Technical Report

Unicorn Steroids

Michael Tarng, Boris Jonica

Cassie Schwendiman, Ryan Prater

Bryan Vuong, Dereck Sanchez

Phase 3

November 28, 2012

Table of Contents

Introduction

Problem

While there has always been many different crises affecting the world, the rise of technology has allowed people to become more in touch with what occurs around the world on a more personal level. Yet, there are so many crises occurring around the world that staying informed and knowledgeable on present and past events that have occurred has become a daunting challenge. Many different organizations have been established to help tackle these situations across the globe. Some are specific to a cause, while others aim to provide general aid with a wider reach. Outstanding figures also rise up to offer assistance where they can and often become a personal representative towards a cause. The network of information that spans these different crises, groups and people encompasses an enormous and cumbersome amount of information. This makes it difficult to keep track of. Our goal for this project was to make this information more available and manageable to users that are looking to learn more about a certain crisis, organization, or person. By using Google App Engine (GAE) we were able to create a web application that gathers all related data into a central location that is easily navigable and informative to users and will greatly assist in spreading knowledge about emergencies happening around the world.

Solution

We envision multiple different uses for our crisis web app with a focus on educating users and providing them with proper tools to be informed. Users may visit the website to search for a particular crisis, organization or person. The search engine brings up all the relevant pages to the search terms and allows the user to quickly find the page for the specific item they were looking for. There will be pages pertaining to the actual crisis as well as for the organizations and people that are involved. With these three base templates, users will be able to navigate between them and gather data on a specific crisis such as where it is, who is involved, and who is assisting. From there, users can find live social feeds, if applicable, or information allowing them to get in contact with the groups find ways to help out. This will allow interested individuals to stay up to date about relevant events in their community and around the world.

