jivago Documentation

Kento A. Lauzon

Jan 13, 2019

Documentation

Jivago is an object-oriented, highly-reflective Python framework for building web applications. It relies heavily on type annotations and decorators to enforce typing, providing package auto-discovery and dependency injection out of the box. This leads to less boilerplate code, while maintaining loose-coupling across components.

CHAPTER 1

Installation

Jivago and its dependencies can be installed from PyPi. Python3.6 or greater is required.

```
pip install jivago
```

1.1 Virtualenv

Using a virtual environment is recommended for developing and deploying applications.

```
virtualenv -p python3.6 venv
source venv/bin/activate
pip install jivago
pip freeze > requirements.txt
```

CHAPTER 2

Quickstart

A minimal Jivago application is shown below :

```
from jivago.jivago_application import JivagoApplication
from jivago.wsgi.annotations import Resource
from jivago.wsgi.methods import GET


@Resource("/")
class HelloResource(object):

    @GET
    def get_hello(self) -> str:
        return "Hello World!"


app = JivagoApplication()

if __name__ == '__main__':
    app.run_dev()
```

Notice that the example is made up of three separate parts:

- A `Resource` class, which defines a route for our application;
- The `JivagoApplication` object, which contains the application itself;
- A `__main__` function which runs our application in a debug environment, listening on port 4000.

Now, pointing a web browser to `http://localhost:4000` should print our `Hello World!` message.

2.1 Component Auto-discovery

While defining our resource classes in our main file is definitely possible, it can become quite unwieldy. In fact, one of the key goals of the Jivago framework is to maintain loose-coupling of our components.

We will therefore move our resource classes into their own files, and use Jivago's built-in package discovery mechanism to automatically register our routes.

hello_resource.py

```
from jivago.wsgi.annotations import Resource
from jivago.wsgi.methods import GET


@Resource("/")
class HelloResource(object):

    @GET
    def get_hello(self) -> str:
        return "Hello World!"
```

application.py

```
import my_hello_world_application
from jivago.jivago_application import JivagoApplication

app = JivagoApplication(my_hello_world_application)

if __name__ == '__main__':
    from werkzeug.serving import run_simple

    run_simple('localhost', 4000, app)
```

```
my_hello_world_application
 ├── __init__.py
 └── resources
     ├── __init__.py
     └── hello_resource.py
application.py
```

Note that, when creating the JivagoApplication object, a reference to the application's root package is passed as the first argument. The root package should contain *all* Jivago-annotated classes. (i.e. `@Resource`, `@Component`, etc.)

The app object should be outside of the explored package.

Warning : Since all python files are imported at run-time, any lines of code outside a class or a function will be executed before the application is started. It is therefore highly advised to avoid having any line of code outside a declarative block.

2.2 The Resource Class

The resource class is the fundamental way of declaring API routes. To define a route, simply declare the path inside the `@Resource` decorator on the class. Sub-paths can be defined on any of the class' methods using the `@Path` decorator. Allowed HTTP methods have to be explicitly defined for each routing function. Use `@GET`, `@POST`, `@PUT`, `@DELETE`, etc.

Unlike other Python web framework, method invocation relies heavily on type annotations, which resemble the static typing present in other languages like C++ and Java. Given missing parameters, a method will not be invoked and simply be rejected at the framework level. For instance, declaring a route receiving a `dict` as a parameter matches a JSON-encoded request body. `Request` and `Response` objects can be requested/returned, when having direct control over low-level HTTP elements is required.

When resolving string and numeric parameters, path parameters and query parameters are tried. In that case, the key should match the parameter variable name.

A complex resource example

```
from jivago.wsgi.annotations import Resource, Path
from jivago.wsgi.methods import GET, POST


@Resource("/hello")
class HelloWorldResource(object):

    @GET
    def get_hello(self) -> str:
        return "Hello"

    @POST
    @Path("/{name}")
    def post_hello(self, name: str) -> str:
        return "name: {}".format(name)

    @Path("/request/json")
    @POST
    def read_request_body_from_dict(self, body: dict) -> dict:
        return {"the body": body}

    @GET
    @Path("/query")
    def with_query(self, name: str) -> str:
        return "Hello {}!".format(name)
```

While return type annotations are not strictly required, they are nonetheless recommended to increase readability and enforce stylistic consistency.

For manual route registration, see Manual Route Registration.

