# Microservice Architecture Blueprint

## Episode 7 "REST, Reliability and Chaos"

**Kim Horn**

**Version 0.82** **1 August 2017**

# Agenda

- REST
  - Search
  - API Evolution and Versioning;
- Reliability
  - HTTP is not reliable
  - Microservices Collaboration / Coupling
  - Client Resiliency:
    - HTTP Stability Patterns
    - Notifications
  - Server Resiliency
    - Protecting Providers
    - Throttling
  - Cohesive Framework
- Chaos Engineering

# REST Topics

# Plural Resources

The resource should always be **plural** in the API endpoint and if we want to access one instance of the resource, we pass the id in the URL.

- method GET path /companies should get the list of all companies
- method GET path /companies/34 should get the detail of company 34
- method DELETE path /companies/34 should delete company 34

If we have resources under a resource, e.g Employees of a Company, then example API endpoints would be:

- GET /companies/3/employees should get the list of all employees from company 3
- GET /companies/3/employees/45 should get the details of employee 45, which belongs to company 3
- DELETE /companies/3/employees/45 should delete employee 45, which belongs to company 3
- POST /companies should create a new company and return its details

# Use Two URLs per Resource

One URL for:

- the collection and
- one for a certain element

- /employees #collection URL
- /employees/56 #element URL

- To add a new resource POST to the collection

# Nouns and Verbs

Use Nouns for Resources

Use Verbs for Non-Resource Responses

- Sometimes a response to an API call doesn’t involve resources (like calculate, translate or convert).
- Example:
  - GET /translate?from=de_DE&to=en_US&text=Hallo
  - GET /calculate?para2=23&para2=432

- URI's identify a resource, they should not indicate *what* we're doing to that resource.
  - That’s where HTTP Operations come

# Path Vs Query

A REST API can have parameters in at least two ways:

- **As part of the URL-path** (i.e. /api/resource/parametervalue )
- **As a query argument** (i.e. /api/resource?parameter=value )

The URI standard states the path is for hierarchical parameters and the query is for non-hierarchical parameters.

1. **Path variables** are used for the direct action on the resources, like a contact or a song ex..
   GET etc /api/resource/{songid} or
   GET etc /api/resource/{contactid} will return respective data.
2. **Query perms/argument** are used for the in-direct resources like metadata of a song eg
   GET /api/resource/{songid}?metadata=genres return the genres data for that particular song.

# Searching, Sorting, Filtering – using the URL Query (?)

**Sorting**:
Similar to filtering, a generic parameter sort can be used to describe sorting rules. Accommodate complex sorting requirements by letting the sort parameter take in list of comma separated fields, each with a possible unary negative to imply descending sort order. For example,
***GET /groups?sort=status,-name*** *– Returns list of groups in ascending order of status; Within the same status, groups returned will be sorted by name in descending order*

**Filtering**
Use a unique query parameter for each field that implements filtering.
***GET /groups?status=active*** *– Returns a list of active groups*

**Searching:**
Sometimes basic filters aren't enough and you need the power of full text search. When full text search is used as a mechanism of retrieving resource instances for a specific type of resource, it can be exposed on the API as a query parameter on the resource's endpoint. Let's say q. Search queries should be passed straight to the search engine and API output should be in the same format as a normal list result.
***GET /groups/?status=active&sort=-name&q=test*** *– Return list of all active groups sorted by name (desc) which contain 'test' in their names*

# Search as a Resource

One way to implement a RESTful search (or filter) is to consider the search (or filter) itself to be a resource.

- Then you can use the POST verb because you are creating a search.
- You do not have to literally create something in a database in order to use a POST.

For example:

POST http://example.com/people/searches

```
{
      { "terms":  {
            ”id": "123456789"
      },
      "order":
      { ...
      },
... }
```

# Get has no BODY

Complex queries could use the GET Body to include a JSON search representation. But it is recommended not to use the body on GET. Postman does not even allow it to be specified. However Elasticsearch API uses the Get body and argues:

The RFC for HTTP 1.1 RFC2616 says this:

> *A server SHOULD read and forward a message-body on any request; if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.*

The *recommendation* (note: "SHOULD") is that a GET body shouldn't have any effect on server processing.

