Spynl Documentation

Release 5.50.rc1

Softwear BV

Jun 19, 2017

Contents

1 Installation **3**

2 Short developer tutorial **5**
2.1 Development tutorial 5

3 Short operations tutorial **11**
3.1 Building & Deploying Tutorial 11

4 In-depth documentation **15**
4.1 ini-settings 15
4.2 Your production environment(s) 15
4.3 Parameter Handling 16
4.4 Serialisation 16
4.5 Translations 16
4.6 Routing 17
4.7 HTML emails 17
4.8 Custom get_user_info 18
4.9 Validation per JSON Schema 18
4.10 Aggregation of Logs 18
4.11 Error Handling 18

Spynl is a Python web framework which extends the Pyramid web framework.

Spynl helps you to manage your web application:

- **Build** your app (via Jenkins)
- **Deploy** your app (via Docker)
- Serve **endpoint documentation** for frontend devs (via Swagger)
- **Inspect** settings and meta-data of running instances in the browser
- **Aggregate** performance indicators and error messages in NewRelic and/or Sentry

Spynl also has a few other in-built utilities which are often necessary in a modern professional web application but easily take a few days to get right:

- Manage translations (via Babel)
- Send templatable, translatable HTML emails
- validate JSON input and output with Schemas

CHAPTER 1

Installation

Here is a (very) quick How-To for installing Spynl:

```
$ pip install spynl
$ spynl dev.serve
```

Now you can visit Spynl's in-built /about endpoint:

```
$ curl http://localhost:6543/about
```

And you should get a reponse like this:


```
{
    "status": "ok",
    "language": "en",
    "time": "2017-02-03T10:13:41+0000",
    "plugins": {},
    "message": "This is a Spynl web application. You can get more information at␣
↪about/endpoints, about/ini, about/versions, about/build and about/environment.",
    "spynl_version": "6.0.1"
}
```


So you see there are a few endpoints with more specific information. Try visiting http://localhost:6543/about/endpoints to see the documentation for in-built endpoints and http://localhost:6543/about/ini to see possible settings.

The *plugins* part is empty because we haven't written any code of our own yet. See the next section for that.

CHAPTER 2

Short developer tutorial

A small tutorial where we build up a small app step by step.

Development tutorial

In this minimalistic tutorial we will make a Spynl plugin called *my-package* with one endpoint. It will be served in a dev environment, documented, tested and translated. We will build the code part by part, but you can browse my-package-index.

The *spynl* CLI has support for common tasks in the development cycle, which we'll use as they become relevant.

A plugin with one endpoint

First, you probably want to make a virtual environment and activate it. Remember to use Python3, preferrably >= 3.5.

Then we install Spynl:

```
$ pip install spynl
```

Now we'll create a folder for our package

```
$ cd $VIRTUAL_ENV/src
$ mkdir my-package
$ touch my-package/setup.py my-package/hello_world.py
```

Here is the content of our (very) simple *setup.py*, where we state that *my-package* is a Spynl plugin:

```
from setuptools import setup

setup(name='my-package',
    entry_points={
        'spynl.plugins': [
            'hello_world = 1',
```

```
        ]
    }
  )
```

This says that *hello_world.py* should be plugged into the Spynl application. What this means is that Spynl calls Pyramid's config.include function, passing the *hello_world* module to it. Therefore, Pyramid expects a function *hello_world.includeme*, which we'll write below.

spynl_plugins is a list, so we could add other modules if that would suit our code organisation in *my-package*. For that matter, we could also add other packages who also define *spynl.plugins* entry points.

And here is *hello_world.py*. We write one endpoint and the registration for it:

```
def hello(request):
    return dict(message="Hello, world!")

def includeme(config):
    config.add_endpoint(hello, 'hello')
```

The *hello* function is a pretty vanilla endpoint. It returns a dictionary. This would mean Spynl returns it as *application/json* (it's default response type), but it could also be served as XML or even YAML (read more about *Serialisation*).

The *includeme* function gets a Pyramid config object, on which we can in principle do everything one can do when writing a pure Pyramid application. We don't need anything but *config.add_endpoint* however, which is actually unique to Spynl (it does some extra magic w.r.t. documentation and route management).

Finally, we develop our package so Spynl knows about it and serve the application:

```
$ python setup.py develop
$ spynl dev.serve
```

(As you see, the *spynl* CLI command works from anywhere when you have your virtual environment activated).

