BINP Documentation

Release 0.0.2

Baryshnikov Aleksandr

Feb 14, 2021

CONTENTS:

Provides a platform for automation with code-first approach, with embedded batteries:

- Tracing (journals)
- Internal and user-defined API
- Ultra-light but rich mobile-first UI
- Embedded key-value storage

It is heavily inspired by node-red and aims to provide same enjoyment during development but without mess of nodes and connections for tasks a little bit more complicated than just hello world.

The platform also tries to be easy in deployment and maintaining. Code could be stored in a SCM (ex: git) and persistent storage is just a single file that could be backed up and restored trivially.

Because memory and CPU consumption relatively low a solution based on the platform could be launched even on Raspberry Pi Zero with 512MB RAM.

Requires:

- python 3.8+

CHAPTER

ONE

INSTALLATION

```
pip install binp uvicorn[standard]
```

CHAPTER

TWO

SAMPLE USAGE

Put your code to sample.py and run

`uvicorn --reload sample:binp.app`

- UI: http://localhost:8000
- Internal API: http://localhost:8000/internal/redoc

2.1 Hello world

```
from binp import BINP

binp = BINP()


@binp.action
@binp.journal
async def hello():
    """
    Print hello world in console
    """
    print("hello world")
```

2.2 Hello world with API

```
from binp import BINP

binp = BINP()


@binp.app.get("/hello")
@binp.journal
async def hello():
    """
    Return hello world
    """
    return {"message": "hello world"}
```

- Exposed API: http://localhost:8000/redoc

Try by Curl: `curl http://127.0.0.1:8000/hello`

2.3 Real-world example: fetch currencies every day or manually

Requires additional dependencies: `pip install aiohttp aiocron`

```
from datetime import date
from typing import Dict

from aiocron import crontab
from aiohttp import ClientSession
from binp import BINP
from pydantic import BaseModel

binp = BINP()


class Rates(BaseModel):
    base: str = 'USD'
    rates: Dict[str, float] = {}
    date: date


@crontab('0 12 * * *')
@binp.action
@binp.journal
async def currency_rates():
    """
    Fetch currency rates
    """
    # make basic HTTP API request
    async with ClientSession() as session:
        res = await session.get('https://api.exchangeratesapi.io/latest?base=USD')
        assert res.ok, f"result {res.status}"
        data = await res.json()
    # parse response data
    rates = Rates.parse_obj(data)
    # add record to journal about result
    await binp.journal.record('current exchange rates', base_currency='USD',␣
↪rates=rates)
     # save rates to default namespace
    await binp.kv.save(rates)
```

2.3.1 BINP python API

class binp.**BINP**(*kv=<factory>*, *journal=<factory>*, *action=<factory>*, *service=<factory>*)

action: *binp.action.Action*
UI exposed actions (buttons)

property app
Creates FastAPI applications and caches result.

journal: *binp.journals.Journals*
Journal for operation tracing

kv: *binp.kv.KV*
Key-Value default storage

service: *binp.service.Service*
Background services

class binp.**Action**
Expose user-defined action as 'button' in UI

Expose async function as button to ui.

It will not be automatically journaled: it's up to you add `@binp.journal` annotation or not.

Example

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.action
async def invoke():
    '''
    Do something
    '''
    await sleep(3) # emulate some work
    print("done")
```

By default, action will be exposed with name equal to fully-qualified function name and description from docstring (if exists).

Exposed name could by optionally defined manually.

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.action(name='Do Something', description='Emulate some heavy work')
async def invoke():
    await sleep(3)
    print("done")
```

Conflicts

Actions are indexed by name. If multiple actions defined with the same name - the latest one will be used.

property actions
Copy of list of defined actions prepared for serialization.

Return type List[ActionInfo]

async invoke(*name*)
Invoke action by name or ignore. If handler will raise an error, the error will NOT be suppressed.

Parameters **name** (str) – action name

Return type bool

Returns true if action invoked

class binp.**KV**(*namespace='default'*, *db=None*)
Basic Key-Value storage with namespace.

By default - 'default' namespace used. All values are serialized to JSON by standard JSON module or in case value is subclass of pydantic BaseMode - by pydantic `.json()`.

Example

```
from binp import BINP

binp = BINP()

async def save_something():
    # save multiple values to storage
    await binp.kv.set(name="reddec", repo="binp", year=2020)
    # get single value
    name = await binp.kv.get('name')
    assert name == 'reddec'
```

There is some optimization for saving/loading Pydantic classes when class name itself a key.

