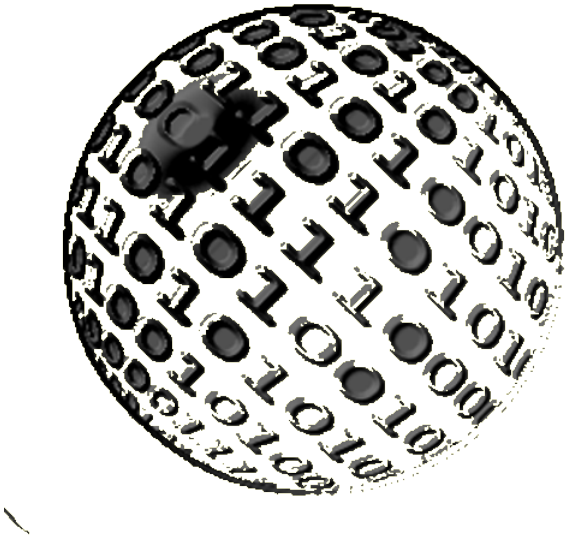
fastapi-serviceutils

Release 2.0.0

Simon Kallfass

Dec 07, 2019

CONTENTS



Services stand for **portability** and **scalability**, so the **deployment** and **configuration** of these service should be as easy as possible. To achieve this a service based on fastapi-serviceutils is configured using a `config.yml`. These settings can be overwritten using **environment-variables**. Dependency management for these services is generalized using a combination of Dephell and Poetry.

For monitoring and chaining of service-calls some default endpoints should always be defined. For example an endpoint to check if the service is alive (`/api/alive`) and an endpoint to access the config of the service (`/api/config`). These endpoints are automatically added in services using fastapi-serviceutils if defined in the `config.yml` of the service.

Because a service should focus on only one task it may be required to create multiple small services in a short time. As always time matters. For this fastapi-serviceutils allows ** fast creation of new services** with `create_service`.

If an error occurs during a service-call it is important to have **detailed logs** with a **good traceback**. To achieve this the default logging of fastapi is optimized in fastapi-serviceutils using `loguru`.

Fastapi allows easily created **swagger-documentation** for service-endpoints. This is optimal for clients wanting to integrate these endpoints. For developers of the service an additional **apidoc-documentation** of the service and the source-code is required (most popular are documentations created using Sphinx or MKDocs). Fastapi-serviceutils based services **serve sphinx-based documentations** using **google-documentation style** in the code and rst-files inside the docs-folder.

The development of these services should be as much generalized as possible for easy workflows, as less manual steps as possible for the developer and short onboarding times. For this fastapi-serviceutils includes a Makefile for most common tasks during development. There is also a Tmuxp-config file to create a tmux-session for development.

CHAPTER

ONE

FEATURES

- **optimized logging** using Loguru
- **optimized exception handling** by additional exception handler `log_exception handler`
- usage of a **config.yml**-file to configure the service
- usage of **environment-variables** (Environment variable overwrites config-value) to configure the service
- easily **serve the apidoc** with the service
- easy deploment using Docker combined with Docker compose
- fast creation of new service with /create_service
- Makefile and Tmuxp-config for easier development of services based on fastapi-serviceutils using **Make** and **tmux-session**

CHAPTER

TWO

CONTENT

Fastapi-serviceutils contains three subpackages:

- `fastapi_serviceutils.app`
- `fastapi_serviceutils.cli`
- `fastapi_serviceutils.utils`

`fastapi_serviceutils.app` contains functions and classes for app-configuration (like config.yml file, logger, etc.), handlers and endpoint creation.

`fastapi_serviceutils.cli` contains executables for easier development like `create_service` to use the fastapi_serviceutils_template.

`fastapi_serviceutils.utils` contain utils to interact with external resources like databases and services, testutils and other utilities.

To see detailed usage of these functions and classes, and also recommended service-structure, see *exampleservice*.

CHAPTER

THREE

TABLE OF CONTENTS

3.1 Usage

3.1.1 exampleservice

The easiest way to explain how to use fastapi-serviceutils is to demonstrate usage inside an exampleservice. Here we will explain the parts of the service and which functions and classes when to use.

