Web Application A to Z

Moshe Zadka – https://cobordism.com

PyCon 2018

Welcome to the Web applications, A-to-Z tutorial. A-to-Z means we will take care of all steps needed to write and deploy web applications. Modern web applications are written for, at least continuous integration, and almost always continuous deployment – at least to a test environment.

Even companies who do human-managed releases have *a* fast-paced environment which gets continuously deployed. Continuous deployment is no longer a luxury: it is now the table stakes of web application development.

Likewise, having an API and a continously updating UI is also table stakes. Since browsers only know one language (JavaScript) we have a total of one choice of language for the front-end. Everywhere else, there are many choices – with complicated trade-offs. This tutorial will weave a path, making some choices.

Goal

- Understand high-level
- Get needed skills
- ...but perfecting them takes a lifetime

By the end of this tutorial, you will understand, at a high-level, all the moving pieces that go into assembling a modern web application. You will also get rudimentary skills at assembling those. Perfecting those skills takes a lifetime – and nobody is really a "full-stack" engineer: everyone specializes in one area. But understanding all of them is useful.

Web applications

- Front-end (JavaScript)
- Back-end (Server)
- Storage (Database)

Web applications are commonly built using a three-tier structure: front-end, running on the browser, in JavaScript. A back-end, running on servers, written in any one of many languages. A storage engine keeping long-term states – users, content and metadata.

Web applications - Front-End

React – JavaScript framework

For our front end we will choose the popular React framework. JavaScript frameworks are hotly debated. We choose React partly for familiarity. However, React also has the advantage that it has been around for four years (a long time in JavaScript time), and still going strong (2nd most popular framework on Stack Ovrerflow survey). React is heavily developed by Facebook, and has great self-learning resources.

Web applications - Back-End

- Python
- WSGI
- Pyramid

The backend in web applications is completely under our control. The language of choice is Python – a popular choice, and has been a popular choice since 2001. Not always the *most* popular choice, but a lot of the previously popular choices have been left in the dust, while Python always has a "new thing" to contend with.

The most popular way to develop web applications with Python is WSGI. While some non-WSGI options exist, Tornado and Klein among them, almost all big Python applications use WSGI.

WSGI, however, is too low-level. Django is the 800lb gorilla, of course. Flask is the fast-and-small thing. Pyramid occupies a nice Goldilock zone – neither hyper-optimized for small applications, nor assuming that you need three extra files just for configuration.

Web applications - Storage

- SQLite
- SQLalchemy

Most applications have some persistent state. The traditional route is to keep the state in a SQL-based database. For simplicity, we will use SQLite, since it is an in-app database. We will use the SQLalchemy library to interact with it.

Web applications - Platform

- Twisted
- Linux
- Docker

Applications need to run *on* something. We will use Twisted as our WSGI container – its Python oriented configuration mechanism allows us to use it for all layers of web serving, eliminating some elements sometimes seen in these stacks, such as nginx.

We will use Linux as our operating system, which in 2018 is the vanilla default choice. We will use Docker as our application isolation mechanism. Docker manages to pull several duties here: it is both a packaging format, as well as a way to start and stop applications.

Web applications - End to End Testing

- Selenium
- Geckodriver
- Firefox
- Python
- Docker

Finally, we need to run tests. Unit tests are, of course, great to have. But ultimately, we need an end-to-end testing framework. For that, we will use Selenium. Selenium is wrapping browsers, so we need a browser – we will choose Firefox. Between Selenium and the browser comes a specialized abstraction layer called a web driver – this is what lets Selenium easily support several browsers – and the one for Firefox is called Geckodriver.

We need a language to write the tests in, and luckily, Python is pretty popular in the app-testing space. Finally, we need to package this complexity app, and once again we will use Docker as our packaging solution.

We will start by showing a simple Pyramid application. This application just displays a greeting to the user.

First, we need to import the module.

Pyramid: Imports

```
from pyramid import config, response
```

Next, we implement the business logic – such as we have here.

Pyramid: Logic

```
def hello_world(request):
    return response.Response('Hello World!')
```

Finally, we configure the routing table and make a WSGI application – which is just a Python object.

