

A REAL CASE SCENARIO

REST API

MY NAME IS FABRIZIO

- ▶ I'm Italian but I live in Berlin
- ▶ I'm a PHP Backend Developer for Flaconi
- ▶ Founder of Startup Cicero (travel planner)
- ▶ It's my first talk, be nice :)

WHAT IS THIS TALK ABOUT

- ▶ Issues when we migrate from a Monolithic application to a separated BE/FE;
- ▶ REST best practices;
- ▶ Keep an eye on Testing;
- ▶ Be prepared for Micro-services;

WHAT THIS TALK IS -NOT- ABOUT

- ▶ Not a talk about REST fundamentals;
- ▶ It's not about OAuth;
- ▶ We will not speak about caching and rate limits;
- ▶ We will not speak about HATEOAS;

WHAT IS
REST?

REST & RESTFUL?

- ▶ REST is not a technology, neither a defined standard.
- ▶ REST is an approach. There are a set of rules to follow, but not specific rules for every situation.
- ▶ That means there is not 'silver bullet' for your specific problem.
- ▶ REST summarise good practices, and most likely you don't need previous knowledge to use those specific API.

THE PROBLEM

THE SOFTWARE YOU ARE
WORKING ON IS PROBABLY
ALREADY THE LEGACY SYSTEM

Fabrizio Ciacchi

LEGACY SYSTEM

- ▶ Your company might have:
 - ▶ monolithic application;
 - ▶ BE/FE with API but not well organised;
 - ▶ Or you have several (micro)service, looking like the flying spaghetti monster :)
- ▶ Software lifespan is 18 months.

BE/FE + MAGENTO

- ▶ When I joined Flaconi, they already started to split BE/FE, but the BE APIs were not following any rule;
- ▶ They were implemented each time for that particular need and in different ways (endpoint, verbs, response code, json structure);
- ▶ Plus the legacy part was in Magento and needed to be migrated.