Design

XML

In order to import data to the disaster database, we used xml. The xml schema is composed of a root node, world-crises, with three main subtypes: crises, organizations, and people. These subtypes possess a one-to-many relationship with their singularly-named individual instances (eg. person). From here, the types possess common data, such as name, location, description, etc. and relevant, specific information. For example, a crisis can contain valuable information such as ways to help, resources that are desperately needed in the crisis-zone, an organization instance will provide phone numbers and addresses for contact, and a person's brief biography and contributions will be provided.

```
<xs:complexType name="link">
    <xs:sequence>
  <xs:element name="source" type="strict-uri"/>
  <xs:element name="description" type="short-token"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="images">
    <xs:sequence>
  <xs:element name="image" type="link" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

Excerpt from our XML schema

In addition, with the rise of new media in the internet, integration with its many forms of media, such as Twitter, Facebook and Youtube was added. To account for this, the schema allows for people, organizations, and crises to reference these objects and more. The ability to embed media such as videos and twitter feeds, enhances the overall versatility and appeal of each page. For example, a visitor to the site who has no idea about the Libyan civil war will be presented with videos of the crisis, social feeds of continuous information, and a map of where the crisis is happening. As a result, we have designed a robust and expansive schema that will accommodate many forms of data while maintaining referential and data integrity. Additionally, considerations

were also made to export the data according to a set standard. This allows for sharing the data with groups who are interested in the listed data on our website.

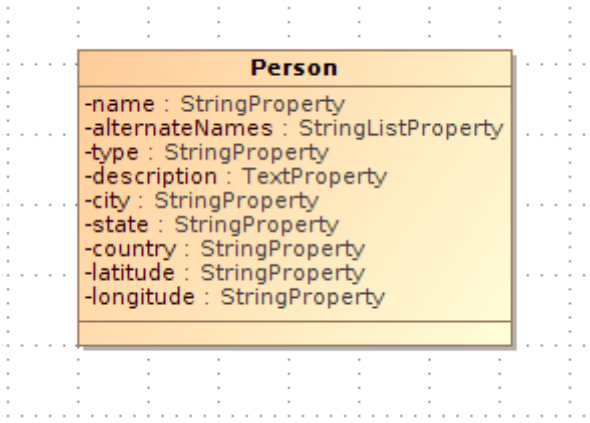
UI

We took a simplistic approach to our UI by just creating tabs separating the different categories which a visitor can browse through. This allows our user to easily and quickly navigate the different crisis, people, and organizations to find the information they are interested in. Importing was also made effortless by adding a file browser where you can upload straight from your personal storage. Export is accessible with the press of one link making it hassle-free where the xml instance of all the data stored is displayed in the browser. We were able to achieve this simplistic approach by utilizing Twitter's CSS framework, Bootstrap. This provided many default styles to easily give a cohesive appearance to the entire app.

Implementation

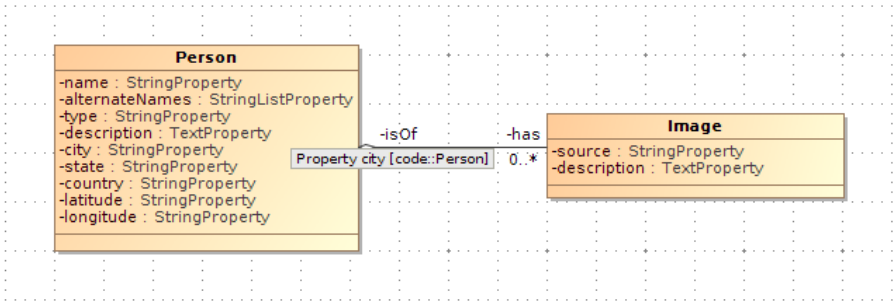
GAE

By using the Google App Engine, we were able to create three main dynamic entities (classes): Crisis, Organization, and Person. In each of these classes, we provided corresponding attributes (fields) as specified per the XML schema. After navigating and learning the GAE documentation, we provided property types and constraints for our fields. For example, a Person class contains a Name attribute that is required and of type StringProperty, while the economic impact of a crisis is a IntegerProperty. This maintains integrity by ensuring malicious data types (eg. None, int, dict) cannot be set as a person's name. Each Person is also given a Description of TextProperty, which allows for a longer character field (greater than length 500) and cannot be filtered or sorted.



A UML model of Person class

