# Developing an annotations system for the collaborative web application MAGI (Mutation Annotation and Genome Interpretation)

John Shen
Advisor: Ben Raphael
Masters' Report

1/6/2015

## Section 1: Introduction

As genomic sequencing data from cancer samples becomes more available due to efforts such as The Cancer Genome Atlas (TCGA), the need for visualization and collaboration tools to make sense of genetic data also increases. MAGI (for Mutation Annotation and Genome Interpretation) was created for the purpose of visualizing and sharing information on tumor sequences[1]. MAGI provides preloaded, public cancer datasets from TCGA but also allows users to load their own datasets. Another goal of MAGI is to allow users to annotate genetic entities with literature references that may be helpful in sharing information. The final goal, on the developer's side, is to make MAGI easier to maintain, extend, and deploy.

While MAGI has met the first goal of combining and visualizing genomic datasets, it has been difficult to implement references and annotations in a maintainable way. In this paper I describe my approach and accomplished work to provide flexible annotation capabilities within MAGI, including all basic creation, modification, viewing, and deletion operations, as well as user provenance, and a structured interface. I outline a relational data model for mutations and protein interactions that allows for multiple, extensible mutation types. In addition, I implemented the pages within Django, a new web framework. Finally, I also discuss related work on the operations portions of MAGI towards faster deployment and testing.

## Section 2: Development Background and Approach

### *1. Previous Work*

In order to understand our choice of frameworks we describe MAGI's particular application requirements. MAGI has two primary components; an interactive visualization application for

[1] Leiserson, M., Gramazio, C.C., Hu, J., Wu, H., Laidlaw, D.H., & Raphael, B.J. *Nature Methods* **12***,* 483–484 (2015)

large, mostly static datasets and a collaborative annotation platform for tagging and cross-referencing genes, mutations, and protein interactions.  In this paper, we focus on reference and annotations that refer to genetic aberrations and protein interactions.

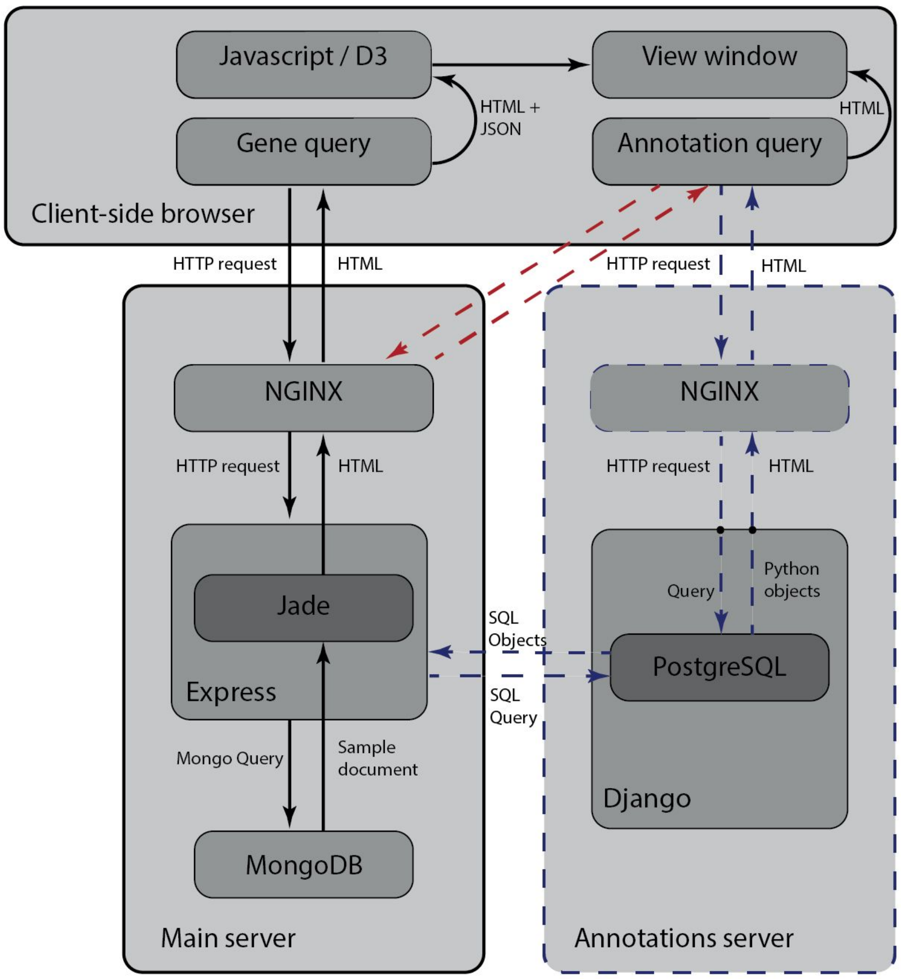


Figure 1: Software stack diagram for previous (solid black) and proposed (dashed blue) MAGI.  Red dashed indicates connections that will be deleted.  The proposed MAGI architecture uses two servers to handle samples and annotations separately.  Annotation queries initiated from the user  occur through the annotations server; from the MAGI web page they are SQL queries initiated by Express.js.

The previous version of MAGI was built with an ad-hoc set of web frameworks that were readily available to developers.  These included a visualization engine on the client side built in D3, a back-end for storing tumor samples built in MongoDB, and a server written in Node.js (Figure 1). While user queries that retrieved and created tumor samples containing a given gene were well served, the frameworks were poorly suited user queries that acted upon annotations for reasons we describe below.

## *2. New Functional Requirements for Annotations*

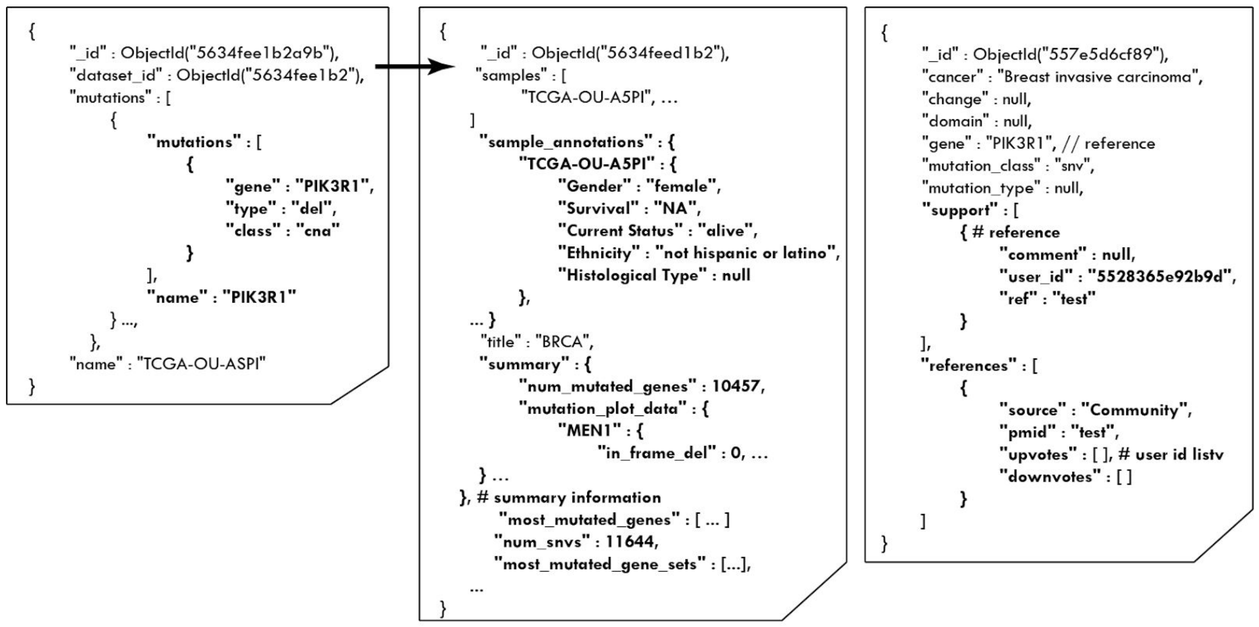
One goal was to extend MAGI's abilities to researchers for managing annotations.
We now allow users to individually add and manage references and related, curated information to as wide a variety of genetic mutations and protein interactions as possible. In detail, users should be able to

- add a genetic aberration (of varying detail) and attach a literature reference that discussed that aberration
- add an annotation to any reference to describe the reference's content (w.r.t to a particular mutation)
- view both curated and user-defined references (and attached annotations) that mention a gene and mutation
- approve the consensus annotation for a particular reference
- specify a particular protein-protein (PPI) interaction and attach a literature reference to it
- upvote or downvote a reference attached to a PPI
- delete their own annotations and references but not those of others.

