Query Universal Interface

Haak Saxberg and Jess Hester

December 9, 2011

Contents

Listings

List of Figures

1 Introduction

The need to store persistent data is a headache for many developers. Often, they encounter this problem early on, and due to time or budget pressure, their solution is necessarily tailored to the current realities of their applications – with a limited view, if any, of their application's future, scaled-up needs.

This makes transporting an application from one storage engine to another a time-consuming and expensive affair; all too often, the costs associated with the migration prevent the application from migrating for an extended period of time, which generally means that users aren't getting 100% use.

In the past, this issue was annoying but manageable, since persistent storage was frequently implemented using relational databases. By the mid 90's, nearly all relational databases were mostly SQL-compliant. Every vendor, of course, had their own flavor of SQL which presented the eternal threat that a critical query wouldn't run on a new database, but for the most part SQL was SQL.

In the modern era, companies are increasingly looking to non-relational databases as storage solutions, because they can be scaled up much more cost-effectively and quickly – a must in today's data-driven business world. However, non-relational databases don't yet have a common language like SQL; one database's SELECT could very well be another's INSERT.

Enter the Query Universal Interface (QUI).

QUI attempts to ease the transition between non-relational databases, like other ORMs do for relational databases. QUI does not pretend to implement a universal querying syntax, but rather a universal interface through which code can talk to data with minimal, if any, change.

2 Language Overview

2.1 Basic Computation

At its heart, QUI is a translation engine. It converts class-like data into a form that is understandable to a backend's API.

2.2 Basic Data Structures

QUI has two foundational elements: Models and Fields.

2.3 Basic Control Structures

Users can fine-tune their interactions with their databases in a couple of ways:

1. Trivially, by passing in configuration arguments to the pre-created Field subclasses
2. Defining their own fields, with custom behaviors
3. Defining their own FieldMixins, customizing data requirements of the
4. Defining their own ModelMixins, customizing their interaction with the databases

2.4 Input/Output

As input, QUI will accept...a valid Python class definition. As output, QUI gives users...an altered (dare we say, improved?) class definition.

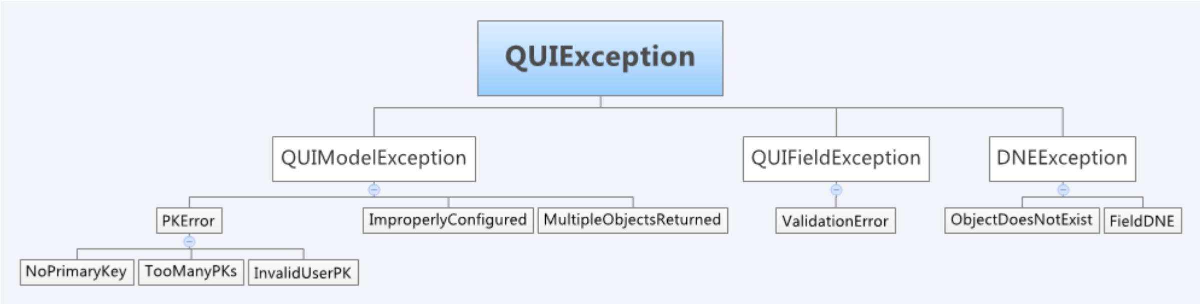

Figure 1: QUI's exception hierarchy

2.5 Error Handling

QUI defines several custom subclass of `Exception`, `QUIException`, so that the user can be informed about exactly what QUI thinks has gone wrong, and where, and with what values. QUI implements a few "levels" of exception, insofar as Python supports "levels" of program running. At the "compilation" level, QUI raises `ImproperlyConfigured` exceptions for any settings that are not valid or missing, with appropriate error messages alongside.

Every time a `QUIException`, or one of its derived exceptions, is raised, QUI prints an informative error message, detailing its best guess for what caused the exception. Along side, it prints the exception raised by the raw Python—just in case the guess is wrong.