Pyramid: Routing

```
with config.Configurator() as cfg:
    cfg.add_route('hello', '/')
    cfg.add_view(hello_world, route_name='hello')
    app = cfg.make_wsgi_app()
```

In order to run it, we use the Twisted WSGI web plugin.

Pyramid: Running

```
$ python -m twisted web --wsgi hello.app
```

Now that we have greeted the world, it is time to write an application with actual logic. We will write a simple "shared blog". Each post is just a blob of text. We will start with some hard-coded blobs of text, in order to exercise our code.

Returning Posts: Hardcoding

```
POSTS = [
    "Just chillin'",
    "Writing code",
    "Being awesome",
]
```

Now, let's implement a Pyramid method that formats the posts as JSON and returns them. JSON is the lingua franca of the web – particularly since browsers can parse it efficiently into JavaScript objects.

Returning Posts: Logic

```
def posts(request):
    body = json.dumps(POSTS).encode('utf-8')
    return response.Response(
                content_type='application/json',
                body=body)
```

Since the definition of a "blog" is something you can post to, we need a method to post something. JSON works in this direction too – browsers know how to serialize JSON efficiently as well.

Adding Post: Logic

```
def add_post(request):
    POSTS.append(request.json_body['content'])
    return response.Response('ok')
```

Finally, we need to put this all in Pyramid WSGI application, and set the routing. Note that in a typical RESTful way, we use the same URL for both actions – but with different HTTP methods. "That's how GitHub does it", for example.

Returning Posts: Routing

```
with config.Configurator() as cfg:
    cfg.add_route('posts', '/posts',
                  request_method='GET')
    cfg.add_view(posts, route_name='posts')
    cfg.add_route('add_post', '/posts',
                  request_method='POST')
    cfg.add_view(add_post, route_name='add_post')
    app = cfg.make_wsgi_app()
```

Now, since it is a *shared* blog, we really should let people attach their names to posts. Let's go ahead and add an "author" – make each post a pair of author and content.

Hands on: Add Author

15 minutes :25-:40

- Get application to run as-is
- Add an author field to post
- Finished? Add a date field
- Finished? Allow updating a post

This is the "backend" – logic that runs on the server. However, human beings do not like to interact with JSON directly. For the humans, we are going to build a glitzy UI. In order to accelerate the process, we will leave the glitz behind – and concentrate on the UI part.

As mentioned, we are using the React framework. More sophisticated uses would have us use webpack and nvm, but we are going to go raw – just put the React logic in the HTML, without preprocessing.

Using React: Header

```
<!DOCTYPE html>
<html>
<head>
<script src="https://fb.me/react-0.14.1.js">
</script>
<script src="https://fb.me/react-dom-0.14.1.js">
</script>
<script src=
```

```
"https://unpkg.com/babel-core@5.8.3/browser.min.js"
></script>
</head>
```

Next, React works by having an element that it controls. In more complicated use cases, this allows to set basic parameters around it, but in this case – React gets to control the entire body.

Using React: Body

```
<body>
<div id="content"></div>
```

Finally, we want to display some "hello world" to see that everything works as expected. React works using a model of "components", each of which is eventually used as a higher-level HTML elements. In complicated examples, this lets a UI team define high-level elements like "blog-item" or "username". However, we are not going to have many levels of items, since our UI is pretty simple – as is our application.

Using React: Code

```
<script type="text/babel">
const element = (
  <h1>
    Hello
  </h1>
);
ReactDOM.render(
  element,
  document.getElementById('content')
);
</script>
```

We have seen that some pages we will need to deliver are generated JSON – different, possibly, each time we run the code. Now we have a static page that needs to be delivered "as is". Real applications, of course, will have many assets – images, CSS files, JavaScript files and more.

The traditional solution is to let the WSGI container run the application producing the JSON, and put another web server in front of it as a reverse proxy. This means we have two things that can go wrong, instead of just one.

Static and Dynamic

- HTML, JavaScript are "static"
- Other URLs are dynamic
- Proxy in front of WSGI

- Custom Twisted plugin

Twisted web, however, is a fully featured web server. There is no reason not to trust it with the static files. We *can* still run a reverse proxy in front of it, and make sure static files are cached, but for our current scaling needs, this is an overkill.

