cob Documentation

Release 0.24.1

Rotem Yaari

Feb 09, 2022

Contents

CHAPTER 1

What is Cob?

Everyone knows how to build a web application in theory. Most resources and tutorials go as far as a basic "hello world" webapp, and leave many details up for the developer to find out. Almost all serious webapps need to take care of deployment, configuration, background tasks, DB models/connections/migrations, static files location and configuration, and much much more.

Cob is a framework for building a solid scaffold for your webapps, and takes care of intertwining the various parts of a robust webapp for you. It also tries to leave as much of the glue code as possible inside Cob itself, so you can avoid as much boilerplate code in your own project as possible.

The core webapp backend part of Cob is based on Flask, but other features are flexible and can be easily tailored to any specific needs.

Cob is relatively opinionated about how projects should be structured, and provides utilities for generating the various components out of well-known templates.

Cob is greatly inspired by *Ember CLI*, which also aimed at solving a similar problem.

Contents:

1.1 Getting Started

In this section we will explore building a complete webapp with Cob. We will cover the most frequent tasks often tackled when starting an app from scratch, and will also cover how to deploy your app once it's ready for prime time.

Our examples will focus around building a TODO app, which will have all the parts a modern full-stack app has: a modern front-end UI, a backend for API calls, static files and assets and database integration.

1.1.1 First Steps

Prerequisites

Cob requires Python 3.6 in order to run, so you'll have to install it on your system. For docker-related operations such as building images, pushing them etc., you'll also need `docker` and `docker-compose` installed. Due to format

changes made in `docker-compose` in the past, you'll need version 1.13 or greater of `docker-compose`.

For deployment requirements, see *the relevant section in the deployment documentation*.

Installing Cob

To get started with cob, you will need to have cob installed on your system. Cob is basically a Python package, and as such - is most easily installed via `pip`:

```
$ pip install cob
```

The remainder of this tutorial will make extensive use of the `cob` command, which is the entry point for the various operations we will be using.

Creating your Project

Any Cob project lives in its own directory with a conventional structure. Although an empty project is very simple and minimalistic, Cob helps us create it through the `cob generate` command. `cob generate` always receives the type of thing we want to generate (in this case a *project*), and the name of what we're creating:

```
$ cob generate project todos
```

The above command will create a new project directory called **todos** which we are going to use for our project. A minimalistic cob project is just a directory with a special YAML file, called `.cob-project.yml`.

We will be returning to this file very soon enough, so don't worry.

Tip: Now would be a good time to start tracking your project with *git*:

```
$ cd todos
$ git init
$ git add .
$ git commit -m 'Initial version'
```

Creating a Simple Backend

Our application will need a backend serving our JSON API for fetching and updating TODO items. Cob is built around the Flask web framework and leverages its composability, while removing the need for boilerplate code.

Let's start by adding our route for serving TODO items. Let's create a file named `backend.py`, and add the following content to it:

```
# cob: type=views mountpoint=/api

from cob import route
from flask import jsonify

@route('/todos')
def get_all():
    return jsonify({'data': []})
```

Let's break down the example to see what's going on. The first line is a special one, used by cob to let it know how this file should be dealt with. It tells Cob that the type of this file is "views" (meaning a collection of Flask view functions), and that it should be "mounted" under `/api` in the resulting webapp.

Next, we import `route` from Cob. This is a shortcut to the `app.route` or `blueprint.route` in Flask, but is managed by Cob, and allows it to "discover" the routes that each file provides.

Running the Test Server

Before we move on, let's give our little app a try. Go into your project directory and run `cob testserver`. If this is the first time you're doing this in your project, Cob will initialize the private project environment for you first. For more information about how this works you can read about it *here*.

After the setup is done, your server will run on the default port:

```
$ curl http://127.0.0.1/api/todos
{
    "data": []
}
```

Note: `cob testserver` only runs the Flask backend serving your routes. Later on we will see how to set up a more complete testing environment

Interlude – What Have We Just Created?

Files of the like of `backend.py` above are very critical in Cob's functionality. Cob takes away the boilerplate in your project – but in order to do so it needs to know the parts that constitute your project. Each such part is loaded separately, and its dependencies are automatically taken care of by cob.

These "pieces" of your project are called **grains** in cob. Each grain has a *type* (for instance we have just created a *views grain*), telling Cob how to handle it. As we move forward we will meet more types of grains that we can use in Cob.

1.1.2 Working with Data

Most web applications work on data, usually in the form of records in a database. This is one of the most "boilerplate"-ish tasks in backend development, so naturally Cob aims at simplifying it as much as possible.

Cob takes care of loading models from your project, and also takes care of connecting to your database and migrations. Let's take a coser look at how it's done as we improve our app to actually keep track of todo tasks.