Example

```
from binp import BINP
from pydantic.main import BaseModel

binp = BINP()

class Author(BaseModel):
    name: str
    repo: str
    year: int

async def save_something():
    value = Author(name="reddec", repo="binp", year=2020)
    # save class with key name equal to fully-qualified class name
    await binp.kv.save(value)
    # restore
    saved_value = await binp.kv.load(Author)
    assert value == saved_value
```

async get(*name*)
Get save value by name.

Return type Union[str, int, float, bool, dict, None]

async load(*klass*)
Load and parse value by key name equal to class name (fqdn)

Parameters **klass** (Type[~T]) – BaseModel inherited class to load and parse

Return type Optional[~T]

async namespaces()
Fetch all namespaces in selected database.

Return type List[str]

async remove(**names*)
Remove multiple values by names

async save(*value*)
Save value with key name equal to class name (fqdn)

select(*namespace*)
Get accessor to another namespace in a same database

Return type *KV*

async set(***values*)
Sav multiple values into storage. All values should be serializable to JSON.

2.3.2 Journal

class binp.journals.**Headline**(***data*)
Journal header

description: str
operation description

duration: Optional[float]
accurate operation duration

error: Optional[str]
error message if any

finished_at: Optional[datetime.datetime]
execution end time

classmethod from_database(*info*, *labels*)

Return type *Headline*

id: int
Journal unique ID

labels: List[str]
assigned labels

operation: str
operation name

started_at: datetime.datetime
journal creation time

class binp.journals.**Journal**(***data*)
Journal entity with header and linked records

records: List[*binp.journals.Record*]
journal records

class binp.journals.**Journals**(*database=None*)
Journal of logged invokes.

Should be used as decorator for async methods. Will create, track and record changes automatically. In case of exception, the error will be logged and an exception re-raised.

Example

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.journal
```

(continues on next page)

(continued from previous page)

```
async def invoke():
    '''
    Do something
    '''
    await sleep(3) # emulate some work
    print("done")
```

By default, journal will be created with name equal to fully-qualified function name and description from docstring (if exists).

Name and description could by optionally defined manually.

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.journal(operation='Do Something', description='Emulate some heavy work')
async def invoke():
    await sleep(3)
    print("done")
```

It's possible to add multiple record to journal. Each record contains text message and unlimited number of key-value pairs, where value should JSON serializable objects or be subclass of BaseModel (pydantic).

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.journal
async def invoke():
    await binp.journal.record("begin work", source="http://example.com", by=
↪"reddec")
    await sleep(3)
    await binp.journal.record("work done", status="success")
```

It's safe to combine journal annotation with any other decorators.

To get current journal ID use `current_journal`

```
from binp.journals import current_journal

binp = BINP()

@binp.journal
async def invoke():
    print("Journal ID:", current_journal.get())
```

`current_journal` can be used only with function decorated by @journal in call chain. Otherwise it will return None. The variable is context-depended and coroutine-safe.

Events

- `journal_updated` - when journal created or updated. Emits journal ID
- `record_added` - when record added. Emits journal ID

Important! Never set current journal manually.

property current
Helper to get current journal ID

Return type Optional[int]

async get(*journal_id*)
Get single journal by ID

Return type Optional[*Journal*]

async headline(*journal_id*)
Get single journal headline (without records) by ID

Return type Optional[*Headline*]

async history(*offset=0*, *limit=20*)
Get journal headlines in reverse order (newest - first).

Return type List[*Headline*]

async labels(**labels*)
Assign labels to journal. Duplicated labels will be ignored. Only under @journal function.

async record(*message*, ***events*)
Add record to journal. Will work only if there is @journal function in call chain.

Parameters

- **message** (str) – short message, describes record
- **events** (Union[BaseModel, str, int, float, bool]) – key->value of events, where key is event name, and value is basic type or pydantic model. Value will be serialized as JSON

async remove_dead()
Remove all records without finish_at timestamp. Should be called only once BEFORE any writes.

async search(*operation=None*, *failed=None*, *pending=None*, *labels=None*, *offset=0*, *limit=20*)
Search journals. Each condition joined by AND operator. Null conditions will not be applied. If no conditions defined, it's equal to plain history() operation. Result ordered in reverse order (newest - first).

Parameters

- **operation** (Optional[str]) – operation name
- **failed** (Optional[bool]) – with error message
- **pending** (Optional[bool]) – without finished_at attribute
- **labels** (Optional[Collection[str]]) – labels names (at least one of list)
- **offset** (int) – how many records to skip
- **limit** (int) – maximum number of records to return

Return type List[*Headline*]

class binp.journals.**Record**(***data*)
Single record in journal

created_at: datetime.datetime
record creation time

message: str
message provided by invoker

params: Dict[str, Any]
fields defined for record

2.3.3 Action

class binp.action.**Action**
Expose user-defined action as 'button' in UI

Expose async function as button to ui.

It will not be automatically journaled: it's up to you add `@binp.journal` annotation or not.

Example

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.action
async def invoke():
    '''
    Do something
    '''
    await sleep(3) # emulate some work
    print("done")
```

