# aiohttp-demos Documentation

***Release 0.2***

**contributors**

**Sep 21, 2023**

# Contents

If you want to create an application with *aiohttp* there is a step-by-step guide for *Polls* application (*Getting started*). The application is similar to the one from Django tutorial. It allows people to create polls and vote.

There are also many other demo projects, give them a try!

CHAPTER 1

# Example Projects

We have created for you a selection of fun projects, that can show you how to create application from the *blog* to the applications related to data science. Please feel free to add your open source example project by making Pull Request.

- Shortify - *URL shortener* with *Redis* storage.
- Moderator - UI and API for classification of offensive and toxic comments using *Kaggle* data and *scikit-learn*.
- Moderator bot - Slack bot for moderating offensive and toxic comments using provided model from *Moderator AI*
- Motortwit - *Twitter* clone with *MongoDB* storage.
- Imagetagger - Example how to deploy deep learning model with *aiohttp*.
- Chat - Simple *chat* using websockets.
- Polls - Simple *polls* application with PostgreSQL storage.
- Blog - The *blog* application with *PostgreSQL* storage and *Redis* session store.
- Graphql - The simple real-time chat that based on the *GraphQL* api and *Apollo client*.

CHAPTER 2

# Contents

## 2.1 Preparations

Start with an empty folder and create files alongside with the tutorial. If you want the full source code in advance or for comparison, check out the demo source.

### 2.1.1 Project structure

At the end of the tutorial, this project's structure should look very similar to other Python based web projects:

```
.
├── aiohttpdemo_polls
│   ├── static
│   │   ├── images
│   │   │   └── background.png
│   │   └── style.css
│   ├── templates
│   │   ├── 404.html
│   │   ├── 500.html
│   │   ├── base.html
│   │   ├── detail.html
│   │   ├── index.html
│   │   └── results.html
│   ├── db.py
│   ├── __init__.py
│   ├── __main__.py
│   ├── main.py
│   ├── middlewares.py
│   ├── routes.py
│   ├── settings.py
│   ├── utils.py
│   └── views.py
├── config
```

(continues on next page)

(continued from previous page)

```
│   ├── polls_test.yaml
│   └── polls.yaml
├── tests
│   ├── conftest.py
│   ├── __init__.py
│   └── test_integration.py
├── init_db.py
├── Makefile
├── README.rst
├── requirements.txt
├── setup.py
└── tox.ini
```

### 2.1.2 Environment

We suggest you to create an isolated Python virtual environment:

```
$ python3 -m venv env
$ source env/bin/activate
```

During the tutorial, you will be instructed to install some packages inside this activated environment. For example, you will use `$ pip install aiopg` to install `aiopg` before doing the database related sections.

---

**Note:** If you decided to run the application from the repo's source code, install the app and its requirements:

```
$ cd demos/polls
$ pip install -e .
```

---

Check your Python version (tutorial requires Python 3.5 or newer):

```
$ python -V
Python 3.7.3
```

Install `aiohttp`

```
$ pip install aiohttp
```

Check the aiohttp version:

```
$ python3 -c 'import aiohttp; print(aiohttp.__version__)'
3.5.4
```

### 2.1.3 Database

#### Running server

We could have created this tutorial based on a local `sqlite` solution, but `sqlite` is almost never used in real-world applications. To better reflect a production example, we decided to use Postgres for the tutorial.

Install and run the PostgreSQL database server: http://www.postgresql.org/download/. To use PostgreSQL in a more isolated way, you may use Docker as an alternative:

```
$ docker run --rm -it -p 5432:5432 postgres:10
```

### Initial setup

We need to create a running database and a user with write access. For these and other database related actions, consider one of the following options:

- prepare manually using the database's interactive prompt
- prepare and execute `.sql` files
- use migration tool
- use default database/user `postgres`

Whichever option you choose, make sure you remember the corresponding values to put them into a config file. Here are example commands to run manually

```
$ psql -U postgres -h localhost
> CREATE DATABASE aiohttpdemo_polls;
> CREATE USER aiohttpdemo_user WITH PASSWORD 'aiohttpdemo_pass';
> GRANT ALL PRIVILEGES ON DATABASE aiohttpdemo_polls TO aiohttpdemo_user;
```

Use the *psql* commands, `\l` and `\du`, to check results.

---

**Note:** If you decided to run the application from the repo's source code, this script ( `init_db.py` ) will create a database and running server, as well as create tables and populate them with sample data

```
$ python init_db.py
```

---

## 2.2 Getting started

Let's start with basic folder structure:

- project folder named `polls`. A root of the project. Run all commands from here.
- application folder named `aiohttpdemo_polls` inside of it
- empty file `main.py`. The place where web server will live

We need this nested `aiohttpdemo_polls` so we can put config, tests and other related files next to it.