Adding Models

Our first step will be to add a model for our Todos. We'll use `cob generate` again to generate our models file:

```
$ cob generate models
```

This will create a file named `models.py` in our project directory. The file already imports the db component of cob, onto which we can define models:

```
$ cat models.py
# cob: type=models
from cob import db
```

Models grains use Flask-SQLAlchemy for defining models. Let's create our task model:

```
...
class Task(db.Model):

    id = db.Column(db.Integer, primary_key=True)
    description = db.Column(db.Text, nullable=False)
    done = db.Column(db.Boolean, default=False)
```

Note: By default, Cob uses an SQLite database located in the project's *.cob* directory for development, and switches to use Postgres for production (e.g. when being deployed as a docker container). In some cases you may want to use Postgres during development as well. In such cases, make sure you have a local Postgres instance running, and change the relevant DB URL to point at it. This can be done by adding the following to *.cob-project.yml* under your project root directory:

```
flask_config:
    SQLALCHEMY_DATABASE_URI: postgres://localhost/your_db
```

Initializing Migrations

We would like Cob to manage migrations for us, which will be useful when the time comes to modify and evolve our app, even after it's already deployed. Cob allows us to easily create a migration for our data. First we will initialize the migrations data (only needs to happen once):

```
$ cob migrate init
```

Then we will create our automatic migration script:

```
$ cob migrate revision -m "initial revision"
$ cob migrate up
```

Note: `cob migrate` uses Alembic for migration management. you can refer to Alembic's documentation for more information on how to customize your migration scripts

Note: Cob is configured, by default, to use an sqlite file under `.cob/db.sqlite`. See *the database section* to learn more on how it can be configured

Using Models in our Backend API

Using models is very simple now that we have defined our model. Let's go back to our `backend.py` file and modify it to load and store our tasks from the database:

```
# cob: type=views mountpoint=/api

from cob import route
from flask import jsonify, request

from .models import db, Task


@route('/tasks')
def get_all():
    return jsonify({
        'data': [
            _serialize(task)
            for task in Task.query.all()
        ]})

@route('/tasks', methods=['POST'])
def create_todo():
    data = request.get_json()['data']
    task = Task(
        description=data['attributes']['description']
    )
    db.session.add(task)
    db.session.commit()
    return jsonify(_serialize(task))


def _serialize(task):
    return {
        'type': 'task',
        'id': task.id,
        'attributes': {
            'description': task.description,
        }
    }
```

We now have a working, simple TODO app, with a REST API to add and view tasks.

1.1.3 Testing

Now that our app is beginning to grow some logic, it's time to start adding tests. Cob makes testing easy with the help of **pytest** and several related tools.

Let's add our first test – create a directory called `tests` under your project root, and create your first test file – let's name it `test_todo.py`:

```
def test_add_todo(webapp):
    message = 'some message'
    webapp.post('/api/tasks', data_json={
        'data': {
            'attributes': {
                'description': message,
            }
        }})
    all_todos = webapp.get('/api/todos')['data']
    last_todo = all_todos[-1]['attributes']
    assert last_todo['description'] == message
```

We wrote a single test function for use in **pytest**, with a single fixture called webapp, which is an instance of `cob.utils.unittest.Webapp`, a helper Cob exposes for tests.

To run our tests, all we need to do is run `cob test` from our project root.

Tip: `cob test` is just a shortcut for running **pytest** in your project. All options and arguments are forwarded to pytest for maximum flexibility.

1.1.4 Adding Third-Party Components

Cob is aimed at being the backbone of your webapp. Most web applications eventually need to bring in and use third party components or libraries, and Cob makes that easy to do.

We are going to improve our Todo app by using a third-part tool for serialization, marshmallow. The first order of business is to get Cob to install this dependency whenever our project is being bootstrapped. This can be done easily by appending it to the `deps` section of `.cob-project.yml`:

```
# .cob-project.yml
...
deps:
    - marshmallow
```

Now we can use this library to serialize our data, for instance create a file named `schemas.py` with the following:

```
from marshmallow import Schema, fields, post_dump, post_load, pre_load
from .models import Task


class JSONAPISchema(Schema):

    @post_dump(pass_many=True)
    def wrap_with_envelope(self, data, many): # pylint: disable=unused-argument
        return {'data': data}

    @post_dump(pass_many=False)
    def wrap_objet(self, obj):
        return {'id': obj.pop('id'), 'attributes': obj, 'type': self.Meta.model.__
↪name__.lower()}

    @post_load
    def make_object(self, data):
        return self.Meta.model(**data)

    @pre_load
    def preload_object(self, data):
        data = data.get('data', {})
        returned = dict(data.get('attributes', {}))
        returned['id'] = data.get('id')
        return returned


class TaskSchema(JSONAPISchema):
    id = fields.Integer(dump_only=True)
    description = fields.Str(required=True)
    done = fields.Boolean()
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = Task

tasks = TaskSchema(strict=True)
```

