Watson - Framework

Release 2.2.1

March 12, 2014

It's elementary my dear Watson

Watson is an easy to use framework designed to get out of your way and let you code your application rather than spend time wrangling with the framework. It follows the convention over configuration ideal, although the convention can be overriden if required. Out of the box it comes with a standard set of defaults to allow you to get coding straight away!

CHAPTER 1

Requirements

Watson is designed for Python 3.3 and up.

CHAPTER 2

Dependencies

Watson currently requires the following modules to work out of box:

- **Jinja2**
  - For view templating

These will be installed automatically if you have installed Watson via pip.

CHAPTER 3

Optional Dependencies

Some packages within Watson require third party packages to run correctly, these include:

- **Memcached**
  - pip package: python3-memcached

Notes about these dependencies can be found within the relevant documentation in the *Reference Library*.

CHAPTER 4

Installation

```
pip install watson-framework
```

CHAPTER 5

Testing

Watson can be tested with py.test. Simply activate your virtualenv and run `python setup.py test`.

CHAPTER 6

Benchmarks

Using falcon-bench, Watson received the following requests per second (Django and Flask supplied for comparative purposes).

1. watson.....11,920 req/sec or 83.89 μs/req (3x)
2. django.....7,696 req/sec or 129.94 μs/req (2x)
3. flask.....4,281 req/sec or 233.58 μs/req (1x)

CHAPTER 7

Contributing

If you would like to contribute to Watson, please feel free to issue a pull request via Github with the associated tests for your code. Your name will be added to the AUTHORS file under contributors.

CHAPTER 8

Table of Contents

8.1 Getting Started

8.1.1 Installation

All stable versions of Watson are available via pip and can be installed using the following command `pip install watson-framework` via your CLI of choice.

Watson is maintained at Github, and can be used to get the latest development version of the code if required.

Setting up a virtualenv

We recommend creating a standalone environment for each new project you work on to isolate any dependencies that it may need. To do so enter the following commands in your terminal:

```
>>> pyvenv /where_you_want_to_store_venv
>>> source /where_you_want_to_store_venv/bin/activate
```

Verifying the installation

To ensure that Watson has been installed correctly, launch `python` from your CLI and then enter the following:

```
>>> import watson
>>> print(watson.__version__)
>>> # latest watson version will be printed here
```

Once you've got Watson installed, head on over to the *Your First Application* area to learn how to create your first web application.

8.1.2 Configuration

Introduction

While Watson is primarily built with convention over configuration in mind, there are still plenty of configuration options that can be modified to override the default behaviour.

Note: To override values within the default configuration, you only need to replace those values within your own configuration file. The application with automatically merge the defaults with your new options.

Application Configuration

Configuration for Watson is just a standard python module (and should be familiar to those who have used Django previously). Available keys for configuration are:

- debug
- dependencies
- views
- session
- events
- logging

You can see the default configuration that Watson uses within the `watson.framework.config` module.

Debug

Debug is responsible for determining if the application is running in debug mode, and the relevant profiling settings.

watson.framework.config

```
debug = {
   'enabled': False,
   'panels': {
      'watson.debug.panels.request.Panel': {
         'enabled': True
      },
      'watson.debug.panels.application.Panel': {
         'enabled': True
      },
      'watson.debug.panels.profile.Panel': {
         'enabled': True,
         'max_results': 20,
         'sort': 'time',
      },
      'watson.debug.panels.framework.Panel': {
         'enabled': True
      },
   }
}
```

Dependencies

The configuration of your application will automatically be added to the container, which can then be retrieved via the key `application.config`.

See the dependency injection *Key Concepts* for more information on how to define dependencies and container parameters.

Views

Watson utilizes multiple renderers to output the different views that the user may request. Each renderer is retrieved from the dependency injection container (see above), with the name key being the same as the relevant dependency name.

watson.framework.config

```
views = {
    'default_format': 'html',
    'renderers': {
        'default': {
            'name': 'jinja2_renderer',
            'config': {
                'extension': 'html',
                'paths': [os.path.join(os.getcwd(), 'views')]
            }
        },
        'xml': {'name': 'xml_renderer'},
        'json': {'name': 'json_renderer'}
    },
    'templates': {
        '404': 'errors/404',
        '500': 'errors/500'
    }
}
```

The above configuration sets the default renderer to use Jinja2. It also specifies two other renderers, which will output XML and JSON respectively. There are also a set of templates defined, which allows you to override templates that will be used. The format of these being 'existing template path': 'new template path' (relative to the views directory).

Session

By default Watson will use File for session storage, which stores the contents of each session in their own file within your systems temporary directory (unless otherwise specified in the config).

watson.framework.config

```
session = {
    'class': 'watson.http.sessions.File',
    'options': {}  # a dict of options for the storage class
}
```

See the storage methods that are available for sessions in the *Reference Library*.

Events

Events are the core to the lifecycle of both a request and the initialization of a Watson application. The default configuration sets up 5 events which will be executed at different times of the lifecycle.

watson.framework.config

```
events = {
    events.EXCEPTION: [('app_exception_listener',)],
    events.INIT: [
        ('watson.debug.profilers.ApplicationInitListener', 1, True)
    ],
    events.ROUTE_MATCH: [('watson.framework.listeners.RouteListener',)],
    events.DISPATCH_EXECUTE: [('app_dispatch_execute_listener',)],
    events.RENDER_VIEW: [('app_render_listener',)],
}
```

Logging

Watson will automatically catch all exceptions thrown by your application. You can configure the logging exactly how you would using the standard libraries logging module.

```
logging = {
    'callable': 'logging.config.dictConfig',
    'ignore_status': {
        '404': True
    },
    'options': {
        'version': 1,
        'disable_existing_loggers': False,
        'formatters': {
            'verbose': {
                'format': '%(asctime)s - %(name)s - %(levelname)s - %(process)d %(thread)d - %(messa
            },
            'simple': {
                'format': '%(asctime)s - %(levelname)s - %(message)s'
            },
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'DEBUG',
                'formatter': 'verbose',
                'stream': 'ext://sys.stdout'
            },
        },
        'loggers': {},
        'root': {
            'level': 'DEBUG',
            'handlers': ['console']
        }
    }
}
```

The callable key allows you to change the way the logging it to be configured, in case you want to use a different method for logging. ignore_status allows you to ignore specific status codes from being logged (chances are you don't want to log 404 errors).

