World Crisis Database

Phase 2 Technical Report

University of Texas at Austin
Computer Science
CS373 Software Engineering

Team Crisix

Tom Torres
Ejenio Capetillo
Jonathan Velasquez
Enam Ayivi
Matt Thurner
Dalton Schmidt

Introduction

Problem

On any given day, some crisis in the spotlight blazes through the media, only to be forgotten when a new crisis spawns. Technology has enabled us to communicate swiftly and effectively, and the result is a short attention span and desensitization of our society as a whole. Given seven billion people, unpredictable weather patterns, and general instability in terms of worldwide economy and government, something is bound to go wrong at any given moment. There's always some event, some crisis, wreaking havoc on all those who pretend themselves masters of their surroundings. Whether it be an environmental disaster, terrorist attack, or nation in revolt, an influx of crises leads to a temporary focus on one, followed by an immediate shift to another. It's frighteningly easy to forget about all of the past and present turmoil, as chaos displaces more chaos. The key to this problem lies in education, not persuasion, therefore it is important that this data is stored and can be easily accessible by everyone, so that anyone with the means and inclination to help can do exactly that.

Mission

The goal of the Crisix Database is to supply relevant and current information on world crises in an effort to "help us help ourselves." Among this information, a synopsis of each crisis, its impact on society, along with organizations and people involved, are to be provided. This information will be easily accessible on the Internet with each item (crisis, person, or organization) having a unique, user-friendly page.

Each page will consist of a summary, images, videos, citations, external links, and various other pieces of related media and information, providing users with a wealth of resources related to a crisis. A successful implementation will provide a perennial framework that gives users a hub to explore numerous aspects of a crisis, pointing to ways in which one can help provide relief to those affected.

Use Case

Outlined below are some typical use cases via Alistair Cockburn that follow the UML "actor" paradigm:

Title: Research a crisis
Goal: Search WCDB for information about a particular crisis

Actor: A computer user with Internet access
Scope: Have a successful query that is in the database
Level: User-goal, sea-level, return a successful query
Story: The user goes to the WCDB and queries a crisis, person, or organization. On a successful search, the user will be directed to a page containing the relevant data. The data will be item specific but will relate to some crisis. From here, the user can click on links that contain information on a related item, or click on links to external websites, or even multimedia.

Title: Stay on track
Goal: Watch a crisis page for updates
Actor: A user directly affected by a crisis
Scope: Allow for imported data, update page dynamically
Level: User-goal, sea-level, render dynamic page
Story: The user has a relative that is experiencing the fallout of some crisis. The user does not have any contact with the relative. The user stays updates as more data is imported into the database as the events of the crisis unfold.

Title: Lend a helping hand
Goal: Provide relief to crisis victims
Actor: A user with the resources to help
Scope: Display appropriate content
Level: User-goal, sea-level, provide user direction
Story: The user has the resources to help victims affected by some crisis. However, the user doesn't know how to help or which organizations are involved. The user navigates to the crisis page and accesses appropriate information.

To reiterate, providing a "fountain of knowledge" for the sake of knowledge is a fine goal, but this is only a secondary concern. The main goal of the Crisix Database is to supply information without suffocation, to educate and direct users without robbing them of the opportunity to help.

Design

XML Schema

The information reflected in the Crisix Database is stored in an XML file, the design of which is validated using an XSD schema. This schema employs ID/IDREFs to avoid data duplication and to denote existing relationships between crises, people, and organizations. Each ID is ten

characters long, with the first three fixed (CRI, ORG, or PER) followed by an underbar and six uppercase letters, used to identify each entity. An entity ID is required and must be unique.

The schema groups crises, people and organizations together, in that order. Attributes shared by all three types of entities include name, location, kind, and summary. Attributes specific to crises include date and time, human impact, economic impact, resources needed, and ways to help while organizations maintain history and contact information. Apart from these attributes, there is also common embedded data, (objects of type WebElement) such as images, videos, maps, and social network feeds (e.g. Facebook, Twitter).

The schema also comprises of a ListType element, denoted by the `<li>` tag. ListType is simply a set of `<li>` subelements. Each subelement can store text, a hyperlink and/or an embed link to multimedia such as images, videos, citations, and feeds. The optional text attribute defines information about the corresponding embedded object.

