Build a 12 factor microservice with MicroProfile

Alasdair Nottingham: Open Liberty Lead

@nottycode

12 Factors in a nut shell

- A methodology
- Best Practices
- Manifesto

https://12factor.net/ by Heroku

THE FACTORS

1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, Release, Run
6. Processes
7. Port binding
8. Concurrency
9. Disposability
10. Dev / Prod parity
11. Logs
12. Admin Processes

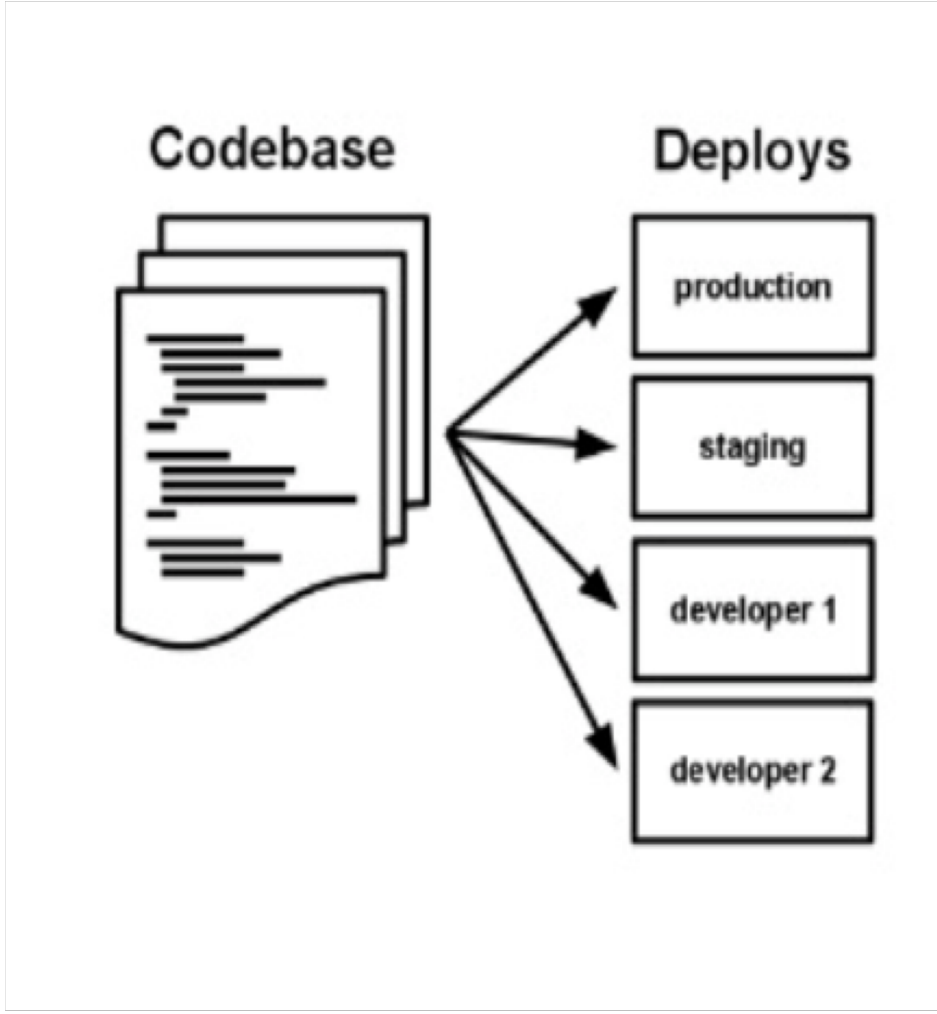
I. Codebase

"One codebase tracked in revision control, many deploys."

- Dedicate smaller teams to individual applications or microservices.
- Following the discipline of single repository for an application forces the teams to analyze the seams of their application, and identify potential monoliths that should be split off into microservices.

