Giotto Documentation

Release 0.11.0

Chris Priest

October 13, 2016

Contents

Giotto on pypi: http://pypi.python.org/pypi/giotto/

Giotto on Github: https://github.com/priestc/giotto

Giotto pycon presentation: http://www.slideshare.net/priestc/visualizing-mvc-and-an-introduction-to-giotto

Contents:

CHAPTER 1

Introduction

Giotto is a framework for building applications in a functional style. It is based on the concept of Model, View and Controllers.

The framework is designed to enforce a clean style that results in code that is maintainable over a long period. Other popular web frameworks are built with a mindset of launching fast. This results in code that will deploy quickly, but falls under it's own complexity after many iterations.

1.1 Invocation cycle

The invocation cycle is as folows:

1. The controller process is started. An example of a controller process is Apache, or gunicorn. A manifest is given to the controller process when it is started. All incoming requests to the controller process will be routed to a program contained within the manifest. A manifest is just a collection of programs.
2. A user makes a request to the controller process. This can be a web request, or a command line invocation, or any other action that is handled by a controller process.
3. The controller packages up the request into a `request` object. In the image above, the invocation being depicted is basically a user saying, "Give me the contents of blog # 3 in an html document"
4. That request in inspected by the controller. The appropriate program is determined from the manifest based on attributes of the request. In the image above, the path of the request (`/blog`) determines the program. A program is a collection of a model, a view, a cache expire time, a set of input middleware classes, and a set of output middleware classes.
5. Once the program has been determined, the request object is routed through the input middleware. The middleware objects are functions that take in a request object, and return a request object.
6. After each input middleware class associated with the program has been ran, the controller sends off data to the model, and the model is executed. In the example above, the model is a function that retrieves blog data from a database. The data it requires is the ID of the blog. In this case, the ID is 3.
7. When the model is done, it returns its data, and it gets passed in directly to the view. At the same time, the controller sends the mimetype of the invocation to the view as well. In the example, the mimetype is `text/html`.
8. The view renders the blog data into an HTML document. The data is then passed onto the controller, but before that, it is stored in the cache. How long the data will stay in the cache is an attribute of the program.
9. The controller takes this html document and packages it up into a response object.

10. This response object is passed on through the output middleware classes. Such classes are very similar to input middleware classes, except they take in a response, and return a response.
11. That response gets returned to the user.
12. The next time a user requests blog #3 in HTML, the invocation will be the same, except for one difference. Instead of executing the model and the view, the controller will grab the HTML document from the cache.

CHAPTER 2

Getting Started with Giotto

First, install giotto:

```
$ pip install giotto==0.11.0
```

Note: Giotto is very actively under development, The version on pypi is most definitely stale. Instead of the above command, you can optionally install the latest version from github:

```
$ pip install git+git://github.com/priestc/giotto.git
```

Now create a new directory:

```
$ mkdir demo
```

and inside that directory, run this command:

```
$ cd demo
$ giotto create http cmd --demo
```

This will create a `manifest.py` file, which contains your program manifest. It will also create a series of "concrete controller files", which will act as a gateway between your application and the outside world. The concrete controller files will be called `http_controller.py` and `cmd_controller.py` This utility will also add a `config.py`, `secrets.py` and `machine.py` file, which will be where you add database connection information (and other things).

If you only want to interact with you application through the command line, then you could leave off the `http` flag when calling `giotto` (and vice versa). The option `--demo` tells giotto to include a simple "multiply" program to demonstrate how giotto works.

Inside the `manifest.py` file, you will see the following:

```
class ColoredMultiplyView(BasicView):
    @renders('text/plain')
    def plaintext(self, result):
        return "{{ obj.x }} * {{ obj.y }} == {{ obj.product }}"

    @renders('text/html')
    def html(self, result):
            return """<!DOCTYPE html>
            <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.4/jquery.min.js"></script>
            <html>
                <body>
                    <span style="color: blue">%(x)s * %(y)s</span> ==
```

```
                    <span style="color: red">%(product)s</span>
                </body>
            </html>""" % result

    @renders('text/x-cmd')
    def cmd(self, result):
        from colorama import init, Fore
        init()
        return "{blue}{x} * {y}{reset} == {red}{product}{reset}".format(
            blue=Fore.BLUE,
            red=Fore.RED,
            reset=Fore.RESET,
            x=result['x'],
            y=result['y'],
            product=result['product'],
        )

    @renders('text/x-irc')
    def irc(self, result):
        return "{blue}{x} * {y}{reset} == {red}{product}{reset}".format(
            blue="\x0302",
            red="\x0304",
            reset="\x03",
            x=result['x'],
            y=result['y'],
            product=result['product'],
        )


def multiply(x, y):
    return {'x': int(x), 'y': int(y), 'product': int(x) * int(y)}

manifest = Manifest({
    'multiply': Program(
        controllers = ('http-get', 'cmd', 'irc'),
        model=[multiply, {'x': 3, 'y': 3, 'product': 9}],
        view=ColoredMultiplyView
    )
})
```

All Giotto applications are made up a collection of Giotto Programs. Each program class defines a model, a view, and a set of controllers.

