Testing in a Microservice System

By: Laura Johannet, Senior Software Engineer Care/Of, former Principal Engineer at Zocdoc
In the beginning there was the monolith....
Monolith Pros

- Focused on one main technical goal/not a lot of siloed areas
- Small project, so easy to maintain
- Simpler/less expensive than distributing things
  - logging, rate limiting, & security features such as audit trails, easy to share those services
The time came to distribute services

- No language limit
- Can scale the high traffic/hot parts of the app
- Deploying microservice applications can be easier
New System, New Testing Concerns
Testing the Service

1. Run a lint-er:
2. Run database tests
3. Run unit tests
4. Deploy service to staging
5. Run http tests against service in staging (end to end)
6. Deploy service to production
7. Run http tests against service in production (product tests)
   - Tests run every 10 minutes
Microservice Dependencies in Dev/Test Mode

Microservice A hits micro service B - what do you hit in development and in testing?

There was still a monolith consuming a lot of microservices. What does the monolith hit in its end to end testing flows?
What Did Not Work

- Hit staging
  - Staging can be flakey.
- Hit production
  - Big move, requires re-thinking a lot of code flows, we were not setup to be hitting production at our test load/were not sure we wanted to do that
public void GivenPetServiceGetPetReturnsDogThenSitterServiceReturnsFalse() {
  GivenPetServiceGetPetReturnsDog();

  var result = sitterService.availablePetSitter();

  Assert.IsFalse(result);
}

public void GivenPetServiceGetPetReturnsDog() {
  Mock<PetService>().Setup(x => x.GetPet).Returns(dog);
}
Introducing Open API

- OpenApi is an API description format for RESTful APIs
  - Language agnostic
  - Allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.
- API’s were all documented out using OpenAPI specification.
Swagger Example

/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
Swagger and Interfaces Looks Similar...

- Swagger allows you to do the same thing! Describes
  - What the endpoints are
  - What the expected input types are
  - What the response types are
Swagger Express Middleware - Mock Services

- **swagger express middleware**
  - NPM package/express middleware
  - takes in a Swagger document and then mocks out responses based on the swagger schema examples.

```json
/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
            example:
              id: 1
              name: "rover"
```
Swagger Express Middleware - Mock Services

- Node service + swagger express middleware + swagger doc = service with all endpoints mocked out
- localhost:3000/pet/1 would return pet object

```
/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
            example:
              id: 1
              name: "rover"
```
Deploying Mock Services

- Engineer writes code in pet-service
- Engineer merges code to master
- Testing flow occurs
- Docker image with node mock service + swagger document + swagger is generated
  - Tagged latest
- Docker image gets pushed up named pet-service-mock
- Consumers of pet-service pull latest mock from docker
  - pet-service-mock:latest
More Than Just One Mock

How can we setup mocks for different pet types? Right now swagger-express-middlewire only works off of the latest example in swagger document

```csharp
public void GivenPetServiceGetPetsReturns(Pet dog) {
    x => x.GetPet).Returns(dog);
}
```

We want to test different responses (all the use cases!)

```csharp
public void GivenPetServiceGetPetsReturns() {
    Mock<IPetService>().Setup(
        x => x.GetPet).Returns(swaggerLastestMock);
}
```

```csharp
public void GivenPetServiceGetPetsReturns(Pet dog) {
    Mock<IPetService>().Setup(
        x => x.GetPet).Returns(dog);
}
```
Done!

- Parrot lets you hit the mock service before each test and tell it what you want it to respond with for a given input
- https://github.com/mustafar/parrot
Mocked response example

Call `PUT /mock` to mock another route

```javascript
// mock
const mock = {
    method: 'GET',
    path: '/batman/location',
    status: 201,
    response: 'arkham',
};
await fetch('
    '${apiBase}/mock',
    {
        method: 'PUT',
        body: JSON.stringify(mock),
        headers: { 'content-type': 'application/json' }
    },
);

// invoke (should return "arkham (201)"
await fetch('${apiBase}/batman/location', { method: 'GET' });
```

DELETE /mock will reset all mocked behavior.
public void GivenPetServiceGetPetsReturns(Pet pet) {
    response = await fetch('mockService/mock',{ method: 'PUT', body: JSON.stringify(mock),
    headers: { 'content-type': 'application/json' } });
}

public void TestOne() {
    GivenPetServiceGetPetsReturns(dog);
    // assert things
}

public void TestTwo() {
    GivenPetServiceGetPetsReturns(cat);
    // assert things
}

public void GivenPetServiceGetPetsReturns(Pet pet) {
    response = await fetch('mockService/mock',{ method: 'PUT', body: JSON.stringify(mock),
    headers: { 'content-type': 'application/json' } });
}
Keeping Mocks Up to Date

- Run tests against OpenAPI document to ensure OpenAPI contracts = service contracts
- `swagger-test` is a tool that lets you do this.
  - Takes in OpenAPI document
  - Makes sure endpoints in doc exist in service with correct input/response
- Plan is to include that in Parrot
You can mock out microservices using up to date contracts.
- OpenAPI to get meaningful contracts
- swagger-express-middleware to create mock services from OpenAPI docs
- Make sure OpenAPI contracts == service real contracts, `swagger-test`
- **Parrot**
  - Deploy a mock service with each service deployment that mocks out each swagger endpoint
  - customize mock responses per test