<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Application Creation</td>
<td>1</td>
</tr>
<tr>
<td>2 Requests, Responses, and Controllers</td>
<td>3</td>
</tr>
<tr>
<td>3 Engines</td>
<td>11</td>
</tr>
<tr>
<td>4 Indices and tables</td>
<td>13</td>
</tr>
</tbody>
</table>
class gimme.app.App(name='gimme', engine=<gimme.engines.Jinja2Engine object at 0x3fa0850>, logger=<class 'gimme.servers.logger.logger.SysLogger'>)

The central class that ties a Gimme application together.

There are a few configuration parameters that may be passed to App objects upon instantiation, although most of the configuration is done via middleware.

Basic usage may look something like the following:

```python
class gimme.app.App

app = App('some_app')

# Add middleware.
app.use(gimme.middleware.session())

# Add routes to a controller that we have not shown here.
app.routes.get('/', SomeController.index + 'index.html')
app.routes.get('/about', SomeController.about + 'about.html')

# Start the app’s development web server.
app.listen()
```

### Variables

- **dirname** – Stores the path information from where the app was started from.
- **routes** – An instance of `gimme.routes.Routes`, which is used for mapping routes to controllers, etc.

### Parameters

- **name** *(str)* – The name to use for the app in the logger.
- **engine** – The template engine adapter to use.
- **logger** – The log class to use.

```python
get(key)
```

Gets a key from the app, previously set with `set()`.

#### Parameters

- **key** – The key to fetch

```python
listen(port=8080, host='127.0.0.1', http_server=<class 'gimme.servers.http.server.HTTPServer'>)
```

Starts the built-in development webserver.
• **port** (*int*) – The port to listen on.

• **host** (*str*) – The hostname/IP address to listen on.

• **http_server** – What class to use for the HTTP server.

```bash
set (key, value)
```

Sets a config value. These key/value pairs can be used to store arbitrary information. Retrieve via `get()`.

**Parameters**

• **key** – The key to store under.

• **value** – The value to store.

```bash
use (middleware)
```

Add middleware to the app:

```javascript
app.use(gimme.middleware.body_parser())
app.use(gimme.middleware.session())
```

For more information, see `gimme.middleware.Middleware`.

**Parameters** **middleware** – The middleware to add to the app.
Of course, one of the primary things that any web framework must do, is handle requests and make responses. The MVC paradigm generally ties these two elements together via the controller. Gimme is no different.

Typically, the application developer will create controllers and their corresponding methods, then create routes to those methods. A very simple application may do something like the following:

```python
app = gimme.App()

class RootController(gimme.Controller):
    def index(self):
        self.response.headers['Some-Header'] = 'value'
        return {
            'today': datetime.datetime.now(),
            'user_name': self.request.session['user_name']
        }

app.routes.get('/', RootController.index + '/path/to/template.html')
```

There’s really not much more to it than that (although I do recommend checking out the documentation for the `gimme.request.Request` and `gimme.response.Response` classes).

One important gotcha: controller methods that are to be bound to routes should NEVER start with `_`.

```python
class gimme.controller.Controller(app, request, response)
```

Like most frameworks, the controller is what bridges the gap between the model (business logic) and the view. In Gimme, one thing that is a little different from the classic MVC methodology is that controllers are not directly tied to views. This is explained further in the `gimme.routes.Routes` documentation.

Controllers are instantiated automatically by Gimme on an as-needed basis. If a controller’s method is needed to fulfill a request, the controller will be instantiated and the method called and its data returned. This means that controllers are instantiated per request, as opposed to once, as the application is started.

One thing to note is that, while Gimme instantiates controllers automatically, in order to perform unit tests, you must instantiate controllers in your tests. This can be done like so:

```python
controller = YourController(your_app, request_obj, response_obj)
```

Of course, to do the above, you must have request and response objects. To fetch these objects, the easiest way is generally via the `gimme.routes.Routes.match()` method:

```python
request, response = your_app.routes.match(wsgi_environ)
```

(Where the `wsgi_environ` variable is a dictionary-like object with necessary WSGI environ parameters, such as `PATH_INFO`, `REQUEST_METHOD`, etc.)
One other thing that may be worth noting is that this class uses a custom metaclass that, upon definition, scans for any method whose name does not start with a _, and replaces it with an instance of `gimme.controller.ControllerMethod`.

Variables

- **app** – The gimme app.
- **request** – The request object.
- **response** – The response object.

```python
class gimme.controller.ControllerMethod(cls, fn)
```

ControllerMethod objects wrap regular controller methods, as defined by the application programmer.

In order to allow greater flexibility and expressiveness, all controller methods whose name do not start with _ are replaced with instances of ControllerMethod. These objects can be called just as the controller methods can, but in addition, other methods and operators are made available.