A "Giotto application" is the overall project, such as a blog application, a content management system, or a twitter clone. A "Giotto Program" is a "page" within an application. An example of a program is "create blog", "view tweet" or "register user".

A Giotto program is made up of (at minimum) a model, a view, and a set of controllers. In the above example, our application only contains one program called "mutiply". All it does is take two numbers, and multiply them together.

To see our example `multiply` program in action, start up the development server by running the following command:

```
$ giotto http --run
```

This will run the development server (you must have werkzeug installed). Point your browser to: http://localhost:5000/multiply?x=4&y=8

The browser should now be displaying *4 * 8 == 32*. With the part before the == in blue, and the part after in red.

The following order of events are occurring:

1. You make a web request to the development server that is hooked up to our demo application, with the help of Giotto.
2. HTTP request is received by Giotto.
3. Giotto inspects the request and dispatches the request off to the `Multiply` program. Giotto knows to dispatch the request to the Multiply program because:
   (a) The program is configured to use the 'http-get' controller, and this is a HTTP GET request.
   (b) The url matches the `name` attribute on the program class.
4. Calls the model with the arguments from the GET vars.
5. Takes the output from the model and passes it into the view object.
6. Calls the appropriate rendering method on the view class, depending on (in this case) the `Accept` headers.

Now, open up your browser's javascript console (firebug if you're a firefox user). Type in the following:

```
$.ajax(window.location.href).done(function(r) {console.log(r)})
```

You should see a json representation of the page. The HTTP controller automatically changes the return mimetype to "application/json" when the request comes from ajax.

Lets take a look at this program as viewed from the command line. Press *ctrl+c* to stop the dev server.

Form the shell, run the following command:

```
$ giotto cmd multiply x=4 y=8
```

The output should be exactly the same. It should say *4 * 8 == 32* with the *32* in red and the *4 * 8* in blue.

The model that is being called here is exactly the same as we saw being called from the browser. The only difference is the way the result is visualized, and the data moves between the user and the computer through the command lone, instead of a browser..

2.1 Using Mocks

On the Program object, add a `model_mock` object to the list along with the model. A model mock is an object that gets returned in lieu of executing the model function. This object should be the same form as what the model returns:

```
manifest = Manifest({
    'multiply': Program(
        controllers=('http-get', 'cmd', 'irc'),
        model=[multiply, {'x': 10, 'y': 10, 'product': 100}],
        view=ColoredMultiplyView,
    )
})
```

When you run the dev server include the `--model-mock` flag:

```
$ giotto http --run --model-mock
```

Now no matter what arguments you place in the url, the output will always be `10 * 10 == 100`. If your model makes calls to the database or third party service, the moel mock option will bypass all of that. This feature is useful for front end designers who do not need to run the full model stack in order to create HTML templates. This feature is also sometimes called "generic models".

2.2 Cache

Add a `cache` attribute to the program:

```
manifest = Manifest({
    'multiply': Program(
        controllers = ('http-get', 'cmd', 'irc'),
        model=[multiply, {'x': 10, 'y': 10, 'product': 100}],
        cache=3600,
        view=ColoredMultiplyView
    )
})
```

Restart the cache server (this time leave off the `--model-mock` flag). Also, add a pause to the model method:

```
def multiply(x, y):
    import time; time.sleep(5)
    return {'x': int(x), 'y': int(y), 'product': int(x) * int(y)}
```

This will simulate a heavy calculating model. You also need to have either Redis or Memcache installed and running. Configure the cache by setting the following to the `cache` variable in the `machine.py` file:

```
cache_engine = 'memcache'
cache_host = 'localhost'
```

To use the redis cache, change the engine to `redis`. Now when you load a page, it will take 5 seconds for the first render, and subsequent renders will be served from cache.

2.3 Now You're Done!

Thats it in a nutshell. To learn more, read around the docs, and build things!

CHAPTER 3

Manifests

Manifests are objects that keep track of all the programs that make up your application. Think of them as urlconfs in django. An example Manifests looks like this:

```
from giotto.programs import Manifest
from myapp.models import homepage, signup, auth_manifest

manifest = Manifest({
    '': RootProgram(
        model=[homepage],
        view=BasicView,
    ),
    'auth': auth_manifest,
    'static': StaticServe('/var/www/static'),
    'privacy_policy': SingleStaticServe('/var/www/privacy.html'),
    'signup': [
        SignupProgram(
            view=JingaTemplateView('signup.html')
        ),
        Program(
            controllers=['http-post'],
            model=[signup]
            view=Redirection('/'),
        ),
    ]
}
```

When this application is served to the outside world through a controller, this manifest tells the controller which programs are to be available.