A common logging setup may look similar to the following:

```
logging = {
    'options': {
        'handlers': {
            'error_file_handler': {
                'class': 'logging.handlers.RotatingFileHandler',
                'level': 'DEBUG',
                'formatter': 'verbose',
                'filename': '../data/logs/error.log',
                'maxBytes': 10485760,
                'backupCount': '20',
                'encoding': 'utf8'
            },
        },
        'loggers': {
            'my_app': {
                'level': 'DEBUG',
```

```
                'handlers': ['error_file_handler']
            },

        },
    }
}
```

Integrating Sentry

Sentry is a great piece of software that allows you to aggregrate your error logs. Integrating it into Watson is straightfoward, and only requires modifying the configuration of your application.

```
logging = {
    'options': {
        'handlers': {
            'sentry': {
                'level': 'ERROR',
                'class': 'raven.handlers.logging.SentryHandler',
                'dsn': 'http://SENTRY_DSN_URL_GOES_HERE',
            },
        },
        'loggers': {
            'my_app': {
                'level': 'DEBUG',
                'handlers': ['sentry']
            }
        }
    }
}
```

You can then access the logger from within your app with the following code:

```
import logging
logging.getLogger(__name__)
logger.error('Something has gone wrong')
```

Extending the Configuration

There are times when you may want to allow other developers to get access to your configuration from dependencies retrieved from the container. This can easily be achieved by the use of lambda functions.

First create the settings you wish to retrieve in your settings:

app/config/config.py

```
my_class_config = {
    'a_setting': 'a value'
}
```

And then within your dependency definitions you can reference it like this:

app/config/config.py

```
dependencies = {
    'definitions': {
        'my_class': {
            'item': 'my.module.Klass',
```

```
            'init': [lambda ioc: ioc.get('application.config')['my_class_config']]
        }
    }
}
```

When my.module.Klass is initialized, the configuration settings will be passed as the first argument to the __init__ method.

8.1.3 Your First Application

Directory Structure

Watson has a preferred directory structure for it's applications which can be created automatically by the `watson-console.py newproject [project name] [app name]` command.

```
/project_root
    /app_name
        /config
            config.py
            dev.py.dist
            prod.py.dist
            routes.py
        /controllers
        /views
            /layouts
        app.py
    /data
        /cache
        /logs
        /uploads
    /public
        /css
        /img
        /js
    /tests
    console.py
```

Tip: For example **watson-console.py newproject sample.com.local sample** creates a new project named sample.com.local and an application package named sample

---

The application will be created within the current working directory, unless you override it with the `-d DIR` option.

Once the structure has been created, you can use `./console.py` to perform related console commands from within the application, for example: `./console.py routes` to display a list of routes for the application.

Configuration

By creating your project using the **newproject** command Watson will generate 3 configuration files for your application as well as a route file.

1. config.py
2. dev.py.dist
3. prod.py.dist

4. routes.py

config.py is the basic configuration to get your application up and running locally and is identical to dev.py.dist. By default dev.py.dist will enable profiling and debugging of the application. If you have retrieved the application from a VCS then you would make a copy of dev.py.dist with the name config.py and modify the settings within there.

app/config/config.py

```
from project.config.routes import routes

debug = {
    'enabled': True,
}
```

When deploying to a production environment you would make a copy of prod.py.dist and name it config.py to load the relevant production settings.

The dist files are designed to maintain a consistent configuration when an application is being worked on by multiple developers. We recommend adding [app_name]/config/config.py to your .gitignore file to prevent your personal configuration from being used by another developer.

routes.py contains all the routes associated with the application. For more detail on how to define routes, please see the *MVC* Key Concept area.

app/config/routes.py

```
routes = {
    'index': {
        'path': '/',
        'defaults': {'controller': 'project.controllers.Index'}
    }
}
```

Putting it all together

Most likely you'll want to develop locally first and then deploy to a production environment later. Watson comes packaged with a command to run a local development server which will automatically reload when changes are saved. To run the server simply change to the project directory and run `./console.py rundev` and then visit http://127.0.0.1:8000 in your favorite browser where you'll be greeted with a page saying welcome to Watson.

A initial controller is created for you in app_name/controllers/index.py which will response to a request for / in your browser (from the above routes.py definition)

app/controllers/index.py

```
from watson import __version__
from watson.framework import controllers

class Index(controllers.Rest):
    def GET(self):
        return 'Welcome to Watson v{0}!'.format(__version__)
```

Being a Rest controller any request will be routed to the instance method matching the HTTP_REQUEST_METHOD environ variable from the associated request. One of the benefits of using a Rest controller is that you no longer need to check the request method to determine how you should respond.

An alternative would be to use an Action controller instead. This would be represented in the following way:

```
from watson import __version__
from watson.framework import controllers

class Index(controllers.Action):
    def index_action(self):
        return 'Welcome to Watson v{0}!'.format(__version__)
```

All Action controller methods are suffixed with _action. For a more indepth look at what functions a controller can perform, check out the *common_usage* area for controllers. For a general overview of how controllers are used within Watson, check out the *MVC* Key Concept area.

The presentation layer (or view) is matched based on lowercased versions of the the class name and action of the controller. For the above request the following view is rendered:

app/views/index/get.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Watson!</title>
    </head>
    <body>
        <h1>{{ content }}</h1>
        <p>You are now on your way to creating your first application using Watson.</p>
        <p>Read more about Watson in <a href="http://github.com/watsonpy/watson-framework">the docume
    </body>
</html>
```

For more information on views, check out the *MVC* Key Concept area.

You will also want to make sure that you unit test your application, and you can do that by running `./console.py runtests`. A simple unit test is already included when the **newproject** command is run. It is designed to fail so make sure you go in and make the required changes for it to pass!

All tests are located under the tests directory. For example the demo unit test is located at tests/[app name]/controllers/test_index.py.

Watson supports both nose and py.test for use with the runtests command.

8.2 Key Concepts

8.2.1 Events

Events are a major part of how Watson wires together your application. You can hook into the events and register your own event listeners by modifying your application configuration.

The event dispatcher holds a record of all listeners and the their associated priority, number of executions, and the event name that they are to be executed on.

Note: The basic flow for the event system within Watson is the following:

Create dispatcher > Add listeners > Trigger event > Return results from triggered listeners

---

The anatomy of an Event

An event is used to pass around data within an application without introducing a tight coupling between objects. A basic event contains the following:

A name The name of the event that will trigger listener callbacks

A target What triggered the event

A set of parameters Data sent through with the event

When an event is triggered from an event dispatcher, all listeners that are listening for a particular event name will be triggered and their responses returned.

Inbuilt events

The lifecycle of a Watson application is maintained by 5 different events defined in watson.framework.events:

event.framework.init Triggered when the application is started

event.framework.route.match Triggered when the application attempts to route a request and returns the matches

event.framework.dispatch.execute Triggered when the controller is executed and returns the response

event.framework.render.view Triggered when the controller response is processed and the view is rendered

event.framework.exception Triggered when any exception occurs within the application and the executes prior to the render view to generate any 400/500 error pages

These events are triggered by the shared_event_dispatcher which is instantiated from the applications IocContainer.

Creating and registering your own event listeners

By default several listeners are defined within the watson.framework.config module, however additional listeners can be added to these events, and even prevent the default listeners from being triggered.

Let's assume that we want to add a new listener to the watson.framework.events.INIT event. First lets add a new events key to the applications configuration module. Replace app_name with the applications name.

app_name/config/config.py

