GET OVER THE BOUNDARIES BETWEEN CLIENT AND SERVER IN WEB APP DEVELOPMENT

ALBERTO BERTI

ALBERTO@ARSTECNICA.IT

EuroPython 2017 in Rimini
TABLE OF CONTENTS

- What I mean with **web app**
- How a web app is today built using Python tools?
- Dealing with JavaScript is inevitable
- Back to Python
- Welcome Raccoon
- The idea
- reduce mind context-switching burden while coding both Python and JS code
- Anatomy of a Raccoon user session
- An example of a raccoon application **context**
- Data synchronization originates on the server
- There's **no routing**
- an example
- Scaling
- Finally
WHAT I MEAN WITH WEB APP

- an interface to relational data
- replacement for desktop database applications
- data intensive with features like
  - filtering
  - reordering
- optimized to show many records
- complex forms
- master-detail
• often heavily customized to meet customer's needs
• narrower user base than public web publishing
• often installed on premise or cloud distributed for intranet use
• usually they are called **SPA**, or Single Page Applications
HOW A WEB APP IS TODAY BUILT USING PYTHON TOOLS?

- develop a database structure that best helps persisting domain data
- pick your server framework
- optionally develop an ORM to access the data
- expose the data using REST or some other solution
BUT THEN...

- pick a JavaScript application framework
- develop the application logic and user interaction
DEALING WITH JAVASCRIPT IS INEVITABLE

- even if it has many inconsistencies
- every now and then a new trendy framework appears and reinvents the wheel, in a cooler way
- it has a broader developer base than Python
- often the libraries and packages have poor quality
BUT ES6 IS BETTER!

- it is way better than previous iterations
  - Classes
  - Promises
  - iterators
  - generators
  - **Map** and **Set** implemented natively
MY REACTION TO WEAKMAP AND WEAKSET

- finally! Fantastic!
- What I was waiting for!
BUT THEN I DISCOVERED THAT

• it isn't possible to known which elements or keys (or values) the object contains
• it is equally impossible to iterate over any of them
• it is only possible to check if a given element or key is contained
WHAT?
JS DEVELOPERS SEEMS HAPPY WITH IT

Some have even found an use for them:
From Exploring ES6:
It is impossible to inspect the innards of a WeakMap, to get an overview of them.
[...]
These restrictions enable a security property. Quoting Mark Miller: “The mapping from weakmap/key pair value can only be observed or affected by someone who has both the weakmap and the key. [...]”
They call it a security property...
Ride the TypeScript hype!
This seems fun to me:
```typescript
class Animal {}  
class Bird extends Animal {}  

const foo: Array<Bird> = [];  

foo.push(new Animal()); // ok in typescript  
```
from reddit's blog entry: Why We Chose Typescript of just few days ago
WHAT?
BACK TO PYTHON
THE ROLE OF PYTHON IN MODERN WEB APPS

- the role of the Python server has become that of a data hub
- no application-level development, it moved to the JS app... sad
- usually the fun ends with the completion of the database structure - ORM part
HOW WEB FRAMEWORKS DO THEIR JOB?
Most major Python web frameworks (used to build the server part of our applications) are modeled around HTTP with its request-response model
• handlers attach to (choose your level of complexity) resource paths
• a client makes a request
• the request is the main context object often with the help of session data.
• objects are created, data is retrieved, a response object with numeric result codes and your content is created
• the response is serialized, some state is saved to the session
• the objects are destroyed
When do we really need REST APIs we think they are really needed when your application has to interface with other services and your service provides an API to its users.
IS IT POSSIBLE TO IMAGINE A DIFFERENT MODEL?

Desktop applications using PyQt or PyGTK are driven completely by Python objects, interfacing with the toolkit's ui elements.
WELCOME RACCOON

In late 2016 we decided to replace an old application named Safety with a new application and develop a new framework along with it to try bring back the fun when developing a web app with Python.

Safety is an application to assess and report working environment health risks.