There are cases when not all attributes apply to a given crisis, person, or organization. The schema is designed such that many attributes are optional (done by employing the “minOccurs” and “maxOccurs” constraints), which is crucial for validation of an XML instance. Given that most attributes are, in fact, optional, it is important to remember the goal of the Crisix Database in providing users a plethora of information and resources. To encourage a fair amount of data mining, many of the display fields on the Crisix website will appear blank (as opposed to not appearing at all).

Validating an XML instance can be accomplished in two ways. The first is by importing data into the database via the Crisix website (explained below). The second is by running the following terminal command:

```
xmllint --noout --schema WCDB2.xsd.xml WCDB2.xml
```

If the XML instance validates, the following message is displayed:

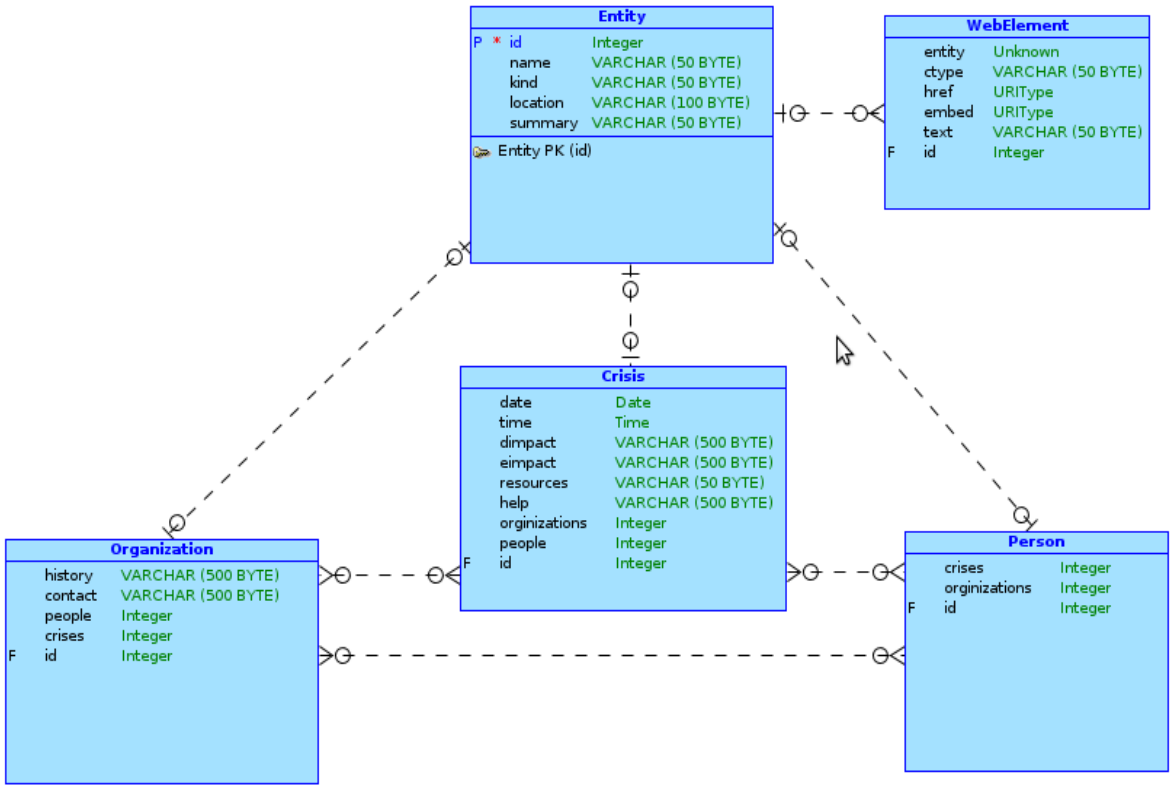
```
WCDB2.xml validates
```

Otherwise, the terminal will output the line number of the infringing XML. Common errors are syntax-based and are exclusively XML rules, but are easy to find and can be dealt with quickly.