And use it in `backend.py`:

```
from cob import route
from flask import jsonify, request

from .models import db, Task
from .schemas import tasks as tasks_schema

@route('/tasks')
def get_all():
    return jsonify(tasks_schema.dump(Task.query.all(), many=True).data)

@route('/tasks', methods=['POST'])
def create_todo():
    json = request.get_json()
    if json is None:
        return "No JSON provided", 400

    result = tasks_schema.load(json)
    if result.errors:
        return jsonify(result.errors), 400
    db.session.add(result.data)
    db.session.commit()
    return jsonify(tasks_schema.dump(result.data).data)
```

1.1.5 Building a UI

Cob makes it easy to integrate front-end code in the same repository as your backend, and helps you build, test and deploy it too.

Setting Up

In our example we will be using Ember to use our UI. We'll start by creating our front-end grain:

```
$ cob generate grain --type frontend-ember webapp
```

Note: In order for the above to work, you need to have Ember CLI installed on your system

This will bootstrap your `webapp` subdirectory with our UI code, and take care of initial setup.

While this looks like black magic, what happens here is quite simple - Cob creates a directory called `webapp`, and places a `.cob.yml` file inside it, letting Cob know that this is a grain of type `frontend-ember`:

```
# In webapp/.cob.yml
type: frontend-ember
```

The `.cob.yml` file is just a different way to write the markup we used in the first comment line of our previous grains. Once we marked our webapp directory in this way, Cob knows how to treat it as one containing Ember front-end code.

Writing our Front-end Logic

We won't dive in to how to develop using Ember, so we'll just create a minimal front-end app for displaying and adding our TODOs.

Note: We won't cover Ember here in depth – for more information you can refer to the excellent Ember Guides. For now, just take our word for it

---

```
// webapp/app/routes/tasks.js
import Ember from 'ember';

export default Ember.Route.extend({

    model() {
        return this.store.findAll('task');
    },

    setupController(controller, model) {
        this._super(...arguments);
        controller.set('tasks', model);
    },
});
```

```
// webapp/app/controllers/tasks.js
import Ember from 'ember';

export default Ember.Controller.extend({

    new_task: '',

    actions: {
        add_task() {
            let task = this.store.createRecord('task', {
                description: this.get('new_task'),
            });
            task.save();
        }
    }
});
```

```
// webapp/app/models/task.js
import DS from 'ember-data';

export default DS.Model.extend({

    description: DS.attr(),
    done: DS.attr('boolean'),
});
```

```
// webapp/app/adapters/application.js
import DS from 'ember-data';
```

(continues on next page)

(continued from previous page)

```
export default DS.JSONAPIAdapter.extend({
    namespace: '/api',
});
```

And finally our template:

```
<!-- webapp/app/templates/tasks.hbs -->
{{#each tasks as |task| }}
  <div class="task">
    <h3>{{task.description}}</h3>
  </div>
{{/each}}

{{input value=new_task}}
<button {{action "add_task"}}>Add</button>
```

Developing Front-end and Backend Simultaneously

Now that we have multiple components to track during development (our Flask app and our Front-end compilation) we can make use of yet another handy tool Cob provides for us: `cob develop`:

```
$ cob develop
```

This command will fire up `tmux` (you'll have to have it installed beforehand though), with two windows – one for running the backend server and the other running `ember build --watch` to compile your front-end. Cool huh?

1.1.6 Deploying your Application

Cob uses **Docker** for deploying apps. It is the best way to guarantee reproducible, composable setups and also allow reuse between development and deployment.

Cob separates deployment to two stages: building your deployment image and running it.

Building your Application Image

From your project directory, run:

```
$ cob docker build
```

This will create a basic Docker image, labeled `todos` by default (Cob uses the app name from the project's configuration to name to label its images), and that image will be later on used for running your app.

Running your Application in Deployment

To run your app via `docker-compose`, running its various pieces properly linked, run:

```
$ cob docker run
```

This will start your app in the foreground.

See also:

For more information on deploying your apps with Cob, see the *Deployment* section of the docs

1.2 Basic Grains

1.2.1 Views Grains

Views is the simplest type of grain, and it contains plain Flask route functions.