Creating new service

To create a new service we use the tool `create_service` which is available after installing fastapi-serviceutils.

```
create_service -n exampleservice \
    -p 50001 \
    -a "Dummy User" \
    -e dummy.user@something.info \
    -ep example \
    -o /tmp
```

This creates the service **exampleservice** inside the folder **/tmp/exampleservice**. As author with email we define **Dummy User** and **dummy.user@something.info**. The initial endpoint we want to create is **example**. The service should listen to port **50001**.

If we change into the created directory we will have the following folder-structure:

```
exampleservice
├── app
│   ├── config.yml
│   ├── endpoints
│   │   ├── __init__.py
│   │   └── v1
│   │       ├── errors.py
│   │       ├── example.py
│   │       ├── __init__.py
│   │       └── models.py
│   ├── __init__.py
│   └── main.py
├── .codespell-ignore-words.txt
├── docker-compose.yml
├── Dockerfile
├── docs
│   └── ...
```

(continues on next page)

(continued from previous page)

```
├── .gitignore
├── Makefile
├── .pre-commit-config.yaml
├── pyproject.toml
├── .python-version
├── README.md
├── setup.cfg
├── tests
│   └── __init__.py
└── .tmuxp.yml
```

The files `docker-compose.yml` and `Dockerfile` are required for deployment of the service as docker-container.

`.tmuxp.yml` is used for development of the service if you prefer to develop inside tmux in combination with for example vim or emacs.

The `.python-version` defines which python-version this service uses and is used by poetry / dephell workflow inside virtual-environments.

The `pyproject.toml` is used for dependency-management and package-creation.

`setup.cfg` contains configurations for tools used during development like yapf, flake8, pytest, etc.

The `.pre-commit-config.yaml` allows the usage of pre-commit and is also used in the make command `make check`. It enables running of multiple linters, checkers, etc. to ensure a fixed codestyle.

The `Makefile` contains helper command like initializing the project, updating the virtual-environment, running tests, etc.

Because codespell is used inside the configuration of pre-commit, the file `.codespell-ignore-words.txt` is used to be able to define words to be ignored during check with codespell.

Initialising project

To initialise the project after creation we run:

```
make init
```

This creates the virtual-environment and installs the dependencies as defined in the `pyproject.toml`. It also initialises the project as a git-folder and creates the initial commit.

We now activate the poetry-shell to enable the environment:

```
poetry shell
```

Attention: Please ensure to always enable the poetry-shell before development using:

```
poetry shell
```

The Makefile assumes the environment is activated on usage.

Folder-structure

Following shows code-relevant files for an exampleservice as created using the create_service-tool of fastapi-serviceutils.

```
exampleservice
├── app
│   ├── config.yml
│   ├── endpoints
│   │   ├── __init__.py
│   │   └── v1
│   │       ├── errors.py
│   │       ├── example.py
│   │       ├── __init__.py
│   │       └── models.py
│   ├── __init__.py
│   └── main.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── service_test.py
```

pyproject.toml

The dependencies and definitions like the package-name, version, etc. are defined inside the `pyproject.toml`. This file is used by Poetry and Dephell. Following the `pyproject.toml` for our exampleservice:

Listing 1: the `pyproject.toml` of the exampleservice.

```
[tool.poetry]
name = "exampleservice"
version = "0.1.0"
description = "Exampleservice to demonstrate usage of fastapi-serviceutils."
authors = ["Dummy User <dummy.user@something.info>"]
readme = "README.md"
include = ["README.md", "app/config.yml"]

[tool.poetry.dependencies]
python = ">=3.7,<4"
fastapi-serviceutils = ">=2"

[tool.poetry.dev-dependencies]
autoflake = ">=1.3"
coverage-badge = ">=1"
flake8 = ">=3.7"
ipython = ">=7.8"
isort = ">=4.3"
jedi = ">=0.14"
neovim = ">=0.3.1"
pre-commit = ">=1.18.3"
pudb = ">=2019.1"
pygments = ">=2.4"
pytest = ">=5"
pytest-asyncio = ">=0.10"
pytest-cov = ">=2"
pytest-xdist = ">=1.30"
```

(continues on next page)