It looks like this:

```
polls                    <-- [current folder]
└── aiohttpdemo_polls
    └── main.py
```

aiohttp server is built around `aiohttp.web.Application` instance. It is used for registering *startup/cleanup* signals, connecting routes etc.

The following code creates an application:

```
# aiohttpdemo_polls/main.py
from aiohttp import web

app = web.Application()
web.run_app(app)
```

Save it and start server by running:

```
$ python aiohttpdemo_polls/main.py
======== Running on http://0.0.0.0:8080 ========
(Press CTRL+C to quit)
```

Next, open the displayed link in a browser. It returns a `404: Not Found` error. To show something more meaningful than an error, let's create a route and a view.

## 2.3 Views

Let's start with the first views. Create the file `aiohttpdemo_polls/views.py` and add the following to it:

```
# aiohttpdemo_polls/views.py
from aiohttp import web

async def index(request):
    return web.Response(text='Hello Aiohttp!')
```

This `index` view is the simplest view possible in Aiohttp.

Now, we should create a route for this `index` view. Put the following into `aiohttpdemo_polls/routes.py`. It is a good practice to separate views, routes, models etc. You'll have more of each file type, and it is nice to group them into different places:

```
# aiohttpdemo_polls/routes.py
from views import index

def setup_routes(app):
    app.router.add_get('/', index)
```

We should add a call to the `setup_routes` function somewhere. The best place to do this is in `main.py`:

```
# aiohttpdemo_polls/main.py
from aiohttp import web
from routes import setup_routes

app = web.Application()
setup_routes(app)
web.run_app(app)
```

Start server again using `python aiohttpdemo_polls/main.py`. This time when we open the browser we see:

```
Hello Aiohttp!
```

**Success!** Now, your working directory should look like this:

```
.
├── ..
```

(continues on next page)

(continued from previous page)

```
└── polls
    └── aiohttpdemo_polls
        ├── main.py
        ├── routes.py
        └── views.py
```

## 2.4 Configuration files

**Note:** aiohttp is configuration agnostic. It means the library does not require any specific configuration approach, and it does not have built-in support for any config schema.

Please note these facts:

1. 99% of servers have configuration files.
2. Most products (except Python-based solutions like Django and Flask) do not store configs with source code.

   For example Nginx has its own configuration files stored by default under `/etc/nginx` folder.

   MongoDB stores its config as `/etc/mongodb.conf`.
3. Config file validation is a good idea. Strong checks may prevent unnecessary errors during product deployment.

Thus, we **suggest** to use the following approach:

1. Push configs as `yaml` files (`json` or `ini` is also good but `yaml` is preferred).
2. Load `yaml` config from a list of predefined locations, e.g. `./config/app_cfg.yaml`, `/etc/app_cfg.yaml`.
3. Keep the ability to override a config file by a command line parameter, e.g. `./run_app --config=/opt/config/app_cfg.yaml`.
4. Apply strict validation checks to loaded dict. trafaret, colander or JSON schema are good candidates for such job.

One way to store your config is in folder at the same level as *aiohttpdemo_polls*. Create a `config` folder and config file at desired location. E.g.:

```
.
├── ..
└── polls                    <-- [BASE_DIR]
    │
    ├── aiohttpdemo_polls
    │   ├── main.py
    │   ├── routes.py
    │   └── views.py
    │
    └── config
        └── polls.yaml       <-- [config file]
```

Create a `config/polls.yaml` file with meaningful option names:

```
# config/polls.yaml
postgres:
  database: aiohttpdemo_polls
```

(continues on next page)

(continued from previous page)

```
  user: aiohttpdemo_user
  password: aiohttpdemo_pass
  host: localhost
  port: 5432
  minsize: 1
  maxsize: 5
```

Install `pyyaml` package:

```
$ pip install pyyaml
```

Let's also create a separate `settings.py` file. It helps to leave `main.py` clean and short:

```
# aiohttpdemo_polls/settings.py
import pathlib
import yaml

BASE_DIR = pathlib.Path(__file__).parent.parent
config_path = BASE_DIR / 'config' / 'polls.yaml'

def get_config(path):
    with open(path) as f:
        config = yaml.safe_load(f)
    return config

config = get_config(config_path)
```

Next, load the config into the application:

```
# aiohttpdemo_polls/main.py
from aiohttp import web

from settings import config
from routes import setup_routes

app = web.Application()
setup_routes(app)
app['config'] = config
web.run_app(app)
```

Now, try to run your app again. Make sure you are running it from `BASE_DIR`:

```
$ python aiohttpdemo_polls/main.py
======== Running on http://0.0.0.0:8080 ========
(Press CTRL+C to quit)
```

For the moment nothing should have changed in application's behavior. But at least we know how to configure our application.

