Gummy Pandas

CS373

Members:

Arturo Lemus

Russell Jahn

Honghui Choi

Young Seo

Hoang Pham

Zhen Yang

Introduction:

With countless tragedies and crises happening around the world today, staying informed about them can help mitigate its consequences and possibly prevent similar crises from happening in the future. To help people understand the impacts and consequences of the crises that are happening around the world, the Gummy Pandas' WCDB website was created. Its main goal is to hopefully educate the visitors on some of the current world crises and on the people and the organizations involved in them.

Possible visitors to the Gummy Pandas' WCDB website include fellow students or professors pursuing academic research, interested in certain crisis, person, or organization on the website and random visitors searching for something similar. With the given information, each visitor can inform themselves of the crises and even get involved to help with a crisis. The visitor can navigate through the website by clicking on either the different pictures from the homepage that will take her to a new page with information on the picture that she just clicked on, or by clicking on either the Crises, People, or Organization tab on the navigation bar that will filter all of the pictures to corresponding clicked type and from there she can click on the particular picture to get more information. Each picture has a caption of its name so that if the picture is ambiguous, visitors can still know what the picture is supposed to represent.

Design:

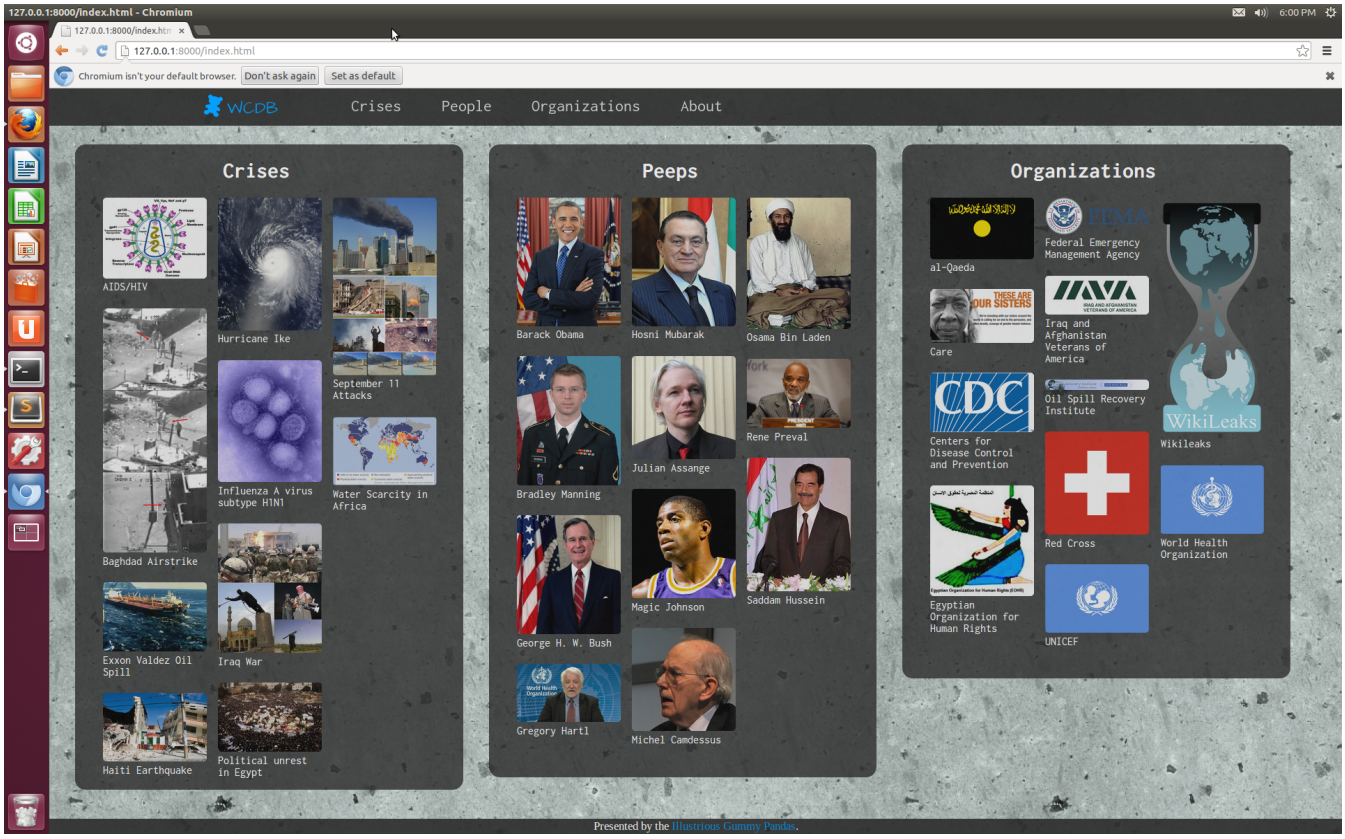
XML

The XML Schema only requires an ID and a name for every crisis, organization, and person. Other than that, everything else is optional. There is also a complex type called common that includes general information like citations, external links, images, videos, maps, feeds, and summary that you can add into the big 3 (crisis, organization, person). For crisis, you can have the IDs of people and organizations that are involved in it. Then you can add in date, time, and the locations of the crisis. Furthermore, there are human impact, economic impact, resources needed and ways to help field that you can fill to give the reader a deeper understanding of the crisis. For organization, the optional elements are crises that it is involved with and the people that are in it. Plus you can fill in in more information about the organization including its type, locations, history, and contact info. Finally, we have person type that have elements like crises and organizations that the person gave a hand to. Additionally, the person's job and location are there if you feel like you want to tell everyone what this person is up to. To make sure that each elements from the big 3 are unique, we provided 3 simple id types called CrisisIDType, PersonIDType, and OrgIDType. These 3 id types will require either ORG_, PER_, or CRI_ in front and then the unique ID of organization, people, and crisis at the end.

