# In Search of the Perfect Cloud Native Developer Experience

Daniel Bryant

## DevEx: 3 Components

* DevOps 🡪 Book: DevOps Handbook
* Lean Software Development
* UX 🡪 Book: The Design of Everyday Things

## The Ideal Workflow

Waterfall 🡪 Agile (“Mini Waterfall”) 🡪 Cross functional teams

## Progressive Delivery

* Feature Flags, Canary Release
* Monitor deployment of new features

Cross-cutting concerns gehören in eine zentrale Plattform!

## Necessary for microservices/Paas/K8s

Understand your domain

Problem is domain complex

Are you product/market fit?

## Arguments for serverless

Is your solution event-driven and simple?

Should you be adding value elsewhere?

## Tool für Infrastructure/Deployment as Code

Flux (in GitHub)

## Automate inner dev loop

* Draft
  + Automates “inner loop” build-push-deploy
  + Utilises Helm
* Gitkube
  + Automated build-push-deploy
  + Provides Heroku / CF like experience
* Skaffold
  + Automated build-push-deploy
  + Watches source code
  + Provides “dev” and “run” (CD) modes
* Tilt
  + Automates “inner loop” build-test-deploy
* Garden
  + Automates local build-push-test-deploy
* Helm (CNCF project)
  + Package manager for k8s
  + Deploy and manage (ready-made) charts
* Ksonnet
  + Define k8s manifests in jsonnet
  + Create composable config/services
* Telepresence (CNCF project)
  + Enables local-to-prod development

# Serverless Kickstarter with AWS Lambda

Lars Röwekamp

## Step1: IaaS

* Elastic Beanstalk
* ECS (Elastic Container Service)

## Step 2: PaaS

* RDS
* S3 Storage
* EC2

## Step 3: Business Logic as a Service

1. Off-the-shelf BL: Cognito (AWS service für Auth)
2. Domain specific BL: Serverless (Lambda) 🡪 FaaS

## FaaS with AWS Lambda

* Stateless code
* Max. runtime of a function in AWS: 300s
* Min. runtime to pay for: 100ms (fraction of a cent)
* For testability, the function itself should just call the real business logic
* You pay for runtime and configured max. memory. More memory will mean less runtime (and thus less money for runtime) until the application dependent memory footprint is needed. So it is necessary to optimize the memory config so that money paid (and runtime) is minimal.
* First start is a cold start and takes way longer than subsequent warm starts – thus being much more expensive. Amazon predicts time of week and day when starts are highly likely to prepare cold starts.

## Learnings

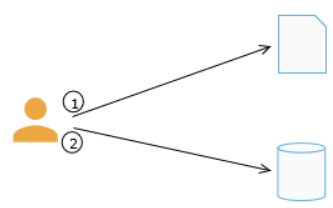
* Separate business code from infrastructure code
* Lambda function is the glue

# Domain Storytelling

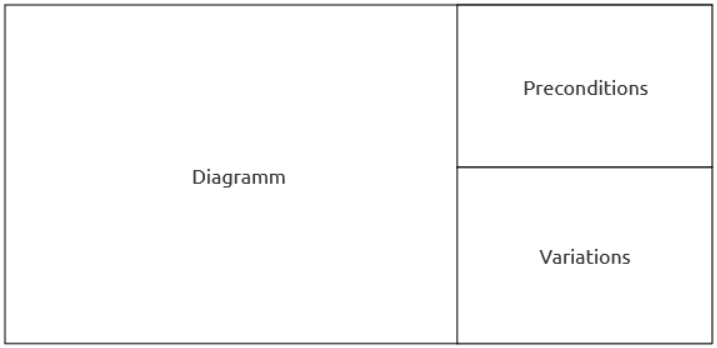
Henning Schwentner

## Introduction

* Active listening using concrete stories
* Pictographic: no if/switch/or



* Actors once, work objects several times



* 3 good examples are better than 1 bad abstraction
* Small models instead of one big model (leads to microservices)
* What we want to depict in the model depends on the context (compare atlas: different views on the world map depending on context)
  + Many models necessary, one for each subdomain
  + Leads to different classes with same name, e.g.:
    - class LeasingContract { public void sign() {} }
    - class LeasingContract { public void vote() {} }

## Bounded Context: indications for boundaries

* Different language
* One-way information flow
* Different usage of the same thing
* Different triggers (e.g. time vs. on-demand)
* Ask the domain experts