By default, action will be exposed with name equal to fully-qualified function name and description from docstring (if exists).

Exposed name could by optionally defined manually.

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.action(name='Do Something', description='Emulate some heavy work')
async def invoke():
    await sleep(3)
    print("done")
```

Conflicts

Actions are indexed by name. If multiple actions defined with the same name - the latest one will be used.

property actions
Copy of list of defined actions prepared for serialization.

Return type List[ActionInfo]

async invoke(*name*)
Invoke action by name or ignore. If handler will raise an error, the error will NOT be suppressed.

Parameters **name** (str) – action name

Return type bool

Returns true if action invoked

2.3.4 KV

class binp.kv.**KV**(*namespace='default'*, *db=None*)

Basic Key-Value storage with namespace.

By default - 'default' namespace used. All values are serialized to JSON by standard JSON module or in case value is subclass of pydantic BaseMode - by pydantic `.json()`.

Example

```
from binp import BINP

binp = BINP()

async def save_something():
    # save multiple values to storage
    await binp.kv.set(name="reddec", repo="binp", year=2020)
    # get single value
    name = await binp.kv.get('name')
    assert name == 'reddec'
```

There is some optimization for saving/loading Pydantic classes when class name itself a key.

Example

```
from binp import BINP
from pydantic.main import BaseModel

binp = BINP()

class Author(BaseModel):
    name: str
    repo: str
    year: int

async def save_something():
    value = Author(name="reddec", repo="binp", year=2020)
    # save class with key name equal to fully-qualified class name
    await binp.kv.save(value)
    # restore
    saved_value = await binp.kv.load(Author)
    assert value == saved_value
```

async get(*name*)

Get save value by name.

Return type Union[str, int, float, bool, dict, None]

async load(*klass*)

Load and parse value by key name equal to class name (fqdn)

Parameters **klass** (Type[~T]) – BaseModel inherited class to load and parse

Return type Optional[~T]

async namespaces()

Fetch all namespaces in selected database.

Return type List[str]

async remove(**names*)

Remove multiple values by names

async save(*value*)
Save value with key name equal to class name (fqdn)

select(*namespace*)
Get accessor to another namespace in a same database

Return type *KV*

async set(***values*)
Sav multiple values into storage. All values should be serializable to JSON.

2.3.5 Service

class binp.service.**Handler**(*info*, *events*, *handler*, *task=None*)

class binp.service.**Info**(***data*)
Service information

autostart: bool
is service marked to be started automatically

description: str
service description

name: str
service name

restart: bool
is service has to be restarted automatically

restart_delay: float
interval between restarts

status: *binp.service.Status*
actual service status

class binp.service.**Service**
Annotate async function as service (background task). Supports automatic (default) and manual start, restarts, restarts delays.

Useful to interact with environment in unpredictable schedule (ex: listen for low-level network requests).

```
from binp import BINP
from asyncio import sleep

binp = BINP()

@binp.service
async def check_messages():
    while True:
        await sleep(3)
        print("checks")
```

For scheduling by time better use *aiocron* instead:

```
pip install aiocron
```

```
from binp import BINP
from aiocron import crontab
```

(continues on next page)

(continued from previous page)

```
binp = BINP()

