Lesson: Microservices 101 Part 1/2

March 29, 2016

(My name is Diego Pino Navarro
I work at metro.org)

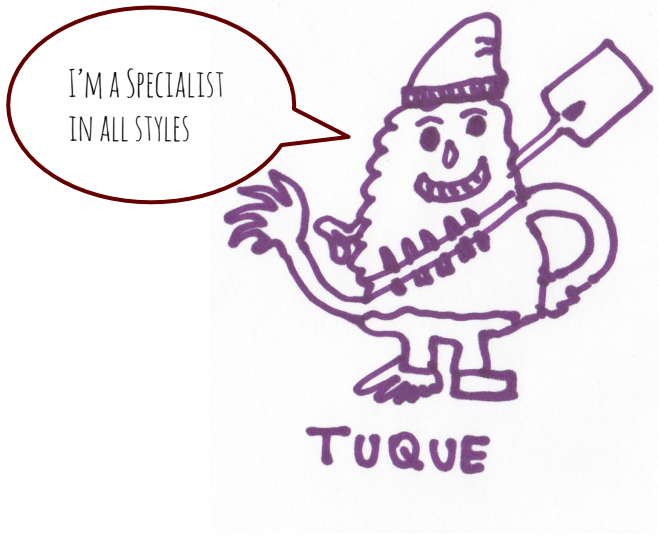
Last session i said Next Session: Let’s write some Camel routes (lier, lier)

What are microservices in CLAW?

A simple idea

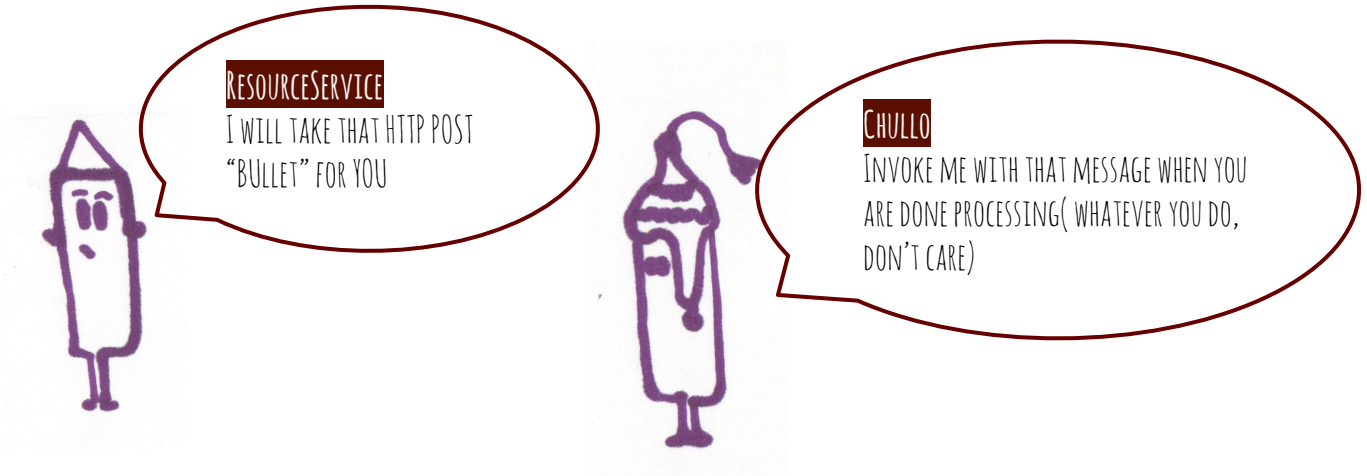
Instead of having a Big chunk of all purpose code

We build slim, lightweight PHP Services that do a particular task, reusing code, classes or full packages



A simple idea

Micro Services: These are not just classes /php code. These are services that run at an HTTP URL/PORT listening and processing what you send them.



Role separation: Chullo, where all started (not a service)

https://github.com/Islandora-CLAW/chullo

Uses *EasyRDF* and *Guzzle* to talk to Fedora 4. Has interfaces.