➢Use a single source code repository for a single application (1:1 relation). Deployment stages are different tags/branches

➢i.e. use a central git repo (external Github/GitHub Enterprise also suitable)



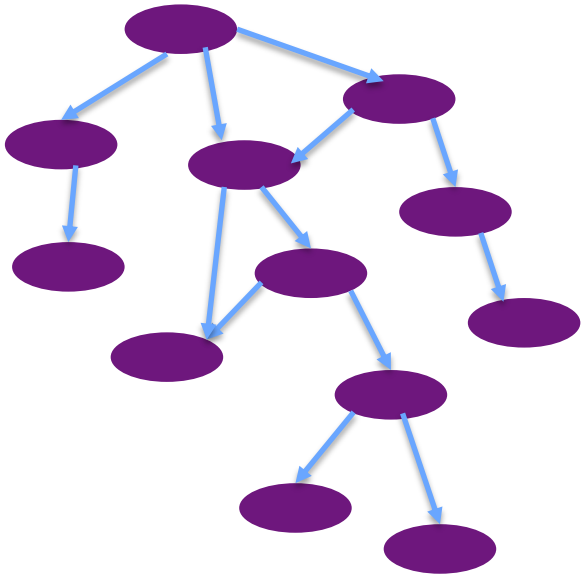
II. Dependencies

"Explicitly declare and isolate dependencies"

A cloud-native application does not rely on the pre-existence of dependencies in a deployment target.

Developer Tools declare and isolate dependencies

- Maven and Gradle for Java



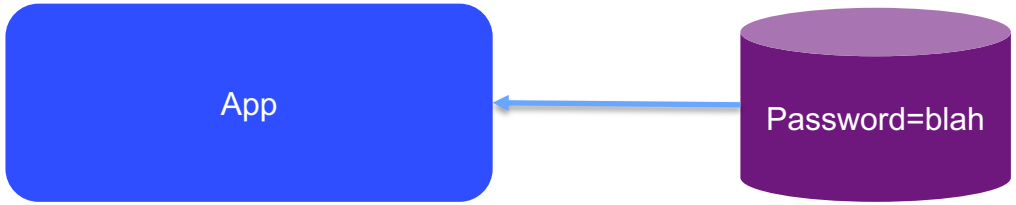
➢Each microservice has its own dependencies declared (e.g. pom.xml)

III. Config

MICROPROFILE
OPTIMIZING ENTERPRISE JAVA

"Store config in the environment"

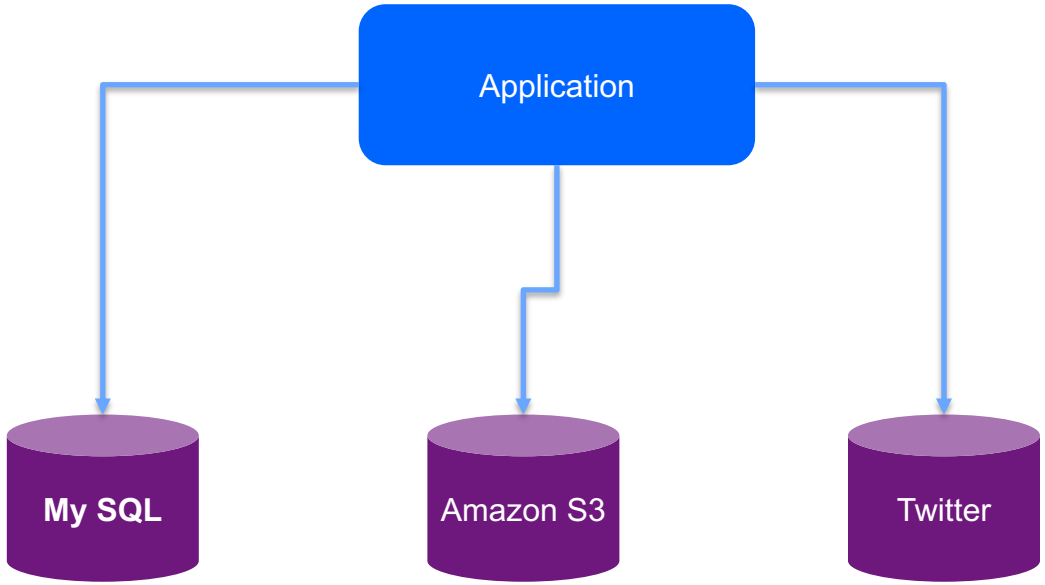
- Changing config should not need to repackage your application
- Use Kubernetes configmaps and secrets for container services, rather than environment variables specified in the container image
- Use MicroProfile Config to inject the config properties into the microservices



IV. Backing services

MICROPROFILE
OPTIMIZING ENTERPRISE JAVA

"Treat backing services as attached resources"



V. Build, release, run

"Strictly separate build and run stages"

- Source code is used in the build stage. Configuration data is added to define a release stage that can be deployed. Any changes in code or config will result in a new build/release
- Needs to be considered in CI pipeline

