Modernizing systems with Microservices, Hystrix and RxJava

Holger Kraus, Arne Landwehr

Berlin Expert Days, Berlin, Sep 18, 2015
A typical system
The context
Current problems

- Maintenance is difficult
- New features need a lot of time
- Very unstable
- Outdated technology
- Doesn’t scale

+ frustrated developers :(
Stabilize first!
External dependencies

Here!

Here!

Here!

And also here!
Cascading Failures
Stability patterns

- Timeouts
- Circuit Breaker
- Bulkheads

... Fail Fast, Steady State, Handshaking, Test Harness, Decoupling Middleware
Timeout
Bulkheads

https://www.flickr.com/photos/10413717@N08/6935206524
Ships

Bulkheads and IT

- Thread pools
- Database connection pools
- Instances
- Server
- Data center
Circuit Breaker

http://martinfowler.com/bliki/CircuitBreaker.html
Hystrix

- Library from Netflix
- Resilience Library
- Command Pattern
- Metrics
- Dashboard
Use Cases

let's pick this one!
Search Products

1. search products
2. find external products
3. find internal products
4. return internal + external products
Search Products

```
$ http GET http://monozon:8080/products | jq '.
[
  "internalProduct_1",
  "internalProduct_2",
  "merchant_1",
  "merchant_2",
  "merchant_3",
  "merchant_4"
]
```
Call without Hystrix

```java
private List<Product> findProducts(String query) {
    ClientResponse clientResponse =
        Client.create()
            .resource("http://merchant1/products/")
            .QueryParam("query", query)
            .get(ClientResponse.class);

    return toProduct(clientResponse);
}
```

cascading failures incoming!
Simple Command

```java
public class GetMerchant1Products extends HystrixCommand<List<Product>> {

    private final String query;

    public GetMerchant1Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant1"));
        this.query = query;
    }

    @Override
    protected List<Product> run() throws Exception {
        return findProducts(query);
    }
}
```
Execute it!

```java
public List<Product> findExternalProducts(String query) {
    List<Product> productList =
        new GetMerchant1Products(query).execute();

    List<Product> merchant2Products =
        new GetMerchant2Products(query).execute();

    productList.addAll(merchant2Products);

    return productList;
}
```
Execute it asynchronously

```java
public List<Product> findExternalProducts(String query) {
    Future<List<Product>> merchant1ProductsFuture =
        new GetMerchant1Products(query).queue();
    Future<List<Product>> merchant2ProductsFuture =
        new GetMerchant2Products(query).queue();

    List<Product> productList = new ArrayList<>();
    productList.addAll(merchant1ProductsFuture.get());
    productList.addAll(merchant2ProductsFuture.get());

    return productList;
}
```
Fallback

```java
@Override
protected List<Product> run() throws Exception {
    return findProducts(query);
}

@Override
protected List<Product> getFallback() {
    return Collections.emptyList();
}
```
In case Merchant 2 is down:

```bash
→ / http GET http://monozon:8080/products | jq '.
[
  "internalProduct_1",
  "internalProduct_2",
  "merchant_1",
  "merchant_3",
  "merchant_4"
]
```
The stabilized system
Demo
And now?
Current problems

- Maintenance is difficult
- New features need a lot of time
- Very unstable => enables further distribution
- Outdated technology
- Doesn’t scale
Monozon is a typical Monolith

- hidden dependencies
- module boundaries are not clear
- distributed business processes
- just one technical platform

Everything depends on everything
We need a clear cut!
Our mission:

Creating smaller systems that are

‣ understandable
‣ enhanceable
‣ have clear boundaries and responsibilities
‣ allow technological diversity
The typical cut

UI

Business logic

Persistence
Vertical Services, one UI
Self contained Systems
Micro architecture

- Separate (redundant) persistence
- Internal, separate logic
- Domain models & implementation strategies
- Separate UI
- Separate development and evolution
Domain architecture

- System boundaries reflect business dependencies
- defines who is responsible for which data
- follows the principle of
  - loose coupling
  - high cohesion
From internal structure
To autonomous systems

monozon.com

Search  Inventory  Orders  Customers
Recommendation  Products  Payment  Distribution
Ask the monolith

- identify Bounded Contexts
- define boundaries explicitly
- The experience with the monolith helps to create clear system boundaries
Connect!

https://www.flickr.com/photos/npobre/8189066572
Macro architecture

- defines standards across systems
  - UI integration
  - communication protocols
  - representation formats
  - data redundancy
  - logging, monitoring, security
Smart endpoints, dump pipes!

