AARHUS UNIVERSITET

Microservices and DevOps

Scalable Microservices

Versioning

Henrik Bærbak Christensen

Motivation

- Microservices = independently deployable services, collaborating to form a whole system.
- *However, as we make changes to add features, we need to be careful not to break consuming applications [Nygard, p 263]*
  - Do not force consumers to match your release schedule

- Postel's robustness principle
  - Be conservative in what you do, be liberal in what you accept from others.

‘Own’ Versions

Those that we ourselves control

‘Inter-organization services’

Non-breaking Changes

- Non-breaking = obey all agreements on all levels of the stack (http, tcp, ip, ….)
- Request-reply is asymmetric wrt. ‘robustness’
  - Can accept *more but never* less, and never require more (Req)
  - Can return more, but never return less (Reply)

- Nygard reflection
  - Is the specification the *documented one* or the *implemented one?*

  - *Nygard’s standpoint: it is the implemented one*
    - Which must keep obeying the robustness principle

Testing Concerns

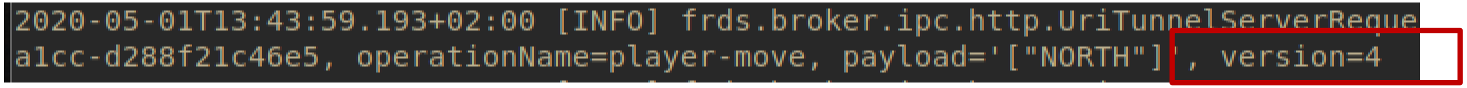
- Nygard advocate *random generative testing* against your service, to find ‘gaps’ between spec and implementation
  - Resemble his ‘test harness’ pattern to generate ‘out-of-spec’ tests

- Alas
  - Randomized tests that do weird thing given the structure of the API
    - Forget keys in JSON post, send null values, empty arrays, etc.
- Ex
  - I found a bug in my ‘CaveService’ as I POST’ed a room with user id as key ‘id’ instead of ‘creatorId’. The service just made a room without a creator ☹

Breaking Changes

- A Breaking Change is necessary. What to do?

- Principle 1:
  - *Use version numbers on the message format*
    - Not an application version, but a *format version*
    - *The ‘format indicator’ pattern in Messaging (Hohpe & Woolf, 2004)*
  - Helps in debugging and detection



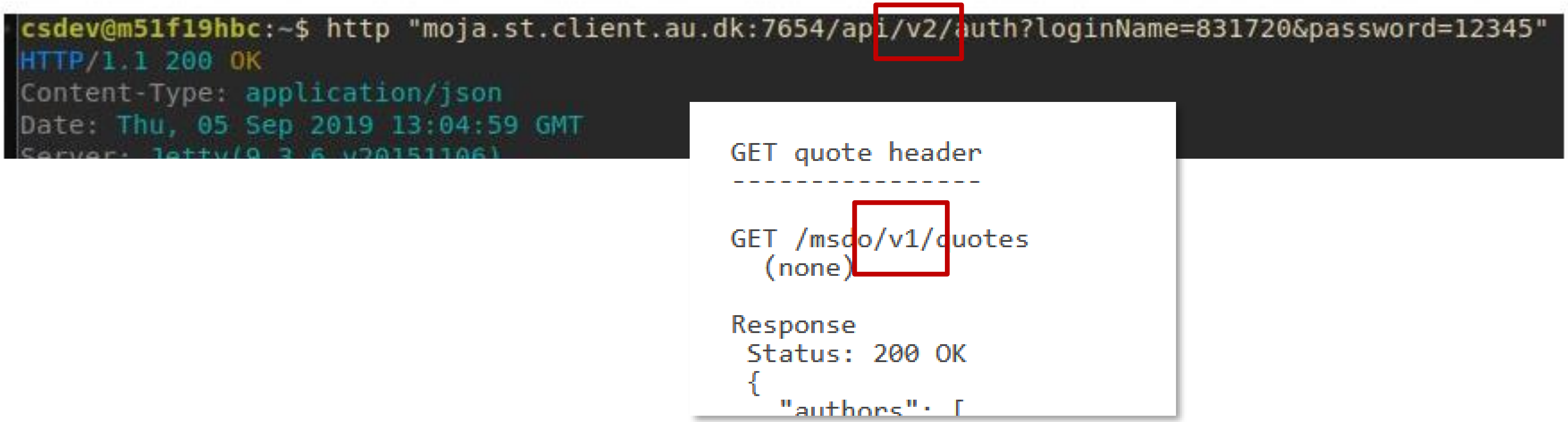
```
2020-05-01T13:43:59.193+02:00 [INFO] frds.broker.ipc.http.UriTunnelServerReque
a1cc-d288f21c46e5, operationName=player-move, payload='["NORTH"]', version=4
```