The endpoint http://localhost:6543/hello answers:

```
{
    "status": "ok",
    "message": "Hello, world!"
}
```

Adding documentation for the endpoint

Now let's document the endpoint for frontend developers:

```
def hello(request):
    """
    Say hello to the world.

    ---
    get:
    description: >

        ####Response

        JSON keys | Content Type | Description\n
        --------- | ------------ | -----------\n
```

```
        status    | string | 'ok' or 'error'\n
        message   | string | Hello, world!\n

    tags:
        - my-package
    show-try: true
    """
    return dict(message="Hello, world!")

def includeme(config):
    config.add_endpoint(hello, 'hello')
```

Then, the Swagger doc at http://localhost/about/endpoints actually lists our endpoint:

Spynl Endpoints

A list of all endpoints on this Spynl instance, with a short description on how to use them.

All endpoints usually return application/json, unless otherwise specified here or requested differently by the request. They will have a "status" (ok|error) field and all error responses also will have a "message" field.

about — Show/Hide | List Operations | Expand Operations

contact — Show/Hide | List Operations | Expand Operations

my-package — Show/Hide | List Operations | Expand Operations

GET /hello — Say hello to the world.

Click on the endpoint to see details or try it out:

my-package — Show/Hide | List Operations | Expand Operations

GET /hello — Say hello to the world.

Implementation Notes

This is a simple endpoint used for the basic Spynl tutorial.

Response

JSON keys	Content Type	Description
status	string	'ok' or 'error'
message	string	Hello, world!

Try it out!

We are not using Swagger to its full potential here w.r.t. to its schema capabilities, we know. We chose not to, you can choose otherwise.

Serve on localhost

You already saw how to serve the app:

```
$ spynl dev.serve
```

Getting help about spynl CLI tasks

Now that we begin using the *sspynl* CLI, we should note that for each CLI task, you can get help:

```
$ spynl --help dev.serve
Usage: spynl [--core-opts] dev.serve [other tasks here ...]

Docstring:
Run a local server. The ini-file development.ini is searched for in
installed Spynl plugins. If there is none, minimal.ini is used.

Options:
none
```

Testing the endpoint

Let's write a simple test in *my-package/test_hello.py*:

```
import pytest
from webtest import TestApp
from spynl.main import main

@pytest.fixture(scope="session")
def app():
    spynl_app = main(None)
    return TestApp(spynl_app)

def test_hello(app):
    response = app.get('/hello', status=200)
    assert response.json['message'] == "Hello, world!"
```

Then, we can run:

```
$ spynl dev.test
```

I hope you saw this (the dot says it succeeded):

```
[spynl dev.test] Testing package: my-package
============================== test session starts ===============================
platform linux -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /home/nicolas/workspace/spynl-git/venv/src/my-package, inifile:
plugins: sugar-0.8.0, cov-2.4.0, raisesregexp-2.1
collected 1 items

test_hello.py .
```

Adding translations

Then we have support for translating the app. Let us add a translatable string to the hello_world endpoint:

```
from spynl.main.locale import SpynlTranslationString as _

def hello(request):
    return dict(message=_('hello-msg', default="Hello, world!"))
```

We can now refresh the translation catalogue of our package:

```
$ spynl dev.translate --packages my-package --languages nl --action refresh
[spynl dev.translate] Package: my-package ...
[spynl dev.translate] Creating locale folder ...
running extract_messages
extracting messages from hello_world.py
extracting messages from setup.py
extracting messages from test_hello.py
writing PO template file to ./locale/messages.pot
[spynl dev.translate] File ./locale/nl/LC_MESSAGES/my-package.po does not exist.␣
↪Initializing.
running init_catalog
creating catalog ./locale/nl/LC_MESSAGES/my-package.po based on ./locale/messages.pot
[spynl dev.translate] Done with language nl.
--------------------------------------------------
```

Spynl created all necessary folders and initialised a catalogue. Now a human needs to translate our string to Dutch. Make this change in *my-package/locale/nl/LC_MESSAGES/my-package.po*:

```
#: hello_world.py:23
msgid "hello-msg"
msgstr "Hallo, Wereld!"
```

Then we can compile the catalogue, so that Spynl will serve Dutch when it should:

```
$ spynl dev.translate --packages my-package --languages nl
[spynl dev.translate] Package: my-package ...
[spynl dev.translate] Located locale folder in /home/nicolas/workspace/spynl/venv/src/
↪my-package ...
running compile_catalog
compiling catalog /home/nicolas/workspace/spynl/venv/src/my-package/locale/nl/LC_
↪MESSAGES/my-package.po to /home/nicolas/workspace/spynl/venv/src/my-package/locale/
↪nl/LC_MESSAGES/my-package.mo
[spynl dev.translate] Done with language nl.
--------------------------------------------------
```

There are ony two actions, *refresh* and *compile*. If the *–action* parameter is not given, *spynl dev.translate* compiles.