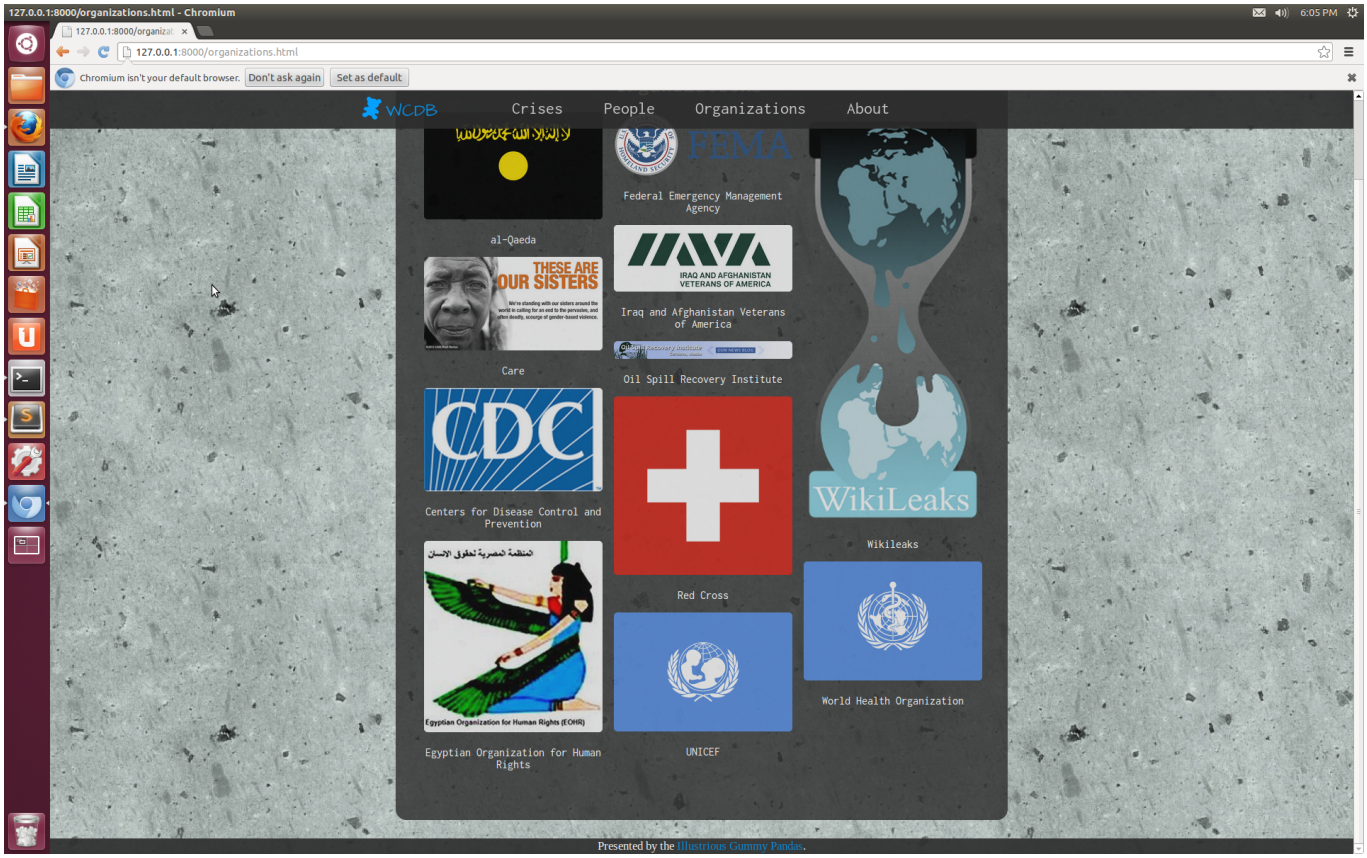
Django models

UML

UI - The design of the website was based largely on the ease of navigation. Visitors will not have to read through large paragraphs of text to get more information on the crises, people, and organization, but instead can simply click on given pictures and their names to pick and choose what subject to learn about. The homepage has all of the Gummy Pandas' crises, people, and organizations on it where visitors can click on their pictures to go to the corresponding page that contains more details of the clicked picture. When/if the database of the crises expands, the homepage will have a featured crisis, person, and organization that are randomly picked from the database. The Gummy Pandas' website uses pictures and names of the pictures for means of navigation. The homepage has all of the pictures and their names separated into columns according to their type (crises, people, or organization) but visitors are given the option to filter them by type by clicking on either the Crises, People, or Organizations tab on the navigation bar. One click on either one of these tabs will take the visitor to a new page containing on the pictures of the clicked tab. From there, the visitor can click on a picture that will bring them to a new page containing more information about the clicked picture. The pictures and the background columns have rounded edges for a more visually pleasing effect. This website is meant to be simple to use while looking organized and clean.


127.0.0.1:8000/index.html - Chromium
6:00 PM
127.0.0.1:8000/index.html
Chromium isn't your default browser.
Don't ask again
Set as default
WCDB
Crises
People
Organizations
About
Crises
AIDS/HIV
Hurricane Ike
September 11 Attacks
Influenza A virus subtype H1N1
Water Scarcity in Africa
Baghdad Airstrike
Iraq War
Exxon Valdez Oil Spill
Haiti Earthquake
Political unrest in Egypt
Peeps
Barack Obama
Hosni Mubarak
Osama Bin Laden
Bradley Manning
Julian Assange
Rene Preval
George H. W. Bush
Magic Johnson
Saddam Hussein
Gregory Hartl
Michel Camdessus
Organizations
al-Qaeda
Federal Emergency Management Agency
Care
Iraq and Afghanistan Veterans of America
Centers for Disease Control and Prevention
