@albertomazaira

CROSS-SERVICE UNIT TESTS FOR YOUR MICRO SERVICES SANITY.

FIDOR SOLUTIONS

- Product Owner Community.
- Product Owner Crypto.
- Whatever that pops up.
- I fell last week in the snow 🏂, forgive me if I ⌐╯

FIDOR GROUP

- Fintech, bank, startup, consulting….
- Bank vs. Solutions.

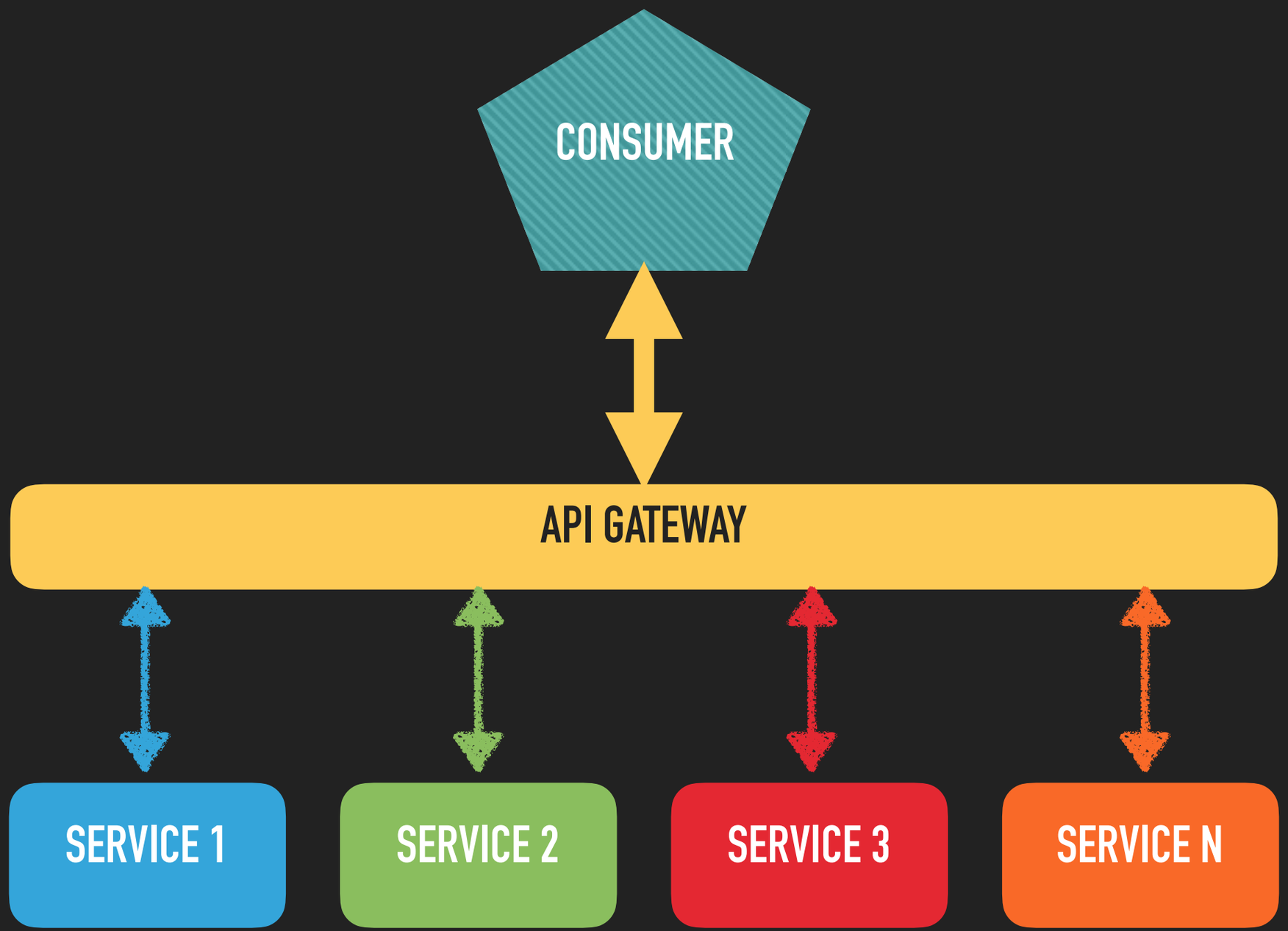
PRESENTATION OF THE PROBLEM

- You have multiple services with different set of APIs.

PRESENTATION OF THE PROBLEM

- You have multiple distributed teams with their own ways of working.
    - What is common sense for you usually is not common sense to others.

COMMON SITUATION



MAIN PAIN POINTS FOR CONSUMER

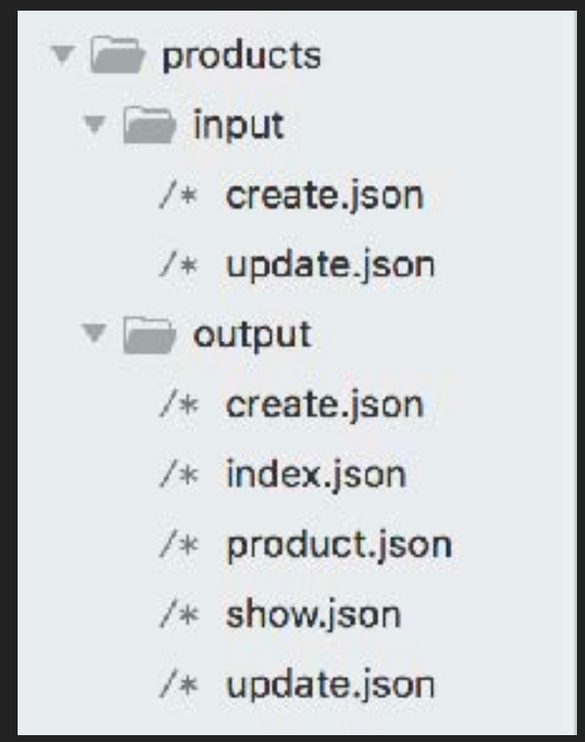
- Different ways of endpoint serialization.
  - API standards.
- Different ways of API discovery/aggregation.
  - Compound documents.
- Different error handling.
  - The worst and most painful thing ever. Random 422 keys.
- Different/Absence_of authentication.