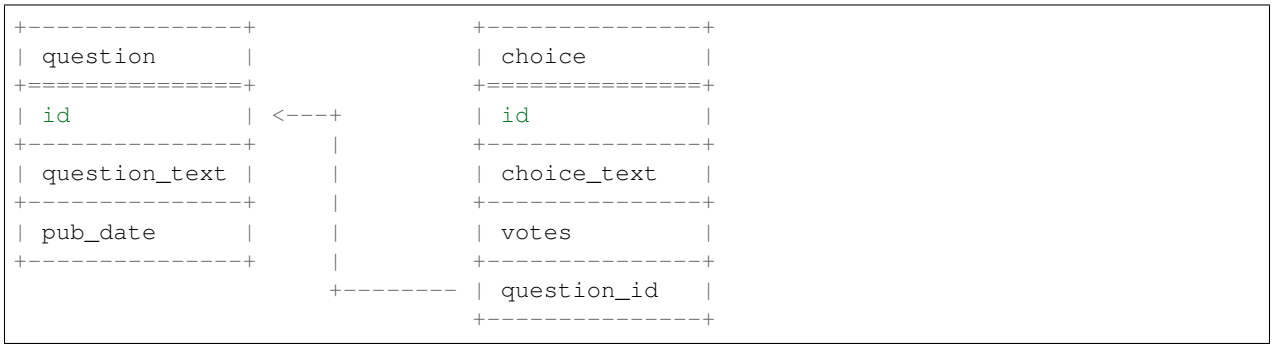
## 2.5 Database

### 2.5.1 Server

Here, we assume that you have running database and a user with write access. Refer to *Database* for details.

### 2.5.2 Schema

We will use SQLAlchemy to describe database schema for two related models, `question` and `choice`:




Create `db.py` file with database schemas:

```
# aiohttpdemo_polls/db.py
from sqlalchemy import (
    MetaData, Table, Column, ForeignKey,
    Integer, String, Date
)

meta = MetaData()

question = Table(
    'question', meta,

    Column('id', Integer, primary_key=True),
    Column('question_text', String(200), nullable=False),
    Column('pub_date', Date, nullable=False)
)

choice = Table(
    'choice', meta,

    Column('id', Integer, primary_key=True),
    Column('choice_text', String(200), nullable=False),
    Column('votes', Integer, server_default="0", nullable=False),

    Column('question_id',
           Integer,
           ForeignKey('question.id', ondelete='CASCADE'))
)
```

**Note:** It is possible to configure tables in a declarative style like so:

```
class Question(Base):
    __tablename__ = 'question'

    id = Column(Integer, primary_key=True)
    question_text = Column(String(200), nullable=False)
    pub_date = Column(Date, nullable=False)
```

But it doesn't give much benefits later on. SQLAlchemy ORM doesn't work in asynchronous style and as a result `aiopg.sa` doesn't support related ORM expressions such as `Question.query`.

`filter_by(question_text='Why').first()` or `session.query(TableName).all()`.

You still can make `select` queries after some code modifications:

```
from sqlalchemy.sql import select
result = await conn.execute(select([Question]))
```

instead of

```
result = await conn.execute(question.select())
```

But it is not as easy to deal with as update/delete queries.

---

Now we need to create tables in database as it was described with sqlalchemy. Helper script can do that for you. Create a new file `init_db.py` in project's root:

```
# polls/init_db.py
from sqlalchemy import create_engine, MetaData

from aiohttpdemo_polls.settings import config
from aiohttpdemo_polls.db import question, choice


DSN = "postgresql://{user}:{password}@{host}:{port}/{database}"

def create_tables(engine):
    meta = MetaData()
    meta.create_all(bind=engine, tables=[question, choice])


def sample_data(engine):
    conn = engine.connect()
    conn.execute(question.insert(), [
        {'question_text': 'What\'s new?',
         'pub_date': '2015-12-15 17:17:49.629+02'}
    ])
    conn.execute(choice.insert(), [
        {'choice_text': 'Not much', 'votes': 0, 'question_id': 1},
        {'choice_text': 'The sky', 'votes': 0, 'question_id': 1},
        {'choice_text': 'Just hacking again', 'votes': 0, 'question_id': 1},
    ])
    conn.close()


if __name__ == '__main__':
    db_url = DSN.format(**config['postgres'])
    engine = create_engine(db_url)

    create_tables(engine)
    sample_data(engine)
```

---

**Note:** A more advanced version of this script is mentioned in *Database* notes.

---

Install the `aiopg[sa]` package (it will pull `sqlalchemy` alongside) to interact with the database, and run the script:

```
$ pip install aiopg[sa]
$ python init_db.py
```

**Note:** At this point we are not using any async features of the package. For this reason, you could have installed `psycopg2` package. Though since we are using sqlalchemy, we also could switch the type of database server.