Oil Spill Recovery Institute
Wikileaks
Red Cross
World Health Organization
Egyptian Organization for Human Rights
UNICEF
Presented by the Illustrious Gummy Pandas.

Homepage



After clicking on Organization tab

Implementation:

To import the xml files, elementTreeWrapper.getTree() is called to return the elementtree object. Elements with tag Crisis are searched and follow the procedure:

1. Get ID and name by making a dictionary, then get all the items of the element then stored into that array. Finally, save the ID and name from that dictionary into new variables to access them

later.

2. Find elements with tag Organization and People. Since a Crisis can have many people and organizations, we use a while loop to get every people and organization that are involved in the crisis and then append them into a list.
3. Find elements with tag Kind, Date, and Time then save them into their own String variables.
4. Find elements with tag HumanImpact, EconomicImpact, Locations, ResourcesNeeded, WaysToHelp. Since there can be elements in the above tags that are surrounded by <li>, we make sure to get all of them by using a while loop and append them into their own arrays.

For People and Organization, the same four steps above are done but with different tags. In the schema, there is a complex element called Common whose each element is a list type, and each list type consists of one or more href, or embed, or text. Since the big three elements (Person, Crisis, and Organization) share the 'Common' element, a function called 'getCommonData' is created to extract the information that are needed for Common and call it at the end of the big three elements. Within the Common function, the same implementation as step four was done but with tags: Citations, ExternalLinks, Images, Videos, Maps, Feeds, Summary.

Now that all the information needed from the xml file is obtained, they need to be stored into our django models. To do that, an empty list called models is created. At the end of each of the big three elements, the data for Crisis or Person or Organization is stored into the models list. Finally, all the data stored in the models are saved on the Crisis, Person, and Organization objects by calling 'save()'.