Figure 1 describes the exceptions provided by QUI, and their relation to each other.

2.6 Tool Support

2.7 Alternatives to QUI

There are a few interfaces which share a common purpose with QUI – the universalization of database access. None, however, make it their focus to unify *non-relational* databases. Packages like SQLAlchemy and Django's proprietary ORM work to universalize access to many SQL-based relational databases, but neither officially supports non-relational databases of any sort.

QUI attempts to address an untapped niche market; its declared purpose is unique amongst ORMs.

3 Example Programs

3.1 Model Definition

Here's what it might look like if you wanted to define a simple model:

```python
from qui.models import Model
from qui.fields import *
from qui.decorators.storage import stored
from qui.decorators.field_decorators import class_field

@stored(backend="AppEngine")
class FileModel(Model):
    """ A simple file model.

    This model will inherit directly from a supplied mixin - in this case, one
    that enables the model to talk to Google's AppEngine service.

    If your backend changes, the only code you need to change for this model is
    the the decorator argument - instead of "AppEngine", put "MongoDB"
    (for example), or whatever is appropriate for your backend.
    """

    count = class_field(IntegerField)

    name = StringField
    size = IntegerField
    filetype = StringField
    notes = StringField
    created = DateField
    is_safe = BooleanField

    class_var = 100

    def my_size(self):
        """ A function specific to this model, unmanaged by QUI """
        return u"{} bytes".format(self.size)

    def __unicode__(self):
        if self.name:
            return u"{}".format(self.name)
        else:
            return u"{}".format("Unnamed File")
```

Listing 1: filemodel.py – defining a simple model

`FileModel`'s family tree, after decoration, looks like this:

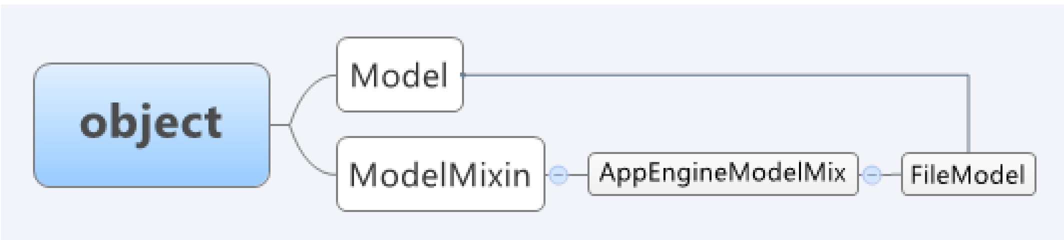

Figure 2: `FileModel`'s family tree.

3.1.1 Step-by-step

Let's step through this definition. First, we import the necesassary things from qui:

```python
from qui.models import Model
from qui.fields import *
from qui.decorators.storage import stored
from qui.decorators.field_decorators import class_field
```

`qui.fields` contains all the supported field types, like `StringField, IntegerField`, etc. At the moment, QUI supports a bare minimum of fields - just the most basic types.

Next, we decorate the class with `storage()`:

```python
@stored(backend="AppEngine")
class FileModel(Model):
```

the `backend` keyword argument to the `storage` decorator tells QUI which mixins it needs to add to your model's inheritance tree, as well as the inheritance trees of any `Fields` you've defined for the model.

`storage` will accept arbitrary keyword arguments, but only acts on a few:

1. `backend`, which defaults to `None` and will (generally) raise an `ImproperlyConfigured` exception if not defined, or if you pass in a backend-identifying string that isn't recognized.
2. `host`, which takes a string and defaults to `'localhost'`.
3. `db`, which takes a string and defaults to a backend-specific value.
4. `port`, which takes an integer, and defaults to a backend-specific port.

Our simple `FileModel`, then, uses the AppEngine backend, and will be stored locally, with default settings. Any subclasses of `FileModel` will automatically inherit these settings, but fine control is possible by defining `_host, _db, _port` on the subclass in question.