Provides Classes

Class Namespace (hu!)

Class Name

Islandora\Chullo

- FedoraAPI (full api)
- Chullo (simplified access to api)
- TriplestoreClient (talks to triple store)

Islandora\Chullo\Uuid

- UuidGenerator (UUID V4 and V5 generator)

Before going further: let’s talk about modern PHP (this millennium)

PHP can be a modern OOP language: Namespaces

- Some app defines class ‘User’
- My app defines class ‘User’
- PHP says NO!

Only one ‘User’ class!

Since PHP 5.3 we have Namespaces

- We can give each class a context

We can have multiple ‘User’ classes

http://php.net/manual/en/language.namespaces.importing.php

PHP can be a modern OOP language: Namespaces

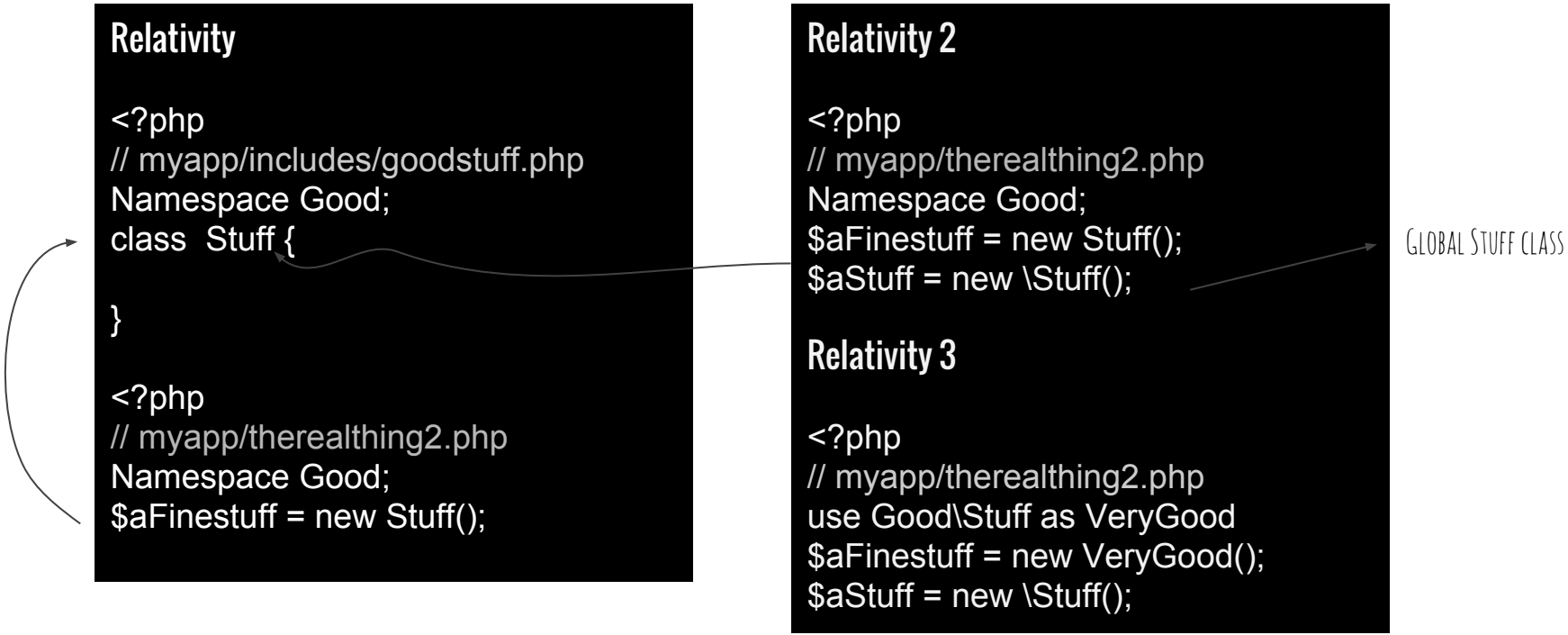
The global Namespace(or no Namespace)

```
<?php
// myapp/includes/stuff.php
class  Stuff {

}

<?php
// myapp/therealthing.php
$somestuff = new Stuff();
```

