Developer Documentation for the Cartwheel Bioinformatics Toolkit

C. Titus Brown

October 9, 2002

1 Introduction

This is the developer documentation for Cartwheel, a bioinformatics toolkit that supports community analysis (and soon, annotation!) of genomic sequence. This documentation is intended for people with some programming background who want to add features to the toolkit.

The purpose, use, and administration of Cartwheel is described elsewhere; see http://cartwheel.caltech.edu/ for more information.

Cartwheel is written in Python. If you don’t like Python, tough luck; some parts of Cartwheel may be accessible in Java or Perl, but it’s not (yet) a high priority. There are a bunch of reasons for this, and I’d be happy to argue with you about them, but it’s still going to be written in Python at the end of the day ;). Cartwheel also uses three additional technologies, AOLserver, PyWX, and PostgreSQL, which are almost entirely encapsulated by the toolkits (Figure 1).

At its core, Cartwheel uses two outside toolkits: Quixote and cucumber. Quixote is a Web application server framework that (in Cartwheel) is used to serve the Cartwheel Analysis Server (canal) site. cucumber is a simple object-relational mapping between Postgres and (potentially) any interpreted language capable of introspection; at the moment, only a Python implementation exists. Most uses of cucumber are transparent within the code, so you don’t really need to understand it to add simple features. Quixote, however, underlies much of what you see via the canal site, so I will go over it briefly in the beginning.
Figure 1: Software required by Cartwheel
1.1 Where to start

If you are an experienced programmer, with a background in SQL databases, XML, and CGI-style Web programming, I would suggest looking at the source code while skimming this documentation; there’s not too much point in reading this document thoroughly unless you want to understand the rationale behind some of the choices I’ve made, or the basics of the toolkits upon which Cartwheel is based.

If you’re a less experienced programmer, then I’m sorry to say that you’re not going to learn much about CGI-style Web programming, XML, or databases from this document: there are several abstraction layers (implemented by the toolkits mentioned above) that encapsulate these essentially low-level issues. On the other hand, this document should help you understand how things work, which will be a good start for diving into the source code and starting to make modifications.

Either way, the best place to start is to set up your own Cartwheel development directory, install the data model, and run through Appendix A on cucumber. Then return here and go straight through ’til the end.

Now, to get started, let’s look at the directory layout!

1.2 Installation and the directory layout

You should go acquire the Cartwheel source code somehow. The best way is to go retrieve the “ctbnew” revision from the CVS archive on SourceForge; this will involve going to the Cartwheel project on SourceForge and following the instructions for anonymous checkout of the project source code. Cartwheel is entirely contained within the cartwheel/ subdirectory of the CVS archive, so that’s all you need.

Now you need to install Cartwheel. Cartwheel runs directly out of its source directory, which is a major convenience for developers! However, you do need to load the data model in to Postgres and do a bunch of other things, so go follow the instructions in INSTALL.txt; there are specific tags in this file for developers to follow, and you should do these steps and not the default installation steps.

Now, assuming you have the full package within the cartwheel/ directory, change into that directory and take a look around. The main thing to look at is the subdirectories; you should have several. Let’s start with the two simple ones: bin/ and data-model/.
bin/ contains a bunch of command-line programs that let you run the batchqueue and do various analyses; the main documentation for the stuff in here is in the Admin Docs, which inconveniently haven’t been written yet. We’ll cover anything you need to know for development in this manual, though. For now, ignore this directory.

data-model/ contains the basic SQL data model that you have already loaded into Postgres. This is the very heart and soul of Cartwheel, and you’ll be spending a lot of time in here. Note that because of cucumber we make extensive use of Postgres’ object-oriented features; you can attribute any non-standard SQL statements in those files to this! But for the moment, ignore this directory, too.

Let’s check out one more relatively uninteresting subdirectory, before moving on to more complicated issues: test/. This contains a bunch of tests that check to make sure that the installation is good: in particular, they check that the various required analysis programs are where you said they were in the configuration, that the developers didn’t screw up in any really obvious places, and that the data model is loaded. We’ll come back here when we talk about adding analyses; it’s your job to make sure that any new analyses you add are also added into the testing framework, so that you (and I) can tell with a single command whether or not there’s something obvious wrong.

