Clastic

Release 20.0

May 14, 2021

Contents:

Clastic is a Pythonic microframework for building web applications featuring:

- Fast, coherent routing system
- Powerful middleware architecture
- Built-in observability features via the `meta` Application
- Extensible support for multiple templating systems
- Werkzeug-based WSGI/HTTP primitives, same as Flask

CHAPTER 1

Installation

Clastic is pure Python, and tested on Python 2.7-3.7+, as well as PyPy. Installation is easy:

```
pip install clastic
```

Then get to building your first application!

```
from clastic import Application, render_basic

app = Application(routes=[('/', lambda: 'hello world!', render_basic)])

app.serve()
# Visit localhost:5000 in your browser to see the result!
```

CHAPTER 2

Getting Started

Check out our *Tutorial* for more.

2.1 Tutorial: Time zone convertor

Note: This document starts out with a fairly simple application code and proceeds by building on it. Therefore, it would be helpful to the reader to code along and try out the various stages of the application. In this manner, completing it should take about an hour.

While Clastic supports building all sorts of web applications and services, our first project will be a traditional HTML-driven web application. It will convert a given time (and date) between two time zones. The user will enter a date and a time, and select two time zones from a list of all available time zones, one for the source location and one for the destination location. A screenshot of the final application is shown below.

Before we start, a note about time zones: these are represented in "region/location" format, as in "Australia/Tasmania". While most such codes have two components, some contain only one (like "UTC"), and some contain more than two (like "America/North_Dakota/New_Salem"). Also note that spaces in region and location names are replaced with underscores. Refer to the "List of tz database time zones" for a full list.

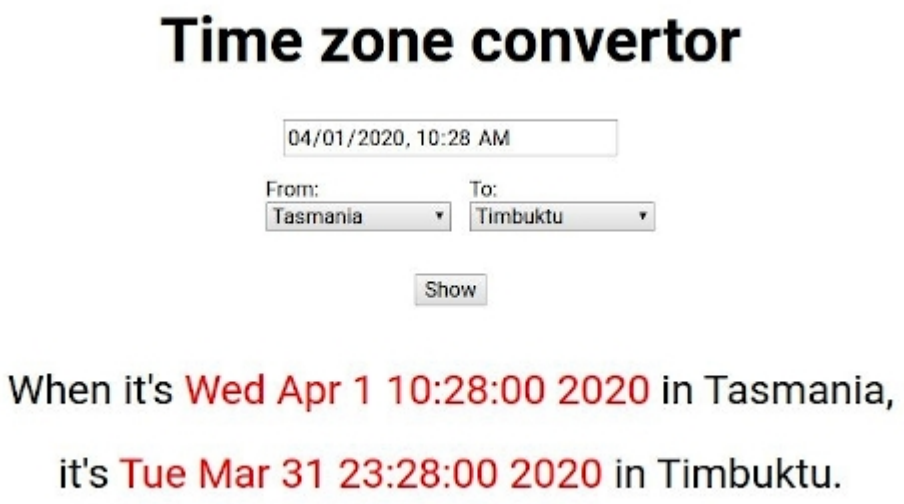

Fig. 1: After selecting the time and two time zones, clicking the "Show" button will display the given time in the source location and the corresponding time in the destination location.

2.1.1 Prerequisites

It's common practice to work in a separate virtual environment for each project, so we suggest that you create one for this tutorial. Read the "Virtual Environments and Packages" section of the official Python documentation for more information.

Clastic works with any version of Python. Let's start by installing it:

```
pip install clastic
```

The example application also makes use of the dateutil package. Note that the PyPI name for that package is *python-dateutil*:

```
pip install python-dateutil
```

2.1.2 Getting started

Let's start with an application that just displays the form, but doesn't handle the submitted data. It consists of a Python source file (`tzconvert.py`) and an HTML template file (`home.html`), both in the same folder.

Here's the Python file:

```
import os
from datetime import datetime

from clastic import Application
from clastic.render import AshesRenderFactory
from dateutil import zoneinfo


CUR_PATH = os.path.dirname(os.path.abspath(__file__))
```

(continues on next page)

(continued from previous page)

```
def get_location(zone):
    return zone.split("/")[-1].replace("_", " ")


def get_all_time_zones():
    zone_info = zoneinfo.get_zonefile_instance()
    zone_names = zone_info.zones.keys()
    entries = {get_location(zone): zone for zone in zone_names}
    return [
        {"location": location, "zone": entries[location]}
        for location in sorted(entries.keys())
    ]


ALL_TIME_ZONES = get_all_time_zones()


def home():
    render_ctx = {
        "zones": ALL_TIME_ZONES,
        "default_src": "UTC",
        "default_dst": "UTC",
        "now": datetime.utcnow().strftime("%Y-%m-%dT%H:%M"),
    }
    return render_ctx


def create_app():
    routes = [("/", home, "home.html")]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)


app = create_app()

if __name__ == "__main__":
    app.serve()
```

Let's go through this code piece by piece, starting at the bottom and working our way up.

In the last few lines, we create the application and start it by invoking its *`serve()`* method:

```
app = create_app()

if __name__ == "__main__":
    app.serve()
```

Application creation is handled by the `create_app()` function, where we register the routes of the application. Every *`Route`* associates a path with a function (*endpoint*) that will process the requests to that path. In the example, there is only one route where the path is `/` and the endpoint function is `home`:

```
def create_app():
    routes = [("/", home, "home.html")]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)
```

The route also sets the template file `home.html` to render the response. Clastic supports multiple template engines; in this application we use Ashes. We create a render factory for rendering templates for our chosen template engine (in this case an `AshesRenderFactory`) and tell it where to find the template files. Here, we tell the render factory to look for templates in the same folder as this Python source file. The *`Application`* is then created by giving the sequence of routes and the render factory.

The `home()` function generates the data that the template needs (the "render context"). In the template, there are two dropdown lists for all available time zones, so we have to pass that list. Here, we store this data in the `ALL_TIME_ZONES` variable, which we have constructed using the `get_all_time_zones()` function, as a list of dictionaries containing the location names and the full time zone code. The location name is the last component of the time zone code, extracted using the `get_location()` function. The location name will be displayed to the user, whereas the full code will be transmitted as the data. The entries will be sorted by location name. We also pass default values for the form inputs: "UTC" for both the source and destination time zones, and the current UTC time for the date-time to be converted:

```
def home():
    render_ctx = {
        "zones": ALL_TIME_ZONES,
        "default_src": "UTC",
        "default_dst": "UTC",
        "now": datetime.utcnow().strftime("%Y-%m-%dT%H:%M"),
    }
    return render_ctx
```

The `home.html` template is given below. In the selection options, for each element in the render context's `zones` list, the `location` key is used for display and the `zone` key is used for the value:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Time zone convertor</title>
</head>
<body>
  <h1>Time zone convertor</h1>
  <form action="/show" method="POST">
    <input type="datetime-local" name="dt" value="{now}" required>

    <div class="timezones">
      <div class="timezone">
        <label for="src">From:</label>
        <select name="src" id="src">
          {#zones}
          {@eq key=location value="{default_src}"}
          <option value="{zone}" selected>{location}</option>
          {:else}
          <option value="{zone}">{location}</option>
          {/eq}
          {/zones}
        </select>
      </div>

      <div class="timezone">
        <label for="dst">To:</label>
        <select name="dst" id="dst">
          {#zones}
          {@eq key=location value="{default_dst}"}
```

(continues on next page)

(continued from previous page)