But GET is a better semantic fit for search than POST:

> *In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.*

Also mentioned in the spec:

> *A cache or origin server receiving a conditional request, other than a full-body GET request, MUST use the strong comparison function to evaluate the condition.*

# Googles' Model

While a simple search could be modeled as a resourceful API, eg., **dogs/?q=red**, a more complex search across multiple resources requires a different design.

**Global search**

/search?q=fluffy+fur

Here, search is the verb; **?q** represents the query.

**Scoped search**

To add scope to your search, you can prepend with the scope of the search. For example, search in dogs owned by resource ID 5678

/owners/5678/dogs/search?q=fluffy+fur

**Formatted results**

For search or for any of the action oriented (non-resource) responses, you can prepend with the format as follows:

/search.xml?q=fluffy+fur

# Creating (and storing) actual Resources for Searching

Consider treating the set of possible queries as a collection resource, e.g. /jobs/filters, and create new entities as first class resources.

POST request entities to this resource, with the query parameters in the body, will either create a new resource or identify an existing equivalent filter and return a URL containing its ID:

/jobs/filters/12345.

The id can then be used in a GET request for jobs:

/jobs?filter=12345.

Subsequent GET requests on the filter resource will return the definition of the filter.

**Advantage**: it frees you from the query parameter format for filter definition, potentially providing you with more power to define complex filters. OR conditions are one example that I can think of that are difficult to accomplish with query strings.

**Disadvantage**: you lose readability of the URL (although this can be mitigating by retrieving the definition though a GET request for the filter resource).

# Complex Query Parameters

Passing JSON Objects as a parameter. Complex search statements could be created in JSON, and passed as a parameter with GET , rather than in the body with POST.

- This gets messy and should use URL encoded string.
- Other coded string could be used. However it gets hard to interpret:
  - /cars;color-blue+doors-4+type-sedan
  - /garage[id=1-20,101-103,999,!5]/cars[color=red,blue,black;doors=3]
  - /cars[?;]color[=-:]blue[,;+&], *

**ODATA:**

OData queries operate as a GET call but allow you to restrict the properties which are returned and filter them.

Tokens such as $select= and $filter= are used to build ‘complex’ statements.

A URI would look something like this:

/users?$select=Id,Name$filter=endswith(Name, 'Smith')

# HTTP SEARCH Method – The Draft

"Using a GET request with some combination of query parameters included within the request URI is arguably the most common mechanism for implementing search in web applications."

- Implementations are required to parse the request URI into distinct path (everything before the '?') and query elements (everything after the '?').
- The path identifies the resource processing the query (in this case 'http://example.org/ feed') while the query identifies the specific parameters of the search operation.

Issues:

- Query expressions included within a request URI must either be restricted to relatively simple key value pairs or encoded such that the query can be safely represented in the limited character- set allowed by URL standards. Such encoding can add significant complexity, introduce bugs, or otherwise reduce the overall visibility of the query being requested.
- While most modern browser and server implementations allow for long request URIs, there is no standardized minimum or maximum length for URIs in general.
  - Using the payload provides a way aroudn this but GET does not recommend using the payload.
- The client has not specified to the server that a SEARCH Operation is being requested.

# HTTP SEARCH Method - Draft

"As an alternative to using GET, many implementations make use of the HTTP POST method to perform queries, as ……. In this case, the input parameters to the search operation are passed along within the request payload as opposed to using the request URI."

Issues:

- Not specified that a search Operation is being requested. Indeed POST is usually used to create a resource.
- An HTTP Search Operation has been suggested and defined by RFC5323 to get around these issues of GET and POST
- Messes with caching; as search is probably transient

"The SEARCH method is used to initiate a server-side search. Unlike the HTTP GET method, which requests that a server return a representation of the resource identified by the effective request URI (as defined by [RFC7230]), the SEARCH method is used by a client to ask the server to perform a query operation (described by the request payload) over some set of data scoped to the effective request URI. The payload returned in response to a SEARCH cannot be assumed to be a representation of the resource identified by the effective request URI."