Now let’s look at some more interesting directories: lib/ and website/. Both of these are complicated, and rightly so: they contain most of the behavior of Cartwheel!

The lib/ directory is a particularly complicated one, since it contains lots and lots of subfiles and subdirectories. The programs/, formats/, and cartwheel/ subdirectories are the important ones. programs/ and formats/ can be treated as black boxes. programs/ provides utilities to handle all actual program execution – set-up and tear-down of working directories, timing of analyses, encapsulation of results, and the translation of options to command-line flags. formats/ provides utilities to deal with the variety of output formats spewed by the common analysis programs.

The lib/cartwheel/ directory contains all of the actual cucumber objects used in Cartwheel (which are in turn based partly on the SQL in the data-model/ directory). Why this separation? cucumber does a pretty good job of hiding SQL from you: it loads attributes and linked objects automatically, it allows you to construct lists of objects matching simple constraints, and, when set up properly, can completely encapsulate almost all of the
generic queries you need to do. The price to pay for this is that the cucumber class files are necessary at runtime, unlike the SQL files, which are used only when creating the database. So the class files, and the functions that enable them to do their work, must be in an easily accessible place. In this case, we’ve put them in a Python package, lib/cartwheel/, which is separate from the data model. By the way, just to confuse you, we’re going to call the lib/cartwheel/ directory the Cartwheel Library Package from now on.

The Cartwheel Library Package can be imported from within Python by issuing an import cartwheel command, just like any other Python package. The run-python command in the top-level bin/ directory automatically does this, and also creates an object manager to boot. More about this convenience later.

The last and final directory to look at is website/, which contains the code supporting the canal Web site. Underneath the website/ directory, there are several subdirectories. The canal/ directory contains almost all of the canal source code (it doesn’t contain the Web site objects, which are in the Cartwheel Library Package under cartwheel.website); the other two directories website/cgi/ and website/pywx/ contain Apache-style CGI access code and AOLserver/PyWX access code for canal. You can ignore both of these latter directories, because you shouldn’t have to ever change anything in them. website/canal/ is the important one, and what I’ll go over next is how Quixote serves a Web site from this package.

2 Quixote and canal

Quixote is a nifty Web application framework that takes advantage of the introspection features of Python to dynamically serve Web sites; in Cartwheel, it is used to serve canal. In addition to its basic function (serving Web pages!), Quixote has a neat “widgets” framework that can dramatically simplify making forms, and there’s a templating system contained within it as well. The complicated part is in how Quixote serves Web pages, however, so it’s what we’ll go over; the rest can be learned “on the run”.

Let’s start with an example. Load canal/pages.ptl and hunt down the function show_login_form. This is what shows the login form that you see when you first connect to a canal site. It’s called by the specially named function _q_index, which is what Quixote uses to serve URLs ending in ‘/’. 
There are a couple of interesting things to note.

First off, unlike normal Python functions returning content via CGI, there’s no “return” call returning the page. So where does the text come from? Well, because this is declared as a “template”, rather than as a function, and it’s in a .ptl file, Quixote automatically constructs the return value from the strings that make up the body – including strings returned from function calls. Were this a regular Python function defined by def, you would have had to concatenate all those strings together into a single string and then returned that string. (This is done elsewhere in canal, in places where template functions can’t be used, such as class definition files.)

Note also that unlike standard CGI programming there’s no Content-type: declaration; by default, Quixote assumes you’re serving “text/html”. This can be changed if needed, but it’s a good default.

Now, what about all those render(request) calls? They are telling Quixote form widgets constructed in make_login_widgets to render themselves as actual HTML. If you take a look in make_login_widgets you can see that it’s constructing a dictionary full of these different widgets; the dictionary is a container that makes it easy to find the widgets. In the function process_login_form you can see the counterpart to rendering widgets, which is parse(request). This returns the actual value (if any) typed into the form widget on submission. Because Quixote’s widgets make this so easy, this:

```python
    d = make_form_dict()
    if request.form:
        process_form_dict(d)
    else:
        print_form_dict(d)
```

is a common code flow in canal.