```
          <option value="{zone}" selected>{location}</option>
          {:else}
          <option value="{zone}">{location}</option>
          {/eq}
          {/zones}
        </select>
      </div>
    </div>

    <button type="submit">Show</button>
  </form>
</body>
</html>
```

With these two files in place, run the command `python tzconvert.py` and you can visit the address `http://localhost:5000/` to see the form.

2.1.3 Handling request data

At first, our application will not display the converted time on the same page. Instead, it submits the form data to another page (the `/show` path), therefore we need an endpoint function to handle these requests. First, let's add the corresponding route:

```
def create_app():
    routes = [
        ("/", home, "home.html"),
        ("/show", show_time, "show_time.html"),
    ]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)
```

Next, we'll implement the endpoint function `show_time()`. Since this function has to access the submitted data, it takes the *request* as parameter, and the data in the request is available through `request.values`. After calculating the converted time, the function passes the source and destination times to the template, along with the location names. Source and destination times consist of dictionary items indicating how to display them (`text`), and what data to submit (`value`).

```
from dateutil import parser, tz


def show_time(request):
    dt = request.values.get("dt")
    dt_naive = parser.parse(dt)

    src = request.values.get("src")
    src_zone = tz.gettz(src)

    dst = request.values.get("dst")
    dst_zone = tz.gettz(dst)

    dst_dt = convert_tz(dt_naive, src_zone, dst_zone)
    render_ctx = {
        "src_dt": {
            "text": dt_naive.ctime(),
            "value": dt
```

(continues on next page)

(continued from previous page)

```
        },
        "dst_dt": {
            "text": dst_dt.ctime(),
            "value": dst_dt.strftime('%Y-%m-%dT%H:%M')
        },
        "src_location": get_location(src),
        "dst_location": get_location(dst),
    }
    return render_ctx
```

The only missing piece is the `convert_tz()` function that will actually do the conversion:

```
def convert_tz(dt_naive, src_zone, dst_zone):
    src_dt = dt_naive.replace(tzinfo=src_zone)
    dst_dt = src_dt.astimezone(dst_zone)
    return dst_dt
```

And below is a simple `show_time.html` template. Note how the `text` and `value` subitems are used:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Time zone convertor</title>
</head>
<body>
  <h1>Time zone convertor</h1>
  <p class="info">
    When it's <time datetime="{src_dt.value}">{src_dt.text}</time>
    in {src_location},<br>
    it's <time datetime="{dst_dt.value}">{dst_dt.text}</time>
    in {dst_location}.
  </p>
  <p>Go to the <a href="/">home page</a>.</p>
</body>
</html>
```

2.1.4 Static assets

As our next step, let us apply some style to our markup. We create a subfolder named `static` in the same folder as our Python source file and put a file named `custom.css` into that folder. Below is the example content for the file:

```
body {
  font-family: 'Roboto', 'Helvetica', 'Arial', sans-serif;
}

h1 {
  font-size: 3em;
}

p, h1 {
  text-align: center;
}

form {
```

(continues on next page)

(continued from previous page)

```
  display: flex;
  flex-direction: column;
  align-items: center;
}

input, select, button {
  font: inherit;
}

label {
  display: block;
}

div.timezones {
  display: flex;
  justify-content: space-between;
  margin: 1rem 0;
}

div.timezone {
  width: 45%;
}

p.info {
  font-size: 2em;
  line-height: 2;
}

time {
  color: #ff0000;
}
```

The changes to the application code will be quite small. First, we define the file system path to the folder that contains the static assets:

```
CUR_PATH = os.path.dirname(os.path.abspath(__file__))
STATIC_PATH = os.path.join(CUR_PATH, "static")
```

And then we add a route by creating a `StaticApplication` with the static file system path we have defined, and we set it as the endpoint that will handle the requests to any application path under `/static`:

```
from clastic.static import StaticApplication


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        ("/show", show_time, "show_time.html"),
        ("/static", static_app),
    ]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)
```

Don't forget to add the stylesheet link to the templates:

```
<head>
  <meta charset="utf-8">
  <title>Time zone convertor</title>
  <link rel="stylesheet" href="/static/custom.css">
</head>
```

2.1.5 Working with JSON

Our last task is to display the converted time in the same page as the form instead of moving to a second page. In order to achieve this, we're going to implement a basic JSON API endpoint to update the page with data sent to and received from the application.

Actually, we can use our `show_time()` function for this purpose, with minimal changes. Instead of accessing the submitted data through `request.values`, we just load it from `request.data`. No changes are needed regarding the returned value.

```
import json


def show_time(request):
    values = json.loads(request.data)

    dt = values.get("dt")
    dt_naive = parser.parse(dt)

    src = values.get("src")
    src_zone = tz.gettz(src)

    dst = values.get("dst")
    dst_zone = tz.gettz(dst)

    dst_dt = convert_tz(dt_naive, src_zone, dst_zone)
    render_ctx = {
        "src_dt": {
            "text": dt_naive.ctime(),
            "value": dt
        },
        "dst_dt": {
            "text": dst_dt.ctime(),
            "value": dst_dt.strftime('%Y-%m-%dT%H:%M')
        },
        "src_location": get_location(src),
        "dst_location": get_location(dst),
    }
    return render_ctx
```

The next thing is to set the renderer to `render_json()` for this route:

```
from clastic import render_json


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        ("/show", show_time, render_json),
```

(continues on next page)

(continued from previous page)

```
        ("/static", static_app),
    ]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)
```

At this point, you should be able to test this route using curl:

```
curl -X POST -H "Content-Type: application/json" \
  -d '{"dt": "2020-04-01T10:28", "src": "Australia/Tasmania", "dst": "Africa/Timbuktu
↪"}' \
  http://localhost:5000/show
```

And the home page template becomes:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Time zone convertor</title>
  <link rel="stylesheet" href="/static/custom.css">
  <script>
    async function showResult(event, form) {
      event.preventDefault();
      let formData = new FormData(form);
      let response = await fetch('/show', {
        method: 'POST',
        body: JSON.stringify(Object.fromEntries(formData))
      });
      let json = await response.json();
      document.getElementById('src_dt').innerHTML = json['src_dt']['text'];
      document.getElementById('src_dt').setAttribute('datetime', json['src_dt']['value
↪']);
      document.getElementById('src_location').innerHTML = json['src_location'];
      document.getElementById('dst_dt').innerHTML = json['dst_dt']['text'];
      document.getElementById('dst_dt').setAttribute('datetime', json['dst_dt']['value
↪']);
      document.getElementById('dst_location').innerHTML = json['dst_location'];
      document.querySelector('.info').style.display = 'block';
    }
  </script>
</head>
<body>
  <h1>Time zone convertor</h1>
  <form action="." method="POST" onsubmit="showResult(event, this)">
    <input type="datetime-local" name="dt" value="{now}" required>

    <div class="timezones">
      <div class="timezone">
        <label for="src">From:</label>
        <select name="src" id="src">
          {#zones}
          {@eq key=location value="{default_src}"}
          <option value="{zone}" selected>{location}</option>
          {:else}
          <option value="{zone}">{location}</option>
          {/eq}
          {/zones}
```

(continues on next page)

(continued from previous page)