2.3 Serialization

Jivago supports the definition of *DTO* classes, which can be directly serialized/deserialized. These classes explicitly define a JSON schema and attribute typing, negating the need to use an external schema validator. To define a DTO, use the `@Serializable` decorator :

```
from jivago.lang.annotations import Serializable


@Serializable
class MyDto(object):
    name: str
    age: int
```

If a constructor is declared, it is used when deserializing. Otherwise, each attribute is set using `__setattr__`.

See Serialization for more details.

2.4 Dependency Injection

To allow for modularity and loose-coupling, dependency injection is built into the framework. Resource classes can therefore request dependencies from their constructor.

```
from jivago.lang.annotations import Inject
from jivago.lang.registry import Component
from jivago.wsgi.annotations import Resource


@Component
class CalculatorClass(object):

    def do_calculation(self) -> int:
        return 4


@Resource("/calculation")
class CalculatedResource(object):

    @Inject
    def __init__(self, calculator: CalculatorClass):
        self.calculator = calculator
```

`@Component` is a general-purpose annotation which registers a class to the internal service locator. Whenever a class requires dependencies from their constructor, those get recursively instantiated and injected. Note that the `@Inject` annotation is required.

See Dependency Injection for advanced configurations.

2.5 View Rendering

Jivago also supports rendered HTML views, using the Jinja2 templating engine.

templated_resource.py

```
from jivago.templating.rendered_view import RenderedView
from jivago.wsgi.annotations import Resource
from jivago.wsgi.methods import GET


@Resource("/template")
class TemplatedResource(object):

    @GET
    def get(self) -> RenderedView:
        return RenderedView("my-template.html", {"name": "john"})
```

my-template.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
```

(continues on next page)

(continued from previous page)

```
</head>
<body>
<h1>Hello {{ name }}</h1>
<form method="post">
    <input name="name" />
    <input type="submit">
</form>
</body>
</html>
```

By default, the framework looks for a `views` package directly underneath the root package.

```
my_hello_world_application
 ├── __init__.py
 ├── application.py
 └── views
     ├── __init__.py
     └── my-template.html
```

CHAPTER 3

Reflection

Jivago provides its own *reflection*-style registration mechanism. We will define as *annotations* decorators which do not alter the decorated functions or classes but add a means of programmatically inspecting said decorated functions or classes.

Accessing annotated elements is done by interrogating the *Registry* object. Two types of annotations are defined in Jivago :

- *Annotation*: A general-purpose registering decorator.
- *ParametrizedAnnotation*: Allows the passing of arguments when the annotation is used.

The *Registry* object contains references to all annotated elements, and provides a `get_annotated_in_package` method, which returns all registrations for a specific annotation, for which the package name starts with the given string. Below is an example where all classes with the `@Component` annotation in any package are requested.

```
from jivago.lang.registry import Registry, Component

registry = Registry.INSTANCE

registrations = registry.get_annotated_in_package(Component, "")

for registration in registrations:
    registered_class = registration.registered  # Registered class or function
    annotation_parameters = registration.arguments  # empty dictionary for standard␣
↪annotations
```

3.1 Declaring Custom Annotations

Standard annotations can be defined using either the python-esque *decorator-style syntax* by adding the `@Annotation` decorator to a simple pass-through decorator, or the simpler *object-style syntax* by invoking the Annotation constructor.

```
from jivago.lang.registry import Annotation


# Decorator-style syntax
@Annotation
def MyAnnotation(x: type) -> type:
    return x


# Object-style syntax
MyAnnotation2 = Annotation()


@MyAnnotation
@MyAnnotation2
class MyAnnotatedClass(object):
    pass
```

Parametrized annotations can only defined using the decorator-style syntax. To create a new parametrized annotation, use the `@ParametrizedAnnotation` decorator on a function which returns a pass-through function. (See the example below.)

Unnamed argument will be saved in the dictionary with the declared parameter name as the key.

```
from jivago.lang.registry import ParametrizedAnnotation


@ParametrizedAnnotation
def MyAnnotation(param1: str, param2: str):
    return lambda x: x


@MyAnnotation(param1="foo", param2="baz")
class MyAnnotatedClass(object):
    pass
```

CHAPTER 4

Runnable Components

Jivago provides a mechanism for running background tasks and exposes application initialization hooks. For both of those purposes, the `Runnable` interface is used.