In addition, developers should be able to easily modify and attach new schemas to the existing database (new types of mutations, new fields for a reference or annotation), and guarantee the integrity and consistency of the extant database. This will in turn enhance the maintainability and extensibility of the program.

## *3. Choosing a database designed for read-write transactions*

In previous versions, MAGI used MongoDB, a document-based database, as its backend, because static dataset samples lend themselves more to hashing views in a document-like format. However, there are a few issues with this hashed representation (Figure 1). For instance, not all the data for a given sample is stored within the sample; sample annotations are kept within the dataset. Also, there is no index that relates annotations for a given gene or mutation to samples that contain that mutation. This relationship can be easily captured by a gene relation with an index in an SQL framework.



```
{
    "_id" : ObjectId("5634fee1b2a9b"),
    "dataset_id" : ObjectId("5634fee1b2"),
    "mutations" : [
        {
            "mutations" : [
                {
                    "gene" : "PIK3R1",
                    "type" : "del",
                    "class" : "cna"
                }
            ],
            "name" : "PIK3R1"
        } ...,
    },
    "name" : "TCGA-OU-ASPI"
}
```

```
{
    "_id" : ObjectId("5634feed1b2"),
   "samples" : [
        "TCGA-OU-A5PI", ...
  ]
   "sample_annotations" : {
        "TCGA-OU-A5PI" : {
            "Gender" : "female",
            "Survival" : "NA",
            "Current Status" : "alive",
            "Ethnicity" : "not hispanic or latino",
            "Histological Type" : null
        },
  ... }
   "title" : "BRCA",
   "summary" : {
        "num_mutated_genes" : 10457,
        "mutation_plot_data" : {
            "MEN1" : {
                "in_frame_del" : 0, ...
   } ...
 }, # summary information
      "most_mutated_genes" : [ ... ]
     "num_snvs" : 11644,
     "most_mutated_gene_sets" : [...],
  ...
}
```

```
{
    "_id" : ObjectId("557e5d6cf89"),
    "cancer" : "Breast invasive carcinoma",
    "change" : null,
    "domain" : null,
    "gene" : "PIK3R1", // reference
    "mutation_class" : "snv",
    "mutation_type" : null,
    "support" : [
        { # reference
            "comment" : null,
            "user_id" : "5528365e92b9d",
            "ref" : "test"
        }
    ],
    "references" : [
        {
            "source" : "Community",
            "pmid" : "test",
            "upvotes" : [ ], # user id listv
            "downvotes" : [ ]
        }
    ]
}
```

Figure 2: MongoDB examples of an A) sample, B) dataset, and C) an annotation. One advantage is that summary data for a dataset is hashed within the dataset document. However, we also note that sample data are distributed between the sample document and dataset document (for sample annotations). Also, the data for an annotation for a mutation and reference are completely unlinked to other documents; therefore lookups for related annotations and mutations are less efficient.

We currently use both MongoDB for samples and datasets and PostgreSQL for annotations. For annotations, we chose a relational data model over a document database model for two reasons. First, write operations on annotations are more clearly expressed and require less maintenance to establish consistency. In a document-based database such as Mongo above, annotations would either need to be manually given an index, or otherwise replicated and rewritten for each write and update. In database language, PostgreSQL supports the transactions of adding, editing and removing annotations more simply than MongoDB.

Second, relational data models more naturally represent the relational nature of adding annotations. In the long-term, implementing samples and datasets in PostgreSQL will provide greater benefits in development and space savings that offset the performance benefits in MongoDB. The new, proposed data model is described in section 3.1.

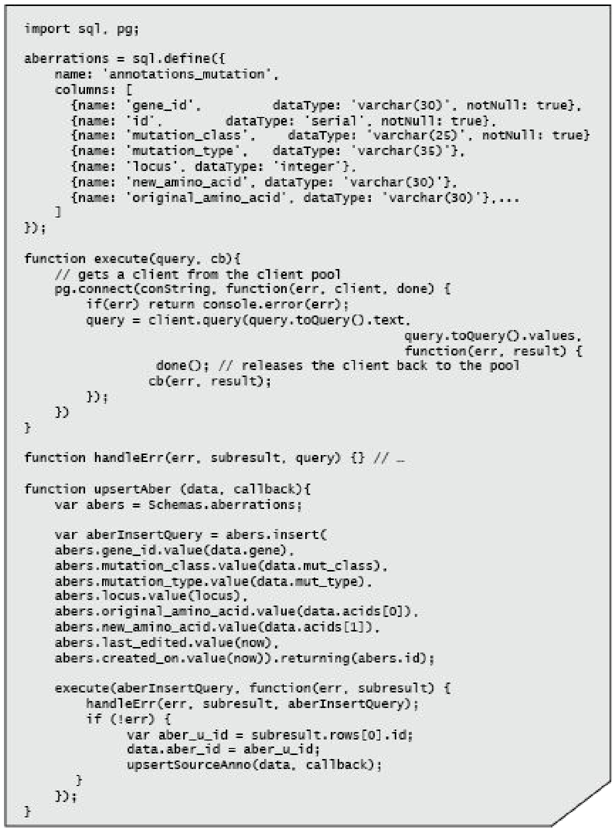
## *3. Choosing a web framework with modern features*

In addition to a database, we also chose a web framework, Django, built atop Python, that implements many of the commonplace web features needed in MAGI (Figure 1). The previous iteration of MAGI uses the Express.js framework built on top of Node.js, in Javascript. On one hand, Django and Node.js both follow the Model View Controller framework (MVC). In these frameworks, the model describes the physical organization of data and resources within the database, and the view describes the user's interface to the data.  The controller, which lies between these two layers, determines the logic of the application; which resources are retrieved, modified or displayed for a given page.  As a note, Django refers to its framework as a Model Template View (MTV) framework, where templates are the views of MVC and views are the controllers.

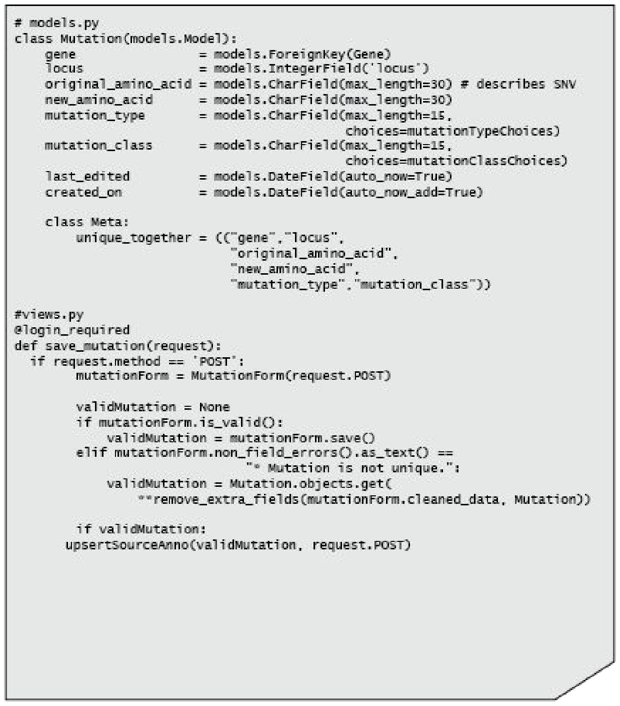
On the other hand, Express.js and Django differ most in the availability of built-in web features. Express.js supplies a rather flexible framework, without committing to specific databases or templating systems.  While this admits a powerful web application that can handle many persistent users, the MAGI application most requires static retrieval and viewing of resources, without persistent connections.  Therefore, Express.js requires several additional layers of specification, such as the need for maintaining client pools for databases, specifying custom SQL commands (Figure 3A), or custom markup-language templating via Jade (Figure 3B), all with relatively little benefit.

