Description of the program generator utility, Generv

An overview with historical perspectives.

The generator utility is a simple Python script that will, with a little guidance, create a controller and a set of views from a basic model, the model, of course, maps onto a database table (in SQL terms).

I don't believe that the generator offers any new approaches and to a large extent it does borrow from the equivalent tools in Grails (grails.org) but as far as I could ascertain it is not a common approach amongst Python web frameworks. And for a variety of reason I do rather prefer the approach of having HTML and simple controllers but that is just my opinion.

I'm posting it here for the primary purpose of finding out if the approach is of interest to others. I wrote the tool to facilitate some developments, commercial and possibly free/open source, that my company is undertaking now and so it will become a part of our development tools but it seems a good idea to make the tool itself generally available – free and open source - and others would then be free to use it as it is and may be developed by us or to take it in different directions to those in which we may go.

And so bearing that in mind, and the fact that this is my first venture into Python please don't bother to make too many comments about the code of the generator – it's very much a work in progress and has lots of oddities, many of which are a legacy of how it was developed – in an Agile manner, or as I prefer it 'Make It Up As You Go Along'. I intend to carry out a large-scale tidying exercise in the near future when once I am satisfied that the general techniques, both within the generator and the code that it creates do work as required. And in the same vein the code that the generator produces may not stand up to the highest scrutiny – but it should work. As I've said the main aim in publishing it now is to discover if the end result is of interest to others

It's called generv, nothing to to with comedies about vintage cars, nor in homage to IEBGENER (just be grateful it wasn't for XPJC) it was simply that it was originally to generate views. And anyway names can be changed...

Generv consists of a single Python script containing a few functions with a very basic user interface and a set of standard templates (for the model and the controller). You first run Generv to create a model file; it uses the standard template and inserts the name that you give it. Then you should complete the model by adding the field definitions in the call to define_table(). When once this is complete you can then run Generv to create the controller and the views and then you will have a simple application that will run as it has been created without any additional coding necessary and offers fairly standard create, amend, list and delete functions for that model. Obviously if you change the fields in the model you can re-run the view generation but unless you amend the 'hasOne' or 'hasMany' options the controller need not be re-generated. Of course the views and controllers can subsequently be changed manually without any further recourse to Generv and in reality I rather view the generator as a method for making the first version of the application and then you use that as the basis for the real, dare I say it, more interesting coding. However I do feel that there is a considerable benefit in using a generator of this sort especially in large applications because it will result in all of the controllers and the views having a standard structure which should simplify any amendments and

maintenance – “seen one, seen 'em all”. The directly generated version may also be of use as a basic administrative tool should that be required.

Of course if you have some objections to the structure that Generv provides then you can change the generator code to produce a different structure to better meet your requirements.

Now, I've already mentioned a few terms that need clarification but first let's go back to basics.

I started to look a Python for some web applications in 2014 after a few years developing in the Groovy/Grails/Griffon area. I was intending to use Grails as the basis of a re-write of an desktop application that we have but for a variety of reason decided to look elsewhere and move away from Java . I was going to run some comparisons with web2py, Django and Turbogears but at the moment have settled with web2py and will develop a small application as a preliminary to our major task shortly. I suppose that knowing how Grails does create a lot of standard code I was surprised at how that was not addressed in the Python frameworks and loh ! Generv was born. I do find that there is so much essential code that is also abyssmally boring to write, and re-write (with minimal changes) that generators look very attractive.