(continued from previous page)

```
sphinx = ">=2"
sphinx-autodoc-typehints = ">=1.6"
sphinx-rtd-theme = ">=0.4.3"
yapf = ">=0.27"

[tool.poetry.extras]
devs = [
    "autoflake", "coverage", "coverage-badge", "flake8", "ipython", "isort",
    "jedi", "neovim", "pre-commit", "pudb", "pygments", "pytest",
    "pytest-asyncio", "pytest-cov", "pytest-xdist", "sphinx",
    "sphinx-autodoc-typehints", "sphinx-rtd-theme", "yapf"
]

[tool.dephell.devs]
from = {format = "poetry", path = "pyproject.toml"}
envs = ["main", "devs"]

[tool.dephell.main]
from = {format = "poetry", path = "pyproject.toml"}
to = {format = "setuppy", path = "setup.py"}
envs = ["main"]
versioning = "semver"

[tool.dephell.lock]
from = {format = "poetry", path = "pyproject.toml"}
to = {format = "poetrylock", path = "poetry.lock"}

[tool.poetry.scripts]
exampleservice = "app.main:main"

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

app/config.yml

The service is configured using a config-file (`config.yml`). It is possible to overwrite these setting using environment-variables. An example for the `config.yml` of the exampleservice is shown below:

Listing 2: `config.yml` of exampleservice.

```
service:
    name: 'exampleservice'
    mode: 'devl'
    port: 50001
    description: 'Example tasks'
    apidoc_dir: 'docs/_build'
    readme: 'README.md'
    allowed_hosts:
        - '*'
    use_default_endpoints:
        - alive
        - config
external_resources:
    services: null
```

(continues on next page)

(continued from previous page)

```
    databases: null
    other: null
logger:
    path: './log/EXAMPLESERVICE'
    filename: 'service_{mode}.log'
    level: 'debug'
    rotation: '1 days'
    retention: '1 months'
    format: "<green>{time:YYYY-MM-DD HH:mm:ss.SSS}</green> | <level>{level: <8}</
↪level> | <cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> [id:
↪{extra[request_id]}] - <level>{message}</level>"
available_environment_variables:
    env_vars:
        - SERVICE__MODE
        - SERVICE__PORT
        - LOGGER__LEVEL
        - LOGGER__PATH
        - LOGGER__FILENAME
        - LOGGER__ROTATION
        - LOGGER__RETENTION
        - LOGGER__FORMAT
    external_resources_env_vars:
        - EXTERNAL_RESOURCES__API__URL
        - EXTERNAL_RESOURCES__API__SCHEMA
    rules_env_vars: []
```

The config contains four main sections:

- **service**
- **external_resources**
- **logger**
- **available_environment_variables**

config: [service]

Inside this section we define the name of the service `name`. This name is used for the **swagger-documentation** and **extraction of the environment-variables**.

The `mode` define the **runtime-mode** of the service. This mode can be overwritten with the environment-variable `EXAMPLESERVICE__SERVICE__MODE` (where `'EXAMPLESERVICE'` is the name of the service, meaning if you have a service named `SOMETHING` the environment-variable would be `SOMETHING__SERVICE__MODE`).

The `port` configure the **port the service will listen to**. This can also be overwritten using the environment variable `EXAMPLESERVICE__SERVICE__PORT`.

The `description` is used for the swagger-documentation.

To define the folder where the to find the **apidoc to serve by route** `/api/apidoc/index.html` the keyword `apidoc_dir` is used.

`readme` defines where to get the readme from to be used as main description for the swagger-documentation at `/docs` / `/redoc`.

To controll if only specific hosts are allowed to controll the service we use `allowed_hosts`. Per default a service would allow all hosts (`'*'`) but this can be customized here in the config.

To define which default endpoints should be included in our service we use `use_default_endpoints`. Currently we support the default endpoints `/api/alive` (inside config: `'alive'`) and `/api/config` (inside config: `'alive'`).

config: [external_resources]

Inside this section external dependencies (resources) are defines. A service can depend on other services, databases, remote-connections or files / folders.

Dependencies to other services should be defined inside `services`. Database connections inside `databases` (currently only postgres is supported). If any other dependency exist define it in `other`.

Defined services can be accessed in the code using `app.config.external_resources.services` or `ENDPOINT.config.external_resources.services` depending if you are in a main part of the app or inside an endpoint.

Databases are automatically included into the `startup` and `shutdown` handlers. You can access the database connection using `app.databases['DATABASE_NAME']` or `ENDPOINT.databases['DATABASE_NAME']` depending if you are in a main part of the app or inside an endpoint.

config: [logger]

All settings inside this section are default Loguru settings to configure the logger. You can control where to log (`path`) and how the logfile should be named (`filename`). Also which minimum level to log (`level`). To control when to rotate the logfile use `rotation`. `retention` defines when to delete old logfiles. The `format` defines the format to be used for log-messages.

config: [available_environment_variables]

The environment-variables are seperated into three types:

- `env_vars`
- `external_resources_env_vars`
- `rules_env_vars`

Here you can control which environment-variables to use if they are set.

The environment-variables are named like the following: `<SERVICENAME>__<MAJOR_SECTION>__<PARAMETER_NAME>`. The servicename would be `'EXAMPLESERVICE'` in our example. The major-section is one of:

- `'SERVICE'`
- `'LOGGER'`
- `'EXTERNAL_RESOURCES'`

`env_vars` control the sections `service` and `logger`. `external_resources_env_vars` control the configurations inside the section `external_resources`. The `rules_env_vars` should overwrite settings of a ruleset of the service. Such a ruleset defines constants and other rules for the logic of the endpoints. For example a default time-range for your pandas dataframes, etc. Currently this is not implemented, so you would have to use these definitions yourself to overwrite your ruleset-definitions.

app/__init__.py

Inside the `__init__.py` file of the app we only define the version of our service.

Note: We use semantic-versioning style for services based on fastapi-serviceutils.

This means we have the following version-number: `<MAJOR>.<MINOR>.<PATCH>`.

For details about semantic-versioning see Semver.

If we bump the version using either `dephell bump {major, minor, fix}` or `poetry version {major, minor, patch}`, both the version defined here, and the version defined inside the `pyproject.toml` will be increased.

Listing 3: `__init__.py` of app.