Django Models

A set of Django models are used to represent information in the database. In terms of object-oriented design, the models (classes) exhibit inheritance. The models consist of a parent class, named Entity, which contains all of the generic attributes. The direct children of the Entity class are the Crisis, Organization, and Person classes. The end result is intuitive: constructing a Crisis, Organization, or Person implicitly constructs an Entity object. The way in which this translates into the SQL database is similar.

Crisix Logical Model

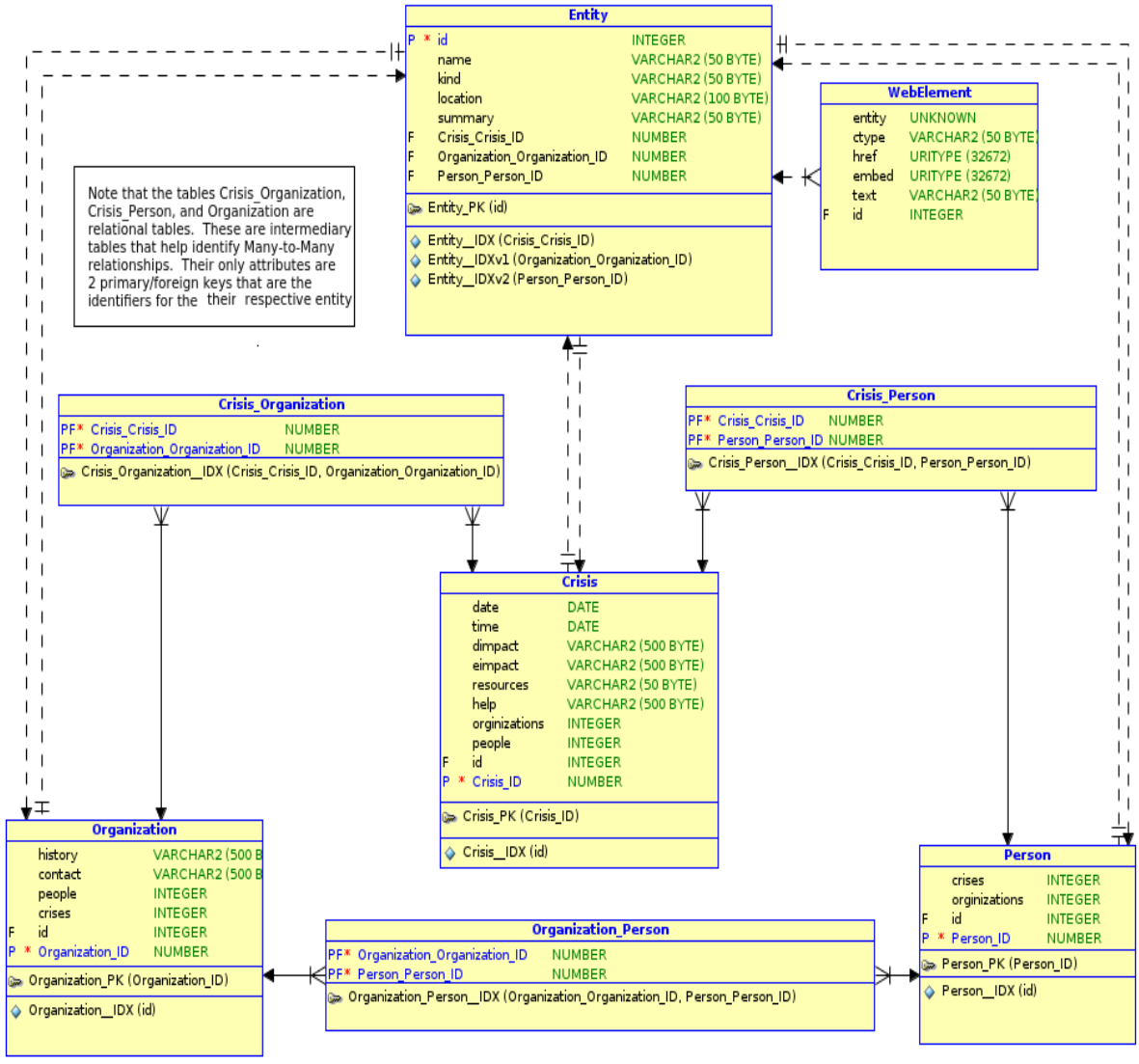


In addition to the Crisis, Organization, and Person tables, a fourth table, Entity, is created. This table is entirely separate and relates to its children via unique one-to-one relationships. An ID in the Entity table is unique and corresponds to some Crisis, Organization, or Person. Each Entity has many WebElements (one-to-many relationship) and each Crisis, Organization, and Person

relates to zero or more Crises, Organizations or People (many-to many relationships).