Next, we set some fields that we think a `FileModel` should have:

```
    count = class_field(IntegerField)

    name = StringField
    size = IntegerField
    filetype = StringField
    notes = StringField
    created = DateField
    is_safe = BooleanField

    class_var = 100
```

Notice that we don't *initialize* any of the fields here - we're just defining what kind of `Field` each attribute should be. Although QUI will store any attributes that an instance of `FileModel` may have (except private ones and callable ones), these `Field` attributes will do run-time compatibility checking. For example:

```
$ python -i FileModel.py
>>> f = FileModel()
>>> f.size = "really big"
Traceback (most recent call last):
...
...
fields.ValidationError: Could not convert to int: really big
>>> f.size = 100
>>> f.size
100
>>>
```

Listing 2: Session 1 – demonstrating run-time validation

`count` has been marked as a class attribute by the `class_field` decorator, and it will behave just like any other class attribute. Also, notice that you can still have "regular" class variables, like `class_var`; these will be stored by QUI just like all other non-callable attributes.

There's a (pedantic) function definition, just to show that we can, and (finally) we have `__unicode__`, which controls the way that a QUI model is printed via the `print` commands. :

```
    def my_size(self):
        """ A function specific to this model, unmanaged by QUI """
        return u"{} bytes".format(self.size)

    def __unicode__(self):
        if self.name:
            return u"{}".format(self.name)
        else:
            return u"{}".format("Unnamed File")
```

Since `my_size` is a callable, QUI ignores it - but doesn't remove it from the model.

```
$ python -i FileModel.py
>>> f = FileModel()
>>> f
<FileModel: Unnamed File>
>>> print f
Unnamed File
>>> isinstance(f, FileModel)
True
>>> f.size
>>> f.my_size()
'None bytes'
>>> f.size = 100
>>> f.my_size()
'100 bytes'
>>>
```

Listing 3: Session 2 – using fields with functions

3.1.2 Subclassing a Model

```
from filemodel import FileModel
from qui.decorators.storage_decorators import subclass
from qui.decorators.field_decorators import class_field

class ImageFile(FileModel):
    pass


class TextFile(FileModel):
    word_count = IntegerField
    txt_file_count = 100

@subclass
class MoreClassFields(FileModel):
    special_count = class_field(IntegerField)

class RemoteFile(FileModel):
    _port = 91711

    def __init__(self, host):
        super(Remote_File, self).__init__(self)
        self._host = host
```

Listing 4: file_subclasses.py – subclassing from FileModel

This is just an example of a few ways that you can customize your QUI models. `ImageFile` is a direct subclass of `FileModel`, and has no further customizations. QUI will store it instances of `ImageFile` in a different 'place' than it does instances of `FileModel`, if that makes sense for the backend.

`TextFile` shows that you can define further `Fields` or regular class member variables, as you'd expect from a subclass.

`MoreClassFields` demonstrates the `subclass` decorator, which is only needed if your subclass defines more class-wide `Fields`.

`Remote_File` exhibits the ability to redefine where the model should be stored, by customizing `_host` and `_port` members. The difference between defining them as class members and as instance members is subtle: all QUI functions will use the class member values, if they exist; only `put()` will use the instance values. So, `Remote_Files` will be **gotten** from the wherever `FileModel` has said to find them, but will be **put** to wherever `_host` has been initialized to. In this case, if `x = Remote_File.create('otherhost')`, it will be `created` to `FileModel`'s host (localhost), but on port 91711. Then `x.put()` will store it on otherhost, also using port 91711. This feature is great for migrating data between hosts; you'll be `get()`ing from one host and `put()`ing to another!

Note that if you define a custom `__init__`, you also need to call `super`'s `__init__` before you start modifying any instance attributes, so that QUI can do the necessary setup.

3.1.3 Regarding Fields

We really can't stress enough that you generally *do not initialize `Fields` yourself.* All direct subclasses of `Field` are abstract, and even if they weren't, they do not define all the methods that `Field` requires in order to be instantiated. They have to be combined with a subclass of `FieldMixin` which fulfills their remaining requirements.

