Saratoga Documentation

Release 0.6.0

HawkOwl

September 14, 2014

Contents

Saratoga is a framework for easily creating APIs. It uses Python and Twisted, and supports both CPython 2.7 and PyPy.

Saratoga revolves around versions – it is designed so that you can write code for new versions of your API without disturbing old ones. You simply expand the scope of the unchanged methods, and copy a reference into your new version. It also has helpers for I/O validation and authentication.

You can get the MIT-licensed code on GitHub, or download it from PyPI, with:

```
$ pip install saratoga
```

CHAPTER 1

Introduction

A look at Saratoga, starting from the ground up.

1.1 Introduction – Getting Off The Ground

Saratoga was made to help you create simple APIs that divide cleanly over versions, with minimal fuss – Saratoga takes care of routing, assembly, parameter checking and authentication for you. All you have to do is provide your business logic.

1.1.1 Installing

To install, use pip:

```
$ pip install saratoga
```

This'll install Saratoga and its base dependencies. You can also install the `httpsig` package to enable HMAC authentication, which will be covered later.

1.1.2 A Very Simple Example

In this introduction, we will create a *Planet Information* API. We will create something that will allow us to query it, and will return some information about planets. So, first, let's define our API.

Simple API Definition

Put the following in `planets.json`:

```
{
    "metadata": {
        "name": "planetinfo",
        "friendlyName": "Planet Information",
        "versions": [1]
    },
    "endpoints": [
        {
            "name": "yearlength",
            "friendlyName": "Year Length of Planets",
            "endpoint": "yearlength",
```

```
            "getProcessors": [
                {
                    "versions": [1]
                }
            ]
        }
    ]
}
```

Now, we have made a `metadata` section that gives three things:

- The 'name' of our API.
- The 'friendly name' (human-readable) name of our API.
- A list of versions that our API has (right now, just "1").

We then define an `endpoints` section, which is a list of our different APIs. We have defined only one here, and we have said that:

- It has a name of 'yearlength'.
- It has a human-readable name of 'Year Length of Planets'.
- It has an endpoint of 'yearlength'. Saratoga structures APIs as `/<VERSION>/<ENDPOINT>`, so this means that a v1 of it will be at `/v1/yearlength`.

There is also then `getProcessors` – a list of *processors*. A processor in Saratoga is the code that actually does the heavy lifting. This one here only has one item in it, a list of versions that this processor applies to (in this case, just `1`).

Using this API description, we can figure out that our future API will be at `/v1/yearlength`.

Now, lets make the processor behind it.

Simple Python Implementation

Put the following in `planets.py`:

```
import json
from saratoga.api import SaratogaAPI

class PlanetAPI(object):
    class v1(object):
        def yearlength_GET(self, request, params):
            pass

APIDescription = json.load(open("planets.json"))
myAPI = SaratogaAPI(PlanetAPI, APIDescription)
myAPI.run(port=8094)
```

This example can be this brief because Saratoga takes care of nearly everything else.

So, let's break it down.

1. First we import `SaratogaAPI` from `saratoga.api` – this is what takes care of creating your API from the config.
2. We then create a `PlanetAPI` class, and make a subclass called `v1`. This corresponds to version 1 of your API.
3. We then create a method called `yearlength_GET`. This is done in a form of `<NAME>_<METHOD>`. It has three parameters - `self` (this is special, we'll get to it later), `request` (the Twisted Web Request for the API call) and `params` (rather than have to parse them yourself, Saratoga does this for you).

Currently, `yearlength_GET` does nothing, so lets fill in some basic functionality – for brevity, we'll only support Earth and Pluto.

```
def yearlength_GET(self, request, params):
    planetName = params["params"]["name"].lower()
    if planetName == "earth":
        return {"seconds": 31536000}
    elif planetName == "pluto":
        return {"seconds": 7816176000}
```

As you can see, we access `params`, which is a dict of all the things given to you in the API call. This is sorted out by Saratoga, according to your API description – it makes sure that all required parameters are there, and throws an error if it is not.

We then return a `dict` with our result. Saratoga will automatically serialise it to JSON for consumption, although you can use `different output formats` if you want a different format.

Running

Let's try and run it!

```
$ python planets.py
```

Now, go to `http://localhost:8094/v1/yearlength?name=earth` in your web browser. You should get the following back:

```
{
    "data": {
        "seconds": 31536000
    },
    "status": "success"
}
```

1.1.3 Going Further

The next article is about adding global state to your Saratoga API.