Now there should be one record for *question* with related *choice* options stored in corresponding tables in the database.

Use `psql`, `pgAdmin` or any other tool you like to check database contents:

```
$ psql -U postgres -h localhost -p 5432 -d aiohttpdemo_polls
aiohttpdemo_polls=# select * from question;
 id | question_text |  pub_date
----+---------------+------------
  1 | What's new?   | 2015-12-15
(1 row)
```

## 2.6 Doing things at startup and shutdown

Sometimes it is necessary to configure some component's setup and tear down. For a database this would be the creation of a connection or connection pool and closing it afterwards.

Pieces of code below belong to `aiohttpdemo_polls/db.py` and `aiohttpdemo_polls/main.py` files. Complete files will be shown shortly after.

### 2.6.1 Creating connection engine

For making DB queries we need an engine instance. Assuming `conf` is a `dict` with the configuration info for a Postgres connection, this could be done by the following async generator function:

```
async def pg_context(app):
    conf = app['config']['postgres']
    engine = await aiopg.sa.create_engine(
        database=conf['database'],
        user=conf['user'],
        password=conf['password'],
        host=conf['host'],
        port=conf['port'],
        minsize=conf['minsize'],
        maxsize=conf['maxsize'],
    )
    app['db'] = engine

    yield

    app['db'].close()
    await app['db'].wait_closed()
```

Add the code to `aiohttpdemo_polls/db.py` file.

The best place for connecting to the DB is using the `cleanup_ctx` signal:

```
app.cleanup_ctx.append(pg_context)
```

On startup, the code is run until the `yield`. When the application is shutdown the code will resume and close the DB connection.

**Note:** We could also have used separate startup/shutdown functions with the `on_startup` and `on_cleanup` signals. However, a cleanup context ties the 2 parts together so that the DB can be correctly shutdown even if an error occurs in another startup step.

### 2.6.2 Complete files with changes

```
# aiohttpdemo_polls/db.py
import aiopg.sa
from sqlalchemy import (
    MetaData, Table, Column, ForeignKey,
    Integer, String, Date
)

__all__ = ['question', 'choice']

meta = MetaData()

question = Table(
    'question', meta,

    Column('id', Integer, primary_key=True),
    Column('question_text', String(200), nullable=False),
    Column('pub_date', Date, nullable=False)
)

choice = Table(
    'choice', meta,

    Column('id', Integer, primary_key=True),
    Column('choice_text', String(200), nullable=False),
    Column('votes', Integer, server_default="0", nullable=False),

    Column('question_id',
           Integer,
           ForeignKey('question.id', ondelete='CASCADE'))
)


async def pg_context(app):
    conf = app['config']['postgres']
    engine = await aiopg.sa.create_engine(
        database=conf['database'],
        user=conf['user'],
        password=conf['password'],
        host=conf['host'],
        port=conf['port'],
        minsize=conf['minsize'],
        maxsize=conf['maxsize'],
    )
```

(continues on next page)

(continued from previous page)

```
    app['db'] = engine

    yield

    app['db'].close()
    await app['db'].wait_closed()
```

```
# aiohttpdemo_polls/main.py
from aiohttp import web

from settings import config
from routes import setup_routes
from db import pg_context

app = web.Application()
app['config'] = config
setup_routes(app)
app.cleanup_ctx.append(pg_context)
web.run_app(app)
```

Since we now have database connection on start - let's use it! Modify index view:

```
# aiohttpdemo_polls/views.py
from aiohttp import web
import db


async def index(request):
    async with request.app['db'].acquire() as conn:
        cursor = await conn.execute(db.question.select())
        records = await cursor.fetchall()
        questions = [dict(q) for q in records]
        return web.Response(text=str(questions))
```

Run server and you should get list of available questions (one record at the moment) with all fields.