## Brown field process

When in the “brown field” (legacy systems, in contrast to green field)

1. Draw your “ideal” context map
2. Analyze your current architecture
3. Extract a supporting domain to learn (small subdomain)
4. Extract the/a core domain

## Modes

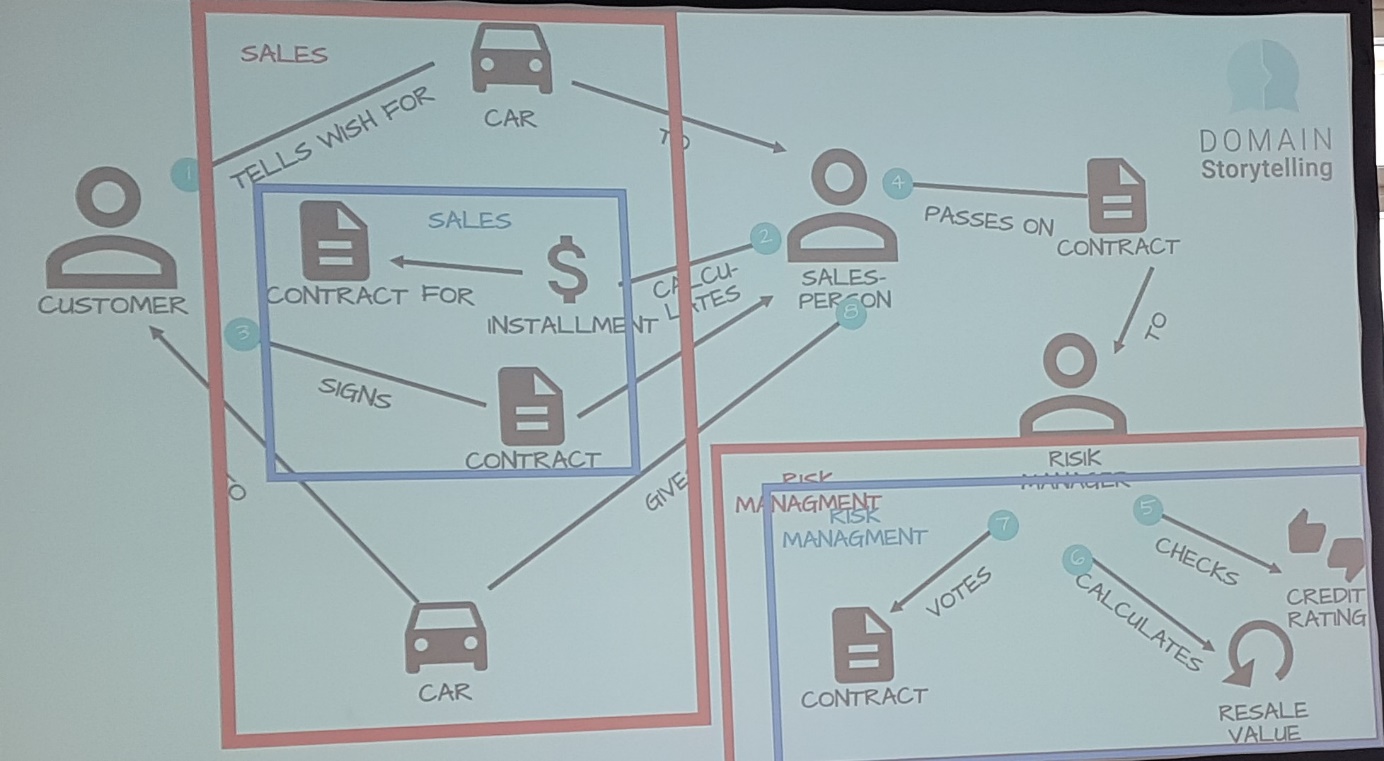
* Co-op (e.g. whiteboard, everybody gathered around)
* Moderated

## Tools

* <https://github.com/WPS/domain-story-teller>
  + <https://www.wps.de/modeler> (based on bpmn.io)
* BIC Cloud Design mit eigener Custom Methode
* Whiteboard

## Further reading

* <http://LeasingNinja.io> (coming…)
* <https://speakerdeck.com/hschwentner> (Vortrag war aus zwei Präsentationen zusammengeführt)



# Provisioning and deploying to serverless

Ivan Culjak

## Building blocks

* Container instances (ECS, Fargate)
* Kubernetes services + virtual nodes (EKS, Fargate)
* Functions (Lambda)