We will write a custom Twisted plugin to run our blog WSGI application.

Static Files: Imports

```
import os

from zope import interface

from twisted.python import (usage, reflect,
                            threadpool, filepath)
from twisted import plugin
from twisted.application import (service,
                                 strports,
                                 internet)
from twisted.web import (wsgi, server,
                         static, resource)
from twisted.internet import reactor

import blog.app
```

As usual, the import list is not very interesting. Of interest, however, is to note we are importing the WSGI support as well as the static file support – there will be a single "source of truth" as to the interaction between the two.

Static Files: Resource

```
class DelegatingResource(resource.Resource):

    def __init__(self, wsgi_resource):
        resource.Resource.__init__(self)
        self.wsgi_resource = wsgi_resource

    def getChild(self, name, request):
        print("Got getChild", name, request)
        request.prepath = []
        request.postpath.insert(0, name)
        return self.wsgi_resource
```

The delegating resource is designed to be a root resource. The root resource is special – it will never be rendered, only queried for children. The reason is that when a URL ends with "/", Twisted will treat it as a request for an empty child – otherwise there is no way to distinguish between foo and foo/.

Therefore, even the root URL is a child of the root resource. The delegating resource will defer to the getChild of the wrapped resource – after pretending that it has never seen the request by unrolling one path step. It is a bit of dirty trick, but it does allow us to serve the WSGI application, as well as static resources, flexibly.

Static Files: Pool

```
def getPool(reactor):
    pool = threadpool.ThreadPool()
    reactor.callWhenRunning(pool.start)
    reactor.addSystemEventTrigger('after',
                                  'shutdown',
                                  pool.stop)
    return pool
```

Building a thread pool in Twisted is, somewhat needlessly, complex. However, building it ourselves does mean we get to tune its performance, if and when we need to, by controlling its parameters. This is true for every decent WSGI container – however, Twisted does have the distinction that the configuration is done in Python code. For example, we are free to look at environment variables, or system parameters, before setting values.

Static Files: Root

```
def getRoot(pool):
    application = blog.app.app
    wsgi_resource = wsgi.WSGIResource(reactor,
                                      pool,
                                      application)
    root = DelegatingResource(wsgi_resource)
    static_resource = static.File('index.html')
    root.putChild(b'', static_resource)
    static_resource = static.File('static')
    root.putChild(b'static', static_resource)
    return root
```

The root resource is a WSGI resource, wrapped by our delegator. This allows us to add children which will be special. We will only use it for static files, but the possibilities are endless.

Static Files: Service

```
class Options(usage.Options):
    pass

def makeService(options):
    pool = getPool(reactor)
```

```
    root = getRoot(pool)
    site = server.Site(root)
    ret = strports.service('tcp:8080', site)
    return ret
```

Finally, we define the Twisted service. Note that in this case we decided to "hard code" almost all parameters instead of parameterizing. "Hard code", is a misnomer, though. We are writing this in Python, a dynamic software programmming language. Changing this is not hard. If we later on see some parameters we really do want to change more easily – perhaps it needs to be different between environments, we can always change that and add a parameter.

This is one of the dirty secrets between SaaS applications and open source projects.

Static Files: Running

```
$ python -m twisted blog
```

Since we have "hard coded" all parameters, running this becomes a simple matter of calling the plugin.

Hands On: Custom Plugin

:50-65
15 minutes

- Get it to run
- Finished? Add a favicon. (Hint: `/favicon.ico`)
- Finished? Add posts with requests
- Finished? Add parameter to control the port

Wonderful! We know how to serve staic files from the same web server that serves the dynamic – this is important, by the way, for the web security model "same origin" policy – and we know how to write basic React, let's put the two together and write a real React class that does something useful.

We will write a React class that will manage our "Posts" element. JavaScript's "this" is similar to Python's "self", but with approximately 100Therefore, there are some cases where "this" gets captured by the wrong class. In order to get around it, we explicitly bind the methods we want to pass around as callbacks.