What all this means, in practice, is that you can do things like this in your route definitions:

```python
app.routes.get('/', RootController.index + 'index.html')
```

I hope to elaborate on this and how it works in the future.

```python
__add__(other)
```

Creates a new `gimme.controller.MethodRenderer` object with the other object as the first renderer. Or, if other is a string, instantiates a `gimme.renderers.Template` object with the string and passes that to the `gimme.controller.MethodRenderer` instead.

Example:

```python
method_renderer = SomeController.some_method + gimme.Template('index.html')
# or, a shortcut:
method_renderer = SomeController.some_method + 'index.html'
```

Parameters **other** – Either a string or an instance of `gimme.renderers.BaseRenderer`.

```python
__eq__(content_type)
```

Returns a `gimme.renderers.Format` object that responds to HTTP Accept headers as specified by `content_type`.

Example:

```python
format_obj = YourController.your_method == 'text/html'
```

Parameters **content_type** (str) – The content type to customize format to.

```python
__weakref__
```

List of weak references to the object (if defined)

```python
compress()
```


```python
json()
```


```python
template(template_path)
```

Returns a `gimme.controller.MethodRenderer` object with a `gimme.renderers.Template` renderer directed at the path provided by `template_path`.

---

Chapter 2. Requests, Responses, and Controllers
Parameters **template_path** *(str)* – Where to point the template to.

class gimme.controller.MethodRenderer *(items=[])*
The MethodRenderer provides an interface for multiple renderers to all manipulate the controller method’s output in an orderly fashion.

It should be noted that the MethodRenderer class inherits from list(). The first item in the list should always be an instance of gimme.controller.ControllerMethod. Every other item should inherit from gimme.renderers.BaseRenderer.

__add__(other)
Adds another BaseRenderer to the list.

Parameters **other** *(BaseRenderer)* – The renderer to add.

Returns self

__call__()
Calls the ControllerMethod and then calls render() on each of the BaseRenderer objects.

Returns The final output of the execution chain described above.

__eq__(content_type)
Returns a Format object with all of the renderers in this object inside it. Sets the Format content type to content_type.

This is typically used in something like the following:

```python
app.routes.get('/', SomeController.index + (  
    (SomeController.index == 'text/plain') | # <--  
    (SomeController.index.json() == 'application/json') |  
    (SomeController.index.template('template.html') == 'text/html')  
))
```

Parameters **content_type** *(str)* – The content type to set the Format object to.

Returns A Format object.

__weakref__  
list of weak references to the object (if defined)

compress()
Adds a Compress object to the list.

Returns self

json()
Adds a Json object to the list.

Returns self

template *(template_path)*
Adds a Template object with the provided path to the list.

Parameters **template_path** *(str)* – The path to the template

Returns self

class gimme.request.Request *(app, environ, match=None)*
The Request class is responsible for parsing the request headers and body into Pythonic shindigs that are (supposed to be) very easy to work with.

Variables
• **app** – The Gimme application.
• **environ** – The WSGI environ dict.
• **headers** – The request headers (an instance of `gimme.headers.RequestHeaders`).
• **wsgi** – The WSGI headers. Also an instance of `gimme.headers.RequestHeaders`.
• **params** – A `gimme.dotdict.DotDict` of the parameters from the route URI.
• **query** – The query string in a dictionary-like object.
• **accepted** – A handy parsing of the HTTP “Accept” header. An instance of `gimme.parsers.accepted.AcceptedList`.
• **accepted_languages** – A handy parsing of the HTTP “Accept-Language” header. An instance of `gimme.parsers.accepted.AcceptedList`.
• **acceptedCharsets** – A handy parsing of the HTTP “Accept-Charset” header. An instance of `gimme.parsers.accepted.AcceptedList`.
• **cookies** – The raw HTTP “Cookie” string.
• **type** – Either an instance of `gimme.parsers.contenttype.ContentType` if the HTTP header “Content-Type” is present or `None`.

**Parameters**

• **app** – The Gimme application.
• **environ** – The WSGI environ dict.
• **match** – The matching route, if any.

`accepts(content_type)`
Tests if a given content type is in the HTTP “Accept” header.

**Parameters**
**content_type** – The content type to check for.

`accepts_charset(charset)`
Tests if a given charset is in the HTTP “Accept-Charset” header.

**Parameters**
**charset** – The charset to check for.

`accepts_language(language)`
Tests if a given language is in the HTTP “Accept-Language” header.

**Parameters**
**language** – The language to check for.

`get(key)`
Return the specified request header.

**Parameters**
**key** – The header to return.

`host`
A shortcut for the HTTP “Host” header.