# Aliases and Field Selection

**Aliases for common queries**

To make the API experience more pleasant for the average consumer, consider packaging up sets of conditions into easily accessible RESTful paths. For example, when querying for mostactive,recommended groups etc, we can have endpoints like

**GET /groups/mostactive** – *Returns list of mostactive groups*

Default values can be used for the sort parameters.

**Field selection**

The API consumer doesn't always need the full representation of a resource (mobile clients for example). The ability select and chose returned fields goes a long way in letting the API consumer minimize network traffic and speed up their own usage of the API. Use a fields query parameter that takes a comma separated list of fields to include.

**GET /groups/?fields=id,name,owner,status&status=active&sort=-name**

# Pagination

The right way to include pagination details today is using the Link header introduced by RFC 5988. An API that uses the Link header can return a set of ready-made links so the API consumer doesn't have to construct links themselves.

**GET** /groups?offset=20&limit=20

The response should include pagination information through links as below:

```
{
        "start": 1,
        "count": 20,
        "totalCount": 100,
        "totalPages": 5,
        "links": [
                {
                        "href": "https: //<url>/offset=40&limit=20",
                        "rel": "next"
                },
                {
                        "href": "https: //<url>/offset=0&limit=20",
                        "rel": "previous"
                }
        ]
}
```

# Error Handling

**Error Response**

An API should provide a useful error message in a known consumable format in case of errors in API's.

Both successful and error response can be handled using the HTTP Status codes.

However, the representation of an error should be no different than the representation of any resource, just with its own set of fields. A JSON error body should provide a few things for the developer – a useful error message, a unique error code (that can be looked up for more details in the docs) and possibly a detailed description. JSON output representation for something like this would look like:

```
{
    "code" : 1234,
    "message" : "Something bad happened :(",
    "description" : "More details about the error here"
}
```

# API Evolution & Versionng

# Service Evolution Patterns

**Single Message Argument** - start with one argument, then 2, then 3, then one big object does everything. One Giant JSON;

**Dataset Amendment** - points in schema where you can add things.

**Tolerant Reader** – take only what you need;

**Schema Versioning** – Major, Minor, Patch;

**Extension Endpoints** - UserField1, UserField2....

**Consumer Driven Contract**

# Tolerant Reader

The governing law for decoupled collaboration should be Postel's Law:

*“be conservative in what you do, be liberal in what you accept from others.”*

Jon Postel

Tolerant Reader is an integration pattern that helps creating robust communication systems. The idea is to be as tolerant as possible when reading data from another service. This way, when the communication schema changes, the readers shouldn’t  break.

# Tolerant Reader

Use the Tolerant Reader pattern when:

- the communication schema can evolve and change and yet the receiving side should not break
- Good for JSON – the client should only read what it needs, no more.
- Evolving APIs – only use the fields and object you need. In this way objects and fields can be added without impacting existing consumers.

Extract only what is needed from a message and ignore the rest.

- Rather than implementing a strict validation scheme, make every attempt to continue with message processing if/when potential schema violations are detected.
- Exceptions are only thrown when the message structure prevents the reader from continuing, or the content clearly violates business rules.

Ignore new message items, the absence of optional items, and unexpected data values as long as this information does not provide critical input to the service logic.

# Contract Types – Martin Fowler

| Contract | Open | Complete | Number | Authority | Bounded |
|---|---|---|---|---|---|
| Provider | Closed | Complete | Single | Authoritative | Space/time |
| Consumer | Open | Incomplete | Multiple | Non-authoritative | Space/time |
| Consumer-Driven | Closed | Complete | Single | Non-authoritative | Consumers |

# Semantic Versioning 2.0.0

MAJOR - version when you make incompatible changes

MINOR - version when you add functionality in a backward compatible manner

PATCH - backward compatible bug fixes.

# Provider Driven

As a Provider of a Service we can give our consumers a contract of what we expose, e.g. in our API Portal we list our services, and:

- A description of the business capability, the semantics;
- Interfaces, the syntax;
- Policy, more semantics;
- Schemas;
- Assumed NFRs as a SLA ( as distinct to our SLOs ).