```
from watson.framework import events

events = {
}
```

Note: Whatever defined in here will be **appended** to Watsons default configuration.

Next, we'll need to create a listener, which just needs to be a callable object. As the listener is going to be retrieved from the IocContainer, it is useful to subclass watson.di.ContainerAware so that the container will be injected automatically. The triggered listener is passed a single event as the argument, so make sure that you allow for that.

app_name/listeners.py

```
from watson.di import ContainerAware
from watson.framework import listeners

class MyFirstListener(listeners.Base, ContainerAware):
    def __call__(self, event):
```

```
        # we'll perform something based on the event and target here
        pass
```

Finally we'll need to register the listener with the event dispatcher. Each listener needs to be added as a tuple, which takes the following arguments: (object, int priority, boolean once_only). If no priority is specified a default priority of 1 will be given. The highest priority will be executed first. If only_once is not specified then it will default to False.

app_name/config/config.py

```
events = {
    events.INIT: [
        ('app_name.listeners.MyFirstListener', 2, True)
    ]
}
```

Now once your application is initialized your event will be triggered.

8.2.2 Dependency Injection

Introduction

Dependency injection is a design pattern that allows us to remove tightly coupled dependencies from our code, making it easier to maintain and expand upon as an application grows in size.

A hardcoded dependency

```
class MyClass(object):
    def __init__(self):
        self.some_dependency = SomeDependency()

my_class = MyClass()
```

Utilizing dependency injection

```
class MyClass(object):
    def __init__(self, some_dependency):
        self.some_dependency = some_dependency

my_class = MyClass(SomeDependency())
```

As you can see above, the latter removes the dependency from the class itself, creating a looser coupling between components of the application.

The lifecycle of a dependency

Dependencies within Watson go through two events prior to being retrieved from the container.

watson.di.container.PRE_EVENT Triggered prior to instantiating the dependency

watson.di.container.POST_EVENT Triggered after instantiating the dependency, by prior to being returned

These events are only triggered once per dependency, unless the dependency is defined as a 'prototype', in which case a new instance of the dependency is retrieved on each request.

Example Usage

Watson provides an easy to use IoC (Inversion of Control) container which allows these sorts of dependencies to be managed easily. Lets take a look at how we might instantiate a database connection without dependency injection (for a more complete example of this, check out watson-db)

app_name/db.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

some_engine = create_engine('postgresql://scott:tiger@localhost/')
Session = sessionmaker(bind=some_engine)
session = Session()
```

app_name/controllers/user.py

```
from watson.framework import controllers
from app_name import db

class Profile(controllers.Rest):
    def GET(self):
        return {
            'users': db.session.query(User).all()
        }

    def POST(self):
        user = User(name='user1')
        db.session.add(user)
        db.session.commit()
```

One thing to note here is that the configuration for the collection is stored within the code itself. While we could abstract this out to another module, there would still be some sort of dependency on retrieving the configuration from that module. We also introduce a hardcoded dependency by requiring the db module. By using the IocContainer, we can abstract both of these issues out keeping our codebase clean.

Using the IocContainer

First we'll create code required to connect to the database, removing any hardcoded configuration details (note this is purely an example).

app_name/db.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

Session = sessionmaker()

def create_session(container, connection_string):
    some_engine = create_engine(connection_string)
    return Session(bind=some_engine)
```

Next we have to configure the dependency within the applications configuration settings. Learn more about the ways to configure your dependencies.

app_name/config/config.py

```
dependencies = {
    'definitions': {
```

```
        'db_read': {
            'item': 'app_name.db.create_session',
            'init': {
                'connection_string': 'postgresql://read:access@localhost/'
            }
        },
        'db_write': {
            'item': 'app_name.db.create_session',
            'init': {
                'connection_string': 'postgresql://write:access@localhost/'
            }
        }
    }
}
```

We now have two dependencies defined in the applications configuration settings. One of the additional benefits of using the IoC container is that subsequent requests for a dependency will return an already instantiated instance of the dependency (unless otherwise specified).

Now all that's left is to retrieve the dependency from the container. We can do this by calling container.get(dependency_name). As controllers are retrieved from the container and extend ContainerAware, our container is automatically injected into them.

app_name/controllers/user.py

```
from watson.framework import controllers

class Profile(controllers.Rest):
    def GET(self):
        # we only want to read from a slave for some reason
        db = self.get('db_read')
        return {
            'users': db.query(User).all()
        }

    def POST(self):
        # we only want writes to go to a specific database
        db = self.get('db_write')
        user = User(name='user1')
        db.add(user)
        db.commit()
```

We can also take this a step further and remove the container itself so that we're not utilizing it as a service locator (db = self.get('db_*')). We do this by adding the controller itself to the dependency definitions, and injecting the dependency either as a property, setter, or through the constructor. We can get access to the container itself (for retrieving dependencies or configurtion) via lambdas, or just by the same name as the definition. Note that you can also omit the 'item' key if you are configuring a controller.

app_name/config/config.py

```
dependencies = {
    'definitions': {
        'db_read': {
            'item': 'app_name.db.create_session',
            'init': {
                'connection_string': 'postgresql://read:access@localhost/'
            }
        },
        'db_write': {
```

```
            'item': 'app_name.db.create_session',
            'init': {
                'connection_string': 'postgresql://write:access@localhost/'
            }
        },
        'app_name.controllers.user.Profile': {
            'property': {
                'db_read': 'db_read',  # References the db_read definition
                'db_write': 'db_write'
            }
        }
    }
}
```

Now we simply modify our controller to suit the new definitions...

app_name/controllers/user.py

```
from watson.framework import controllers

class Profile(controllers.Rest):
    db_read = None
    db_write = None

    def GET(self):
        return {
            'users': self.db_read.query(User).all()
        }

    def POST(self):
        user = User(name='user1')
        self.db_write.add(user)
        self.db_write.commit()
```

Configuring the container

The container is defined within your applications configuration under the key 'dependencies' as seen below.