There are two basic types of objects that Quixote serves: namespaces and functions. Namespaces are things like packages, modules, and objects; functions are plain old Python functions. Namespaces are what you see in the URL line between the base canal URL and the last thing in the URL; for example, in the URL /canal/lab/folder/1/hide, lab/, folder/, and 1/ all represent namespaces, and hide is a function. Functions, of course, are the only things that actually return content to the Web browser, but they can be located deep within nested namespaces, as in the example just given. next, add:


- an explicit example

2.1 canal organization

The organization of most of canal is pretty simple: it’s just a series of top-level directories that deal with serving views of various objects, like labs (lab/), folders (folder/), analysis groups (group/), and sequence lists (sequence/). The one really nasty area – an area into which we will not venture until much later! – is the analyses/ subdirectory, which is responsible for serving views of analyses. It’s pretty complicated stuff, and you only really need to understand it if you’re adding an actual analysis type to the system (see Section 4).

The remaining directories, admin/ and ds/, serve the administrator interface and the data to FamilyRelations, respectively.

2.2 Serving the canal Web site

So, now that you’ve seen a bit of how Quixote works, and the organization of canal, the rest is pretty simple: the user starts at the base URL (canal/), logs in, and is then presented with a lab view. After this, they make their way into whatever part of the canal hierarchy they need to use. In order to find what part of the code you need to modify to add a particular feature, you can usually just look at the URL above the place where you want to add that behavior, find the last namespace in the URL that corresponds to a top-level directory, and follow the URL down into the directory hierarchy.

Note that this is slightly atypical for a Quixote application: usually you simply follow the directories down into the namespaces, without looking for a top-level match after you’ve gone down one level. For organizational reasons, I’ve designated lab/, folder/, and group/ as top-level namespaces, even though folder/ is only reachable from within lab/, and group/ from within folder/.

2.3 canal objects

I’ve said that most of canal is dedicated to serving various types of objects. What do I mean by objects? Standard Python objects, of course – but these persist between visits to the Web site. Where are they stored, and where are they defined? This is the part of Quixote that interfaces with cucumber:
the objects are implicitly linked into the database by cucumber, and their classes are defined in the Cartwheel Library Package under the website/ directory (i.e. in lib/cartwheel/website/). These are the subject of the next section.

3 The Cartwheel Object Model

Cartwheel consists of two distinct (but connected) object models: the batchqueue objects, and the Web site objects. These objects are the core of Cartwheel, and both restrict and enable the behavior users see. They also provide an extensible and modifiable framework within which the developer must work. Both object models are described below.

3.1 The Batchqueue objects

The purpose of the batchqueue system is to provide a batching and queueing interface for running compute-intensive applications, which are often (or mostly!) developed independently. There are several problems with running compute-intensive and/or 3rd-party software in tandem with a user interface such as a Web server:

- First, the mechanisms by which jobs are requested militate a different type of system from that which processes the jobs: Web servers are generally not particularly beefy, are often connected to the outside world, and often have multiple services running on them. Compute servers are frequently not accessible from the outside world, are often powerful machines with lots of RAM, and usually are dedicated to one task.

- Second, because Web servers generally do very little computation but must be able to handle load spikes, they are often threaded, which is incompatible with running 3rd-party closed-source programs, because this requires fork() calls; threads and fork()ing interact unpredictably.

- Third, when running 3rd-party software, one often needs to be able to flexibly deal with passing in parameters and parsing a variety of different input and output formats.
The batchqueue system deals with these issues by encapsulating external programs in a Python interface and coordinating the execution of jobs through a PostgreSQL database (mediated by cucumber).

3.1.1 Batchqueue concepts

The batchqueue system itself is quite simple: it’s meant to be easily understandable and easily extensible, as well as reusable in other contexts. The flow of information through the system works as shown in Figure 2: through one of several possible interfaces, either local (a Python or Perl API, for example) or remote (SOAP or another RPC mechanism) a program deposits a job request into the database. Compute backends poll the database for new jobs; when a job is available, they mark it as ‘taken’ and process it, perhaps invoking one or more 3rd party programs to do so. The job results are then saved back into the database, at which point they can be exported through any of several different APIs in any of several different formats.