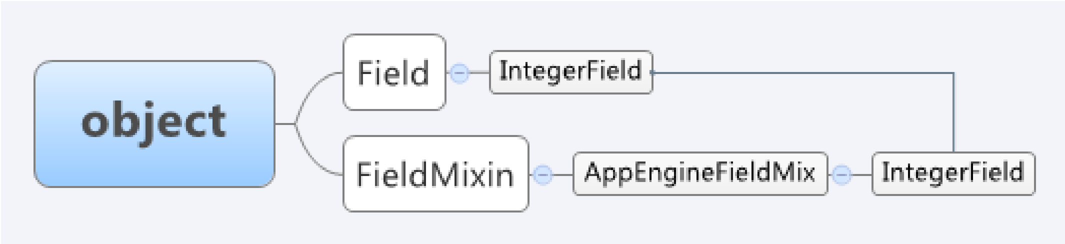

Figure 3: A family tree for `FileModel.size`.

QUI does this for you when you decorate a class definition, and instantiates the resulting fields at the appropriate times—so that every `FileModel` has its own `name`, but the same `count`. Defining `Fields` on your model isn't required; one of the best parts about non-relational databases is that they don't enforce a strict schema. Remember `x`, from above? If we had, somewhere down the line, decided that `x.sensitive =True`, QUI would happily store the `sensitive` attribute right along with all of `x`'s other attributes. Defining an attribute as a `Field` is mostly useful for enforcing a type, or coercing into a certain format:

```
>>> x.name = 123
>>> x.name
u'123'
```

3.2 Using QUI

QUI's ease of use is best exhibited by entering the Python interpreter:

```
$ python -i FileModel.py
>>> f = FileModel.create(
... name="arrow.jpg",
... size=100,
    title="Greenboy")
>>> f
<FileModel: arrow.jpg>
>>> f.title
'Greenboy'
>>> f.size
100
>>> f.size = "really big"
Traceback (most recent call last):
...
...
fields.ValidationError: Could not convert to int: really big
>>>
```

Listing 5: Session 3 – creating an instance with arbitrary attributes

4 Language Design

4.1 Getting, Putting, and Creating

The idea behind QUI is to support a uniform set of 'verbs' across all backends. As an internal language focusing on storage, we implemented these verbs as class methods and instance methods.

Since our time for this project was limited, we focused our efforts on what we saw as the three most important verbs for a persistent storage solution:

1. **Creating** an instance of a stored model, at run-time, on the fly, and having it stored into the backend.
2. **Putting** an instance of a stored model into the backend.

3. **Getting** an instance of a stored model from the data stored in the backend.

4.2 Syntax

QUI strives for an optimal combination of clarity and concision. It achieves this by only officially exposing three extra methods on stored models: `get()`, `create()`, and `put()`.

4.3 Semantic Abstractions/Building Blocks

5 Language Implementation

5.1 Host Language

QUI is implemented in Python, which was chosen for three main reasons:

1. Our own familiarity with it.
2. Excellent metaprogramming facilities.
3. Availability of backend APIs.

5.2 Parsing

As an internal DSL, QUI doesn't do 'parsing', in the same sense as an external one — there's no need for an abstract syntax tree, and we can rely on Python to catch syntactically illegal expressions in many cases.

5.3 Execution

As an internal DSL, QUI's semantics do not differ from those of its host language, Python. Figure 4 describes the operations that QUI is doing behind the scenes when a class is decorated with `storage`.

The `subclass` decorator is similar, but only performs the operations relating to class-wide fields, since a subclass of a QUI model will already have much of the setup done by its parent class.

6 Evaluation

For the most part, it all seems to work; multiple databases are supported, and our design is modular enough that support for more databases would be simple to add, if we so wished.