To generate a new views grain, run:

```
$ cob generate grain myview --type views
```

Views grains normally look like this:

```
$ cat myview.py
```

```
# cob: type=views mountpoint=/

from cob import route

@route('/sayhi')
def hi():
   return 'hi'
```

1.2.2 Blueprint Grains

Blueprint grains are very similar to views grains, but allow a more direct control over the blueprint they create:

```
$ cob generate grain blueprint --type blueprint
...
$ cat blueprint.py
```

```
# cob: type=blueprint mountpoint='/bp'
from flask import Blueprint

blueprint = Blueprint(...)
```

Note: You do not have to name your blueprint variable `blueprint`. Cob will look for your Blueprint instance among the module's globals

1.2.3 Template Grains

Template grains are used to host Flask (Jinja2) templates:

```
$ cob generate grain templates --type templates
...
```

You can now create a basic template under `templates/index.html`. Rendering it is fairly simple:

```
$ cat index.py
```

```
# cob: type=views mountpoint=/
from cob import route
from flask import render_template

@route('/')
def index():
    return render_template('index.html')
```

1.2.4 Tasks Grains

Tasks grains are used to run Asyc tasks:

```
$ cob generate grain tasks --type tasks
...
$ cat tasks.py
```

```
# cob: type=tasks mountpoint={{mountpoint}}
from cob import task, periodic_task
from celery.schedules import crontab

## Your tasks go here

# use the next syntax for tasks you'll send "manually" through REST API
@task()
def task_func1(x,y,z ...):
        ...

# use this syntax to set a task binded to an non-default queue ("celery").
# queue arg can also be used with the periodic_task decorators.
@task(queue=<queue name>)
def task_func2(...):
        ...

# use this syntax to create a task that will be dispached every predefined period.
@periodic_task(every=<num of seconds>)
def periodic_task_func3(...):
        ...

schedule_dict = {
    '<choose a name>': {
             'schedule': crontab(...) or <number of seconds>,
             'args': <tuple of arguments according to the args your task_func needs>
    }
 }

 # use this syntax to create a more "complex" schedule for the task
 @periodic_task(schedule=schedule_dict)
 def periodic_task_func4(...):
        ...
```

1.2.5 Bundle Grains

In some cases, as we'll see later, you may want to bundle several grains into a single directory. However, Cob only searches for grains in the project root by default.

Luckily you can create a directory and declare it as a bundle to tell Cob it should traverse the top level of that directory as well:

```
$ cob generate grain my_addon --type bundle
$ cat my_addon/.cob.yml
type: bundle
$ cob generate grain my_addon/index.py --type views
...
```

1.3 Working with Static Files

Managing static assets can be a pain, especially during deployment and testing. Cob can be instructed to look for static files in several places, including custom locations

1.3.1 Static Grains

Static grains are just like regular grains, but are used to store static files.

To create static grains:

```
$ cob generate grain staticfiles --type static
$ cat staticfiles/.cob.yml
type: static
root: ./root
mountpoint: /static
```

This instructs cob to look for static files under the `root` subdirectory of the `staticfiles` directory, when accessed via the `/static/` URL prefix.

Note: We can instruct the root directory to become `.`, instead of `./root`, but this would expose `.cob.yml` itself to be fetched as a static file

---

1.4 Working with Databases

1.4.1 Models Grains

Models grains provide the ability to define your models to be used by your applications. Their structure is very straightforward, and relies on SQLAlchemy:

```
$ cob generate grain my_models --type models
```

Then you can add your models in my_models.py:

```
# cob: type=models
from cob import db

class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

1.4.2 Controlling the Database URI

By default, Cob uses an SQLite database located in the project's *.cob* directory for development. In production (when deployed via docker) - it switches to Postgres.

In some cases, however, you may want to use Postgres during development as well, or move to use a different database altogether. For such cases, Cob supports setting the database URI explicitly through the following methods:

1. You can set your database URI in the configuration file. Simply set `SQLALCHEMY_DATABASE_URI` config value in your `.cob-project.yml` file:

   ```
   name: myproj
   flask_config:
       SQLALCHEMY_DATABASE_URI: postgres://db-server/your-db
   ```

2. You can override the database URI used by Cob through the `COB_DATABASE_URI` environment variable. This method takes precedence over the local project configuration, and therefore is useful for overriding the database connection settings during testing or in CI scenarios.

1.4.3 Database Initialization

Note that Cob, by default, does not initialize any DB structure for you when run. This means that for your code to work, an extra step is required to make sure the database is properly initialized.

If your project **does not** use migrations, you can quickly create all tables and structures using:

```
$ cob db createall
```

However, the recommended practice for long-lasting projects is to use migrations. For projects using migrations, creating the DB is done via `cob migrate up`, which is described in the *Using Migrations* chapter of this guide.

Note: Cob takes care of migrations automatically during deployment - the statement above about having to initialize your database hold only for the development phase of your project.

1.5 Using Migrations

Cob makes it easy for you to create migrations for your data, and will help you to execute them during deployment and testing, as we will see later on.

Cob uses Alembic for database migration management - for more information please refer to Alembic's docs.

1.5.1 Initializing the Migrations Directory

You will first need to create your migrations directory (this is a one-time operation)

```
$ cob migrate init
```

This will create a `migrations` directory, and within it `versions`, which will contain your migration scripts.

1.5.2 Create a Revision

