ddd-for-python

Release 0.9.5

David Runemalm

Mar 13, 2022

USER GUIDE

Welcome to the ddd-for-python framework.

This framework is used to do DDD with python. Below is an example of a bounded context implemented with ddd-for-python.

Check out the *user guide* to get started building your own contexts.

CHAPTER

ONE

EXAMPLE

```
from ddd.application.config import Config
from ddd.infrastructure.container import Container

from shipping.utils.dep_mgr import DependencyManager
from shipping.application.shipping_application_service import \
ShippingApplicationService


if __name__ == "__main__":

    # Config
    config = Config()

    # Dependency manager
    dep_mgr = \
        DependencyManager(
            config=config,
        )

    # Application service
    service = \
        ShippingApplicationService(
            customer_repository=dep_mgr.get_customer_repository(),
            db_service=dep_mgr.get_db_service(),
            domain_adapter=dep_mgr.get_domain_adapter(),
            domain_publisher=dep_mgr.get_domain_publisher(),
            event_repository=dep_mgr.get_event_repository(),
            interchange_adapter=dep_mgr.get_interchange_adapter(),
            interchange_publisher=dep_mgr.get_interchange_publisher(),
            job_adapter=dep_mgr.get_job_adapter(),
            job_service=dep_mgr.get_job_service(),
            log_service=dep_mgr.get_log_service(),
            scheduler_adapter=dep_mgr.get_scheduler_adapter(),
            shipment_repository=dep_mgr.get_shipment_repository(),
            max_concurrent_actions=config.max_concurrent_actions,
            loop=config.loop.instance,
        )

    # ..register
    dep_mgr.set_service(service)
```

(continues on next page)

(continued from previous page)

```
    # Container
    container = \
        Container(
            app_service=service,
            log_service=dep_mgr.get_log_service(),
        )

    # ..run
    loop = config.loop.instance
    loop.run_until_complete(container.run())
    loop.close()
```

1.1 Installation

Install using pip:

```
$ pip install ddd-for-python
```

1.2 Example application

We have included an example application in the source distribution, so that you can see a full example of a bounded context implementing using this framework.

You can find this webshop application in the github repository of this project.

Note: The example application is incomplete and will be finished before the release of v1.0.0.

1.3 Building Blocks

These are the building blocks of the framework:

- *Config*
- *Env files*
- *Dependency manager*
- *main.py*
- *Container*
- *Makefile*

The *Config* object will be accessible from anywhere in your code. This object will contain all the configuration settings of your bounded context. Use the *Env files* to define these settings for each of the environments your context will run in. These environments will typically be local, dev, staging and production, or any subset of them. By utilizing *Env files*, you can avoid the "configuration hell" where settings are scattered across different locations, and/or hard-coded

into the application code, making it impossible to configure the context for different environments without a new code release.

The config object will be registered with the *Dependency manager*. This is how you will be able to access it from all over the code. The config objects is not the only dependency you will register with the *Dependency manager*. You can register any depdency that you want to be able to reference from anywhere in the code. The dependency manager brings the inversion of control principle and the dependency injection pattern to the framework.

The creation of dependencies are done in the *main.py* file. Here they are also registered with the Dependency Manager. The final step in main.py is to instantiate a *Container* object and schedule it to run on the python event loop.

The *Makefile* is used to automate your daily tasks so that you can run them easily and quickly from the command line. These tasks are e.g. building, running the unit tests, running your deployment pipeline locally, etc. You can add your own makefile targets to this file as you wish, but there are some standard tasks that follows with the project template setup. Check out the shipping makefile example for inspiration.

1.3.1 Config

The `Config` object holds all the configuration settings of your bounded context .

If you don't have custom settings added to your env file, you can simply instantiate a config object from the base `Config` class. If you do have custom settings however, which you typically do, you need to subclass the base `Config` and override a couple of methods to instruct `Config` where to find the new settings in the env file and where in the `Config` object to store them.