In contrast, Django provides several features by default, which we illustrate with MAGI in detail below.  I illustrate the model and its advantages in detail in section 3.2.  Most of these features are provided within the MTV framework by subclassing Django's classes and using Django scripts.  Specifically, Django has its own object-relational model (ORM) that wraps database objects within its own Python objects, as well as a templating language that sits over regular HTML.  Because of Django's features, several otherwise difficult features are much simpler to implement; these are further described in section 4.  Overall, Django embodies the design tradeoff of **convention over configuration**.

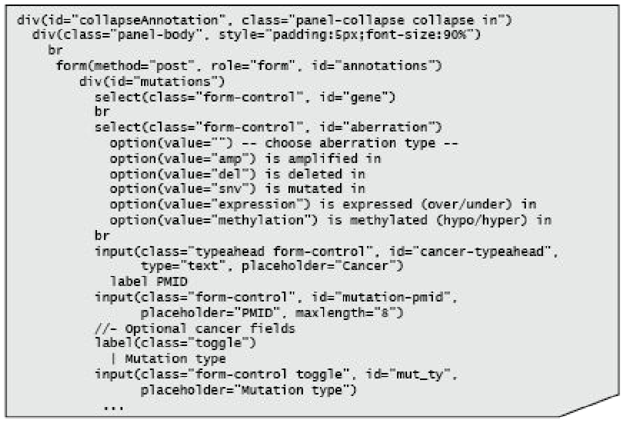
One additional benefit of Django is a built-in testing framework for models and controllers (views in Django).  While we do not yet implement tests at this level in Django, testing is important for maintaining the correctness of the application.  I have some demo tests within the original MAGI repository written in Watir over Ruby: these are described within the original MAGI repository on Github.



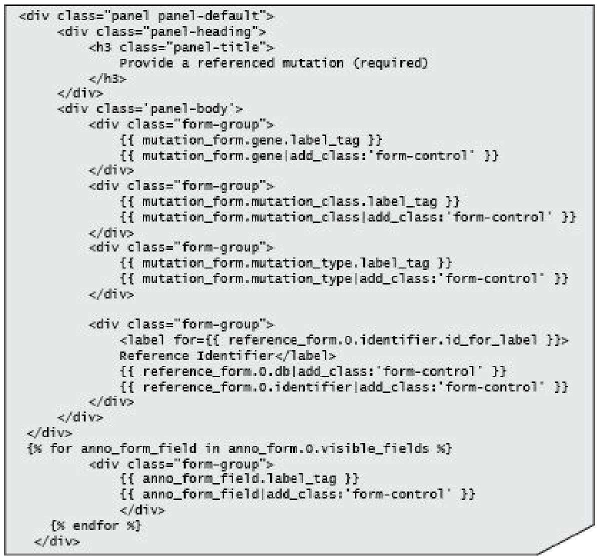
```
import sql, pg;

aberrations = sql.define({
    name: 'annotations_mutation',
    columns: [
      {name: 'gene_id',         dataType: 'varchar(30)', notNull: true},
      {name: 'id',        dataType: 'serial', notNull: true},
      {name: 'mutation_class',    dataType: 'varchar(25)', notNull: true}
      {name: 'mutation_type',   dataType: 'varchar(35)'},
      {name: 'locus', dataType: 'integer'},
      {name: 'new_amino_acid', dataType: 'varchar(30)'},
      {name: 'original_amino_acid', dataType: 'varchar(30)'},...
    ]
});

function execute(query, cb){
    // gets a client from the client pool
    pg.connect(conString, function(err, client, done) {
        if(err) return console.error(err);
        query = client.query(query.toQuery().text,
                                                  query.toQuery().values,
                                                  function(err, result) {
                 done(); // releases the client back to the pool
                cb(err, result);
        });
    })
}

function handleErr(err, subresult, query) {} // …

function upsertAber (data, callback){
    var abers = Schemas.aberrations;

    var aberInsertQuery = abers.insert(
    abers.gene_id.value(data.gene),
    abers.mutation_class.value(data.mut_class),
    abers.mutation_type.value(data.mut_type),
    abers.locus.value(locus),
    abers.original_amino_acid.value(data.acids[0]),
    abers.new_amino_acid.value(data.acids[1]),
    abers.last_edited.value(now),
    abers.created_on.value(now)).returning(abers.id);

    execute(aberInsertQuery, function(err, subresult) {
        handleErr(err, subresult, aberInsertQuery);
        if (!err) {
             var aber_u_id = subresult.rows[0].id;
             data.aber_id = aber_u_id;
             upsertSourceAnno(data, callback);
       }
    });
}
```



```
# models.py
class Mutation(models.Model):
    gene                = models.ForeignKey(Gene)
    locus               = models.IntegerField('locus')
    original_amino_acid = models.CharField(max_length=30) # describes SNV
    new_amino_acid      = models.CharField(max_length=30)
    mutation_type       = models.CharField(max_length=15,
                                           choices=mutationTypeChoices)
    mutation_class      = models.CharField(max_length=15,
                                           choices=mutationClassChoices)
    last_edited         = models.DateField(auto_now=True)
    created_on          = models.DateField(auto_now_add=True)

    class Meta:
        unique_together = (("gene","locus",
                            "original_amino_acid",
                            "new_amino_acid",
                            "mutation_type","mutation_class"))

#views.py
@login_required
def save_mutation(request):
  if request.method == 'POST':
        mutationForm = MutationForm(request.POST)

        validMutation = None
        if mutationForm.is_valid():
            validMutation = mutationForm.save()
        elif mutationForm.non_field_errors().as_text() ==
                              "* Mutation is not unique.":
            validMutation = Mutation.objects.get(
                **remove_extra_fields(mutationForm.cleaned_data, Mutation))

        if validMutation:
      upsertSourceAnno(validMutation, request.POST)
```



```
div(id="collapseAnnotation", class="panel-collapse collapse in")
  div(class="panel-body", style="padding:5px;font-size:90%")
    br
     form(method="post", role="form", id="annotations")
        div(id="mutations")
          select(class="form-control", id="gene")
          br
          select(class="form-control", id="aberration")
            option(value="") -- choose aberration type --
            option(value="amp") is amplified in
            option(value="del") is deleted in
            option(value="snv") is mutated in
            option(value="expression") is expressed (over/under) in
            option(value="methylation") is methylated (hypo/hyper) in
          br
          input(class="typeahead form-control", id="cancer-typeahead",
                type="text", placeholder="Cancer")
            label PMID
          input(class="form-control", id="mutation-pmid",
                placeholder="PMID", maxlength="8")
          //- Optional cancer fields
          label(class="toggle")
            | Mutation type
          input(class="form-control toggle", id="mut_ty",
                placeholder="Mutation type")
            ...
```



```
<div class="panel panel-default">
     <div class="panel-heading">
         <h3 class="panel-title">
             Provide a referenced mutation (required)
         </h3>
     </div>
     <div class='panel-body'>
         <div class="form-group">
             {{ mutation_form.gene.label_tag }}
             {{ mutation_form.gene|add_class:'form-control' }}
         </div>
         <div class="form-group">
             {{ mutation_form.mutation_class.label_tag }}
             {{ mutation_form.mutation_class|add_class:'form-control' }}
         </div>
         <div class="form-group">
             {{ mutation_form.mutation_type.label_tag }}
             {{ mutation_form.mutation_type|add_class:'form-control' }}
         </div>

         <div class="form-group">
             <label for={{ reference_form.0.identifier.id_for_label }}>
             Reference Identifier</label>
             {{ reference_form.0.db|add_class:'form-control' }}
             {{ reference_form.0.identifier|add_class:'form-control' }}
         </div>
     </div>
 </div>
 {% for anno_form_field in anno_form.0.visible_fields %}
         <div class="form-group">
             {{ anno_form_field.label_tag }}
             {{ anno_form_field|add_class:'form-control' }}
             </div>
    {% endfor %}
  </div>
```

Figure 3: Express.js (left) vs Django code (right) for mutation annotations (A, top) and form templating (B, bottom). Note that Express.js requires more boilerplate code for database transactions (at top) and also explicit declarations of choices and defaults for form fields.