```
dependencies = {
    'params': {
        'param_name': 'value'
    },
    'definitions': {
        'name': {
            'item': 'package.module.object',
            'type': 'singleton',
            'init': {
                'keyword': 'arg'
            },
            'property': {
                'attribute': 'value'
            },
            'setter': {
                'method_name': {
                    'keyword': 'arg'
                }
            }
        }
```

```
    }
}
```

Lets break this down into it's different components:

params

```
'params': {
    'param_name': 'value'
}
```

Params are arguments that can be inserted into dependencies via init, property or setter processors. Any argument that is being used in one of the above processor definitions will be evaluated against the params and replaced with it's value. If a param value has the same name as a dependency, then that dependency itself will be injected.

An example dependency using params

```
dependencies = {
    'params': {
        'host': '127.0.0.1'
    },
    'definitions': {
        'db': {
            'item': 'app.db',
            'init': {
                'hostname': 'host'
            }
        }
    }
}
```

When the above dependency is retrieved, the 'host' param will be injected into the objects constructor.

8.2.3 MVC

Model View Controller (MVC) is a design pattern that encourages you to write code that adheres to seperation of concerns and DRY principles. It's also begun to be widely adapted into just about every single web framework available.

While Watson follows your standard mvc design pattern, it does not force you to use any particular ORM as your way to interact with models. It is up to you, the developer, to determine the most appropriate database abstraction method.

Terminology

Throughout the documentation various controllers, models and views will be referenced many times and it is important that they are interpreted within the context of the framework.

1. **Model**: The application data
2. **View**: The interface the user is presented with
3. **Controller**: Interprets a request and converts it to the relevant output

The basic lifecycle of a request

What Watson does have an opinion on is lifecycle that a request must go through to become a response.

1. **Browser request comes in** A standard http request which is processed by server
2. **Application run method executed** This begins the processing of the request by Watson
3. **Environ variables converted into watson.http.message.Request object** This request object is considered immutable and should not be modified
4. **Request matched against application routes** Defined within your applications configuration file
5. **Controller initialized** A new controller is initialized on each request
6. **Controller dispatch method executed returning a particular view** The method called is based on the Request params, or the Request method depending on the controller type
7. **Controller response converted to a watson.http.message.Response** Used by the application to deliver the response
8. **Response delivered to browser** The relevant markup is sent to the users browser

8.3 Common Usage

8.3.1 Controllers

Watson provides two different types of controllers which are called Action and Rest respectively. Each one has its own uses and there is no one size fits all solution. A controller is only initialized once, and will not be initialized on each request. Due to this, you must not store any sort of state on the controller. Everything relating to the request the controller is currently dealing with can be retrieved by Controller.event.params['context'].

Creating controllers

Action controllers

Action controller methods are defined explicitly within the applications route configuration. In the following example, when a request is made to `/` then the `app_name.controllers.Public` controller is initialized, and the `indexAction` method is invoked.

app_name/config/config.py

```
routes = {
    'index': {
        'path': '/',
        'defaults': {'controller': 'app_name.controllers.Public', 'action': 'index'}
    },
}
```

app_name/controllers/__init__.py

```
from watson.framework import controllers

class Public(controllers.Action):
    def index_action(self):
        pass
```

RESTful controllers

RESTful controller methods are based upon the HTTP request method that was made by the user. In the following example, when a request is made to / the `app_name.controllers.User` controller is initialized, and the relevant HTTP request method is invoked.

app_name/config/config.py

```
routes = {
    'index': {
        'path': '/',
        'defaults': {'controller': 'app_name.controllers.User'}
    },
}
```

app_name/controllers/__init__.py

```
from watson.framework import controllers

class User(controllers.Rest):
    def GET(self):
        pass

    def POST(self):
        pass

    def PUT(self):
        pass

    def DELETE(self):
        pass
```

Common tasks

Accessing Request and Response objects

No changes should be made to the request object, and they should be treated as immutable. However any modifications can be made to the response object, as it will be used when the application renders the response to the user.

```
from watson.framework import controllers

class Controller(controllers.Rest):
    def GET(self):
        request = self.request  # the watson.http.messages.Request object
        response = self.response  # the watson.http.messages.Response object
```

For more information on request and response objects see the *Reference Library*.

Redirecting a request to another route or url

```
from watson.framework import controllers

class Controller(controllers.Rest):
    def GET(self):
        self.redirect('/')  # redirect the user to specific url
```

```
    def POST(self):
        self.redirect('home')  # redirect the user to a named route
```

For more information on the various arguments that can be passed to redirect() see the *Reference Library*.

When a user is redirected, any POST or PUT variables will be saved within the users session to solve the PRG (Post Redirect Get) issue. These variables may then be accessed to populate a form for example and are stored within the `redirect_vars` attribute of the controller. They can subsequently be cleared via the `clear_redirect_vars()` method on the controller.

Flash messaging

Flash messaging is a way to send messages between requests. For example, a user may submit some form data to be saved, at which point the application would

```
from watson.framework import controllers
from app_name import forms

class Controller(controllers.Rest):
    def GET(self):
        return {
            'form': forms.Login(),  # form has a POST method
        }

    def POST(self):
        form = forms.Login()
        form.data = self.request.post
        if form.is_valid():
            self.flash_messages.add('Successfully logged in', 'info')
        else:
            self.flash_messages.add('Invalid username or password', 'error')
        self.redirect('login')
```

```
<html>
    <head></head>
    <body>
        {% for namespace, message in get_flash_messages() %}
        <div class="{{ namespace }}">{{ message }}</div>
        {% endfor %}
        {{ form.open() }}
        {{ form.username.render_with_label() }}
        {{ form.password.render_with_label() }}
        {{ form.submit }}
        {{ form.close() }}
    </body>
</html>
```

Once flash messages have been iterated over, they are automatically cleared from the flash message container.

404 and other http errors

Raising 404 Not Found errors and other HTTP error codes are simple to do directly from the controller.

```
from watson.framework import controllers, exceptions

class Controller(controllers.Rest):
```

```
    def GET(self):
        raise exceptions.NotFoundError()
```

To raise a custom error code, you can raise an ApplicationError with a message and code specified.

```
from watson.framework import controllers, exceptions

class Controller(controllers.Rest):
    def GET(self):
        raise exceptions.ApplicationError('Some horrible error', status_code=418)
```

8.3.2 Views

Views within Watson are considered 'dumb' in that they do not contain any business or application logic within them. The only valid 'logic' that should be contained within a view would be simple for loops, if statements, and similar constructs.

The templating engine prefered by Watson is Jinja2, however this can easily be switched to another engine if required.

```
views = {
    'renderers': {
        'default': {
            'name': 'my_new_renderer',
        }
    }
}
```

`my_new_renderer` needs to be configured within the IocContainer to instantiate the new renderer.

Specifying different response formats

To output the response in different formats is quite a simple task and only involves modifying the route itself (it can be modified without changing the route, however this is not really encouraged).

```
routes = {
    'home': {
        'path': '/',
        'defaults': {
            'format': 'json'
        }
    }
}
```

and the subsequent controller...

```
from watson.framework import controllers

class Public(controllers.Rest):
    def GET(self):
        return {'hello': 'world'}
```

The user can also be made responsible for determining the response format by correctly defining the route to support this. This is particularly useful if you're creating an API and need to support multiple formats such as XML and JSON.

