Technical Report –
Design and Implementation of the World Crisis Database

- Group name: team
- Group members:
    - Stephen Chiang
    - Aaron Stacy
    - Blake Johnson
    - Jason Brown
    - David Coon
    - Qunvar Arora
- Group URL: http://cs373-wcdb.herokuapp.com

Introduction

Problem

People interested in learning about world crises do not have a central repository of world crisis information. The average person would have to conduct a series of online searches in a patchwork attempt to gather related information. This approach is time consuming, inefficient, and its expensive time commitment detracts others from gaining awareness into major world issues. An ideal solution involves a single website that can provide easy-to-access information on world crises. This website would act as a hub of information for the person to learn about influential people, organizations and media related to the crisis of interest.

Use Cases

Readers can gain information on a particular world crisis event, such as human impact, economic impact, resources used to address the issue, and ways to help those affected by the crisis. They can learn about influential people involved in the event, organizations involved with the event, and related media resources. Media resources include online articles, blogs, images, videos, social networks, and maps.

Readers can do the following: select an event and see the related people and organizations, select a person and see the related events and organizations for which that person has an influence upon, select an organization and see related events and people.

For those people interested in contributing information to the website, an administrator page is available for them to share information.

As the public becomes more educated on this issues, they are more likely to take constructive action to address and/or resolve these issues (e.g. elect representatives, form grassroots support campaigns, volunteer, donate money).

How To Use The Site

Terminal Command (in root project folder)	Resulting Action
make run	Runs the server
make test	Runs the unit test
make clear-cache	Clears the cache

Design

Directory Structure

<Blake>

cs373-wcdb/ : Git Repo

- assets/ :Various Project and Website Artifacts
  - log/ : Time Log
  - report/ : Technical Report
  - tests/ : Public Project Unit Tests (see make file to run)
  - ui/ : Website Artifact for Updating Icon
  - xml/ : Public Project XML Schema and Instance
- crises/ : Django Project Folder
- crises_app/ : Django App Folder
  - converters/ : Converters to Help Move Info To and From DB
    - prune_xml.py :Gets Models from Class DB
    - to_db.py :Converts Xml to Model Form for DB
    - to_xml.py :Converts Models from DB to Xml
    - url_to_embed.py :Follows URL and Outputs Embed
  - fixtures/ : Various Fixtures for Early Development
  - management/ : New Manage.py Commands
  - static/ : Static Files (css/img/js)
  - templates/ : Django Templates
  - Various Django modules : Modules to Support the Django Framework
- doc/ : Pydoc Documentation
- Procfile : Heroku Configuration
- README : Project Readme File
- makefile : Project Makefile (commands in tech report)
- manage.py :Django Manage Module
- requirements.txt : Heroku Required Python Modules

Differences between crisis directory and crisis_app directory

The project folder ("crisis") contains site-wide data. It contains error handling pages (404/500 server errors) and the base site (background, header, footer, etc.)

The app folder ("crisis_app") contains specific site data. It contains content pages, the search feature, and pretty much everything else that isn't site-wide.

XML Schema

<tbd: Aaron>

In order to facilitate efficient transmission of data between systems, it is often helpful to define a structure for the data such that it can be imported and exported efficiently in an automated fashion with as little human involvement as possible.

The Data Format

The first step in choosing such a structure is to decide on a data format. Common formats would include JSON, YAML, and XML. There are various tradeoffs to each.

JSON
JSON is designed to be lightweight and easily parsed. It assumes several implicit types including the following:

- Strings
- Numbers (though it doesn't distinguish between integers vs. floating point)
- Lists
- Maps (or dictionaries, or associative arrays, or records, or whatever you'd like to call them -- the keys must be stings)

JSON is actually a strict subset of some common scripting languages including JavaScript (from which it gets its name), and Python. JSON cannot represent references to other items in the document, multi-line strings, and it is relatively inflexible about syntax. For instance it does not allow single-quoted strings or trailing commas on the final item in a collection.

While this may seem limiting, it turns out to strike a good balance between simplicity and power, and it is gaining popularity in a number of arenas, including configuration, archiving, and probably most prevalently, API's.

YAML
YAML aims to be human readable above all. It is pleasant to write and read, and it includes implicit types like JSON (though many more). However it is very difficult to parse (compare the lines of code in the PyYAML versus simplejson Python libraries). This also makes it difficult to standardize implementations and eradicate security issues. While its complexity makes it undesirable for something like an API that must be unambiguous, it is frequently used for configuration files.

XML
XML is extensively standardized, available in nearly every language, and very well understood by the industry. It is the most expressive of the formats presented here, which explains its applications to everything previously mentioned, but even more intricate problems such as user interface definition (see .xaml files, .xib files, glade files, and arguably web apps written in xhtml).

XML is also the most verbose of the formats. If JSON is a gentleman's handshake, then XML is a 100-page legal document. While JSON and YAML have implicit types, XML leaves that function up to the application.

If the project team could have chosen a data interchange format for this assignment, it would have used JSON since it is the most convenient and more than expressive enough for our needs, however were required to use XML.