## Section 3: Design Process and Implementation Details:

### *1. Designing an SQL Data Model for Maximal Reuse and Consistency*

Our model must first satisfy the functional requirements, so each entity mentioned in those requirements should have a corresponding field or entity in the database; mutations, gene aberrations, protein interactions, references, annotations and votes) . However, I also designed our data model to remove redundancies and duplications; this solves the problem of creating and maintaining multiple instances of the same data across multiple documents (in MongoDB) or relations (as in SQL).

The practice of minimizing redundancy in a database is known as **normalization,** and it corresponds to optimizing disk and memory space for an algorithm. For instance, if many tumor samples contained the same mutation, then the data for that mutation should be stored in a single relation, rather than repeated in multiple samples. In MongoDB, the same mutation was duplicated across multiple samples and references (see Figure 1), but in order to find related samples, the system would have to lookup all documents by a hash of the mutation, which may be unindexed and require slower lookups. Therefore, data can be reused A relational database design such as SQL allows us to quickly cross-reference references to samples, and to update a single source if we decide to add new fields to the mutation, such as affected pathways.

However, beyond normalization, we also desire proper encapsulation of entities. This allows us to maintain database integrity without joining relations unnecessarily or using unnatural NULL entries. For instance, if a user's email must be a valid email, and annotations of a reference must specify the user that annotated them, then a user ID field belongs in the annotation, but not the user email; the user email belongs in a separate relation.

The fact that reference annotations must specify a proper user is a dependency, and that a user email must be proper is a constraint. Dependencies require placing only the necessary fields in a given relation. Broadly, the practice of encapsulating entities to keep dependencies within a single relation is known as **dependency preservation** in database design, and it corresponds to maintaining correctness (integrity) within the database with minimal time complexity. Practically, this means that relations ought to refer to one another via a unique primary id.

With those goals in mind, we can identify entities that are multiply referred by several separate MAGI domains:

- users,
- cancers,
- mutations,
- genes,
- and samples.

These entities are given a unique relation, and are given a natural primary key by their name. We can see how these relations connect to other entities in (Figure 3). Normalization allows for data to be non-duplicated and indexed more quickly; for example, mutations can naturally indicate all samples that contain them, and all references that refer them.

In addition, we can identify entities that should be given separate relations:

- multiple annotations can be given to a single reference (and by different users), so references and annotations are separated
- multiple votes can be given to an interaction reference (and by different users), so interaction references and votes are separated
- all mutations can be included in samples and referenced, but different mutation types have different attached data; thus, each mutation class should have a separate relation for annotations specific to that class. For example, single nucleotide variants should have a separate relation from copy number annotations. Incidentally, this is an area where document database models such as Mongo perform more naturally.

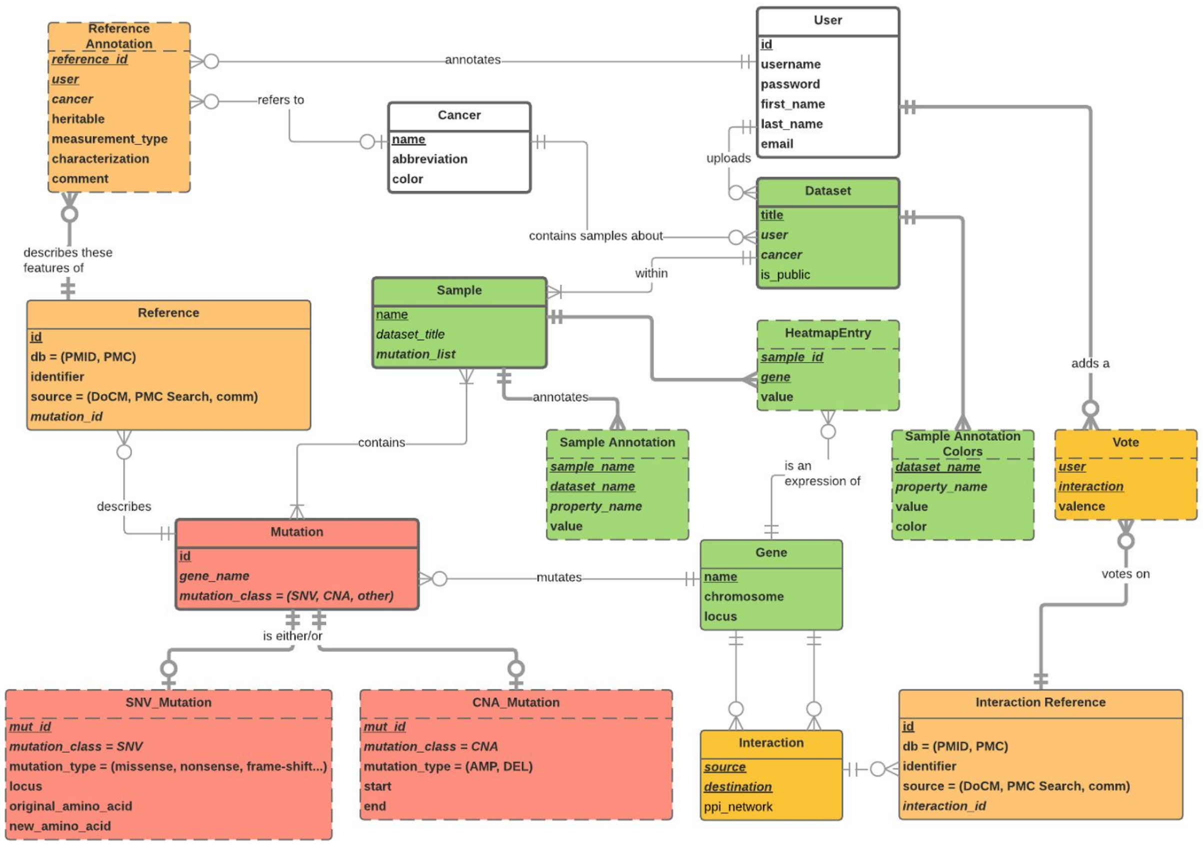


Figure 4: A normalized schema for the MAGI application as a crows' foot E-R diagram. Underlined fields are primary keys: italicized fields are fields that refer to other tables. Entities with a dashed boundary are dependent on their

parent entity (thick line). For example, a sample is implemented as a name, dataset, and list of mutations and a linked group of annotations, with dependent sample annotations and heatmap values.
Colors indicate related entities. Foundational entities that are referenced multiple times are the user, gene, cancer, mutation, and reference. A sample is implemented as a list of mutations and a linked group of annotations. Normalization allows for data to be non-duplicated and indexed more quickly.

One design choice of note is the duplication of a reference table for both mutation and protein interactions. This illustrates a design tradeoff between normalization and dependency preservation. According to normalization, we would create a single relation for any reference, including references that attach to both mutations and protein interactions, and each reference would be included once, even if it referenced multiple mutations and protein interactions.

However, if we recall our functional requirements, we want to be able to annotate references attached to a mutation, and vote on references attached to a protein interaction. Therefore, if we create one reference relation, we would have two difficulties:

1. In order to verify whether an annotation was correctly applied to a mutation reference (and not a protein interaction reference), we would at least have to check the reference type within the reference relation, which requires an extra join, thus violating dependency preservation.
2. If we want to annotate two mutations within the same reference, we must distinguish which mutation we are annotating, as well as which reference we annotate. This further violates dependency preservation, because we need to make sure the reference and mutation are associated in the reference relation. This illustrates that, in the one-reference schema, an annotation actually describes the relationship between a mutation and reference.

The second point demonstrates that an annotation is actually a tertiary relationship between a mutation, reference, and the facts of an annotation. In the one-reference schema, we would require an in-between relation that pairs annotations and mutations. An annotation, in turn, would contain a foreign key to. Therefore, we choose an alternate, denormalized approach:

1. we use two relations for references, one for mutations and one for protein interactions. This mitigates the first difficulty.
2. a reference relation encapsulates both the identifying facts of a reference, and its relationship to a particular mutation (thus, references contain the mutation they refer to). Therefore, annotations can contain a foreign key to the reference itself, with the tradeoff that we allow multiple references within the database to exist to refer the same logical reference.