The simplicity of the system means that all of this works in somewhat under a thousand lines of code, and most of the logic is encapsulated in about 20 lines of code. In trade for this simplicity, however, comes the lack of certain capabilities: for example, the system does not, by itself, allow multiple compute nodes to work on a single problem. One can easily take advantage of other systems (such as PVM) and encapsulate several compute nodes (say, a Beowulf) in the job processor, but this is not implemented by the batchqueue system itself. There is also no complex scheduling: compute nodes can take jobs at any rate they wish, but (by default) they simply monitor the queue and take jobs whenever they are available.

On the flip side, the simplicity does make it extremely easy to re-use in different contexts. At the Transcription Factor Research Center at Caltech, we have a Beowulf that services several different queues, including a BLAST queue for the Sea Urchin Genome Project and a canal server.

3.1.2 Concrete description: the objects themselves

There are only two core types of objects in the batchqueue system: BatchqueueRequest and BatchqueueResult. (The class files for these types and their descendants are located in lib/cartwheel/batchqueue/.) BatchqueueRequest objects encapsulate the necessary information for a request, while BatchqueueResult objects encapsulate the results of requests.
Figure 2: Batchqueue information flow
All requests must have several pieces of information: an owner object ID, such as a Web site user (to help track requests and store information); a priority, which is a simple scalar quantity that can be used by request processors however they wish; and three boolean flags, available, taken, and completed, which respectively inform request processors of the job’s availability for processing, whether or not the job is being processed right now, and whether or not the job has been completed. Manipulation of the latter two flags is done by the batchqueue system automatically, so all that the programmer has to do is worry about making the job available.

Result objects require even less information than requests: they contain a field indicating the amount of time used in processing the job, and an ID for the request for which they are a result. The various subclasses have members that contain the specific results, of course.

To see this all in action, take a look at test/test-queue.py. This script enters a simple test request (BatchqueueTestRequest), which sleeps for some number of seconds and then returns a BatchqueueTestResult object indicating completion.

Of course, in most cases one will want to do more than simply sleep for a number of seconds, which is where a good deal of the support infrastructure that’s present in the batchqueue subdirectory of the Cartwheel Library Package comes in handy. A good case in point is the BlastRequest object, which takes in a pair of data types to BLAST against each other. The 'query' object can be a sequence or sequence list, while the 'against' object can be a sequence, sequence list, or BLAST database; these query objects are then grokked and passed into BLAST via the programs package (under lib/programs/). Once the BLAST is done, a CommandLineResult is returned. You can look at the test/test-blast-* files to see how the various types of BLAST are set up.

Now that you have an idea of how the batchqueue system works, let’s move on to the Web site objects.

3.2 The Web site objects

The Web site (currently only canal, although more Web interfaces may be added later) is also fairly simple. Most of the objects are container objects, either for other Web site objects (labs contain folders, and folders contain folders and analysis groups) or for batchqueue objects (analysis groups contain batchqueue analyses); see Figure 3 for the relationships between
these objects. The few exceptions are extensions to batchqueue objects that are Web-site-specific: for example, FolderSequences are BatchqueueSequences that contain information about what folder the sequence is in, and UploadedAnalysisData represents data uploaded by the user to the Web site.

The figure shows that there are three basic types of container objects: Labs, Folders, and AnalysisGroups. Labs are just convenient ways to logically group top-level folders and users; they consist of just a name. Folders are a little bit more interesting: they come in two flavors, LeafFolders and NodeFolders, which respectively contain actual projects (AnalysisGroups and FolderSequences) and other folders (both LeafFolders and NodeFolders).

AnalysisGroups are actual groups of analyses – that is, BatchqueueRequests – that are performed on one or more sequences, represented by FolderSequences. There are currently four main types of AnalysisGroups, although I expect to add more over time; these are: SingleAnalysisGroups, which contain analyses of a single sequence; objects of a descendant class, SugarAnalysisGroup, which contain a stereotyped set of analyses that we find useful for analyzing Sea Urchin sequences; PairAnalysisGroups, which contain pairwise analyses of two sequences; and TripleAnalysisGroups, which contain three-way analyses of a set of sequences. Each AnalysisGroup contains information about which sequence or set of sequences is analyzed by all analyses within that group, so every analysis in e.g. a PairAnalysisGroup deals with the same two sequences.