The Data Schema
The format describes the structure of the data at the lowest level. It standardizes the beginning and ending points of different areas of the data and the means by which hierarchies are expressed. If one were to only agree on a format, then it would be easy to convert the text data into data structures in

memory such as objects, strings, and integers, but then all of the semantic information would be lost. For example, consider the following two snippets of XML:

Snippet#1:

```
<phone type=”cell”>812-764-8203</phone>
```

Snippet #2:

```
<PhoneNumbers>
  <Cell value=”812-64-8203” />
</PhoneNumbers>
```

As a human, it is relatively easy to infer that in both cases we’re looking at a single cell phone number, and the number is 812-764-8208. If however one wanted to write software to translate XML (or any format for that matter) into a list of strings of phone numbers, it would be intractable to handle all of the possible ways this could be expressed. This issue calls for a rigorous definition of not just the format, but also the structure of the data: a schema.

XML has well established standards for defining and tools for validating schemas. The team considered two options for schema definition in this class, DTD and XSD.

DTD

Document Type Definitions, or DTD’s are annotations at the top of an XML file (or a least referenced from the top of an XML file) which specifying the schema of the document. They have a different syntax than XML, and are generally viewed as a less-powerful precursor to XSD’s.

XSD

XML Schema Definitions or XSD’s share DTD’s purpose in defining a schema for an XML document, however they are written in XML syntax. XSD’s came after DTD’s and were meant to make up for DTD’s shortcomings in expressiveness, specifically they provided namespaces.

Schema Development Process

The class decided to proceed with an XSD schema instead of DTD because it is believed that student will more likely encounter XSD’s in industry jobs. In order to facilitate communication between groups, the schema was posted in a public GitHub repo, and issues were discussed openly. The class developed a system where any proposed changed up-voted by members from three different groups was ratified, and then merged as soon as someone volunteered to implement it.

Schema Description

The schema starts with a root ‘WorldCrises’ element. Since the team is required to track crises, people involved, and organizations, it represented each of these with the Crisis, Person, and Organization elements respectively. These can be referred to these as the “main elements.” Each main element is tracked by an “ID” attribute which serves as a substitute key for the underlying entity. Main elements are also required to have a “Name” attribute. The team did not try to enforce anything more in-depth, such as differentiating between first and last names.

Per the project requirements, each main element also has fields that are specific to information that must be gathered for that type of entity. For example, one requirement includes gathering information on organizations’ history, so the Organization element has a History child.

List Item Elements for Complex Data Attributes

In cases where a child of a main element could have multiple items, one used <li> children to delineate:

```
<Organization>
  <!-- ... other info ... -->
  <Location>
    <li>Uttarakhand, India</li>
    <li>Himachal Pradesh, India</li>
  </Location>
  <!-- ... other info ... -->
</Organization>
```

Associations Between Entities

The project team represented associations between entities by starting each main element with a child for each of the other two main elements, named by the plural. So the first two children of the “Crisis” element is the “People” and “Organizations” elements. Similarly, the first two children of the “Person” main element are “Crises” and “Organizations”. These relationship elements’ children served as the foreign keys to the other main elements:

```
<Crisis ID="CRI_NRINFL" Name="2013 Northern India Floods">
  <People>
    <Person ID=”PER_MNSNGH”>
  </People>
  <!-- ... other info ... -->
</Crisis>
```

These model many-to-many multiplicity in the associations. It is important to note that these relationships are implicitly two-way, so if one has the above link from the Northern India Floods to Manmohan Singh, one should necessarily have a link from Manmohan Singh to the floods. However there is nothing in the XML schema to prevent this, so validating the schema alone does not guarantee the integrity of the data.

Links

Each main element can have a “Common” child element that contains links to other information. The links fall in two categories:

- “href” links - these are the sort of thing one might expect to be in an anchor tag (i.e. <a href=”...”>...</a>) and when the user clicks it the browser navigates to a new webpage.

“embed” links - these tend to be media such as images that would be embedded in the page, though not necessarily from the site’s domain.

The project team did not really see the distinction between these two types of links, since a URL is a URL and the containing tag classifies the linked content, but the team found it more important to support the first working implementation of the schema than to bike shed the details. The link elements all follow the convention of using <li> elements to provide multiple children:

```
<ExternalLinks>
  <li href="...">Descriptive text</li>
</ExternalLinks>
<Images>
  <li embed="..." text="Descriptive text" />
</Images>
```

The team important to distinguish as little as possible between the links in order to increase the variety of data our system could describe while limiting the complexity. If the team would have imposed the arbitrary restriction that links had to be to certain content types (i.e. HTML for those in the ExternalLinks tag, images for links in the Image tag, etc.) the team would have run into problems when it encountered media like video, which has a very diverse and poorly-specified range of techniques for embedding. Instead the team just treated everything as a URL, and then allowed the view layer of the application to process the link in such a way that it could display or embed the data correctly.

An example of this is YouTube videos. These videos are not link-able in the same manner as images, where one can simply have an <img> element with a “src” attribute, rather they must be embedded as <iframe> elements. If the team wanted to encode this in the XML schema, the team would have to add extra elements or attributes for the embedding information and some way to distinguish that from, say, MPEG-4 videos which can be embedded in <video> tags in modern browsers. This complexity jeopardizes the process of standardizing the XML schema among the different groups, since it adds time and is could easily be implemented correctly if the team forgot any details. Alternatively the team could recognize that every YouTube video link contains the ID of the video:

```
http://www.youtube.com/watch?v=dQw4w9WgXcQ ├──> YouTube ID
```

And transforming that into the appropriate <iframe> HTML snippet would be relatively easy:

```
<iframe src="//www.youtube.com/embed/dQw4w9WgXcQ"></iframe>
```

Import and Export

One of the requirements includes providing “Create [a password protected] import/export facility from the XML into the Django models and back using ElementTree.” This was an extremely poorly specified requirement, and the team found out later this was code words for an HTML upload form to import the XML and a URL on the project website where an XML export could be downloaded.