Below is the relational model which corresponds to the tables created by Django when the web application is deployed.

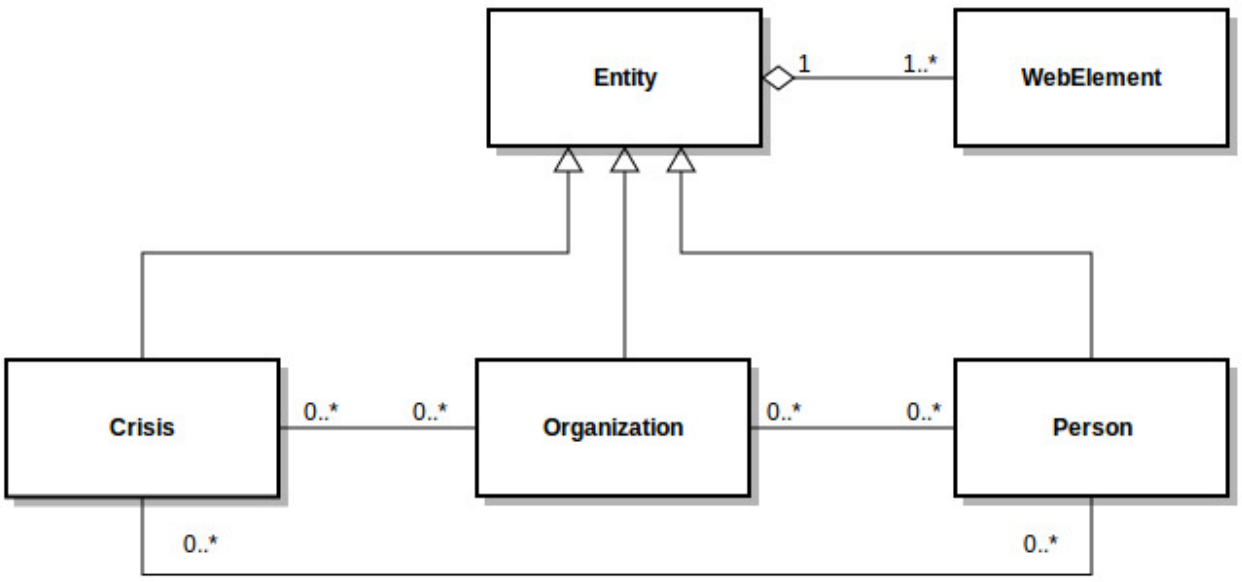
Crisix Relational Model



As demonstrated above, a many-to-many relationship is represented by a third table that serves as an index between the two parent tables. These tables are not represented as Django models, since they are generated implicitly when specifying a many-to-many relationship. In Django,

related entities may be accessed with a hidden field, which the plural form of each entity name. For instance, given a Crisis object c, related Person objects can be accessed via a call to `c.people.all()`.

UML



The UML model above demonstrates the multiplicity and associations that exist between Crisix models. Just as in the logical and relational models, the UML model shows that Crisis, Organization, and Person classes inherit directly from Entity. Each of these items contains a many-to-many relationship with other entities of differing types (denoted by the 0..*). Each Entity contains an aggregate relationship with one or more WebElements, denoted by an unfilled diamond. This aggregate relationship can be best understood by acknowledging that each Entity has many objects of type WebElements and the lifetime of each WebElement is directly related to the lifetime of its corresponding Entity (strong aggregation).

HTML and Bootstrap

The Crisix website utilizes a styling framework via **Twitter Bootstrap** for its core design. Twitter Bootstrap provides the perfect balance between a speedy learning curve and coherent results. More specifically, Bootstrap provides a set of keywords that can be added to different elements in the HTML structure of each page. For instance, one can define multiple links, each as a `<li>` element. The block of `<li>` elements can then be surrounded by a set of `<div>` tags with "navbar" and "navbar-fixed-top" as attributes. This tells Bootstrap to format the block

as a left-sided navigational bar.

