jivago Documentation

Kento A. Lauzon

Dec 01, 2019

Documentation

1 Installation **3**
1.1 Virtualenv . . . 3

2 Quickstart **5**
2.1 Component Auto-discovery . . . 5
2.2 The Resource Class . . . 6
2.3 Serialization . . . 7
2.4 Dependency Injection . . . 8
2.5 View Rendering . . . 8

3 Deploying Jivago Applications **11**
3.1 Running in Production . . . 11

4 Dependency Injection **13**
4.1 Basic Usage . . . 13
4.2 Collections . . . 14
4.3 Scopes . . . 14
4.4 Factory Functions . . . 15
4.5 Manual Component Registration . . . 16
4.6 Service Locator Object . . . 16

5 Reflection **19**
5.1 Declaring Custom Annotations . . . 19

6 Runnable Components **21**
6.1 Background Workers . . . 21
6.2 Application Initialization Hooks . . . 22
6.3 Scheduled Tasks . . . 22

7 Serialization **25**

8 The HTTP Resource Class **27**
8.1 Allowed Parameter Types . . . 28
8.2 Manual Route Registration . . . 28
8.3 Serving static files . . . 29

9 Application Configuration **31**
9.1 Configuration methods . . . 32

9.1.1 Router Configuration . . . 32
9.2 ApplicationProperties and SystemEnvironmentProperties . . . 34

10 Error handling with exception mappers **37**

11 Global Event Bus **39**
11.1 Event payload parameter and handler responses . . . 40
11.2 Event handler types . . . 40
11.3 *Synchronous* vs *Asynchronous* event dispatching . . . 41

12 Functional-style operations using Streams **43**
12.1 Wrapping None values using Nullables . . . 43
12.2 Other examples . . . 43

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 frameworks, 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.

To use query or path parameters, parameters should be declared using the `QueryParam[T]`, `OptionalQueryParam[T]` or `PathParam[T]` typing generics. In this case, *T* should either be `str`, `int` or `float`.

A complex resource example

```
from jivago.wsgi.annotations import Resource, Path
from jivago.wsgi.invocation.parameters import PathParam, QueryParam
from jivago.wsgi.methods import GET, POST, PUT
from jivago.wsgi.request.request import Request
from jivago.wsgi.request.response import Response


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

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

    @POST
    @Path("/{name}")
    def post_hello(self, name: PathParam[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: QueryParam[str]) -> str:
        return "Hello {}!".format(name)

    @GET
    @Path("/request/raw")
    def read_raw_request(self, request: Request) -> Response:
        return Response(200, {}, "body")
```

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):
```

(continues on next page)

(continued from previous page)

```
    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.inject.annotation import Component
from jivago.lang.annotations import Inject
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
```

(continues on next page)

(continued from previous page)

```
    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>
</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

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)
```

3.1 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.

CHAPTER 4

Dependency Injection

Jivago provides a powerful dependency injection engine as a means of implementing inversion of control.

4.1 Basic Usage

Classes annotated with `@Component` or `@Resource` are automatically registered in the built-in service locator. Dependencies are constructor-injected, and require proper typing hints to be used.

```
from jivago.inject.annotation import Component
from jivago.lang.annotations import Inject
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
```

Always make sure that the type hint corresponds exactly to the requested object. (i.e. The type annotation could be used directly as a constructor.) An identically-named, but otherwise different class will not work.

4.2 Collections

Using a collection type hint, all children of a class can be requested. Take a look at the following example :

```
import random
from typing import List

from jivago.inject.annotation import Component
from jivago.lang.annotations import Override, Inject


class Calculator(object):

    def do_calculation(self, input: int) -> int:
        raise NotImplementedError


@Component
class ConstantCalculator(Calculator):

    @Override
    def do_calculation(self, input: int) -> int:
        return 5


class RandomCalculator(Calculator):

    @Override
    def do_calculation(self, input: int) -> int:
        return random.randint(0, 100)


@Component
class CalculationService(object):

    @Inject
    def __init__(self, calculators: List[Calculator]):
        self.calculators = calculators

    def calculate(self, input: int) -> List[int]:
        return [calculator.do_calculation(input) for calculator in self.calculators]
```

The *CalculationService* class is injected with a list of all components which implement the *Calculator* interface.

4.3 Scopes

By default, all components are re-instantiated when a request is received. However, a `@Singleton` annotation is provided for when unicity is important. (e.g. when making a simple persistence mechanism held in memory.)