Import Facility
The user interface for the import facility is a standard Django form and the associated template and view. The form has a single file-input field, and for validation it follows the Django convention of overriding the "clean()" method, where it calls the MiniXSV method for validating the XML.

The view is authenticated with Django's login_required decorator. The same view method is used whether the user is arriving at the page to upload the form or submitting the form after they've already arrived. When the user submits new XML, the website attempt to store it in the database, invalidate the cached copy of the generated XML, and then redirect the user to the XML download page. Any errors are added to the form and rendered on the page.

The conversion from XML to database models is in a separate Python module. The module primarily consists of classes that represent conversions from one of the main elements to a Django model instance and a "convert()" function that handles reading the XML, making the conversion instances, and then saving the models to the database. The conversion code makes use of the dynamic nature of Python by dynamically choosing methods to invoke and introspecting the class name to choose the appropriate database model.

Export Facility
The user interface for the import facility is a view that generates the XML and serves the response with an 'application/xml' mimetype. The view implements an ad-hoc caching method instead of either HTTP caching or Django's caching framework.

The export facility's conversion code is in the same module as the import conversion. As the requirements specify, it uses Python's standard library ElementTree module to build the XML structure. It makes use of object-oriented design principles by defining a base conversion class and then leveraging inheritance to customize its behavior based on the different main elements.

Django Models
<tbd: Dave>

Stephen says the models are equiv to the UML, SQL dump

Get Stephen's comments

Event Model

The event model stores information about events which led to world crises. The primary key uses an ID field (an integer which has no character limit, practically speaking) instead of an XML ID due to the six-character limitation invoked by XML ID ([A-Z]{6}). Django TextFields are limited to 4096 characters to prevent abuse of server resources. Events contain a name identifier corresponding to the event's name (e.g. "Hurricane Sandy"), which provides a natural way to refer to them. Event objects contain fields for the event’s name, type, location of occurrence, date/time, human impact, economic impact, resources used, and aid provided.

Person Model

The person model contains information about influential persons' responses to events which led to world crises. The primary key uses an ID field like in the event model. Each person object contains a name identifier corresponding to the person’s name (e.g. “Barack Obama”). Person objects contain fields for the person's name, role, primary location, and events with which they are associated.

Organization Model

The organization model contains information about organizations which were influential during events which led to world crises. The primary key uses an ID field like in the event model. Django TextFields are limited to 4096 characters to prevent abuse of server resources. Each organization object contains a name identifier corresponding to the organization’s name (e.g. “World Maritime Organization”). Organization objects contain fields for the organization's name, type, location of operation, contact, history, associated events and associated people.

Embed Model

The embed model contains all information not directly hosted by the server. URL inputs are limited to 255 characters because Django does not allow CharFields or TextFields over 255 characters to be unique. Each embed object contains a name identifier corresponding to its URL. Embed objects contain the URL to various artifacts that help describe an event, person or organization (e.g. images, videos and maps).