```
        </select>
      </div>

      <div class="timezone">
        <label for="dst">To:</label>
        <select name="dst" id="dst">
          {#zones}
          {@eq key=location value="{default_dst}"}
          <option value="{zone}" selected>{location}</option>
          {:else}
          <option value="{zone}">{location}</option>
          {/eq}
          {/zones}
        </select>
      </div>
    </div>

    <button type="submit">Show</button>
  </form>

  <p class="info">
    When it's <time id="src_dt" datetime="2020-01-01T18:00">Jan 1 2020</time>
    in <span id="src_location">UTC</span>,<br>
    it's <time id="dst_dt" datetime="2020-01-01T18:00">Jan 1 2020</time>
    in <span id="dst_location">UTC</span>.
  </p>
</body>
</html>
```

The changes are:

- The template for showing the result has been merged. It contains dummy information.
- The JavaScript code for updating the page is added. It gets called when the form gets submitted (when the button is clicked).

One last thing to do is to hide the result markup before the user clicks the "Show" button. This can be easily achieved in CSS:

```
p.info {
  display: none;
}
```

This concludes the introductory tutorial. The full application code can be found in the repo. Check out the *second part* to learn more about Clastic's features.

2.2 Tutorial: Link Shortener

Note: This document continues from where the *first part* left off. As in the first part, we proceed by developing an example application step by step. We suggest that you code along and try out the various stages of the application. In this manner, completing it should take about an hour.

The first part of the tutorial covered some basic topics like routing, form handling, static assets, and JSON endpoints. This second part will show examples for resource handling, redirection, errors, and middleware usage.

The example application will be a link shortener. There will be an option for letting shortened links expire, based on time or on the number of clicks. Users can select the shortened names (aliases) themselves, or let the application generate one. Expired aliases will not be reusable.

A screenshot of the application is shown below:

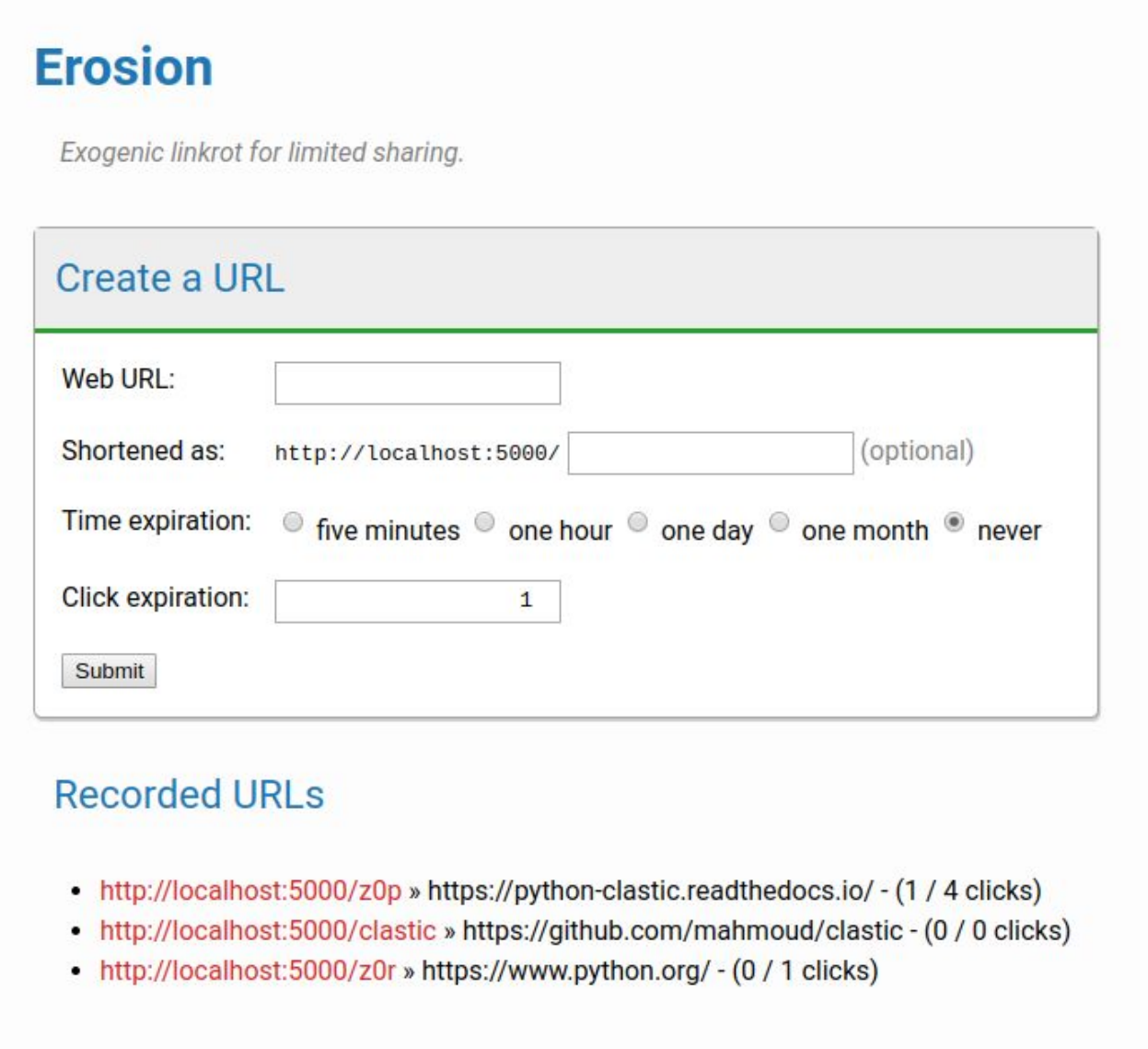

Fig. 2: The user can fill in a form to create a new link, or view recorded links. The first and last recorded links are autogenerated, whereas the second one is user-supplied.

For the sake of simplicity, we'll use the `shelve` module in the Python standard library as our storage backend. A stored link entry will consist of the target URL, the alias, the time when the link will expire, the maximum number of clicks, and the current number of clicks. The alias will be the key, and the full link data will be the value. Below is a simple implementation (file `storage.py`), without alias generation and link expiration features:

```
import os
import shelve
import time

```

(continues on next page)

(continued from previous page)

```
class LinkDB:
    def __init__(self, db_path):
        self.db_path = db_path
        if not os.path.exists(self.db_path):
            with shelve.open(self.db_path, writeback=True) as db:
                db["last_id"] = 41660
                db["entries"] = {}

    def add_link(self, target_url, alias=None, expiry_time=0, max_count=0):
        with shelve.open(self.db_path, writeback=True) as db:
            now = time.time()
            entry = {
                "target": target_url,
                "alias": alias,
                "expires": now + expiry_time if expiry_time > 0 else 0,
                "max_count": max_count,
                "count": 0,
            }
            db["entries"][alias] = entry
        return entry

    def get_links(self):
        with shelve.open(self.db_path) as db:
            entries = db["entries"].values()
        return entries