```
from typing import List

from jivago.inject.annotation import Component, Singleton


@Component
```

(continues on next page)

(continued from previous page)

```
@Singleton
class InMemoryMessageRepository(object):

    def __init__(self):
        self.content = []

    def save(self, message: str):
        self.content.append(message)

    def get_messages(self) -> List[str]:
        return self.content
```

A *singleton* component will be instantiated when it is first requested, and reused for subsequent calls.

4.4 Factory Functions

When complex scoping is required for a given component, for example when handling a database connection, factory functions can be used to instantiate and cache components using the `@Provider` annotation. In this case, the return type hint defines the class to which the function is registered.

```
from jivago.inject.annotation import Provider, Singleton


class DatabaseConnection(object):

    def __init__(self):
        # open a connection, etc.
        pass

    def query_database(self) -> int:
        # use the opened connection, etc.
        return 5


connection = None


@Provider
def get_database_connection() -> DatabaseConnection:
    global connection
    if connection is None:
        connection = DatabaseConnection()
    return connection


@Provider
@Singleton
def get_singleton_bean(my_dependency: Dependency) -> MySingletonBean:
    # Will only be called once
    return Dependency(...)
```

The provider function can take any registered component as arguments. By adding `@Singleton` to the provider function, it will be lazily instantiated only once, thereby exhibiting the same behaviour as components.

4.5 Manual Component Registration

When fine-tuned control is necessary, the service locator should be manually configured by extending the *Context* object. In order to do so, first override either `ProductionJivagoContext` or `DebugJivagoContext`. This will be your new application context, which should be passed to the JivagoApplication object. The `configure_service_locator` is where component registration is done. Use the `self.serviceLocator.bind` method to manually register components. Note that Jivago decorators will not be taken into consideration when using manual component registration.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.lang.annotations import Override


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def configure_service_locator(self):
        super().configure_service_locator()
        self.serviceLocator.bind(MessageRepository, InMemoryMessageRepository)
```

The `bind(interface, implementation)` methods registers an **implementation** to its **interface**. The service locator acts as a dictionary, where the *interface* is the key, and the *implementation* is the value. The interface should always be a class.

The *implementation* can be any of the following :

- A class
- An instance of a class
- A function which, when called, returns an instance of a class

When a class is given, the default behaviour is applied : a new instance is created whenever the interface is requested. Registering an instance of the class causes it to act as a singleton. Finally, a registered function will be invoked whenever the interface class is requested.

4.6 Service Locator Object

Similarily, components can be manually requested by directly invoking the *ServiceLocator* object. A reference to the *ServiceLocator* object can be obtained either through dependency injection, or statically.

```
from jivago.config.abstract_context import AbstractContext
from jivago.inject.annotation import Component
from jivago.inject.service_locator import ServiceLocator
from jivago.lang.annotations import Inject


@Component
class Calculator(object):
    def do_calculation(self) -> int:
        return 5

# ServiceLocator injection
@Component
class CalculationService(object):
```

(continues on next page)

(continued from previous page)

```
    @Inject
    def __init__(self, service_locator: ServiceLocator):
        self.service_locator = service_locator
        self.calculator = self.service_locator.get(Calculator)


# Static access to the ServiceLocator object from anywhere
def calculate() -> int:
    service_locator = AbstractContext.INSTANCE.service_locator()
    calculator = service_locator.get(Calculator)
    return calculator.do_calculation()
```

The service locator has `get` and `get_all` methods for requesting components.