BAD EXAMPLES

/cms/block/id/{id}

/cms/block/key/{key}

/cms/block/keys/{key1,key2,...}

/line/{id}

/product-review/get/{id}

/customer/4

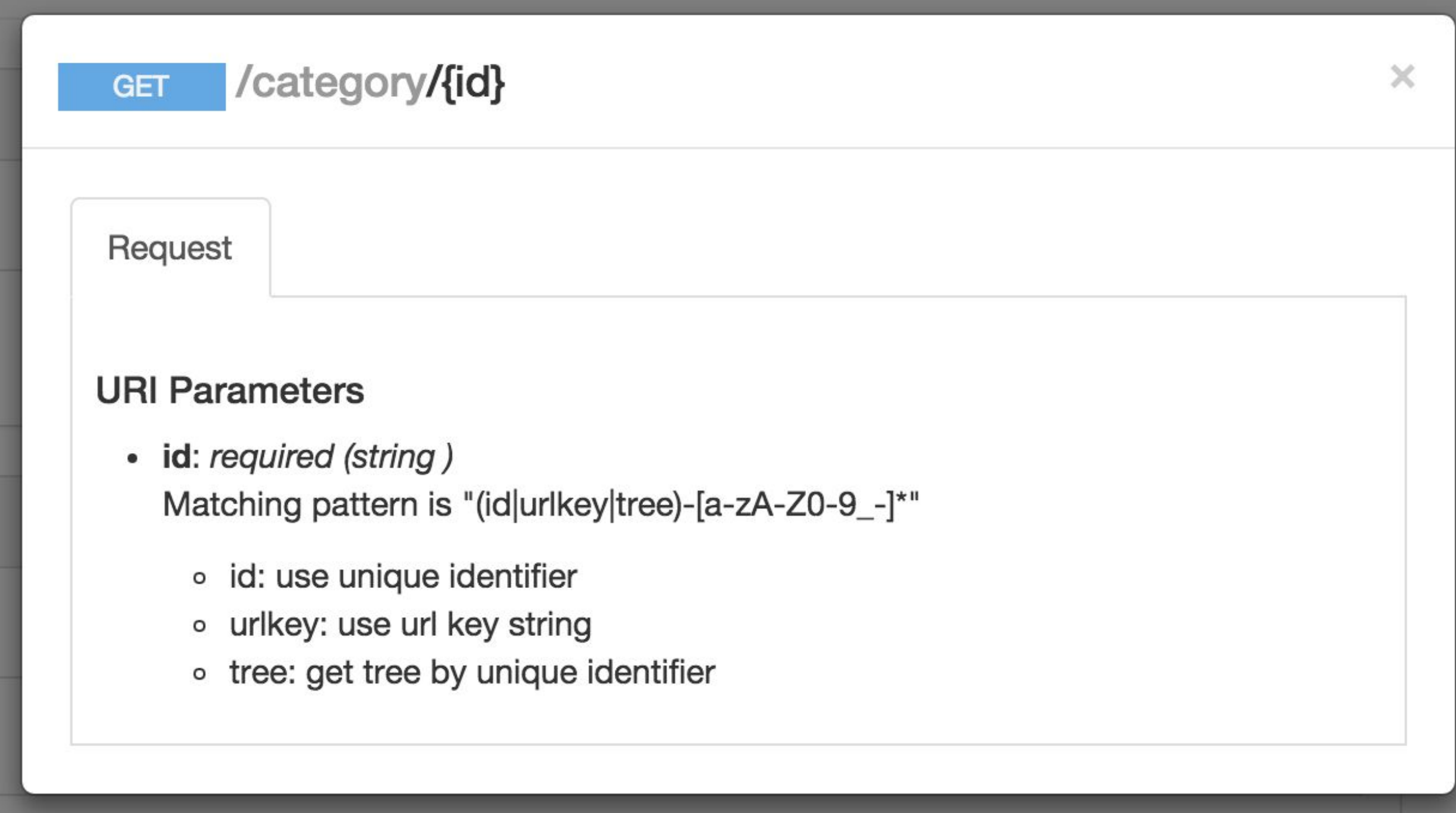
/customer/email/{base64}

/customer/4/orders/1900002

/customer/5/orders/1900002 ← Possible to retrieve an order for another user

LACK OF DOCUMENTATION 1

- ▶ What does that mean?



GET /category/{id}

Request

URI Parameters

- **id:** *required (string)*
Matching pattern is "(id|urlkey|tree)-[a-zA-Z0-9_-]*"
 - id: use unique identifier
 - urlkey: use url key string
 - tree: get tree by unique identifier

LACK OF DOCUMENTATION 2

- ▶ How do I know that the object is inside the key transfer?

```
{
  "transfer_customer_credit": {
    "certId": "5269865",
    "certNumber": "GUTHABEN-12612",
    "balance": "5.0000",
    "currencyCode": "EUR",
    "status": "A",
    "expireAt": "2015-07-07 23:59:59"
  },
  "status_code": 200,
  "api_status": "successful"
}
```

WHAT DID
WE LEARN?

CLEAR BEHAVIOUR

- ▶ There are many HTTP verbs and you can apply them on both collections or elements. Make it simple and stick to it.

	Safe	Idempotent	Possible on	Success Response
GET	Yes	Yes	Collections Elements	200
POST	No	No	Collections*	201 + Location
PUT	No	Yes	Elements**	204
DELETE	No	Yes***	Elements**	204/404***

PUT VS PATCH

- ▶ Infinite debate:
 - ▶ PUT pass all the info
 - ▶ PATCH pass only the info to update
- ▶ Not passing or passing empty/null values in PUT should reset the correspondent values;
- ▶ Choose only one for your system, and stick to it.
- ▶ DELETE 204 vs 404? Depends on your security.

1-TO-1 MAPPING OF VERBS

- ▶ HTTP verbs should be mapped 1-to-1 to CRUD operations.

Operation	SQL	HTTP
Create	INSERT	POST
Read (Retrieve)	SELECT	GET
Update (Modify)	UPDATE	PUT
Delete (Destroy)	DELETE	DELETE

- ▶ Why? Consistency!