Breaking Changes: REST

- Http REST service: Changes in the API structure ?
- Proposals
  - Put version in URL: /v1/xya
  - Use ‘Accept’ header and ‘Content-Type’ header
  - Intro a ‘api-version’ custom header
  - Intro a version key in the request body
- All are bad, but least pain is first proposal
- Principle 2:
  - *Version the API by adding version id in the URL*
    - Easy to understand by developers
    - No fiddling with load-balancers, caches, proxies, etc.

SkyCave Examples

- From our own backyard



```
csdev@m51f19hbc:~$ http "moja.st.client.au.dk:7654/api/v2/auth?loginName=831720&password=12345"
HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 05 Sep 2019 13:04:59 GMT
Server: Jetty(9.3.6.v20151106)
```

```
GET quote header
----------------

GET /msdo/v1/quotes
  (none)

Response
 Status: 200 OK
 {
   "authors": [
```

- Btw: Did you do it in the REST services ☺?

Breaking Changes

- Principle 3:
  - *Both old and new version must be supported 'for some time'*
    - That is, side-by-side operation
    - Test heavily with a mix of versions
      - CREATE with new API and READ with old often poses problems
  - All new paths must be available at the same time
    - It is a no-no to have half of the features migrated to /v2 but forcing clients to access the other half using /v1 !!!

Breaking Changes

- Principle 4
  - *Use a (1 version deep)* ***translation pipeline*** *for the old version code*

- That is
  - /v2 controller code forward directly to business logic layer
  - /v1 controller code convert incoming to new format, call business logic, convert result back to v1 format and return…

‘Others’ Versions

Intra-organization versioning

If ‘Others’ Do Not Behave Well

- Growth scenario
  - Your API adds three new fields to a query
- Reflection
  - All combinations of weird/missing assignments to these new fields are to be expected!

- Principle 5
  - Your software should remain cynical! Protect your service, apply the stability patterns to each and every integration point.