UML

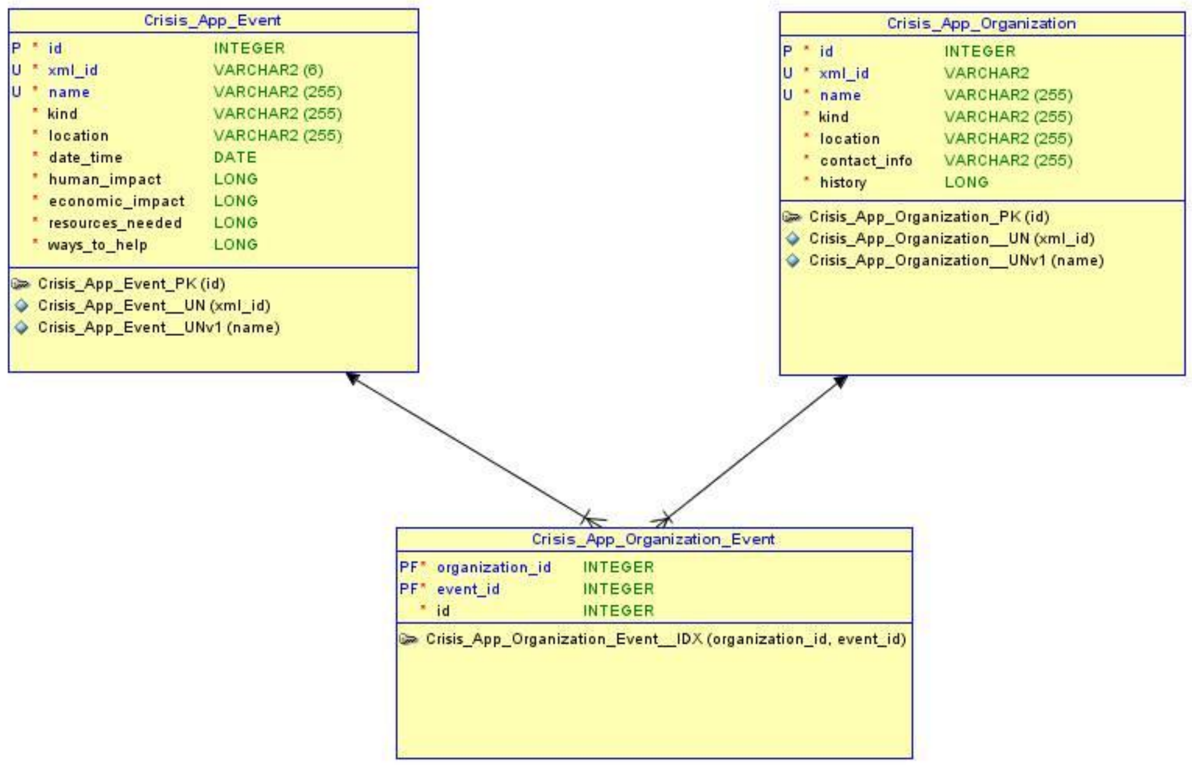
<tbd: Qunvar, Dave>

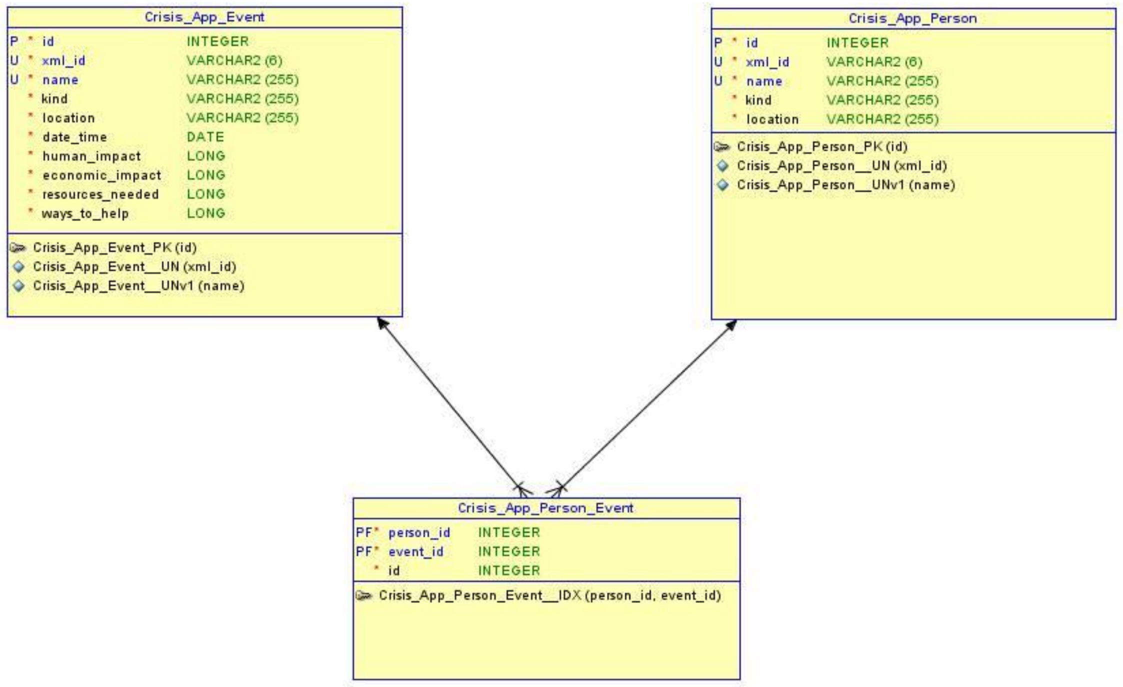
This is the big picture view of the models