Manifests must contain only strings in the keys, and only `Program` instances or lists of `Program` instances as the values. The strings may contain any characters except for periods (`.`).

If you want to restrict a program to a certain controller, include the controller in the `controllers` argument of the program:

```
{
    'signup': [
        Program(
            view=JingaTemplateView('signup.html')
        ),
        Program(
            controllers=['http-post'],
            model=[signup]
            view=Redirection('/'),
```

```
        ),
    ]
}
```

In the above example, all invocations except for HTTP POST requests will go to the first program.

All concrete controllers look for a project manifest object named `manifest` in the file named `manifest.py`.

CHAPTER 4

Terminology

- **Contrete Controller** - A file that launches a *controller process*. This file contains configuration information, such as hostname, port to listen to. The result of running the `giotto_project` utility is the creation of one or more concrete controller files.
- **Controller Class** - A type of controller, such as IRC, HTTP and CMD. Giotto ships with three controller classes: IRC, HTTP and CMD.
- **Controller Process** - A process that listens for user input, and invokes a giotto program when the user gives it data.

CHAPTER 5

Built-in Controller Classes

Giotto ships with three controllers classes.

5.1 HTTP

This controller class is used to allow program invocations via the HTTP protocol. To run your Giotto application through HTTP, you first must create a concrete controller file for this controller class. Run this command from your project's root folder:

```
$ giotto new_controller http
```

This will add a file into your controllers folder named `http_controller.py`. To run the development server, run the following command:

```
$ giotto http --run
```

To change the port and hostname of the development server, edit the call to *serve* in the concrete controller file.

All requests are rendered with the `text/html` mimetype unless the accept headers are set otherwise. Also, for any request that comes in through an ajax request, the controller will attempt to render that model with the `application/json` mimetype.

5.2 Command Line

This controller class is used to allow program invocations via the command line. To run your Giotto application through the command line, you first must create a concrete controller file for this controller class. Run this command from your project's root folder:

```
$ giotto new_controller cmd
```

This will add a file into your controllers folder named `cmd_controller.py`. To invoke your application, run this generated script (called a concrete controller) like so:

```
$ giotto cmd path/to/myprogram --var1=foo --var2=bar
```

This will call the `myprogram` program with `var1` and `var2` as arguments.

All models are rendered by the views with the `text/cmd` mimetype.

5.3 IRC

This controller class is used to allow program invocations via an IRC server (either through a channel, or private message). To run your Giotto application through IRC, you first must create a concrete controller file for this controller class. Run this command from your project's root folder:

```
$ giotto new_controller irc
```

This will add a file into your controllers folder named `irc_controller.py`. Edit the generated file and add the username of the bot, the hostname of th server you want to connect to, and any other details you want. To invoke your application, run this generated script (called a concrete controller) like so:

```
$ giotto irc
```

Once the bot has been connected to the server, to invoke programs, enter a channel and type the following:

```
!giotto path/to/myprogram
```

This will call the `myprogram` program and return the output via the same channel. The bot will prefix the message with the username of the user who invoked it. the `!giotto` part is called the "magic token" and can be configured in the concrete controller file.

You can also invoke the bot through private message:

```
/msg giotto-bot path/to/myprogram
```

and the bot will respond with a private message. In this example, the bot is named "giotto-bot", but this can be configured through the concrete controller file.

All models are rendered by the views with the `text/irc` mimetype.

5.4 Overriding default mimetypes

Whenever you invoke a program, the mimetype used to render the model data is determined by that controller's default mimetype. The default mimetypes for the HTTP, IRC and CMD controllers are `text/html`, `text/irc`, and `text/cmd` respectively. To override this, just add the extension of your preferred type to the end of the program name:

```
$ giotto cmd path/to/program.json --x=4
```

This will return the result of `program` in JSON format instead of the default `text/cmd`. This also works for positional arguments:

```
!giotto path/to/myprogram.txt/argument
```

CHAPTER 6

Creating your own controllers

Custom controllers are implemented as a subclass of `giotto.controllers.GiottoController`. A custom controller can be used to run a giotto application through a new mode of invocation.

For instance, you could write a controller that monitors an email inbox. The controller would take in any new message, parse the message, and reply to with a new message.

All custom controller classes must implement the following methods:

`get_raw_data``()`
: This function returns the "payload data" of the incoming request. The HTTP controller returns the GET or POST parameters. A dictionary should be returned.

`get_invocation``()`
: The function should return the name of the invocation, e.g. "path/to/program.txt" The HTTP controller returns the value of `request.path`

`mimetype_override``()`
: All responses from a controller are rendered with the mimetype that is set in the property `default_mimetype` unless this function returns a different one. For instance, all invocations from HTTP are rendered as text/html, with the exception of when a request is made with alternate `Accept` headers. If this function returns `None`, then the value in `default_mimetype` is used.