```
routes = {
    'home': {
        'path': '/something.:format',
```

```
        'requires': {
            'format': 'json|xml'
        }
    }
}
```

In the above route, any request being sent to /something.xml or /something.json will output the data in the requested format.

Customizing the view path By default Watson will try to load views from `project_name/app_name/views/controller_name/action.html` where project_name is the name of your project, app_name is the name of your application module, controller_name is the name of the controller that was executed and action is http request method (if the controller is a Rest controller) or the specified action from the route (if the controller is an Action controller).

This above convention is defined within watson.framework.config.views, however this can be overridden if required.

The views settings within watson.framework.config

```
views = {
    'default_format': 'html',
    'renderers': {
        'default': {
            'name': 'jinja2_renderer',
            'config': {
                'extension': 'html',
                'paths': [os.path.join(os.getcwd(), 'views')]
            }
        },
        'xml': {'name': 'xml_renderer'},
        'json': {'name': 'json_renderer'}
    },
    'templates': {
        '404': 'errors/404',
        '500': 'errors/500'
    }
}
```

Jinja2 Helper Filters and Functions

There are several Jinja2 helpers available:

url(*route_name*, *host=None*, *scheme=None*, ***kwargs*)

Convenience method to access the router from within a Jinja2 template.

Parameters

- **route_name** – the route to build the url for
- **host** – the host name to add to the url
- **scheme** – the scheme to use
- **kwargs** – additional params to be used in the route

Return type string matching the url

merge_query_string(*obj*, *values*)

Merges an existing dict of query string values and updates the values.

Parameters **obj** – the original dict

config()
Convenience method to retrieve the configuration of the application.

get_request()
Convenience method to retrieve the current request.

8.3.3 Routing

Routing is an important part of Watson as it ties a request directly to a controller (and subsequently a view). Routes are generally defined within the `project/app_name/config/routes.py` file, which is then imported into your applications configuration file.

The anatomy of a route

Routes within Watson consist of several key parts, and at a bare minimum must contain the following:

1. A name to identify it
2. A path to match against
3. A controller to execute

A route is defined within a simple dict() in the following way:

```
routes = {
    'route_name': {
        'path': '/',
        'options': {
            'controller': 'package.module.Controller'
        }
    }
}
```

Attention: 0.2.6 introduced a breaking change that separated options from defaults

When a user hits / in their browser, then a new instance of package.module.Controller will be instantiated, and the relevant view will be rendered to the browser.

Ordering of routes

The ordering of routes is important, however as dicts are unordered you must supply a priority within the route.

```
routes = {
    'route_name': {
        'path': '/resource',
    },
    'route_name_post': {
        'path': '/resource',
        'accepts': ('POST',)
        'priority': 1
    }
}
```

When /resource is sent to the browser, the response from route_name will always be returned first, regardless of the http request method being used. However by adding priority to route_name_post, if the POST request method is used, then route_name_posts contents will be returned.

Creating complex routes

There are times when you may wish to only allow access to a particular route via a single http request method, or perhaps only if a specific format is requested.

Accepting specific request methods

Simply add a list/tuple of valid http request methods to the 'accepts' key on the route.

```
routes = {
    'route_name': {
        'path': '/resource',
        'accepts': ('GET', 'POST')
        'options': {
            'controller': 'package.module.Controller'
        }
    }
}
```

Url	Verb	Matched
/resource	GET	Yes
/resource	PUT	No

Subdomains

Simply add the subdomain to the 'subdomain' key on the route (it also accepts a tuple of subdomains).

```
routes = {
    'route_name': {
        'path': '/resource',
        'subdomain': 'clients'
    }
}
```

Url	Host	Matched
/resource	www.site.com	No
/resource/123	clients.site.com	Yes

Creating segment driven routes

A segment route is basically a route that contains a series of placeholder values. These can be mandatory, or optional depending on how they are configured. Any segments will be sent as keyword arguments to the controllers that they execute, though they can be ignored.

Mandatory segment

```
routes = {
    'route_name': {
        'path': '/resource/:id',
    }
}
```

Url	Matched	id
/resource	No	
/resource/123	Yes	123

Optional segment

```
routes = {
    'route_name': {
        'path': '/resource/:id[/:resource_action]',
        'defaults': {
            'resource_action': 'view'
        }
    }
}
```

Url	Matched	id	resource_action
/resource	No		
/resource/123	Yes	123	view
/resource/123/edit	Yes	123	edit

Optional segment with required values

```
routes = {
    'route_name': {
        'path': '/resource/:id[/:resource_action]',
        'defaults': {
            'resource_action': 'view'
        },
        'requires': {
            'resource_action': 'view|edit|delete'
        }
    }
}
```

Url	Matched	id	resource_action
/resource	No		
/resource/123	Yes	123	view
/resource/123/edit	Yes	123	edit
/resource/123/show	No		

Generating urls from routes

Routes can be converted back to specific urls by using the assemble method on either the router object itself, or the assemble method on the route. Most of the time a url needs to be generated within the controller action, and as such the controller class provides a url() method which takes the same arguments as assemble(). Any keyword arguments that are passed to these functions replace any segments within the route path.

Route configuration (leaving out default key for berevity)

```
routes = {
    'route_name': {
        'path': '/resource/:id',
    }
}
```

In a controller within your application

```
class Resource(controllers.Rest):
    def GET(self):
        resource = self.url(id=3)   # /resource/3
        # could also be represented as self.get('router').assemble(id=3)
```

8.3.4 Requests

For the following we're assuming that all requests come through the route:

```
routes = {
    'example': {
        'path': '/path',
        'options': { 'controller': 'Public' }
    }
}
```

Accessing request variables

Accessing GET variables

Assuming the following http request:

```
/path/?query=string&value=something
```

```
class Public(controllers.Rest):
    def GET(self):
        query = self.request.get['query']  # string
```

Accessing POST variables

Assuming the following http request:

```
/path
```

With the following key/value pairs of data being posted: data: something

```
class Public(controllers.Rest):
    def GET(self):
        data = self.request.post['data']  # something
```

Accessing FILE variables

Assuming the following http request:

```
/path
```

With

```
<input type="file" name="a_file" />
```

being posted.

```
class Public(controllers.Rest):
    def GET(self):
        file = self.request.files['a_file']  # cgi.FieldStorage
```

Accessing cookies

Assuming the following http request:

```
/path
```

```
class Public(controllers.Rest):
    def GET(self):
        cookies = self.request.cookies  # CookieDict
```

Accessing session data

Assuming the following http request:

```
/path
```

With the following data being stored in the session: data: value

```
class Public(controllers.Rest):
    def GET(self):
        session = self.request.session
        session_data = session['data']  # value
        session.id  # id of the session
```

Accessing SERVER variables (environ variables)

```
class Public(controllers.Rest):
    def GET(self):
        server = self.request.server['PATH_INFO']  # /path
```

8.3.5 Forms

Forms are defined in a declarative way within Watson. This means that you only need to define fields you want without any other boilerplate code.