One other concern with database models is how to extract entities from existing resources and transform them data to be loaded into the database (ETL). In order to preload the annotations with public cancer sample datasets, I wrote a set of Python scripts to convert the existing MAGI format .tsv files into a serialized JSON format. These JSON files are called fixtures and are

Django's way of providing initial data to the database. Instructions are given in annotations/fixtures/README.md for loading these fixtures.

## *2. Providing Annotation Manageability with Django*

In this section I describe the pages served by the annotations side of MAGI. We can categorize these pages according to their operations they perform on the database: those that create, retrieve, update, and delete entries from the database, known as CRUD operations. Django wraps these database operations as API calls on models, so that development is simplified. In addition, Django also provides a simple API for registering user logins.

Authentication:

From the main MAGI page, we currently allow server-side Javascript to access the underlying database for read requests to annotations MAGI for speed and convenience (Figure 2). However, for all write operations on the database, we provide pages through the annotations MAGI site, because we want to track all reference and annotation additions to with a particular user, which requires authentication. For annotations, it is easier to provide a single login on the annotations page rather than require that a login from the main MAGI page be transferred to the annotation server. Because the main MAGI page and annotation page are currently on different domains, it is difficult to provide a secure single-sign on system. Therefore, we currently require users to sign on to the annotations MAGI page, and defer sign on requests from main MAGI to annotations MAGI.

Retrievals:

I wrote pages to retrieve annotations for a particular mutation reference or for a gene. In the latter case, annotations are presented in table form by summarizing across mutations and references, and users can delete an annotation from that page if they are the user that provided it.

Similarly, I wrote a page to retrieve protein interactions through a list of genes, also in table form. The user has the option to upvote and downvote references attached to a protein interaction through the tabular page.

Input forms (creation and updates):

Several pages add or update content to the annotations database, using web forms defined in the Django system. Beyond only saving data, we also would like input validation and user tracking. These add or update pages should have the following properties:

- If a page reports success, it should save data to the underlying page and redirect either to a page that displays the information submitted, or a new page to create another item.
- A page should require a current user login; if an input page is accessed without a login, then the user should be allowed to authenticate, and then redirect to the requested page.
- An input page should be able to be pre-filled with query parameters from the web address.
- If a user tries to input an object that already exists, the page should not fail but refer to the existing entity.
- If a page reports failure while trying to save data, then no objects should be saved to the database.
- If a page reports failure while trying to save data because of user input error, then the page should indicate how the user can fix the error (for example, an input out or range).

Individual annotations for a mutation reference can be created as the same page that displays the detailed annotations for a mutation reference via a form. In addition, if the reference the user wants to annotate does not exist in the database, then users should also be able to add a new reference via a compound form. Users can also add a reference to a protein interaction using a form.

Also, users should be able to add simple objects to the database that do not require any input fields. Therefore, I wrote routes to save annotations based on the majority annotation for a given mutation reference, and also to add upvotes and downvotes to interaction references.

### *3. Example Django code for database operation*

Figures 4A-D show standard Django code for adding an interaction and its reference. The components of the page follow the MTV pattern, which we describe below.

<u>Models: Database abstraction</u>

The models for an interaction are described in the *models.py* file (Figure 5A). These models files are the authoritative source for an object relational model (ORM), which abstracts the database schema as a set of Python objects. We can declare types for fields, choices within those fields, and also express foreign keys and many-to-many relationships to other models, and create unique constraints on the model. This allows the code in the view to create and retrieve objects within the database without having to write raw SQL.

Django also provides a built in user relation, which is referenced in the interaction reference. References and annotations are tracked with user input, but biological entities (genetic aberrations and protein interactions) are not, for simplicity.

```
# models.py

class Gene(models.Model):
  name = models.CharField(max_length=30, primary_key = True)
  chromosome = models.CharField(max_length=2, null = True)


# protein-protein interactions
class Interaction(models.Model):
  source = models.ForeignKey(Gene, related_name='source')
  target = models.ForeignKey(Gene, related_name='target')
  input_source = models.CharField(max_length=25)

  class Meta:
    unique_together = (("source", "target", "input_source"))


dbChoices = (('PMID', 'PubMed'), ('PMC', 'PubMed Central'))
class InteractionReference(models.Model):
  identifier = models.CharField(max_length=40)
  db = models.CharField(max_length=30, choices=dbChoices)
  interaction = models.ForeignKey(Interaction)
  user = models.ForeignKey(User, null = True)

  class Meta:
    unique_together = (("identifier", "interaction"))
```

Figure 5A: Django model code.  Unique constraints are specified by the unique_together attribute.

Django defines fields in a database-independent format, so all standard flavors of SQL such as SQLite, Oracle SQL, and PostgreSQL.  Django provides utility commands to help maintain a consistent database by providing migrations when the database schema changes.

## Views: Form management

The views (controllers in MVC) in Django are further divided into two portions; those that describe the logic of a web form in *forms.py* (Figure 5B), and those that describe interactions with the database in *views.py* (Figure 5C).

The web form is responsible for presenting HTML to the user that contains the proper form and form fields, as well as validating and converting the returned form data back to an object (which is shown in the clean method).   For example, Django provides character fields with a select widget to restrict the possible values of the field to the choices in the set provided.  I can also control the rendering for fields; for instance, we choose the input_source field to be hidden.

```
# forms.py
```

```python
class InteractionForm(ModelForm):
    reference_identifier = forms.CharField(max_length=40)
    db = forms.CharField(max_length=20,
                         widget = forms.Select(choices=dbChoices))
    source = GeneField()
    target = GeneField()

    class Meta:
        model = Interaction
        fields = ['source', 'target', 'input_source']
        widgets = {'input_source': forms.HiddenInput()}

        error_messages = {
            NON_FIELD_ERRORS: {
                'unique_together': '%(model_name)s is not unique.'
            }
        }

class GeneWidget(forms.TextInput):
    def __init__(self, *args, **kwargs):
        if 'attrs' not in kwargs:
            kwargs['attrs'] = {'class': 'gene-typeahead'}
        elif 'class' not in kwargs['attrs']:
            kwargs['attrs']['class'] = 'gene-typeahead'
        else:
            kwargs['attrs']['class'] += ' gene-typeahead'

        super(forms.TextInput, self).__init__(*args, **kwargs)

    class Media:
        js = ('components/d3/d3.min.js',
              'components/typeahead.js/dist/typeahead.bundle.min.js',
               'components/handlebars/handlebars.min.js',
              'js/gene-typeahead.js')

def validate_gene(val):
    if Gene.objects.filter(name=val).count() > 0:
        return True
    else:
        raise ValidationError(_('Gene %(value)s not known'),
                              code='Unknown',
                              params = {'value': val})

class GeneField(forms.CharField):
    description = "typeahead field for selecting genes"
    default_validators = [validate_gene]
    widget = GeneWidget

    def clean(self, value):
        cleaned_data = super(GeneField, self).clean(value)
        return Gene.objects.get(name=cleaned_data)
```

Figure 5B: Django form code, and the typeahead widget that it generates.  Widgets have built-in code that specifies the HTML that is displayed.

In the example above I declare a custom field, the GeneField, in order to provide additional features: first, in order to provide user-defined validation for logic, and second, to customize both the Javascript (and even the HTML) that is needed for a field.  For example, the GeneField validates that the returned string corresponds to a Gene object, and the clean method converts the returned string to a Gene Object.  In order to implement typeahead, the GeneWidget adds the class attribute 'gene-typeahead' to the HTML, and also stipulates the necessary Javascript for the field.  In this way, multiple forms can use the GeneField with a minimum of code duplication in the template (the view in MVC), one instance where Django implements a don't repeat yourself (DRY) philosophy.

Views: Application logic

On the views.py script (Figure 5C), the view handles both form generation (when the incoming HTML request is a GE)T and retrieving form input (when the request is a POST).  If the request is a GET, the view pre-fills the form with query parameters and then returns a rendered form.  If the request is a POST, then the application prepares to save the interaction.  If the interaction is valid, then it is saved, but if it exists already, then the existing interaction is retrieved.   This framework allows us to add further logic to individual fields to restrict interactions.  For example, a validate method on the form could ensure that the source and target protein were not identical.