To export our Django models to a xml file, a root called “WorldCrises” is created by using ET.Element(“WorldCrises”). Then obtain all the objects in the Crisis Table. For each of the object becomes the child of “WorldCrises” thanks to the ET.SubElement command, which creates Crisis class. Then add other information under Crisis as children of Crisis. After that, the same procedure is done for Person and Organization. Next, indent the file to make it look pretty and make it into a tree by using ET.ElementTree.

Testing:

At the end of the project, the unit tests are created in order to test the implementation of xmlParser.py. The parser is the crucial part of the project. It extracts the data from the XML and store them into database. The functions involved in accomplishing this job in our xmlParser file are 'getTextAndAttributes', 'getCommonData', 'elementTreeToModels', 'modelsToDjango'.

For 'getTextAndAttributes' function, a brief sample input of XML is created and casted into a tree and passed to 'getTextAndAttributes' function as an argument, which returns a dictionary. Assertions functions are then called to check if the contents returned by the keys match what we expected.

For 'testgetCommonData,' again a short sample XML was built and parsed into a tree, which returns a list of dictionaries. The expected output is compared to the actual output for each content of the keys and the array itself that contains dictionaries.

For 'indent' function, we used a sample XML code, and called the function on that. Then, the output of the function is compared to the expected output.

For 'elementTreeToModels'. An element tree created from the sample XML is passed to this function as an argument. The output is a list of models. The output of the function is compared with the expected output.

For 'isNotDuplicate', a dictionary is created and pass to this function along with a value. If the the value is already in the dictionary, the function would return false, otherwise return true.

To test the data is being inserted correctly into the MySQL database and the respective tables, the ‘elementTreeToModels’ function initialilzes the attributes in ‘models.py’. Instances of the Person, Crisis, and Organization classes are created and then passed into ‘elementTreeToModels’ function and assertion statements for each class and their attributes are created and then compared to expected values to evaluate correctness.

Model:

There are 5 classes in the Model.py: Crisis, Person, Organization, Lists, and Common. Django generates tables in database according to the classes in the Model.py. The command is 'python manage.py syncdb'.

After entering the command into terminal, at least five tables are generated. Each class has one table labeled by the class name. The tables for class are wcdb_crisis, wcdb_person, wcdb_organization, wcdb_list, and wcdb_common. The columns are the variables in the class. Each row is a set of data.

For class crisis, class person, class organization, the variable 'slug' in SlugField is for the url. The others are the attributes and elements for them according to the XML scheme. The attribute 'ID' is required and unique, so set 'ID' to be the prime key. For the elements with type "xsd:token", “xsd:date”, and "xsd:time", use charField and set null=True if minOccurs="0".

In CrisisType, the elements “Locations”, "HumanImpact", "EconomicImpact", "ResourcesNeeded", "WaysToHelp" are ListType, which means they have multiple Lists. To import into database easily, set the field of these variables to be TextField, then store a string of all the Lists separated by commas into database. When exporting from database, this string could be split into array of List. Use the same way to import/export for the elements "History" and "ContactInfo" in OrgType.

In the XML, the elements with type “CrisisWithID”, “PersonWithID”, and “OrgWithID” show the relationships between Crisis, Person, and Organization. In model, the field for these variables are TextField, which could make the relationships clear and easy to handle.

For example:

```
<Crisis ID="CRI_BAGAIR" Name="Baghdad Airstrike">
        <People>
                <Person ID="PER_BRAMAN" />
                <Person ID="PER_JULASS" />
        </People>
        <Organizations>
                <Org ID="ORG_WIKLKS" />
        </Organizations>
</Crisis>
```

In this XML, Person ID and Org ID are type “PersonWithID”, and “OrgWithID”. They show the people and organizations involved into crisis "Baghdad Airstrike". xmlParser.py sets people equal to the string “['PER_BRAMAN' , 'PER_JULASS']” and store into database. Then when exporting from database, xmlParser.py splits the string into array ['PER_BRAMAN' , 'PER_JULASS']. In this way, it is really easy to manage the relationship between Class Crisis and Class Person.