```
from jivago.lang.annotations import Override
from jivago.lang.runnable import Runnable


class MyRunnableComponent(Runnable):

    @Override
    def run(self):
        print("hello!")
```

4.1 Background Workers

For running continuous tasks on a background thread, use the `@BackgroundWorker` annotation. These components will be started on separate threads when the app has started successfully. Components instantiated in this manner support all of the usual dependency injection features.

```
import time

from jivago.lang.annotations import Override, BackgroundWorker, Inject
from jivago.lang.runnable import Runnable


@BackgroundWorker
class MyBackgroundWorker(Runnable):

    @Inject
    def __init__(self, component: MyComponent):
        self.component = component
```

(continues on next page)

(continued from previous page)

```
    @Override
    def run(self):
        while True:
            print("hello from the background")
            time.sleep(5)
```

4.2 Application Initialization Hooks

`@PreInit`, `@Init` and `@PostInit` hooks are provided for running one-off tasks at startup and are invoked identically to background workers. These are, however, required to exit before the application can start.

```
from jivago.config.startup_hooks import PreInit, Init, PostInit
from jivago.lang.annotations import Override
from jivago.lang.runnable import Runnable


@PreInit
class FirstHook(Runnable):

    @Override
    def run(self):
        print("First!")


@Init
class SecondHook(Runnable):

    @Override
    def run(self):
        print("Second!")


@PostInit
class ThirdHook(Runnable):

    @Override
    def run(self):
        print("Third!")
```

- **PreInit** is invoked right after the service locator and application properties are configured.
- **Init** is invoked after initializing the routing table. At this stage, the application is in a coherent state.
- **PostInit** is invoked after starting background workers and scheduled tasks. No further initialization task is left to be done.

4.3 Scheduled Tasks

One-off background tasks can be scheduled over a longer period of time using scheduled tasks. The `@Scheduled` annotation takes either a "cron" or "every" parameter.

- `cron` : Takes a cron-style string.

- `every`: Takes a *Duration* enum. (Duration.SECOND, Duration.MINUTE, Duration.HOUR, Duration.DAY)
- `start` : *Optional*. Specifies a start time before which the task will not be run.

```
from jivago.lang.annotations import Override
from jivago.lang.runnable import Runnable
from jivago.scheduling.annotations import Scheduled, Duration


@Scheduled(every=Duration.HOUR)
class ScheduledTask(Runnable):

    @Override
    def run(self):
        print("hello")
```

CHAPTER 5

Serialization

Jivago provides an `ObjectMapper` object which can be used to serialize and deserialize complex objects. Mapped classes do not need to be annotated with the `@Serializable` annotation.

object_mapper.py

```
from jivago.serialization.object_mapper import ObjectMapper


class Dto(object):
    name: str


object_mapper = ObjectMapper()

dto: Dto = object_mapper.deserialize('{"name": "paul" }', Dto)

json_str = object_mapper.serialize(dto)
```

If a constructor (`__init__`) function is declared on the mapped class, parameters are injected, otherwise parameters are set using the `__setattr__` method.

CHAPTER 6

Manual Route Registration

Additionnal URL routes can be registered by creating a new `RoutingTable` which references classes and their methods. Note that the appropriate classes should be imported beforehand. The referenced resource class can be either an instance, or the actual class. In that case, it will be instantiated by the ServiceLocator, and should therefore be registered manually in the `configure_service_locator` context method.

```
from jivago.wsgi.methods import GET, POST
from jivago.wsgi.routing.tree_routing_table import TreeRoutingTable

my_routing_table = TreeRoutingTable()

my_routing_table.register_route(GET, "/hello", MyResourceClass, MyResourceClass.get_
↪hello)
my_routing_table.register_route(POST, "/hello", MyResourceClass, MyResourceClass.get_
↪hello)
```

This new `RoutingTable` can then be used to configure the `Router` object, which is used to serve all requests. The recommended way of configuring your application is by inheriting from the `ProductionJivagoContext` class, and then overriding the `create_router` method.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.jivago_application import JivagoApplication
from jivago.lang.annotations import Override
from jivago.lang.registry import Registry
from jivago.wsgi.routing.router import Router


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def create_router(self) -> Router:
        router = super().create_router()
        router.add_routing_table(my_routing_table)
        return router
```

(continues on next page)

(continued from previous page)

```
app = JivagoApplication(my_package,context=MyApplicationContext)
```

6.1 Serving static files

While it is not generally recommended to serve static files from a WSGI application for performance reasons, Jivago supports static file serving. The `StaticFileRoutingTable` dynamically defines routes for serving files.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.lang.annotations import Override
from jivago.wsgi.routing.router import Router
from jivago.wsgi.routing.serving.static_file_routing_table import␣
↪StaticFileRoutingTable