Listing Posts: Constructor

```
class Posts extends React.Component {
  constructor(props) {
    super(props);
    this.state = { posts: [] };
    this.handleSubmit =
```

```
      this.handleSubmit.bind(this)
    this.handleInputChanged =
      this.handleInputChanged.bind(this)
  }
```

When the component is rendered, "mounted" in React paralance, this method is called. In it, we use fetch to bring the data. Fethc is the modern JavaScript replacement for XHR, which gave AJAX its "X" part in the name. The API is much nicer, and no third party packages are needed for good HTTP clients!

Listing Posts: Data fetch

```
  componentDidMount() {
    fetch("/posts").then(
      response => response.json()).then(
      response => this.setState({posts: response}))
  }
```

Each React component is supposed to render one DOM element. The standard way to draw more than one is to render extra div and span outer elements. This is what everybody does everywhere in the web, so that's OK. Note that we do not need to escape explicitly – React will do the right thing for us.

Listing Posts: Rendering

```
  render() {
    return <div>
           <ul>
           {this.state.posts.map(post =>
              <li>{post[0]}: {post[1]}</li>
            )}
           </ul>
           {this.getForm()}
           </div>
  }
```

The form render is the subtlest part. Note that we can encode our callbacks directly into the DOM – React will set up the right references.

Adding Post: Form

```
  getForm() {
    return <form onSubmit={this.handleSubmit}>
      <p><label>Post:</label>
      <input type="text" name="post"
          onChange={this.handleInputChanged}/></p>
       <p><label>Author:</label>
       <input type="text" name="author"
```

```
          onChange={this.handleInputChanged}/></p>
        <p><input type="submit" value="Post"/></p>
        </form>
}
```

Now we get to the input handling routines. The "right way" is to keep track of the inputs at all time, and use them appropriately. For this, we need some instance variables, and judicious callbacks.

Adding Post: Manage Input

```
handleInputChanged(event) {
  this.setState({[event.target.name]:
                   event.target.value});
  event.preventDefault();
}
```

Finally, we send the data, JSON formatted. Note that because we are using custom JSON formatting, we are resistant to CSRF. This allows us to skip CSRF protection.

Adding Post: Sending Data

```
handleSubmit(event) {
  fetch("/posts",
  {
    headers: {"Content-Type":
               "application/json"},
    method: "POST",
    body: JSON.stringify(
            {author: this.state.author,
             content: this.state.post})
```

Finally, since by definition adding a post changes the state of posts, we also want to re-retrieve the list of posts. We could have, of course, modified the local data ourselves – but there are many reasons to use this to synchronize with the server.

Adding Post: Refreshing

```
    }).then(fetch("/posts").then(
          response => response.json()).then(
          response =>
            this.setState({posts: response})));
  event.preventDefault();
}
```

Indeed, sometimes others will post – and we want to see their posts. Let's have a button that refreshes without posting.

Hands On: Refreshing Posts

:70-90 (20m)

- Get it to run as is
- Add a button to refresh posts
- Finished? Add a loop to refresh posts every 5 seconds
- Finished? Allow turning the loop off and on
- Finished? Allow setting the interval time

Enjoy the break at 10:30, see you back at 10:50

Welcome back

:110

Containers are supposed to be stateless. Most infrastructure around managing them assumes it implicitly. However, some applications – like our modest shared blog – need to keep persistent state. We cannot just save it in any file in the container. Stopping and removing a container is likely to happen.

We can keep it on a "mounted volume", or in a network service. Many production solutions involve network services, but for simplicity, we will mount volumes. This mean that a part of the container's file namespace will point to a part of the host's file namespace. Files we write there will not be part of the container, and will not get removed with it.

Containers and state

- Not inside
- Volumes
- Network

We are going to be using the SQLite database – conveniently for us, SQLite uses a local file as its storage medium. This simplifies deployment for us, since we do not need to worry about an extra network server. Note that this does put a damper on our scaling out hopes.

However, note that we do *not* explicitly import the sqlite library. Instead, we assume we are given a SQLalchemy-compatible URL, and that it will be reasonable. This allows the final choice to switch database to be made without modifying the code. This is in line with a general principle of using containers, of giving as much power as reasonable to the deployer.