FIRST TRIES ON THE RIGHT DIRECTION

- Adopt an specification: JSONAPI.
  - Serialization.
  - Pagination.
  - Filtering: GET /products?filter[category]=60
  - Sparse fieldsets. GET /posts?fields[posts]=title,body
  - Compound documents.

FIRST TRIES ON THE RIGHT DIRECTION

- New JSON schemas:
  - Input and output.
  - Dependent on the verb.



- Update POSTMAN collection.

FIRST TRIES ON THE RIGHT DIRECTION

- Unit test:
  - Error compliance: 401, 404, 409, 412, 415, 422…
  - Acceptance headers.
  - Pagination.
  - Includes.
  - Filtering.
  - Roles.
  - JSON schema compliant.
  - JSON API specification compliant.
  - Data dictionary:
    - Add to json_schema_data definitions.
    - Object vs data_dictionary.
- Integration flag: turn off mocking and perform real calls.

FIRST TRIES ON THE RIGHT DIRECTION

- DoD wikis.
- Acceptance criteria for coding standards (in wiki too).
- Core team discussion.
- Technote to explain the new bright way.

WHAT HAPPENED?

ONLY ONE TEAM ADOPTED IT...

MINE.

NEED TO MOVE FROM WIKI TO GITHUB.

US

- We are mostly a Ruby based company.
- The selected framework for the base is called Rspec, pretty similar to Mocha or Jest.

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

- Leverage on standards.
- Leverage on polymorphism and a bit of metaprogramming.
- It works exhaustively but with some tradeoffs for the sake of performance.
  - Only headers on read, mostly only files on write.

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

- How it works?
  - To be applied both to the C and the M levels.
  - Controller level takes care about the action.
  - Model level takes care about the data structure.

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

- How it works?
  - Define only the data model and state how it should behave.

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

```ruby
describe 'POST #create' do
  let(:title) { 'title' }
  let(:body) { 'body' }
  let(:category) { FactoryGirl.create(:category, :wanted_product) }
  let(:category_id) { category.id }
  let(:photo) do
    "\
    #{Base64.strict_encode64(File.read('./spec/fixtures/to_upload/line.png'))}"
  end
  let(:photo_name) { Faker::Lorem.word }
  let(:params) do
    {
      data: {
        type: 'wanted_product',
        attributes: {
          title: title,
          body: body,
          photo: photo,
          photo_name: photo_name,
          category_id: category_id
        }
      }
    }
  end

  it_behaves_like 'creating object',
                  model: :wanted_product,
                  required_fields: [:title, :body, :category_id],
                  included: 'author,category'

  it_behaves_like 'image created',
                  model: :wanted_product,
                  attributes: [:photo]
end
```

MOUNT A LIBRARY AND PLUG IT EVERYWHERE

```ruby
context 'valid params' do
  before do
    sign_in current_user
    post :create, params: params, format: :json
  end

  it 'input validates with custom JSON schema' do
    expect(params).to match_input_schema
  end

  it 'creates object' do
    class_name = model.to_s.classify.constantize
    expect(class_name.count).to eq 1
  end

  it 'response is success' do
    expect(response).to be_success
  end

  include_examples 'JSONAPI validation'
  include_examples 'JSONAPI validation with custom schema'
end

required_fields.each do |attribute|
  context "missing attribute #{attribute}" do
    let(attribute.to_sym) { nil }
    before do
      sign_in current_user
      post :create, params: params, format: :json
    end

    it 'does not create object' do
      class_name = model.to_s.classify.constantize
      expect(class_name.count).to eq 0
    end

    include_examples 'validation error'
    include_examples 'JSONAPI validation'
  end
end
```

DATA DICTONARY

```
  },
  "title": {
    "type": "string",
    "pattern": ".*{1,255}",
    "maxLength": 255
  },
  "photo": {
    "type": ["string", "null"],
    "pattern": "^($|^(.+)\/([^\/]+)$
  },
"image_url": {
  "type": ["string", "null"]
},
"body": {
  "type": ["string", "null"],
  "pattern": ".*{1,4096}",
  "maxLength": 4096
},
"date": {
  "type": "string",
  "format": "date-time"
},
"date_nullable": {
  "type": ["string", "null"],
  "format": "date-time"
},
"site": {
```

CHALLENGES

- Difficult to track across teams.
- No backwards compatibility.
- Motivates /v2 of APIs.
- Requires refactor in core parts.

BENEFITS

- You have standard APIs.
- Reliability.
- Way less code.
- Empowers TDD.
- Eases path for testers.

FROM TEST TO IMPLEMENTATION

- Once you have a standard expectation for an API, you can have a standard way to produce them.
- We decided to create another lib, to actually create the APIs for us.

BASE CRUD

- Library which generates CRUD services from a data specification.
- Support nested resources and relations.
- Support multiple dbs.
- Minimum implementation.
- Everything overridable.

USE CASES

- Proof of concept apps.
- Client on site workshops.
- Dummy endpoints.
- 3rd party integrations.

SUM UP

From a situation where we had a nightmare of

API specifications, we came up with the idea to test all of them vs. the same standards, and we ended up auto generating them on an automated way.

THANKS.
QUESTIONS?