RESPONSE CODES

- ▶ There are many HTTP response codes, use it!
 - ▶ 200 (GET)
 - ▶ 201 + Location (POST)
 - ▶ 204 (PUT/DELETE)
 - ▶ 301 Moved Permanently
 - ▶ 404 Resource not found (/customers/5 -> does not exists)
 - ▶ 405 Method not allowed (should tell which verbs are allowed)
 - ▶ 409 Conflict (try to insert twice the same resource)
 - ▶ 415 Unsupported media type
 - ▶ 422 Unprocessable entity (wrong payload POST)
 - ▶ 500 Managed Exceptions

URL

- ▶ Always end a collection with a slash "/"
 - ▶ /customers/
 - ▶ /carts/
- ▶ Never end an element, by ID or attribute, with a slash:
 - ▶ /customers/4
 - ▶ /customers/4/carts/active

DDD APPROACH

- ▶ Use the 'Domain' definition as in the Domain Driven Design to define your API:
 - ▶ Customer is one domain;
 - ▶ Cart is one domain;
 - ▶ Wishlist is one domain;
- ▶ In this way the APIs are decoupled and can be split in the future into micro-services.

RESPONSE GUIDELINES

- ▶ Don't use plain response in your json.
- ▶ It's better to use an 'element' or a 'collection' key. It's easier to parse.
- ▶ Start using "_link", as a first step to HATEOAS
- ▶ Use the proper vendor HTTP header:
 - ▶ application/vnd.flaconi.customers+json
- ▶ You can also add versioning:
 - ▶ application/vnd.flaconi.customers.v1+json

YOU CAN SERVE DIFFERENT FORMATS

- ▶ Using the media type header to drive the response format
(Accept on request / Content-type on answer)

JSON Representation:

```
{ "user": { "name": "Chuck Norris", "occupation": "martial artist" } }
```

XML Representation:

```
<user>  
<name>Chuck Norris</name> <occupation>martial artist</occupation>  
</user>
```

HTML Representation:

```
<html>  
<head><title>Web page of Chuck Norris</title></head> <body><p>Name: Chuck Norris</p>  
<p>Occupation: martial artist</p></body>  
</html>
```

HOW DOES IT LOOK LIKE NOW?

- ▶ Endpoints:
 - ▶ /carts/4/products/
 - ▶ /cms/chanel
 - ▶ /customers/fabrizio@ciacchi.it
 - ▶ /customers/4/orders/
 - ▶ /wishlist/3

AND THE RESPONSE?

```
{
  "element": {
    "id": "12612",
    "email": "magento_12612@example.de",
    "createdAt": "2016-03-12 16:00:00",
    "isActive": "1",
    "defaultBilling": "30556",
    "defaultShipping": "59952",
    "firstname": "firstname_12612",
    "gender": "m",
    "lastname": "lastname_12612",
    "middlename": null,
    "passwordHash": "XXXXXXXXXXXX",
    "prefix": "Herr",
    "phone": null
  },
  "_links": {
    "_self": "http://backend.flaconi.de/customers/12612"
  },
  "status_code": 200,
  "api_status": "successful",
  "message": "OK"
}
```


TAKEAWAYS

- ▶ Always start from the design of your API. Try to understand the requirements with a long view, and be prepared to be flexible.
- ▶ Be practical. If it makes sense to return the list of products within your "cart" response, do it. You'll save 50% of the API calls to your shopping cart.
- ▶ Never implement only one verb. Having a POST but not having a GET, even if it's ok for your product, will not make your life easier when you test/debug.
- ▶ Always write the documentation for your API. There are great tools like RAML!

SHOULD I BE ALWAYS RESTFUL?

- ▶ The answer is NO
- ▶ Example:
 - ▶ /wishlist/id/421
 - ▶ /wishlist/customer/2
 - ▶ /wishlist/share/abc82jdh287ha
- ▶ It's not rest at all, but it works!

WHY WE
BUILD API
WITH REST?

TO SAVE MONEY

- ▶ As soon as we went online with the new REST API, we found out that:
 - ▶ 1/3 of the API calls FE to BE where returning 404;
 - ▶ It was a call to get optimised content for a brand, but would return 404 if the request was for 2 or more brands.
- ▶ The cost of the servers is up to 10.000 €/month
 - ▶ Saved around 3.000 €/month

TO HAVE A MORE RELIABLE SYSTEM

- ▶ But we also notice other things:
 - ▶ A lot 422 responses for Customers (invalid registration);
 - ▶ Several 405 for Applying a Coupon (empty coupon code);
 - ▶ Most of the 500 errors related to get Products (wrong join);
- ▶ You want to have a clean, reliable system. Because when there is something wrong, you want to see it.