```
from watson import form
from watson.form import fields

class Login(form.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    submit = fields.Submit(value='Login', button_mode=True)
```

Which when implemented in a view would output:

```
<html>
    <body>
        <form>
            <label for="username">Username</label><input type="text" name="username" />
            <label for="password">Password</label><input type="text" name="password" />
            <button type="submit">Login</button>
        </form>
    </body>
</html>
```

Field types

Fields are referenced by their HTML element name. Whenever a field is defined within a form any additional keyword arguments are used as attributes on the element itself. Current fields that are included are:

Field	Output
Input	<input type="" />
Button	<button></button>
Textarea	<textarea></textarea>
Select	<select></select>

There are also a bunch of convenience classes as well which may add additional validators and filters to the field.

Field	Output
Input	<input type="" />
Radio	<input type="radio" />
Checkbox	<input type="checkbox" />
Text	<input type="text" />
Date	<input type="date" />
Email	<input type="email" />
Hidden	<input type="hidden" />
Csrf	<input type="csrf" />
Password	<input type="password" />
File	<input type="file" />
Submit	<input type="submit" /> or <button>Submit</button>

These can all be imported from the `watson.form.fields` module.

Populating and binding objects to a form

Form data can be populated with any standard Python dict.

```
form = forms.Login()
form.data = {'username': 'Simon'}
```

These values can then be retrieved by:

```
form.username.value  # Simon
```

If the field has been through the validation/filter process, you can still retrieve the original value that was submitted by:

```
form.username.original_value  # Simon
```

Binding an object to the form

Sometimes it's worth being able to bind an object to the form so that any posted data can automatically be injected into the object. This is a relatively simple task to achieve:

Object entities

```
class User(object):
    username = None
    password = None
    email = None
```

Edit user form

```
from watson import form
from watson.form import fields

class User(forms.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    email = fields.Email(label='Email Address')
```

Controller responsible for saving the user

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def POST(self):
        user = User()
        form = forms.User('user')
        form.bind(user)
        form.data = self.request.post
        if form.is_valid():
            user.save()  # save the updated user data
```

When is_valid() is called the POST'd data will be injected directly into the User object. While this is great for simple CRUD interfaces, things can get more complex when an object contains other objects. To resolve this we have to define a mapping to map the flat post data to the various objects (we only need to define the mapping for data that isn't a direct mapping).

A basic mapping consists of a dict of key/value pairs where the value is a tuple that denotes the object 'tree'.

```
mapping = {
    'field_name': ('attribute', 'attribute', 'attribute')
}
```

We'll take the same example from above, but modify it slightly so that our User object now also contains a Contact object (note that some of this code such as the entities would be handled automatically by your ORM of choice).

Object entities

```
class User(object):
    username = None
    password = None
    contact = None

    def __init__(self):
        self.contact = Contact()

class Contact(object):
    email = None
    phone = None
```

Edit user form

```
from watson import form
from watson.form import fields

class User(forms.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    email = fields.Email(label='Email Address')
    phone = fields.Email(label='Phone Number')
```

Controller responsible for saving the user

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def POST(self):
        user = User()
        form = forms.User('user')
        form.bind(user, mapping={'email': ('contact', 'email'), 'phone': ('contact', 'phone')})
        form.data = self.request.post
        if form.is_valid():
            user.save()  # save the updated user data
```

Filters and Validators

Filters and validators allow you to sanitize and modify your data prior to being used within your application. By default, all fields have the Trim filter which removes whitespace from the value of the field.

When the is_valid() method is called on the form each field is filtered, and then validated.

To add new validators and filters to a field you simply add them as a keyword argument to the field definition.

```
from watson import form
from watson.form import fields
from watson import validators

class Login(form.Form):
    username = fields.Text(label='Username', validators=[validators.Length(min=10)])
    password = fields.Password(label='Password', validators=[validators.Required()])
    # required can actually be set via required=True
    submit = fields.Submit(value='Login', button_mode=True)
```

For a full list of validators and filters check out filters and validators in the reference library.

Validating post data

Validating forms is usually done within a controller. We'll utilize the Login form above to demonstrate this...

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def GET(self):
        form = forms.Login('login_form', action='/login')
        form.data = self.redirect_vars
        # populate the form with POST'd data to avoid the PRG issue
        # we don't really need to do this
        return {
            'form': form
        }

    def POST(self):
        form = forms.Login('login_form')
        form.data = self.request.post
        if form.is_valid():
            self.flash_messages.add('Successfully logged in')
```

```
            self.redirect('home')
        else:
            self.redirect('login')
```

With the above code, when a user hits /login, they are presented with a login form from the GET method of the controller. As they submit the form, the code within the POST method will execute. If the form is valid, then they will be redirected to whatever the 'home' route displays, otherwise they will be redirected back to the GET method again.

Errors upon validating

When is_valid() is called, all fields will be filtered and validated, and any subsequent error messages will be available via form.errors.

Protecting against CSRF (Cross site request forgery)

Cross site request forgery is a big issue with a lot of code bases. Watson provides a simple way to protect your users against it by using a decorator.

```
from watson import form
from watson.form import fields
from watson.form.decorators import has_csrf

@has_csrf
class Login(form.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    submit = fields.Submit(value='Login', button_mode=True)
```

The above code will automatically add a new field (named csrf_token) to the form, which then will need to be rendered in your view. You will also need to pass the session into the form when it is instantiated so that the csrf token can be saved against the form.

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def GET(self):
        form = forms.Login('login_form', action='/login', session=self.request.session)
        form.data = self.redirect_vars
        return {
            'form': form
        }
```

As the form is validated (via is_valid()), the token will automatically be processed against the csrf validator.

Jinja2 Helper Filters and Functions

There are several Jinja2 helpers available:

label()
: Outputs the label associated with the field.

8.4 Advanced Topics

8.4.1 Deployments

Note: In all of the examples below, `site.com` should be replaced with your own site.

uWSGI

```
uwsgi:
    master: true
    processes: 1
    vaccum: true
    chmod-socket: 666
    uid: www-data
    gid: www-data
    socket: /tmp/site.com.sock
    chdir: /var/www/site.com/site
    logoto: /var/www/site.com/data/logs/error_log
    home: /var/virtualenvs/3.3
    pythonpath: /var/www/site.com
    module: app
    touch-reload: /var/www/site.com/site/app.py
```

nginx