CHAPTER 5

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.inject.annotation import Component
from jivago.lang.registry import Registry

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
```

5.1 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 6

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!")
```

6.1 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)
```

6.2 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.

6.3 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 7

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 8

The HTTP Resource Class

Jivago uses *Resource* classes to define HTTP routes. Routable classes should be annotated with the `@Resource` decorator with an URL path on which it should be mounted. Each routing *method* should then be annotated with one of the HTTP verbs (`@GET`, `@POST`, etc.) to make them known to the framework, and can optionally define a subpath using the `@Path` annotation.

```
from jivago.wsgi.annotations import Resource, Path
from jivago.wsgi.invocation.parameters import PathParam, QueryParam
from jivago.wsgi.methods import GET, POST, PUT
from jivago.wsgi.request.request import Request
from jivago.wsgi.request.response import Response


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

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

    @POST
    @Path("/{name}")
    def post_hello(self, name: PathParam[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: QueryParam[str]) -> str:
        return "Hello {}!".format(name)
```

(continues on next page)

(continued from previous page)

```
    @GET
    @Path("/request/raw")
    def read_raw_request(self, request: Request) -> Response:
        return Response(200, {}, "body")
```

8.1 Allowed Parameter Types

When handling a specific request, Jivago reads declared parameter types before invoking the routing function. Passed arguments can come from the query itself, from the body, from the raw request or a combination. Below are all the allowed parameter types :

- *QueryParam[T]* : Reads the parameter *matching the variable name* from the query string. *T* should be either `str`, `int` or `float`.
- *OptionalQueryParam[T]* : Identical to the above, except that it allows `None` values to be passed in place of a missing one.
- *PathParam[T]* : Reads the parameter from the url path. The variable name should match the declared name in the `@Path` or the `@Resource` annotation. Route definitions use the `{path-parameter-name}` to declare these parameters.
- *dict* : The request body which has been deserialized to a dictionary. Requires the body to be deserializable to a dictionary. (e.g. JSON).
- A user-defined DTO : Any declared `@Serializable` class will be instantiated before invoking. This effectively acts as a JSON-schema validation.
- *Request* : The raw `Request` object, as handled by Jivago. Useful when direct access to headers, query strings or the body is required.
- *Headers* : The raw `Headers` object, containing all request headers. This class is simply a case-insensitive dictionary.

8.2 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.table.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_config` method.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.config.router.router_builder import RouterBuilder
from jivago.jivago_application import JivagoApplication
from jivago.wsgi.routing.routing_rule import RoutingRule


class MyApplicationContext(ProductionJivagoContext):

    def create_router_config(self) -> RouterBuilder:
        return super().create_router_config() \
            .add_rule(RoutingRule("/", my_routing_table))


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

8.3 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.config.router.router_builder import RouterBuilder
from jivago.lang.annotations import Override
from jivago.wsgi.routing.routing_rule import RoutingRule
from jivago.wsgi.routing.serving.static_file_routing_table import␣
↪StaticFileRoutingTable


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def create_router_config(self) -> RouterBuilder:
        return super().create_router_config() \
            .add_rule(RoutingRule("/", StaticFileRoutingTable("/var/www"))) \
            .add_rule(RoutingRule("/", StaticFileRoutingTable("/var/www", allowed_
↪extensions=['.html', '.xml'])))
```

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

Additional router configuration options, including specific filter and CORS rules, can be found at Router Configuration.