    def use_link(self, alias):
        with shelve.open(self.db_path, writeback=True) as db:
            entry = db["entries"].get(alias)
            if entry is not None:
                entry["count"] += 1
        return entry
```

The expiry time is given in seconds. Expiry values of zero for both time and clicks means that the link will not expire based on that property. It's also worth noting that the `.add_link()` method returns the newly added link. Since alias generation isn't implemented yet, the users will have to enter aliases themselves.

2.2.1 Getting started

Let's jump right in and start with the following template:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Erosion</title>
    <link rel="stylesheet" href="/static/style.css">
  </head>
  <body>
    <main class="content">
      <h1>Erosion</h1>
      <p class="tagline">Exogenic linkrot for limited sharing.</p>

      <section class="box">
        <h2>Create a URL</h2>
        <form method="POST" action="/submit" class="new">
          <p class="target">
            <label for="target_url">Web URL:</label>
            <input type="text" name="target_url">
          </p>

          <p>
            <label for="new_alias">Shortened as:</label>
            <span class="input-prefix">{host_url}</span>
            <input type="text" name="new_alias">
            <span class="note">(optional)</span>
          </p>

          <p>
            <label for="expiry_time" class="date-expiry-l">Time expiration:</label>
            <input type="radio" name="expiry_time" value="300"> five minutes
            <input type="radio" name="expiry_time" value="3600"> one hour
            <input type="radio" name="expiry_time" value="86400"> one day
            <input type="radio" name="expiry_time" value="2592000"> one month
            <input type="radio" name="expiry_time" value="0" checked> never
          </p>

          <p>
            <label for="max_count">Click expiration:</label>
            <input type="number" name="max_count" size="3" value="1">
          </p>

          <button type="submit">Submit</button>
        </form>
      </section>

      {?entries}
      <section>
        <h2>Recorded URLs</h2>
        <ul>
          {#entries}
          <li>
            <a href="{host_url}{.alias}">{host_url}{.alias}</a> &raquo; {.target} -
            <span class="click-count"> ({.count} / {.max_count} clicks)</span>
          </li>
          {/entries}
        </ul>
      </section>
```

(continues on next page)

(continued from previous page)

```
      {/entries}
    </main>
  </body>
</html>
```

This template consists of two major sections: one for adding a new entry, and one for listing recorded entries. It expects two items in the render context:

- `host_url` for the base URL of the application
- `entries` for the shortened links stored in the application

And now for the application code:

```
import os

from clastic import Application
from clastic.render import AshesRenderFactory
from clastic.static import StaticApplication


CUR_PATH = os.path.dirname(os.path.abspath(__file__))
STATIC_PATH = os.path.join(CUR_PATH, "static")


def home():
    return {"host_url": "http://localhost:5000", "entries": []}


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        ("/static", static_app),
    ]
    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, render_factory=render_factory)


app = create_app()

if __name__ == "__main__":
    app.serve()
```

This is a very simple application that doesn't do anything that wasn't covered in the *first part of the tutorial*. Apart from the static assets, the application has only one route. and its endpoint provides an initial context for the given template.

2.2.2 Resources

The first issue we want to solve is that of passing the host URL to the template because the application will not run on localhost in production. To achieve this, we need a way of letting the endpoint function get the host URL, so that it can put it into the render context. Clastic lets us register *resources* with the application; these will be made available to endpoint functions when requested.

Let's start by adding a simple, ini-style configuration file named `erosion.ini`, with the following contents:

```
[erosion]
host_url = http://localhost:5000
```

Now we can read this file during application creation:

```
from configparser import ConfigParser


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        ("/static", static_app),
    ]

    config_path = os.path.join(CUR_PATH, "erosion.ini")
    config = ConfigParser()
    config.read(config_path)

    host_url = config["erosion"]["host_url"].rstrip("/") + "/"
    resources = {"host_url": host_url}

    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, resources=resources, render_factory=render_factory)
```

The application resources are kept as items in a dictionary (`resources` in the example). After getting the host URL from the configuration file, we put it into this dictionary, which then gets registered with the application during application instantiation.

Endpoint functions can access application resources simply by listing their dictionary keys as parameters:

```
def home(host_url):
    return {"host_url": host_url}
```

Let's apply a similar solution for passing the entries to the template. First, add an option to the configuration file:

```
[erosion]
host_url = http://localhost:5000
db_path = erosion.db
```

Next, add the database connection to the application resources:

```
from storage import LinkDB


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        ("/static", static_app),
    ]

    config_path = os.path.join(CUR_PATH, "erosion.ini")
    config = ConfigParser()
    config.read(config_path)

    host_url = config["erosion"]["host_url"].rstrip('/') + '/'
```

(continues on next page)

(continued from previous page)

```
    db_path = config["erosion"]["db_path"]
    if not os.path.isabs(db_path):
        db_path = os.path.join(os.path.dirname(config_path), db_path)
    resources = {"host_url": host_url, "db": LinkDB(db_path)}

    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(routes, resources=resources, render_factory=render_factory)
```

And finally, use the database resource in the endpoint function:

```
def home(host_url, db):
    entries = db.get_links()
    return {"host_url": host_url, "entries": entries}
```

2.2.3 Redirection

Let's continue with creating new shortened links. The new link form submits its data to the `/submit` path. The endpoint function for this path has to receive the data, and add the new entry to the database. Once this is done, we don't want to display another page, we want to redirect the visitor back to the home page. Since the home page lists all entries, we should be able to see our newly created entry there. We use the `redirect()` function for this:

```
from clastic import redirect
from http import HTTPStatus


def add_entry(request, db):
    target_url = request.values.get("target_url")
    new_alias = request.values.get("new_alias")
    expiry_time = int(request.values.get("expiry_time"))
    max_count = int(request.values.get("max_count"))
    entry = db.add_link(
        target_url=target_url,
        alias=new_alias,
        expiry_time=expiry_time,
        max_count=max_count,
    )
    return redirect("/", code=HTTPStatus.SEE_OTHER)
```

What's left is adding this route to the application. If an endpoint function directly generates a response -as our example does via redirection- there is no need for a renderer:

```
from clastic import POST


def create_app():
    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        POST("/submit", add_entry),
        ("/static", static_app),
    ]

    ...
```

We add this route as a *POST* route. This makes sure that other HTTP methods will not be allowed for this path.

You can try typing the address `http://localhost:5000/submit` into the location bar of your browser, and you should see a *MethodNotAllowed* error. There are also other method-restricted routes, like *GET*, *PUT*, and *DELETE*.

2.2.4 Named path segments

Now let's turn to using the shortened links. Any path other than the home page, the form submission path `/submit`, and static asset paths under `/static` will be treated as an alias, and we'll redirect the browser to its target URL.[1] It makes sense to make this a GET-only route:

```
from clastic import GET


routes = [
    ("/", home, "home.html"),
    POST("/submit", add_entry),
    ("/static", static_app),
    GET("/<alias>", use_entry),
]
```

Important: Note that the ordering of the routes is significant. Clastic will try dispatch a request to an endpoint function in the given order of routes.