With Namespace

```
<?php
// myapp/includes/goodstuff.php
Namespace Good;
class  Stuff {

}

<?php
// myapp/therealthing2.php

$thestuff = new Good\Stuff();
```

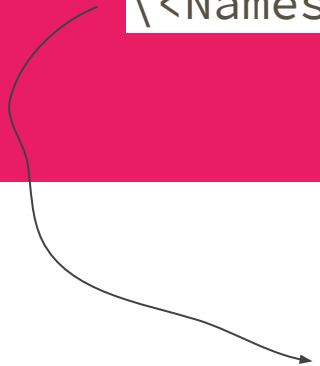
PHP can be a modern OOP language: Namespaces



Namespaces can be large

```
\<NamespaceName>(\<SubNamespaceNames>)*
```

```
Islandora\Chullo\Uuid;
```



Good practice: Make this a Vendor, or unique Personal Namespace!

Namespaces help also organize our Classes

We can replicate a folder structure

This is not JAVA

import Good.*;

(Not possible in PHP)

PHP can be a modern OOP language: Autoloading

```
require_once('myclass.php');
```

= Unpractical for multiple files in different folders that have complex dependencies. But require/include is needed to make PHP aware of your files

¿So, how do we make use of Namespaces and also allow PHP to find files where our classes are?

PHP can be a modern OOP language: Autoloading

PSR-4 +

What is Composer?

“Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.”

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

Key Facts

- It uses PSR-4 style loading
- Manages versions and dependencies for PHP packages
- Uses JSON to define the dependencies
- Downloads them to **// vendor** folder locally
- Creates an **autoload.php** file
- Keeps track of what you are using
- It’s cool and simple (after using it a few times...)

The composer.json

```
{
   "name": "islandora/chullo",
   "description": "A PHP client for interacting with a Fedora 4
server.",
   "type": "library",
   "homepage": "https://github.com/islandora-claw/chullo",
   "support": {
       "issues": "https://github.com/islandora-claw/chullo/issues",
       "irc": "irc://irc.freenode.net/islandora"
   },
   "require": {
       "php": ">=5.5.0",
       "guzzlehttp/guzzle": "^6.1.0",
       "easyrdf/easyrdf": "^0.9.1"
,
       "ml/json-ld": "^1.0.4"   },
   "require-dev": {
       "phpunit/phpunit": "^4.8"
   },
   "license": "GPLv3",
   "authors": [
       {
           "name": "Islandora Foundation",
           "email": "community@islandora.ca",
           "role": "Owner"


       },
       {
           "name": "Daniel Lamb",
           "email": "daniel@discoverygarden.ca",
           "role": "Maintainer"
       },
       {
           "name": "Nick Ruest",
           "email": "ruestn@gmail.com",
           "role": "Maintainer"
       }
   ],
   "autoload": {
       "psr-4": {"Islandora\\Chullo\\": "src/"}
   }
}
```

What in the hell is PSR-4?

PSR-4 by example (http://www.php-fig.org/psr/psr-4/)

Fully qualified Class Name (same as used when creating a new Request Object)	Namespace (prefix)	Base Source Path	Where the Request Class should be present
\Symfony\Core\Request	Symfony\Core	./vendor/Symfony/Core/	./vendor/Symfony/Core/Request.php