These contracts are Closed, Complete, Bounded, Immutable for the time of a contract. Versioning attempts to allow for change and maintain contracts; but versioning is expensive.

How do we really know what consumers need and expect, and how do we test these hypotheses given the need to change, reducing the cost of maintaining version ?

# Consumer Driven Contracts

**Each Service Consumer supplies the Provider with an expectation of the service as test cases. The user EXPLICITLY provides what's to be tested.**

The Provider manages a test suite across all the consumers.

- Allows Provider to make changes independent of Consumer.
- As a result they can automatically check when change violate the contract with a consumer.
- Help stop building stuff you no one needs. Find out what the clients actually wants.

**Issue**: is it is up to consumers to understand their needs and full requirements and be able to communicate this to the provider via quality test cases.

# Contract - Team Responsibility

When components or services are developed continuously it is the responsibility of each team to not break those down stream;

- Other teams dependent on a down stream components are a customer;
- If a dependency is broken then communication and discussion, and then negotiation, is required;

Can this trigger be automated ?

- Consumers can pass a set of unit tests to producers, to ensure their contact is tested. When these break the need for a conversation is automatically triggered.

# Sprint Cloud Contract

Takes a Groovy contract and generates, via a maven plugin:

1) JUnit tests
2) Wiremoc stubs

for the consumer and provider.

WireMoc - tools to create mock servers flow to test. Mock up a response for a Request.

# Example Groovy Contract

```
org.springframework.cloud.contract.spec.Contract.make {
    description(""" the description """)
        request {
            method POST()
            url "/check"
            body( age: 22, name: “Fred" )
            headers {
                contentType(applicationJson())
            }
        }
        response {
            status 200
            body(""" { "status" : "OK" } """)
            headers {
                contentType(applicationJson())
            }
        }
    }
```

# Reliability

- Direct Connect HTTP is Not Reliable
- Platforms are Not Reliable

# Web Scale is about Tolerance

**“Operations at Web Scale is the ability to consistently create and deploy reliable software to an unreliable platform that scales horizontally.”** Jesse Robins

The governing law for decoupled collaboration should be Postel's Law:
***“be conservative in what you do, be liberal in what you accept from others.”*** Jon Postel

As architects we should not try to make platforms more reliable, it’s a venture doomed to fail, but make systems more tolerant of failure.

**“The microservice style of architecture is not so much about building individual services so much as it is making the *interactions between* services reliable and failure-tolerant.** While the focus on these interactions is new, the need for that focus is not. We’ve long known that services don’t operate in a vacuum. **Even before cloud economics, we knew that - in a practical world - clients should be designed to be immune to service outages.** The cloud makes it easy to think of capacity as ephemeral, fluid. **The burden is on the client to manage this intrinsic complexity.”**

# HTTP is Poor for Service Integration

HTTP is a pretty limited medium for service integration:

- It is inherently fragile with little provision for fault tolerance;
- no ‘standard’ support for server-initiated or peer-to-peer communications (yet);
- nothing to help you manage transactions or long-running processes;
- no delivery guarantees;
- no provision for any buffering to help smooth out spikes in demand.

# HTTP is not Reliable

In general HTTP suffers from the '**eight fallacies of distributed systems'** and so there is no guarantee a Request will be delivered or followed by a Response, and unless the Response is received it is not known if the Request ever arrived.

- Requests can be retried until a Response is received except for an HTTP POST, that is not idempotent (e.g. an order submission);
  - This is a significant problem for a system relying on HTTP Point to Point calls.
- Multiple retries may mean that multiple new resources are created.

Idempotent HTTP Requests, such as PUT (e.g. an order update) can still be affected by client, network, or server failures and retries are required. If a response is not received and processed by the client then it is not clear that, for example, an update was received or processed.

**Other Issues:**

Another source for failure can be that the Provider system is overloaded and cannot process the messages it is receiving, it times out. This can cascade to clients that are continually sending requests.

