fhirbug Documentation

Vangelis Kostalas

Jan 26, 2019

Contents:

Fhirbug intends to be a full-featured FHIR server for python >= **3.6**. It has been designed to be easy to set up and configure and be flexible when it comes to the rest of tools it is combined with, like web frameworks and database interfaces. In most simple cases, very little code has to be written apart from field mappings.

Fhirbug is still under development! The API may still change at any time, it probably contains heaps of bugs and has never been tested in production. If you are interested in making it better, you are very welcome to contribute!

What fhirbug does:

- It provides the ability to create "real time" transformations between your ORM models to valid FHIR resources through an extensive mapping API.
- Has been designed to work with existing data-sets and database schemas but can of course be used with its own database.
- It's compatible with the SQLAlchemy, DjangoORM and PyMODM ORMs, so if you can describe your database in one of them, you are good to go. It should also be pretty easy to extend to support any other ORM, feel free to submit a pull request!
- Handles many of the FHIR REST operations and searches like creating and updating resources, performing advanced queries such as reverse includes, paginated bundles, contained or referenced resources, and more.
- Provides the ability to audit each request, at the granularity that you desire, all the way down to limiting which of the attributes for each model should be accessible for each user.

What fhirbug does not do:

- Provide a ready to use solution for a Fhir server. Fhirbug is a framework, we want things to be as easy as possible but you will still have to write code.
- Contain a web server. Fhirbug takes over once there is a request string and request body and returns a json object. You have to handle the actual requests and responses.
- Handle authentication and authorization. It supports it, but you must write the implementation.
- A ton of smaller stuff, which you can find in the **Roadmap_**.

CHAPTER 1

Quickstart

Contents:

This section contains a brief example of creating a simple application using fhirbug. It's goal is to give the reader a general idea of how fhirbug works, not to provide them with in-depth knowledge about it.

For a more detailed guide check out the *Overview* and the *API* docs.

1.1 Preparation

In this example we will use an sqlite3 database with SQLAlchemy and flask. The first is in the standard library, you can install SQLAlchemy and flask using *pip*:

```
$ pip install sqlalchemy flask
```

Let's say we have a very simple database schema, for now only containing a table for Patients and one for hospital admissions. The SQLAlchemy models look like this:

Listing 1: models.py

```python
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, DateTime, Integer, String, ForeignKey
from sqlalchemy.orm import relationship

Base = declarative_base()


class PatientModel(Base):
    __tablename__ = 'patients'

    patient_id = Column(Integer, primary_key=True)
    first_name = Column(String)
    last_name = Column(String)
    dob = Column(DateTime)     # date of birth
    gender = Column(Integer)   # 0: female, 1: male, 2: other, 3: unknown


class AdmissionModel(Base):
    __tablename__ = 'admissions'

    id = Column(Integer, primary_key=True)
    status = Column(String) # 'a': active, 'c': closed
    patient_id = Column(Integer, ForeignKey('patients.patient_id'))
    date_start = Column(DateTime)  # date and time of admission
    date_end = Column(DateTime)    # date and time of release

    patient = relationship("patientmodel", back_populates="admissions")
```

To create the database and tables, open an interactive python shell and type the following:

```python
>>> from sqlalchemy import create_engine
>>> from models import Base

>>> engine = create_engine('sqlite:///:memory:')
>>> Base.metadata.create_all(engine)
```

1.2 Creating your first Mappings

We will start by creating mappings between our Patient and Admission models and the Patient and Encounter FHIR resources. In our simple example the mapping we want to create looks something like this:

Table 1: Relationships between db columns and fhir attributes for the Patient

DB column	FHIR attribute	notes
patient_id	id	read-only
first_name, last_name	name	first and last name must be combined into a HumanName resource
dob	birthDate	must be converted to type FHIRDate
gender	gender	values must be translated between the two systems (eg: 0 -> 'female')

Mapping in fhirbug is pretty straightforward. All we need to do is:

1. Subclass the model class, inheriting from FhirBaseModel

2. Add a member class called **FhirMap**
3. Inside it, add class attributes using the names of the fhir attributes of the resource you are setting up.
4. Use *Attributes* to describe how the conversion between db columns and FHIR attributes should happen

Since we are using SQLAlchemy, we will use the `fhirbug.db.backends.SQLAlchemy` module, and more specifically inherit our Mappings from `fhirbug.db.backends.SQLAlchemy.models.FhirBaseModel`

So, we start describing our mapping for the Patient resource from the id field which is the simplest:

Warning: Fhirbug needs to know which ORM the mappings we create are for. Therefore, before importing FhirBaseModel, we must have configured the fhirbug settings. If you write the following code in an interactive session instead of a file, you will get an error unless you configure fhirbug first. To do so, just paste the code described *below*.

Listing 2: mappings.py

```
from models import Patient as PatientModel
from fhirbug.db.backends.SQLAlchemy.models import FhirBaseModel
from fhirbug.models.attributes import Attribute

class Patient(PatientModel, FhirBaseModel):
    class FhirMap:
        id = Attribute('patient_id')
```

Note: The fact that we named the mapper class *Patient* is important, since when fhirbug looks for a mapper, it looks by default for a class with the same name as the fhir resource.

By passing the column name as a string to the `Attribute` we tell fhirbug that the id attribute of the Patient FHIR resource should be retrieved from the `patient_id` column.

For the `birthDate` attribute we get the information from a single database column, but it must be converted to and from a FHIR DateTime datatype. So, we will use the `DateAttribute` helper and let it handle conversions automatically.

We will also add the name attribute, using the `NameAttribute` helper. We tell it that we get and set the family name from the column `last_name` and the given name from `first_name`

Listing 3: mappings.py

```
from models import Patient as PatientModel
from fhirbug.db.backends.SQLAlchemy.models import FhirBaseModel
from fhirbug.models.attributes import Attribute, DateAttribute, NameAttribute

class Patient(PatientModel, FhirBaseModel):
    class FhirMap:
        id = Attribute('patient_id')
        birthDate = DateAttribute('dob')
        name = NameAttribute(family_getter='last_name',
                             family_setter='last_name',
                             given_getter='first_name',
                             given_setter='first_name')
```

1.3 Letting the magic happen

Let’s test what we have so far. First, we must provide fhirbug with some basic configuration:

```
>>> from fhirbug.config import settings
>>> settings.configure({
...     'DB_BACKEND': 'SQLAlchemy',
...     'SQLALCHEMY_CONFIG': {
...         'URI': 'sqlite:///:memory:'
...     }
... })
```

Now, we import or mapper class and create an item just as we would if it were a simple SQLAlchemy model:

```
>>> from datetime import datetime
>>> from mappings import Patient
>>> patient = Patient(dob=datetime(1980, 11, 11),
...                   first_name='Alice',
...                   last_name='Alison')
```

This `patient` object we have created here is a classic SQLAlchemy model. We can save it, delete it, change values for its columns, etc. **But** it has also been enhanced by fhirbug.

Here’s some stuff that we can do with it:

```
>>> to_fhir = patient.to_fhir()
>>> to_fhir.as_json()
{
    'birthDate': '1980-11-11T00:00:00',
    'name': [{'family': 'Alison', 'given': ['Alice']}],
    'resourceType': 'Patient'
}
```

The same way that all model attributes are accessible from the `patient` instance, all FHIR attributes are accessible from `patient.Fhir`:

```
>>> patient.Fhir.name
<fhirbug.Fhir.Resources.humanname.HumanName at 0x7fc62e1cbcf8>
>>> patient.Fhir.name.as_json()
{'family': 'Alison', 'given': ['Alice']}
>>> patient.Fhir.name.family
'Alison'
>>> patient.Fhir.name.given
['Alice']
```

If you set an attribute on the FHIR resource:

```
>>> patient.Fhir.name.family = 'Walker'
```

The change is applied to the actual database model!

```
>>> patient.last_name
'Walker'
```

```
>>> patient.Fhir.birthDate = datetime(1970, 11, 11)
>>> patient.dob
datetime.datetime(1970, 11, 11, 0, 0)
```