IBM
- UrbanCode Deploy
- IBM Cloud Continuous Delivery Service

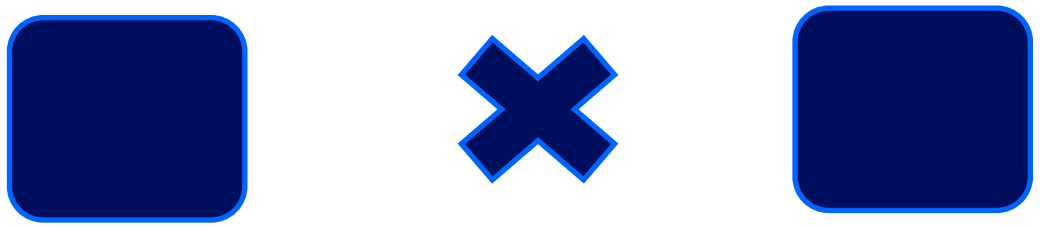
AWS
- AWS CodeBuild
- AWS CodeDeploy
- AWS CodePipeline (not yet integrated with EKS)

Azure
- Visual Studio Team Services (VSTS) (includes git)
- Web App for Containers feature of Azure App Service

VI. Processes

“Execute the app as one or more stateless processes”

- Stateless and share-nothing
- Rest API



VII. Port binding

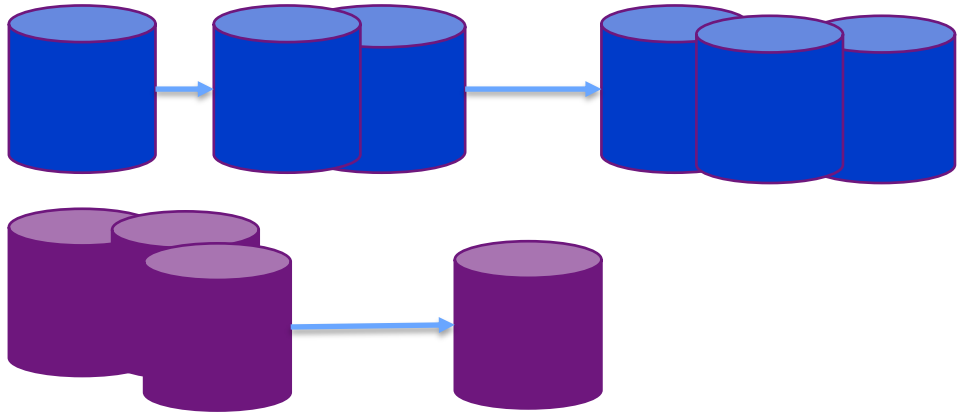
“Export services via port binding”

- Applications are fully self-contained and expose services only through ports. Port assignment is done by the execution environment
- Ingress/service definition of k8s manages mapping of ports

VIII. Concurrency

"Scale out via the process model"

- Applications use processes independent from each other to scale out (allowing for load balancing)
- To be considered in application design
- Cloud autoscaling services: [auto]scaling built into k8s
- Build microservices

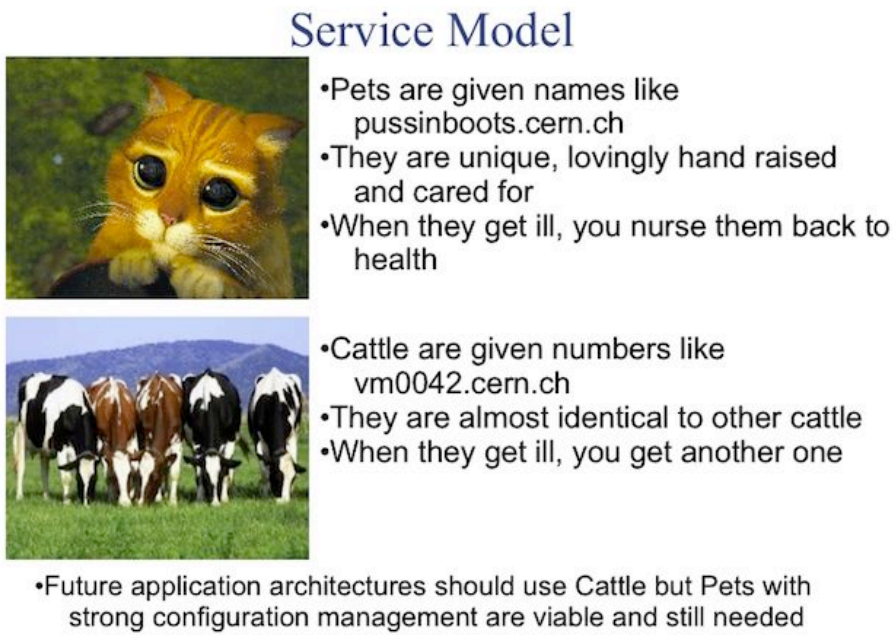


IX. Disposability

MICROPROFILE
OPTIMIZING ENTERPRISE JAVA

"Maximize robustness with fast startup and graceful shutdown"