Angular brackets in route paths are used to name segments. The part of the path that matches the segment will then be available to the endpoint function as a parameter by the same name:

```
def use_entry(alias, db):
    entry = db.use_link(alias)
    return redirect(entry["target"], code=HTTPStatus.MOVED_PERMANENTLY)
```

2.2.5 Errors

But what if there is no such alias recorded? A sensible thing to do would be to return a `NotFound` error:

```
from clastic.errors import NotFound


def use_entry(alias, db):
    entry = db.use_link(alias)
    if entry is None:
        return NotFound()
    return redirect(entry["target"], code=HTTPStatus.MOVED_PERMANENTLY)
```

2.2.6 Using middleware

Clastic allows us to use *middleware* to keep endpoint functions from having to deal with routine tasks such as serialization, logging, database connection management, and the like. For example, the `PostDataMiddleware` can be used to convert submitted form data into appropriate types and make them available to endpoint functions as parameters:

[1] You should remember that a browser can make an automatic request for the site's favicon at an address like `/favicon.ico`. Our code will treat this as a missing alias.

```
from clastic.middleware.form import PostDataMiddleware


def create_app():
    new_link_mw = PostDataMiddleware(
        {"target_url": str, "new_alias": str, "expiry_time": int, "max_count": int}
    )

    static_app = StaticApplication(STATIC_PATH)
    routes = [
        ("/", home, "home.html"),
        POST("/submit", add_entry, middlewares=[new_link_mw]),
        ("/static", static_app),
        GET("/<alias>", use_entry),
    ]

    ...
```

The endpoint function doesn't need to get the data from `request.values` anymore:

```
def add_entry(db, target_url, new_alias, expiry_time, max_count):
    entry = db.add_link(
        target_url=target_url,
        alias=new_alias,
        expiry_time=expiry_time,
        max_count=max_count,
    )
    return redirect("/", code=HTTPStatus.SEE_OTHER)
```

2.2.7 Cookies

At the moment, after adding a new entry, the endpoint function only redirects to the home page. Say we want to display a notice to the user indicating that the entry was successfully added. This requires passing the new entry data from the `add_entry()` endpoint function to the `home()` endpoint function. But redirection means a new HTTP request and we need a way of passing data over this new request. One way to achieve this would be using a cookie: the `add_entry()` function places the data in a cookie, and the `home()` function picks it up from there.

Cookies can be accessed through `request.cookies`, but in this example we want to use a signed cookie. Clastic includes a `SignedCookieMiddleware` for this purpose. This time we're going to register the middleware at the application level rather than for just one route. The secret key for signing the cookie will be read from the configuration file:

```
from clastic.middleware.cookie import SignedCookieMiddleware


def create_app():
    ...

    cookie_secret = config["erosion"]["cookie_secret"]
    cookie_mw = SignedCookieMiddleware(secret_key=cookie_secret)

    render_factory = AshesRenderFactory(CUR_PATH)
    return Application(
        routes,
        resources=resources,
```

(continues on next page)

(continued from previous page)

```
        middlewares=[cookie_mw],
        render_factory=render_factory,
    )
```

If a function wants to access this cookie, it just has to declare a parameter named `cookie`.

Here's how the first endpoint function stores the new alias in the cookie:

```
def add_entry(db, cookie, target_url, new_alias, expiry_time, max_count):
    entry = db.add_link(
        alias=new_alias,
        target_url=target_url,
        expiry_time=expiry_time,
        max_count=max_count,
    )
    cookie["new_entry_alias"] = new_alias
    return redirect("/", code=HTTPStatus.SEE_OTHER)
```

And here's how the second endpoint function gets the alias from the cookie, and puts it into the render context:

```
def home(host_url, db, cookie):
    entries = db.get_links()
    new_entry_alias = cookie.pop("new_entry_alias", None)
    return {
        "host_url": host_url,
        "entries": entries,
        "new_entry_alias": new_entry_alias,
    }
```

And a piece of markup is needed in the template to display the notice:

```
<h1>Erosion</h1>
<p class="tagline">Exogenic linkrot for limited sharing.</p>

{#new_entry_alias}
<p class="message">
  Successfully created <a href="{host_url}{.}">{host_url}{.}</a>.
</p>
{/new_entry_alias}
```

For the alias generation and link expiration features, you can refer to the full application code in the repo. To make this example into a real-world application, the storage module must be modified to handle concurrent requests.

2.3 Applications and Routes

When it comes to Python and the web, the world speaks WSGI (Web Server Gateway Interface). And a Clastic provides exactly that: A WSGI `Application`.

Clastic Applications are composed using Python code, plain and simple. No decorators, no `settings.py`, no special configuration file. Just constructed objects, used to construct other objects.

Specifically, Applications consist of *Routes*, Resources, *Middleware*, and *Error Handlers*.

2.3.1 The Application

class clastic.**Application**(*routes=None*, *resources=None*, *middlewares=None*, *render_factory=None*, *error_handler=None*, ***kwargs*)

The central object around which Clastic revolves.

The Application initializer checks that all endpoints, render functions, and middlewares have their dependencies satisfied before completing construction. If the signatures don't line up, an `NameError` will be raised.

Parameters

- **routes** (*list*) – A list of Route instances, SubApplications, or tuples. Defaults to `[]`. Add more with *add()*.
- **resources** (*dict*) – A dict which will be injectabled to Routes and middlewares in this Application. Keys must all be strings, values can be any Python object. Defaults to `{}`.
- **middlewares** (*list*) – A list of *Middleware* objects. Defaults to `[]`.
- **render_factory** (*callable*) – An optional callable to convert render arguments into callables, such as `AshesRenderFactory`.
- **debug** (*bool*) – Set to `True` to enable certain debug behavior in the application.
- **error_handler** – *Advanced*: An optional *ErrorHandler* instance. Defaults to *ErrorHandler*. If *debug* is `True`, defaults to *ContextualErrorHandler*.
- **slash_mode** (*str*) – *Advanced*: Controls how the Application handles trailing slashes. One of `clastic.S_REDIRECT`, `S_STRICT`, `S_REWRITE`. Defaults to `S_REDIRECT`.

In addition to arguments, certain advanced behaviors can be customized by inheriting from *Application* and overriding attributes: *request_type*, *response_type*, *default_error_handler_type*, and *default_debug_error_handler_type*.

add(*entry*, *index=None*, ***kwargs*)

Add a *Route* or *SubApplication*. A tuple may also be passed, which will be converted accordingly.

Note that as each Route is bound, the Application checks whether the Route's dependencies can be satisfied by the Application.

default_debug_error_handler_type

alias of *clastic.errors.ContextualErrorHandler*

default_error_handler_type

alias of *clastic.errors.ErrorHandler*

get_local_client()

Get a simple local client suitable for using in tests. See Werkzeug's test Client for more info.

request_type

alias of `werkzeug.wrappers.request.Request`

response_type

alias of `werkzeug.wrappers.response.Response`

serve(*address='0.0.0.0'*, *port=5000*, *use_meta=True*, *use_lint=True*, *use_reloader=True*, *use_debugger=True*, *use_static=True*, *static_prefix='static'*, *static_path=None*, *processes=None*, ***kw*)

Serve the Application locally, suitable for development purposes.

Parameters

- **address** (*str*) – IP address to bind to (defaults to `"0.0.0.0"`, which works for all IPs)

- **port** (*int*) – Port to bind on (defaults to `5000`)
- **use_meta** (*bool*) – Whether to automatically add the *MetaApplication* to `/_meta/`. Defaults to `True`.
- **use_reloader** (*bool*) – Whether to automatically reload the application when changes to the source code are saved. Defaults to `True`.
- **use_debugger** (*bool*) – Whether to wrap the Application in werkzeug's debug middleware for interactive debugging. (Note that a PIN will be output on stdout and must be used to interact with the error pages.)
- **use_static** (*bool*) – Whether to automatically serve *static_path* under *static_prefix*. Defaults to `True`.
- **static_prefix** (*str*) – The URL path where static assets will be served. Defaults to */static/*.
- **static_path** (*str*) – The filesystem path to static assets to serve if *use_static* is `True`. Defaults to a path named "static" in the current directory (`"./static/"`).
- **processes** (*int*) – Number of processes to serve (not recommended for use with *use_debugger*). (Use sparingly; not for production.)