In the backend, Bootstrap retrieves its styling guidelines from three predefined folders, stored in the static folder of the project root directory:

img/
css/bootstrap.css
js/bootstrap.js

These files in these directories correspond to images, styling, and javascript animations, respectively.

Overall, the Crisix site page is divided into four HTML blocks (using `<div>` tags). These blocks are as follows: the aforementioned sidebar, a top right "hero-unit" (Bootstrap lingo for a header that strongly stands out), a container for the main content of each page, and a footer. Within the main container, there exists 'mini-bootstrap' containers that provide virtual grids in which page elements reside. As an example, videos and photos for entities are displayed in separate containers within the main container.

Dynamic Page Rendering

Each page is rendered dynamically using Django templates. Each entity type uses its own template, but all templates inherit from a base template. This guarantees a unified style and design across all pages. The URL for an entity is the entity type followed by the last six characters of its unique ID.

When a URL of this form is requested, the appropriate view extracts the corresponding information from the database. Because calculations (e.g. string manipulations) and function calls are limited within templates, each view is responsible for preparing the information so that it can be readily displayed to the user.

This includes calculating URLs for related pages, transforming database columns so they are human readable, and preserving the context of the original request object. Each view also converts photos into thumbnails so that the rendered page doesn’t overflow or contort. After the data has been processed, the view returns a response object, passing in a dictionary with the processed information. The template receives this information and trivially displays it to the user.

Aside from entity templates, there exists a template for the utility page. This template is unique in

that display information is not preprocessed. The dictionary object passed to this template does not contain data, rather it is in the form of strings as template arguments. The utility page will render a message dependent upon the arguments in the dictionary. The only exception is if a form is passed, in which case the utility template will render the form. This is to accommodate import, which renders a file upload form. The utility template is polymorphic and is designed in such a way that it mimics a fairly simple administrator interface.

User Interface

Crisix was originally deployed on the UTCS host 'z', but is now served by Heroku. The full website can be reached at **crisix.herokuapp.com**.

In terms of content, each page is divided into three main areas: the previously mentioned hero-unit (hereinafter 'hero'), sidebar, and body. Hero is used to display the 'main idea' of each page, while the body is more elaborate. The sidebar generally displays supplementary information for each entity, with the exception of the Crisix index. On this page, the sidebar displays all entities stored in the database as hyperlinks. In the next phase, this will be migrated to the navigation bar at the top of each page.

Clicking on a link in the sidebar renders a new page for the requested entity. As mentioned previously, this page contains related links, which allows for easy navigation between entities. A typical entity page contains a title and summary (inside hero), followed by all of the information stored in the database for that entity.

If applicable, a page may also contain WebElements (social media feeds, maps, videos, and photos). These have been strategically placed so that each page is both aesthetically pleasing and pregnant with information. To maintain the same look and feel for each page, only two videos and a maximum of three photos are displayed for each entity. In the next phase, each page will also link to additional available photos and videos to accommodate merged data.

Photos are processed by **PIL** (Python Image Library) and converted into thumbnails that link to the original image. This is done lazily. Provided that the supplied hyperlink in the database is valid (the file exists and is an image), the photo will be downloaded, resized, and cropped upon each page request. This aids in presenting a clean, unified interface without overflow.

```
def thumbnail(e):
      ...
```

```
    tmp = os.path.join(settings.THUMB_ROOT, t['file'])
    if not os.path.exists(tmp):
        urlretrieve(t['embed'], tmp)
        im = Image.open(tmp)
        im.thumbnail(...)
        th = im.crop((0, 0, 180, 180))
        th.save(tmp, 'PNG')
    return thumbs
```

The module **urlretrieve** performs the actual download. A thumbnail is only generated if it doesn't already exist. Therefore, thumbnails are effectively cached. The cache is cleared on import.

Another area of interest is the utility page. Though minimal, this page is for general database management. A user is allowed to run unit tests and export data from the database to a file, and an authenticated user is further allowed to import data into the database.

