Web Application A to Z

Moshe Zadka – https://cobordism.com

PyBay 2018

Welcome to the Web applications, A-to-Z tutorial. A-to-Z means we will take care of all steps needed to write web applications.

Having an API and a continously updating UI is nowadays 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

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 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

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():
    db = os.environ['BLOG_DATABASE']
    return sqlalchemy.create_engine(db)
```

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),
)
```

In order to simplify our lives, we define a simple helper method. It automatically closes a connection at the end of the block. This will mostly make our code fit better on the slides.

Backing Database: Context

```
def context_connect(engine):
    return contextlib.closing(engine.connect())
```

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 context_connect(engine) as conn:
        select = sql.select([posts])
        cursor = conn.execute(select)
        return list(cursor)
```

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 context_connect(engine) as conn:
        insert = posts.insert()
        ins_content = insert.values(
                            content=content)
        conn.execute(ins_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

```
with config.Configurator() as cfg:
    engine = storage.get_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

So far, we have been writing code like it is 2005 – directly in the HTML. We relied on Babel transpiling our JSX to JS on browser load time. We sent our raw sources over the wire. This is not how modern JavaScript development is done. Modern JavaScript is written to be compiled to browser JavaScript. The tool that does this compilation? Webpack.

Webpack

Real JavaScript developers minify and transpile and...

For us to properly use webpack, our JavaScript needs to be in a separate file. Let's break out our code. Notice that now we have to explicitly require the libraries we use – react and reactdom.

Webpack: Breaking Out JS

```
const ReactDOM = require('react-dom');
const React = require('react');

class Posts extends React.Component {
  constructor(props) {
```

Now we need to install and run webpack. There are several ways to do that, but we will choose Yarn. Yarn is... a little bit like pip for JavaScript.

Webpack: Yarn

Yarn manages JavaScript environments

We define our "desired" depdencies in a JSON file. Yarn, when installing it, will document the specific dependencies it chose in a lock file, in order to allow us to reproduce the environment.

Webpack: Yarn: Configuration

```
{
  "devDependencies": {
    "webpack": "^4.12.0",
    "webpack-cli": "^3.0.8"
  },
  "dependencies": {
    "@material-ui/core": "^1.3.0",
    "babel-core": "^6.26.3",
    "babel-loader": "^7.1.4",
    "babel-preset-react": "^6.24.1",
    "react": "^16.4.1",
    "react-dom": "^16.4.1"
  },
  "private": true
}
```

Note that yarn uses a "virtual environment" by default. It will install all the packages in the current working directory. Common practice is to just check in the node-modules directory to source control, and accept that it will change drastically if we upgrade. There are those who oppose it, and they better all hope none of their dependencies are leftpad.

Webpack: Yarn: Install

```
$ yarn install
```

Webpack needs its own, separate, configuration. Most of it is pretty straightforward.

Webpack: Configuration

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: './src/index.jsx',
  output: {
    path: path.resolve('dist'),
    filename: 'index_bundle.js'
  },
```

```
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: "babel-loader",
      }
    ]
  }
}
```

Webpack, in turn, calls babel. We need to configure that too.

Webpack: Babel configuration

```
{
  "presets": ["react"]
}
```

Finally, webpack is not on our path – with yarn, there is no concept of "activating" a virtual environment. The right way to run commands is to use "yarn command". We run webpack with yarn.

Webpack: Running

```
$ yarn webpack
```

Finally, we play some games with symbolic links in order to be able to properly configure everything.

Webpack: Layout

```
$ ls -l static/dist
... static/dist -> ../dist/
```

Since webpack produces a JavaScript file, we need some HTML file to load it into a browser. This is the smallest possible HTML file.

Webpack: Bootstrap

```
<!DOCTYPE html><html>
<head><meta charset="utf-8"/></head>
<body><div id="content"></div>
<script
src="/static/dist/index_bundle.js"
></script>
</body></html>
```

Webpack: Hands-On

:160-:180

- Get it to run
- Finished? Generate source maps.

Let's get in and try to run it. If you are done, get webpack to generate source maps.

Styling

Look and feel

Modern web applications look good, as well. They are styled to look and feel almost like desktop applications: colors, 3D animations, and more.

CSS

- Classes
- Selectors
- Styles

The main tool used for that on the web is CSS: Cascading style sheets. They use classes to mark elements, selector expressions to pick elements, based on class and more, and actual styles to decide on font, color, placements and more.

CSS and JavaScript/React

- CSS styles
- JavaScript makes components
- Interactions can be surprising...
- ...and hurt modularity

CSS styles are "global". But a typical React application builds components into an application. This means that figuring out, say, what kind of things a selector statement will hit might be non-trivial. This can easily lead to problems like nobody ever deleting selector statements, or a single element that looks out of place.

Material UI 1.0

- JavaScript +
- CSS +
- Design

Material UI is a set of React components that auto-generate CSS so that we do not have to think of it. It supports its own notion of theming, which is higher-level that raw CSS. It is opinionated: it has opinions about correct UI elements, for example.

Material UI: Imports

```
const React = require('react');
import Button from
       '@material-ui/core/Button';
import TextField from
       '@material-ui/core/TextField';
import List from
       '@material-ui/core/List';
import ListItem from
       '@material-ui/core/ListItem';
import ListItemText from
       '@material-ui/core/ListItemText';

class Posts extends React.Component {
  constructor(props) {
```

In order to use Material UI elements, we need to import them.

Material UI: Form

```
  getForm() {
    return <form onSubmit={this.handleSubmit}>
     <p><TextField inputProps={{id: "post"}}
          label="Post"
          onChange={this.handleInputChanged}/></p>
     <p><TextField inputProps={{id: "author"}}
          label="Author"
          onChange={this.handleInputChanged}/></p>
     <p><Button variant="contained"
          onClick={this.handleSubmit}>
          Post</Button></p>
       </form>
  }
```

We use higher-level "TextField" and "Button" in order to make our design look modern. Those contain a lot of UI-logic: the label will become smaller when we are typing in the field. the button will animate when pressed;

Material UI: Posts

```
render() {
  return <div>
         <List>
         {this.state.posts.map(post =>
            <ListItem><ListItemText
                        primary={post[0]+
                                 ": " +
                                 post[1]}
            /></ListItem>
          )}
         </List>
         {this.getForm()}
         </div>
}
```

Let's spend the rest of the time we have adding material UI elements.

Hands On: Add elements

:200-:215 (15m)

- Get it to run
- Add a title
- Finished? Add a bar
- Finished? Play with alignments
- Finished? Play with colors

Web application, A to..what?

What's missing?

- Packaging (with Docker)
- Testing (with Selenium)
- Continuous integration (CircleCI, Jenkins, etc.)
- Deployment (K8s)
- Continuous Deployment (CI + 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
- Twisted
- React
- Webpack
- Material UI

We covered a few technologies here – SQLalchemy for interacting with databases, Pyramid for writing backend web applications, Twisted for serving web applications, React for the front-end, webpack to bundle the front-end, and Material UI as a set of React component which make it easier to style.

Questions

Any questions?