- Processes start up fast.
- Processes shut down gracefully when requested.
- Processes are robust against sudden death
  - Use MicroProfile Fault Tolerance to make it resilient



From "*CERN Data Centre Evolution*"

X. Dev/prod parity

“Keep development, staging, and production as similar as possible”

- Development and production are as close as possible (in terms of code, people, and environments)
- Can use helm to deploy in repeatable manner
- Use (name)spaces for isolation of similar setups

XI. Logs

“Treat logs as event streams”

- App writes all logs to stdout
- Use a structured output for meaningful logs suitable for analysis. Execution environment handles routing and analysis infrastructure

XII. Admin processes

“Run admin/management tasks as one-off processes”

- Tooling: standard k8s tooling like “kubectl exec” or Kubernetes Jobs
- Also to be considered in solution/application design
- For example, if an application needs to migrate data into a database, place this task into a separate component instead of adding it to the main application code at startup

THE FACTORS

1. Codebase
2. Dependencies
3. Config MICROPROFILE™ OPTIMIZING ENTERPRISE JAVA
4. Backing Services MICROPROFILE™ OPTIMIZING ENTERPRISE JAVA
5. Build, Release, Run
6. Processes
7. Port binding
8. Concurrency
9. Disposability MICROPROFILE™ OPTIMIZING ENTERPRISE JAVA
10. Dev / Prod parity
11. Logs
12. Admin Processes

MicroProfile Config

Why?

– Configure Microservice without repacking the application

How?

– Specify the configuration in configure sources

– Access configuration via

- Programmatically lookup

```
Config config =ConfigProvider.getConfig();

config.getValue("myProp", String.class);
```

- Via CDI Injection

```
@Inject
@ConfigProperty(name="my.string.property")
String myPropV;
```

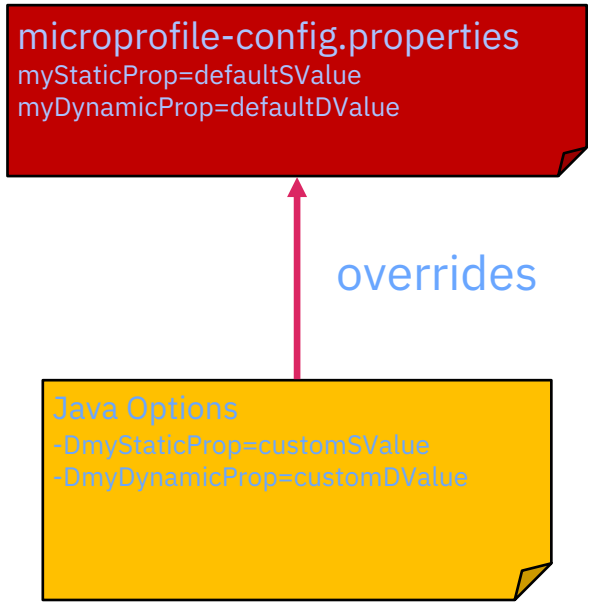
MicroProfile Config

Static Config

```
@Inject
@ConfigProperty(name="myStaticProp")
private String staticProp;
```

Dynamic Config

```
@Inject
@ConfigProperty(name="myDynamicProp")
private Provider<String> dynamicProp;
```



MicroProfile Fault Tolerance

A solution to build a resilient microservice

- Retry - `@Retry`
- Circuit Breaker - `@CircuitBreaker`
- Bulk Head - `@Bulkhead`
- Time out - `@Timeout`
- Fallback - `@Fallback`

References

- Code sample to demonstrate 12-factor app
  - https://github.com/Emily-Jiang/12factor-deployment
  - https://github.com/Emily-Jiang/12factor-app-a
  - https://github.com/Emily-Jiang/12factor-app-b
- http://microprofile.io
- http://openliberty.io
- https://www.12factor.net/