> **Warning:** The server provided by this method is not intended for production traffic use.

set_error_handler(*error_handler=None*)

Sets the *ErrorHandler* instance. Call without arguments to reset the error handler to default.

> **Note:** This method does not reset error handlers in Routes which have already been bound.

2.3.2 Route Types

class clastic.**Route**(*pattern*, *endpoint*, *render=None*, *render_error=None*, ***kwargs*)

While Clastic may revolve around the *Application*, Applications would be nothing without the Routes they contain.

The *Route* object is basically a combination of three things:

1. A path *pattern*
2. An *endpoint* function
3. A *render* function or argument

Put simply, when a request matches a Route's *pattern*, Clastic calls the Route's *endpoint* function, and the result of this is passed to the Route's *render* function.

In reality, a Route has many other levers to enable more routing features.

Parameters

- **pattern** (*str*) – A *Pattern Mini-Language*-formatted string.
- **endpoint** (*callable*) – A function to call with *Injectables*, which returns a Response or a render context which will be passed to the Route's *render* function.

- **render** (*callable*) – An optional function which converts the output of *endpoint* into a Response. Can also be an argument which is passed to an Application's render_factory to generate a render function. For instance, a template name or path.
- **middlewares** (*list*) – An optional list of middlewares specific to this Route.
- **resources** (*dict*) – An optional dict of resources specific to this Route.
- **methods** (*list*) – A list of text names of HTTP methods which a request's method must match for this Route to match.
- **render_error** (*callable*) – *Advanced*: A function which converts an *HTTPException* into a Response. Defaults to the Application's *error handler*.

class clastic.**GET**(**a*, ***kw*)
A *Route* subtype which only matches for GET requests.

class clastic.**POST**(**a*, ***kw*)
A *Route* subtype which only matches for POST requests.

class clastic.**PUT**(**a*, ***kw*)
A *Route* subtype which only matches for PUT requests.

class clastic.**DELETE**(**a*, ***kw*)
A *Route* subtype which only matches for DELETE requests.

Note: Method-specific subtypes have identical signatures to `Route`.

The only steps necessary to make a Route method-specific is to import the type and add it to the tuple:

```
Application(routes=[("/home/", home_ep, render_func)])
```

Becomes:

```
from clastic import GET
...
Application(routes=[GET("/home/", home_ep, render_func)])
```

If an Application contains Routes which match the path pattern, but none of the Routes match the method, Clastic will automatically raise a `MethodNotAllowed` exception for you, which results in a `405` HTTP error response to client.

2.3.3 SubApplications

Clastic features strong composability using straightforward Python constructs. An `Application` contains `Route` instances, and those Routes can come from other Applications, using `SubApplication`.

class clastic.**SubApplication**(*prefix*, *app*, *rebind_render=False*, *inherit_slashes=True*)
Enables *Application* instances to be embedded in other Applications.

Note that Routes are copied into the embedding Application, and further modifications to the Application after embedding may not be visible in the root Application.

Parameters

- **prefix** (*str*) – The path prefix under which the embedded Application's routes will appear. `/` is valid, and will merge the routes in at the root level of the parent application.
- **app** (`Application`) – The Application instance being embedded.

- **rebind_render** (*bool*) – **Advanced**: Whether render arguments should be reinterpreted by the embedding application's render factory. Defaults to `False`.
- **inherit_slashes** (*bool*) – **Advanced**: Whether to inherit the embedding application's handling of trailing slashes. Defaults to `True`.

Note: This object rarely needs to be constructed manually, because this behavior is built in to the default Application routes argument handling. `Application(routes=[('/prefix', other_app)])` automatically creates a SubApplication and embeds it.

2.3.4 Injectables

Clastic automatically provides dependencies to middlewares and endpoint/render functions. These dependencies can come from one of four sets:

1. **Route path pattern**
2. **Application resources** - Arguments which are valid for the lifespan of the `Application`, like configuration variables.
3. **Middleware provides** - Arguments provided by an Application's middleware. See *Middleware* for more information.
4. **Clastic built-ins** - Special arguments that are always made available by Clastic. These arguments are also reserved, and conflicting names will raise an exception. A list of these arguments and their meanings is below.

Clastic provides a small, but powerful set of built-in arguments for every occasion. These arguments are reserved by Clastic, so know them well.

Note: Advanced and primarily-internal built-ins are prefixed with an underscore.

Built-in injectables

- *request*
- *next*
- *context*
- *_application*
- *_route*
- *_error*
- *_dispatch_state*

request

Probably the most commonly used built-in, `request` is the current `Request` object being handled by the Application. It has the URL arguments, POST parameters, cookies, user agent, other HTTP headers, and everything from the WSGI environ. *request*

next

`next` is only for use by Middleware, and represents the next function in the execution chain. It is called with the arguments the middleware class declared that it would provide. If the middleware does not provide any arguments, then it is called with no arguments.

`next` allows a middleware to not worry about what middleware or function comes after it in the chain. All the middleware knows is that the result of (or exception raised by) the `next` function is the Response that a client would receive.

Middleware functions must accept `next` as the first argument. If a middleware function does not accept the `next` argument, or if a non-middleware function accepts the `next` argument, an exception is raised at Application initialization.

context

`context` is the output of the endpoint side of the middleware chain. By convention, it is almost always a dictionary of values meant to be used in templating or other sorts of Response serialization.

Accepting the `context` built-in outside of the render branch of middleware will cause an exception to be raised at Application initialization. *context*

_application

The `Application` instance in which this middleware or endpoint is currently embedded. The Application has access to all routes, endpoints, middlewares, and other fun stuff, which makes `_application` useful for introspective activities, like those provided by Clastic's built-in `MetaApplication`.

_route

The Route which was matched by the URL and is currently being executed. Also mostly introspective in nature. `_route` has a lot of useful attributes, such as `endpoint`, which can be used to shortcut execution in an extreme case.

_error

Only available to the `render_error` functions/methods configured, this built-in is available when an `HTTPException` has been raised or returned.

_dispatch_state

An internally-managed variable used by Clastic's routing machinery to generate useful errors. See `DispatchState` for more info.

And, that's it! All other argument names are unreserved and yours for the binding.

2.3.5 Clastic Routing in a Nutshell

- Routes are always checked in the same order they were added to the Application. Some frameworks reorder routes, but not Clastic.
- Route methods must also match, or a `MethodNotAllowed` is raised.

- If a Route pattern matches, except for a trailing slash, the Application may redirect or rewrite the request, depending on the Application/Route's `slash_mode`.

2.3.6 Pattern Mini-Language

Route patterns use a minilanguage designed to minimize errors and maximize readability, while compiling to Python regular expressions remaining powerful and performant.

- Route patterns are a subset of regular expressions designed to match URL paths, and is thus aware of slashes. Slashes separate "segments", which can be one of three types: string, int, float.
- By default a pattern segment matches one URL path segment, but clastic also supports matching multiples of segments at once: (":" matches one segment, "?" matches zero or one segment, "*" matches 0 or more segments, and "+" matches 1 or more segments).
- Segments are always named, and the names are checked against other injectables for conflicts.
- Be careful when getting too fancy with URL patterns. If your pattern doesn't match, by default users will see a relatively plain 404 page that does not offer much help as to why their URL is incorrect.