Running unit tests is as simple as clicking the generated link in the sidebar. The test output is streamed to the user and displayed in hero. This was accomplished using the module **subprocess**. When a request is made, the corresponding database view forks a child process that starts the test runner. Due to database restrictions imposed by Heroku, an alternate test suite (one that neither creates nor destroys databases) is used. As with any shell process, the child process output is piped to `stderr` and `stdout`. The database module converts `stderr` into a generator, which is used to construct an HttpResponse object which is returned to the user.

```
def results(request):
    cmd = [...]
    process = subprocess.Popen(cmd, ...)
    return HttpResponse((c for c in capture(...,
process)))
```

The HttpResponse object consumes the generator, streaming test output in real-time. Because an HttpResponse object cannot render a template, achieving the base style requires a small hack. The generator function first reads and yields the contents of an HTML/CSS file. Because the default mimetype for an HttpResponse is **text/html**, the desired result is achieved. The generator then iteratively yields output until a **StopIteration** exception is raised.

```
newlines = ['\n', '\r\n', '\r']
def capture(request, process):
    yield open(...).read()
    while True:
        out = process.stderr.read(1)
        if out == '' and process.poll() != None:
            break
        if out in newlines:
            yield '</br>'
        if out != '':
            yield out
```

The rub is that the HttpResponse object renders this information on an entirely separate page. The workaround is that the utility page actually renders an `<iframe>` within hero that renders this page.

When the utility page is rendered, hero contains two buttons, import and export, at the top right-hand corner. Clicking the export button prompts the user to download an XML file that contains all of the information in the Crisix Database. Clicking the import button does one of two things. If the user is already authenticated, a file upload form is displayed which allows the user to specify a file for import. Upon submission, an appropriate message is displayed to the user depending on the validity of the file. If the user is not authenticated, a password prompt is displayed. The corresponding Python module used for authentication is **django-lockdown**.

Currently, the **password for import is "django"**.

Implementation

Environment

A virtual environment (via the Python module **virtualenv**) was used for application development. This greatly simplified module use and mollified sudo-user restrictions on UTCS machines. Incidentally, the suggested deployment process for Django on Heroku involves virtualenv. The directory structure is as follows:

cs373-wcdb/

bin/	Binary files.
lib/	Library files.

static/	CSS, Javascript, image files
templates/	Django templates
crisix/	Django project
database/	Database application
media/	Directory for user uploaded files

For local development, the binary and source files exist in one place with appropriate permissions. The git repository, however, simply contains a script that activates the virtual environment. To activate the environment, do:

```
python env.py
```

in the project root. Activating the environment is equivalent to modifying the `PYTHONPATH` variable. However, virtualenv also implements a sandboxed environment in which module installations are possible.

Import

As mentioned previously, a successful import is an implicit validation against the database schema. This, along with user authentication, ensures that the integrity of the database is not compromised.

The import facility uses the modules **ElementTree**, **MiniXsv**, and **minidom**.
Upon proper authenticatication, a FileUploadForm (a Django form with a FileField attribute) is displayed to the user. It is important to note that the style of this form is **determined by the operating system**. Upon submission, the uploaded file is stored as an in-memory file. The file is then written to disk and validated against the database schema. If the submitted file is invalid, an appropriate message is displayed to the user. Otherwise, an ElementTree Element object is returned from MiniXsv's `parseAndValidateXmlInput` method.

This object represents the root of a valid ElementTree that contains all of the information provided by the user. The root is passed in a call to `insert()` which in turn invokes three main handler functions, one for each entity. The upload process for all three is similar. A `getEntity()` function is called with parameters `etype` and `eid`. If `etype` with ID `eid` exists in the database, the object is returned. Otherwise, a new entity of type `etype` and ID `eid` is created, stored, and returned. This purpose of this function is to handle the following invariant for each insertion: the input has been validated, therefore the current entity must exist in

the database following the import.

```
def getEntity(etype, eid):
    e = None
    try:
        e = etype.objects.get(id=eid)
    except etype.DoesNotExist:
        e = etype(id=eid)
    e.save()
    return e
```

Django's `get()` method throws an exception if the object does not exist in the database. This function is a streamlined version of `get()` that uses exception handling to accommodate import. This function is also useful when establishing relationships between entities. As the ElementTree instance is processed, IDREFs are used to get or create related entities that do not yet exist. For example, a Crisis object might reference a Person object that has not been saved to the database. This method will return a new Person object which can be related to the Crisis object via Django's `add()` method. Later, when the Person element is processed, the corresponding Person object is simply retrieved from the database.