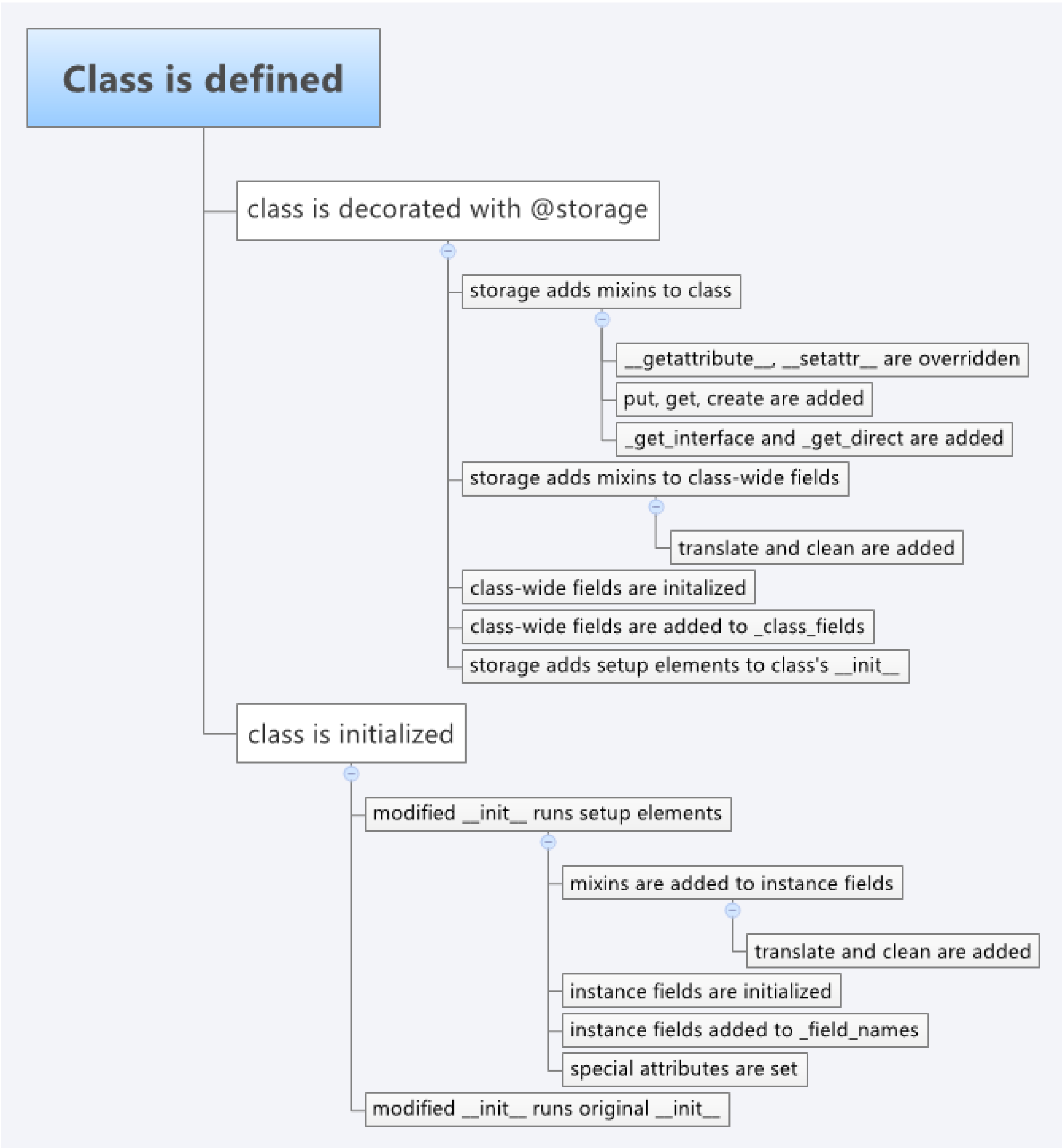

Figure 4: QUI's order of operations