A STEP TOWARDS THE FUTURE

- ▶ Being RESTful is also a must-step if you want to:
 - ▶ Implement HATEOAS
 - ▶ Switch to micro-services
 - ▶ Test your application

TESTING WITH CODECEPTION

TEST THE GET 1/4

```
INSERT INTO `customer_entity` (`entity_id`, `entity_type_id`, `attribute_set_id`,  
`website_id`, `email`, `group_id`, `increment_id`, `store_id`, `created_at`,  
`updated_at`, `is_active`, `email_sender`, `email_long_order`, `recency`,  
`frequency`, `monetary_sum`, `monetary_avg`, `first_sale_date`, `last_order_sum`,  
`customer_group`, `currency_code`, `last_sale_date`, `blacklist`, `current_points`,  
`used_points`, `bp_coupon_code`, `trigger_status`, `arvato_comda_number`)
```

VALUES

```
(4, 1, 0, 1, 'jon-snow@example.de', 1, NULL, 1, '2012-06-15 18:04:42',  
'2015-11-26 16:28:09', 1, 0, 1, '0', 0, NULL, NULL, NULL, NULL, NULL, 'EUR',  
NULL, 0, 0, 0, NULL, 0, NULL);
```


TEST THE GET 2/4

```
$endPoints = [  
  'customer' => [  
    'get' => [  
      'params' => '[:id]',  
      'uri' => 'customers/[:id]',  
    ],  
  ],  
];
```

TEST THE GET 3/4

```
public function getWithSuccess($scenario, $id, $response = 200)
{
    /** @var ApiTester $I */
    list($I, $uri) = $this->initiateTest($scenario, $id);

    $I->wantToTest('Check that ' . $this->domain . ' GET works');
    $I->sendGET($uri);

    $I->seeResponseCodeIs($response);

    $I->seeResponseIsJson();

    $I->seeHTTPHeader('Content-Type', 'application/vnd.flaconi.' . $this->domain . '+json;v=1; charset=utf-8');

    $I->canSeeResponseJsonMatchesJsonPath('_links._self');

    $I->seeResponseContains('"element":');
    $I->seeResponseContains('"id": "' . $id . '"');
}
```

TEST THE GET 4/4

```
$customerId = 4;
```

```
$invalidCustomerId = 3;
```

```
include_once(__DIR__ . '/../_base/Customers/GetCustomer.php');
```

```
use Codeception\Customers\GetCustomer;
```

```
$customerTest = new GetCustomer($locale);
```

```
/** @var \Codeception\Scenario $scenario */
```

```
$customerTest->getWithSuccess($scenario, $customerId);
```

```
$customerTest->getNotFound($scenario, $invalidCustomerId);
```

```
$customerTest->getWithEmptyId($scenario);
```

```
$customerTest->getWithException($scenario);
```

ANY QUESTION?

fabrizio@ciacchi.it

LINKS

- ▶ REST APIs with Symfony2: The Right Way
 - ▶ <http://williamdurand.fr/2012/08/02/rest-apis-with-symfony2-the-right-way/>
- ▶ Best Practices for Designing a Pragmatic RESTful API
 - ▶ <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>
- ▶ Getting Started with REST and Zend Framework 2
 - ▶ <http://hounddog.github.io/blog/getting-started-with-rest-and-zend-framework-2/>

LINKS

- ▶ REST: From GET to HATEOAS
 - ▶ <http://www.slideshare.net/josdirksen/rest-from-get-to-hateoas>
- ▶ Using HTTP Methods for RESTful Services
 - ▶ <http://www.restapitutorial.com/lessons/httpmethods.html>
- ▶ Building a Hypermedia-Driven RESTful Web Service
 - ▶ <https://spring.io/guides/gs/rest-hateoas/>
- ▶ Implementing a RESTful Service Server-Side
 - ▶ <http://dojotoolkit.org/reference-guide/1.10/quickstart/rest.html>