`get_controller_name``()`
: This should return the name of the controller.

`get_concrete_response``()`
: This function is used to return the response object that is specific to the controller context. For instance, the HTTP controller returns a werkzeug `Response` object. This function should call `get_data_response()`, which returns the output of the view. Keep in mind, `get_data_response()` can throw an error, so `get_concrete_response()` must try to catch these errors and expose them properly through the controller.

`get_primitive`(*primitive*)
: Primitives are the abstracted interface between the controller and the model. This function should extract the proper data from the `raw_data` dictionary (see above). Not all primitives need to be implemented, but it is a good idea to implement as many as possible.

6.1 Concrete controller template

Additionally, the controller file must also include a template for creating the concrete controller. The snippet should be named `[name]_execution_snippet`, where `[name]` is the controller name. The snippet will be added to the bottom of the concrete controller file and is used to instantiate the controller class.

CHAPTER 7

View Classes

View classes take in data that the model returns, and returns data that is viewable. For instance, if the model returns a simple dictionary:

{'x': 3, 'y': 10, 'result': 30}

The the view's job is to take this data and return a visual representation, such as:

```
3 * 10 == 30
```

or:

```
<!DOCTYPE html>
<html>
    <body>
        <p>3 * 10 == 30</p>
    </body>
</html>
```

7.1 Creating View Classes

All view classes must descent from `giotto.views.BaseView`. Each class needs to implement at least one mimetype method:

```
from giotto.views import BaseView

class MyView(BaseView):
    @renders('text/plain')
    def plaintext(self, result):
        return "%(x)s * %(y)s == %(result)s" % result
```

Each method can be named whatever you want, and must be decorated with the `@renders` decorator. The `renders` decorator takes multiple string arguments representing the mimetype that method renders. When creating views, make sure there is no 'impedance mismatch' between the data that the model returns, and the data the view is written to take in. For instance, the above mimetype method is designed to display a dictionary with three keys (`x`, `y`, and `result`). If the model was changed to return a list, this view method will crash.

7.2 Return values

Each mimetype render method should return either a string:

```
return "return a string"
```

or a dictionary with body and mimetype tags:

```
return {'body': "this is whats returned, 'mimetype': 'text/plain'}
```

If the method returns just a string, the controller wil treat the content as the first mimetype passed into the `renders` decorator::

```
@renders('text/plain', 'text/html', 'text/x-cmd')
def multi_render(self, result):
    return "text"
```

This content will be treated as `text/plain` by the controller.

7.3 Persisting data

Sometimes, instead of mearly displaying model-generated values to the user, you want to persist that value. An example of this is saving session data to a cookie instead of displaying it in the HTML. To do this, add the data to the `persist` key in the dictionary the render function returns:

```
{'body': "this is whats returned, 'mimetype': 'text/plain', 'persist': {'cookie_name': 'cookie value
```

7.4 BasicView

There is a view class called `BasicView` that was created to be a quick and dirty way to view most any data. While developing your application, it is a good idea to use `BaseView` until you have settled on a consistent data type that your model returns. Also you should inherit all custom views from `BasicView` for convenience.

7.5 Overriding Renderers

In the manifest file, you can override a renderer by passing in the renderer function as a keyword argument to the view class:

```
manifest = Manifest({
    'multiply': Program(
        model=[multiply],
        view=BasicView(
            html=lambda m: str(m)
        ),
    ),
})
```

This program will output a string representation of the model output when viewed in an HTML context. When viewed in a json context, the output will be the json renderer defined in `BasicView`.

7.6 Renderer functions

Renderer functions take two arguments, the first argument is the object that the model returns, and the second argument is the errors that may have came from a previous invocation. The second argument is optional. Renderer functions can

be either defined as a method on the view class, or passed in to the view class constructor.

CHAPTER 8

Rendering Templates

8.1 Configuring the template engine

Giotto comes configured to render templates with the Jinja2 library. In your config file, there should be a variable called `jinja_env`. By default, it is configured to look for templates in the root folder of your project. To change this, modify the path that gets sent into the `Environment` constructor.

8.2 Rendering Templates

Giotto comes with support for rendering jinja template out of the box. To render a template with a program, add the `jinja_template` renderer function to your view class:

```
from giotto.programs import Program, Manifest
from giotto.views import BasicView, jinja_template

manifest = Manifest({
    'test': Program(
        model=[lambda x,y : {'x': x, 'y':y, 'result': int(x) * int(y)],
        view=BasicView(
            html=jinja_template('mytemplate.html'),
        ),
    ),
})
```

In the template, the result of the model will be the only object in the context. Access ths data by accessing the `data` variable:

```
<!DOCTYPE html>
<html>
    {{ data.x }} * {{ data.y }} == {{ data.result }}
</html>
```

To change the name of the object in the context, pass in the `name` attribute to the renderer:

```
BasicView(
    html=jinja_template('mytemplate.html', name="model"),
)
```