The compilation step is not necessary and you don't have to include the binary .mo file in your SCM. When we build a Docker image on Jenkins (see below), Jenkins runs the compile action.

we need to tell Pyramid that the new locale directory exists. Add this to the *include_me* function in *my-package/hello_world.py*:

```
config.add_translation_dirs('%s/src/my-package/locale'
                            % os.environ['VIRTUAL_ENV'])
```

Now we want to see our app serve Dutch. We still need to configure the list of languages we serve in our app. This is a great opportunity to start using our own *.ini* file. Copy Spynl's *minimal.ini* to *my-package/development.ini* and add the *spynl.languages* setting in the *[app:main]* section:

```
[app:main]
use = egg:spynl
spynl.pretty = 1
spynl.languages = nl,en
```

It is crucial which language is first in this list. Because *nl* is first, we'll get a Dutch reply from Spynl, e.g.by visiting http://localhost:6543:

FIXME: However, http://localhost:6543/hello still returns english ...

Tab completion for the spynl CLI

Now that we're *spynl* power users, it's time to reveal an important feature: There is tab completion for the *spynl* CLI. To activate it, run

```
$ source $VIRTUAL_ENV/lib/python3.5/site-packages/spynl/spynl/cli/zsh.completion
```

(you might need to adapt the path to spynl, it depends on your environment and method of installation)

You can list (a subset of) tasks by pressing TAB and if the task is complete also the available options. To see options, type a dash ("-") and the press TAB.

This is available for *bash* and *fish* as well, simply replace *zsh* in the command.

Installing the package from SCM

Of course, we will want to use Source Code Management (SCM) for our own code, e.g. on github or bitbucket. *spynl* provides a task called *dev.install* which makes it easy to get started in a new dev environment with developing your app further.

Let's assume your project uses git as SCM and lives in a bitbucket repo:

```
$ spynl dev.install --scm-url git@bitbucket.org:my-team/my-package.git
```

spynl dev.install will clone the code and develop it.

In general, Spynl also supports mercurial repositories.

There are some configuration options here (try *spynl –help dev.install* for all of them). For example, let's assume you work want to work with a feature branch and you want/need to specify in which directory the code should be installed:

```
$ spynl dev.install --scm-url git@bitbucket.org:my-team/my-package.git --revision me/
↪some-feature --src_path path/to/my/virtualenv/src
```

spynl dev.install can also install non-Python dependencies for you or do any other things pre- or post-installation. See *setup.sh.template* in the main Spynl repo. (TODO: point to actual documentation of *setup.sh*).

CHAPTER 3

Short operations tutorial

We show what steps are necessary to get your Spynl-based app built, deployed and smoke-tested.

Building & Deploying Tutorial

(work in progress)

At some point, the *my-package* app should be built, so it can run somewhere else than on localhost:6543. In Spynl, this means starting a job on Jenkins, where the following stages happen:

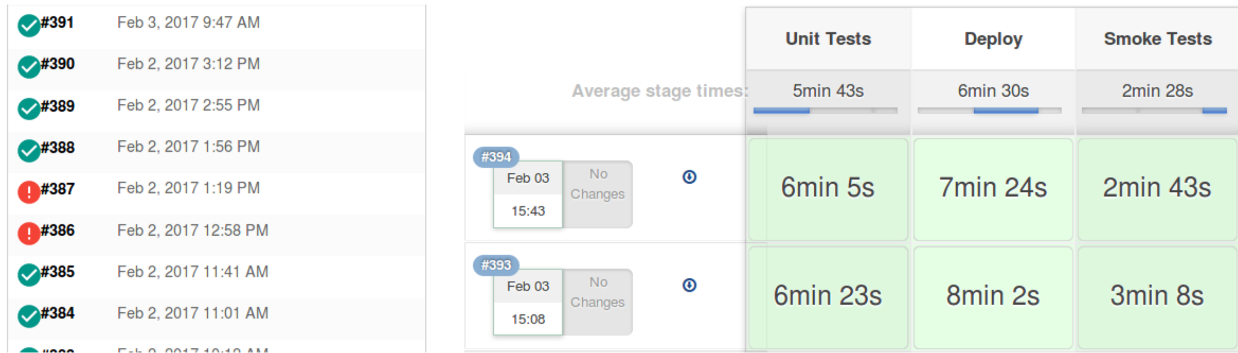
- All tests are run
- A Docker image is built and deployed to your dev container registry
- Smoke tests are run against a freshly-started container based on that tnew image

There is only one *spynl* command necessary here: *spynl dev.start_build*. However, some services need to be set up and configured, namely Jenkins and one or two container registries.