*Note: Strips any port off of the header.*

`ip`
A shortcut for the “REMOTE_ADDR” environ parameter.

`original_url`
A shortcut for the “REQUEST_URI” environ parameter.

`param(key)`
A simple helper method that looks for a given key in three places: self.params, self.body, and self.query.
**Parameters** `key` – The parameter to look for.

### path
A shortcut for the “PATH_INFO” environ parameter.

### protocol
Gimme only supports HTTP as of the time of this writing.

### raw_body
The raw request body.

### secure
Right now, Gimme only runs in standard HTTP mode. SSL should be implemented at the HTTP server end and piped to Gimme via FastCGI.

### subdomains
A list of subdomains per the `host` attribute.

### xhr
Whether or not the HTTP header “X-Requested-With” is present and set to “XMLHttpRequest”.

#### class `gimme.response.Response` *(app, route, request)*
The Response class is responsible for aggregating all of the information required for generating a response and for rendering said response.

**Variables**

- `app` – The Gimme application.
- `route` – The route that matched the request/response objects.
- `request` – The request object.
- `headers` – An instance of `gimme.headers.ResponseHeaders` that will eventually be converted to a string appropriate for the HTTP response headers.
- `body` – The body of response.
- `locals` – An instance of `gimme.dotdict.DotDict` that is intended to store arbitrary information that is unique to a given response object.

#### attachment
A helper for setting the “Content-Disposition” and “Content-Type” HTTP headers for a given file. This makes it easy to pass a file to the client for downloading.

Note, this is implemented as a setter-only property. In other words, you cannot “get” the attachment information with this property, but only set it.

This is valid:

```python
response.attachment = "something.jpg"
```

But this is **not**:

```python
print response.attachment
```

#### charset
A helper for getting and setting the charset portion of the “Content-Type” header.

#### `clear_cookie` *(key)*
A helper method for expiring a previously-set cookie.

**Parameters** `key` – The key/name of the cookie to expire.
cookie(key, value=None, expires=None, http_only=False, secure=False, path='/', domain=None)
A helper method for setting the HTTP “Set-Cookie” header.

Parameters
- **key** – The key/name of the cookie.
- **value** – What to set the cookie to.
- **expires** – Number of seconds that the cookie should last before being expired. Defaults to None, which makes the cookie last indefinitely.
- **http_only** – Set the HttpOnly flag.
- **secure** – Set the Secure flag.
- **path** – Limit the URI path that the cookie applies to.
- **domain** – Limit the domain that the cookie applies to.

get(key)
A shortcut for getting a response header.

Parameters
- **key** – The key of the header to get.

links
A helper for setting the “Link” header. To set links, simply do something like:

```python
response.links = {
    'http://google.com/meta.rdf': 'meta'
}
```

Note that this is a setter-only property. Reading from it is not valid.

location
A shortcut for getting and setting the HTTP “Location” header.

redirect(path, code=302)
A shortcut for setting the response status and HTTP “Location” header appropriately to redirect the requesting client.

Parameters
- **path** (str) – The location to send the client to.
- **code** – The status code to set.

set(key, value)
A shortcut for setting a response header.

Parameters
- **key** – The key of the header to set.
- **value** – The value of the header.

status
The response status code. This is an instance of StatusCode and can be set using any of the following methods:

```python
response.status = 404
response.status = "404 Not Found"
response.status.set(404)
```

Checking a status is as simple as doing any of the following:
response.status == 404
response.status == "404 Not Found"

**type**

A shortcut for getting and setting the HTTP “Content-Type” header.
Gimme makes it very easy to implement adapters to interface with your favorite template engine (I call these adapters “engines”, terminology taken from Express JS).

All that is needed is to subclass of the `gimme.engines.BaseEngine` (abstract base) class and implement the `render()` method.

Example:

```python
class SomeEngine(gimme.engines.BaseEngine):
    def render(self, template, data):
        return fetch_template(template, data)
```

Now, to use your new engine, simply pass it to your app on instantiation:

```python
app = App(engine=SomeEngine())
```

```python
class gimme.engines.BaseEngine
    The base class that all engines should derive from.

    render(template, data)
    Returns a rendered template.

    Parameters
    • template – The template to render (usually a filename/path).
    • data – The data to send to the template engine. Usually a dict.
```

```python
class gimme.engines.Jinja2Engine(template_path='views', environment=None)
    Provides Jinja2 integration.

    Parameters
    • template_path – The path, relative the current working directory, that Jinja2 should look for templates.
    • environment – A Jinja2 Environment object. Defaults to None. If provided, template_path is ignored.
```
Indices and tables

- genindex
- modindex
- search