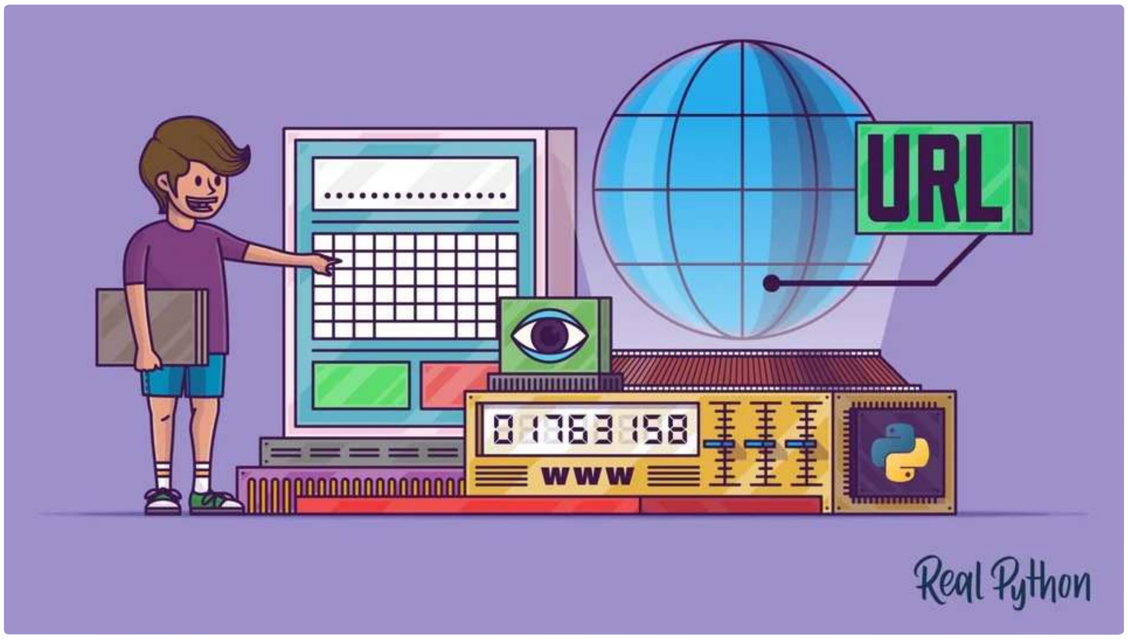


Python Web Applications with Flask – Part I

by Real Python Aug 14, 2013 32 Comments basics flask web-dev

Mark as Completed

Tweet Share Email

Table of Contents

Please note: This is a collaboration piece between Michael Herman, from Real Python, and Sean Vieira, a Python developer from De Deo Designs.

Help

Articles in this series:

1. **Part I: Application setup** ← *CURRENT ARTICLE*
2. Part II: Setup user accounts, Templates, Static files
3. Part III: Testing (unit and integration), Debugging, and Error handling

Updated: November 17th, 2013

Recently, thanks to an awesome gig, I re-introduced myself to Flask, which is "a microframework for Python". It is a great little framework; unfortunately, like Sinatra (which inspired it), making the transition from small "self-contained" applications to larger applications is difficult. There are a number of boilerplates out there (including my own Flask-Boilerplate) to help make the transition easier. However, using a boilerplate doesn't help us understand the "why" of the boilerplate's conventions.

After reviewing the available tutorials out there I thought, "there must be a better way" - then I saw Miguel Grinberg's mega-tutorial. The focus that building an *application* rather than a boilerplate gave to his articles impressed me. So in this tutorial series we are going to be building a mid-sized application of our own and as a side-effect generating a boilerplate. The hope is that when we are finished we will have a better appreciation for what Flask provides and what we can build around it.

What we are building

This series will be following the development of a web analytics solution called Flask-Tracking. The goal of this series is to have a working application that will enable users to:

- Register with the application.
- Add sites to their account.
- Install tracking codes on their site(s) to track various events as the events occur.
- View reports on their site(s)' event activity.

At the end of the day, *our* goal (over and above having a working application that conforms to the above napkin spec) is to:

- Learn how to develop a mid-sized application using Flask.
- Get practical experience with refactoring and testing.
- Gain a deeper appreciation of building composable systems.
- Develop a boilerplate that we can re-use for future projects.