```
server {
    listen 80;
    server_name site.com;
    root /var/www/site.com/public;

    location /css {
        access_log off;
    }

    location /js {
        access_log off;
    }

    location /img {
        access_log off;
    }

    location /fonts {
        access_log off;
    }

    location / {
        include uwsgi_params;
        uwsgi_pass unix:/tmp/site.com.sock;
    }
}
```

8.5 Reference Library

8.5.1 watson.framework.applications

class watson.framework.applications.**Base**(*config=None*)
The core application structure for a Watson application.

It makes heavy use of the IocContainer and EventDispatcher classes to handle the wiring and executing of methods. The default configuration for Watson applications can be seen at watson.framework.config.

_config dict
The configuration for the application.

global_app Base
A reference to the currently running application.

__init__(*config=None*)
Initializes the application.

Registers any events that are within the application configuration.

Example:

```
app = Base()
```

Events: Dispatches the INIT.

Parameters **config** (*mixed*) – See the Base.config properties.

config
Returns the configuration of the application.

container
Returns the applications IocContainer.

If no container has been created, a new container will be created based on the dependencies within the application configuration.

register_events()
Collect all the events from the app config and register them against the event dispatcher.

trigger_init_event()
Execute any event listeners for the INIT event.

class watson.framework.applications.**Console**(*config=None*, *argv=None*)
An application structure suitable for the command line.

For more information regarding creating an application consult the documentation.

Example:

```
application = applications.Console({...})
application()
```

__init__(*config=None*, *argv=None*)

class watson.framework.applications.**Http**(*config=None*)
An application structure suitable for use with the WSGI protocol.

For more information regarding creating an application consult the documentation.

Example:

```
    application = applications.Http({..})
    application(environ, start_response)
```

8.5.2 watson.framework.config

Sphinx cannot automatically generate these docs. The source has been included instead:

```
# -*- coding: utf-8 -*-
# Default configuration for a Watson application.
# The container itself can be referenced by a simple lambda function such as:
# lambda container: container
#
# Consult the documentation for more indepth setting information.
import os
from watson.framework import events

# Debug settings
debug = {
    'enabled': False,
    'panels': {
        'watson.framework.debug.panels.request.Panel': {
            'enabled': True
        },
        'watson.framework.debug.panels.application.Panel': {
            'enabled': True
        },
        'watson.framework.debug.panels.profile.Panel': {
            'enabled': True,
            'max_results': 20,
            'sort': 'time',
        },
        'watson.framework.debug.panels.framework.Panel': {
            'enabled': True
        },
    }
}

# IocContainer settings
dependencies = {
    'definitions': {
        'shared_event_dispatcher':
        {'item': 'watson.events.dispatcher.EventDispatcher'},
        'router': {
            'item': 'watson.routing.routers.DictRouter',
            'init':
            [lambda container: container.get(
             'application.config').get('routes', None)]
        },
        'profiler': {
            'item': 'watson.framework.debug.profilers.Profiler',
            'init':
            [lambda container: container.get(
             'application.config')['debug']['profiling']]
        },
        'exception_handler': {
            'item': 'watson.framework.exceptions.ExceptionHandler',
            'init':
```

```
            [lambda container: container.get(
             'application.config').get('debug', {})]
        },
        'jinja2_renderer': {
            'item': 'watson.framework.views.renderers.jinja2.Renderer',
            'init': [
                lambda container: container.get('application.config')[
                    'views']['renderers']['default'].get('config', {}),
                lambda container: container.get('application')
            ]
        },
        'json_renderer': {'item': 'watson.framework.views.renderers.json.Renderer'},
        'xml_renderer': {'item': 'watson.framework.views.renderers.xml.Renderer'},
        'app_dispatch_execute_listener': {
            'item': 'watson.framework.listeners.DispatchExecute',
            'init':
            [lambda container: container.get(
             'application.config')['views']['templates']]
        },
        'app_exception_listener': {
            'item': 'watson.framework.listeners.Exception_',
            'init': [
                lambda container: container.get('exception_handler'),
                lambda container: container.get(
                    'application.config')['views']['templates']
            ]
        },
        'app_render_listener': {
            'item': 'watson.framework.listeners.Render',
            'init':
            [lambda container: container.get(
             'application.config')['views']]
        }
    }
}

# View settings
views = {
    'default_format': 'html',
    'renderers': {
        'default': {
            'name': 'jinja2_renderer',
            'config': {
                'extension': 'html',
                'paths': [os.path.join(os.getcwd(), 'views')],
                'filters': ['watson.framework.support.jinja2.filters'],
                'globals': ['watson.framework.support.jinja2.globals'],
            }
        },
        'xml': {'name': 'xml_renderer'},
        'json': {'name': 'json_renderer'}
    },
    'templates': {
        '404': 'errors/404',
        '500': 'errors/500'
    }
}

```

```
# Logging settings
logging = {
    'callable': 'logging.config.dictConfig',
    'ignore_status': (404,),
    'options': {
        'version': 1,
        'disable_existing_loggers': False,
        'formatters': {
            'verbose': {
                'format': '%(asctime)s - %(name)s - %(levelname)s - %(process)d %(thread)d - %(messa
            },
            'simple': {
                'format': '%(asctime)s - %(levelname)s - %(message)s'
            },
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'DEBUG',
                'formatter': 'verbose',
                'stream': 'ext://sys.stdout'
            },
        },
        'loggers': {},
        'root': {
            'level': 'DEBUG',
            'handlers': ['console']
        }
    }
}

# Session settings
session = {
    'class': 'watson.http.sessions.File',
    'options': {
        'timeout': 3600
    }
}

exceptions = {
    'class': 'watson.framework.exceptions.ApplicationError'
}

# Application event settings
events = {
    events.EXCEPTION: [('app_exception_listener',)],
    events.INIT: [
        ('watson.framework.debug.listeners.Init', 1, True),
        ('watson.framework.logging.listeners.Init', 1, True)
    ],
    events.ROUTE_MATCH: [('watson.framework.listeners.Route',)],
    events.DISPATCH_EXECUTE: [('app_dispatch_execute_listener',)],
    events.RENDER_VIEW: [('app_render_listener',)],
}
```

8.5.3 watson.framework.controllers

class watson.framework.controllers.**Action**
A controller thats methods can be accessed with an _action suffix.

Example:

```
class MyController(controllers.Action):
    def my_func_action(self):
        return 'something'
```

class watson.framework.controllers.**Base**
The base class for all controllers.

class watson.framework.controllers.**FlashMessagesContainer**(*session*)
Contains all the flash messages associated with a controller.

Flash messages persist across requests until they are displayed to the user.

__init__(*session*)
Initializes the container.

Parameters **session** (*watson.http.session.StorageMixin*) – A session object containing the flash messages data.

add(*message*, *namespace='info'*)
Adds a flash message within the specified namespace.

Parameters

- **message** (*string*) – The message to add to the container.
- **namespace** (*string*) – The namespace to sit the message in.

add_messages(*messages*, *namespace='info'*)
Adds a list of messages to the specified namespace.

Parameters

- **messages** (*list|tuple*) – The messages to add to the container.
- **namespace** (*string*) – The namespace to sit the messages in.

clear()
Clears the flash messages from the container and session.

This is called automatically after the flash messages have been iterated over.

class watson.framework.controllers.**HttpMixin**
A mixin for controllers that can contain http request and response objects.

_request
The request made that has triggered the controller

_response
The response that will be returned by the controller

clear_redirect_vars()
Clears the redirected variables.

event
The event that was triggered that caused the execution of the controller.

Returns watson.events.types.Event

flash_messages
Retrieves all the flash messages associated with the controller.

Example:

```
# within controller action
self.flash_messages.add('Some message')
return {
    'flash_messages': self.flash_messages
}

# within view
{% for namespace, message in flash_messages %}
    {{ message }}
{% endfor %}
```

Returns A watson.framework.controllers.FlashMessagesContainer object.

forward (*controller*, *method=None*, **args*, ***kwargs*)
Fowards a request across to a different controller.

controller string|object
The controller to execute

method string
The method to run, defaults to currently called method

Returns Response from other controller.

redirect (*path*, *params=None*, *status_code=302*, *clear=False*)
Redirect to a different route.

Redirecting will bypass the rendering of the view, and the body of the request will be displayed.

Also supports Post Redirect Get (http://en.wikipedia.org/wiki/Post/Redirect/Get) which can allow post variables to accessed from a GET resource after a redirect (to repopulate form fields for example).

Parameters

- **path** (*string*) – The URL or route name to redirect to
- **params** (*dict*) – The params to send to the route
- **status_code** (*int*) – The status code to use for the redirect
- **clear** (*bool*) – Whether or not the session data should be cleared

Returns A watson.http.messages.Response object.

redirect_vars
Returns the post variables from a redirected request.

request
The HTTP request relating to the controller.

Returns watson.http.messages.Request

response
The HTTP response related to the controller.

If no response object has been set, then a new one will be generated.

Returns watson.http.messages.Response

url (*route_name*, *host=None*, *scheme=None*, ***params*)
Converts a route into a url.

Parameters

- **route_name** (*string*) – The name of the route to convert
- **host** (*string*) – The hostname to prepend to the route path
- **scheme** (*string*) – The scheme to prepend to the route path
- **params** (*dict*) – The params to use on the route

Returns The assembled url.

class watson.framework.controllers.**Rest**
A controller thats methods can be accessed by the request method name.

Example:

```
class MyController(controllers.Rest):
    def GET(self):
        return 'something'
```

8.5.4 watson.framework.debug

watson.framework.debug.abc

watson.framework.debug.listeners

class watson.framework.debug.listeners.**Init**
Attaches itself to the applications INIT event and initializes the toolbar.

watson.framework.debug.panels

watson.framework.debug.panels.application

watson.framework.debug.panels.framework

watson.framework.debug.panels.logging

watson.framework.debug.panels.profile

watson.framework.debug.panels.request

watson.framework.debug.profile

watson.framework.debug.profile.**execute**(*func*, *sort_order='cumulative'*, *max_results=20*, **args*, ***kwargs*)
Profiles a specific function and returns a dict of relevant timings.

Parameters

- **sort_order** (*string*) – The order by which to sort
- **max_results** (*int*) – The maximum number of results to display

Example:

```
def func_to_profile():
    # do something

response, stats = profile.execute(func_to_profile)
```

watson.framework.debug.toolbar

8.5.5 watson.framework.events

Sphinx cannot automatically generate these docs. The source has been included instead:

```
# -*- coding: utf-8 -*-
INIT = 'event.framework.init'
ROUTE_MATCH = 'event.framework.route.match'
DISPATCH_EXECUTE = 'event.framework.dispatch.execute'
RENDER_VIEW = 'event.framework.render.view'
EXCEPTION = 'event.framework.exception'
COMPLETE = 'event.framework.complete'
```

8.5.6 watson.framework.exceptions

exception watson.framework.exceptions.**ApplicationError**(*message*, *status_code=None*)
A general purpose application error.

ApplicationError exceptions are used to redirect the user to relevant http status error pages.

status_code int
The status code to be used in the response

__init__(*message*, *status_code=None*)

class watson.framework.exceptions.**ExceptionHandler**(*config=None*)
Processes an exception and formats a stack trace.

__init__(*config=None*)

exception watson.framework.exceptions.**InternalServerError**(*message*, *status_code=None*)
500 Internal Server Error exception.

exception watson.framework.exceptions.**NotFoundError**(*message*, *status_code=None*)
404 Not Found exception.

8.5.7 watson.framework.listeners

8.5.8 watson.framework.logging

watson.framework.logging.listeners

class watson.framework.logging.listeners.**Init**
Attaches itself to the applications INIT event and initializes the logger.

8.5.9 watson.framework.support

watson.framework.support

watson.framework.support.console.commands

watson.framework.support.console.commands.application

watson.framework.support.jinja2

watson.framework.support.jinja2.filters

watson.framework.support.jinja2.filters.**get_qualified_name**(*obj*)
Retrieve the qualified class name of an object.

watson.framework.support.jinja2.filters.**label**(*obj*)
Render a form field with the label attached.

watson.framework.support.jinja2.filters.**merge_query_string**(*obj*, *values*)
Merges an existing dict of query string values and updates the values.

Parameters

- **obj** – The original dict
- **values** – The new query string values

Example:

```
# assuming ?page=2
request().get|merge_query_string({'page': 1})  # ?page=1
```

watson.framework.support.jinja2.globals

class watson.framework.support.jinja2.globals.**Config**
Convenience method to retrieve the configuration of the application.

class watson.framework.support.jinja2.globals.**Url**
Convenience method to access the router from within a Jinja2 template.

Example:

```
url('route_name', keyword=arg)
```

watson.framework.support.jinja2.globals.**config**
alias of Config

watson.framework.support.jinja2.globals.**flash_messages**(*context*)
Retrieves the flash messages from the controller.

Example:

```
{{ flash_messages() }}
```

watson.framework.support.jinja2.globals.**request**(*context*)
Retrieves the request from the controller.

Deprecated: Just use 'request'

Example:

```
{{ request() }}
```

watson.framework.support.jinja2.globals.**url**
alias of Url

8.5.10 watson.framework.views

watson.framework.views.decorators

watson.framework.views.decorators.**view**(*template=None*, *format=None*)
Return the view model in a specific format and with a specific template.

This will not work if the response returned from the controller is of the watson.http.messages.Response type.

Parameters

- **func** (*callable*) – the function that is being wrapped
- **template** (*string*) – the template to use
- **format** (*string*) – the format to output as

Returns The view model in the specific format

Example:

```
class MyClass(controllers.Rest):
    @view(template='edit')
    def create_action(self):
        return 'something'
```

W