CHAPTER 9

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

from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.config.router.router_builder import RouterBuilder
from jivago.event.event_bus import EventBus
from jivago.jivago_application import JivagoApplication
from jivago.lang.registry import Registry


class MyApplicationContext(ProductionJivagoContext):

    def __init__(self, root_package: "Module", registry: Registry, banner: bool =␣
↪True):
        super().__init__(root_package, registry, banner)

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

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

    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_config(self) -> RouterBuilder:
        return super().create_router_config()
```

(continues on next page)

(continued from previous page)

```
    def get_default_filters(self):
        return super().get_default_filters()

    def create_event_bus(self) -> EventBus:
        return super().create_event_bus()

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


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

9.1 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_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_config() This method is used to configure the `Router` object which is used to resolve requests. See Router Configuration for details.

get_default_filters() Used only on subclasses of `ProductionJivagoFilter`. This method is called from `create_router_config` to define the default `FilteringRule`.

create_event_bus() Used to register event handlers.

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

9.1.1 Router Configuration

The request router is configured in the `create_router_config()` method of your application context.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.config.router.router_builder import RouterBuilder
from jivago.wsgi.routing.routing_rule import RoutingRule
from jivago.wsgi.routing.serving.static_file_routing_table import␣
↪StaticFileRoutingTable


class MyApplicationContext(ProductionJivagoContext):

    def create_router_config(self) -> RouterBuilder:
        return super().create_router_config() \
            .add_rule(RoutingRule("/static", StaticFileRoutingTable("/var/www")))
```

Configuration itself is done by adding new rules to the router builder, using the `add_rule` method.

Routing rules

Each routing rule requires a prefix path which acts as a root path from which requests are served, and a `RoutingTable` which contains the actual route definitions. The `rewrite_path` parameter defaults to `True` and is used to remove the path prefix from the request object before invoking the resource class.

```
from jivago.wsgi.methods import GET
from jivago.wsgi.routing.routing_rule import RoutingRule
from jivago.wsgi.routing.serving.static_file_routing_table import␣
↪StaticFileRoutingTable
from jivago.wsgi.routing.table.tree_routing_table import TreeRoutingTable

my_routing_table = TreeRoutingTable()
my_routing_table.register_route(GET, "/hello", HelloClass, HelloClass.get_hello)

root_rule = RoutingRule("/", my_routing_table)
my_rule = RoutingRule("/static", StaticFileRoutingTable("/var/www"), rewrite_
↪path=True)
```

Example : Mapping all routes to the `/api` prefix To completely override the default *production* or *debug* configuration, omit the *super()* call, and start with a fresh `RouterBuilder`.

```
from jivago.config.production_jivago_context import ProductionJivagoContext
from jivago.config.router.filtering.auto_discovering_filtering_rule import␣
↪AutoDiscoveringFilteringRule
from jivago.config.router.filtering.filtering_rule import FilteringRule
from jivago.config.router.router_builder import RouterBuilder
from jivago.lang.annotations import Override
from jivago.wsgi.routing.routing_rule import RoutingRule
from jivago.wsgi.routing.table.auto_discovering_routing_table import␣
↪AutoDiscoveringRoutingTable


class MyApplicationContext(ProductionJivagoContext):

    @Override
    def create_router_config(self) -> RouterBuilder:
        return RouterBuilder() \
            .add_rule(FilteringRule("*", self.get_default_filters())) \
            .add_rule(AutoDiscoveringFilteringRule("*", self.registry, self.root_
↪package_name)) \
            .add_rule(RoutingRule("/api", AutoDiscoveringRoutingTable(self.
↪registry, self.root_package_name)))
```

Note the required default rules for proper operation :

- **`FilteringRule('*', self.get_default_filters())`** This rule adds all Jivago filters which are required for proper error and serialization handling.
- **`AutoDiscoveringFilteringRule('*', self.registry, self.root_package_name)`** This rule registers user-defined request filters using the *@RequestFilter* annotation.
- **`RoutingRule('/', AutoDiscoveringRoutingTable(self.registry, self.root_package_na`** This is where the reflectively declared routes are registered. Without this rule, `@Resource`, `@GET`, `@POST`, `...` annotations will not be parsed. Edit the prefix path to your liking.

Filtering rules

While additional request filters can be added to all requests by using the `@RequestFilter` annotation, specific filtering rules can be added to apply filters to specific routes only. The `FilteringRule` rule uses a path pattern which is used to select which filters to apply on any given incoming request. The pattern can either be given using a simple *-style wildcard, or using a regexp pattern.