```
__version__ = '0.1.0'
```

app/main.py

Inside this file we glue all parts of our service together.

Here the `app` is created which is used either in development inside the function `main` or in production using `uvicorn` from command line (or docker-container).

Listing 4: `main.py` of app.

```
from pathlib import Path
from typing import NoReturn

from app import __version__
from app.endpoints import ENDPOINTS

from fastapi_serviceutils import make_app

app = make_app(
    config_path=Path(__file__).with_name('config.yml'),
    version=__version__,
    endpoints=ENDPOINTS,
    enable_middlewares=['trusted_hosts',
                        'log_exception'],
    additional_middlewares=[]
)


def main() -> NoReturn:
    import uvicorn
    uvicorn.run(app, host='0.0.0.0', port=app.config.service.port)


if __name__ == '__main__':
    main()
```

We define where to collect the config-file of the service from, the version of the service and which endpoints and middlewares to use.

app/endpoints/v1/example.py

The following shows the example-endpoint we created:

Listing 5: `example.py` in version 1. Define the endpoint example.

```
from app.endpoints.v1.models import Example as Output
from app.endpoints.v1.models import GetExample as Input
from fastapi import APIRouter
from fastapi import Body
from starlette.requests import Request

from fastapi_serviceutils.app import create_id_logger
from fastapi_serviceutils.app import Endpoint

ENDPOINT = Endpoint(router=APIRouter(), route='/example', version='v1')
SUMMARY = 'Example request.'
EXAMPLE = Body(..., example={'msg': 'some message.'})


@ENDPOINT.router.post('/', response_model=Output, summary=SUMMARY)
async def example(request: Request, params: Input = EXAMPLE) -> Output:
    _, log = create_id_logger(request=request, endpoint=ENDPOINT)
    log.debug(f'received request for {request.url} with params {params}.')
    return Output(msg=params.msg)
```

The `ENDPOINT` includes the `router`, `route` and the `version` of our endpoint.

Inside the endpoint-function we create a new bound logger with the request-id of the request to allow useful traceback.

Note: Defining endpoints like this allows our worklow with endpoint-versioning and usage of `fastapi_serviceutils.endpoints.set_version_endpoints()` inside `app/endpoints/v1/__init__.py` and `app/endpoints/__init__.py`.

---

app/endpoints/v1/models.py

The models.py contains models for the endpoints in version 1 of our exampleservice.

For each endpoint we create the model for the input (request) and the model for the output (response).

The models are of type `pydantic.BaseModel`

Listing 6: `models.py` of endpoints of version 1.

```
from pydantic import BaseModel


class GetExample(BaseModel):
    msg: str


class Example(BaseModel):
    msg: str


__all__ = ['Example', 'GetExample']
```

More complex models could look like the following:

```
"""
In special cases also an ``alias_generator`` has to be defined.
An example for such a special case is the attribute ``schema`` of
:class:`SpecialParams`. The schema is already an attribute of a BaseModel,
so it can't be used and an alias is required.