1.4 Handling requests

We will finish this quick introduction to fhirbug with a look on how requests are handled. First, let's create a couple more entries:

```
>>> from datetime import datetime
>>> from fhirbug.config import settings
>>> settings.configure({
...     'DB_BACKEND': 'SQLAlchemy',
...     'SQLALCHEMY_CONFIG': {
...         'URI': 'sqlite:///:memory:'
...     }
... })
>>> from fhirbug.db.backends.SQLAlchemy.base import session
>>> from mappings import Patient
>>> session.add_all([
...     Patient(first_name='Some', last_name='Guy', dob=datetime(1990, 10, 10)),
...     Patient(first_name='Someone', last_name='Else', dob=datetime(1993, 12, 18)),
...     Patient(first_name='Not', last_name='Me', dob=datetime(1985, 6, 6)),
... ])
>>> session.commit()
```

Great! Now we can simulate some requests. The mapper class we defined earlier is enough for us to get some nice FHIR functionality like searches.

Let's start by asking for all Patient entries:

```
>>> from fhirbug.server.requestparser import parse_url
>>> query = parse_url('Patient')
>>> Patient.get(query, strict=False)
{
    "entry": [
        {
            "resource": {
                "birthDate": "1990-10-10T00:00:00",
                "name": [{"family": "Guy", "given": ["Some"]}],
                "resourceType": "Patient",
            }
        },
        {
            "resource": {
                "birthDate": "1993-12-18T00:00:00",
                "name": [{"family": "Else", "given": ["Someone"]}],
                "resourceType": "Patient",
            }
        },
        {
            "resource": {
                "birthDate": "1985-06-06T00:00:00",
                "name": [{"family": "Me", "given": ["Not"]}],
                "resourceType": "Patient",
            }
        },
    ],
    "resourceType": "Bundle",
    "total": 3,
    "type": "searchset",
}
```

We get a proper Bundle Resource containing all of our Patient records!

1.5 Advanced Queries

This quick guide is almost over, but before that let us see some more things Fhirbug can do. We start by asking only one result per page.

```
>>> query = parse_url('Patient?_count=1')
>>> Patient.get(query, strict=False)
{
    "entry": [
        {
            "resource": {
                "birthDate": "1990-10-10T00:00:00",
                "name": [{"family": "Guy", "given": ["Some"]}],
                "resourceType": "Patient",
            }
        }
    ],
    "link": [
        {"relation": "next", "url": "Patient/?_count=1&search-offset=2"},
        {"relation": "previous", "url": "Patient/?_count=1&search-offset=1"},
    ],
    "resourceType": "Bundle",
    "total": 4,
    "type": "searchset",
}
```

Notice how when defining our mappings we declared `birthDate` as a `DateAttribute` and name as a `NameAttribute`? This allows us to use several automations that Fhirbug provides like advanced searches:

```
>>> query = parse_url('Patient?birthDate=gt1990&given:contains=one')
>>> Patient.get(query, strict=False)
{
    "entry": [
        {
            "resource": {
                "birthDate": "1993-12-18T00:00:00",
                "name": [{"family": "Else", "given": ["Someone"]}],
                "resourceType": "Patient",
            }
        }
    ],
    "resourceType": "Bundle",
    "total": 1,
    "type": "searchset",
}
```

Here, we ask for all `Patients` that were born after 1990-01-01 and whose given name contains `one`.

1.6 Further Reading

You can dive into the actual documentation starting at the *Overview* or read the docs for the *API*.

CHAPTER 2

Overview

2.1 Creating Mappings

Fhirbug offers a simple declarative way to map your database tables to Fhir Resources. You need to have created models for your tables using one of the supported ORMs.

Let's see an example using SQLAlchemy. Suppose we have this model of our database table where patient personal infrormation is stored.