Creating a Jenkins job

Jenkins is an open-source build server. We'll assume in this tutorial that you installed one locally and have it Configuring your Spynl plugin for Deployment running at http://localhost:8080. Jenkins needs Docker engine installed on it, plus aws-utils (?) and any libraries you need for tests to be run.

Spynl uses the Pipeline feature of Jenkins. Here is how Jenkins displays your build history:



If any error happens, your build has failed - the stage is coloured red and the pipeline aborts.

After you have set up a Jenkins server, you need to create a *Pipeline* job called "Spynl". In the Jenkins web interface, navigate to *Jenkins -> New Item -> Pipeline*.

Configure the Pipeline "Definition" to be a *Pipeline script from SCM*, the "SCM" to be *Git* and the "Repository URL" to be *https://github.com/SoftwearDevelopment/spynl.git*. under "Branches to build", add *refs/heads/$spynlbranch*.

Finally you need to add a few String parameters to the job:

- *scm_urls*
- *revision*
- *fallbackrevision* (default: *$revision*)
- *task*
- *spynlbranch*

That's it! Save the Jenkins job.

Configuring your Spynl plugin for Jenkins

Add the following ini-setting to *development.ini*:

```
[app:main]
spynl.ops.jenkins_url: http://localhost:8080
```

Now we can start a build:

```
$ spynl ops.start_build
```

This will build the *master* branch on *spynl* as well as *my-package*. See *spynl –help ops.start_build* for more options.

Configuring your Spynl plugin for Deployment

(TODO: more verbose, this is an outline)

At the moment, Jenkins will be able to build a Docker image, but will fail to push it anywhere.

Set at least these two ini-settings:

- *spynl.ops.ecr.dev_tasks*
- *spynl.ops.ecr.dev_url*

You also need to set the *task* parameter to *spynl ops.start_build*. It needs to be one (or more) of *spynl.ops.ecr.dev_tasks* and a task that exists in your development container service (e.g. in AWS):

```
spynl ops.start_build --task dev
```

Now Jenkins can deploy that image to your development container registration and for that task and restart that task so it will serve your new container.

Configuring your Spynl plugin for Smoke Testing

The third stage in the pipeline is the smoke test. Out of the box, Spynl checks if a container actually exists at the location you want your container service to serve them. Add the following ini-setting:

spynl.ops.dev_url

Spynl also checks if this image has been built within the last 15 minutes.

Your app can specify it's own smoke tests (TODO: write one in my-package)

Deploying to your production environment

(TODO: more verbose)

Set this ini-setting:

```
spynl.ops.ecr.prod_url:
```

Run the *dev.start_build* task

```
spynl ops.start_build --task production
```

TODO: At the moment, Spynl only pushes the new image to that registry. We could make it an ini-setting if Spynl shoiuld try to restart a task there.

CHAPTER 4

In-depth documentation

Shining more light on a few important topics (work in progress):

ini-settings

/about/ini

Spynl checks required settings on startup

Documenting your own settings

```
from spynl.main.docs.settings import ini_doc
my_ini_doc = ...
ini_doc.extend(my_ini_doc)
```

Your production environment(s)

You'll probably have a dev environment and at least one production(-like) environment. Spynl helps you to:

- keep the code consistent between deploys to each of them
- Make sure test do not affect real users and unfinished things are turned off in production
- allow a third party to control your prodcution pipeline

Docker

We use Docker to ship Spynl. You can be sure there that you look at the same code in dev and production. (*/about/versions* can help you to look up precisely which code is in there).

/about/build helps you to see when the Image was made (built on Jenkins) and when it was started.

The image is Ubuntu-based.

Custom pre-install and post-install hooks are possible in *setup.sh* (also works for *dev.install*)

prepare-docker-run.sh can influence *production.ini* or other relevant things in the Docker container right before it is run.