Backing Database: Imports

```
import contextlib
import os

import sqlalchemy
from sqlalchemy import sql
```

Next, we define a SQLaclchemy "Metadata" object. This object will hold the information about our table structure. It will be useful, among other things, to initialize the database.

Backing Database: Configuration

```
metadata = sqlalchemy.MetaData()

create_all = metadata.create_all
```

We also need to decide how to know which database to access. The "12-factor" way is to use an environment variables. There are quite a few advantages to the technique: it allows us to configure it at the container level, for example.

Backing Database: Location

```
def get_engine():
    return sqlalchemy.create_engine(os.environ['BLOG_DATABASE'])
```

Next we need to define a SQL schema. Since out data model is stupidly naive, it ends up being one simple table with few constraints. In real life, of course, this part can easily be multiple hundreds of lines.

Backing Database: Schema

```
posts = sqlalchemy.Table('posts', metadata,
            sqlalchemy.Column('content', sqlalchemy.String),
)
```

We write a retrieval method, to abstract the database logic from the rest of the web application. This is particularly useful if the data model is more complicated, since understanding how to map with the underlying data belongs here.

Backing Database: Retrieval

```
def get_posts(engine):
    with contextlib.closing(engine.connect()) as conn:
        return list(conn.execute(sql.select([posts])))
```

For similar reasons, we also write abstractions for storing the data. Some applications write a real "Store" object, which manages a few related tables and presents a simple API. Our case is simple enough to be solvable with a couple of functions.

Backing Database: Storing

```
def add_post(engine, content):
    with contextlib.closing(engine.connect()) as conn:
        conn.execute(posts.insert().values(content=content))
```

We need to decide how to know where the database is. We will choose to do it with environment variable. WSGI applications cannot take command-line arguments. In our specific set-up, we could wire the command-line argument to the tap plugin, and somehow poke it into the WSGI application. However, this would tie the application to the Twisted WSGI container, which is not a good idea.

We put the engine in the settings for the application. This allows any handler function to get it, which means there is only one place where the decision is being made – and only one place we need to change if we want to do it some other way. Again, this is typical.

Backing Database: App Changes: Configuration

```
engine = storage.get_engine()
cfg.add_settings(dict(engine=engine))
```

Now, the retrieval function needs to get the engine from the settings, use the function from storage, and finally, format as JSON.

Backing Database: App Changes: Retrieval

```
engine = request.registry.settings['engine']
posts = [(content, 'anonymous')
         for content, in storage.get_posts(engine)]
```

The same ideas are true, with necessary changes, for the handler that stores a new post.

Backing Database: App Changes: Storage

```
engine = request.registry.settings['engine']
content = request.json_body['content']
storage.add_post(engine, content)
```

Finally, we need to initialize the database with the right schema. In production, this is often done with migration frameworks, unifying the concepts of "upgrade" and "initial deployment". This makes sense, but for our purposes, we will just write a little ad-hoc Python expression to do it.

Backing Database: Initialization

```
$ BLOG_DATABASE=sqlite:///blog.db python -c \
  'from blog.storage import *;create_all(get_engine())'
```

Backing Database: Hands-On

:125-:140 (15m)

- Get it to run
- Add author
- Finished? Normalize author to different table
- Finished? Add auto-calculated date

We have written a full-stack application! It might be simple, but it has all the parts of more complicated applications. If this was a regular tutorial about web applications, we would start adding features. But in the A-to-Z tutorial, we are not satisfied with just code – we want to actually have it deployable.

There are many ways to ship and run Python applications. However, starting with a greenfield, usually people will choose some variant on Docker. Docker is nice in that it solves both the problem of packaging and of runtime isolation. Especially for this tutorial, this is useful!

We will start with a "naive" Docker build. This Docker build will work, and is fairly common in the wild – but is far from best practice.

Naive Docker Build

```
FROM python:latest

ENV BLOG_DATABASE sqlite:////mnt/db/blog.db

RUN pip install pyramid sqlalchemy twisted
RUN mkdir /src/
COPY twisted /src/twisted/
COPY blog /src/blog/
COPY index.html /src/
COPY static /src/static

WORKDIR /src

ENTRYPOINT ["/usr/local/bin/python", "-m", "twisted", "blog"]
```

In order to build this, we run the "docker build" command. The last argument is the current directory, which is where it expects to find files referenced in COPY commands. The tag is what will allow us to run it later.