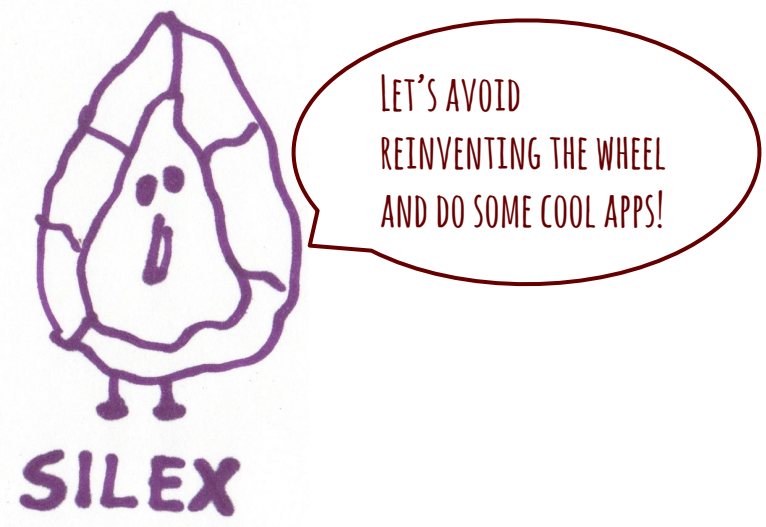
In Claw's Chullo

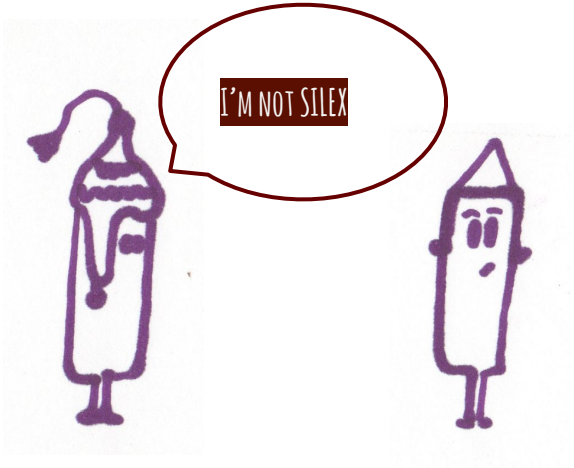
```
"autoload": {
        "psr-4": {"Islandora\\Chullo\\": "src/"}
    }
```

So much new stuff

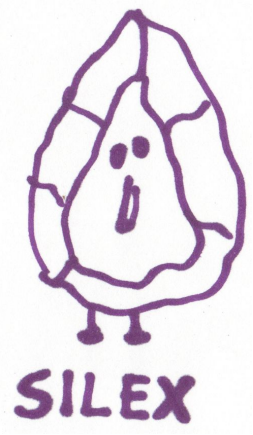
Simple ideas require new knowledge

There is More: We built these Microservices using a PHP Framework named SILEX!





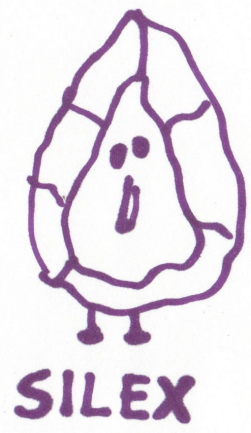
Silex in a few words



http://silex.sensiolabs.org/doc/usage.html

- It’s a Micro-framework (you can build a one file app!)
- Imagine something “like” Camel but in PHP with less stuff.
- Makes handling HTTP requests so simple
- It’s based on a larger framework named **Symfony** (DRUPAL 8 is Symfony!)

Silex in a few words



- Simple idea (Event based Routing system):
  - You define **Routes** (URLs) that are bound to a Method (GET, POST, etc)
  - Each route has a **Controller** that does stuff when invoked
  - Each route has **Middleware** that allows us to change what we get before invoking the **Controller**
  - It’s a like a Piping system
  - Silex handles all http protocol for us: Headers, body, etc
  - Lastly: A container system with **lazy loading** and uses **Dependency Injection** too.