A further problem is that a response is received with an Error Code, resulting is repeated retries, wasting resources, due to a server issues. This may also bring down clients;

# Solution Mechanisms to Provide Quality – Stability Patterns

- Retries
- Timeouts
- Circuit Breakers
- Fallbacks
- BulkHeads
- Load Shedding
- Creative Routing
- Client Side Load Balancing
- Service Discovery
- Asynchronous messaging
- Fail Fast and Replace
- Throttling
- Guids and Prototypes

Most of these components are very complex, difficult and expensive to implement by hand. A framework is essential.

These solution components are discussed in other presentations. Unless they are implemented as a complete approach new issues arise, PTO….

# The Recovery-Oriented Computing (ROC)

**Berkley and Stanford research project.** The founding principles are as follows:

- Failures are inevitable, in both hardware and software.
- Modeling and analysis can never be sufficiently complete.
- *A priori* prediction of all failure modes is not possible.
- Human action is a major source of system failures.

Their research runs contrary to most work in system reliability. Whereas most work focuses on eliminating the sources of failure, ROC accepts that failures will inevitably happen.

Their work aims to improve survivability in the face of failures.

# Throttling

# Rate Limiting - Throttling

To prevent abuse, it is standard practice to add some sort of rate limiting to an API. RFC 6585 introduced a HTTP status code 429 Too Many Requests to accommodate this.

- The 429 status code indicates that the user has sent too many requests in a given amount of time ("rate limiting").

However, it can be very useful to notify the consumer of their limits before they actually hit it. This is an area that currently lacks standards but has a number of popular conventions using HTTP response headers.

At a minimum, include the following headers:

- X-Rate-Limit-Limit- The number of allowed requests in the current period
- X-Rate-Limit-Remaining – The number of remaining requests in the current period
- X-Rate-Limit-Reset – The number of seconds left in the current period

# Why do you need Throttling ?

- You can deliver consistent applications by making sure that a single client is not suffocating your applications. Enhanced performance will drastically improve the end-user experience.
  - Enforce license agreements on usage
- You can control user authentication and access by rate limiting APIs at various levels - resource, API, or application.
- You can design a robust API that can be leveraged by multiple groups based on their access level. Simplified API monitoring and maintenance can help reduce your costs.
- APIs are a gateway to your backend resources and throttling offers you an extra layer of protection for those resources.

- **This is a business feature not a technology mandate. If users don't need it then remove *it.***

- Security and DDOS attacks are often not a good reason and other solutions (WAF) exist to solve this problem

# Types Of Throttling

- **Rate-Limit Throttling**: (API burst limit or the API peak limit). Allow requests to pass through until a limit is reached for a time interval. A throttle may be incremented by a count of requests, size of a payload, or it can be based on content; for example, a throttle can be based on order totals. This is also known as the **IP-Level Throttling**: Restrict access to a certain list of whitelisted IP addresses. You can also limit the number of requests sent by a certain client IP.
- **Scope Limit Throttling**: Based on the classification of a user, you can restrict access to specific parts of the API - certain methods, functions, or procedures. Implementing scope limits can help you leverage the same API across different departments in the organization.
- **Concurrent Connections Limit**: An application cannot respond to more than a certain number of connections. Need to limit the number of connections from a user/account to make sure that other users don't face a DoS (Denial of Service) error. This kind of throttling also helps secure your application against malicious cyberattacks.
- **Resource-Level Throttling** (Hard Throttling): If a certain query returns a large result set, you can throttle the request so that your SQL engine limits the number of rows returned by using conditions attributes like TOP, SKIP, SQL_ATTR_MAX_ROWS, etc.

# Where to Throttle

- **Tiers of Throttling**: Throttling can be applied at multiple levels in your organization:
  - API-level throttling at the Edge – API Gateway.
  - Application-level throttling – each application protects itself.
  - MicroService-level throttling – each microservice protects itself.
  - User-level throttling.
  - Account-level throttling;
  - Load balancer throttling - but can't differentiate clients.

# MicroService Level Throttling

- Service should manage themselves. Adding Throttling at an API Gateway adds another dependency. How do we know who the Gateway is protecting when it does this work?
  - If there is a chain of service calls, and at some point rate limiting is required, doing this right up the front hides the reason from view, bad practice as loose traceability.
  - When a services load gets too great it should throttle, not the Gateway. We know the reason and why.