In addition, we created special classes for common entities that possessed common attributes and were used among all of our main classes. These classes include Image, Video, Social, ExternalLink, Citation, and Map. From a database point of view, these can be viewed as subclasses that contain their own unique identification and attributes, but possess a foreign key to a Crisis, Organization, or Person.



A UML model of the Person - Image relation

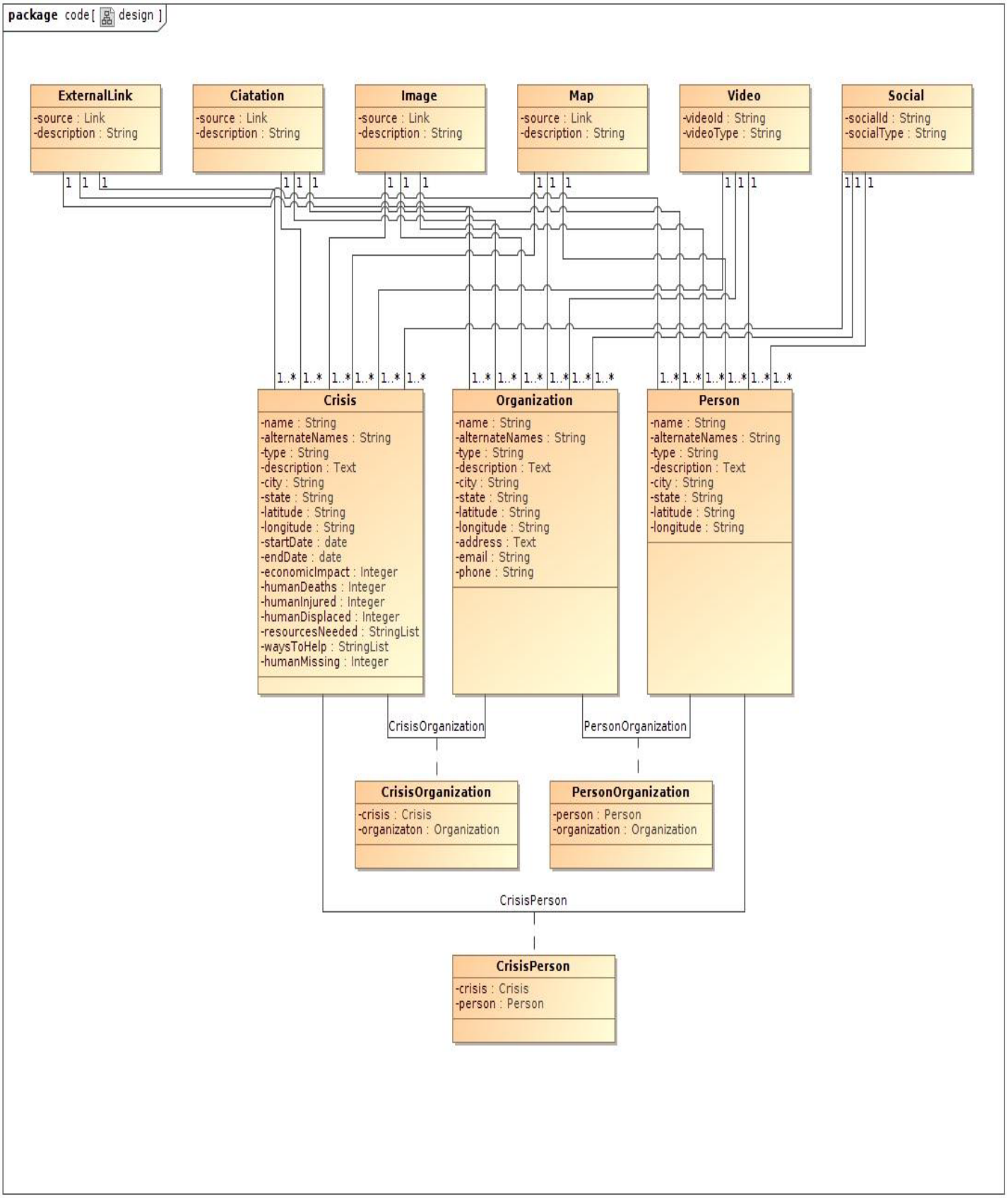
By leveraging the GAE API to import these models into the app datastore, we were able to parse the XML instances into model instances, and instantiate a cloud database of our information. From there, we could select our root datastore node and iterate through it, parsing out information and exporting our data back into XML that maintains validation with our schema.

In order to allow collaboration of data between multiple parties, the GAE must be able to merge and display any data for entities submitted from different parties. In order to realize this goal, we permitted our Importer function to import duplicate entities. Although information for the different entities may be different, they are linked by a common ID. Our datastore is permitted to store multiple instances of the “same” entity as long as the IDs match. Since only Organizations and People are allowed to have duplicates, a merge function isn’t necessary for Crisis. When a user requests a page for an entity, it runs a server-side gql query that returns all instances in the datastore with the ID of the page requested. Merge functions then take all of the data from each instance and re-packages it into a dictionary which is then displayed on the entity’s page.

We later expanded this system to utilize GAE’s task queue. This enabled multiple import jobs to be queued up to run when the server was free. We were able to implement this using the default Google App Engine push queue. We simply wrote a handler that adds a new import task to the default queue anytime a user imports a new valid file. Then the push queue is automatically handled by GAE and runs the tasks on its own.

UML

The design of our database includes 3 main entities: crisis, person, and organization. Rather than having objects such as media sources, maps, citations, and external links directly listed under each of the main entities as attributes, we created an entity for each object that references one of the 3 main entities it’s related to in a many-to-one relationship. In addition, by establishing a many-to-many relationship between the 3 main entities, we created join tables in the GAE datastore that stores what entity references what.


package code[design]
ExternalLink
-source : Link
-description : String
Ciatation
-source : Link
-description : String
Image
-source : Link
-description : String
Map
-source : Link
-description : String
Video
-videoId : String
-videoType : String
Social
-socialId : String
-socialType : String
Crisis
-name : String
-alternateNames : String
-type : String
-description : Text
-city : String
-state : String
-latitude : String
-longitude : String
-startDate : date
-endDate : date
-economicImpact : Integer
-humanDeaths : Integer
-humanInjured : Integer
-humanDisplaced : Integer
-resourcesNeeded : StringList
-waysToHelp : StringList
-humanMissing : Integer
Organization
-name : String
-alternateNames : String
-type : String
-description : Text
-city : String
-state : String
-latitude : String
-longitude : String
-address : Text
-email : String
-phone : String
Person
-name : String
-alternateNames : String
-type : String
-description : Text
-city : String
-state : String
-latitude : String
-longitude : String