Naive Docker Build: Building

```
$ docker build -f naive.docker \
               -t naive-blog .
```

There are many ways to run Docker. In fact, in production we rarely run it directly: we describe how to run to an orchestration framework. Running it locally also can be done with tools like docker compose or minikube. However, we can also run it directly with "docker run". We want to indicate the port to be forwarded, so we can test it with our browser.

Naive Docker Build: Running

```
$ docker run --rm -it \
             -p 8080:8080 \
             -v $(pwd):/mnt/db \
             naive-blog
```

For a good Docker build to work, we really should start by treating our code with more respect. Instead of throwing files around like animals, we should use modern Python packaging. That starts by writing a setup.py file.

Multistage Docker Build: Truly Minimal Setup

```
import setuptools
setuptools.setup(
packages=setuptools.find_packages() +
         ['twisted.plugins'],
install_requires=['Twisted',
                  'pyramid',
                  'sqlalchemy'],
)
```

We use the python:3 library docker image as our base. It is a premade Debian-based image, with a custom Python 3 interpreter installed.

Multistage Docker Build: Base

```
FROM python:3
```

Luckily, in Python 3, virtual environments are built-in. We create a virtual environment for all of our things to go in

Multistage Docker Build: Virtual environment

```
RUN python -m venv /appenv
```

We copy the files into the image. Note that we do several COPY lines. Some people do not like create extra layers and use all kinds of dirty tricks to avoid those. We will use a clean nice hack to avoid those!

Multistage Docker Build: Copying

```
RUN mkdir /src/
COPY twisted /src/twisted/
COPY blog /src/blog/
COPY setup.py /src/
```

pip can install directories that contain setup.py. This is no different than installing from source distributions, which are just zipped or tarred directories. This will also install all dependencies.

Multistage Docker Build: Installing

```
RUN /appenv/bin/pip install /src/
```

Next, we copy the static files that we need to serve into our directory. Note that we made sure all interesting files are under /appenv.

Multistage Docker Build: Static files

```
RUN mkdir /appenv/website
COPY index.html /appenv/website/
COPY static /appenv/website/static
```

The second FROM line identifies this as a multistage Docker build. Note that everything done until now is going to be thrown away at the end of the build. Our new base, "slim", does not contain build utlities, and is much smaller.

Multistage Docker Build: Production base

```
FROM python:slim-stretch
```

We copy the entire directory of /appenv. This is where both our code resides, its dependences and the static files for the web server.

Multistage Docker Build: Copy build products

```
COPY --from=0 /appenv /appenv/
```

Since the blog plugin relies on an environment variable and the current working directory, we need to set those parameters in the Docker setup. We could, of course, have fed the parameters command-line. However, this is not a realistic possibility in many cases, so it was good to be able to show how to handle this real-life situation.

Multistage Docker Build: Parameters

```
ENV BLOG_DATABASE sqlite:////mnt/db/blog.db
WORKDIR /appenv/website/
EXPOSE 8080
```

Finally, we set up the entrypoint to run our plugin automatically in start-up. This is why Docker is sold as a solution for both packaging and deployment.

Multistage Docker Build: Entrypoint

```
ENTRYPOINT ["/appenv/bin/python", "-m", "twisted", "blog"]
```

Let's get some real experience using Docker.

Hands On: Running Locally

:155-165 (10m)

- Run and visit with your browser
- Finished? Remove the `-p` and use `docker inspect`
- Finished? Use the `webbrowser` and `docker inspect` to automatically open the browser.

We ran it, and it looked like it worked. However, if we have to run manual checks like that every time we change something, it is going to be a long time – and we will likely make mistakes or forget. Of course, it is good to have unit tests for the code – but there are many resources on testing your Python and your JavaScript separately. It is, however, important – and less trivial – to run end-to-end tests.

For that, we use the Selenium framework. Packaging all the bits that make Selenium work is tricky, and we would like to only do that once. We will turn to the same packaging tool that we used to package our application – Docker.