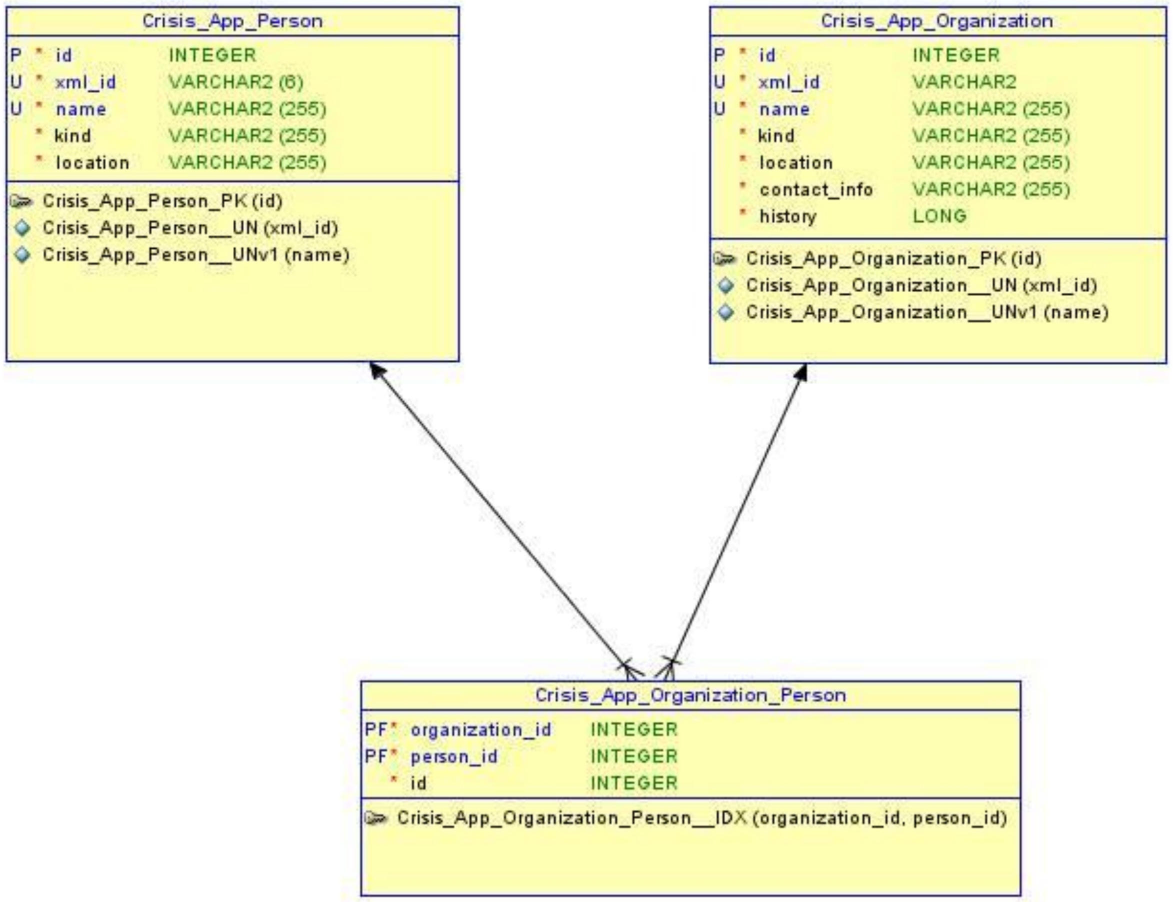
UML Models

<note from Jason: The following UML models are based off of screenshots Qunvar sent me. We’ll have to go through them and figure out formatting and write-ups. Should UML descriptions be sufficient?>

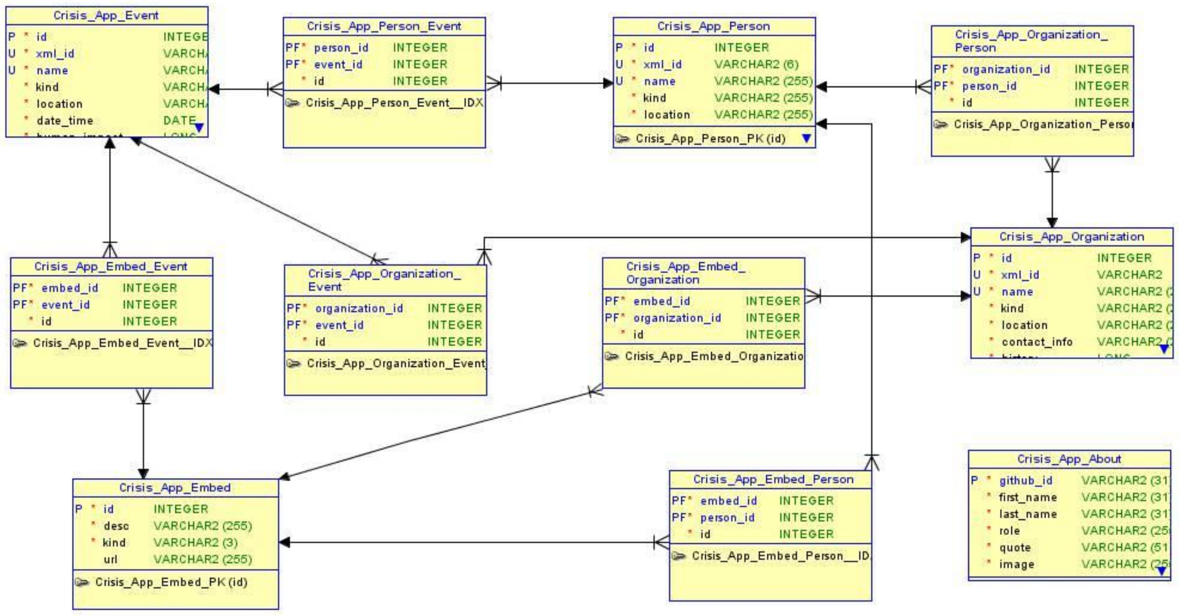
<Q from Qunvar: Do you want like individual descriptions for each picture or should I just try to incorporate it into my tech report description of the uml?>


Crisis_App_Event
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
* date_time DATE
* human_impact LONG
* economic_impact LONG
* resources_needed LONG
* ways_to_help LONG
Crisis_App_Event_PK (id)
Crisis_App_Event__UN (xml_id)
Crisis_App_Event__UNv1 (name)
Crisis_App_Organization
P * id INTEGER
U * xml_id VARCHAR2
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
* contact_info VARCHAR2 (255)
* history LONG
Crisis_App_Organization_PK (id)
Crisis_App_Organization__UN (xml_id)
Crisis_App_Organization__UNv1 (name)
Crisis_App_Organization_Event
PF* organization_id INTEGER
PF* event_id INTEGER
* id INTEGER
Crisis_App_Organization_Event__IDX (organization_id, event_id)