(Note that we have named the model Patient. This allows Fhirbug to match it to the corresponding resource automatically. If we wanted to give it a different name, we would then have to define *__Resource__ = 'Patient'* after the *__tablename__*)

```
from sqlalchemy import Column, Integer, String

class Patient(Base):
    __tablename__ = "PatientEntries"

    id = Column(Integer, primary_key=True)
    name_first = Column(String)
    name_last = Column(String)
    gender = Column(Integer)  # 0: unknown, 1:female, 2:male
    ssn = Column(Integer)
```

To map this table to the Patient resource, we will make it inherit it `fhirbug.db.backends.SQLAlchemy.FhirBaseModel` instead of Base. Then we add a class named **FhirMap** as a member and add all fhir fields we want to support using `Attributes`:

Note: You do not need to put your FhirMap in the same class as your models. You could just as well extend it in a second class while using FhirBaseModel as a mixin.

```
from sqlalchemy import Column, Integer, String
from fhirbug.db.backends.SQLAlchemy import FhirBaseModel
from fhirbug.models import Attribute, NameAttribute
from fhirbug.db.backends.SQLAlchemy.searches import NumericSearch

class Patient(FhirBaseModel):
    __tablename__ = "PatientEntries"

    pat_id = Column(Integer, primary_key=True)
    name_first = Column(String)
    name_last = Column(String)
    gender = Column(Integer)  # 0: unknown, 1:female, 2:male, 3:other
    ssn = Column(Integer)

    @property
    def get_gender(self):
        genders = ['unknown', 'female', 'male', 'other']
        return genders[self.gender]

    @set_gender.setter
    def set_gender(self, value):
        genders = {'unknown': 0, 'female': 1, 'male': 2, 'other': 3}
        self.gender = genders[value]

    class FhirMap:
        id = Attribute('pat_id', searcher=NumericSearch('pid'))
        name = NameAttribute(given_getter='name_first', family_getter='name_last')
        def get_name(instance):
        gender = Attribute('get_gender', 'set_gender')
```

2.2 FHIR Resources

Contents:

Fhirbug uses fhir-parser to automatically parse the Fhir specification and generate classes for resources based on resource definitions. It's an excellent tool that downloads the Resource Definition files from the official website of FHIR and generates classes automatically. For more details, check out the project's repository.

Fhirbug comes with pre-generated classes for all FHIR Resources, which live inside `fhirbug.Fhir.resources`. You can generate your own resource classes based on a subset or extension of the default resource definitions but this is not currently covered by this documentation.

2.2.1 Uses of FHIR Resources in Fhirbug

As return values of `mapper.to_fhir()`

FHIR Resource classes are used when a mapper instance is converted to a FHIR Resource using `.to_fhir()`.

Supposing we have defined a mapper for the Location resource, we could see the following:

```
>>> Location
mappings.Location
>>> location = Location.query.first()
<mappings.Location>
>>> location.to_fhir()
<fhirbug.Fhir.Resources.patient.Patient>
```

As values for mapper Attributes

FHIR Resources are also used as values for mapper attributes that are either references to other Resources, Backbone Elements or complex datatypes.

For example, let's return back to the Location example. As we can see in the FHIR specification, the Location.address attribute is of type Address. This would mean something like this:

```
>>> location.Fhir.address
<fhirbug.Fhir.Resources.address.Address>
>>> location.Fhir.address.as_json()
{
    'use': 'work',
    'type': 'physical',
    [...]
}
>>> location.Fhir.address.use
'work'
```

2.2.2 Creating resources

You will be wanting to use the Resource classes to create return values for your mapper attributes.

The default way for creating resource instances is by passing a json object to the constructor:

```
>>> from fhirbug.Fhir.resources import Observation
>>> o = Observation({
...     'id': '2',
...     'status': 'final',
...     'code': {'coding': [{'code': '5', 'system': 'test'}]},
... })
```

As you can see, this may get a but verbose so there are several shortcuts to help with that.