Possibilities to turn off endpoints and whole resources on production

(TODO: add issue about a more generic approach)

Do not send emails to real addresses from non-production environments

TODO: link to the email section

Using different container registries for dev and for production

Useful if you keeo them separated or a thrid party manages your production pipeline (e.g. when using a DTAP approach).

Parameter Handling

Assumption: both GET and POST work (reason for this or scrap it)

all parameters get collected in request[”args”]

Serialisation

JSON is default choice

methods of selecting a different type: Content-Type Header, file extension, ...

supported types: XML, CSV, HTML, YAML

Incoming HTTP data is decoded and outgoing data encoded. Special data type (de)serialisations are easy to add, useful e.g. for DateTime objects.

Translations

Maintaining a translation (i18n) infrastructure is basically a solved problem. But setting it up is tedious and making sure all strings get translated, no matter where they live, can present pitfalls.

Translating a string in Spynl

dev.translate to simplify the workflow

A word about when Spynl translates

Spynl translates everything at the end (so you capture all strings, e.g. also constants)

Routing

Spynl takes control over many aspects of routing.

URLDispatch

(no traversal) default routes /{resource}/{method} /{method}

Endpoint registration

config.add_endpoint

It's custom (do not use Pyramid's *config.add_view*) - why? (at least to have a grip on documentation of endpoints, TODO: look for other reasons)

Custom resource registration

config.add_resource

A resource class at least needs a *paths* attribute.

Multiple paths (aliases) is possible in Spynl.

Custom routes

Basically adding meta data and a function to *spynl.resource_routes_info*. TODO: show tutorial, argue why it is better to do it this way than simple using Pyramid's *config.add_route* directly. Hint: It has to do with applying routes to resources unknown in the current plugin. It has/had a use case but maybe everyone is better off now without it. Research.

HTML emails

Custom base template

(show this in my-package?)

Write a content template

(show this in my-package?)

Non-production email behaviour

Custom get_user_info

A lot of code needs information about the current (authenticated) user, e.g. in endpoints or for logging.

The in-built get_user_info function

Writing your own

```
def my_user_info(request, purpose=None)
    pass # TODO

config.add_settings(user_info_function=my_user_info)
```

Validation per JSON Schema

Aggregation of Logs

NewRelic

spynl.newrelic.key

Sentry

- *spynl.sentry.project*
- *spynl.sentry.key*

Error Handling

Error Views

All HTML 400 errors go through the `error400` error view. All SpynlException errors will go through the `spynl_error` error view. All unexcected errors go trhough the `error500` view and will only get `internal server error` as a message.

SpynlException class

An exception of the `SpynlException` (sub)class will go through the `spynl_error` exception endpoint. The response that is returned is defined in `SpynlExcpetion` and the error view makes sure that the exception is properly logged.

Messages

There are several message types that can be set for a SpynlException in the `__init__`.

The `message` is the message that is intended for the end user, it should be easy to read and not contain sensitive data.

The `developer_message` is intended for third party developers and while it can be technical, it should not contain any sensitive data.

The `debug_message` is meant for debugging, it will not be sent in the response, it will only be logged. It should be a normal string and never a translation string.

Extending the response

If you want to expose more information than just the `message` and the `developer_message` you can extend the response in a subclass.

```
class CustomException(SpynlException):

    def make_response(self):
        response = super().make_response()
        response.update({'custom_info': 'This is a custom response'})
        return response
```

Mapping external Exceptions

You can map external exceptions to internal exceptions, so they raise a SpynlException with a proper error message, instead of resulting in an internal server error.

For this you need to import the external error and register it at an internal error:

```
from external_package import ExternalException

@SpynlException.register_external_exception(ExternalException)
class InternalException(SpynlException):

    def __init__(self):
        message = 'This is a Spynl message'
        super().__init__(message=message)
        self.extra = ''

    def set_external_exception(self, external_exception):
    """ This particular external exception has an entry called extra """
        super().set_external_exception(external_exception)
        self.debug_message = str(self._external_exception)
        self.extra = self._external_exception.extra

    def make_response(self):
    """ To add the extra information to the response, you need to extend it. """
        response = super().make_response()
        response.update({'extra': self.extra})
        return response
```

To be able to use this functionality, you will need to activate the `view_deriver` that handles catching the external errors and the mapping in one of your `includeme()` functions:

```
from spynl.main.exceptions import catch_mapped_exceptions

    def includeme(config):
        # register the view deriver to catch mapped exceptions
        config.add_view_deriver(catch_mapped_exceptions)
```