Starting off

This series assumes that you have a passing familiarity with Python and Flask already. You should already be comfortable with everything in Python's tutorial, be familiar with the command line (or, at the very least, with `pip`), and you should have read Flask's Quickstart and Tutorial. If you want more practice, take it a step further and read my beginning tutorial on Flask here. That being said, if you are a newcomer to Python and Flask you should still be able to follow along.

Our goal today is to implement the core of our application - tracking visits to a site. We will allow multiple sites, but we will not worry about users or access control yet.

First principles

Help

Before we dive in I would like to touch on some of the principles that will guide our development - we will not be delving too deeply into the "whys" of some of these best practices, but where possible will provide links to articles that explain what these practices provide us. Please consider all of the opinions in this series to be strong opinions, weakly held.

The Zen of Python

If you did not already know, Python has a “development philosophy” called the Zen of Python (also known as PEP 20). To give it a read, simply open a python interpreter and type:

Python >>>

```
>>> import this
```

and then you’ll see:

> **The Zen of Python, by Tim Peters**
>
> Beautiful is better than ugly.
>
> Explicit is better than implicit.
>
> Simple is better than complex.
>
> Complex is better than complicated.
>
> Flat is better than nested.
>
> Sparse is better than dense.
>
> Readability counts.
>
> Special cases aren’t special enough to break the rules.
>
> Although practicality beats purity.
>
> Errors should never pass silently.
>
> Unless explicitly silenced.
>
> In the face of ambiguity, refuse the temptation to guess.
>
> There should be one– and preferably only one – obvious way to do it.
>
> Although that way may not be obvious at first unless you’re Dutch.
>
> Now is better than never.
>
> Although never is often better than *right* now.
>
> If the implementation is hard to explain, it’s a bad idea.
>
> If the implementation is easy to explain, it may be a good idea.
>
> Namespaces are one honking great idea – let’s do more of those!

These are words to work by. Read Daniel Greenfeld’s notes on Richard Jones’ 19 Pythonic Theses talk for a more detailed explanation.

Acronym explosion

We will also conform our development practices to the following principles:

- While there are many tools we could build to make our lives easier, we are going to confine ourselves to building what is necessary to make our application *work*, remembering that in many cases YAGNI (You Ain’t Gonna Need It).
- That being said, when we come across a pattern of code that we are repeating across the application we will be refactoring our code to keep it DRY.

Help

- Our trigger to cross over from YAGNI to DRY will be the three strikes and you refactor principle. Using a pattern in three places will qualify it for extraction. Two uses will still fall under YAGNI. (In large projects many suggest a modified version of this rule - when you need to reuse it, refactor it.)

A note on the repository structure

Throughout this series you will be able to find the finished code for these exercises in the `flask-tracking` repository on Github. Each part of this tutorial has a branch in the repository and a release. The code for this section is in the part-1 branch. If you choose to check out the repository, you can work with the code for any article in this series by simply running:

Shell

```shell
$ git checkout part-N
```

where `N` is the number of the article the code is for (so for this article use `git checkout part-1`).

Remove ads

Starting the project

With that front-matter out of the way let’s start by creating a new folder to hold our project and activating a virtual environment (if you are unsure how to setup a virtual environment, take a moment and review the guide in Flask’s quickstart):

Shell

```shell
$ mkdir flask-tracking
$ cd flask-tracking
$ virtualenv --no-site-packages venv
$ source venv/bin/activate
```

Dependencies

Like Alice in Wonderland we want to get our dependencies “just right”. Too many dependencies and we will not be able to work on the codebase after leaving it for a month without spending a week reviewing documentation. Too few dependencies and we will spend our time developing everything except our application. To ensure that we keep our dependencies “just right” we will use the following (imperfect) rule - each dependency must solve at least one *difficult* problem well. We will start with dependencies on three libraries - Flask itself for managing the request / response cycle, Flask-WTF for its CSRF protection and data validation and finally Flask-SQLAlchemy for database connection pooling and object / relational mapper. Here is our requirements.txt file:

Shell

```shell
Flask==0.10.1
Flask-SQLAlchemy==1.0
Flask-WTF==0.9.3
```

Now we can run `pip install -r requirements.txt` to install our requirements into our virtual environment.