Resource instances can be created:

- by passing a dict with the proper json structure as we already saw
- by passing the same values as keyword arguments:

```
>>> o = Observation(
...     id='2', status='final', code={'coding': [{'code': '5', 'system': 'test'}]}
... )
```

- when an attribute's type is a Backbone Element or a complex type, we can pass a resource:

```
>>> from fhirbug.Fhir.resources import CodeableConcept
>>> test_code = CodeableConcept(coding=[{'code': '5', 'system': 'test'}])
>>> o = Observation(id='2', status='final', code=test_code)
```

- When an attribute has a cardinality larger than one, that is its values are part of an array, but we only want to pass one value, we can skip the array:

```
>>> test_code = CodeableConcept(coding={'code': '5', 'system': 'test'})
>>> o = Observation(id='2', status='final', code=test_code)
```

Fhirbug tries to make it as easy to create resources as possible by providing several shortcuts with the base contructor.

2.2.3 Ignore missing required Attributes

If you try to initialize a resource without providing a value for a required attribute you will get an error:

```
>>> o = Observation(id='2', status='final')
FHIRValidationError: {root}:
'Non-optional property "code" on <fhirbug.Fhir.Resources.observation.Observation␣
↪object>
is missing'
```

You can suppress errors into warnings by passing the `strict=False` argument:

```
>>> o = Observation(id='2', status='final', strict=False)
```

Fhirbug will display a warning but it will not complain again if you try to save or serve the instance. It's up to you make sure that your data is well defined.

2.2.4 The base Resource class

Tis is the abstract class used as a base to provide common functionality to all produced Resource classes. It has been modified in order to provide a convenient API for *Creating resources*.

class fhirbug.Fhir.base.fhirabstractbase.**FHIRAbstractBase**

2.3 Auditing

With Fhirbug you can audit requests on three levels:

- **Request level**: Allow or disallow the specific operation on the specific resource, and
- **Resource level**: Allow or disallow access to each individual resource and/or limit access to each of its attributes.
- **Attribute level**: Allow or disallow access to each individual attribute for each resource.

Warning: The Auditing API is still undergoing heavy changes and is even more unstable than the rest of the project. Use it at your own risk!

2.3.1 Auditing at the request level

All you need to do do in order to implement request-level auditing in Fhribug is to provide the built-in `fhirbug.server.requesthandlers` with an extra method called `audit_request`.

This method should accept a single positional parameter, a `FhirRequestQuery` and should return an `AuditEvent`. If the outcome attribute of the returned `AuditEvent` is "0" (the code for *"Success"*), the request is processed normally.

```
from fhirbug.server.requesthandlers import GetRequestHandler
from fhirbug.Fhir.resources import AuditEvent

class CustomGetRequestHandler(GetRequestHandler):
    def audit_request(self, query):
        return AuditEvent(outcome="0", strict=False)
```

The simplest possible auditing handler, one that approves all requests.

In any other case, the request fails with status code `403`, and returns an OperationOutcome resource containing the `outcomeDesc` of the `AuditEvent`. This way you can return information about the reasons for failure.

```
from fhirbug.server.requesthandlers import GetRequestHandler
from fhirbug.Fhir.resources import AuditEvent

class CustomGetRequestHandler(GetRequestHandler):
    def audit_request(self, query):
        if "_history" in query.modifiers:
            if is_authorized(query.context.user):
                return AuditEvent(outcome="0", strict=False)
            else:
                return AuditEvent(
                    outcome="8",
                    outcomeDesc="Unauthorized accounts can not access resource␣
↪history.",
                    strict=False
                )
```

Note: Notice how we passed `strict=False` to the AuditEvent constructor? That's because without it, it would not allow us to create an AuditEvent resource without filling in all its required fields.

However, since we do not store it in this example and instead just use it to communicate with the rest of the application, there is no need to let it validate our resource.

Since Fhirbug does not care about your web server implementation, or your authentication mechanism, you need to collect and provide the information neccessary for authenticationg the request to the `audit_request` method.

Fhirbug's suggestion is passing this information through the `query.context` object, by providing `query_context` when calling the request handler's `handle` method.