We start with the same base, python:3, that served us well before. Note that because this is not going to run in "production", we are not as interested in a multi-stage build that produces a minimal image.

Selenium: Docker: Base

```
FROM python:3
```

We install the Selenium Python library, which we will use to write tests. We also install the requests library, which will allow us to communicate with the API directly.

Selenium: Docker: Python packages

```
RUN pip install \
    selenium \
    requests
```

Selenium does not come with its own browsers. Writing a browser, especially a modern one, is hard. It can use most browsers, but in our example we will use Firefox. However, in order to make Firefox "drivable", so that Selenium can drive it, we need to install the "Geckodriver"

Selenium: Docker: Gecko driver

```
RUN export BASE=https://github.com/mozilla/geckodriver && \
    export LOCATION=releases/download/v0.19.1  && \
    export NAME=geckodriver-v0.19.1-linux64.tar.gz && \
    curl -L $BASE/$LOCATION/$NAME \
    | (cd /usr/local/bin; tar xzf -)
```

Next we install Firefox. Geckodriver expects Firefox to be in the path. While it is possible to configure it with a specific location, it is much easier to just put Firefox in the path, with some symbolic linking.

Selenium: Docker: Firefox

```
RUN mkdir /firefox
RUN export CDN=https://download-installer.cdn.mozilla.net/ && \
    export RELEASE=pub/firefox/releases/58.0.2/ && \
    export DL=linux-x86_64/en-US/firefox-58.0.2.tar.bz2 && \
    curl $CDN/$RELEASE/$DL \
    | (cd /firefox; tar xjf -)
RUN ln -s /firefox/firefox/firefox \
    /usr/local/bin/firefox
```

Next we install Debian packages from upstream. The highlights include Xvfb, which is a "headless X" server, and enough packages to add a new source for installing packages.

Selenium: Docker: Debian packages

```
RUN apt-get update
RUN apt-get install -y \
    libgtk-3-0 libdbus-glib-1-2 libX11-xcb1 \
    Xvfb apt-transport-https ca-certificates \
    curl gnupg2 software-properties-common
```

We add a new source in order to be able to install the Docker command-line inside of our container. Since we want our test to be self-contained, we would like it to manage the setup and teardown of the tested server.

Selenium: Docker: Docker

```
RUN . /etc/os-release && \
    export DOWNLOAD=https://download.docker.com/ && \
    curl -fsSL $DOWNLOAD/linux/$ID/gpg \
    | apt-key add -
RUN . /etc/os-release && \
    export DOWNLOAD=https://download.docker.com/ && \
    add-apt-repository \
    "deb [arch=amd64] $DOWNLOAD/linux/$ID \
```

```
    $(lsb_release -cs) \
    stable"
RUN apt-get update
RUN apt-get install -y docker-ce
```

We also need to copy the test code. In real life, these might be written as unit tests for a test runner to run, with proper consideration for optimizing the number of start-ups and shutdowns. However, for simplicity, we will do everything in a plain Python script.

Selenium: Docker: Test

```
COPY run-test.py /
```

Lastly, we set the testing script as our entrypoint. This means we do not need to do anything other than run the container, and it will start a test server, run tests against it, and shut it down.

Selenium: Docker: Entrypoint

```
ENTRYPOINT ["python", "run-test.py"]
```

The import lists is again of limited interest. Note, however, that we are importing both Selenium and Requests for this test. We will be using the robotic interface to post, and the human interface to read.

Selenium: Tester: Imports

```
import json
import os
import subprocess as sp
import time

import requests
from selenium import webdriver
```

We start by running Xvfb. This is a headless X server. UI applications will connect to it and ask it to display stuff – and it will dutifully display it to a frame buffer (fb) it keeps internally. Note that it is possible to take screenshots from this frame buffer, although we will not do it here.

Selenium: Tester: Headless X

```
subprocess.Popen(["Xvfb", ":99"])
os.environ['DISPLAY'] = ':99'
```

We use Docker, just like we did from the command-line, to run the test web server. Note that we name it. This means if there is already one running, this will fail. There are advantages and disadvantages for this approach.