AARHUS UNIVERSITET

The Testing Aspect

Testing Aspect

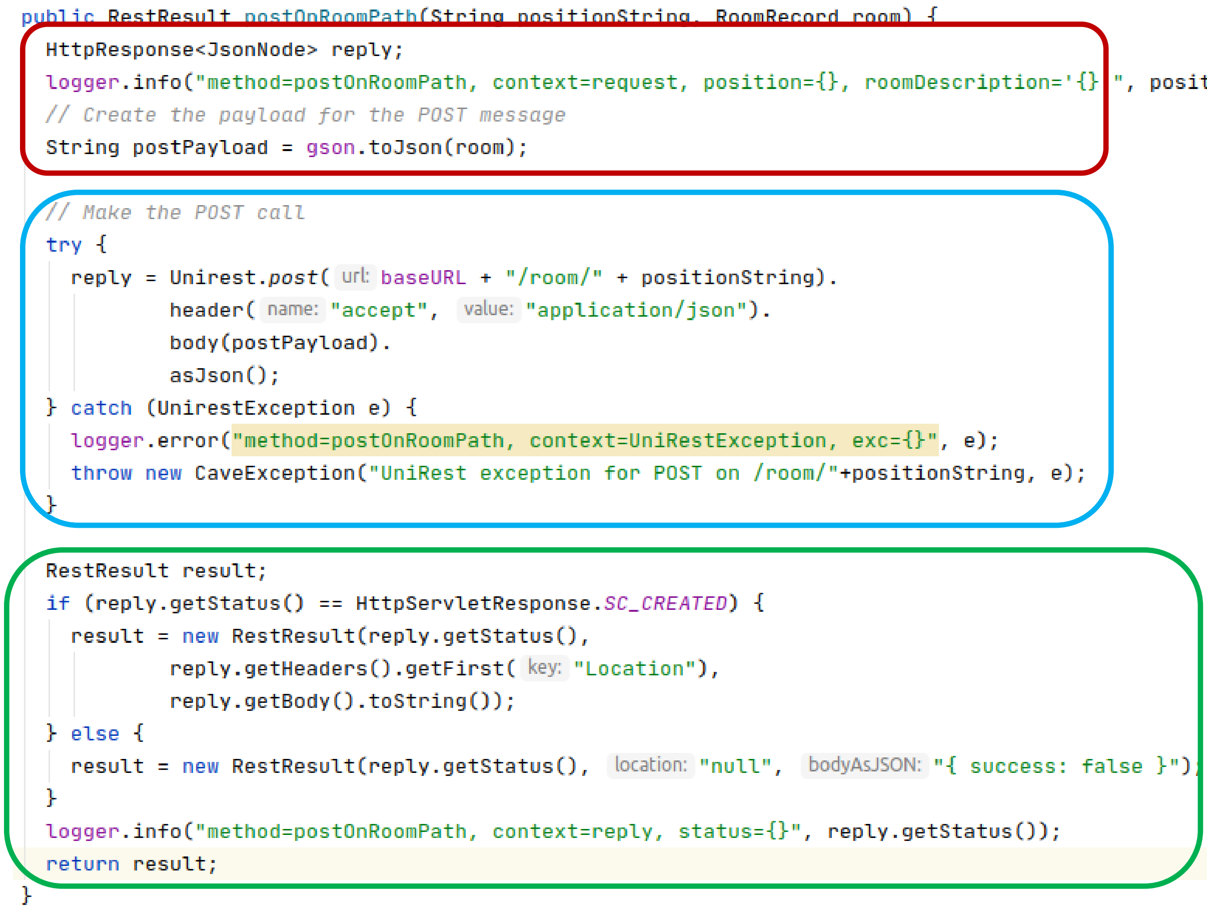
- The ‘call-external-service’ algorithm is basically
  - Convert domain object(s) to REST payload
  - Do the external service call
  - Receive the returned payload
  - Convert payload to domain object(s) and process
- That is
  - Translation, processing, translation
  - The version issue revolves around the ***translations!***

Example: My CaveService Connector

- Translating the room record (domain object) to JSON

- Call the service

- Translate reply to domain object (Hm, hm, so so…)