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def create_router(self) -> Router:
        router = super().create_router()
        router.add_routing_table(StaticFileRoutingTable("/var/www"))
        router.add_routing_table(StaticFileRoutingTable("/var/www", allowed_
↪extensions=['.html', '.xml']))

        return router
```

The `StaticFileRoutingTable` can also be used with a `allowed_extensions` parameter to explicitly allow or disallow specific file types.

6.2 Defining path prefixes

When registering a new routing table, using the `path_prefix` parameter maps the new routing table to part of the path hierarchy. For instance, static files can be served from `/static/my_file.html`.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.lang.annotations import Override
from jivago.wsgi.routing.router import Router
from jivago.wsgi.routing.serving.static_file_routing_table import␣
↪StaticFileRoutingTable


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def create_router(self) -> Router:
        router = super().create_router()

        router.add_routing_table(StaticFileRoutingTable("/var/www"), "/static")

        return router
```

CHAPTER 7

Application Configuration

Configuration is done using a context class, which defines various methods which can be overridden. The recommended way of defining an application context is by inheriting from either `ProductionJivagoContext` or `DebugJivagoContext`, and overriding specific methods.

```
from typing import List, Type

from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.jivago_application import JivagoApplication
from jivago.wsgi.filter.filter import Filter
from jivago.wsgi.routing.router import Router


class MyApplicationContext(ProductionJivagoContext):

    def configure_service_locator(self):
        super().configure_service_locator()

    def scopes(self) -> List[type]:
        return super().scopes()

    def get_filters(self, path: str) -> List[Type[Filter]]:
        return super().get_filters(path)

    def get_views_folder_path(self) -> str:
        return super().get_views_folder_path()

    def get_config_file_locations(self) -> List[str]:
        return super().get_config_file_locations()

    def create_router(self) -> Router:
        return super().create_router()

    def get_banner(self) -> List[str]:
        return super().get_banner()
```

(continues on next page)

(continued from previous page)

```
app = JivagoApplication(my_package, context=MyApplicationContext)
```

7.1 Configuration methods

configure_service_locator() This method can be used to manually bind classes to the internal ServiceLocator. See Dependency Injection for more details.

scopes() This method defines component scopes for the ServiceLocator which determine when to instantiate new components. By default, only the `Singleton` exists.

get_filters() This method returns a list of Filters which should be applied to a specific request. It is called once for every request.

get_views_folder_path() This method defines the folder in which template files are stored for `RenderedView` responses. Defaults to the `views` submodule of the root package.

get_config_file_locations() Defines a list of files which should be tried when importing the application properties. The `ApplicationProperties` is creating using the first existent file in this rule. Defaults to `["application.yml", "application.json", "properties.yml", "properties.json"]`.

create_router() This method is used to configure the `Router` object which is used to resolve requests.

get_banner() Defines the ASCII-art banner which is printed in the console at every startup.

7.2 ApplicationProperties and SystemEnvironmentProperties

Both the `ApplicationProperties` and `SystemEnvironmentProperties` dictionaries can be injected into a component class, thus providing access namely to the contents of the application config file, and to the environment variables. For instance, for an `application.yml` file placed in the working directory, an appropriate `ApplicationProperties` object is created.

application.yml

```
my_property: "foobar"
```

my_component.py

```
from jivago.config.properties.application_properties import ApplicationProperties
from jivago.lang.annotations import Inject
from jivago.lang.registry import Component


@Component
class MyComponent(object):

    @Inject
    def __init__(self, application_properties: ApplicationProperties):
        self.application_properties = application_properties

    def do_something(self):
        print(self.application_properties["my_property"])
```

CHAPTER 8

Deploying Jivago Applications

Jivago implements the WSGI interface for web applications. Therefore, a WSGI server is required for serving requests. While developing, Werkzeug is the recommended WSGI server, as it is easily started and provides convenient debug features.

```
from jivago.jivago_application import JivagoApplication

app = JivagoApplication()

if __name__ == '__main__':
    # using the bundled werkzeug server
    app.run_dev(port=4000, host="localhost")

    # or alternatively
    from werkzeug.serving import run_simple

    run_simple('localhost', 4000, app)
```

8.1 Running in Production

For production purposes, other WSGI servers are available, such as *gunicorn* and *uwsgi*. See here for a complete deployment example for heroku.