8.3 Partial Template Renderings

Sometimes you want to render a template with multiple data sources. For instance, you want the view to render the template with data from the model, but you also want a csrf protection token to be in your template that comes from output middleware. You can achieve this by using a partial template render:

```
from giotto.views import partial_jinja_template

BasicView(
    html=partial_jinja_template('mytemplate.html'),
)
```

If you template looked like this:

```
<!DOCTYPE html>
<html>
    {{ data.x }} * {{ data.y }} == {{ data.result }}<br>
    {{ something_else }}
</html>
```

The render would look like this:

```
<!DOCTYPE html>
<html>
    12 * 10 == 120<br>
    {{ something_else }}
</html>
```

Since the `something_else` variable is undefined, it is ignored by the rendering in the view. Now your middleware class can parse the body of the response and render it again, this time with the `something_else` variable defined.

CHAPTER 9

Error Pages

When your application throws an error, whie developing, you want to see a traceback. When running the development server via `./http --run`, errors show up with a traceback and debug information. But when your application goes into production, you do not want your users to see this debug page. The default error page allows the user to run arbitrary python code on the server. This is a huge security problem. Instead, you can define an error page template. When the HTTP controller gets an exception coming from your program, it will return to the user your error template rendered with the details of the error.

9.1 Configuring error pages

In your application's config file, set the `error_template` variable to the path of the template you want gitto to render in order to make your error page. This path should be assessible by the jinja2 enviornment defined in the same config file. This template, when rendered, will contain the following variables:

- **code** - This is the error code, such as 400, 404, 500, 403, etc.
- **exception** - The name of the exception that was caught, such as `IndexError`, `ValueError`, etc.
- **message** - The message that is associated with the exception (the string passed into the exception that was raised)
- **traceback** - The traceback of the exception. You may not want to render this, but its there if you want it. If your project's taget audience are technically capable people, it may benefitial to include tracebacks in the error template.

9.2 Switching between error pages and the debug page

In your config file, there is a setting called `debug`. If it is set to `True` the debug page will be used when your application throws an exception. Set it to `False` to use the error template.

CHAPTER 10

Middleware

All middleware classes must implement a method for each controller name. For input middleware, each method should take one argument, *request*, and return a new *request* object:

```
class SomeInputMiddleware(object):
    def http(self, request):
        do_some_stuf()
        return request

    def cmd(self, request):
        do_some_stuf()
        return request

    def sms(self, request):
        do_some_stuf()
        return request
```

Output middleware should take two arguments, *request* and *response*, and should return a new *response* object:

```
class SomeOutputMiddleware(object):
    def http(self, request, response):
        do_some_stuf()
        return response

    def cmd(self, request, response):
        do_some_stuf()
        return response

    def sms(self, request, response):
        do_some_stuf()
        return response
```

The appropriate method will be called depending on how the program has been invoked.

CHAPTER 11

Cache

Cache is a way to improve performance in your application. It works by storing the return value of the model and view, so subsequent requests can skip the calculation.

11.1 Configuring cache

To enable caching for your application, set a value for the `cache` variable in your project's `config.py` file. The value must be an instance of a class derived from the `GiottoKeyValue` class. Additionally, if you are only going to use default settings (localhost, default port), you can set it to just a string, e.g. "redis", "memcache", "database", etc.

11.1.1 RedisKeyValue

By default, this backend tries to connect to redis running on `localhost`, port `6379`. To change this, pass in connection data to the constructor:

```
from giotto.keyvalue import RedisKeyValue
cache = RedisKeyValue(host="10.10.0.5", port=4000)
```

Or, if just using the defaults:

```
cache = "memcache"
```

11.1.2 MemcacheKeyValue

Additionally, if you want to use memcache:

```
from giotto.keyvalue import MemcacheKeyValue
cache = MemcacheKeyValue()
```

By default, this backend tries to connect to memcache on `localhost`, port `11211`. To change this, pass in connection data to the constructor:

```
cache = MemcacheKeyValue(hosts=['10.10.0.5:11211'])
```

11.1.3 DatabaseKeyValue

You can also use a cache backend that stores its data onto the database:

```
from giotto.keyvalue import DatabaseKeyValue
cache = DatabaseKeyValue()
```

Before this backend can be used, you must run the `syncdb` program to create the database tables.

11.1.4 LocMemKeyValue

For development, you can use the `LocMemKeyValue` which stores its data internally in a python dictionary:

```
from giotto.keyvalue import LocMemKeyValue
cache = LocMemKeyValue()
```

Note: `LocMemKeyValue` only saves data as long as the concrete controller lives. For instance, while the http server is running, data will be saved, but if you restart the server, all data will be lost. This key/value backend is virtually useless with the command line controller, as all keys are cleared after each invocation.