The majority of the Web site, then, consists of interface code to deal with creating and deleting labs, folders, and analysis groups; setting up analyses; uploading sequences; and so on. In the abstract, this code is both pretty boring and very readable; and since the code for each Web page is largely isolated from the code for other Web pages, each page can be understood separately. The most complex area is that which deals with setting up specific analyses, so we’ll focus on that part in the remainder of this section.

### 3.2.1 Setting up analyses

The process of setting up individual analyses as part of analysis groups is diagrammed in Figure 4. The user starts by creating an analysis group of some type (during which the sequence or sequences with which that analysis group will be dealing are specified). She then edits the analysis group and adds one or more analyses by hand, changing parameters from the default settings if necessary. Finally, each analysis is entered into the queue for
Basic canal container object organization
(Labs, Folders, and Analysis Groups)

23.5.2002

Figure 3: Web site container objects
What makes this a nasty mess in the source code is that, of necessity, every type of analysis has its own set of parameters, and thus requires its own user interface to set those parameters. Moreover, the analysis groups usually specify some of the information for the analyses automatically, but differently for each type of analysis group - for example, a BLAST analysis in a PairAnalysisGroup will always want to BLAST its two sequences against each other, while a SingleAnalysisGroup will want to set the 'query' sequence but not the 'against' sequence. However, we don’t want to set up a different editing interface for each analysis depending on whether or not it’s in a SingleAnalysisGroup or a PairAnalysisGroup, because that’s ugly and not particularly extensible.

The way that I have dealt with this is to set up two different types of API for each analysis – a creation interface, and an editing interface. The creation interface is fairly programmable, and is called by each analysis group type in a different way. The editing interface is generic, and allows the user to set all of the parameters on each analysis that are mutable, e.g. threshold for BLAST. Looking at Figure 4, then, the “add analysis” box uses the creation interface, and the “edit analysis” and “submit analysis to queue” boxes use
the editing interface. This is all discussed in more detail below, in Section 4 on extending Cartwheel; for now, you should avoid this area of the source code!

4 Adding an Analysis to Cartwheel and canal

A Using cucumber

cucumber is a fairly generic toolkit, but because Cartwheel depends heavily on its functionality, it is worth looking at some explicit examples of cucumber use.

Fundamentally, cucumber is all about persistent objects. In conjunction with the class files mentioned elsewhere, it handles loading, saving, and creation of objects in a fairly transparent manner.

Below, I’ll give you some examples of its use, and discuss a little of what’s happening “behind the scenes”. First, we’ll create a list of all the objects of a particular type that are in the database; then, we’ll talk about pulling out a specific object by id, and finally, we’ll go over the ObjectManager class.

A.1 Creating a catalog of Labs

In the Cartwheel root directory, execute python -i bin/run-python. You should get a Python interpreter command prompt. Now type:

catalog = m.createCatalog('cartwheel.website.Lab')

for lab in catalog:
    print lab.id, lab.long_name

You should see a list of all the labs in your installation, preceded by a number.

Let’s go over what happened. A “catalog” is a list of objects pulled from the database by a query – in this case, a query that asked for all labs, with no restrictions. If we had written

catalog = m.createCatalog('cartwheel.website.Lab', (‘short\_name=’test’ ’,))
we would have pulled out a list of all labs with a short_name of ‘test’. The restrictions are specified in the notation of a WHERE clause in SQL, which is a whole ‘nother language; reading the source code is your best bet to understand the simplest aspects of this language, for now.

So, once we had a catalog, we could use it as just another list, and iterate over it, pulling out each object individually and printing various attributes. This is very useful in situations like folder display, where we’re going over the list of sub folders and printing out their name and other attributes. But what if we don’t want a list, but rather a single object?

A.2 Pulling out an individual object

After entering the run-python program (as above), pick some number (one of the lab IDs printed out above will work) and type (replacing N with the number you picked):

```python
obj = m.load(N)
print obj
```

This loads an individual object by ID. The ID is a unique integer value larger than 0; each ID belongs to at most one object. Until an object is deleted from the database, you can always retrieve it with its ID.

If you enter the following line (after loading the object):

```python
print obj.values.keys()
```

you'll see all of the different fields on this object that you can access. Most of them are specified in the various class definitions contained in the files underneath the Cartwheel Library Package; the remaining ones (such as 'klass') are part of the cucumber package.