A few select attributes, such as an organization's 'history' attribute or 'human impact' for crises are stored as ListType elements using the `<li>` tag in accordance with the XML schema. Because this is also valid HTML, these attributes are simply stored as-is. When it is rendering or exporting this information, the data is simply extracted and displayed without any further processing.

Each handler function makes a call to another handler for common elements. This handler is essentially the same as the others in terms of function with one slight difference. The handlers for main entities append new data, while the handler for common data queries for a specific attribute (either 'href' or 'embed') and writes to the database depending on the result. If the element exists, the insert is a no-op. Otherwise, the data is inserted. The reason for this difference is due to the nature of common elements. Each WebElement object exists as a single entry in the database, and therefore exhibits a fairly small granularity. The attributes specific to each Entity object are a conglomeration of perhaps multiple entries for the same entity.

Export

Export is much simpler than import. The process literally iterates over the entire database,

extracts data, and stores it an ElementTree object. Like import, each entity has a specific handler, and each handler is responsible for extracting appropriate information and invoking a handler for common data. A series of checks is performed to avoid empty tags/attributes. Again, because of the ways certain attributes are stored, some of the information is exported as-is without further processing.

Once the data has been extracted from the database, the export facility generates a response object with a mimetype 'text/xml'. The XML is written to this response (using **minidom**'s writexml, which is able to write to a file-like object with proper indentation) and displayed to the user as an XML file.

Testing

Unit Tests

Testing is performed with Django's built-in app unittests. Test cases are responsible for ensuring that the upload and download procedures execute correctly. This includes validation via MiniXsv, and correctness when pushing to and pulling from the database. For instance, when creating an entity, several checks are performed to assert that the proper relationships are established (as well as backwards relationships) and that the data is durable (it sticks). To run the tests, on the terminal command line, navigate to the project root and do:

```
manage.py test database
```

The unit tests currently consist of three separate testing suites that each verify the correctness of the import, export and view modules. The view testing suite contains three sets of unit tests that verify the HTML pages for each of the three main models (Crisis, Person, and Organization). To properly perform these view tests, sample models are created, populated and related to each other. Each test also utilizes the **RequestFactory** Python module to send dummy requests to a view. This request allows for dummy HTMLs to be created which are then returned as response objects. The content of the returned response objects are checked for proper status codes and for proper HTML strings that conform to the format contained within the templates.

The TestUpload suite verifies that the information inside the XML is properly parsed and placed into the necessary model attributes. ElementTree is employed in these tests on sample XML files, each compose of a few sample entities. The attributes inside the ElementTree objects are then converted into appropriate Django models. The tests check for the presence of various attributes in the models that also exist in the supplied XML. Unit tests in the TestUpload suite

specifically parse test XML files to assert the validity of the XML as a model. Certain entities are tested by cross-checking data from the XML file with the newly created models to verify that each model is constructed correctly.

The TestDownload suite ensures that the export facility properly generates XML files. This is done using ElementTree to construct the XML and testing for the validity of child nodes. The ElementTree object is created with a root that matches the root from the supplied XML. Each subsequent function serves to build the XML by inserting data appropriately in relation to the root of the tree. The unit tests also validate the regression of a model to XML and back into a model.

Conclusion

The preceding report presents the design decisions and tools used to create a viable and useful website in an effort to provide an easily accessible platform where users can attain valuable information.

The Crisix website (**crisix.herokuapp.com**) offers a pleasant UI that displays an abundance of information and affords users the opportunity to help the disaster-stricken. Crisix provides import/export facilities and guarantees the integrity of the database backend by imposing user authentication and file validation. Also available is real-time unit test output to further assert correctness and validity of the website itself as well as the database backend.

Each page is rendered dynamically, meaning users are able to import new data and render content immediately. Site pages are displayed in a unified manner to maintain easy readability and a pleasant user experience, while also presenting an attractive way to navigate between related entities.

The Crisix Database implementation fulfills several critical roles by providing flexibility and allowing the data to be scaled. The overall experience provided to the end user is friendly and efficient, while successfully meeting the necessary technical requirements.