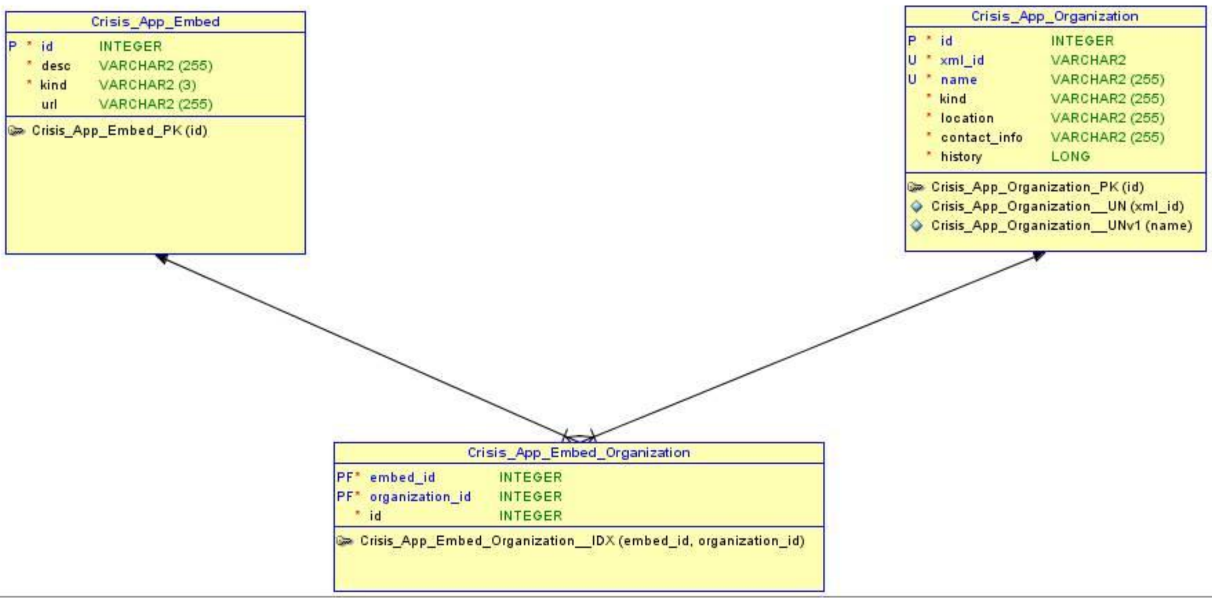

Crisis_App_Event
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
* date_time DATE
* human_impact LONG
* economic_impact LONG
* resources_needed LONG
* ways_to_help LONG
Crisis_App_Event_PK (id)
Crisis_App_Event__UN (xml_id)
Crisis_App_Event__UNv1 (name)
Crisis_App_Person
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
Crisis_App_Person_PK (id)
Crisis_App_Person__UN (xml_id)
Crisis_App_Person__UNv1 (name)
Crisis_App_Person_Event
PF* person_id INTEGER
PF* event_id INTEGER
* id INTEGER
Crisis_App_Person_Event__IDX (person_id, event_id)

