the first 2 phases of our project, particularly because of the learning curve with the tools we were using, indeed, the GAEUnit framework was the last part of our first phase features to be up and running. In retrospect, our inexperience with the tools that we were using made the tests even more important, because the chances of us producing incorrect code was significantly elevated by this fact. However, during the first phase we often felt like we were flying blind, for even though we had read documentation on the tools we were using we had major difficulties with getting the various pieces to fit together. Thus, most of the time our code wouldn't even compile, which made any tests that we would have written quite useless. We were able to run acceptance tests during this phase, however, so we were confident that the code we produced worked correctly.

By the second phase, we had figured out all the glitches with getting all of our pieces to fit together to the point that every member of the team could run the project from his terminal and see the results of their changes, both in the GAEUnit framework and in the site at large. Yet, we still were having difficulties getting tests to run in the GAEUnit framework, because GAEUnit would not always run our tests, even though they were named appropriately and located in the correct folder. Fortunately, by the end of the second phase we had been able to solve this issue and the test page has been a working part of our site, so that any team member, or user, can see if these tests are 100%.

In the third phase, we expanded the testing from the second phase by re-using the snippets of XML we built in string and passing them to our import function. This time, we tested our merge function, so we created XML that looked very similar with changes in spots that the merge import might have troubles with. For example, we tested checking to ensure that all histories, biographies, and miscellaneous fields were concatenated if the two weren’t substrings of each other. The taskqueue we had trouble testing but we ran it quite a few times with many different XML instances to ensure that it works properly.

At the end of all 3 phases, we found that we needed to do a round of testing and, potentially, correcting of the hosted site. We found that some features that worked a particular way on our local machines did not always produce produce the same functionality when hosted on the web. For example, throughout phase 2 we used a diff feature extensively to test our import/export feature. This was done with the expectation that if our import and export scripts were running correctly, the XML instance produced by a command to export should exactly match our import file. By the end of the phase, this was always the case. Yet, when we hosted the site to GAE, the diff's began to fail. This was because on our own machines, our data were the only things in the Datastore, However, the on-line Datastore holds data for almost every website that they host. So, while the data would be correct, the entities were not described in the same order. In the end, we added a few lines to guarantee order so that our diff tools could still be used for testing in future phases. This wasn't really a problem, but we did waste a lot of time trying to find the problem with our code. However, had it been important to the functionality of our code, the results could have been disastrous if we had posted what we thought was a fully functional site to the web.

Unit Tests

All of our unit tests All the tests are contained in a single file, TestWC3.py, and are run via the GAEUnit framework. This framework automatically runs all test of our tests whenever a user, be it developer or other, goes to the /test page. This page either displays a green bar with an “all passed” message, or it displays a red bar with information on the number of failed test, the number of tests that produced errors, and where these failures and errors occurred. It never shows the tests (only a developer of the site can see these), but the results page always shows the total number of tests that are ran. There is a link to this page in the 'Data' drop down menu that is accessible from any page in the site.

The unit tests have focused primarily on the Import and Export feature. These tests use a bit of a shortcut to check the whole import and export process, thus testing RunImport.py, Import.py, RunExport.py, and Export.py. This includes whether these programs are storing and accessing information to/from the Datastore correctly via the models in Models.py.

Performance Tests

Besides the unit tests, we thoroughly tested our import and export facility with other XML instances of other groups. Corefiling XML schema validator we used as a last resort when we could not get an XML instance to validate with our application. We plan on writing some more systematic performance tests with time output for import/export and searches.

We also tested our website by importing various XML files into the database and seeing the results. We created some test XML files to test basic importing. Then created a few other similar test XML files to test the merge importing. This helped us flesh out the small details of inconsistencies between the two files.

We boosted our performance by implementing a simple multi-threaded task queue to perform some of our work in the background. Whenever a user imports an XML instance, the actual call to import takes a while, especially if the user has to wait to see the results. So we added taskqueue, an API provided by google, to perform all this work in the background. It has the slight complication of not having immediate results in our website, but the entities will appear eventually.