1.2 Introduction - Adding Global State with Service Classes

Most APIs are only useful with some form of global state - say, a database, or in-memory record of values. Saratoga does this through **Service Classes** – a separate class where you put all of your global state, without affecting any of your API implementation.

1.2.1 A Basic Service Class

Using the same `planets.json` as in the Introduction, modify `planets.py` to look like this:

```
import json
from saratoga.api import SaratogaAPI, DefaultServiceClass

class PlanetServiceClass(DefaultServiceClass):
    def __init__(self):
        self.yearLength = {
```

```
            "earth": {"seconds": 31536000},
            "pluto": {"seconds": 7816176000}
        }

class PlanetAPI(object):
    class v1(object):
        def yearlength_GET(self, request, params):
            planetName = params["params"]["name"].lower()
            return self.yearLength.get(planetName)

APIDescription = json.load(open("planets.json"))
myAPI = SaratogaAPI(PlanetAPI, APIDescription, serviceClass=PlanetServiceClass())
myAPI.run(port=8094)
```

So, let's have a look at what's different here - we now have a service class.

- We import `DefaultServiceClass` from `saratoga.api` and subclass it, adding things into it.
- We then pass in an instance (not a reference to the class, an instance) of our custom service class.
- In `yearlength_GET`, we then access the `yearLength` dict defined in `PlanetServiceClass`, using self.

The `self` of all of your processors is automatically set to your service class. This means that you can do global state fairly easily, with the caveat that every version of your API accesses the same service class. Saratoga won't be able to help you too much with global state differences, such as database schemas or the like, but you can isolate business logic changes in different API versions.

Accessing `http://localhost:8094/v1/yearlength?name=earth` in your web browser again will get the following back:

```
{
    "data": {
        "seconds": 31536000
    },
    "status": "success"
}
```

1.2.2 Going Further

Next, we'll have a look at Saratoga's parameter checking.

1.3 Introduction – Authentication

Note: This hasn't been adapted from Haddock yet, so is wrong in places (pretty much everywhere).

Some APIs may need authentication before accessing - for example, if you are writing a service rather than just a public data API. Haddock allows you to either do the authentication yourself, or hook in a "Shared Secret Source" which will request a user's shared secret from your backend.

1.3.1 Using a Shared Secret Source

For this example, we will be using a new API Description.

API Description

Put this in `authapi.json`:

```
{
    "metadata": {
        "name": "authapi",
        "friendlyName": "An Authenticated API",
        "versions": [1],
        "apiInfo": true
    },
    "api": [
        {
            "name": "supersecretdata",
            "friendlyName": "Super secret data endpoint!!!!",
            "endpoint": "supersecretdata",
            "requiresAuthentication": true,
            "getProcessors": [
                {
                    "versions": [1]
                }
            ]
        }
    ]
}
```

The new part of this is `requiresAuthentication` in our single API, which is now set to `true`.

Python Implementation

Put this into `authapi.py`:

```
import json
from haddock.api import API, DefaultServiceClass
from haddock import auth

class AuthAPIServiceClass(DefaultServiceClass):
    def __init__(self):
        users = [{
            "username": "squirrel",
            "canonicalUsername": "secretsquirrel@mi6.gov.uk",
            "password": "secret"
        }]
        self.auth = auth.DefaultHaddockAuthenticator(
            auth.InMemoryStringSharedSecretSource(users))

class AuthAPI(object):
    class v1(object):
        def supersecretdata_GET(self, request, params):
            return "Logged in as %s" % (params.get("haddockAuth"),)

APIDescription = json.load(open("authapi.json"))
myAPI = API(AuthAPI, APIDescription, serviceClass=AuthAPIServiceClass())
myAPI.getApp().run("127.0.0.1", 8094)
```

In our implementation, we now import `haddock.auth`, and use two portions of it when creating our service class. We set `self.auth` to be a new instance of `auth.DefaultHaddockAuthenticator`, with a `auth.InMemoryStringSharedSecretSource` as its only argument, with that taking a list of users.

How Authentication in Haddock Works

Before your API method is called, Haddock checks the API description, looking for `requiresAuthentication` on the endpoint. If it's found, then it will look in the HTTP `Authorized` header for `Basic`. It will then call `auth_usernameAndPassword` on the Haddock authenticator, which will then check it and decide whether or not to allow the request.

Since this is boilerplate, Haddock abstracts it into the `DefaultHaddockAuthenticator`, which takes a `SharedSecretSource`. Currently, the source requires only one function - `getUserDetails`. This is called, asking for the details of a user, which the authenticator will then check against the request. If it is successful, the authenticator will return either the user's *canonical username* or their username.