To be able to add post-parse-methods the pydantic ``dataclass`` can be
used.
An example for this can be seen in :class:`Complex`.
"""

from pydantic import BaseModel
from pydantic import Schema
from pydantic.dataclasses import dataclass

@dataclass
class Complex:
    """Represent example model with attribute-change of model after init."""
    accuracy: str

    def __post_init_post_parse__(self) -> NoReturn:
        """Overwrite self.accuracy with a mapping as defined below."""
        accuracy_mapping = {
            'something': 's',
            'match': 'm',
        }
        self.accuracy = accuracy_mapping[self.accuracy]

def _alias_for_special_model_attribute(alias: str) -> str:
    """Use as ``alias_generator`` for models with special attribute-names."""
    return alias if not alias.endswith('_') else alias[:-1]

class SpecialParams(BaseModel):
    """Represent example model with special attribute name requiring alias."""
    msg: str
    schema_: str = Schema(None, alias='schema')

    class Config:
        """Required for special attribute ``schema``."""
        alias_generator = _alias_for_special_model_attribute
```

app/endpoints/v1/__init__.py

Inside this file we include our example-endpoint to the version 1 endpoints.

Note: If additional endpoints are available, these should be added here, too.

The created `ENDPOINTS` is used inside `app/endpoints/__init__.py` later.

Note: If we would increase our version to version 2 and we want to change the endpoint `example` we would add an additional folder inside `app/endpoints` named `v2` and place the new version files there.

Listing 7: `__init__.py` of v1.

```
from app.endpoints.v1 import example

from fastapi_serviceutils.app.endpoints import set_version_endpoints

ENDPOINTS = set_version_endpoints(
    endpoints=[example],
    version='v1',
    prefix_template='/api/{version}{route}'
)

__all__ = ['ENDPOINTS']
```

app/endpoints/__init__.py

In this file we import all endpoint-versions like in this example `from app.endpoints.v1 import ENDPOINTS as v1`.

Note: If we would have an additional version 2 we would also add `from app.endpoints.v2 import ENDPOINTS as v2`.

Then we use `fastapi_serviceutils.endpoints.set_version_endpoints()` with the latest version endpoints to create `LATEST`.

Note: If we would have version 2, too we would replace parameter `endpoints` with `v2`.

The `ENDPOINTS` is a list of all available versions.

These `ENDPOINTS` are used inside `app/main.py` to include them to the service.

Listing 8: `__init__.py` of endpoints.

```
from app.endpoints.v1 import ENDPOINTS as v1

from fastapi_serviceutils.app.endpoints import set_version_endpoints

LATEST = set_version_endpoints(
    endpoints=v1,
    version='latest',
    prefix_template='{route}'
)

ENDPOINTS = LATEST + v1

__all__ = ['ENDPOINTS']
```

tests

The tests for the exampleservice are using Pytest. We also used the testutils of `fastapi-serviceutils`. An example for simple endpoint tests of our exampleservice:

Listing 9: `tests/service_test.py`

```
import pytest
from app.main import app

from fastapi_serviceutils.app.service_config import Config
from fastapi_serviceutils.utils.tests.endpoints import json_endpoint


def test_endpoint_example():
    json_endpoint(
        application=app,
        endpoint='/api/v1/example/',
        payload={'msg': 'test'},
        expected={'msg': 'test'}
    )


@pytest.mark.parametrize(
    'endpoint, status_code',
    [
        ('/api/v1/example',
         307),
        ('/api/',
         404),
        ('/api/v1/',
         404),
        ('/api/v1/example/',
         200),
    ]
)
def test_endpoint_invalid(endpoint, status_code):
    json_endpoint(
        application=app,
        endpoint=endpoint,
        status_code=status_code,
        payload={'msg': 'test'}
    )
```

3.1.2 External resources

Databases

config.yml

If we use a database in our service we declare the connection info in the `config.yml` of the service like the following:

Listing 10: `app/config.yml`

```
...
external_resources:
    services: null
    databases:
```

(continues on next page)

(continued from previous page)