Outgrowing small

Most Flask applications start out small and then are refactored as the scope of the project grows. For those of us who have not written and then refactored a small Flask application we have a single module version of our tracking application in the `part-0` branch (it weighs in at 145 lines *total*). Feel free to skip to the next section if you have alr
written a single module application in Flask.

For those of you who remain, please `git checkout part-0` or download Part 0 from the releases section of the repository. There should be a single module inside the application, `tracking.py`, and the overal structure should look like this:

```
├── flask-tracking.db
├── requirements.txt
├── templates
│   ├── data_list.html
│   ├── helpers
│   │   ├── forms.html
│   │   └── tables.html
│   ├── index.html
│   ├── layout.html
│   └── validation_error.html
└── tracking.py
```

If you crack it open, the first thing you see after the imports is:

Python

```python
_cwd = dirname(abspath(__file__))

SECRET_KEY = 'flask-session-insecure-secret-key'
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + join(_cwd, 'flask-tracking.db')
SQLALCHEMY_ECHO = True
WTF_CSRF_SECRET_KEY = 'this-is-not-random-but-it-should-be'


app = Flask(__name__)
app.config.from_object(__name__)

db = SQLAlchemy(app)
```

We set up some basic config settings - the `SECRET_KEY` is used to sign Flask's sessions, the `SQLALCHEMY_DATABASE_URI` is the path to our database (we are using SQLite for now), and the `WTF_CSRF_SECRET_KEY` is used to sign WTForms' CSRF tokens. We initialize a new Flask application and tell it to configure itself (with our `app.config.from_object` call) with all of the `ALL_CAPS` symbols in the current module. Then we initialize our Flask-SQLAlchemy extension with our application.

From there, it is pretty much a straight shot - we set up our models:

Python

```python
class Site(db.Model):
    __tablename__ = 'tracking_site'

    id = db.Column(db.Integer, primary_key=True)
    base_url = db.Column(db.String)
    visits = db.relationship('Visit', backref='tracking_site', lazy='select')

    def __repr__(self):
        return '<Site %r>' % (self.base_url)

    def __str__(self):
        return self.base_url


class Visit(db.Model):
    # ... snip ...
```

and forms:

Help

Python

```python
class SiteForm(Form):
    base_url = fields.StringField()


class VisitForm(Form):
    # ... snip ...
```

and routes that utilize these models and forms:

Python

```python
@app.route("/")
def index():
    site_form = SiteForm()
    visit_form = VisitForm()
    return render_template("index.html",
                           site_form=site_form,
                           visit_form=visit_form)


@app.route("/site", methods=("POST", ))
def add_site():
    form = SiteForm()
    if form.validate_on_submit():
        site = Site()
        form.populate_obj(site)
        db.session.add(site)
        db.session.commit()
        flash("Added site")
        return redirect(url_for("index"))
    return render_template("validation_error.html", form=form)
```

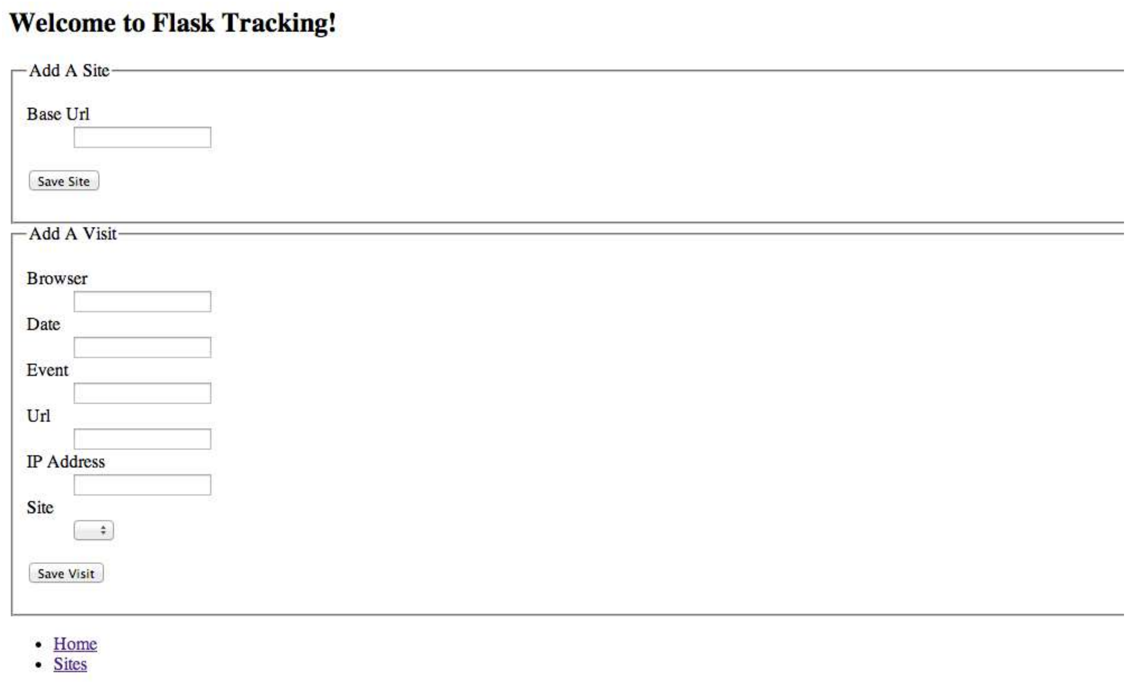
There are also a couple of helper functions to turn SQLAlchemy query objects into lists of lists of data for our templates and then at the bottom of the module we set up a `main` block to create our tables if they do not exist and run the application in debug mode:

Python

```python
if __name__ == "__main__":
    app.debug = True
    db.create_all()
    app.run()
```

If you run `python tracking.py` and then navigate to [localhost:5000]() in your browser you will see the extremely bare-bones application. It is functional (you can create sites and add visits), but it has no users and no access control - everyone sees everything.

Help


Welcome to Flask Tracking!

Add A Site

Base Url

Save Site

Add A Visit

Browser

Date

Event

Url

IP Address

Site

Save Visit

- Home
- Sites

However, as you can see, `tracking.py` is already quite a smörgåsbord of functionality - controllers, models, helpers, and command line setup all jammed into one file. Breaking it up into separate areas of functionality will make it easier to maintain. In addition, re-packaging this application will make it clearer where to add all of the other features that we want (users, access control, etc.). When you are done playing around with the Part 0 code just run `git checkout part-1` and move on to the next section.

A place for everything

Let's start by creating a package for our application to live in (we are going to be using a modified form of the structure described in Flask's documentation for packages and blueprints):

```
flask-tracking/       # Our working root
    flask_tracking/   # The application package
        __init__.py
    requirements.txt  # Meta data needed by our application
    README.md         # and developers
```

Keeping in mind that our goal today is simply to track visits to a site, we will avoid creating a sub-package for our tracking code … yet (remember, YAGNI until you need it). Let's go ahead and add separate `models`, `forms`, and `views` modules to hold our domain models, our data translation layer, and our view code respectively. Let's also create a `templates` subdirectory to hold the resources we will use to render the site. Finally, we will add a configuration file and a script to run the application. Now we should have a directory structure that looks like this:

```
flask-tracking/
    flask_tracking/
        templates/    # Holds Jinja templates
        __init__.py   # General application setup
        forms.py      # User data to domain data mappers and validators
        models.py     # Domain models
        views.py      # well ... controllers, really.
```

Help

```
    config.py         # Configuration, just like it says on the cover
    README.md
    requirements.txt
    run.py            # `python run.py` to bring the application up locally.
```

Domain models

Our domain models are simple - a site has a root URL and a visit has metadata about the visitor (browser, IP address, URL, etc.). We include __repr__ methods to give us better details on the command line (`<Visit for site ID 1: / - 2013-11-09 14:10:11>` is more useful than `<package.module.Visit object at 0x12345>`). `Site` includes a __str__ method to control what we display when we display a list of sites in a drop down menu.

Python

```python
from flask.ext.sqlalchemy import SQLAlchemy

db = SQLAlchemy()


class Site(db.Model):
    __tablename__ = 'tracking_site'

    id = db.Column(db.Integer, primary_key=True)
    base_url = db.Column(db.String)
    visits = db.relationship('Visit', backref='tracking_site', lazy='select')

    def __repr__(self):
        return '<Site {:d} {}>'.format(self.id, self.base_url)

    def __str__(self):
        return self.base_url


class Visit(db.Model):
    __tablename__ = 'tracking_visit'

    id = db.Column(db.Integer, primary_key=True)
    browser = db.Column(db.String)
    date = db.Column(db.DateTime)
    event = db.Column(db.String)
    url = db.Column(db.String)
    ip_address = db.Column(db.String)
    site_id = db.Column(db.Integer, db.ForeignKey('tracking_site.id'))

    def __repr__(self):
        r = '<Visit for site ID {:d}: {} - {:%Y-%m-%d %H:%M:%S}>'
        return r.format(self.site_id, self.url, self.date)
```

Note that we have not initialized our `flask.ext.sqlalchemy` object with any application so these models are not bound to this particular application (this flexibility comes with some small costs, which we will encounter shortly).