Simplest Silex app


```php
// src/app.php

require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

$app->get('/hello/{name}', function($name) use($app) {
    return 'hello '.$app->escape($name);
});

$app->run();
```

Our Composer autoloader

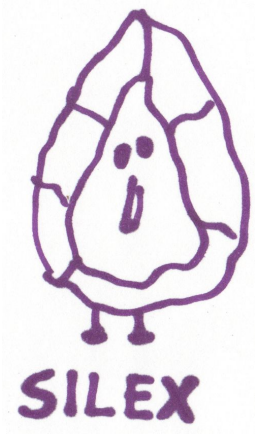
Base Container/Silex Bootstrap

Route

Controller

Main Loop, Kernel, Event Routines

Terms we used (buzzwords) explained



Route

- Silex provides PHP methods for all HTTP methods (->get,->post, ->etc..)

```
$app->get('/route1/{id}', function(Silex\Application $app, $id) use($somevar) {
      if ('hydra' == $id) {

            $app->abort(404, "This is Islandora");

      }

      return 'Found '.$app->escape($id);
});
```

- Each route is matched using a pattern of fixed and/or dynamic params that build an URL.
- **$app** is typed and **injected** into the closure.
- **$id** (and any other dynamic argument of the pattern also)
- **'use'** is a normal parameter passing to the closure
- '**return**' in this scope means an HTTP CODE 200 response with a body given by the returned value.
- We can **abort** (shortcut for cutting the pipe and returning)

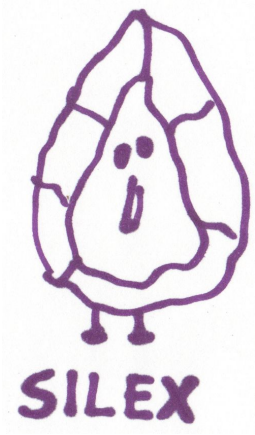
Another example (Let's use namespaces!)


```php
// src/app.php
require_once __DIR__.'/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$app = new Silex\Application();

$app->post('/receiver', function(Request $request) use($app) {
      $sentstuff = $request->get('stuff');

      …

      return new Response('Gracias, got your stuff', 201);

});

$app->run();
```

Terms we used (buzzwords) explained

Controller

- The worker. Does the processing.
- Can be
  - Inline
  - A Class/Method (lazy loaded, only when route is called)
  - A Service (next Session)

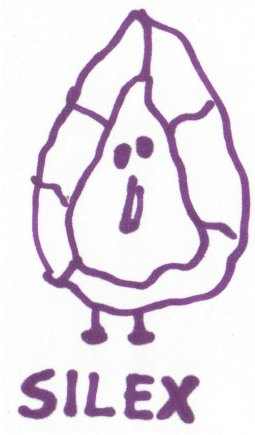



```
$app->post('/route1', 'Islandora\\Magic::fillrepo');
```

Terms we used (buzzwords) explained

Middleware

- Intercepts request (message, vars, etc)and can modify them before/after passing to the controller
- ->before, ->after, ->finish
- Some can be used globally (directly on $app)
- Can be chained

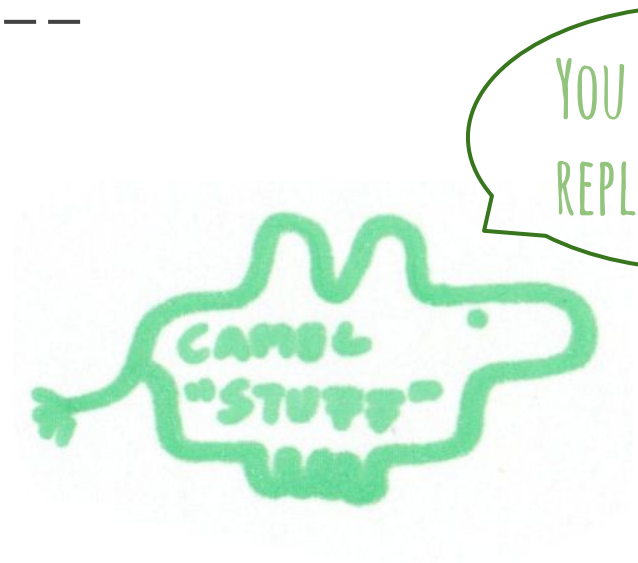



```
$app->post('/route1', 'Islandora\\Magic::fillrepo')
->before($dumpfedora3)
->before($converttordf)
->after($takearest);
```

There is so much more...

Silex will replace CAMEL?

– – –



Show me real code !