2.3.7 Advanced Routing

- Unlike Werkzeug/Flask's default routing, clastic does not reorder routes. Routes are matched in order.
- Applications can choose whether to redirect on trailing slashes
- Clastic's one-of-a-kind routing system allows endpoint functionsand middlewares to participate in routing by raising certain standard errors, telling clastic to continue to check other routes
- It's even possible to route to a separate WSGI application (i.e., an application not written in Clastic)
- NullRoute (configurable)

class clastic.**RerouteWSGI**(*wsgi_app*)
Raise or use as a route endpoint to route to a different WSGI app.

Note that this will have unintended consequences if you have done stateful operations to the environ (such as reading the body of the request) or already called start_response or something similar.

It's safest to put this high in the routing table (and middleware stack).

class clastic.application.**DispatchState**
The every request handled by an `Application` creates a *DispatchState*, which is used to track relevant state in the routing progress, including which routes were attempted and what exceptions were raised, if any.

Note: Objects of this type are constructed internally and are not really part of the Clastic API, except that they are one of the built-in injectables.

2.4 Middleware

Coming soon!

- "M"-based design (request, endpoint, render)
- Dependency injection (like pytest!)
- autodoc * inventory of all production-grade middlewares (separate doc(s)?)

2.5 Errors

Errors matter in the HTTP/RESTful ecosystem.

Clastic offers a full set of exception types representing standard HTTP errors, and a base HTTPException for creating your own exception types.

The `errors` module also contains the Error Handler subsystem for controlling how a Clastic Application behaves in error situations.

Contents

2.5.1 Error Handlers

You can control how Clastic responds to various error scenarios by creating or configuring an *Error Handler* and passing it to your `Application` instance.

Error Handlers are able to:

- Control which specific error types are raised on routing failures (e.g., `NotFound` and *`MethodNotAllowed`*).
- Control the error type which is raised when the endpoint or render function raises an uncaught exception (e.g., `InternalServerError`)
- How uncaught exceptions are rendered or otherwise turned into HTTP Responses
- Configure a WSGI middleware for the whole *`Application`*

The easiest way to control these behavior is to inherit from the default `ErrorHandler` and override the attributes or methods you need to change:

class clastic.errors.**ErrorHandler**(***kwargs*)

The default Clastic error handler. Provides minimal detail, suitable for a production setting.

Parameters **reraise_uncaught** (*bool*) – Set to *True* if you want uncaught exceptions to be handled by the WSGI server rather than by this Clastic error handler.

exc_info_type

alias of `boltons.tbutils.ExceptionInfo`

method_not_allowed_type

alias of *`MethodNotAllowed`*

not_found_type

alias of `NotFound`

render_error(*request*, *_error*)

Turn an `HTTPException` into a Response of your

Like endpoints and render functions, `render_error()` supports injection of any built-in arguments, as well as the *_error* argument (an instance of `HTTPException`, so feel free to adapt the signature as needed.

This method is attached to Routes as they are bound into Applications. Routes can technically override this behavior, but generally a Route's error handling reflects that of the Error Handler in the root application where it is bound.

By default this method just adapts the response between text, HTML, XML, and JSON.

server_error_type
alias of `InternalServerError`

uncaught_to_response(*_application*, *_route*, ***kwargs*)
Called in the `except:` block of Clastic's routing. Must take the currently-being-handled exception and **return** a response instance. The default behavior returns an instance of whatever type is set in the *server_error_type* attribute (`InternalServerError`, by default).

Note that when inheriting, the method signature should accept `**kwargs`, as Clastic does not inject arguments as it does with endpoint functions, etc.

wsgi_wrapper = None

The default error handler presents the minimal detail to the client when an error occurs.

Clastic ships with a couple special Error Handlers which it uses to enable debuggability.

class clastic.errors.**ContextualErrorHandler**(**a*, ***kw*)
An error handler which offers a bit of debugging context, including a stack and locals (for server errors) and routes tried (for 404s).

Might be OK for some internal tools, but should generally not be used for production.

exc_info_type
alias of `boltons.tbutils.ContextualExceptionInfo`

not_found_type
alias of *ContextualNotFound*

server_error_type
alias of *ContextualInternalServerError*

class clastic.errors.**REPLErrorHandler**(**a*, ***kw*)
This error handler wraps the Application in a Werkzeug debug middleware.

2.5.2 The Base HTTPException

class clastic.**HTTPException**(*detail=None*, ***kwargs*)
The base `Exception` for all default HTTP errors in this module, the HTTPException also inherits from `BaseResponse`, making instances of it and its subtypes valid to use via raising, as well as returning from endpoints and render functions.

Parameters

- **detail** (*str*) – A string with information about the exception. Appears in the body of the HTML error page.
- **code** (*int*) – A numeric HTTP status code (e.g., `400` or `500`).
- **message** (*str*) – A short name for the error, (e.g., `"Not Found"`)
- **error_type** (*str*) – An error type name **or link** to a page with details about the type of error. Useful for linking to docs.
- **is_breaking** (*bool*) – *Advanced*: For integrating with Clastic's routing system, set to `True` to specify that this error instance should not preempt trying routes further down the routing table. If no other route matches or succeeds, this error will be raised.

- **source_route** (Route) – *Advanced*: The route instance that raised this exception.
- **mimetype** (*str*) – A MIME type to return in the Response headers.
- **content_type** (*str*) – A Content-Type to return in the Response headers.
- **headers** (*dict*) – A mapping of custom headers for the Response. Defaults to None.

Note: The base HTTPException includes simple serialization to text, HTML, XML, and JSON. So if a client requests a particular format (using the Accept header), it will automatically respond in that format. It defaults to text/plain if the requested MIME type is not recognized.

2.5.3 Standard HTTP Error Types

In addition to error handling mechanisms, clastic.error ships with exception types for every standard HTTP error.

Because these standard error types inherit from HTTPException, which is both an exception and a Response type, they can be raised or returned.

Errors are organized by error code, in ascending order. Note that the message attribute is sometimes called the "name", e.g. "Not Found" for 404.

exception clastic.errors.**BadRequest**(*detail=None*, ***kwargs*)

code = 400

detail = 'Your web client or proxy sent a request that this endpoint could not understa

message = 'Bad Request'

exception clastic.errors.**Unauthorized**(*detail=None*, ***kwargs*)

code = 401

detail = 'The endpoint could not verify that your client is authorized to access this

message = 'Authentication required'

exception clastic.errors.**PaymentRequired**(*detail=None*, ***kwargs*)

HTTP cares about your paywall.

code = 402

detail = "This endpoint requires payment. Money doesn't grow on HTTPs, you know."

message = 'Payment required'

exception clastic.errors.**Forbidden**(*detail=None*, ***kwargs*)

code = 403

detail = "You don't have permission to access the requested resource."

message = 'Access forbidden'

exception clastic.errors.**ContextualNotFound**(**a*, ***kw*)

to_dict()
One design ideal, for showing which routes have been hit: [{'route': ('pattern', 'endpoint', 'render_func'), 'path_matched': False, 'method_matched': False, 'slash_matched': False}]

to_html(*a, **kw)

exception clastic.errors.**MethodNotAllowed**(*allowed_methods=None*, **args*, ***kwargs*)

code = 405

detail = 'The method used is not allowed for the requested URL.'