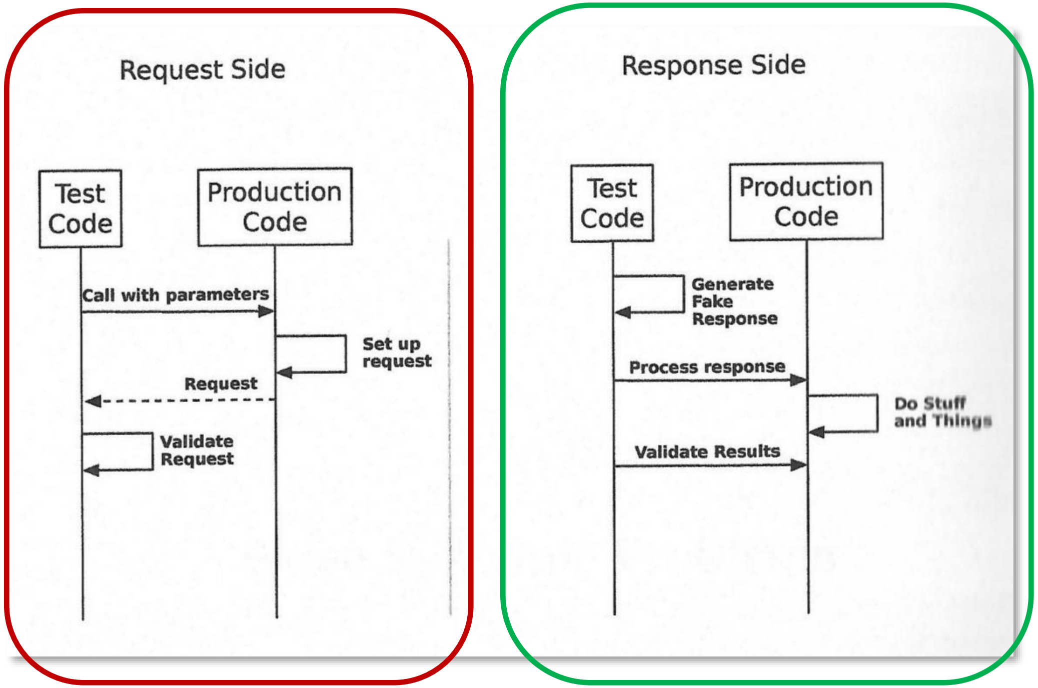
```
public RestResult postOnRoomPath(String positionString, RoomRecord room) {
  HttpResponse<JsonNode> reply;
  logger.info("method=postOnRoomPath, context=request, position={}, roomDescription='{}'", posit
  // Create the payload for the POST message
  String postPayload = gson.toJson(room);

  // Make the POST call
  try {
    reply = Unirest.post( url: baseURL + "/room/" + positionString).
            header( name: "accept",  value: "application/json").
            body(postPayload).
            asJson();
  } catch (UnirestException e) {
    logger.error("method=postOnRoomPath, context=UniRestException, exc={}", e);
    throw new CaveException("UniRest exception for POST on /room/"+positionString, e);
  }

  RestResult result;
  if (reply.getStatus() == HttpServletResponse.SC_CREATED) {
    result = new RestResult(reply.getStatus(),
            reply.getHeaders().getFirst( key: "Location"),
            reply.getBody().toString());
  } else {
    result = new RestResult(reply.getStatus(),  location: "null",  bodyAsJSON: "{ success: false }");
  }
  logger.info("method=postOnRoomPath, context=reply, status={}", reply.getStatus());
  return result;
}
```

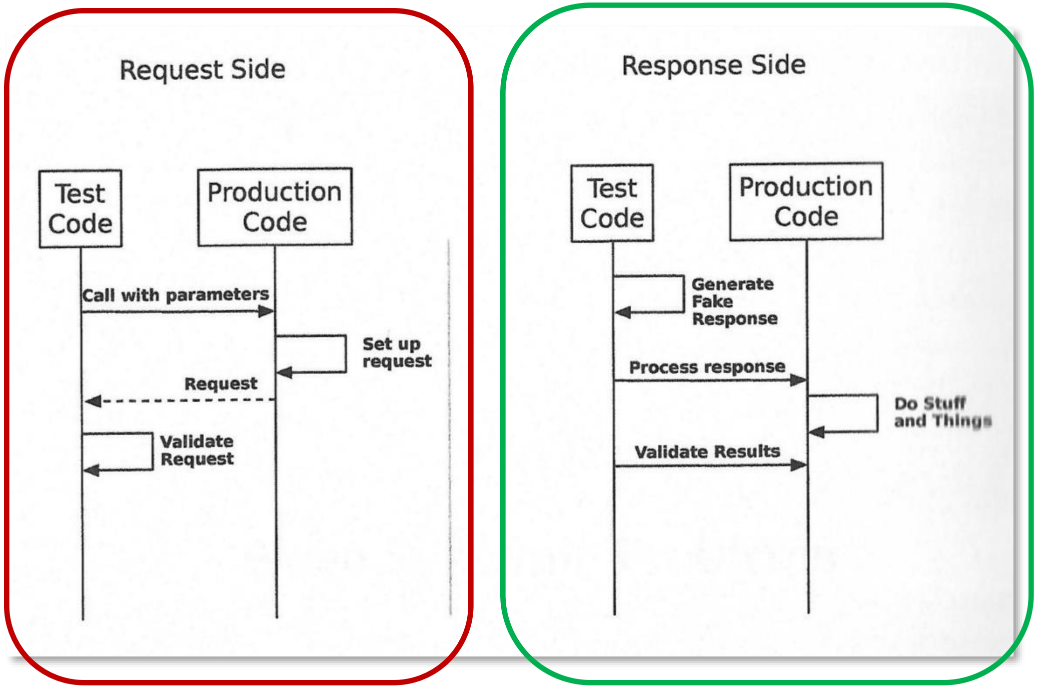
Testing Aspect

- Nygard recommend separate testing of the two translations to prepare for out-of-spec issues…