@crontab("*/5 * * * *")
@binp.journal
async def poll_something():
    print("do something every 5 minutes....")
```

Conflicts

Services are indexed by name. If multiple services defined with the same name - the old one will be stopped and the latest one will be used.

Events

- `service_changed` - when service status changed. Emits Info

property services
List of all services

Return type `List[`*Info*`]`

start(*name*)
Starts single service by name. Does nothing if no such service or service not yet stopped.

stop(*name*)
Stops service by name. Does nothing if no such service or service stopped.

class binp.service.**Status**(*value*)
Service life status

restarting = 'restarting'
service stopped, is waiting before restart

running = 'running'
service up and running

starting = 'starting'
service scheduled to start

stopped = 'stopped'
service stopped and will not be restarted

2.3.6 Utils

class binp.events.**Emitter**
Typed event emitter based on async queues.

Event emitting is non-blocking operation. After subscription, listener will not miss any event regardless of processing time (in exchange of memory). Events order are strictly the same as emitting order.

Can be used as decorator.

Example

```
on_something : Emitter[str] = Emitter()

@on_something
async def subscriber(payload: str):
    print("payload:", payload)

def emitter():
    on_something.emit('hello world')
```

Also can be used without decorator

Example

```
on_something : Emitter[str] = Emitter()

async def subscriber():
    with on_something.subscribe() as queue:
        while True:
            payload: str = await queue.get()
            print("payload:", payload)

def emitter():
    on_something.emit('hello world')
```

emit (*payload*)
Emit event. Non-blocking operation.

subscribe (*own_queue=None*)
Create queue that will listen for the event. Queue will be automatically unsubscribed. A new queue will be created if no own queue will be provided.

2.3.7 Docker

There is no limitation to use BINP in docker.

Ensure

- Ensure, that your requirements.txt file contains `binp` and `uvicorn[standard]` (or just use `pip freeze` if you already installed it)
- Pack your application as described in DockerHub.
- Define volume `/usr/src/app/data` and expose port `80`, add env `DB_URL=sqlite:////data/data.db`
- Change entry point to `uvicorn --host 0.0.0.0 --port 80 <your main script>:binp.app`

After that you can mount `/data` for persistent storage

Note: If you are using Debian/Ubuntu as source platform and used `pip freeze` - be sure that you removed `pkg-resource` dependencies: it's bug.

Dockerfile

Assuming that your main script same as in examples: `example.py`

```
FROM python:3.8
ENV DB_URL=sqlite:////data/data.db
ENV PYTHONUNBUFFERED=TRUE
EXPOSE 80
VOLUME /data
WORKDIR /usr/src/app

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD [ "uvicorn", "--host", "0.0.0.0", "--port", "80", "example:binp.app"]
```

One line to build and run (first build will take some time):

```
docker run --rm -p 8080:80 $(docker build -q .)
```

2.3.8 Configuration

Quick and dirty

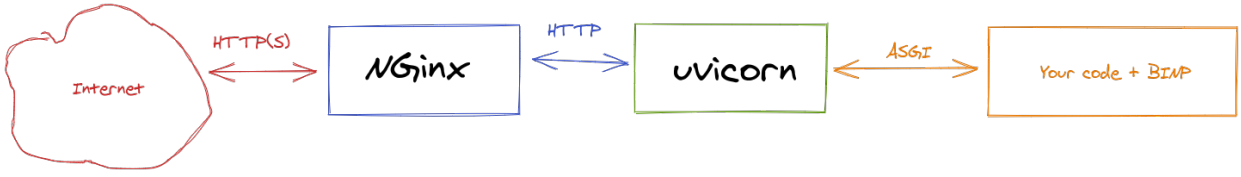
You should bind `uvicron` to external interface or to all interfaces by `uvicron --host 0.0.0.0`

Warning: Not recommended way:

- anyone can reach your service without authorization
- it's bad idea to put application server (uvicorn) outside: it's designed for computing, not for handling zillions requests.

In this scenario we exposed our application built on top of BINP directly to the outer world. Unsafe, but simple.

Better architecture



- Nginx acts as reverse proxy

- Nginx can (and should in public networks) terminate SSL. Check how to setup Let's encrypt certificates with Nginx
- You can setup at least basic authorization

Examples

Assuming that your BINP-based application started by `uvicron` at localhost (127.0.0.1) and default port (8000).

On Debian-based OS, you may just copy-paste configuration to new file under `/etc/nginx/sites-enabled.d/` and restart service after it.

Without authorization

Nginx configuration

```
server {
    listen 80;
    server_name default;


    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_http_version  1.1;
        proxy_cache_bypass  $http_upgrade;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

With basic authorization

Create users file by instruction (chapter "Creating a Password File" only).