## Tools

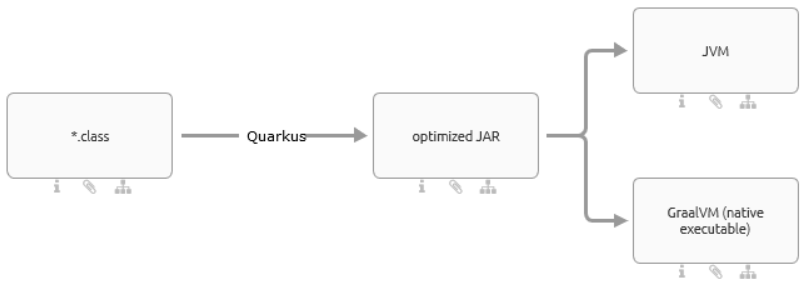
* Function CLI gesteuert installieren (Ansible, CloudFormation, Terraform)
* Pulumi
  + Cross Cloud for Functional/Serverless deployment
  + Azure, AWS, GCP, OpenStack, K8s, Pulumi Cloud Framework
* Integrate Pulumi by VCS push / CI/CD trigger
* Aber: Pulumi ist teuer (wen Pulumi as a Service genutzt wird; OpenSource 🡪 man kann es auch selbst installieren)

# Quarkus: Supersonic Subatomic Java

Peter Palaga

## Technology

* GraalVM statt JVM, eingebaut in ein Linux executable
* Kleines executable, geringer Memory Footprint, schneller Startup



## Compatibility & Features

* Java, Kotlin
* Mvn, Gradle
* Junit 4 & 5
* Diverse Frameworks (JPA, Hibernate, Bean Validation, CDI, …)
* Live reload

## GraalVM

* Ahead of time compilation
* Inkompatibel mit Runtime Magie (Reflection, dynamic Class loading, Annotations gehen nicht)
  + Aber: Reflection geht eingeschränkt, wenn vorher bei Quarkus registriert
* No Java debugging, no JMX, no JRebel, …
  + Aber: opt. JAR kann in JVM debuggt werden
* No finalize(), no InvokeDynamic
* No session scope (stateless CDI)

## Insights

* **Konkurrenz zu Spring Boot**
* **Dream Team mit serverless (no cold starts!!)**

## Resources

* <https://quarkus.io/get-started>

# Battle of the Circuit Breakers: Hystrix vs. Istio

Nicolas Frankel

* Reason for microservices: Teams do not scale 🡪 organizational reason (“2 Pizza team”)
* Conway’s Law

|  |  |  |
| --- | --- | --- |
| **Strategy** |  | **Framework/Tool** |
| Black Box | Fail Fast | Istio |
| White Box | Fallback  Understand business logic | Hystrix  Resilience4J |

* Problem of Istio: no fallback
* Problem of Hystrix:
  + Lot of configuration options (hard to fine-tune)
  + No big picture (e.g. “how many nodes are down?”
  + Discontinued, should be replaced by Resilience4J

## Resources

<https://blog.frankel.ch/>

<https://github.com/exoscale-labs/circuitbreaker>

<https://www.exoscale.com/syslog/istio-vs-hystrix-circuit-breaker/>

# Why FaaS is the least important part of Serverless

Eóin Shanaghy

## Modern software

* Lean
* Low cost experimentation
* Feedback loops
* Building the right Product
* Product discovery
* Continuous Deployment

## Drawbacks

* Monolith/Microservice cannot simply be “transformed” into serverless Lambdas.
* Lambda Feedback loop wastes time (Lambdas cannot be started on a local dev machine like a Container)
* Access management needs fine grained tuning

Lambdas are just one part of “Serverless” – also take the other services into account (Cognito, S3, Glue, Sagemaker, DynamoDB, Athena, Kinesis, API Gateway).