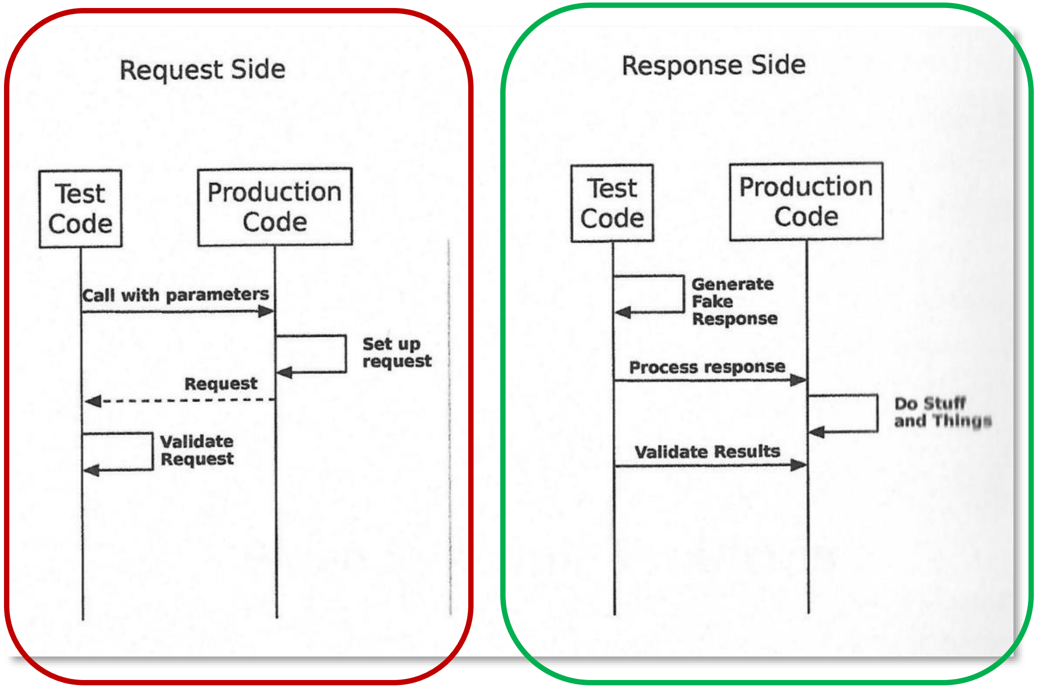
Testing Aspects

- The *request translation* side is as-far-as-I-can-see just normal contract testing
  - *Just checks that requests are created according to provider's requirement*