The Class Common and List are for the CommonType. Since each crisis, person, or organization could have 0 or 1 common, set a ForeignKey relate to Common in Class Crisis, Person, and Organization. Also, since common is optional, set null=True on the ForeignKey. Then, in database, the tables wcdb_crisis, wcdb_person, wcdb_organization have common_id. The elements in CommonType are ListType, which means each common can have multiple citations, externalLinks,

images, videos, maps, or feeds. Therefore, the variables in the class Common are ManyToManyField and associate with Class List. ManyToManyField would generate extra tables to hold the mapping between Common and List, like the tables wcdb_common_citations and wcdb_common_externalLinks.

The Class List is for the ListType variables in CommonType. There are four variables in class List: href, embed, text, and content.

Here is an example:

The List and Common class:

```
class List(models.Model):
    href=models.TextField(null=True)
    embed=models.TextField(null=True)
    text=models.TextField(null=True)
    content=models.TextField(null=True)

class Common(models.Model) :
    citations = models.ManyToManyField(List, related_name ='Citations+', null=True)
    externalLinks = models.ManyToManyField(List, related_name ='ExternalLinks+', null=True)
    images = models.ManyToManyField(List, related_name ='Images+', null=True)
    videos = models.ManyToManyField(List, related_name ='Videos+', null=True)
```

maps = models.ManyToManyField(List, related_name ='Maps+', null=True)

feeds = models.ManyToManyField(List, related_name ='Feeds+', null=True)

summary = models.TextField(null=True)

Create a couple of externalLinks List (the same for citations, images ...):

li1=List(href="https://en.wikipedia.org/wiki/July_12,_2007_Baghdad_airstrike")

li1.save()

li2=List(embed="https://upload.wikimedia.org/wikipedia/commons/7/7c/July_12%2C_2007_Baghdad_airstrike_targets_%281%29.png")

li2.save();

Create a Common:

common=Common()

common.save()

Associate the common with List:

common. ExternalLinks.add(li1)

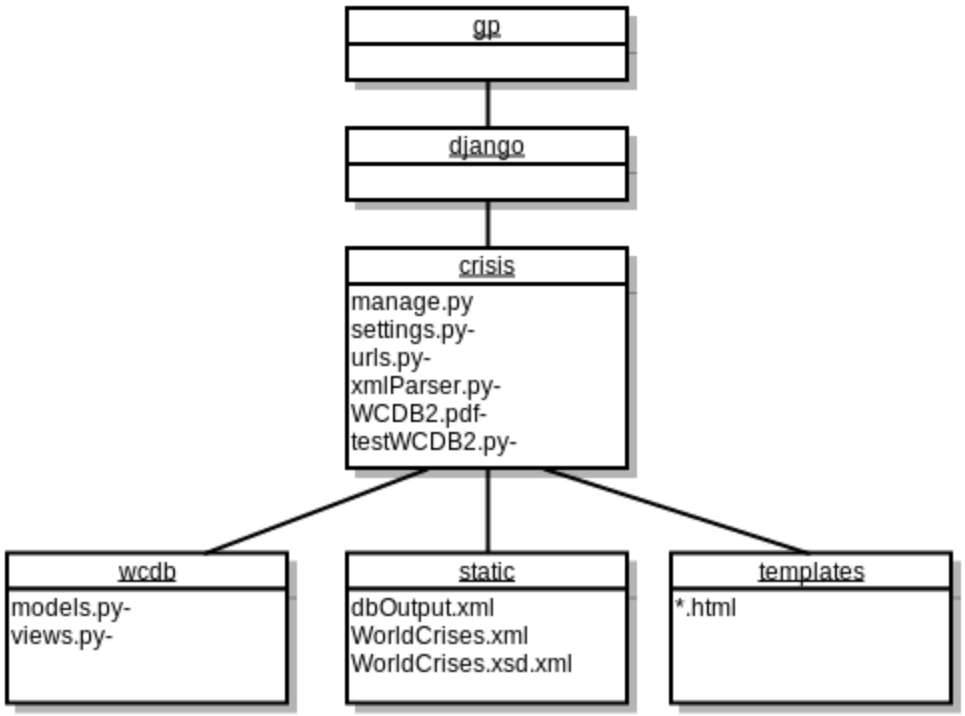
common. ExternalLinks.add(li2)

Access to the ExternalLinks List objects that related to common:

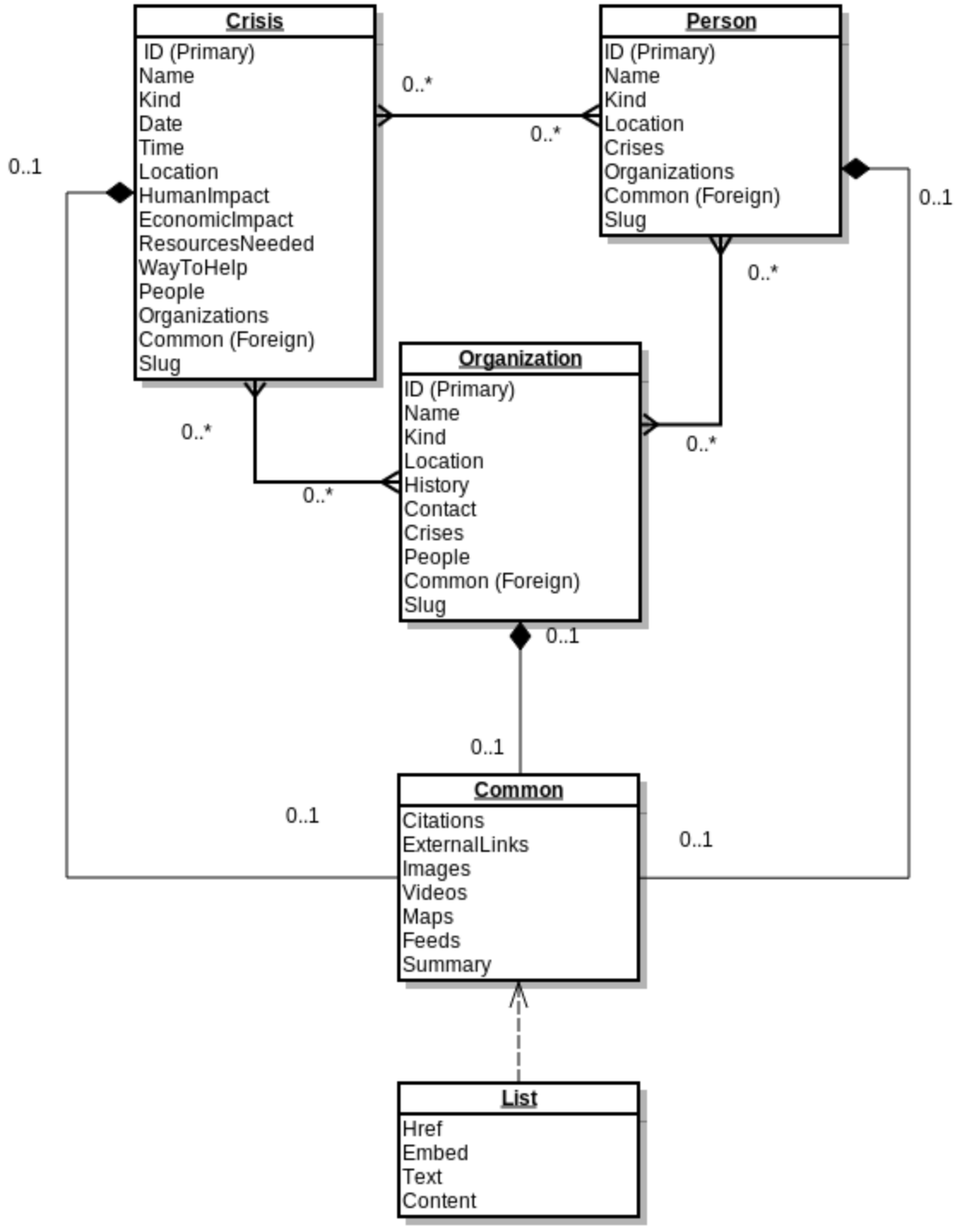
common.externalLinks.all()

Figures

Directory Structure of Django Project 'crisis' , application 'wcdb'



Models.py UML Displaying Relationships and Multiplicity


Crisis
ID (Primary)
Name
Kind
Date
Time
Location
HumanImpact
EconomicImpact
ResourcesNeeded
WayToHelp
People
Organizations
Common (Foreign)
Slug
Person
ID (Primary)
Name
Kind
Location
Crises
Organizations
Common (Foreign)
Slug
Organization
ID (Primary)
Name
Kind
Location
History
Contact
Crises
People
Common (Foreign)
Slug
Common
Citations
ExternalLinks
Images
Videos
Maps
Feeds
Summary
List
Href
Embed
Text
Content
0..*
0..*
0..1
0..*
0..*
0..*
0..*
0..1
0..1
0..1
0..1
0..1