This is an example of how you subclass the base Config class for the shipping context:

Start by creating `<your_domain>.application.config`:

```
touch <your_product>/<your_domain>/application/config.py
```

Then open `config.py` in your favourite text editor:

```
subl <your_product>/<your_domain>/application/config.py
```

Add the class declaration:

```
from ddd.application.config import Config as BaseConfig


class Config(BaseConfig):
    def __init__(self):
        super().__init__()
```

Then override `_declare_settings()` to declare the new settings:

```
def _declare_settings(self):
    self.my_custom_setting = None
    super()._declare_settings()
```

Override `_read_config()` to define which environment variables that contains the new settings:

```
def _read_config(self):
    self.my_custom_setting = os.getenv('MY_CUSTOM_SETTING')
```

Note: `Config` knows how to find the env file and read it's settings as long as it's placed in the root of the project and named "env". If you want another name for your env file, you must pass the path using the `env_file_path` argument of the constructor.

Now you can reference your custom settings from the config object like so:

```
config = dep_mgr.get_config()

my_custom_setting = config.my_custom_setting

print("My custom setting:", my_custom_setting)
```

1.3.2 Env files

An Env file is where you put the configuration for a specific environment.

You will e.g. have the following env files, one each for all of your environments:

- env.local.pycharm
- env.local.minikube
- env.local.test
- env.pipeline.test
- env.staging
- env (production)

Env files are part of The Twelve-Factor App pattern.

1.3.3 Dependency manager

The dependency manger applies the dependency injection pattern and inversion of control (IoC) principle.

This pattern is useful when you want to use mock- repositories and third-party api adapters in your tests, while you use the real (mysql, http, etc.) dependencies in your production environment.

To support a new dependency in the manager, you need to subclass the base Dependency Manager and add the private variable that will hold the new dependency, as well the getters and setters to register and retrieve it.

1.3.4 main.py

As previously mentioned, the `main.py` file can be seen as the starting point for your code that will execute in the container and implement the bounded context.

That means this file will instantiate all the building blocks that comprises the context and then schedule it to run on the event loop.

This is how the shipping example main.py file looks like::

```
from ddd.application.config import Config
from ddd.infrastructure.container import Container

from shipping.utils.dep_mgr import DependencyManager
from shipping.application.shipping_application_service import \
ShippingApplicationService


if __name__ == "__main__":
    """
    This is the container entry point.
    Creates the app and runs it in the container.
    """

    # Config
    config = Config()

    # Dependency manager
    dep_mgr = \
        DependencyManager(
            config=config,
        )

    # Application service
    service = \
        ShippingApplicationService(
            customer_repository=dep_mgr.get_customer_repository(),
            db_service=dep_mgr.get_db_service(),
            domain_adapter=dep_mgr.get_domain_adapter(),
            domain_publisher=dep_mgr.get_domain_publisher(),
            event_repository=dep_mgr.get_event_repository(),
            interchange_adapter=dep_mgr.get_interchange_adapter(),
            interchange_publisher=dep_mgr.get_interchange_publisher(),
            job_adapter=dep_mgr.get_job_adapter(),
            job_service=dep_mgr.get_job_service(),
            log_service=dep_mgr.get_log_service(),
            scheduler_adapter=dep_mgr.get_scheduler_adapter(),
            shipment_repository=dep_mgr.get_shipment_repository(),
            max_concurrent_actions=config.max_concurrent_actions,
            loop=config.loop.instance,
        )

    # ..register
    dep_mgr.set_service(service)

    # Container
    container = \
        Container(
            app_service=service,
            log_service=dep_mgr.get_log_service(),
        )

    # ..run
```

(continues on next page)

(continued from previous page)

```
    loop = config.loop.instance
    loop.run_until_complete(container.run())
    loop.close()
```

Note: The Container will listen to UNIX stop signals (e.g. by a user pressing Ctrl+C in the terminal, or by the Docker Engine stopping the container, for any reason). Upon receiving such a stop signal, it gracefully shuts down the context by first calling `stop()` on the ApplicationService, which in turns calls `stop()` on all the secondary- and primary adapters (in that order). The Container task is then taken off the event loop and the docker container can be destroyed by the orchestrator.