The structure of the generated application starts with Bruno Rocha's

http://www.web2pyslices.com/slice/show/1478/using-modules-in-web2py

where he describes using the /modules directory to hold model class files and so Generv creates model class files in /modules using a superclass which contains a number of utility methods and we'll come back to these later. After creating a new empty model file you then add the field definitions as in a standard web2py define_table() call. But there are also a few new attributes that you can add:

```
hasOne = [...]
hasMany = [...]
isMany = 1
```

hasOne contains a list of other model names with which the current model has a one-to-one relationship and is considered the 'owner' and there must be a field defined whose name is the same as that of the 'owned' model (of type 'integer')
hasMany contains a list of other model names with which the current model has a one-to-many relationship and is considered the 'owner', no field should be defined for any of the owned models.
isMany indicates the the model is on the 'many' side of a one to Many relationship, the identity of the owner(s) is not required because this just enables the generation of the appropriate supporting code which is independent of the owner ('1' is just a 'true' value)

And more similar attributes may be added.

(I should add that the descriptions 'one to one' and 'one to many' are here not used in a pure data analysis meaning and one should look at the examples for clarification)

It is from the models thus defined that Generv can create the controller and the various views with the appropriate components that handle the simple and the one to one and one to many relationships. But before we consider those we need to examine the model class

in a little more detail.

The model class is based on a superclass named 'Smodel' – for supermodel; I suppose that following other naming conventions I could have named it 'Carol' or perhaps 'Kate' or 'Naomi' but I didn't. Initially, taking Bruno Rocha's structure the model has functions which are called by the superclass's __init__:

define_table() - uses db.define_table to define the fields and other attributes
set_validators() - where you would define any validations

and then:

list – returns the results of a query for id>0, i.e. all rows of the table
dbget(r_id)
dbinsert()
dbupdate(r_id)
dbdelete(r_id)

which perform the obvious action given the record id (r_id)

the superclass Smodel also has a number of general purpose functions used by the db<action> functions in particular related to the values of fields passed by web2py's 'request' structure and a new attribute 'vars' is defined which is a dictionary keyed by the field name and holding the current value of that field. When the controller uses 'dbget(record_id) to retrieve a record, dbget automatically populates the 'vars' structure with the values from that row; conversely when issuing a dbinsert(), dbinsert populates the database row from the contents of the 'vars' structure. To achieve these there are two functions:

bindtotable(datadict,...) - where datadict is usually 'request' and populates the model's 'vars' structure with values from the dictionary 'datadict'
toview() - which returns a dictionary suitable to be 'return'ed by a web2py controller action using the key names and values from 'vars'.

These are probably easiest to explain by the use of examples of controller actions:

```
def show():
# shows an individual record
    r_id = request.args(0)
    attrib_instance = Attrib()
    attrib_instance.dbget(r_id)
    vdict = attrib_instance.toview()
    return vdict
```

and

```
def save():
# saves the record created or amended by the user
    attrib_instance = Attrib()
    attrib_instance.bindtotable(attrib_instance.request.post_vars, True)
# validate and if correct save to database
    sid = attrib_instance.dbinsert()
    redirect(URL('show', args=sid))
    return
```

Note that the model is instantiated and then used to access the methods etc. (The superclass __init__ () does set self.request = current.request for convenience)

Having populated the 'vars' structure, after the dbget all of the fields are now available simply by using

attrib_instance.vars['nameofthing']

but notice that 'vars' is just a Python dictionary not a gluon.storage structure; while I could have used the latter I preferred to keep it plain Python in case I wanted to use Generv elsewhere (other frameworks for example).

The controllers are created as very simple scripts with actions:

index

list

select

show

create

save

amend

update

delete

which do the obvious, with the exception of 'select' which is a part of one to one processing and should (and will be) dependent on a flag in the model.

Where the model is the 'owner' in one to one or one to many situations, additional code is added to the relevant actions and when the model is on the 'many' side extra actions are generated but these are discussed later.

I do contend that the views are also quite simple but I suspect that here it does seem to get a little more complicated and it may seem that the justification is just a little dubious but, please bear with me....