```
$ cob migrate revision -m "My revision name"
```

This will compare the needed changes to bring your DB up-to-date. Please make sure to run it with a database that is already migrated to the latest revision.

1.5.3 Upgrade/Downgrade

Migrating up and down the revision tree is done via `migrate up` and `migrate down`:

```
$ cob migrate up
```

```
$ cob migrate down
```

1.6 Background Tasks

Cob lets you define background tasks with relative ease. Background tasks use *Celery* by default, and can be easily triggered from your code. Cob then takes care of setting up a worker and a broker for you during deployment.

1.6.1 Defining Tasks

Tasks are defined in Python files which are declared as *task grains*. You can create such a file yourself very easily - let's call it `tasks.py` and put it in your project's root directory. Start it with the following metadata line:

```
# cob: type=tasks
```

Cob exposes a utility decorator, letting you easily define tasks:

```
# cob: type=tasks
from cob import task

@task
def my_first_task():
    print('task here')
```

1.6.2 Triggering Tasks

Triggering a task is easy. Once defined in `tasks.py`, you can use them through a simple import:

```
### index.py
# cob: type=views
from cob import route
from .tasks import my_first_task
```

(continues on next page)

(continued from previous page)

```
@route('/')
def index():
    my_first_task.delay()
```

1.6.3 Requiring Application Context

By default, tasks are not run with an active application context, meaning they cannot interact with models or rely on the current Flask app. You can opt for having your tasks run with it by passing `use_app_context=True`:

```
# cob: type=tasks
from cob import task

@task(use_app_context=True)
def my_first_task():
    print('task here')
```

1.6.4 Configuring Celery

Queue Names

You can control the queue names which are loaded into the worker by adding `queue_names` to your grain config. For example:

```
# cob: type=tasks queue_names=queue1,queue2
...
```

Adding Worker Arguments

You can control arguments to the Celery workers by setting `celery.worker_args` in your project's config:

```
# .cob-project.yml
...
celery:
   ...
   worker_args: -c 4
```

1.7 Working with Front-end Code

1.7.1 Front-end Grains

Cob recognizes grains dedicated to front-end components, and offers tailored handling for them. The exact behavior depends on the type of framework you are looking to use.

Ember

Cob supports front-ends written in Ember (through ember-cli). It even lets you easily generate a new project:

```
$ cob generate grain --type frontend-ember ./webapp
```

This will create the new grain and even, if ember-cli is detected in your environment, run `ember init` and `npm install` for you.

Note: After generating the grain, you can use all regular `ember-cli` commands. You only have to make sure that you run them from within the grain directory – otherwise they will not work as expected.

The Mount Point

Ember front-end grains, much like many other types of grains, can declare their **mount point**, through the `.cob.yml` file:

```
type: frontend-ember
mountpoint: /app
```

This will make Cob serve your front-end code from the `/app` path of your webapp.

Note: for ember, the value of the `mountpoint` configuration option must be set in accordance with the `rootURL` of your Ember app's configuration (usually `config/environment.js`)

Handling `locationType`

Ember can utilize the web browser's *history API* to simulate browsing URLs directing to the same dynamic single-page app. This capability works great when navigating from the index through internal links, but requires additional support when linking to specific routes externally (as the web server needs to serve the same page even for other path segments).

Cob automatically detects the use of `locationType` in your `config/environment.js` and adjusts its behavior accordingly. Once your app uses `locationType: 'auto'` instead of `'hash'`, Cob will serve all pages under the mountpoint the same way, leading to the frontend code.

1.7.2 Developing with Front-end Compilation

Developing back-end and front-end code in tandem requires some additional ergonomics. Cob supports the `tmux` command for running a complete development environment inside tmux.

After creating your grains, run `cob develop` to start your tmux development session.

1.7.3 Working with Custom Node Versions

Cob installs Node 8.x by default on its docker images, but it can be instructed to use other versions as well. To override the node version being used, override it in your project's configuration:

```
# .cob-project.yml
...
frontend:
    node_version: 8
```

1.8 Services

Cob allows you to work with additional services, in a way that makes them available to your app and automatically set up during deployment.

1.8.1 Redis

To use redis in your app, include the following in your `services` key of your `.cob-project.yml`:

```
# .cob-project.yml
...
services:
  - redis
```

Then in your app, you can use `cob.services.redis` to obtain a working Redis client:

```
from cob import services
...
@cob.route('/')
def index():
    value = services.redis.get('key')
    ...
```

Note: During development, the Redis server is not started for you. Cob relies on the existence of a running Cob server on your local machine

1.9 Testing

Testing a cob application is easy, and can be done in two modes – undockerized and dockerized.

1.9.1 Undockerized Testing

Cob supports running tests against your project using the `pytest` testing framework. Cob exposes a default *fixture* for using your app called `webapp`.

To get started, simply create a `tests` folder inside your project, and create your first test file there. There is no extra cob-specific markup required:

```
# tests/test_view.py

def test_index_view(webapp):
    assert webapp.get('/') == {'some': 'json'}
```

Since the vast majority of web services use JSON for the response format, Cob parses it for you, and thus `get`, `post` etc. return a parsed JSON object by default.

You can override this behavior using `raw_response=True`, getting the actual response object (provided by the `requests` library). Note that in this mode you have to verify the success of your request yourself:

```
# tests/test_view.py
import requests

def test_something_fails(webapp):
    resp = webapp.post('/some/nonexistent/url')
    assert resp.status_code == requests.codes.not_found
```

If you'd like to import a module from your project inside your tests, use:

```
from _cob import <module_name>
```

For example:

```
# models.py

# cob: type=models
from cob import db


class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String)
    last_name = db.Column(db.String)

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"
```

```
# tests/test_models.py
from _cob import models

def test_person():
    p = models.Person(first_name="First", last_name="Last")
    assert p.full_name == "First Last"
```

1.9.2 Dockerized Testing

While the plain, undockerized mode is good for quick API tests and scenarios that don't require an entire environment set up, some tests do impose some constraints on the testing environment. The most notorious examples are ones involving background tasks, interaction with other services such as Redis, etc.

For this purpose Cob supports testing in "dockerized" mode. This mode is slower since it builds your docker-compose environment for you, but once run, you can rest assured that the tests work as if they were in production.

To run the tests in this mode, simply run `cob docker test`.

If needed, additional options can be used with `cob docker test`:

```
$ cob docker test -o <overlay compose file name> -d <service name to depend on>
```

- Overlay compose files for testing are meant to help you keep your testing env contained in case your app should communicate with the outside world. so during testing, you can load containers that mimik the extra "outside world services".
- If you used the overlay option, your app might depend on those extra services, so you can declare this dependency by stating the name of the service to depend on.

Note: Both options can be used multiple times so you can have more then one overlay compose file and you can specify as many depended services as needed.

Note: To use the overlay option, all the files you use should reside under tests/overlay_compose_files/.

1.10 Using Hooks

Cob offers specific hooks to control what happens during your application's lifecycle. Hooks are implmented through the gossip library, and are usually defined in a special file located at your project's root, `project.py`

1.10.1 Configuring your Flask App After Creation

You can use the `cob.after_configure_app` hook to add additional features or configurations to your app:

```
# project.py

import gossip


@gossip.register('cob.after_configure_app')
def after_configure_app(app):

    @app.before_request
    def before_request():
        ...
```

1.11 Project Configuration

Cob puts a strong emphasis on configuration, allowing you to control the various aspects of your project and other 3rd party tools.

1.11.1 Project Config

Assuming you need a place to store project-specific configuration values, you can use the `.cob-project.yml` file located at the root of your project.

Any configuration written in this file will be available to your project via the current project's `config` attribute. For instance, for the following contents of `.cob-project.yml`:

```
name: myproj
some_value: 2
```

You can access `some_value` through the `config` project attribute:

```
...
from cob.project import get_project
...
value = get_project().config['some_value']
```

Note: Some keys in the project config have special meaning, like the project name stored in the `name` key or `flask_config` described below. To avoid name clashes, it is wise to store all of your specific configuration under a specific key as a nested structure, such as `config` or `project-config`

1.11.2 Cob Config Options

- `pypi_index_url` - For use with pypi other than https://pypi.org/simple
- `specific_virtualenv_pkgs` - a string of the form 'pip==19.0.1 setuptools==40.6.3'. this will tell cob to udpate pip/setuptools versions inside cob's virtualenv before installing dependencies.

1.11.3 Managing Dependencies

Apart from the base dependencies needed by cob itself, which takes care of the facilities your project uses (this includes Flask, SQLAlchemy and Celery for example), you can specify additional dependencies your code relies on. This can be done using the `deps` configuration value:

```
# .cob-project.yml
...
deps:
    - requests>=1.1.0
```

1.11.4 Flask Config

You can easily add configuration to Flask's config by specifying it in the `flask_config` key of `.cob-project.yml`:

```
# .cob-project.yml
...
flask_config:
    SQLALCHEMY_DATABASE_URI: sqlite:////path/to/db.db
```

1.11.5 Configuration Loading and Overrides

In addition to the current project's `.cob-project.yml`, Cob looks in other places to load configuration snippets. This allows you to add overrides and overlays used either locally during development, or per-server for deployment.

The following locations are scanned for configuration overrides:

- `/etc/cob/conf.d/<project name>`
- `~/.config/cob/projects/<project name>`

This means that if you would like a project named `testme` to have a private piece of information in its config loaded during deployment and not stored in the source repository, all you have to do is add the following:

```
SOME_SECRET_CONFIG: 'secret'
```

In `/etc/cob/conf.d/testme/000-private.yml` on your server.

1.11.6 Deployment Configuration

Several aspects of Cob's deployment can be configured via the project's configuration files.

Gunicorn Options

By adding a `gunicorn` dictionary to your project's YAML file, you can control Gunicorn options directly:

```
# .cob-project.yml
gunicorn:
    max_requests: 20
```

Exposed Ports

When deploying via Docker, Cob automatically exposes port 80 for your webapp, but leaves other ports private.

You can customize this behavior using the `docker.exposed_ports` configuration. This configuration value is a mapping, containing the list of exposed ports for each service name:

```
# .cob-project.yml
docker:
    exposed_ports:
        wsgi:
            - 443
            - 12345
```

1.12 Deployment

Deploying Cob applications is easy and straightforward. Cob uses Docker for deployment – it helps you build a dockerized version of your app, which you can then push to a repository or deploy directly to the local machine.

1.12.1 Deployment Requirements

Cob has several requirements in order to be successfully deployed on your system/machine.

Python Version

Cob requires Python 3.6 or newer in order to install and run correctly. If you are deploying on Ubuntu 18.04 or newer, this should already be the case for your system.

For Ubuntu 16.04 or older, it is recommended to use the *deadsnakes* PPA for installing 3.6:

```
$ sudo apt-get install -y software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install -y python3.6
```

Docker and Docker-Compose

Deploying a cob app requires *docker* to be installed. In addition, `docker-compose` is needed.

Note: In some cases docker-compose is bundled with `docker` on your platform. However, due to a compatibility issue of the docker-compose format, Cob requires version 1.13 and above, which might not be the case for your installation.

In any case, it is recommended to follow the official guide for setting up `docker-compose`.

1.12.2 Building a Docker Image

First, you'll need to build the Docker image for your project. This is done by running:

```
$ cob docker build
```

This command builds a docker image labeled `<your project name>>:dev` by default. This means that if your project is named "todos", the image would be named `todos:dev`.

1.12.3 Testing Dockerized Apps

To make sure no new issues get introduced as a result of packaging your app through Docker, you can run your tests inside the docker containers comprising your app, meaning it will run very similarly to how it would run in production.

This is done by running `cob docker test`.

1.12.4 Tagging and Pushing Images

In most cases you would probably like to tag your released images and upload them to a Docker registry. This can be done by setting the *image name* for your project before building images.

Under `.cob-project.yml`, add the `docker.image_name` configuration:

```
# .cob-project.yml
...
docker:
    image_name: "your.server.com:4567/myproject"
```

Now when you build or test your project, the docker image created will be `your.server.com:4567/myproject:dev`

Once you're satisfied with a built image, you can tag it directly through docker as your "latest" version:

```
$ cob docker tag-latest
```

Then you can push your image to the repository with a standard `docker push` command:

```
$ cob docker push
```

Note: both `cob docker tag-latest` and `cob docker push` take the image name from the project's configuration, and are intended as shortcuts for `docker tag` and `docker push`.

1.12.5 Deploying on Systemd-based Systems

If your target machine is based on *systemd* (e.g. recent Ubuntu Server releases, CentOS 7.x etc.), you can deploy a dockerized cob project by running:

```
$ cob docker deploy your.server.com:4567/myproject:latest
```

This will pull off the needed information from the Docker image and create appropriate unit files to run your project.

Note: cob uses docker-compose for deployment. a docker-compose-override yml file can be provided which is then used as described in docker-compose overview.

To use the above capability, add `--compose-override <full path to override file>` to the above `cob docker deploy` command. You can use more than one compose-override files by repeating the flag for each file.

1.13 Developing Cob Apps

1.13.1 Using Cob inside Virtual Environments

Cob works by creating a virtual environment for its applications. This is necessary in order to ensure dependency installation and isolation from external environments. It determines the Python interpter to use for installing the virtual environment based on a set of rules, like trying to see where which Python interpreter is currently running it.

In case your current interpreter is already inside a virtual environment (a.k.a. `virtualenv`), Cob will attempt to fall back to the global Python interpreter. This is because there are usually multiple issues with trying to create a virtualenv from within another virtualenv. You can override this default behavior by setting the `COB_FORCE_CURRENT_INTERPRETER` environment variable to a non-empty value.

1.13.2 `cob develop`

This mode is slightly more useful for real-world apps including front-end components. `cob develop` starts a tmux session (named `cob-<your project name>` by default), with a single window for the test server, and additional windows for other components of your app. For instance, projects using *frontend grains*, will set up panes running the build/watch process for your assets.

Note: If you wish to test your flask app alone (without the surrounding components started in `cob develop`, you can run `cob testserver`. This will only run the backend Flask part of your webapp

---

1.14 The Cob Environment

1.14.1 Virtualenv and Dependencies

When you run a cob project, you use the `cob` console script belonging to your external environment, but it, in turn, creates an internal virtual environment (through `virtualenv`) to run your project. This is done automatically for you, and the resulting virtualenv is saved under `.cob/env` in your project root.

Cob takes care of only refreshing this environment when needed (e.g. when it is deleted or when new dependencies exist in your configuration, as described below).

Note: You can always use the `COB_REFRESH_ENV` environment variable to force cob to refresh your virtualenv:

```
$ COB_REFRESH_ENV=1 cob testserver
```

1.14.2 Additional Dependencies

You can install additional dependencies through the `deps` section of the `.cob-project.yml` file:

```
# .cob-project.yml
...
deps:
  - Flask-Security
  - Flask-SQLAlchemy>=0.1.0
```

1.14.3 Environment Variables

PYPI_INDEX_URL - Makes cob use an alternate pypi registry/index. COB_VERSION - Allows setting a specific version of Cob to be used. COB_USE_PRE - make cob run *pip install –pre*. affects testserver virtualenv, docker build during deployment. COB_DEVELOP - Indicates a development version of cob is used, affecting bootstrapping, dockerfile.

1.15 API Reference

1.15.1 Testing

1.16 Cob Commands

1.17 Changelog

- : **:feautre:‘121‘** Improve environment/config customization
- #139: Use POSTRES_HOST_AUTH_METHOD for postres configuration
- #122: Adding docker compose override files to `cob docker deploy --compose-override`
- : Fix Werkzeug’s ProxyFix import
- : Added automation documentaion for commands
- #116: Update PyAML version
- : Pin click version to < 8
- : Pin tmuxp version to < 2.0
- : Use yaspin instead of halo
- : Properly handle pypi index url environment variables

- : Added `-H` parameter to `cob testserver`, specifying the address to bind
- #105: Allow configuring node versions being used during docker image building
- #104: Clean up containers on docker test end
- : : Avoid setting up database if project doesn't have models
- : Added `--no-cache` option to `cob docker test` to support usage in CIs
- : Avoid running `rsync` in `cob docker test` if an image is built during the process
- : Fix pylint errors
- #101: Run migrations on cob docker test
- : Add IPython as a dependency
- : Pin pylint dependency
- : Added debug log output to `cob testserver`
- #90: Add `docker.exposed_ports` configuration for controlling exposed ports in deployment
- #89: Add `--force` to `cob docker deploy` to force overwriting unit files
- #88: Add option to specify more compose file to `cob docker run-image`
- #92: Use journald logging driver when available during docker execution
- #94: Added `cob docker tag-latest` to tag the recent image as latest, and `cob docker push` to push the latest image
- #97: Added `cob shell` command, allowing users to interactively access their modules and code through IPython or the builtin Python interpreter shell
- #96: Pin Celery dependency to 4.1.x because of 4.2.x regression
- #88: Support additional docker-compose files in `docker run-image` with `-o`
- #90: Add `docker.exposed_ports` configuration
- : Fix error formatting when docker could not be located
- #89: Add –force to cob docker deploy
- : Add logging to syslog by default
- : Fix escaping of image names
- #87: Add "cob version" command
- #84: Cob now mounts /etc/localtime inside containers to enforce correct time zone
- #85: Cob now supports symlinks for /etc/cob/conf.d/PROJNAME
- : Pin PyYaml to 3.x
- : Fix escaping of image names when using `cob docker deploy`
- #84: Cob now mounts /etc/localtime inside containers to enforce correct time zone
- #85: Cob now supports symlinks for /etc/cob/conf.d/PROJNAME
- #85: * Changelog
- #85: * Support 'cob docker deploy' command (closes #51)
- #85: * Use port 80 in cob docker deploy
- #85: * cob docker test now uses <project name>:dev image name by default

- : Change default build image to Python3.6-jessie
- #83: Add `docker.image_name` project configuration
- #67: Support redis
- #66: Support the `--image-name` parameter in `cob docker run` to override the image used
- #76: Support `celery.additional_args` to control additional worker arguments through configuration
- #50: `cob docker test` can now be used to run your tests inside a working docker-compose setup
- #77: Cob now required Python 3.6
- #82: Added *cob docker run-image* to run a prebuilt cob image without requiring dependencies
- #51: Support *cob docker deploy* command to conveniently deploy dockerized cob projects on systemd
- : Allow passing celery configuration in project yaml
- #21: Cob now uses multi-stage docker building to reduce image size and speed up the build process
- : Many small fixes and improvements
- #47: Cob now handles cases where docker requires sudo more elegantly
- #59: Front-end ember grains now run npm install
- #42: Cob now supports specifying the pypi index URL to use via *COB_INDEX_URL*
- #44: Allow specifying cob version to use via *COB_VERSION* environment variable
- #40: Added ability to make background tasks run in app context
- #43: Add option to pass arbitrary arguments to celery start-worker
- : First operational release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search