```
from jivago.config.router.filtering.filtering_rule import FilteringRule

# Applies to all subpaths of "/users/". DOES NOT apply to "/users" itself.
FilteringRule("/users/*", [MyFilter])

# Only applies to "/users", and nothing else.
FilteringRule("/users", [MyFilter])

# Applies to "/users" and subpaths of "/users/...".
FilteringRule("/users*", [MyFilter])

# Applies to all requests matching regexp.
# First parameter is not used when regex_pattern is supplied.
FilteringRule(None, [MySpecialFilter], regex_pattern=r"^/users.*$")
```

Note that the simple URL pattern parameter is ignored when a regular expression is supplied.

CORS rules

CORS preflight behaviour can be tuned using CORS rules. The supplied prefix is used to define different behaviours for different sub-paths. The CORS rule does NOT support fuzzy pattern matching like the filtering rule. When multiple rules are applicable to an incoming request, only the longest one is applied.

```
from jivago.config.router.cors_rule import CorsRule

# Applies to all requests
CorsRule("/", {"Access-Control-Allow-Origin": '*'})

# Applies to all requests on a path which starts with '/users'
CorsRule("/users", {"Access-Control-Allow-Origin": 'api.example.com'})
```

By default, using the ‘‘DebugJivagoContext‘‘ adds a `Access-Control-Allow-Origin: *` rule at the root of the route hierarchy.

9.2 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.inject.annotation import Component
from jivago.lang.annotations import Inject


@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 10

Error handling with exception mappers

For basic error handling, Jivago provides an `ExceptionMapper` mechanism, which can automatically create an HTTP response for an uncaught exception. To define a custom exception mapper, create a component which inherits from the `ExceptionMapper` interface, implementing the *handles* and *create_response* methods.

```
from jivago.inject.annotation import Component
from jivago.lang.annotations import Override
from jivago.wsgi.filter.system_filters.error_handling.exception_mapper import␣
↪ExceptionMapper
from jivago.wsgi.request.response import Response


@Component
class TeapotExceptionMapper(ExceptionMapper):

    @Override
    def handles(self, exception: Exception) -> bool:
        return exception == TeapotException

    @Override
    def create_response(self, exception: Exception) -> Response:
        return Response(418, {}, "Error! I am a teapot!")
```

- *handles(exception) -> bool* : Used to find the corresponding exception mapper.
- *create_response(exception) -> Response* : Creates the actual HTTP response.

CHAPTER 11

Global Event Bus

For event-driven programming purposes, Jivago provides a simple `EventBus` interface which can be used from anywhere to trigger events and dispatch messages. This approach has the benefit of completely decoupling the *caller* from the *callee(s)*, which can be beneficial in some large-scale applications. Take a look at the following code snippet :

```
from jivago.event.config.annotations import EventHandler
from jivago.event.event_bus import EventBus
from jivago.lang.annotations import Override, Inject
from jivago.lang.runnable import Runnable
from jivago.wsgi.annotations import Resource
from jivago.wsgi.methods import POST


@EventHandler("Player:Death")
class PlayerDeathEventHandler(Runnable):

    @Override
    def run(self):
        print("Oh dear! You are dead!")


@Resource("/kill-player")
class PlayerResource(object):

    @Inject
    def __init__(self, event_bus: EventBus):
        self.event_bus = event_bus

    @POST
    def kill_player(self) -> str:
        self.event_bus.emit("Player:Death")
        return "OK"
```

In this example, a call to `/kill-player` triggers an event named `Player:Death`, thereby invoking all known handlers.

11.1 Event payload parameter and handler responses

The `emit` and handler methods accepts zero or one argument. In the event (*pun intended*) that said *payload* parameter is supplied while emitting the event, it shall be passed to all handlers which require it. Note that only a single payload parameter is allowed.