I wanted to deal with one to one and one to many In a simple and efficient way and not by, for example in the one to many case, including a lot of code which relate to the 'many' model in the controller and views for the 'owner' model. Also it is important that the user, if it is required, to be able to amend the 'many' table when entering the application from the 'one' side. Supposer you store a number of addresses for a supplier (one supplier with many addresses) but you do not want to have the user use a specific 'address for supplier' application to amend the address of a supplier, you really want the user to be able to call up the supplier application and if they need to amend the addresses of the supplier then just go and deal with that there and then even if it the addresses are in a different database table. And so I decided to handle this with modal popups – now I can see that that may not be to everyone's taste and I am not completely convinced about it myself but in many situations I can see that it will give a satisfactory and logical interface for the user while allowing a sensible compartmentalisation of the code. Indeed if the database design does requires a number of tables to hold, say, a customer's data then it is reasonable to

expect those tables to map well onto the user's view so one might expect a main customer table; a one to many for contact details; and again for orders; and for payments and so on and these are quite logical divisions so for the user to access the 'customer order details' as a modal popup is no great issue, in my opinion.

Having said that I did also wonder about a tabbed page with modal tabs...

But the importance is the adjective: modal; this will force the user to deal with the 'many' table as a separate transaction (as far as the application is concerned) although to the user it should not appear to be separate. With that in mind, Generv creates controllers and views that will give the user a view of any subsidiary one or many records in readonly form on the main view but with a button to allow the amendment of these subsidiary records. It is only when the 'Amend X' button is pressed that the popup appears and within that popup the user has access to the full X application so they can amend or create X records with the proviso that the owner record is, of course, pre-defined and cannot be changed.

Still with me. ?

And so I decided to create the model applications as quasi single-page-applications. Generv creates an index.html that defines for our 'Attrib' model

```
<div id='attribDiv'>
{{include 'attrib/list.html'}}
</div>
```

'list.html' does the real work of displaying the present list of Attribs but all of the actions and buttons that are in the Attrib views use Ajax to repopulate 'attribDiv' (it is defined in 'index.html' only). Then when the Attrib model is on the 'many' side it is in the owner model's various views that Generv will create the popup handling code for the Attribs which will contain a definition of 'attribDiv' which is tied to the popup object. When accessed from the owner application, of course, attrib/index.html is not used but because all of the other Attrib views use Ajax to re-populate 'attribDiv' the Attrib application is kept tightly within the modal popup.

In order to control these effects there does need to be a small amount of Javascript in the views. There are a few ways that this can be done but I've chosen to locate the Javascript in the views to which it relates rather than having a number of separate .js files. This does mean that the Javascript will be reloaded when Ajax re-populates the appropriate ' ' and so the code is written somewhat defensively to avoid multiple definitions of vars and functions etc. but on the other hand the code is close to the parts of the document on which it has its effects so should be easier to maintain.

There are one or two other minor points that do not relate directly to Generv but to the implementation in general. To try to keep the whole exercise more simple I decided to use a separate HTML templating package rather that the web2py facilities and chose HTMLKickstart (www.99lime.com/elements/) it does enough for what I wanted although it does have quite a few features. Secondly I chose jBox (http://stephanwagner.me/jBox/get_started) to provide the modal popup support, this is one of a number of such packages based on Jquery; in fact I did try a few and jBox seemed the best – easiest to get working and it did work well without any need to tinker with it. And, of course, Jquery.

And that's about it really; as it stands it should just produce a working application for each model that you give it. Needless to say there are quite a few restrictions at the moment and I've produced a somewhat shorter more practical document which just summarises the usage, list the restrictions and describes the installation.

One final point; as I've mentioned Generv borrows heavily from Grails' generators (where there are seprate controller and view generators) although as one might imagine there is not one line of Grails code in Generv. But to put this approach in some perspective I should mention that when we started our software development company in 1983, one of our first actions was to buy a program generator that would take a data model and create the data manipulation and reporting programs from it; it was called Sourcewriter and created programs in Microfocus COBOL for use with MS-DOS and CP/M, some of which are still in use today (after a couple of re-writes) and the current work will be the basis for the next replacement. Oh, how software development has changed...