11.1.5 DummyKeyValue

You can also use `DummyKeyValue` which always returns misses for all keys:

```
from giotto.keyvalue import DummyKeyValue
cache = DummyKeyValue()
```

11.2 Enabling caching for programs

To enable cache for a program, add a value (in seconds) to the `cache` attribute of the program instance:

```
def square(x):
    return {'x': x * x}

manifest = Manifest({
    'squared': Program(
        controllers=('http-get', 'cmd'),
        model=[square],
        cache=3600, # one hour
        view=MyViewClass,
    )
})
```

The first request that comes into this program will be calculated. The result of this calculation (the model's return value) will be stored in the cache. Depending on what you've configured, this will be either Redis or Memcache.

The next request that comes in, with the same value for `x`, the calculation will be skipped and the value stored in cache will be returned instead. After one hour has passed, the next request will calculate a new value. To configure the program to never expire cache values, set the `cache` value to 0. To turn off cache, either omit the cache attribute, or set it to `None`.

11.3 Under the hood

A cache key is constructed from each incoming request. The cache key is in the following format:

(model arguments)(program name)(mimetype)

The output of the view is what gets stored under this key. In the example above, the cache key (if invoked from within a web browser), would be the following:

```
invocation -> curl http://localhost:5000/my_program?x=12
cache key -> {'x': 12}(my_program)(text/html)
```

If invoked from within the commandline controller, the cache key would be the following:

```
invocation -> ./giotto-cmd my_program x=12
cache key -> {'x': 12}(my_program)(text/cmd)
```

CHAPTER 12

Models

A model is a function that takes in values from the controller, and returns values to the view. Giotto is very hands off when it comes to the model. Your model function can call other model functions, it can call facebook's API, it can talk to a database, it can do anything. Giotto comes with some helpers to make using SQLAlchemy easier, but it's use is completely optional. Giotto, unlike some popular popular web frameworks, can be ran without any database whatsoever.

12.1 Integration with SQLAlchemy

When defining a model, you should use the baseclass that comes out of the `better_base` function found in `giotto.utils`. Models descending from this base class will have a `todict` method which Giotto uses to easily convert your model to JSON.

In your project's config file, define the base class:

```
from giotto.utils import better_base
Base = better_base()
```

Now, when defining your model class, use this baseclass:

```
from giotto.config import Base
class MyModel(Base):
    """
    This is a model
    """
```

(anything in your config file will be available by importing from `giotto.config`).

You must also define a `engine` and `session` variable in the config file. `engine` is the result of `create_engine` from sqlalchemy. `session` is the result of `sessionmaker`. The default config file created by the `giotto` command line utility contains boilerplate for defining these variables.

12.2 Creating tables

Add the manageent manifest to your project:

```
from giotto.programs import management_manifest, Manifest

manifest = Manifest({
```

```
    'mgt': management_manifest
})
```

And to create the tables, do the following command:

```
$ ./cmd mgt/make_tables
```

This program, by default, will only be available via the command line controller. The `make_tables` command will only find models you have defined with that baseclass, if that model has been imported into your project's manifest. This is unlike Django and other frameworks, that use a `models.py` convention.

12.3 InvalidInput

When a model recieves data from a POST request, and the data does not validate, the model should raise the `giotto.exceptions.InvalidInput` exception. The controller will catch this exception and re-render the get portion of the request. When the exception is raised, you can add key word arguments to the exception constructor describing the dat that was invalid. As an example, consider the following model function that only accepts numbers below 100:

```
from giotto.exceptions import InvalidInput

def multiply(x, y):
    try:
        x = int(x or 0)
        y = int(y or 0)
    except:
        raise InvalidInput("Numbers only", x={'value': x}, y={'value': y})

    if x > 100 and y > 100:
        raise InvalidInput('x and y are too large', x={'message': 'too large', 'value': x}, y={'messa
    if x > 100:
        raise InvalidInput('x is too large', x={'message': 'too large', 'value': x}, y={'value': y})
    if y > 100:
        raise InvalidInput('y is too large', y={'message': 'too large', 'value': y}, x={'value': x})

    return {'x': x, 'y': y, 'product': x * y}
```

The values of the keyword argument should be a dictionary with two keys, `value` and `message`.

CHAPTER 13

Model Mocking

Most development teams are composed of two "halves". One half is composed of backend engineers who are great at writing code and working with complex software such as databases. The other half i composed of designers, who are great with Photoshop, but not so good with setting up databases and writing code.

Model mocking allows an application to be ran in an environment where the database does not need to be ran. The model part of a program is completely ignored, and the mock object is returned instead:

```
from giotto.programs import Program
from giotto.views import BasicView
import psycopg2

def square(x):
    """
    Connect to a postgres server and multiply two numbers together.
    """
    conn = psycopg2.connect("dbname=test user=postgres")
    cur = conn.cursor()
    cur.execute("SELECT %s * %s;", (x, x))
    result = cur.fetchone()
    return {'x': x, 'square': result[0]}

class Square(Program):
    controllers = ('http-get', 'cmd')
    model = [square, {'x': 12, 'square': 144}]
    view = BasicView
```

When this application is ran normally, a connection must be made to a posgres server. If the server is not running, the program would crash. Not only that, but in some cases, there must be data in the database for this program to execute properly. This can cause issues that make developing applications difficult.