Data translation layer

Our `forms` code is standard WTForms, with one exception. If you look at `VisitForm`:

Python

```python
from .models import Site

# ... snip ...

class VisitForm(Form):
    browser = fields.StringField()
    date = fields.DateField()
    event = fields.StringField()
    url = fields.StringField()
    ip_address = fields.StringField("IP Address")
    ip_address = fields.StringField("IP Address")
    site = QuerySelectField(query_factory=lambda: Site.query.all())
```

you will note that we need to wrap `Site.query.all` in a `lambda` rather than passing it in as-is. Since our `db` is not bound to an application when `VisitForm` is being constructed we cannot access `Site.query`. Creating a function that will call `Site.query` only when `VisitForm` is instantiated (e. g. in our views we call `form = VisitForm()`) ensures that we will only access `Site.query` when we will have access to a Flask application instance.

Views

Our views are the most complex part of our application (witness how many dependencies we are importing):

Python

```python
from flask import Blueprint, flash, Markup, redirect, render_template, url_for

from .forms import SiteForm, VisitForm
from .models import db, query_to_list, Site, Visit
```

We start out by creating a blueprint (we could also `from flask_tracking import app` and use the app, but I prefer to switch to blueprints once my application has grown beyond a single file)

Python

```python
tracking = Blueprint("tracking", __name__)
```

We then map our views to routes using the normal decorator syntax - I have added comments around some of the functionality where what we are doing may not be perfectly clear (or where we are repeating ourselves and we will want to refactor later):

Python

```python
@tracking.route("/")
def index():
    site_form = SiteForm()
    visit_form = VisitForm()
    return render_template("index.html",
                           site_form=site_form,
                           visit_form=visit_form)


@tracking.route("/site", methods=("POST", ))
def add_site():
    # The create a form, validate the form,
    # map the form to a model, save the model,
    # and redirect pattern will be pretty common
    # throughout the application. This is an area
    # that is ripe for improvement and refactoring.
    form = SiteForm()
    if form.validate_on_submit():
        site = Site()
        form.populate_obj(site)
        db.session.add(site)
        db.session.commit()
        flash("Added site")
        return redirect(url_for(".index"))

    return render_template("validation_error.html", form=form)


@tracking.route("/site/<int:site_id>")
def view_site_visits(site_id=None):
    site = Site.query.get_or_404(site_id)
    query = Visit.query.filter(Visit.site_id == site_id)
    data = query_to_list(query)
    title = "visits for {}".format(site.base_url)
    return render_template("data_list.html", data=data, title=title)


@tracking.route("/visit", methods=("POST", ))
```

```
@tracking.route("/site/<int:site_id>/visit", methods=("POST",))
def add_visit(site_id=None):
    if site_id is None:
        # This is only used by the visit_form on the index page.
        form = VisitForm()
    else:
        site = Site.query.get_or_404(site_id)
        # WTForms does not coerce obj or keyword arguments
        # (otherwise, we could just pass in `site=site_id`)
        # CSRF is disabled in this case because we will *want*
        # users to be able to hit the /site/:id endpoint from other sites.
        form = VisitForm(csrf_enabled=False, site=site)

    if form.validate_on_submit():
        visit = Visit()
        form.populate_obj(visit)
        visit.site_id = form.site.data.id
        db.session.add(visit)
        db.session.commit()
        flash("Added visit for site {}".format(form.site.data.base_url))
        return redirect(url_for(".index"))

    return render_template("validation_error.html", form=form)


@tracking.route("/sites")
def view_sites():
    query = Site.query.filter(Site.id >= 0)
    data = query_to_list(query)

    # The header row should not be linked
    results = [next(data)]
    for row in data:
        row = [_make_link(cell) if i == 0 else cell
               for i, cell in enumerate(row)]
        results.append(row)

    return render_template("data_list.html", data=results, title="Sites")


_LINK = Markup('<a href="{url}">{name}</a>')


def _make_link(site_id):
    url = url_for(".view_site_visits", site_id=site_id)
    return _LINK.format(url=url, name=site_id)
```