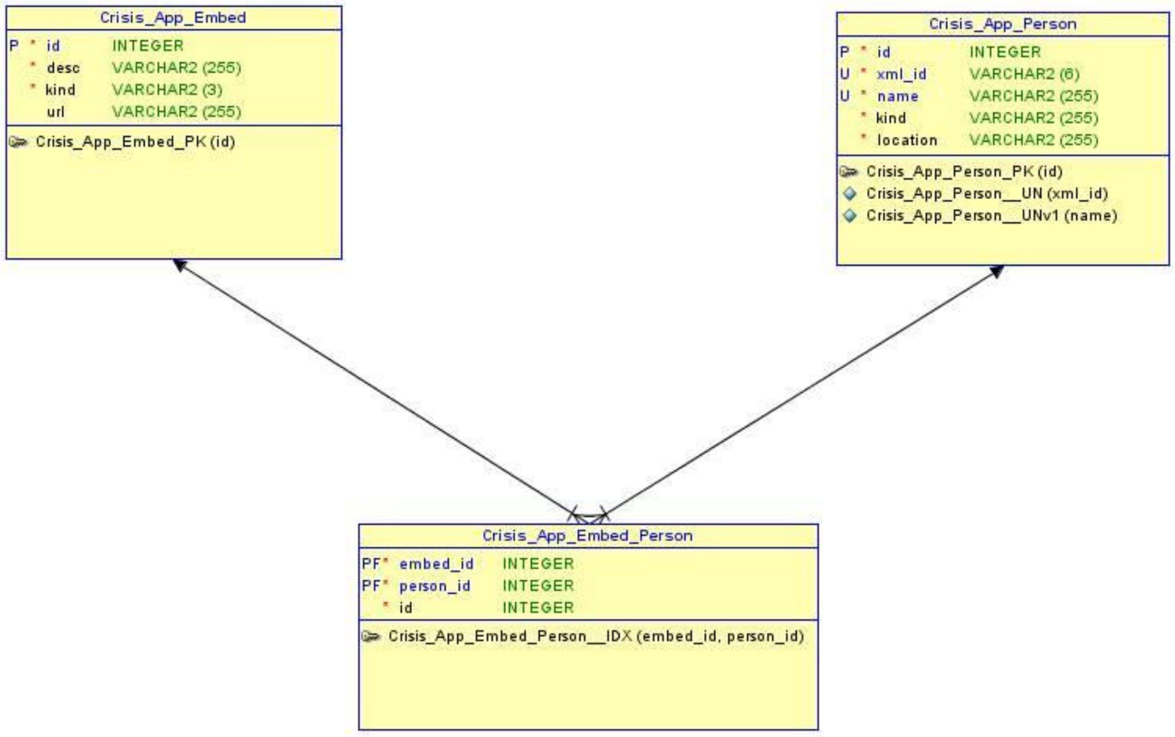


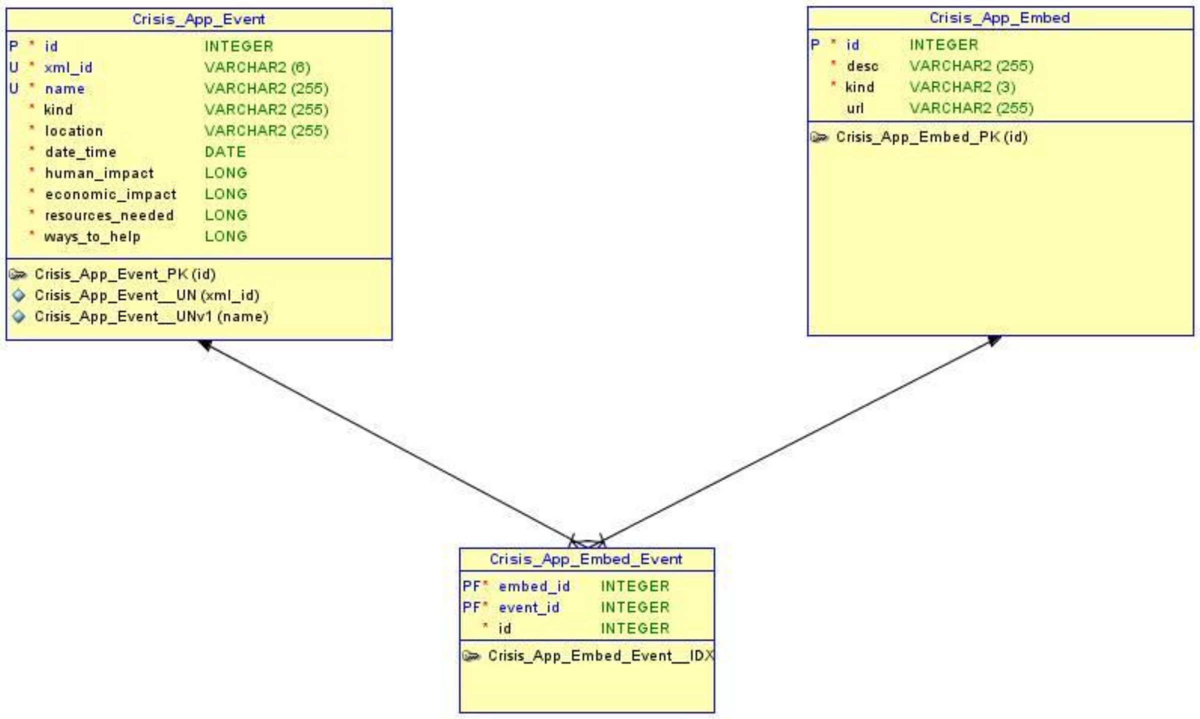
Crisis_App_Person
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
Crisis_App_Person_PK (id)
Crisis_App_Person__UN (xml_id)
Crisis_App_Person__UNv1 (name)
Crisis_App_Organization
P * id INTEGER
U * xml_id VARCHAR2
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
* contact_info VARCHAR2 (255)
* history LONG
Crisis_App_Organization_PK (id)
Crisis_App_Organization__UN (xml_id)
Crisis_App_Organization__UNv1 (name)
Crisis_App_Organization_Person
PF* organization_id INTEGER
PF* person_id INTEGER
* id INTEGER
Crisis_App_Organization_Person__IDX (organization_id, person_id)



Crisis_App_Event
P * id INTEGE
U * xml_id VARCH
U * name VARCH
* kind VARCH
* location VARCH
* date_time DATE
Crisis_App_Person_Event
PF* person_id INTEGER
PF* event_id INTEGER
* id INTEGER
Crisis_App_Person_Event__IDX
Crisis_App_Person
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
Crisis_App_Person_PK (id)
Crisis_App_Organization_ Person
PF* organization_id INTEGER
PF* person_id INTEGER
* id INTEGER
Crisis_App_Organization_Perso
Crisis_App_Embed_Event
PF* embed_id INTEGER
PF* event_id INTEGER
* id INTEGER
Crisis_App_Embed_Event__IDX
Crisis_App_Organization_ Event
PF* organization_id INTEGER
PF* event_id INTEGER
* id INTEGER
Crisis_App_Organization_Event
Crisis_App_Embed_ Organization
PF* embed_id INTEGER
PF* organization_id INTEGER
* id INTEGER
Crisis_App_Embed_Organizatio
Crisis_App_Organization
P * id INTEGER
U * xml_id VARCHAR2
U * name VARCHAR2 (
* kind VARCHAR2 (
* location VARCHAR2 (
* contact_info VARCHAR2 (
Crisis_App_Embed
P * id INTEGER
* desc VARCHAR2 (255)
* kind VARCHAR2 (3)
url VARCHAR2 (255)
Crisis_App_Embed_PK (id)
Crisis_App_Embed_Person
PF* embed_id INTEGER
PF* person_id INTEGER
* id INTEGER
Crisis_App_Embed_Person__ID
Crisis_App_About
P * github_id VARCHAR2 (31
* first_name VARCHAR2 (31
* last_name VARCHAR2 (31
* role VARCHAR2 (25
* quote VARCHAR2 (51
* image VARCHAR2 (25