```
from jivago.event.config.annotations import EventHandler
from jivago.event.event_bus import EventBus


@EventHandler("event")
def handle_event(payload) -> str:
    return "ok"

...
event_bus: EventBus
event_bus.emit("event", {"key": "value"})
...
```

Should event handlers return something, responses are returned to the caller in the form of a tuple containing all non-nil responses. Note that this behaviour is not applicable when using the asynchronous (*async*) event bus.

11.2 Event handler types

Event handlers can be implemented using any of the usual ways. Functions, methods and runnable classes are allowed. Since *EventHandlerClass* and *Runnable* objects are instantiated by the service locator, constructor injection is supported as usual.

```
from jivago.event.config.annotations import EventHandler, EventHandlerClass
from jivago.lang.annotations import Override
from jivago.lang.runnable import Runnable


@EventHandler("event")
def handler_function():
    pass


@EventHandlerClass
class HandlerClass(object):

    @EventHandler("event")
    def handler_method(self):
        pass


@EventHandler("event")
class HandlerRunnable(Runnable):

    @Override
    def run(self):
        pass
```

11.3 *Synchronous* vs *Asynchronous* event dispatching

By default, all events are handled **synchronously**, i.e. on the caller thread. Therefore, calling *EventBus.emit()* will only return once all handlers have been called. When *asynchronicity* is desired, events should be emitted using the `AsyncEventBus`.

```
from jivago.event.async_event_bus import AsyncEventBus
from jivago.lang.annotations import Inject
from jivago.wsgi.annotations import Resource
from jivago.wsgi.methods import POST


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

    @Inject
    def __init__(self, event_bus: AsyncEventBus):
        self.event_bus = event_bus

    @POST
    def send_hello(self) -> str:
        self.event_bus.emit("hello")
        return "Hello has been sent to all listeners!"
```

Unlike the usual *EventBus.emit()*, *AsyncEventBus.emit()* returns immediately and returns nothing. Events are then dispatched by a thread pool executor, which can be configured in your application context.

CHAPTER 12

Functional-style operations using Streams

Jivago provides a `Stream` class which can be used to perform functional-style operations on collections. This mechanism is heavily inspired by *Java*'s identically named *Streams*.

```
from jivago.lang.stream import Stream

# Result : [4, 16, 36]
square_of_even_numbers = Stream([1, 2, 3, 4, 5, 6]) \
    .filter(lambda x: x % 2 == 0) \
    .map(lambda x: x ** 2) \
    .toList()
```

In the previous example, the usual *filter* and *map* operations are used in a sequential manner. While all these operations are all available in Python, using Jivago's *Streams* allows the chaining of these operations to improve readability.

All available functions are documented in jivago.lang.stream.

12.1 Wrapping None values using Nullables

Jivago provides `Nullable` objects which are used to wrap None items. This class follows the same structure as *Java*'s `Optional` class. Terminal `Stream` operations which return a single item use this mechanism.

All available functions are documented in jivago.lang.nullable.

12.2 Other examples

```
from jivago.lang.stream import Stream

first_ten_square_numbers = Stream.range() \
    .map(lambda x: x ** 2) \
    .take(10)
```

(continues on next page)

(continued from previous page)

```
# [1, 4, 9, 16, 25, ...]


Stream.zip(['a', 'b', 'c'], [1, 2, 3]) \
    .forEach(lambda letter, number: print(f"{letter} is the {number}th letter of the␣
↪alphabet."))

Stream.of(1, 2, 3, 4).allMatch(lambda x: x < 10)
# True


def square(x: int) -> int:
    return x ** 2


squares = Stream.of(1, 2, 3, 4).map(square).toList()
# [1, 4, 9, 16]

alphabet = Stream([(1, 'a'), (2, 'b'), (3, 'c')]).toDict()
# { 1 : 'a', ...}
```

The automatically generated documentation is available at Automatically-generated documentation.