Nginx configuration

```
server {
    listen 80;
    server_name default;


    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_http_version  1.1;
        proxy_cache_bypass  $http_upgrade;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        auth_basic "Administrator's Area";
        auth_basic_user_file /etc/apache2/.htpasswd;
    }
}
```

It will secure everything.

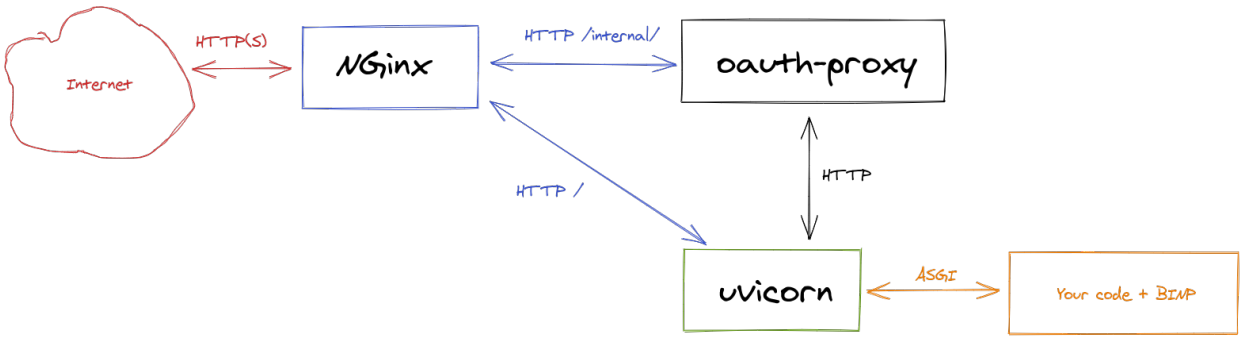
To secure only internal API, but expose user-defined API:

```
server {
    listen 80;
    server_name default;


    location /internal/ {
        proxy_pass http://127.0.0.1:8000/internal/;
        proxy_http_version  1.1;
        proxy_cache_bypass  $http_upgrade;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        auth_basic "Administrator's Area";
        auth_basic_user_file /etc/apache2/.htpasswd;
    }

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_http_version  1.1;
        proxy_cache_bypass  $http_upgrade;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

Best architecture



- Uses OAuth2 to authorize internal API (and UI) by oauth-proxy
- Allows user-defined apps be exposed without authorization (up to admin)

For example, I used auth0 as identity provider.

Examples

TBD - I am tired to write docs.

Administrating

Web server

It's a `uvicorn` or other ASGI server responsibility, however, most common flags for `uvicron` are:

- `--host <host>` - binding host, default 127.0.0.1
- `--port <port>` - binding port, default 8000

For example, using sample from main page, to expose service to all interfaces on port 8080:

> **Warning:** Do not do it on production - anyone can reach your service without authorization

```
uvicorn --host 0.0.0.0 --port 8080 example:binp.app
```

- `0.0.0.0` - special host, means all IP, assigned to your machine

BINP internal configuration

By-default configuration done by environment variables.

Vars

Default configuration can be changed by environment variables:

Note: You can define vars in file and re-use it as: `uvicron --env-file <filename>`

DEV

Boolean, disabled by default.

Enable development mode:

- Adds CORS rules
- Disables UI

Example: `DEV=true uvicorn example:binp.app`

DB_URL

String, default `sqlite:///data.db`

Sqlite URL of database location. In-memory database not supported. Other than sqlite databases may be supported in a future.

Example: `DB_URL=sqlite:///my.db uvicorn example:binp.app`

Customise

You may re-define absolutely each part of BINP instance by using custom values during construction.

For example you want to store journals in other database:

```
from binp import BINP
from binp.journals import Journals

my_custom_db = None # define custom database and apply migrations

binp = BINP(journal=Journals(my_custom_db))


@binp.action
@binp.journal
async def hello():
    """
    Print hello world in console
    """
    print("hello world")
```

Backup and restore

BINP-based application contains two part:

- Your code - I guess it stored somewhere in source control system like git.
- Runtime data

Backup runtime data

Just sqlite database. A single file that can be copied anytime anywhere. By-default, it will be created automatically at first start in a working directory with name `data.db`

Restore

Copy/replace saved database to the DB_URL location. By-default, in a working directory with name `data.db`.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

P

R

S