Afterwards, if a reference is provided by the user, the reference is also saved.  Finally, the user is redirected to a list of annotations, which should contain the interaction and reference just added.

```
# views.py

def add_interaction(request):
    if request.method == 'GET':
        initialInteraction = dict(request.GET.append('input_source': 'Community'))
        base_form = InteractionForm(initial = initialInteraction)

        context = dict(path = request.path,
                       user = request.user,
                       interaction_form = base_form)
        return render(request, 'annotations/add_interaction.html', context)
    elif request.method == 'POST':
```

```
    interaction_form = InteractionForm(request.POST)
    interxn = []
    if interaction_form.is_valid():
        interxn = interaction_form.save(commit = False)
        interxn.save()
    elif interaction_form.non_field_errors().as_text() == '* Interaction is not unique.':
        interxn = Interaction.get(**interaction_form.cleaned_data)

    if interxn:
        ref_id = interaction_form.cleaned_data['reference_identifier'] # create
        db = interaction_form.cleaned_data['db']
        if ref_id:
            # look for an existing reference first
             InteractionReference.get_or_create(identifier=ref_id,
                                                interaction = interxn,
                                                db = db,
                                                defaults = {'user': request.user})

        return redirect('list_interactions', interxn.source.name + ',' + interxn.target.name)


 # in case of validation error
    return render(request, 'annotations/add_interaction.html',
           dict(path = request.path,
              user = request.user,
              interaction_form = interaction_form))
```

Figure 5C: Django view code.  Error handling is checked in the is_valid and get_or_create methods.

If a validation error is produced at any point while creating model instances, then the form with the user's data is rendered and sent back to the user, but Django also adds reported errors back to the form to be rendered as well; the template below provides space for these errors to be displayed.  One functional requirement on this point which is not yet satisfied is that nothing should be saved to the database if an error has occurred while saving.  This ought to be handled with database transactions, which specify an all or nothing semantics - either every entity is saved at once, or none are.

## Templates: User Interface elements

I wrote a simple HTML page with Django's template facility to render the form (Figure 5D). Django allows plain HTML in its template language.  Thus, the tags that include form fields in double branches can be embedded within normal HTML tags.  Also, Django provides simple branching and looping capabilities as well.  For instance, if the form was sent with errors, those errors will be rendered above the form.  One last (possibly overlookable) feature is that Django allows dictionary lookups and no-argument member function calls through the dot operator; this lets much of the retrieval of related objects occur within the template rather than being prefetched in the view.

```
{% extends "layout.html" %}
{% load annotation_tags %}
{% load widget_tweaks %}
{% block title %}MAGI: Add protein-protein interaction {% endblock %}
{% block content %}
<h3> Add protein-protein interactions </h3>

{% if interaction_form.errors %}
<div class = 'alert alert-danger'>
  Errors occurred while adding your referenced interaction.
  {{ interaction_form.errors }}
</div>
{% endif %}
<form method='POST' action="{% url 'annotations:add_interactions' %}">
  {% csrf_token %}
  <div class = "panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title">
        Provide a referenced interaction
      </h3>
    </div>
    <div class='panel-body'>
      <div class='form-group'>
        {{ interaction_form.source.label_tag }}
        {{ interaction_form.source|add_class:'form-control' }}
      </div>
      <div class='form-group'>
        {{ interaction_form.target.label_tag }}
        {{ interaction_form.target|add_class:'form-control' }}
      </div>
      <div class='form-group'>
        {{ interaction_form.reference_identifier.label_tag }}
        {{ interaction_form.db|add_class:'form-control' }}
        {{ interaction_form.reference_identifier|add_class:'form-control' }}
      </div>
      {{ interaction_form.input_source }} {# hidden #}

    </div>
  </div>
  <button class="btn btn-primary">Submit</button>
</form>
{% endblock %}
{% block scripts %}
```

```
{{ interaction_form.media }}
{% endblock %}
```

Figure 5D: Django template code.

Notably, Django also allows template extension (via the extends tag), so that multiple pages on annotation MAGI can have the same MAGI header and footer as main MAGI.

Urls: Providing internal and external addresses

The urlpatterns variable contains a mapping of regular expressions on URIs to both view functions and names within Django (Figure 5E). The URL pattern for adding interactions maps both GET and POST requests to the add_interactions method, and also maps the name ‘add_interactions’ to the URI for certain methods within Django. One instance of this is in the view page (Figure 4C) where a successfully added interaction redirects to ‘list_interactions’, which is another url listed within the patterns. This allows code within Django to rely on individual names rather than hard-coded URIs, another instance of the DRY principle.

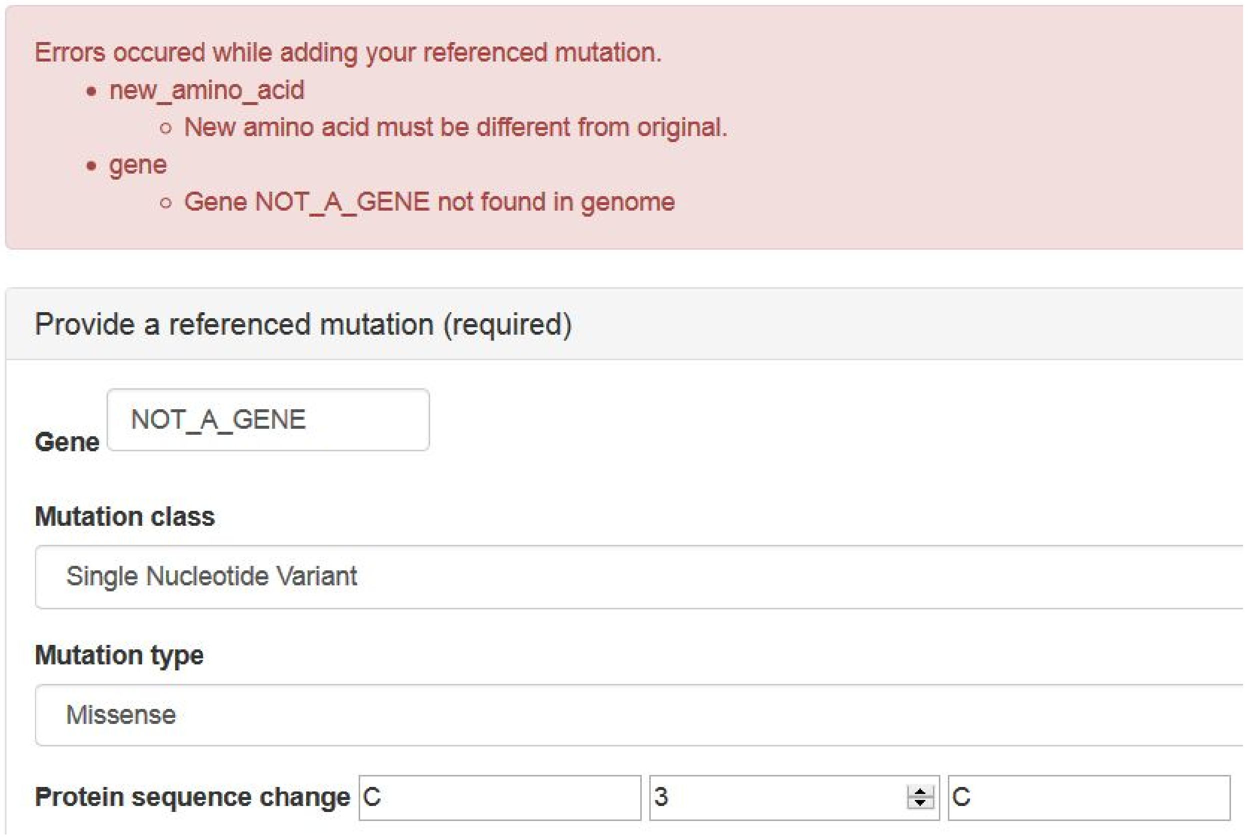
```
# urls.py

from django.conf.urls import url
from . import views

urlpatterns = [
...
    url(r'^interactions/add/$', views.add_interactions, name='add_interactions'),
    url(r'^interactions/(?P<gene_names>[A-Za-z0-9,]+)/$', views.list_interactions, name='list_interactions'),
...
]
```

Figure 5E: Django URL code. URI’s are captured by regular expressions and sent to the corresponding method.