Goodbye Safety
Welcome Raccoon and Ytefas (or ʎʇǝɟɐ made right)
THE IDEA

- use an asynchronous system to ease maintaining the state in the server
- do the same on the client for the state that drives the UI
- connect these two elements with a modern RPC and event system
- bring some application-level logic back to Python
ASYNC FROM THE GROUND UP

- **PostgreSQL** and **PatchDB** to define and maintain the database
- **AsyncPG** and **SQLAlchemy** for data access
- Crossbar's **WAMP** router for RPC and events
- aiohttp for **HTTP**
DATA ACCESS LAYER

- SQLAlchemy's ORM cannot be used in an async environment
- ORM is used anyway in tests and to carry field-level metadata
- AsyncPG is fast but has no symbolic query api
- we plugged SQLAlchemy's symbolic query rendering with AsyncPG
RPC

- **Crossbar** has a lot of **features** and supports clients written in any of the major languages used today
- built with **Twisted**, its Python client library supports both Twisted and **asyncio** applications
- it's the primary implementation of a **WAMP** protocol router
- most of the configuration setup is asynchronous
- uses a dotted string as endpoint/topic address
- error handling
- simple registration/subscription system out of the box
It's based on a **Node** mixin class

- class level definition of **signals** (events), event handlers, and rpc endpoints
- **Node**'s basic API is composed of just four coroutines:
  - node.node_bind(path, node_context=None, parent=None)
  - node.node_add(name, node)
  - node.node_remove(name)
  - node.node_unbind()

and the corresponding signals:

- on_node_bind
- on_node_add
- on_node_unbind
node.node_bind(path, node_context=None, parent=None)
"path" is a dotted string compatible with Crossbar's addresses or a special Path instance. 
"node_context" is an instance of NodeContext which is basically a prototype-like namespace which inherits its members from its parent. Its role is to:
• carry connectivity information and security wrappers
• supplement the role of the request object in other frameworks
Path instances with the help of the node_context are pluggable resolvers
EXAMPLE OF THREE NODES INTERACTION IN PYTHON

```python
@ pytest.mark.asyncio
async def test_node_communication(connection1, connection2):

    import asyncio
    from metapensiero.signal import Signal, handler
    from raccoon.rocky.node import WAMPNode as Node, Path, call

    await when_connected(connection1)
    await when_connected(connection2)

    ev = asyncio.Event()
    rst = Node()

    class Second(Node):
        on_foo = Signal()

        async def call_third(self):
            await self.remote('@third').rpc('hello')

    class Third(Node):
        def __init__(self):
```

---

*Example of three nodes interaction in Python.*

```python
    def __init__(self):
```
22:    self.handler_args = None
23:    self.somenthing = None
24:
25:    @handler('@first.second')
26:    def do_on_second_foo(self, *args):
27:        self.handler_args = args
28:        ev.set()
29:
30:    @call
31:    async def rpc(self, something):
32:        self.something = something
33: `base` = Path('test')
34: `second` = Second()
35: `third` = Third()
36:
37: `await first.node_bind(base + 'first', connection1.new_context())`
38: `await third.node_bind(base + 'third', connection2.new_context())`
39: `await first.node_add('second', second)`
40:
41: `await second.call_third()`
42: `await second.on_foo.notify('hello handler')`
43: `await ev.wait()`
44:
45: `assert third.something == 'hello' and third.handler_args == ('hello handler',)`
46: `await first.node_unbind()`
47: `await third.node_unbind()`
```python
from __globals__ import expect, it, jest

from raccoon__rocky import (WAMPNode as Node, Path, call,
Signal, handler, reversed_promise,
register_signals)

def test_node_communication():
    ctx1, ctx2 = gen_ctx(), gen_ctx()
    ev = reversed_promise()
    first = Node()

    @register_signals
class Second(Node):
        on_foo = Signal()

    async def call_third(self):
        await self.remote('@third').rpc('hello')

    @register_signals
```

.. AND IN JAVASCRIPT
class Third(Node):
    def __init__(self):
        self.handler_args = None
        self.something = None