**cucumber** doesn’t just allow you to retrieve objects; you can create them and change them, too.

A.3 Creating and modifying cucumber objects

(creating!)

Now, to modify an object, go back and choose one of the labs (this can either be the one you created, or the one you loaded earlier, or any other Lab object).

Replacing ’N’ with the object ID, type:

```python
16
obj = m.load(N)
print obj.long_name
obj.long_name = 'Hello'
print obj.long_name
m.commit()

This loads the object, changes its long name to 'Hello', and saves the changes. **Until you do the commit**, the changes are not saved, and will not be seen by any other object manager! Why? To explain this, let’s talk about **transactions**.

### A.4 The ObjectManager in more detail: rollback and transactions

First, an example. Continuing in the same session, do:

```python
obj.long_name = 'Hello2'
print obj.long_name
m.rollback()
obj = m.load(N)
print obj.long_name
```

In the first example, you loaded an object; changed its `long_name`; and committed the change to the database. In the second example, you changed an object’s name, discarded the change, reloaded the object, and showed that its `long_name` remained as it was in the beginning.

The **rollback** command is the complement to **commit**: it discards all of the changes made since the last commit (or, more usually, since you got the object manager). Together, **rollback** and **commit** deal with **transactions**, which ensure that each object manager has a consistent view of the data. For example, when the batchqueue process is done with a job, it has to do two things: update the request `completed` flag, and insert the results of the request into the database. If the Web site looks at the database on a different machine and sees the first change – the update – but not the second, it may try to get the results before they are present.

In order to prevent this sort of inconsistent view, each object manager has a distinct view of the data present in the database. This view contains all of the changes made by that object manager since its last commit or rollback,
and no other changes made by other object managers are visible until a
commit or a rollback is executed. Technically, this sort of transaction system
helps ensure what is known as ACID compliance: changes are “atomic”,
“consistent”, “isolated”, and “durable”.

By and large, developers don’t need to concern themselves with transac-
tions too much (which is really the point! – they would if the transactions
weren’t there). The important thing to remember is that unless a commit is
executed, changes aren’t saved – because Cartwheel, like most database ap-
lications, will automatically execute rollbacks in the case of an unhandled
error.

A.5 What the heck? Some oddities of PostgreSQL.

So far, everything you’ve seen object managers do is garden-variety SQL
database stuff; most modern “industrial” databases (Oracle, SQLserver, Post-
greSQL, and even MySQL) have transactions, in particular. What they don’t
have – that is, what only PostgreSQL has – is TABLE inheritance. To see
what this means, consider the following code:

```python
singles = m.createCatalog('cartwheel.website.SingleAnalysisGroup', ('visible',))
single = singles[0] # pick a single group

pairs = m.createCatalog('cartwheel.website.PairAnalysisGroup', ('visible',))
pair = pairs[0] # pick a pair group

# get *all* groups
groups = m.createCatalog('cartwheel.website.AnalysisGroup', ('visible',))

assert single in groups # show that the 'single' group is in groups
assert pair in groups # show that the 'pair' group is in groups
```

This is pretty straightforward for people used to object-oriented program-
ing. The idea that a list of all analysis groups will contain all specialized
analysis groups is common-sense. However, this is not normally the way most
SQL databases work, and that PostgreSQL does is the primary feature that
makes it well-suited for a cucumber-style approach. Essentially, if tables B
and C inherit from table A, then table A contains all records in tables B and
C. However, to recover the full information available for a record in table B,
one would have to load the record from table B directly and not from table A. This is all taken care of by **cucumber**, but I mention it here for people who have not encountered this deviation from normal SQL.

### A.6 SQLisms: getting a cursor

**cucumber** does all its work through a DB API 2.0 compliant database handle (see [http://www.python.org/](http://www.python.org/) for more information). You can get a cursor handle for the database by calling the `cursor()` method on the object manager:

```python
c = m.cursor()
c.execute('SELECT id, long_name FROM labs')
for (id,name,) in c.fetchall():
    print id, name
```

(This example should give the same results as the first example.)

Something to underscore is that there is no magic in **cucumber**: all of its work is done through standard SQL, and all of the tables and results are available through standard SQL. The only thing non-standard about it is the TABLE inheritance system that is part of PostgreSQL, which is what lets **cucumber** work easily.