Martin Fowler, James Lewis

http://martinfowler.com/articles/microservices.html
Consequences

- Transactions contexts are bound to just one system
- Data should be just changed by the systems that are responsible for it
- Processes need Data that are spread over various systems
Connecting data
Synchronous vs. Asynchronous
Time for RxJava!

- Reactive Extensions for the JVM
- Asynchronous streams
- Elements of
  - Iterator pattern
  - Observable pattern
  - Functional programming
Iterable pull

T next() throws Exception
returns;

Observable push

onNext(T)
onError(Exception)
onCompleted()
RxJava in one picture

Create streams

```java
public class GetMerchant2Products
        extends HystrixObservableCommand<Merchant2Product> {

    @Override
    protected Observable<Merchant2Product> construct() {
        return Observable.from(findProducts(query));
    }

    @Override
    protected Observable<Merchant2Product> resumeWithFallback() {
        return Observable.empty();
    }
}

Doesn't work for us!!!
```
public class GetMerchant2Products
    extends HystrixCommand<List<Merchant2Product>> {

    private final String query;

    public GetMerchant2Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant2"));
        this.query = query;
    }

    @Override
    public List<Merchant2Product> run() {
        return findProducts(query);
    }
}
Converting into a stream

```java
Observable<List<Merchant2Product>> productM2ListObservable = new GetMerchant2Products(query).observe();

Observable<Merchant2Product> productM2Observable = productM2ListObservable.flatMap(Observable::from);
```
Transforming with map

http://reactivex.io/RxJava/javadoc/rx/Observable.html
map in action

```javascript
Observable<SearchResult> searchResultM20observable = productM20observable.map(\this::toSearchResult);
```
Combining with merge

http://reactivex.io/RxJava/javadoc/rx/Observable.html
merge in action

```java
Observable<SearchResult> mergedSearchResultObservable =
    searchResultM1Observable.mergeWith(searchResultM2Observable);
```
Collecting details withflatMap

http://reactivex.io/Rxjava/javadoc/rx/Observable.html
Combining streams with zip

http://reactivex.io/RxJava/javadoc/rx/Observable.html
flatMap and zip in action

```java
public Observable<SearchResult> findInternalProducts(String query) {

    Observable<Long> productIndexObservable = 
        productId.findProducts(query);

    return productIndexObservable.flatMap(productId -> { 

        Observable<Product> productObservable = 
            retrieveProductFromProductSystem(productId);

        Observable<Long> quantityObservable = 
            retrieveQuantityFromInventoryService(productId);

        return Observable.zip(productObservable, 
                quantityObservable, SearchResult::new);

    });
}
Blocking streams

```java
public List<SearchResult> getSearchResults(String query) {
    Observable<SearchResult> searchResults =
        searchService.findInternalProducts(query)
            .mergeWith(searchService.findExternalProducts2(query));

    Iterator<SearchResult> searchResultIterator = searchResults.toBlocking().getIterator();

    return Lists.newArrayList(searchResultIterator);
}
```
Systems connected
How could you start?

https://www.flickr.com/photos/epemsl/8790814488
Our goal was to make you curious!
Find your own way!
Summary

‣ Use Hystrix to stabilize your system!
‣ Use RxJava to increase the amount of async/parallel processes in an easy way!
‣ Introduce Microservices to get control over your system again!
‣ Have fun :)
Thank you!

holger.kraus@innoq.com
arine.landwehr@innoq.com