```
        userdb:
            dsn: 'postgresql://postgres:1234@localhost:5434/userdb'
            databasetype: 'postgres'
            min_size: 5
            max_size: 20
    other: null
...
```

For each database we want to use in our service, we define a new item inside `databases`. The **key** will be the name of our database. The connection itself is defined as `dsn`. The databasetype defines the type of the database we are using. This setting is for future releases of fastapi-serviceutils. Currently we only support `postgres` and this setting has no effect. `min_size` and `max_size` define the minimum and maximum amount of connections to open to the database.

app/endpoints/v1/dbs.py

Inside the module `dbs.py` we define our datatables like the following:

Listing 11: `app/endpoints/v1/dbs.py`

```
from sqlalchemy import Boolean
from sqlalchemy import Column
from sqlalchemy import insert
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    password = Column(String)
```

app/endpoints/v1/models.py

As for each endpoint we declare the input- and output-models we are using in our new endpoints like the following:

Listing 12: `app/endpoints/v1/models.py`

```
from pydantic import BaseModel

class InsertUser(BaseModel):
    email: str
    password: str

class Inserted(BaseModel):
    msg: bool = True

class User(BaseModel):
    id: int
```

(continues on next page)

(continued from previous page)

```
    email: str
    password: str
```

app/endpoints/v1/insert_user.py

Listing 13: app/endpoints/v1/insert_user.py

```
from fastapi import Body
from fastapi import APIRouter
from fastapi_serviceutils.app import Endpoint
from fastapi_serviceutils.app import create_id_logger
from sqlalchemy import insert

from app.endpoints.v1.dbs import User
from app.endpoints.v1.models import InsertUser as Input
from app.endpoints.v1.models import Inserted as Output

ENDPOINT = Endpoint(router=APIRouter(), route='/insert_user', version='v1')
SUMMARY = 'Example request.'
EXAMPLE = Body(
    ...,
    example={
        'email': 'dummy.user@something.info'
        'password': 'an3xampleP4ssword'
    }
)

@ENDPOINT.router.post('/', response_model=Output, summary=SUMMARY)
async def insert_user(params: Input = EXAMPLE, request: Request) -> Output:
    _, log = create_id_logger(request=request, endpoint=ENDPOINT)
    log.debug(f'received request for {request.url} with params {params}.')
    database = app.databases['userdb'].dbase
    async with database.transaction():
        query = insert(User).values(
            email=params.email,
            password=params.password
        )
        await database.execute(query)
    return Output()
```

app/endpoints/v1/get_users.py

Listing 14: app/endpoints/v1/get_users.py

```
from fastapi import Body
from fastapi import APIRouter
from fastapi_serviceutils.app import Endpoint
from fastapi_serviceutils.app import create_id_logger

from app.endpoints.v1.dbs import User
from app.endpoints.v1.models import User as Output
```

(continues on next page)

(continued from previous page)

```
ENDPOINT = Endpoint(router=APIRouter(), route='/get_users', version='v1')
SUMMARY = 'Example request.'

@ENDPOINT.router.post('/', response_model=Output, summary=SUMMARY)
async def get_users(request: Request) -> List[Output]:
    _, log = create_id_logger(request=request, endpoint=ENDPOINT)
    log.debug(f'received request for {request.url}.')
    database = app.databases['userdb'].dbase
    async with database.transaction():
        users = await database.fetch_all(User.__table__.select())
    return users
```

app/endpoints/v1/__init__.py

Finally we include these endpoints to our `ENDPOINTS`.

Listing 15: `__init__.py`

```
from fastapi_serviceutils.endpoints import set_version_endpoints

from app.endpoints.v1 import get_users
from app.endpoints.v1 import insert_user

ENDPOINTS = set_version_endpoints(
    endpoints=[get_users, insert_user],
    version='v1',
    prefix_template='/api/{version}{route}'
)

__all__ = ['ENDPOINTS']
```

The rest of our service, like the `main.py`, the `__init__.py` files of the modules, etc. have the same content as described in `exampleservice`.

Services

If we need to call external services we first have to declare the service inside the `config.yml` like the following:

Listing 16: `app/config.yml`

```
...
external_resources:
    services:
        testservice:
            url: http://someserviceurl:someport
            servicetype: rest
    databases: null
    other: null
...
```

Listing 17: app/endpoints/v1/models.py

```
from pydantic import BaseModel

class CallExternalService(BaseModel):
    street: str
    street_number: str
    zip_code: str
    city: str
    country: str

class ExternalServiceResult(BaseModel):
    longitude: str
    latitude: str
```

Listing 18: app/endpoints/v1/external_service.py