- Assume the load balancer distributes load equally. So will work across a cluster of microservices. When Auto-scaling the number of nodes needs to be factored into the allowed rate per microservice. So this number needs to be available,
  - e.g. AWS has an API to share the number of nodes in a Auto-scaling group.
- To make it user/client specific , need to pass client IDs in header;
  - Need to configure each clients loads per microservice.
  - This does introduce another dependency.

The first Time Throttling kicks in and Customer are Kicked out

The product Team will want Throttling Removed.

# Microservice Colaboration

# Microservices Collaborate

There are 3 main ways Microservices can collaborate**:**

- **Commands** - make other microservices perform actions;
- **Queries** - to get information from other microservices;
- **Events** - allow other microservices to react asynchronously to changes in another microservice, after the happening of interest.

# Coupling and Microservices

4 main alternative models:

| | Events | Query/Command |
|---|---|---|
| Request-Response (synchronous) | RR-E | RR-C |
| Publish-Subscribe (asynchronous) | PS-E | PS-C |

- **Events** only state what has happened and do not describe what should happen. They are observed. The performer doesn't need to know about observers.
- **Queries/Commands** convey what should happen. Queries are a specific requests for information. Commands are a specific requests to take some action.

# RR-E : Request Response with Events

Service A asks Service B for the events that happened, so it can complete its work, e.g. Shipping Service asks Order Service if there are any new orders to ship ?

Two dependencies:

- Service A must know that Service B exists and its semantics to call it to get the events.
- Runtime dependency, as Service B must be available for Service A to complete.
- Service B doesn't need to know anything about Service A.

Note the Observer Pattern

# RR-C : Request Response with Commands

Service A asks Service B to do something, run a command or query, so it can complete its work, e.g. Order Service tells Shipping Service to Ship Order?

Strong Coupling - Two dependencies:

- Service A depends on Service B to run.
- The business service is baked into the architecture;
- Runtime dependency, as Service B must be available for Service A to complete.
- Service B doesn't need to know anything about Service A.

Note Command Pattern

# PS-E : Publish Subscribe with Events

Service A says, to intermediary, its interested in events related to Service B for the events that happened, e.g. Shipping Service says I am interested to know about Order information, such as newly created or completed orders?

Dependencies are reduced:

- Service A is loosely coupled with Service B, via an intermediary.
- Service B doesn't need to know anything about Service A.
- However Service A may need to keep track of events from B, e.g. shipping service needs to be able to work out when it can ship a new order.
- New dependency on intermediary technology.

# PS-C: Publish Subscribe with Commands

Service A registers itself as being able to carry out certain commands or queries with intermediary.  Service B is a consumer of those commands or queries. e.g. Shipping service registers itself as being able to ship stuff, the Order service uses this to ship ?

One dependency:

- Service A doesn't know about Service B.
- Service B doesn't know who will do its work.
- However, Service B knows that some service has to do its work.

# Who Wins ?

- **Request-Response patterns** imply a strong runtime coupling;
- **Command/Query patterns** imply one service knows what another can do – not runtime but functional coupling;
- **Publish-Subscribe with Events** is the one pattern where both services do not need to know about the other at runtime or functionally.

# Unreliable Synchronous HTTP

# Using GUIDs

When a client POSTs an HTTP Request that is not **followed by Response** the only option is to ask the server what happened. However, to do this you need a key, something that uniquely identifies the resource. Given a reply was not received the client does not yet know the ResourceID, supplied by the provider.

Idempotence may seem like a solution (post the item again) but then this requires a unique way to identify a client order. The client has to manage their order IDs.

Options:

**1.Client Key:** The client provides a key. In practice clients are very bad at managing unique keys, and should not be relied on. The solution may also require complex policy around the timeliness of their key, and this policy needs to be shared by the client with the provider, creating unwanted coupling.