2.3.2 Auditing at the resource level

Controlling access to the entire resource

In order to implement auditing at the resource level, give your mapper models one or more of the methods `audit_read`, `audit_create`, `audit_update`, `audit_delete`. The signature for these methods is the same as the one for request handlers we saw above. They accept a single parameter holding a `FhirRequestQuery` and should return an `AuditEvent`, whose `outcome` should be `"0"` for success and anything else for failure.

```
class Patient(FhirBaseModel):
    # Database field definitions go here

    def audit_read(self, query):
        return AuditEvent(outcome="0", strict=False)

    class FhirMap:
        # Fhirbug Attributes go here
```

You can use Mixins to let resources share common auditing methods:

```
class OnlyForAdmins:
    def audit_read(self, query):
        # Assuming you have passed the appropriate query cintext to the␣
↪request handler
        isSuperUser = query.context.User.is_superuser

        return (
            AuditEvent(outcome="0", strict=False)
            if isSuperUser
            else AuditEvent(
                outcome="4",
                outcomeDesc="Only admins can access this resource",
                strict=False,
            )
        )

class AuditRequest(OnlyForAdmins, FhirBaseModel):
    # Mapping goes here

class OperationOutcome(OnlyForAdmins, FhirBaseModel):
    # Mapping goes here

    ...
```

Controlling access to specific attributes

If you want more refined control over which attributes can be changed and displayed, during the execution of one of the above `audit_*` methods, you can call `self.protect_attributes(*attrs*)` and /or `self.hide_attributes(*attrs*)` inside them.

In both cases, `*attrs*` should be an iterable that contains a list of attribute names that should be protected or hidden.

protect_attributes()

The list of attributes passed to `protect_attributes` will be marked as protected for the duration of this request and will not be allowed to change

hide_attributes()

The list of attributes passed to `hide_attributes` will be marked as hidden for the current request. This means that in case of a POST or PUT request they may be changed but they will not be included in the response.

For example if we wanted to hide patient contact information from unauthorized users, we could do the following:

```
class Patient(FhirBaseModel):
    # Database field definitions go here

    def audit_read(self, query):
        if not is_authorized(query.context.user):
            self.hide_attributes(['contact'])
        return AuditEvent(outcome="0", strict=False)

    class FhirMap:
        # Fhirbug Attributes go here
```

Similarly, if we wanted to only prevent unauthorized users from changing the Identifiers of Patients we would use `protect_attributes`:

```
class Patient(FhirBaseModel):
    # Database field definitions go here

    def audit_update(self, query):
        if not is_authorized(query.context.user):
            self.protect_attributes = ['identifier']
        return AuditEvent(outcome="0", strict=False)

    class FhirMap:
        # Fhirbug Attributes go here
```

2.3.3 Auditing at the attribute level

Warning: This feature is more experimental than the rest. If you intend to use it be aware of the complications that may rise because you are inside a desciptor getter (For example trying to get the specific attribute's value would result in an infinte loop)

When declaring attributes, you can provide a function to the `audit_set` and `audit_get` keyword arguments. These functions accept three positional arguments:

The first is the instance of the Attribute descriptor, the second, `query` being the `FhirRequestQuery` for this request and the third being the attribute's name It should return `True` if access to the attribute is allowed, or `False` otherwise.

It's also possible to deny the entire request by throwing an *`AuthorizationError`*

`audit_get` (*descriptor*, *query*, *attribute_name*) → boolean

Parameters

- **query** (*FhirRequestQuery*) – The `FhirRequestQuery` object for this request
- **attribute_name** (*str*) – The name this attribute has been assigned to

Returns True if access to this attribute is allowed, False otherwise

Return type boolean

audit_set (*descriptor*, *query*, *attribute_name*) → boolean

Parameters

- **query** (*FhirRequestQuery*) – The `FhirRequestQuery` object for this request
- **attribute_name** (*str*) – The name this attribute has been assigned to

Returns True if changing this attribute is allowed, False otherwise

Return type boolean

2.4 Logging

Fhirbug's `RequestHandlers` all have a method called `log_request` that is called whenever a request is done being proccessed with several information about the request.

By default, this method returns an `AuditEvent` FHIR resource instance populated with available information about the request.