## Store with Intent

* Optimize for access – polyglot persistence
* Downsample
* Aggregate
* Keep raw data
* Iterate on your approach
* Keep code close to the data – remember the power of SQL

## Explicit Orchestration

E.g. AWS Step Functions (State machine)

## Implicit Orchestration

* Event-driven
* Isolated Steps
* Reactive

## What’s important

* Product
* Automation CI/CD
* Iteration speed
* Expendable components
* Managed services
* Everything that happens after deployment

# Zero Downtime Migration in Microservices Architecture

Alex Soto

## Deployment types

* Deploy Service by service
* Blue/Green Deployment
* Canary Release/Deployment
* Dark Launches
  + Duplicate the request to Blue and Green service version
  + New version does not send a response (just fire and forget)

## Istio

* Makes it possible to route traffic for dark launches
* Works as a sidecar to each service
* Comes with Prometheus by default

## Migrating a Sticky Session

* Use in memory replicated db
* One replica is used by service-v1, the other one by service-v2

## Persistence State

Ex.: Update column name

1. V1: reads/writes “Name”
2. V2: reads “Name”, writes “Fullname” and “Name”
   1. Migrate data
3. V3: reads “Fullname”, writes “Name” and “Fullname”
4. V4: reads/writes “Fullname”
   1. Delete old column

Notes:

* It is possible to do this with Green/Blue or Canary deployments
* It is possible to go back from Step 2 or 3 to 1 again, if deployment did not go well

## Resources

<https://github.com/redhat-developer-demos/istio-tutorial>

<https://bit.ly/istio-cheat-sheet-v1>

# How to secure your microservices with Keycloak

Thomas Darimont

## Features

* Single Sign On/Out
* OAuth 2.0, SAML 2.0
* Authentication/Authorization
* Multi-Factor Auth
* One-time passwords
* Social Login
* LDAP/ADFS
* Centralized User Mgmt.

## Realm

* User
* Clients
* Roles
* Themes
* Identity-Provider (SAML, OpenID Connect, Social)
* Keys
* User Federation (Kerberos, LDAP)

## Roles

* Can be nested
* Can be global

## Token Types

* Access (minutes)
* Refresh (hours: get new access token)
* ID (OpenID)
* Offline (App can use it to “stay logged in” by requesting a new access token)

# Chaos Engineering

Benjamin Wilms

## Fallacies of distributed computing

* The network is reliable
* The latency is zero
* Bandwidth is infinite
* The network is secure
* Topology doesn’t change
* There is only one administrator
* Transport costs are 0€
* The network is homogeneous

## Examples

* Netflix Chaos Monkey
* Netflix Simian Army

## Definition

<http://principlesofchaos.org/>

* People
* Practices
* Process
* Application
* Platform
* Infrastructure

## Cycle

Steady State 🡪 Hypothesis 🡪 experiment 🡪 verify & clean 🡪 finding? Fix it!

## Steady State

* Define metrics about the overall state of your system
* Monitor your metrics
  + Technical metrics: CPU, Memory, …
  + Business metrics
* Business metrics outweigh technical metrics

## Tools

* stress (Linux)
* Traffic control (tc) package iproute2 (Linux)
* Toxiproxy
* Pumba (for docker testing)
* Gremlin (<https://www.gremlin.com/>)
* ChaosToolkit (<https://chaostoolkit.org/>)
* ChaosHub (<https://github.com/chaostoolkit/chaostoolkit-chaoshub>)
* Chaos Monkey for Spring Boot (Codecentric)

<https://speakerdeck.com/mrbw/chaos-engineering-aws-community-days-2018>

# Do proper monoliths before you do microservices

Jan de Vries

## Architecture

* Like Silos
* Per action segregation
* Exchange information via events / message bus
* Each service stores its data separately; it is possible to have different kinds of data store (SQL, NoSQL)

## Anti-Patterns

* Each service is calling each other service
* One big database each service is calling

Duplication is both good and bad.

Design is hard, implementation is easy.

## Process

1. Start with a monolith
2. Do it properly
3. If – and only if – necessary to separate a module from the monolith, extract it into a microservice

# Stop juggling with secrets – HashiCorp Vault for Secret Management

Thomas Störmer, Tilman Göhnert

## HashiCorp Vault

* One central store, single point of truth
* Secure generation, store, distribution
* Support for multiple authentication methods
* Unseal of store needs m of n keys
* Can act as PKI (public key infrastructure)
* Auditing (audit logs can be streamed to e.g. ELK)

## Setup

Spring Cloud Vault holt das Datenbank Passwort aus HashiCorp Vault und stellt es Spring Boot bereit (über bootstrap.yml).

## Take aways

1. Make use of dynamic secrets with limited lifetime
2. Use local vault docker container to start learning
3. Check official vault website for guides and docs
4. Ensure all vault communication channels enforce TLS!
5. Take care of terraform state/plan files
6. Use DB roles to manage privileges outside of vault for dynamic DB users
7. Use ‘set role’ pattern in flyway to fix ownership of created DB objects