If you run the http server with the `--model-mock` option, the model function will get bypassed. No connection is attempted to a PostgreSQL server. The value of the mock will be used instead.

13.1 Defining a model mock

The model mock is defined in the program as the second item in the `model` list:

```
model = [model_func, model_mock]
```

When you define a model mock, make sure the mock is in the same format what the model returns.

CHAPTER 14

Authentication

There is an application within the contrib submodule called `auth` that handles authentication. This application uses SQLAlchemy to store user data.

This `User` model stores two pieces of information: username and password. The password is stored as an encrypted string. Encrypting is done automatically by the bcrypt library.

This user class is not intended to store all information that an application developer would want to store for a user. The developer is meant to create their own user profile table that connects to the User table via foreign key.

14.1 Configuring Authentication

In the config file of your application, add the following variables:

- `auth_session_engine` - To a instance of a GiottoKeyValue class (See the cache documentation), or the string 'database'.
- `auth_regex` - To a regular expression that represents what usernames can be. If left blank, this will be set to `^[\d\w]{4,30}$`.

14.2 Enabling Authentication for your application

An authenticated application typically contains all four of these components:

1. A registration page.
2. A login page.
3. A log out page.
4. Some way to authenticate requests so other parts of the application can utilize authentication.

14.3 Login page

To set up a login page, add the following to your project's manifest:

```
from giotto.contrib.auth.models import is_authenticated
from giotto.contrib.auth.middleware import SetAuthenticationCookies
from giotto.control import Redirection
from giotto.programs import Program
```

```
from giotto.views import JinjaTemplateView

{
    'login': [
        Program(
            input_middleware=[AuthenticationMiddleware, NotAuthenticatedOrRedirect('/')],
            controllers=('http-get',),
            view=JinjaTemplateView('login.html'),
        ),
        Program(
            controllers=('http-post',),
            input_middleware=[AuthenticationMiddleware],
            model=[is_authenticated("Invalid username or password")],
            view=Redirection('/'),
            output_middleware=[SetAuthenticationCookies],
        ),
    ],
}
```

The first program handles generating the login form. The second program handles the submitted data from the login form. You may want to change the name of the template that is passed into the first program. The login template must contain a form that POSTS's its data to a page of the same name of the form. The name of the programs (`login` in this example) can be anything. You can also change the path that the second program redirects to after successful registration. In this case, on successful registration, the user is redirected to the root program: `/`.

The login template should look a little like this:

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Login</h1>
        {% if errors %}
            <span style="color: red">{{ errors.message }}</span>
        {% endif %}
        <form method="post">
            username: <input type="text" name="username">
            <br>
            password: <input type="password" name="password">
            <br>
            <input type="submit">
        </form>
    </body>
</html>
```

The value of `{{ errors }}` will be empty when the login form is first rendered. If the data that is submitted is not valid (it does not match an existing user/pasword), the form is re-generated with the `errors` object containing error information.

14.4 Register page

To add a registration page to your application, add the following to your manifest:

```
from giotto.contrib.auth.models import basic_register
from giotto.contrib.auth.middleware import SetAuthenticationCookies
from giotto.control import Redirection
from giotto.programs import Program
```

```
from giotto.views import JinjaTemplateView

{
    'register': [
        Program(
            controllers=('http-get',),
            view=JinjaTemplateView('register.html'),
        ),
        Program(
            controllers=('http-post',),
            model=[basic_register],
            view=Redirection('/'),
            output_middleware=[SetAuthenticationCookies],
        ),
    ],
}
```

The register template should look like this:

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Register</h1>
        {% if errors %}
            <span style="color: red">{{ errors.message }}</span>
        {% endif %}
        <form method="post">
            <span style="color: red">{{ errors.username.message }}</span><br>
            username: <input type="text" name="username" value="{{ errors.username.value }}">
            <br>
            <span style="color: red">{{ errors.password.message }}</span><br>
            password: <input type="password" name="password">
            password again: <input type="password" name="password2">
            <br>
            <input type="submit">
        </form>
    </body>
</html>
```

The value of the `errors` object will have a `password` and `username` object, which will each contain `message` and `value` keys. `message` contains the error message, and `value` contain the previous value that was entered.

14.5 Logout Page

Adding a logout program is very simple, just add this to your project's manifest:

```
from giotto.programs import Program
from giotto.control import Redirection
from giotto.contrib.auth.middleware import LogoutMiddleware

{
    'logout': Program(
        view=Redirection('/'),
        output_middleware=[LogoutMiddleware],
    ),
}
```

You can change the url that you get redirected to after logging out by changing the value passed into `Redirection`.