## ***Section 4: Feature Highlights:***

To recap, I implemented an extension to main MAGI for annotations on the Django platform. Aside from reproducing the work previously done in Mongo on main MAGI, I also implemented several extensions and improvements.

One highlight of this work is providing simple and extensible validation and error reporting on any input forms (Figure 6). Rather than relying on the database to report which constraints are violated, Django allows the developer to catch and declare errors at the form and model layer. This makes errors more understandable to the user and provides a better experience. I also note that Django’s language makes it possible to handle errors without relying on the cumbersome callback pattern of passing success or failure handlers common to Express.



Errors occured while adding your referenced mutation.

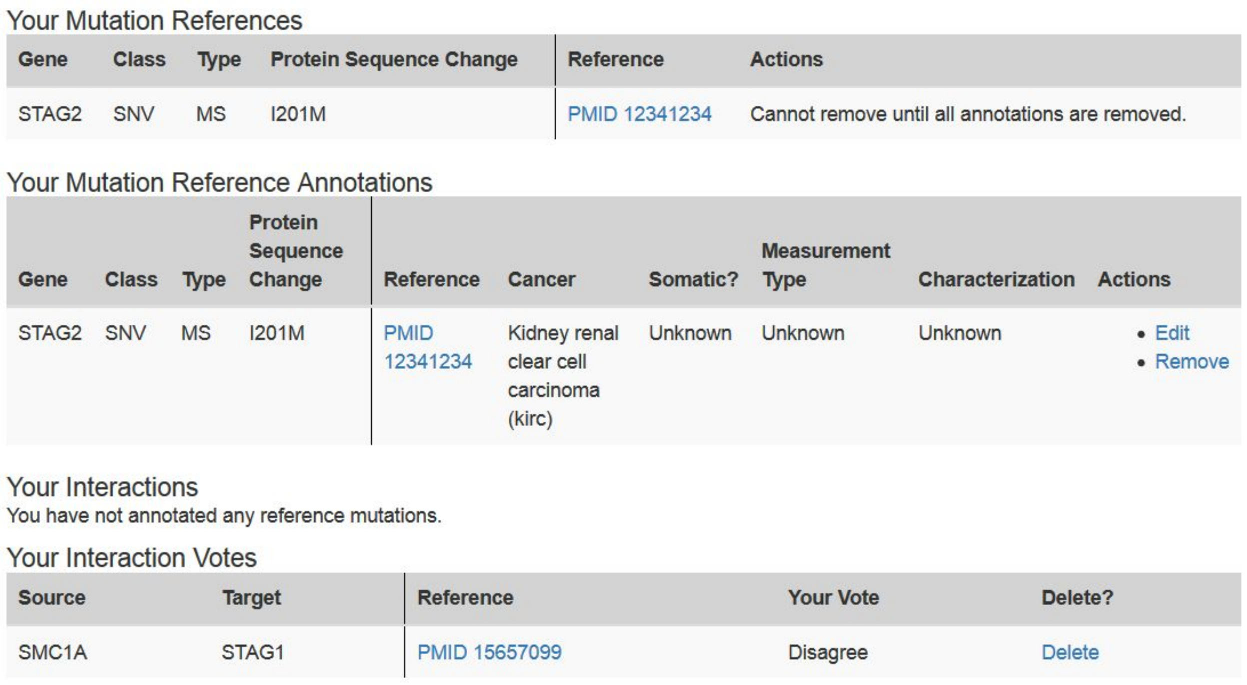
- new_amino_acid
  - New amino acid must be different from original.
- gene
  - Gene NOT_A_GENE not found in genome

Provide a referenced mutation (required)

| | |
|---|---|
| **Gene** | NOT_A_GENE |
| **Mutation class** | Single Nucleotide Variant |
| **Mutation type** | Missense |
| **Protein sequence change** | C 3 C |

Figure 6:Example of error reporting within a form.

In addition I also provided user access to all references and annotations, and allow the user to delete those entities (Figure 7). This capacity is made very simple by Django's ORM, which abstracts the delete operation on objects. This is important for two reasons: it allows defaults such as cascading deletes (the deleting of related objects) to be handled by the Django developer, and also allows simple safety checks for deletion. For instance, we retrieve the object to check for existence, and check whether the user owns the annotation he/she is deleting. These operations would require SQL query generation and manual error checking in Express, but can be succinctly expressed in Django.


Your Mutation References

| Gene | Class | Type | Protein Sequence Change | Reference | Actions |
|---|---|---|---|---|---|
| STAG2 | SNV | MS | I201M | PMID 12341234 | Cannot remove until all annotations are removed. |

Your Mutation Reference Annotations

| Gene | Class | Type | Protein Sequence Change | Reference | Cancer | Somatic? | Measurement Type | Characterization | Actions |
|---|---|---|---|---|---|---|---|---|---|
| STAG2 | SNV | MS | I201M | PMID 12341234 | Kidney renal clear cell carcinoma (kirc) | Unknown | Unknown | Unknown | • Edit<br>• Remove |

Your Interactions
You have not annotated any reference mutations.

Your Interaction Votes

| Source | Target | Reference | Your Vote | Delete? |
|---|---|---|---|---|
| SMC1A | STAG1 | PMID 15657099 | Disagree | Delete |

Figure 7: Interface for logged in users to manage their added references and annotations.

Another highlight is providing cross-origin access to the Django database from main MAGI in the back end. I configured the PostgreSQL database on cobra to allow host-based authentication (HBA) from hosts other than cobra, such as hepburn and cbio-test. I then wrote code for main MAGI to lookup references and annotations from the Django SQL database. This relieves us from having to use Express to manage the SQL database, but we now have to synchronize the database definitions in Django with main MAGI's view of the SQL database schemas.

I also allow annotation MAGI to redirect their URLs to main MAGI. Because we expect that multiple MAGI instances may be interested in the same knowledge curated by different MAGI, we would like them all to see the annotations MAGI pages. However, we would need to ensure that all links on annotation MAGI return the user to the main MAGI pages of origin. Because both main MAGI and annotation MAGI provide the exact same navigation bar

I can do this through reading the referring HTTP page in HTTP request headers, and keeping them within a session cookie for each time the user navigates to the annotation pages. This is a feature in progress; we currently allow a single hop from other main MAGI pages to navigate back to main MAGI from the annotations page.

As experimental work, I implemented a different form of the sample view page in annotations MAGI. This page uses a cascading view to first list mutated genes, then references on those genes, then annotations on those references. In addition, I implemented filters for genes and mutations, so that the user can view only genes or mutations with references or user annotations, respectively.  Designs like this allow sample view pages to be loaded more quickly because less data has to be pulled from the database and sent over the network at once. However, this may be offset by Python's slower runtimes.  The work is developed in the tumorsamples branch on Bitbucket.

## *Section 5. Additional operations work:*

### *1. Docker for fast deployment*

Early in the project, we foresaw a need to deliver private instances of MAGI to other users without requiring complex setup or superuser privileges.  The existing solution was to deliver MAGI as an Amazon machine image, but the installing user would still need to run all the necessary services.  I wrote scripts and instances to start a MAGI instance reproducibly with a minimal number of commands, and also put MAGI on the Docker hub so that other users could potentially pull MAGI as if they were pulling repositories.  These MAGI instances host the standalone MAGI version with one Express.js server, rather than the hybrid system.