Selenium: Tester: Running Tested Server

```
sp.check_call(["docker", "run", "-d",
               "--name", "blog", "multistage-blog"])
sp.check_call(["docker", "exec", "blog",
               "mkdir", "/mnt/db"])
sp.check_call(["docker", "exec", "blog",
               "/appenv/bin/python",
               "-c", "from blog import storage;"
                     "storage.create_all("
                     "storage.get_engine())"])
```

We use Docker inspect, and some JSON manipulation, to read the address of the container. This is instead of mapping the port, and does require the port to be EXPOSEd.

Selenium: Tester: Tested Server Address

```
details = sp.check_output(["docker", "inspect", "blog"])
t = json.loads(details)
address = t[0]["NetworkSettings"]["IPAddress"]
```

This innocent line probably deserves a blog post, or at least a paragraph. "How long to wait" is the bane of any testing facility, but it is doubly subtle and painful question for end-to-end testing. We could have instead done time.sleep in a loop, and checked for access. This would have made it slightly less painful to make maximum wait-time longer. Ultimately, though, there has to be a limit, so this is as good an example as any.

Selenium: Tester: Waiting

```
time.sleep(10)
```

Next, we put data in via the RESTful API. This is often convenient, especially if we needed to seed in a large amount of data – for example to check paging in an application that does paging. Here we do it because showing how to press a button in Selenium would probably be beyond the time constraints.

Selenium: Tester: Seeding

```
requests.post(f'http://{address}:8080/posts',
              json=dict(author='foo',
                        content='hello there friend'))
```

After all the preparation for using Selenium, actually using it is almost too easy. We just navigate to a URL! Of course, more complicated behaviors are possible.

Selenium: Tester: Connecting

```
d = webdriver.Firefox()
d.get(f"http://{address}:8080/")
content = d.find_element_by_tag_name("ul")
```

There are all kinds of ways to check the data we got is what we expected. Here we'll be a bit uncivilized, and grab the innerHTML property directly, to compare it against a string.

Selenium: Tester: Checking

```
innerHTML = content.get_attribute('innerHTML')
if 'hello there friend' not in innerHTML:
    raise ValueError('no friend', innerHTML)
```

Finally we clean the web container up. Nothing is left, except for the success or failure of the test.

Selenium: Tester: Cleaning

```
sp.check_call(["docker",
               "rm", "-f", "blog"])
```

We note that our container only ran a docker client – not a server. In general, this is the right behavior. It is better to run a client inside the container that connects to the real servers. Thus, the container it runs is a sister, not a daughter.

Aside: Docker in Docker

Sisters vs. Daughters

In order to run the testing container, we need to give it the ability to connect to the Docker server. Docker uses a unix-domain socket and unix permissions. Since we are root inside the container, permissions are not an issue. We can mount in as a volume the unix domain socket, and voila.

Selenium: Running test

```
$ docker run --rm \
  -v /var/run/docker.sock:/var/run/docker.sock \
  selenium
```

Let's make sure our test has a negative as well as positive part. Check that our post is actually what adds the data to the page.

Hands On: Improve Test

:180-:190 (10m)

- Get it to run
- Add a test that checks there is no "lalala", then posts "lalala" and checks that it is there now.
- Finished? Try to post from selenium as well.

Whew. So we have an application with a backing store, that has a custom JavaScript front-end, with a good packaging and deployment story, and a good testing story. This is good, because we are almost out of time. Are we done? No.

For a production-grade application, there is still a lot left we did not have time to cover.

Web application, A to..what?

What's missing?

- CSS – make it look nice (CSS with React is a bit different)
- Webpack – translate our React code for nicer sourcemaps and more
- Continuous integration: Running Selenium on your laptop is great, now run it in CircleCI
- Continuous deployment: Integration with ECS/K8s
- TLS – certificate management and rotation
- Monitoring – Pinging, logging, metrics, stack traces, alerting
- Redundancy
- Data migration
- Backup

Maybe this is why most applications take more than three hours to write. We did have a chance to go over a lot of technologies that are useful for web applications.

Summary

- SQLalchemy
- Pyramid
- React
- Twisted
- Docker
- Selenium

Questions