@handler('@rst.second')
def do_on_second_foo(self, *args):
    self.handler_args = args
    ev.resolve()

@call
async def rpc(self, something):
    self.something = something
base = Path('test')
second = Second()
third = Third()

await first.node_bind(base + 'first', ctx1)
await third.node_bind(base + 'third', ctx2)
await first.node_add('second', second)

await second.call_third()
await second.on_foo.notify('hello handler')
await ev

expect(third.something).toEqual('hello')
expect(third.handler_args).toEqual(('hello handler',))

await first.node_unbind()
await third.node_unbind()

it('Basic com works', test_node_communication)
REDUCE MIND CONTEXT-SWITCHING BURDEN WHILE CODING BOTH PYTHON AND JS CODE

**Raccoon** is equally available in both Python and JavaScript thanks to **JavaScripthon** that we use together with **BabelJS** and **Webpack**. We use the same abstractions like generators, async/await, decorators using the same syntax and producing code that can be run down to Firefox 49 (no, we do not test on IE).
Service is an aiohttp application. It publishes an entrypoint in WAMP. Usually, a Controller (server side) and a View (client side) are paired together in what's called "a context" and can use relative paths (beginning with '#') to refer to each other resources.
AN EXAMPLE OF A RACCOON APPLICATION

**Context**

- **ExampleController**
  - `DataCursor(name=master)`
  - `DataCursor(name=detail)`
  - `AddAction(cname=detail, name=add)`
- **ExampleView**
  - `DataProxy(cname=master)`
  - `DataProxy(cname=detail)`
  - `AddDetailButton(provider=detail, name=add)`
  - `DataSource(risk.companies)`
  - `DataSource(risk.employees)`

**UI Toolkit**

- The **controller** has relative address `controller`
- The **view** has relative address `view`
- A **cursor** handles data and has notion of a "currentid"
- A **proxy** drives the UI and sends back information about the currentid and if some change is pending (dirty state)
- An action affects the current **context** or can start a new one
- They are all subclasses of **Node**
DATA SYNCHRONIZATION ORIGINATES ON THE SERVER

- the client sends back to the server status information that allow the server side to re-synchronize its "data sources" and send updates to the client.
- every Node is also a "reactive dictionary" (using the metapensiero.reactive package) capable of storing immutable data and automatically notifies interested parties of data changes.
- a change of currentid in the "master" cursor triggers a reload (async) of the "detail" cursors that's tracking master['current_id'] value.
- this way there is now need to setup "data relations" on the toolkit.
THERE'S NO ROUTING

- Raccoon borrows the Intent concept from Android
- any data that can be expressed using a DataSource (SQL for now) is serializable to a Content instance
- a Controller declares conditions that must be fullfilled for it to be elected as a candidate
- the most important is the kind of Operation it can "realize" (view, create, edit, pick...)
Some textual examples:

- the `Desktop` context/view gets executed because is the only one that can do the operation "view" on an "auth.user" content, the user that just logged in

```python
class Desktop(Controller):
    OPERATION = OPERATIONS.VIEW
    CONTENT = {(Content.source == 'auth.users') & (Content.len == 1)}
    CURSORS = {
        'user': 'auth.users',
    }
    VIEW = {
        'type': 'Desktop',
    }
```

```python
class Logout(Action):

    ID = 'logout'
    CATEGORIES = ('session',)
    LABEL = _('Logout')
    HINT = _('Leave this session.')
    ICON = 'sign-out'
```
@call
async def run(self):
    await self.remote('#view').logout()}
SCALING

- our server probably consumes more memory that other frameworks
- deployed using docker containers in a Rancher environment
- can be scaled HAProxy and Rancher's service sidekiks
Finally

- **Raccoon** isn't public yet but it will be when it's in good shape (documentation, pluggability, more tests) and we decided on the license.
- If you are interested in a demo account to try **Ytefas** and play with it or simply want to know more just ask me or drop me a line.

Thank you
Alberto Berti (alberto@arstecnica.it)
Github: https://github.com/azazel75