Currently, the Brown department VMs have stability issues with Docker containers; long-running Docker processes tend to be stopped after 2-3 weeks, so we instead use detached tmux consoles

Background

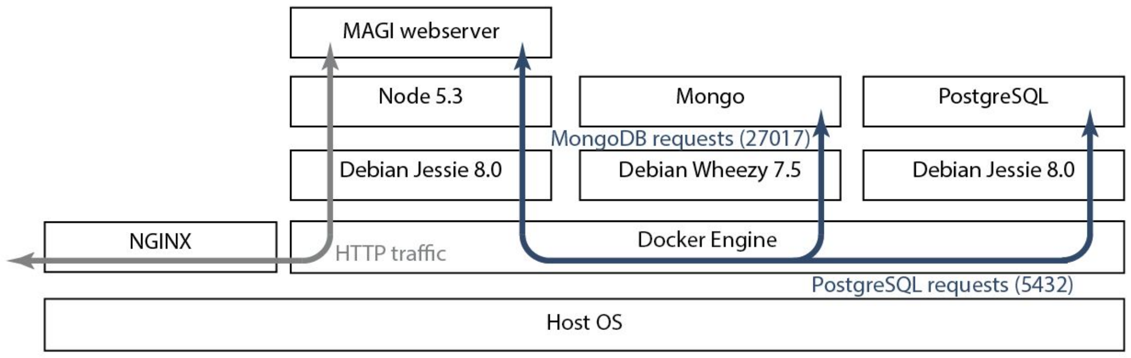
Docker is a container virtualization tool to isolate different processes of a server or operating system environment in a lightweight, portable, and independent manner.  It allows processes that rely on different OSes to work and communicate with each other (similar to a hypervisor). Docker takes advantage of the Linux container system (LXC) to efficiently share memory, networking, and I/O resources, and layered file systems (AUFS) to share disk resources without redundancy (Figure 8A).

For our use, containers are analogous to processes that run on a machine, but within their own environment.  Images are analogous to packages that outline process execution.  Docker is meant to make container execution simple.

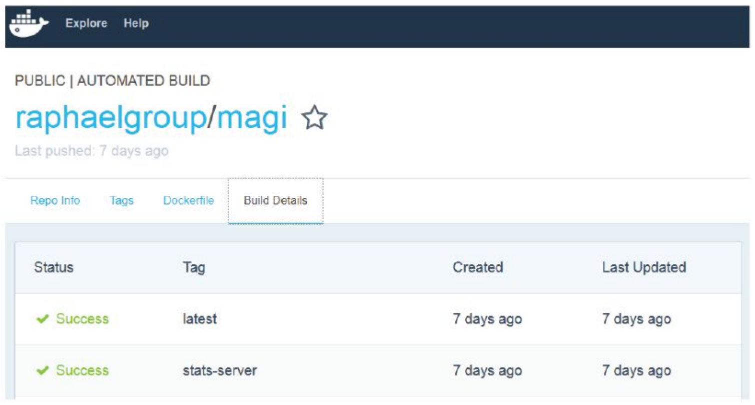
For MAGI, we use Docker to:
- provide a pre-built version of MAGI for rapid deployment
- maintain a separate MAGI server environment from the host machine
- maintain separate database operations from MAGI

Notably, Docker also hosts a centralized registry of containers from other developers called the Docker Hub; most OSes and development environments have containers available that can be pulled from Docker.  As well as providing open-source access to the MAGI docker containers, the Docker Hub also provides automated build testing (Figure 8B).

A



B



PUBLIC | AUTOMATED BUILD

raphaelgroup/magi

Last pushed: 7 days ago

| Status | Tag | Created | Last Updated |
|---|---|---|---|
| ✓ Success | latest | 7 days ago | 7 days ago |
| ✓ Success | stats-server | 7 days ago | 7 days ago |

Figure 8:  MAGI Docker architecture (A) and automated builds within Docker Hub (B)

Installation instructions:

For more details about the Docker command line options, see the documentation.
Currently, MAGI deploys by separately starting three containers within docker.

MAGI web server: This container contains MAGI and all of its dependencies (node/npm, node libraries, gd3, and the TCGA PanCancer data) as well as nginx (a web server).  These dependencies are retrieved while building the underlying image.

The MAGI container relies on the mongo database and the statistics server to run properly. Because these reside as containers, these must be linked when running the MAGI image.  (This requires the --link parameter when starting the container.)

MAGI relies on several different parameter values to run: these are described in MAGI Github. These are equivalent to configuration files for MAGI instances and should be given as key=value pairs to Docker. (This requires the --env-file parameter when starting the container.)

When the container starts, nginx forwards requests from port 80 to the MAGI server port *within the container* before MAGI starts. (In order to forward traffic from a host to a container, use the -p parameter.)

Once the other containers are running, the following command starts the MAGI container

```
> docker run -i -t \ #interactive mode with attached tty
     --link boxed-stats:enricher --link boxed-mongo:mongo \
     -p 80:80 \
     --name magi-instance \
     --env-file=magi.env \
     raphaelgroup/magi:latest
```

Mongo container: This container runs a Mongo database, exposing port 27017 to the host machine and to other containers. This container is provided by Mongodb and can be pulled from the Docker Hub.

Importantly, the Mongo container (and others) can mount a host directory into its container, allowing existing databases to be imported, e.g. from other MAGI instances. (Mounting is done with the -v option.)

To run the mongo container:

```
> docker run -d \ # detached mode
     -v /path/to/database/on/host:/data/db \
     --name boxed-mongo \
     mongo
```

By default, port 27017 is opened to the host.

Implementation details:

Docker containers are built from scripts called Dockerfiles; the MAGI images Github repository stores the current version of the dockerfiles. There is an associated Dockerhub repository for MAGI, with the following images

- raphaelgroup/magi:stat-server
- raphaelgroup/magi:latest

Under the hood, Dockerfiles are responsible for downloading packages and other software to support the container (i.e. node/npm, gd3, nginx, git, etc.)  See the Dockerfile documentation for more info.

Whenever commits are pushed to the MAGI repository on any branches, Docker initiates an automated build of the MAGI Docker image (using Webhooks).  Automatic builds can be checked here or can trigger downstream services with Webhooks.

Startup scripts are provided in the Github repo as well: ./deploy-all.sh is the one-touch startup script.  There is important initial configuration that must occur when an individual container is run: these commands are stored within ./run-server.sh INSIDE the container.  In addition, ./load-TCGA-data.sh downloads and imports the data from the Raphael group website into Mongo.

## Section 6: Deployment and Current Status

We are currently in the process of migrating from the old Express.js version of MAGI.  Each new instance of MAGI will now have two servers, one for the Express.js side of MAGI that includes gene queries, and one for the Django side of MAGI that manages annotations.  Below is a table of which VMs will be used for each instance.

| MAGI Instance | Express server VM | Django server VM |
|---|---|---|
| Public | bogart | annotations (cobra) |
| PAAD | hepburn (paad) | finch |
| Staging | cbio-test | beagle |

Deployment guides are found in the README of each repository.  In brief, the setup steps for the Express server are:

- Install MongoDB and node.js dependencies
- Retrieve the latest version of gd3
- Start MongoDB
- Download and preload public gene and cancer datasets
- Configure environment variables for MAGI, MongoDB and Django connections

The setup steps for the Django server are:

- Install postgres, Python and Bower
- Configure environment variables for MAGI
- Run all migrations to initialize the database
- Convert data files to fixtures and load initial datasets as fixtures

- Create a superuser.
- For remote postgres communication purposes, allow host-based authentication from the express Server VM.

## Section 7: Conclusion

In this paper I described implementing annotations into MAGI within the Django framework atop PostgreSQL. The PostgreSQL backend allows more memory-efficient access that is easier to maintain. Using the Django framework lowers the difficulty and amount of code required for development while also providing useful functionality such as reusable code, database management and interface enhancements. In addition, Django and PostgreSQL can be integrated in parallel with Express.js for a smooth transition between sites.

It will be helpful to extend the scope of MAGI's mutations to include copy number aberrations and other types of mutations. Another point of improvement would be to implement the main view pages of MAGI in Django. Towards this effort, future directions should explore performance enhancements to offset Python's relatively slow runtime. Finally, implementing in-browser testing with Watir in Ruby should improve development cycles and reliability; an appendix describes the work done thus far towards in-browser testing.

## Acknowledgments:

I would like to thank Ben Raphael for advising me through my Master's project and for providing part of my summer funding. Also, many thanks to Max Leiserson for providing direction and for many helpful conversations.