2.4.1 Enhancing or persisting the default handler

Enhancing the generated AuditEvents with extra information about the request and Persisiting them is pretty simple. Just use custom `RequestHandlers` and override the `log_request` method:

```
from fhirbug.Fhir.resources import AuditEventEntity
from fhirbug.config import import_models

class EnhancedLoggingMixin:
    def log_request(self, *args, **kwargs):
        audit_event = super(EnhancedLoggingMixin, self).log_request(*args, **kwargs)

        context = kwargs["query"].context
        user = context.user
        # We populate the entity field with info about the user
        audit_event.entity = [
            AuditEventEntity({
                "type": {"display": "Person"},
                "name": user.username,
                "description": user.userid,
            })
        ]
        return audit_event


class PersistentLoggingMixin:
    def log_request(self, *args, **kwargs):
        audit_event = super(PersistentLoggingMixin, self).log_request(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        models = import_models()
        AuditEvent = getattr(models, 'AuditEvent')
        audit_event_model = AuditEvent.create_from_resource(audit_event)
        return audit_event

# Create the handler
class CustomGetRequestHandler(
    PersistentLoggingMixin, EnhancedLoggingMixin, GetRequestHandler
):
    pass
```

Note: In order to have access to the user instance we assume you have passed a query context to the request handler's handle method containing the necessary info

Note: Note that the order in which we pass the mixins to the custom handler class is important. Python applies mixins from right to left, meaning `PersistentLoggingMixin`'s `super()` method will call `EnhancedLoggingMixin`'s `log_request` and `EnhancedLoggingMixin`'s `super()` method will call `GetRequestHandler`'s

So, we expect the AuditEvent that is persisted by the `PersistentLoggingMixin` to contain information about the user because it is comes before `EnhancedLoggingMixin` in the class definition

2.4.2 Creating a custom log handler

If you don't want to use fhirbug's default log handling and want to implement something your self, the process is pretty much the same. You implement your own `log_request` method and process the information that is passed to it by fhirbug any way you want. Essentially the only difference with the examples above is that you do not call `super()` inside your custom log function.

The signature of the `log_request` function is the following:

Here's an example where we use python's built-in logging module:

```
from datetme import datetime
from logging import getLogger

logger = getLogger(__name__)

class CustomGetRequestHandler(GetRequestHandler):
    def log_request(self, url, status, method, *args, **kwargs):
        logger.info("%s: %s %s %s" % (datetime.now(), method, url, status))
```

CHAPTER 3

API

3.1 Attributes

3.2 Mixins

3.3 fhirbug.server

3.3.1 Request Parsing

3.3.2 Request Handling

3.3.3 Auth

3.4 fhirbug.exceptions

exception `fhirbug.exceptions.`**`AuthorizationError`**(*auditEvent*, *query=None*)

The request could not be authorized.

`auditEvent = None`

This exception carries an auditEvent resource describing why authorization failed It can be thrown anywhere in a mappings `.get()` method.

exception `fhirbug.exceptions.`**`ConfigurationError`**

Something is wrong with the settings

exception `fhirbug.exceptions.`**`DoesNotExistError`**(*pk=None*, *resource_type=None*)

A http request query was malformed or not understood by the server

exception `fhirbug.exceptions.`**`InvalidOperationError`**

The requested opertion is not valid

exception `fhirbug.exceptions.`**`MappingException`**

A fhir mapping received data that was not correct

exception `fhirbug.exceptions.`**`MappingValidationError`**

A fhir mapping has been set up wrong

exception `fhirbug.exceptions.`**`OperationError`**(*severity='error'*, *code='exception'*, *diagnostics="*, *status_code=500*)

An exception that happens during a requested operation that should be returned as an OperationOutcome to the user.

`to_fhir`()

Express the exception as an OperationOutcome resource. This allows us to catch it and immediately return it to the user.

exception `fhirbug.exceptions.`**`QueryValidationError`**

A http request query was malformed or not understood by the server

exception `fhirbug.exceptions.`**`UnsupportedOperationError`**

The requested opertion is not supported by this server

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index