**2.Server key:** The provider provides a GUID service API used to identify new transactions, Before submitting the resource the client asks the provider for a system unique GUID;

1.The GUID is supplied in the next POST request and used after the request to query the server for the status of the transaction;

2.If the resource does not exist, using the GUID, then the POST can be resubmitted;

3.If the resource does exist, the client can obtain the actual resource link, and finalise the transaction. Temporary resources are cleaned up.

# Using Prototypes

An alternative to a GUID is to use a Prototype resource.

1. The client POSTs a resource, however it is not a final resource; a prototype.
2. The client can repeatedly submit requests until a response is received. The response has a link to the last resource. Prior resources exist but we have no link to them, they are now irrelevant;
3. If the resource does exist, then it now has a unique provider supplied ID;
4. The client can then do a PUT to change the resource to a ‘real’ actual resource, e.g. changing Status from ‘temp’ to ‘submit’.
5. A link to the actual resource is returned.
6. The non used prototypes, are cleaned up, using some timing policy.

Now what happens if the PUT fails (timeout or network error) ?

As PUTS are idempotent and we have a key just retry until we get a link to the final resource.

# Problems with PUTS - Updates

Although PUTS are idempotent a client still has a problem if they receive an error message, no response at all, a timeout, a lost connection or their client system has a fault.

If a circuit breaker opens, client input requests may be waiting with no connection to the backend service. There is now, no chance to submit.

In these cases the client may not know if the update was received and processed. Without a unique ID for the Update itself (not the target resource) the client cannot check that the resource actually changed, unless they do a ‘diff’ !

For a system driven by a user in front of a GUI, it may be OK to tell them to try again later, however, this is not a good experience. Caching their input, until the backend is again available, is a better approach.

All these issues also apply to POSTS.

# Problems with PUTS - Solutions

Apart from doing a ‘diff’.

1. Use prior GUID approach with an Update POST resource; that is a command to make a change;
2. As above but use a prototype;
3. Keep retrying until you get a response, as PUT is idempotent. This is OK if the client stores this process in memory and itself does not die, as when it comes up it won’t know what it was doing.
4. Use a Client Backlog to store requests, more resiliently, and use a separate thread or process to submit them.

Complexities:

- When requests are successfully submitted, remove them from the Backlog. The Backlog could be an internal queue or data base table.
- These may need to be wrapped in a transaction.
- Long outages may require halting client processing; circuit breaker.

# Async Order Processing via HTTP Redirects

Another approach is to do as little as possible, no processing, until you get back to the client, as soon as possible hopefully with a preliminary response.

1. URL Request - Order Submit:
   - Reply: 202 – app stores order but not processed, returns a link URL to a Status resource for this order;
2. URL Request - Status:
   - Reply: 200 – Returns the Status, e.g. 'In Progress', with a completion time: e.g. 1 Hour.
   - Keep trying (Polling) until….
3. URL Request - Status:
   - Reply: 303 – Now its no longer in progress, return a message that this is the wrong resource. Provide a new URL Link to the now created Order Resource.
4. URL Request - Order :
   - Reply: 200 – get the new processed order. Now we get the new Order ID.

# Use a Web Hook ?

The client receives an HTTP call back about the status of the order.

However, the is no ID to use to link the callback to the original POST.

# Example - Circuit Breaker

Step 1) When a remote service is called the circuit breaker will monitor the call. If the calls take too long, the circuit breaker will intercede, ***timeout***, and "kill" the call.

Step 2) In addition, the circuit breaker will monitor all calls to a remote resource and if enough calls fail, the circuit break implementation will pop, "failing fast" and preventing future calls to the failing remote resource.

A new issue:

1) The first business process was 'killed', possible terminated forever. Some mechanisms is required now to remember this state and then ***retry*** it later. The more retries the more resources are wasted and at some point this has to stop. Or Instead processing goes on but ***falls backs*** to an alternate business process.
2) Issues for 1 are now magnified across many consumers. Mechanisms are needed to repair and ***replace*** the remote service seamlessly from the consumers POV, and then reestablish the many clients state. Otherwise cascading failures will continue to percolate.

- ***Bold Italics*** – there is no way to do this without dependency on other solution components – it's a set !

# KAFKA Reliable Messages

# How to Submit Data Reliably

Kafka is normally used to process events in the form of ‘Notifications’.