This gives us an application that will let us add a site or a visit on the home page, view a list of sites, and view each site's visits. We have not actually registered the blueprint with an application yet, so …

Application setup

… onwards to `__init__.py`:

Python

```
from flask import Flask

from .models import db
from .views import tracking

app = Flask(__name__)
app.config.from_object('config')
```

```
# Add the `constants` variable to all Jinja templates.
@app.context_processor
def provide_constants():
    return {"constants": {"TUTORIAL_PART": 1}}

db.init_app(app)

app.register_blueprint(tracking)
```

We create an application, configure it, register our Flask-SQLAlchemy instance on our application, and finally register our blueprint. Now Flask knows how to handle our routes.

Configuration and command line runner

One level up our configuration enables Flask's sessions and sets up our SQLite database:

Python

```python
# config.py
from os.path import abspath, dirname, join

_cwd = dirname(abspath(__file__))

SECRET_KEY = 'flask-session-insecure-secret-key'
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + join(_cwd, 'flask-tracking.db')
SQLALCHEMY_ECHO = True
```

And our application runner creates the database tables if they do not exist and runs the application in debug mode:

Python

```python
# run.py
#!/usr/bin/env python
from flask_tracking import app, db

if __name__ == "__main__":
    app.debug = True
    # Because we did not initialize Flask-SQLAlchemy with an application
    # it will use `current_app` instead.  Since we are not in an application
    # context right now, we will instead pass in the configured application
    # into our `create_all` call.
    db.create_all(app=app)
    app.run()
```

See Flask-SQLAlchemy's *Introduction to Contexts* if you want to get a better sense of why we need to pass our `app` into the `db.create_all` call.

And everything in its place

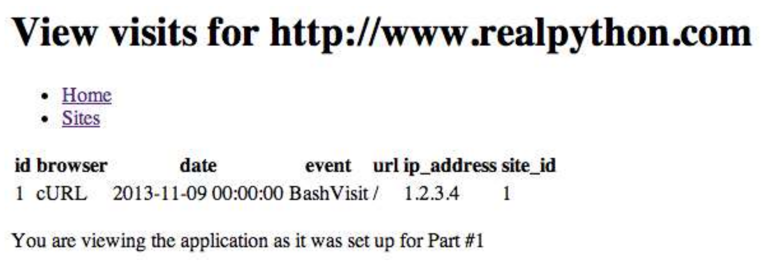
Now we should be able to run `python run.py` and see our application start up. Go to localhost:5000 and create a site to test things out. Then, to verify that others can add visits to the site, try running:

Shell

```shell
$ curl --data 'event=BashVisit&browser=cURL&url=/&ip_address=1.2.3.4&date=2013-11-09' localhost:5000/site/1/v
```

from the command line. When you go back to the application, click on "Sites" and then click on the "1" for your site. A single visit should be displayed on the page:

Help



View visits for http://www.realpython.com

- Home
- Sites

id	browser	date	event	url	ip_address	site_id
1	cURL	2013-11-09 00:00:00	BashVisit	/	1.2.3.4	1

You are viewing the application as it was set up for Part #1

Wrapping Up

That's it for this post. We now have a working application where sites can be added and visits recorded against them. We still need to add user accounts, an easy to use client-side API for tracking, and reports.

1. In Part II we will add users, access control, and enable users to add visits from their own websites. We will explore more best practices for writing templates, keeping our models and forms in sync, and handling static files.
2. In Part III we'll explore writing tests for our application, logging, and debugging errors.
3. In Part IV we'll do some Test Driven Development to enable our application to accept payments and display simple reports.
4. In Part V we will write a RESTful JSON API for others to consume.
5. In Part VI we will cover automating deployments (on Heroku) with Fabric and basic A/B Feature Testing.
6. Finally, in Part VII we will cover preserving your application for the future with documentation, code coverage and quality metric tools.

Thanks for reading and tune in next time!

Mark as Completed

About The Team

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The tear [illegible] rs who worked on this tutorial are:

Help