```
from fastapi import APIRouter
from fastapi import Body
from fastapi_serviceutils.app import Endpoint
from fastapi_serviceutils.app import create_id_logger
from fastapi_serviceutils.utils.external_resources.services import call_service
from starlette.requests import Request

from app.endpoints.v1.models import CallExternalService as Input
from app.endpoints.v1.models import ExternalServiceResult as Output

ENDPOINT = Endpoint(router=APIRouter(), route='/use_service', version='v1')
SUMMARY = 'Example request using an external service.'
EXAMPLE = Body(
    ...,
    example={
        'street': 'anystreetname',
        'street_number': '42',
        'city': 'anycity',
        'country': 'gallifrey'
    }
)

@ENDPOINT.router.post('/', response_model=Output, summary=SUMMARY)
async def use_service(params: Input = EXAMPLE, request: Request) -> Output:
    data_to_fetch = {
        'street': params.street,
        'auth_key': 'fnbkjgkegej',
        'street_number': params.street_number,
        'city': params.city,
        'country': params.country
    }
    return await call_service(
        url=app.databases['testservice'].url,
        params=data_to_fetch,
        model=ExternalServiceResult
    )
```

3.1.3 Helpers

create_service

Create new service following the structure as described in the fastapi_serviceutils documentation. Using Cookiecutter to create the new folder.

```
usage: create_service [-h] -n SERVICE_NAME -p SERVICE_PORT -a AUTHOR -e
                      AUTHOR_EMAIL -ep ENDPOINT -o OUTPUT_DIR

create new service based on fastapi using fastapi_serviceutils.

optional arguments:
  -h, --help            show this help message and exit
  -n SERVICE_NAME, --service_name SERVICE_NAME
                        the name of the service to create. ATTENTION: only
                        ascii-letters, "_" and digits are allowed. Must not
                        start with a digit!
  -p SERVICE_PORT, --service_port SERVICE_PORT
                        the port for the service to listen.
  -a AUTHOR, --author AUTHOR
                        the name of the author of the service.
  -e AUTHOR_EMAIL, --author_email AUTHOR_EMAIL
                        the email of the author of the service.
  -ep ENDPOINT, --endpoint ENDPOINT
                        the name of the endpoint for the service to create.
                        ATTENTION: only lower ascii-letters, "_" and digits
                        are allowed. Must not start with a digit!
  -o OUTPUT_DIR, --output_dir OUTPUT_DIR
```

Makefile

Usual tasks during development are wrapped inside the Makefile. This contains updating of the environment, creation of the docs, etc.

```
Helpers for development of fastapi_serviceutils.

  Usage:

    make <target> [flags...]

  Targets:

    check      Run all checks defined in .pre-commit-config.yaml.
    clean      Clean the working directory from temporary files and caches.
    doc        Create sphinx documentation for the project.
    docs       Create sphinx documentation for the project.
    finalize   Finalize the main env.
    help       Show the help prompt.
    info       Show info about current project.
    init       Initialize project
    tests      Run tests using pytest.
    update     Update environments based on pyproject.toml definitions.

  Flags:
```

(continues on next page)

(continued from previous page)

```
  Note:
      This workflow requires the following programs / tools to be installed:
      - poetry
      - dephell
      - pyenv
```

tmuxp

For a predefined development environment the `.tmuxp.yml` configuration can be used to create a Tmux-session (using Tmuxp) with a window including three panels:

- one panel for **editing files**
- one panel **running the service**
- one panel **running the tests**

Run the following command to create the tmux-session:

```
tmuxp load .
```

3.1.4 Deployment

For more detailed information about deployment of fastapi-based services see FastAPI deployment

Services based on `fastapi-serviceutils` can be easily deployed inside a docker-container.

Before deployment you need to:

- **update the dependencies**
- **run all tests**
- **create the current** `requirements.txt`
- **ensure the** `docker-compose.yml` **is defined correctly including the** `environment-variables`

To run these tasks run:

```
make finalize
```

To run the service using docker-compose customize the `docker-compose.yml` and run:

```
sudo docker-compose up -d
```

Basics

Docker

The basic Dockerfile should look like:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

COPY requirements.txt ./
```

(continues on next page)

(continued from previous page)

```
RUN pip install -r requirements.txt

COPY . /app
```

Docker-compose

The service can be deployed with Docker compose using the Docker compose file:

Listing 19: an example for a `docker-compose.yml` for a service using `fastapi_serviceutils`.

```
version: '3.7'

services:
    <SERVICENAME>:
        build:
            context: .
            dockerfile: Dockerfile
        image: <SERVICENAME>
        ports:
            - "<SERVICE_PORT>:80"
        environment:
            - <SERVICENAME>_SERVICE__MODE="prod"
            - ...
        volumes:
            - type: bind
              source: <LOGFOLDER_ON_HOST>
              target: <LOGFOLDER_INSIDE_DOCKER_CONTAINER>
```

Environment-variables

Setting environment-variables overwrites the default values defined in the config.