Crisis_App_Embed
P * id INTEGER
* desc VARCHAR2 (255)
* kind VARCHAR2 (3)
url VARCHAR2 (255)
Crisis_App_Embed_PK (id)
Crisis_App_Organization
P * id INTEGER
U * xml_id VARCHAR2
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
* contact_info VARCHAR2 (255)
* history LONG
Crisis_App_Organization_PK (id)
Crisis_App_Organization__UN (xml_id)
Crisis_App_Organization__UNv1 (name)
Crisis_App_Embed_Organization
PF* embed_id INTEGER
PF* organization_id INTEGER
* id INTEGER
Crisis_App_Embed_Organization__IDX (embed_id, organization_id)


Crisis_App_Embed
P * id INTEGER
* desc VARCHAR2 (255)
* kind VARCHAR2 (3)
url VARCHAR2 (255)
Crisis_App_Embed_PK (id)
Crisis_App_Person
P * id INTEGER
U * xml_id VARCHAR2 (6)
U * name VARCHAR2 (255)
* kind VARCHAR2 (255)
* location VARCHAR2 (255)
Crisis_App_Person_PK (id)
Crisis_App_Person__UN (xml_id)
Crisis_App_Person__UNv1 (name)
Crisis_App_Embed_Person
PF* embed_id INTEGER
PF* person_id INTEGER
* id INTEGER
Crisis_App_Embed_Person__IDX (embed_id, person_id)



UI

<Stephen>

Introduction

The World Crisis Database strives to provide users an easy and intuitive interface for browsing the content of the site. A well-designed structure either separates or interleaves the navigation from the main content as appropriate. A good user interface should also provide a safety net for bad data. Because this site will need to import information from external sources uncontrollable by the authors of the application, invalid data may arise. The ordinary user should see a page which adapts to the content, regardless of the validity of the data.

Navigation via Links

The home page greets users with a full-width splash screen that allows browsing of some of the most popular events, people, and organizations of the time. The splash screen is interactive, which means that users may choose to focus on one of the specific categories of the site (events, people, or organizations), or focus on a specific event, person, or organization. Users interested can pursue further reading of a given event, person, or organization with a link to that topic's content page.

Three separate pages accessible from the navigation bar anywhere in the site provide the database's complete selection of events, people, or organizations. This information is displayed in a tiled format, with focus on displaying a general overview of each tile by means of a short description and (if applicable) an image. Activating a tile by means of clicking on the one of interest takes users to a content page with more information regarding that topic.

Content pages are the most descriptive method of display. They provide users with a complete overview of any one topic, with information categorized for ease of access. If readers want to view more about a particular topic, they can access the topic's related events, people, and/or organizations. They also can open links to external sources given in online articles, images, videos, and social media sites that provide the latest news feeds on a given topic. To maintain consistency, navigation away from the main site will result in opening a new window or tab, whereas navigation within the site will open the page in the same window or tab.

Navigation via Search

The website also provides an intuitive search interface. On any page, users have immediate access to a search area in the navigation bar. When conducting a search, the user may choose to search through all events, people, or organizations. The user can filter the post-search results into any of the three categories. The search tool detects the active category filter and defaults to this category on subsequent user searches. For example, if People page is active and the user types in a word, the search tool will look for matches in the People category. S

Searches will redirect the user to a search result page with tabbed results, where each tab corresponds to one of the main categories of the site (e.g. event, people, and organizations). Each search result is presented as a tile with the name of the result, a briefing of the topic, and further description which is limited in length to prevent excessively large tiles. At the moment, an image is not provided since not all results will have images, and this inconsistency blemishes the aesthetics of the search results page.

Within each tab of the search, the user may also consider sorting the category in a specific way. By default, each category is sorted in the way that seems most natural (events are sorted by their latest occurrence, people and organizations are sorted by name descending). The user has the ability to change the sorting method by means of a button to the right of the tabs.

<tbd> Stephen has new feature to add

Implementation

Import/export facility

<tbd: Aaron>

include links and explanations of the import/export modules you've developed.
Stephen says: for design, discuss the piazza discussions; for imp, talk about how you coded it

Testing unittest

<tbd: Dave>

Description

The Unit tests for this project cover all the functionality of the JSON creating code, XML parsing and interpreting and validating code, database reading code, and import/export code for interfacing with the database.

The first suite of tests covers JSON code. The first set checks for proper creation of parent nodes that allows for correct reading of dictionary-type data. Other set tests adding children to a parent node and similar calling conventions for getting the data back out of those nodes. The final set tests the make_json function and its compatibility with the json.loads command from the json library.

The second suite of tests covers XML interpreting and validating code. The first set verifies that the correct data is being used and that the .strip() class function (which removes trailing and leading whitespace) results in the correct tag. The second set verifies that the XML remains valid after multiple parse iterations.

The third suite of tests handles database reading code. These tests verify that the website correctly pulls and interprets the XML from the database.

The last suite of tests is for the import/export code. The first test checks for a proper redirect response code to send the user during login attempts. The second test verifies correct response codes, including different response codes in the same declaration.

The project test suite uses a sqlite3 backend instead of MySQL for performance reasons. Sqlite3 runs the tests 5 to 10 times faster than MySQL because it does not save the results to disk. The project codebase follows good design principles of dependency injection, which makes it possible to swap the database engine out and still achieve good test coverage.

How to run the unittests

To run the unit tests - execute the following command in the main directory:

make test

The makefile invokes the manage.py file that is located in the root project directory.

<Issues To Address>

- documenting the results of those decisions
- A technical report is meant to explain your design to someone who is unfamiliar with the project
- <tbd: Create diagrams with captions>