Canonical usernames are returned by the Haddock authenticator when possible, which are then placed in a `haddockAuth` param. Your API method will get this, and know that this is the user which has been successfully authenticated.

The `InMemoryStringSharedSecretSource` Source

The `InMemoryStringSharedSecretSource` takes a list of users, which consists of a `username`, `password` and optionally a `canonicalUsername`.

Running It

Now, since we have got our authentication-capable API, let's test it. Try running `curl http://localhost:8094/v1/supersecretdata`, you should get this back:

```
{"status": "fail", "data": "Authentication required."}
```

Haddock is now checking for authentication. Let's try giving it a username and password, with `curl http://localhost:8094/v1/supersecretdata -u squirrel:secret`:

```
{"status": "success", "data": "Logged in as secretsquirrel@mi6.gov.uk"}
```

As you can see, we returned the canonical username in `supersecretdata_GET`, which is `secretsquirrel@mi6.gov.uk`.

1.3.2 Why Canonical Usernames?

Since this is an API, it may have sensitive data behind it, which you want to control access to. Controlling it via authentication is only solving part of the problem - you need to make sure that if the shared secret is lost, you can rescind access to it. Since changing passwords is a pain for users, a better solution is to have *API specific credentials*, and Haddock's authentication is made to support that.

When giving out access to an API, you should create a set of API specific credentials - that is, a randomly generated username and password which is then used against your API, and can be revoked if required. Simply store the random creds, and a link to the user's real (canonical) username, and give that to the authenticator.

1.3.3 Implementing Your Own Shared Secret Source

This is taken from Tomato Salad, a project using Haddock.

```
class tsSharedSecretSource(object):
    def __init__(self, db):
        self.db = db

    def getUserDetails(self, username):
        def _continue(result):
            if result:
                res = {}
                res["username"] = result["APIKeyUsername"]
                res["canonicalUsername"] = result["userEmail"]
                res["password"] = result["APIKeyPassword"]
                return res
            raise AuthenticationFailed("Incorrect API key.")

        d = self.db.fetchAPIKey(username)
        d.addCallback(_continue)
        return d

class tsServiceClass(DefaultServiceClass):
    def __init__(self):
        self.db = Database(
            {"connectionString": "sqlite:///tomatosalad.db"})
        self.auth = DefaultHaddockAuthenticator(
            tsSharedSecretSource(self.db))
```

CHAPTER 2

Specifications

This specifies the Saratoga API definition format.

2.1 Saratoga API Description

The Saratoga API Description is a standard structure that Haddock uses to build your API. It contains information about your project (`metadata`), your API endpoints (`endpoints`), and the processors behind those APIs (`<METHOD>Processors`).

The API Description ends up having two top-level parts - the `metadata` and the `endpoint`. They are laid out like this:

```
{
    "metadata": {
        ...
    },
    "endpoints": {
        ...
    }
}
```

2.1.1 Metadata

The `metadata` contains three things:

- `name`: The computer-friendly name.
- `friendlyName`: The user-friendly name.
- `versions`: A list of applicable versions. They don't have to be 1, 2, or whatever – they're just used later on in `api`.

2.1.2 Endpoints

The `endpoints` section contains a list of dicts, which are API endpoints. In each API method there is:

- `name`: The computer-friendly name. This is used in naming your functions later!
- `friendlyName`: The user-friendly name.
- `description`: The user-friendly description.

- `endpoint`: The URL endpoint. For example, it will make a processor for v1 be under `/v1/weather`. Alternatively, it can be a regular expression.
- `func`: The name that refers to the processor method. Required if `endpoint` is a regex.
- `requiresAuthentication` (optional): A boolean that defines whether this API needs authentication. Default is false.
- `getProcessors` (optional): A list of processors (see below). These processors respond to a HTTP GET.
- `postProcessors` (optional): A list of processors (see below). These processors respond to a HTTP POST.
- `putProcessors` (optional): A list of processors (see below). These processors respond to a HTTP PUT.
- `deleteProcessors` (optional): A list of processors (see below). These processors respond to a HTTP DELETE.
- `patchProcessors` (optional): A list of processors (see below). These processors respond to a HTTP PATCH.

2.1.3 Processors

Processors are the bits of your API that do things. They are made up of dicts, and contain the following fields:

- `versions`: A list of versions (see `metadata`) which this endpoint applies to.
- `paramsType` (optional): Where the params will be - either `url` (in `request.args`) or `jsonbody` (for example, the body of a HTTP POST). Defaults to `url`.

2.1.4 Example

For a proper example, see the top of `saratoga/test/test_api.py`. It has nearly every option defined in there somewhere.