14.6 Interacting with authentication with other programs

To access the currently logged in user from within a model function, add the `LOGGED_IN_USER` primitive to your model function's arguments:

```
from giotto.primitives import LOGGED_IN_USER
from giotto.programs import Program, Manifest
from giotto.contrib.auth.middleware import AuthenticationMiddleware
from giotto.views import BasicView

def show_logged_in_user(user=LOGGED_IN_USER):
    return {'user': user}

manifest = Manifest({
    'show_logged_in': Program(
        input_middleware=[AuthenticationMiddleware],
        model=[show_logged_in_user],
        view=BasicView,
    )
})
```

The controller knows how to extract `LOGGED_IN_USER` from the incoming request. This primitive can only be used if the `AuthenticationMiddleware` is added to the input middleware stream. All programs that wish to take advantage of the authentication system need to have `AuthenticationMiddleware` added. It may be convenient to create a subclass of `Program` with `AuthenticationMiddleware` baked in:

```
from giotto.programs import Program, Manifest
from giotto.contrib.auth.middleware import AuthenticationMiddleware
from giotto.views import BasicView

class AuthProgram(Program):
    input_middleware=[AuthenticationMiddleware]

def show_logged_in_user(user=LOGGED_IN_USER):
    return {'user': user}

 manifest = Manifest({
    'show_logged_in': AuthProgram(
        model=[show_logged_in_user],
        view=BasicView,
    )
})
```

You can also take advantage of a few middleware classes

14.6.1 AuthenticatedOrRedirect and NotAuthenticatedOrRedirect

These middleware classes, if added to the input middleware stream, will redirect the request to another program (via 302 redirect) depending on authentication status:

```
from giotto.programs import Program
from giotto.contrib.auth.middleware import AuthenticationMiddleware, NotAuthenticatedOrRedirect
from giotto.views import JinjaTemplateView
```

```
Program(
    input_middleware=[AuthenticationMiddleware, NotAuthenticatedOrRedirect('/')],
    controllers=('http-get',),
    view=JinjaTemplateView('login.html'),
),
```

In this example, only non authenticated users will see the `login.html` page. All authenticated users will get redirected to the root program.

14.6.2 AuthenticatedOrDie

This middleware class will return a 403 (error page) if the request is not authenticated:

```
from giotto.programs import Program
from giotto.contrib.auth.middleware import AuthenticationMiddleware, AuthenticatedOrDie
from giotto.views import JinjaTemplateView

{
    'new': Program(
        input_middleware=[AuthenticationMiddleware, AuthenticatedOrDie],
        view=JinjaTemplateView('new_blog.html'),
        controllers=('http-get',),
    ),
}
```

In this example, only authenticated users can create a new blog. All other users will get a 403 page.

CHAPTER 15

Serving Static Files

Handling Static files is very simple in Giotto. Just add the `ServeStatic` program to your manifest:

```
from giotto.contrib.static.programs import StaticServe

manifest = Manifest({
    'static': StaticServe('/var/www/static_files')
})
```

When you run this manifest through a controller, you can access the static files like so:

```
./giotto-cmd static/text/myfile.txt
```

The file at `/var/www/static_files/text/myfile.txt` will be displayed as if you had `cat` the file.

15.1 SingleStaticServe

This program is useful to place a single file onto the manifest:

```
manifest = Manifest({
    'static': StaticServe('/var/www/static_files'),
    'favicon': StaticServe('/var/www/static_files/favicon.ico'),
})
```

The name you give the `SingleStaticServe` program in your manifest has to have the extension omitted.

CHAPTER 16

Deploying Giotto

16.1 uWSGI

When creating the concrete controller file, make sure you include the `http` option. Install uwsgi:

```
pip install uwsgi
```

and run the server by using the http concrete controller file as the wsgi file:

```
uwsgi --http :5000 --wsgi-file http_controller
```

16.2 gunicorn

Install gunicorn:

```
pip install gunicorn
```

Gunicorn works slightly differently than uwsgi. Some modifications of the automatically generated concrete controller is required.

The run the following command:

```
gunicorn --workers=1 --log-level=debug http_controller:application
```

16.3 Apache and Nginx

Since these servers are not pure python, it is more tricky to get them set up. Basically the steps are the same for deploying any other wsgi application. Use `http_controller.py` file as the WSGI file.

CHAPTER 17

WSGI Middleware

Giotto interacts with the WSGI specification very well. To add WSGI middleware to your application, edit the `http` file:

```
application = make_app(manifest, model_mock=mock)

if not config.debug:
    from raven import Client
    from raven.middleware import Sentry

    application = Sentry(
        application,
        Client(config.sentry_dsn)
    )

    application = error_handler(application)
```

`raven` is a tool that is used to report errors in our application to a sentry server.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

G

M