CrisisOrganization
PersonOrganization
CrisisOrganization
-crisis : Crisis
-organizaton : Organization
PersonOrganization
-person : Person
-organization : Organization
CrisisPerson
CrisisPerson
-crisis : Crisis
-person : Person

Framework

After uploading the models into the datastore, we then needed to create dynamic pages for them. We used django templates for these pages because there are many common elements shared across various pages. For example, we wanted to provide navigation for the app on every page. In order to reuse the code for this navigation across all of our pages, we created a base template and used overwritable blocks to make the content dynamic. Similarly, the models all share a lot of common data so in order to reuse the code for this data, we created a base object template and used another overwritable block for the data that was unique to the object type. Using templates kept our code repetition to a minimum, and made the code easier to maintain since we didn't have to change it in multiple places.

Search

Search was implemented with Google App Engine's Search API. The search API works by building search *documents*, which are collections of fields for an object. Much like a Datastore model, there are specified *Field* properties which can be saved and later searched (eg. TextField, HTMLField, etc.). Once the document is built, it can then be added to an *Index*. Indexes are collections of documents and allow accessing methods to their contents. A user can add, delete, or modify documents, as well as query an Index for its schema and contained contents.

After an index is instantiated and added to the App Engine, a user can perform a search (in our case through a frontend searchbox). That search hits a SearchHandler in our main python file, parses the search text, and turns it into a GAE Search Query object. Query objects represent requests to search indexes, and allow the developer to customise and change search options (through use of a QueryOptions class). For example, in order to get search result snippets, we define a snippeted_fields variable in our QueryOptions class, instantiate our Query class with QueryOptions, and then call search.Query with our query string and Query class. A GQL object is returned and the developer is able to iterate through the results via a normal "for" loop.

Tools

In addition to django templates a few of other external tools were used on this project to assist with various tasks that needed to be implemented that were not inherently a part of python or the Google app engine. Minixsv is a schema validator package that was used to ensure any xml instances trying to be imported to the system would be formatted properly according to the schema. This saved us the hassle of trying to write our own xml validator which would be extremely time consuming. Gaeunit was a tool we used to incorporate our unit tests to the website. This allowed us to easily view and run our tests anytime we were viewing the appspot website by simply following the link to the tests.

Testing

In order to ensure consistency and the integrity of our disaster database, we used the unit testing application associated with GAE. We identified our targets for testing as the integrity of the data stored in the GAE datastore. In order to ensure this, we tested the methods used to import our xml data and the methods used to export all the data stored in the GAE datastore. We did this by loading in a valid xml instance, and checking that all values in the datastore are the same as the values in the instance. We also tested helper methods within our models used to format and access a few of their attributes.