However it can be used to send all the data required, a ‘Message Pattern’.

# Using the Framework

Spring Cloud will repeatedly retry requests, based on a number of parameters, this is built into:

- Zuul
- Feign
- REST templates

Need to know the idempotence of the API, a timeout could have started the target process. Getting no response does not mean a POST failed.

# Auto Retry

What does Auto retry mean with a POST ?

```
<dependency>
        <groupId>org.springframework.retry</groupId>
        <artifactId>spring-retry</artifactId>
        <version>1.1.4.RELEASE</version>
</dependency
```

# Chaos Engineering

# Chaos Engineering

*Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.*

*Principles of Chaos*

These experiments follow four steps:

1. Start by defining 'steady state' as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

# Principles - 1

**Build a Hypothesis around Steady State Behavior**

Focus on the measurable output of a system, rather than internal attributes of the system. Measurements of that output over a short period of time constitute a proxy for the system's steady state. The overall system's throughput, error rates, latency percentiles, etc. could all be metrics of interest representing steady state behavior. By focusing on systemic behavior patterns during experiments, Chaos verifies that the system *does* work, rather than trying to validate *how* it works.

**Vary Real-world Events**

Chaos variables reflect real-world events. Prioritize events either by potential impact or estimated frequency. Consider events that correspond to hardware failures like servers dying, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event. Any event capable of disrupting steady state is a potential variable in a Chaos experiment.

# Principles - 2

**Run Experiments in Production**

Systems behave differently depending on environment and traffic patterns. Since the behavior of utilization can change at any time, sampling real traffic is the only way to reliably capture the request path. To guarantee both authenticity of the way in which the system is exercised and relevance to the current deployed system, Chaos strongly prefers to experiment directly on production traffic.

- One difference to 'testing'.

**Automate Experiments to Run Continuously**

Running experiments manually is labor-intensive and ultimately unsustainable. Automate experiments and run them continuously. Chaos Engineering builds automation into the system to drive both orchestration and analysis.

**Minimize Blast Radius**

Experimenting in production has the potential to cause unnecessary customer pain. While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineer to ensure the fallout from experiments are minimized and contained.

# Chaos Monkey - Netflix

**Chaos Monkey** is a service which identifies groups of systems and randomly terminates one of the systems in a group. The service operates at a controlled time (does not run on weekends and holidays) and interval (only operates during business hours).

- In most cases we have designed our applications to continue working when a peer goes offline, but in those special cases we want to make sure there are people around to resolve and learn from any problems.
- With this in mind Chaos Monkey only runs in business hours with the intent that engineers will be alert and able to respond.

It is part of the 'SimianArmy' at Netflix, which also includes:

- **Conformity Monkey** is a service which runs in the Amazon Web Services (AWS) cloud looking for instances that are not conforming to predefined rules for the best practices.
- **Janitor Monkey** is a service which runs in the Amazon Web Services (AWS) cloud looking for unused resources to clean up.
- **Chaos Gorilla** brings down a whole Availability Zone
- Etc….

# Open Source Chaos Tools

**Simoorg** LinkedIn's own failure inducer framework. It was designed to be easy to extend and most of the important components are pluggable.

**Pumba** A chaos testing and network emulation tool for Docker.

**Chaos Lemur** Self-hostable application to randomly destroy virtual machines in a BOSH-managed environment, as an aid to resilience testing of high-availability systems.

**Chaos Lambda** Randomly terminate AWS ASG instances during business hours.

**Blockade** Docker-based utility for testing network failures and partitions in distributed applications.

**Chaos-http-proxy** Introduces failures into HTTP requests via a proxy server.

**Monkey-ops** Monkey-Ops is a simple service implemented in Go, which is deployed into an OpenShift V3.X and generates some chaos within it. Monkey-Ops seeks some OpenShift components like Pods or DeploymentConfigs and randomly terminates them.

**Chaos Dingo** Chaos Dingo currently supports performing operations on Azure VMs and VMSS deployed to an Azure Resource Manager-based resource group.

**Tugbot** Testing in Production (TiP) framework for Docker.