Please ensure to use the **environment-variables** (Environment variable) if you want to overwrite some default-settings of the service.

The environment-variables to use should be defined inside the `config.yml`. Set the values of the environment-variables inside the `docker-compose.yml`.

3.2 Development

3.2.1 Getting Started

After cloning the repository the development environment can be initialized using:

```
make init
```

This will create the dev environment `fastapi_serviceutils/dev`. Activate it.

Note: Make sure to always activate the environment when you start working on the project in a new terminal using

```
poetry shell
```

To update dependencies and `poetry.lock`:

```
make update
```

This also creates `requirements.txt` to be used for Docker.

3.2.2 Dependency management

We use Poetry including the dependency definition inside the `pyproject.toml` and `python-venv` for environment management. Additionally we use Dephell and `make` for easier workflow.

Listing 20: dependency-management files

```
<SERVICENAME>
├── ...
├── poetry.lock
├── pyproject.toml
├── .python-version
└── ...
```

- `pyproject.toml`: stores what dependencies are required in which versions. Required by Dephell and Poetry.
- `poetry.lock`: locked definition of installed packages and their versions of currently used devs-environment. Created by Poetry using `make init`, `make update`, `make tests` or `make finalize`.
- `.python-version`: the version of the python-interpreter used for this project. Created by `python-venv` using `make init`, required by Poetry and Dephell.

3.2.3 Testing

All tests are located inside the folder `tests`. Tests for a module should be names like `<MODULE_NAME>_test.py`.

Note: For often used functions and workflows during testing the functions and classes inside `fastapi_serviceutils.utils.tests` can be used.

To run the tests run:

```
make tests
```

A HTML coverage report is automatically created in the `htmlcov` directory.

See also:

For additional information how to test fastapi-applications:

- https://fastapi.tiangolo.com/tutorial/testing/
- https://fastapi.tiangolo.com/tutorial/testing-dependencies/

For information how to test async functions:

- https://github.com/pytest-dev/pytest-asyncio

3.2.4 Documentation

The project's developer documentation is written using Sphinx.

The documentation sources can be found in the `docs` subdirectory. They are using restructuredText-files.

The API-documentation is auto-generated from the docstrings of modules, classes, and functions. For documentation inside the source-code the Google docstring standard is used.

To generate the documentation, run

```
make docs
```

The created documentation (as html files) will be inside the `docs/_build` directory.

There is also a swagger-documentation to be used for users of the service. After starting the service the documentation can be viewed at:

- http://0.0.0.0:<SERVICE_PORT>/docs
- http://0.0.0.0:<SERVICE_PORT>/redoc

The sphinx-documentation can be viewed after service-started and docs created at http://0.0.0.0:<SERVICE_PORT>/apidoc/index.html.

Listing 21: documentation related files

```
<SERVICENAME>
├── ...
├── docs
│   ├── _build
│   │   └── ...
│   ├── conf.py
│   ├── development.rst
│   ├── index.rst
│   ├── <ADDITIONAL_DOCUMENTATION_PAGE>.rst
│   └── _static
│       ├── coverage.svg
│       └── logo.png
├── ...
├── README.md
└── ...
```

3.3 fastapi_serviceutils package

3.3.1 Subpackages

fastapi_serviceutils.app package

Subpackages

fastapi_serviceutils.app.endpoints package

Subpackages

Submodules

fastapi_serviceutils.utils.tests.endpoints module

3.4 See also

Internal documentation:

- *API Documentation*
- *Development*
- *Deployment*

Used tools:

- Cookiecutter
- Dephell
- Docker
- Docker compose
- Make
- Poetry
- restructuredText
- Sphinx
- Tmux
- Tmuxp

Used packages:

- Databases
- FastAPI
- Loguru
- Requests
- Toolz
- SQLAlchemy

Additional sources:

- FastAPI deployment
- Google docstring standard
- reStructuredText reference
- Type Annotations