message = 'Method not allowed'

exception clastic.errors.**NotAcceptable**(*detail=None*, ***kwargs*)

code = 406

detail = "The endpoint cannot generate a response acceptable by your client (as specif

message = 'Available content not acceptable'

exception clastic.errors.**ProxyAuthenticationRequired**(*detail=None*, ***kwargs*)

code = 407

detail = 'A proxy between your server and the client requires authentication to access

message = 'Proxy authentication required'

exception clastic.errors.**RequestTimeout**(*detail=None*, ***kwargs*)

code = 408

detail = 'The server cancelled the request because the client did not complete the req

message = 'Request timed out'

exception clastic.errors.**Conflict**(*detail=None*, ***kwargs*)

code = 409

detail = 'The endpoint cancelled the request due to a potential conflict with existing

message = 'A conflict occurred'

exception clastic.errors.**Gone**(*detail=None*, ***kwargs*)

code = 410

detail = 'The requested resource is no longer available on this server and there is no

message = 'Gone'

exception clastic.errors.**LengthRequired**(*detail=None*, ***kwargs*)

code = 411

detail = 'A request for this resource is required to have a valid Content-Length heade

message = 'Length required'

exception clastic.errors.**PreconditionFailed**(*detail=None*, ***kwargs*)

code = 412

detail = 'A required precondition on the request for this resource failed positive eva

message = 'Precondition failed'

exception clastic.errors.**RequestEntityTooLarge**(*detail=None*, ***kwargs*)

code = 413

detail = 'The method/resource combination requested does not allow data to be transmit

message = 'Request entity too large'

exception clastic.errors.**RequestURITooLong**(*detail=None*, ***kwargs*)

code = 414

detail = 'The length of the requested URL exceeds the limit for this endpoint/server.'

message = 'Request URL too long'

exception clastic.errors.**UnsupportedMediaType**(*detail=None*, ***kwargs*)

code = 415

detail = 'The server does not support the media type transmitted in the request. Try a

message = 'Unsupported media type'

exception clastic.errors.**RequestedRangeNotSatisfiable**(*detail=None*, ***kwargs*)

code = 416

detail = 'The client sent a ranged request not fulfillable by this endpoint.'

message = 'Requested range not satisfiable'

exception clastic.errors.**ExpectationFailed**(*detail=None*, ***kwargs*)

Can't. always. get. what you want.

code = 417

detail = "The server could not meet the requirements indicated in the request's Expect

message = 'Expectation failed'

exception clastic.errors.**ImATeapot**(*detail=None*, ***kwargs*)

Standards committees are known for their senses of humor.

code = 418

detail = 'This server is a teapot, not a coffee machine, and would like to apologize i

message = "I'm a teapot: short, stout."

exception clastic.errors.**UnprocessableEntity**(*detail=None*, ***kwargs*)

code = 422

detail = 'The client sent a well-formed request, but the endpoint encountered other se

message = 'Unprocessable entity'

exception clastic.errors.**UpgradeRequired**(*detail=None*, ***kwargs*)
Used to upgrade connections (to TLS, etc., RFC2817). Also WebSockets.

code = 426

detail = 'The server requires an upgraded connection to continue. This is expected beha

message = 'Upgrade required'

exception clastic.errors.**PreconditionRequired**(*detail=None*, ***kwargs*)

code = 428

detail = "This endpoint requires a request with a conditional clause. Try resubmitting

message = 'Precondition required'

exception clastic.errors.**TooManyRequests**(*detail=None*, ***kwargs*)

code = 429

detail = 'The client has exceeded the allowed rate of requests for this resource. Pleas

message = 'Too many requests'

exception clastic.errors.**RequestHeaderFieldsTooLarge**(*detail=None*, ***kwargs*)

code = 431

detail = 'One or more HTTP header fields exceeded the maximum allowed size.'

message = 'Request header fields too large'

exception clastic.errors.**UnavailableForLegalReasons**(*detail=None*, ***kwargs*)
Sit back and enjoy the Bradbury

code = 451

detail = 'The resource requested is unavailable for legal reasons. For instance, this

message = 'Unavailable for legal reasons'

exception clastic.errors.**ContextualInternalServerError**(**a*, ***kw*)
An Internal Server Error with a full contextual view of the exception, mostly for development (non-production) purposes.

NOTE: The dict returned by to_dict is not JSON-encodable with the default encoder. It relies on the ClasticJSONEncoder currently used in the InternalServerError class.

to_dict(**a*, ***kw*)

to_html(**a*, ***kw*)

exception clastic.errors.**NotImplemented**(*detail=None*, ***kwargs*)

code = 501

detail = 'The resource requested has either not been implemented or does not yet suppo

message = 'Response behavior not implemented'

exception clastic.errors.**BadGateway**(*detail=None*, ***kwargs*)

```
code = 502
detail = 'The endpoint received an invalid response from an upstream server while proc
message = 'Bad gateway'
```

exception clastic.errors.**ServiceUnavailable**(*detail=None*, ***kwargs*)

```
code = 503
detail = 'The service or resource requested is temporarily unavailable due to maintena
message = 'Service or resource unavailable'
```

exception clastic.errors.**GatewayTimeout**(*detail=None*, ***kwargs*)

```
code = 504
detail = 'The endpoint timed out while waiting for a response from an upstream server.
message = 'Gateway timeout'
```

exception clastic.errors.**HTTPVersionNotSupported**(*detail=None*, ***kwargs*)

```
code = 505
detail = 'The endpoint does not support the version of HTTP specified by the request.'
message = 'HTTP version not supported'
```

2.6 The MetaApplication

Coming Soon

2.7 Static

All web applications consist of a mix of technologies. Most human-facing websites serve up JavaScript, CSS, and HTML.

Some of these files make sense to generate dynamically, others can be served from a file on the filesystem. Here is where Clastic's static serving facilities shine.

- StaticFileRoute * For when you have a single file at a single path * Will check for existence of file at startup by default, to be safe
- StaticApplication * For when you have a directory

2.7.1 Advanced static serving

- StaticApplications can overlap in paths, and if the first Application can't locate the requested resource, the second Application will try, and so on. This makes it easy to serve multiple directories' files from the same URL path.

2.8 Troubleshooting

TODO (should this be "Best Practices" or similar?)

Building web applications is never as easy as it seems. Luckily, clastic was built with debuggability in mind.

2.8.1 Flaw

If you're using the built-in dev server, by default it reloads the application when you save a file that's part of the application. If you accidentally save a typo, Clastic will boot up a failsafe application on the same port and show you a stack trace to help you track down the error. Save again, and your application should come back up.

2.8.2 Debug Mode

By default, when running the built-in dev server, Clastic exposes werkzeug's debug application.

2.8.3 Project structure

TODO

- app.py * create_app() function * app = ...

2.9 Clastic Compared

TODO (might belong in the FAQ instead)

In the Python world, you certainly have a lot of choices among web frameworks. Software isn't a competition, but there are good reasons to use clastic.

- Simple, Python-based API designed to minimize learning curve
- Minimum global state, designed for concurrency
- Respectable performance *in microbenchmarks <https://github.com/the-benchmarker/web-frameworks>* (not that it matters)
- Rich dependency semantics guarantee that endpoints, URLs, middlewares, and resources line up before the Application will build to start up.

2.9.1 Compared to Django

TODO

2.9.2 Compared to Flask

TODO

Python Module Index

C

Index

E

F

G

H

I

L

M

N

P

R

S

T

U

W