Testing Aspects

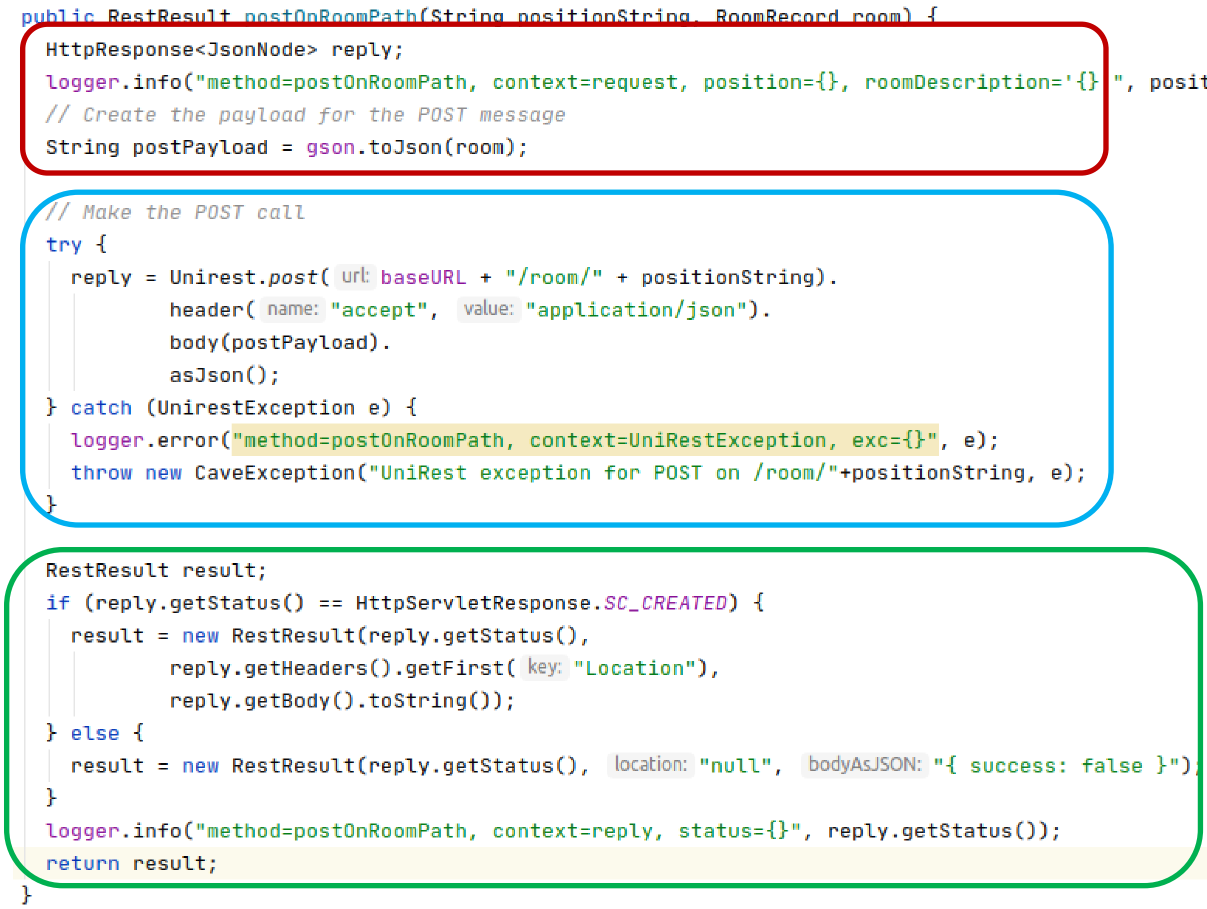
- The *reply translation* side is more interesting IMO
  - *Inject 'weird' responses and validate proper handling*
- Do not require actual remote calls



Requirements

- Have to further refactor my code to support proposed tests

- Each ‘box’ must be individually test units …



```
public RestResult postOnRoomPath(String positionString, RoomRecord room) {
  HttpResponse<JsonNode> reply;
  logger.info("method=postOnRoomPath, context=request, position={}, roomDescription='{}'", posit
  // Create the payload for the POST message
  String postPayload = gson.toJson(room);

  // Make the POST call
  try {
    reply = Unirest.post( url: baseURL + "/room/" + positionString).
            header( name: "accept",  value: "application/json").
            body(postPayload).
            asJson();
  } catch (UnirestException e) {
    logger.error("method=postOnRoomPath, context=UniRestException, exc={}", e);
    throw new CaveException("UniRest exception for POST on /room/"+positionString, e);
  }

  RestResult result;
  if (reply.getStatus() == HttpServletResponse.SC_CREATED) {
    result = new RestResult(reply.getStatus(),
            reply.getHeaders().getFirst( key: "Location"),
            reply.getBody().toString());
  } else {
    result = new RestResult(reply.getStatus(),  location: "null",  bodyAsJSON: "{ success: false }");
  }
  logger.info("method=postOnRoomPath, context=reply, status={}", reply.getStatus());
  return result;
}
```

Summary

- Postel’s principle is easy to state…
- But require quite a lot of coding efforts and testing…
- *Design for failure…*