1.3.5 Container

The container abstraction maps directly to the docker/kubernetes container concept. It doesn't do much more than function as a holder of the context's application service and delegates the stop operations upon receiving unix stop signals when the container is instructed to stop by whatever container orchestrator is operating it.

1.3.6 Makefile

This part of the documentation will be added before the release of v1.0.0.

1.4 Scheduling

This part of the documentation will be added before the release of v1.0.0.

1.5 Migrations

This part of the documentation will be added before the release of v1.0.0.

1.6 Testing

This part of the documentation will be added before the release of v1.0.0.

1.7 Development Process

You can of course develop any way you want, however, this is the process that ddd-for-python (DDD) is especially designed to work well with:

1. **Clone the repository:**
    - Run in terminal:

```
$ git clone https://github.com/runemalm/ddd-for-python.git <cloned_repo_
↪path>
```

2. **Create your project from the template:**

    - Run in terminal:

```
$ cd <cloned_repo_path>
$ cp templates/project <your_projects_path>
$ cd <your_projects_path>
$ mv project <your_product>  # Renames the 'project' folder
$ cd <your_product>
$ mv domain <your_domain>    # Renames the 'domain' folder
```

3. **Create a virtual environment**

    - Run in terminal:

```
$ cd <your_projects_path>/<your_product>/<your_domain>
$ make create-venv
```

4. **Implement your domain model:**

    - Create the application service
    - Create the actions
    - Create the aggregates, entities and value objects
    - Create the repositories
    - Create the primary adapters (e.g. http adapter and event listeners)
    - Create the secondary adapters for your external APIs
    - Write the tests

5. **Deploy the first version:**

    - When your tests pass and you are ready for a first release, deploy your new bounded context using kubernetes, docker or any other orchestrator.

6. **Iterate on features:**

    - Continuously iterate on your domain model and product features from here on.

Note: We rely on the community to come up with more in-depth guides on how to develop with the framework, e.g. how to setup PyCharm or other IDEs and editors.

Tip: If you have a guide you think should be included in this documentation, please submit it to us.

1.8 Troubleshooting

If you suspect something in the ddd package isn't as expected, it will be helpful to increase the logging level of the `ddd-for-python` logger to the `DEBUG` level.

If you do not yet have logging enabled in the first place, you can do this:

```
import logging

logging.basicConfig()
logging.getLogger('ddd-for-python').setLevel(logging.DEBUG)
```

This should provide lots of useful information about what's going on inside the ddd core.

1.9 Version history

0.9.5

- Added documentation.
- Moved db_service related classes.
- Moved event related classes.
- Added `MemoryPostgresDbService` to be able to run tests against an in-memory postgres database.
- Fixed bug: container kwarg in example main.py (thanks euri10).

0.9.4

- Added `context` to log service's log messages.
- Moveed record filtering methods to base repository class.
- Added `uses_service` to Task class. Deprecate `makes_requests`.

0.9.3

- Searching env file from cwd by default in tests, (when no path specified).
- Refactored Task class to make it more simple.
- Refactored the configuration solution by adding a Config class.
- Added example code for `shipping` context of a webshop application.
- Added get_all_jobs() and get_job_count() to scheduler adapter & service.
- Added missing call to _migrate() in a couple of Repository class functions.

0.9.2

- Fixed bug: Env file wasn't loaded in certain circumstances.

0.9.1

- Initial commit.

1.10 Contribution

You are welcome to suggest changes and to submit bug reports at the github repository.

We also look forward to seeing user guides as this project is being adopted.

1.11 API reference