## 2.7 Templates

For setting up the template engine, we install the `aiohttp_jinja2` library first:

```
$ pip install aiohttp_jinja2
```

After installing, setup the library:

```
# aiohttpdemo_polls/main.py
from aiohttp import web
import aiohttp_jinja2
import jinja2

from settings import config, BASE_DIR
from routes import setup_routes
from db import pg_context

app = web.Application()
```

(continues on next page)

(continued from previous page)

```
app['config'] = config
aiohttp_jinja2.setup(app,
    loader=jinja2.FileSystemLoader(str(BASE_DIR / 'aiohttpdemo_polls' / 'templates')))
setup_routes(app)
app.cleanup_ctx.append(pg_context)
web.run_app(app)
```

As you can see from setup above - templates should be placed at `aiohttpdemo_polls/templates` folder.

Let's create simple template and modify index view to use it:

```
<!--aiohttpdemo_polls/templates/index.html-->
{% set title = "Main" %}

{% if questions %}
    <ul>
    {% for question in questions %}
        <li>{{ question.question_text }}</li>
    {% endfor %}
    </ul>
{% else %}
    <p>No questions are available.</p>
{% endif %}
```

Templates are a very convenient way for web page writing. If we return a dict with page content, the `aiohttp_jinja2.template` decorator processes the dict using the jinja2 template renderer.

```
# aiohttpdemo_polls/views.py
import aiohttp_jinja2
import db

@aiohttp_jinja2.template('index.html')
async def index(request):
    async with request.app['db'].acquire() as conn:
        cursor = await conn.execute(db.question.select())
        records = await cursor.fetchall()
        questions = [dict(q) for q in records]
        return {"questions": questions}
```

Run the server and you should see a question decorated in html list element.

Let's add more views:

```
@aiohttp_jinja2.template('detail.html')
async def poll(request):
    async with request.app['db'].acquire() as conn:
        question_id = request.match_info['question_id']
        try:
            question, choices = await db.get_question(conn,
                                                      question_id)
        except db.RecordNotFound as e:
            raise web.HTTPNotFound(text=str(e))
        return {
            'question': question,
            'choices': choices
        }
```

## 2.8 Static files

Any web site has static files such as: images, JavaScript sources, CSS files

The best way to handle static files in production is by setting up a reverse proxy like NGINX or using CDN services.

During development, handling static files using the aiohttp server is very convenient.

Fortunately, this can be done easily by a single call:

```
def setup_static_routes(app):
    app.router.add_static('/static/',
                          path=PROJECT_ROOT / 'static',
                          name='static')
```

where `project_root` is the path to the root folder.

## 2.9 Middlewares

Middlewares are stacked around every web-handler. They are called before the handler for a pre-processing request. After getting a response back, they are used for post-processing the given response.

A common use of middlewares is to implement custom error pages. Example from Middlewares documentation will render 404 errors using a JSON response, as might be appropriate for a REST service.

Here we'll create a little bit more complex middleware custom display pages for *404 Not Found* and *500 Internal Error*.

Every middleware should accept two parameters, a *request* and a *handler*, and return the *response*. Middleware itself is a *coroutine* that can modify either request or response:

Now, create a new `middlewares.py` file:

```
# middlewares.py
import aiohttp_jinja2
from aiohttp import web


async def handle_404(request):
    return aiohttp_jinja2.render_template('404.html', request, {}, status=404)


async def handle_500(request):
    return aiohttp_jinja2.render_template('500.html', request, {}, status=500)


def create_error_middleware(overrides):

    @web.middleware
    async def error_middleware(request, handler):
        try:
            return await handler(request)
        except web.HTTPException as ex:
            override = overrides.get(ex.status)
            if override:
                return await override(request)
```

(continues on next page)

(continued from previous page)

```
            raise
        except Exception:
            request.protocol.logger.exception("Error handling request")
            return await overrides[500](request)

    return error_middleware


def setup_middlewares(app):
    error_middleware = create_error_middleware({
        404: handle_404,
        500: handle_500
    })
    app.middlewares.append(error_middleware)
```

As you can see, we do nothing *before* the web handler. In the case of an `HTTPException`, we use the Jinja2 template renderer based on `ex.status` *after* the request was handled. For other exceptions, we log the error and render our 500 template. Without the `create_error_middleware` function, the same task would take us many more `if` statements.

We have registered middleware in `app` by adding it to `app.middlewares`.

Now, add a `setup_middlewares` step to the main file:

```
# aiohttpdemo_polls/main.py
from aiohttp import web

from settings import config
from routes import setup_routes
from middlewares import setup_middlewares

app = web.Application()
setup_routes(app)
setup_middlewares(app)
app['config'] = config
web.run_app(app)
```

Run the app again. To test, try an invalid url.