Build a 12 Factor Microservice in an Hour

Andrew Rouse: Open Liberty Developer for MicroProfile and CDI, IBM

@azquelt

Contents

12 Factors in a nut shell

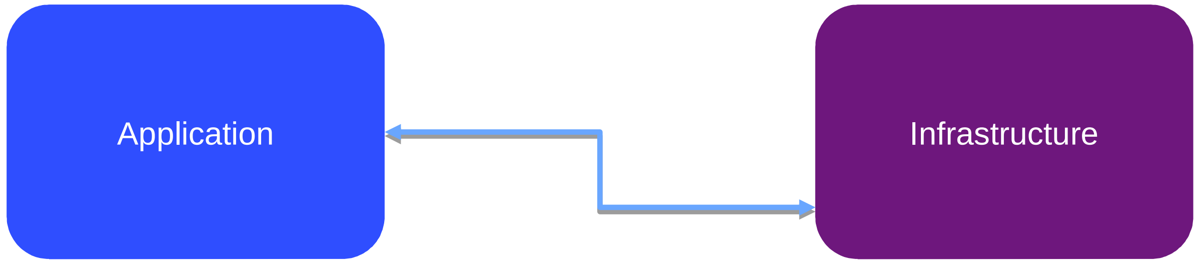
THE TWELVE-FACTOR APP

- A methodology
- Best Practices
- Manifesto

https://12factor.net/ by Heroku

Why 12 factor?

- Define the contract between applications and infrastructure



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

Why 12 factor?

In the modern era, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*. The twelve-factor app is a methodology for building software-as-a-service apps that:

Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;

Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;

Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;

Minimize divergence between development and production, enabling **continuous deployment** for maximum agility;

And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

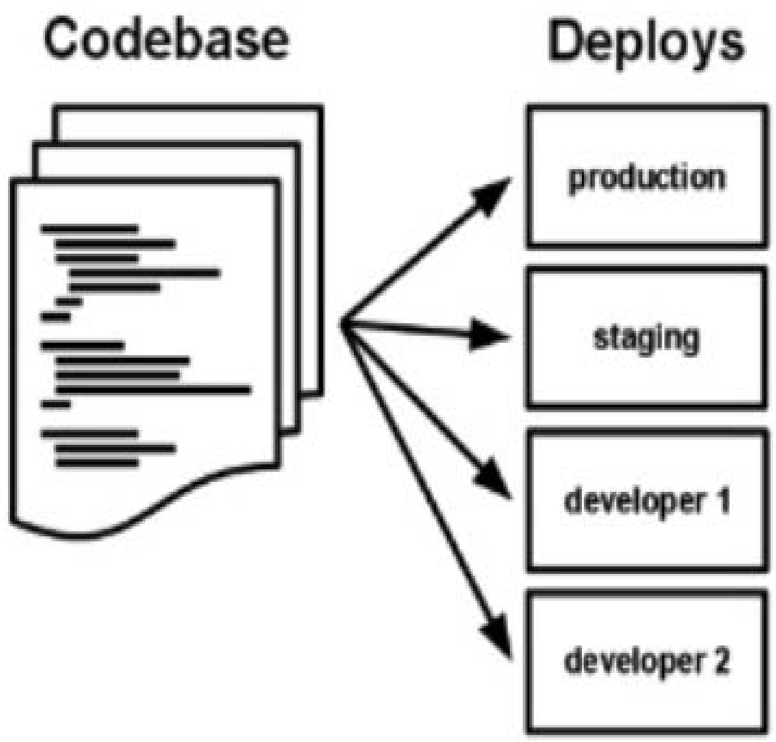
From https://12factor.net

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).
    ➢ E.g. use a central git repo



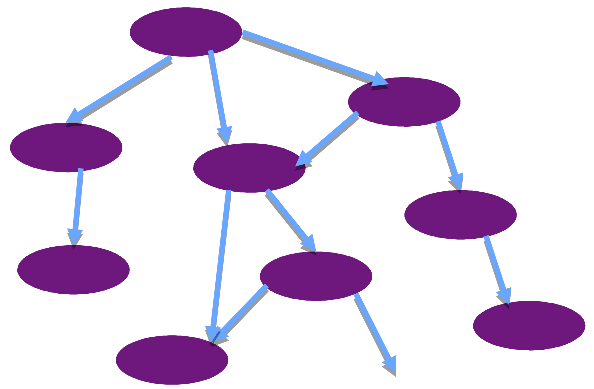
II. Dependencies

“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

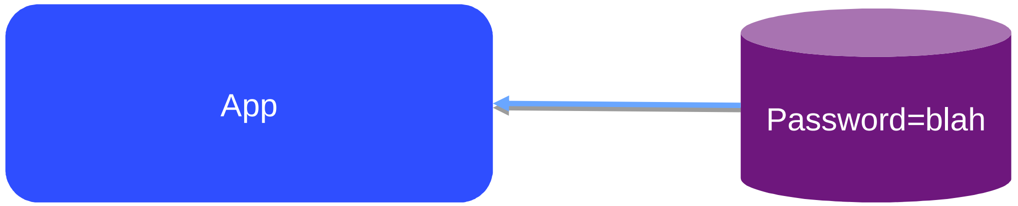
- Maven and Gradle declare Java library dependencies
- Dockerfile declares dependencies on OS and other tools
- Each microservice declares its own dependencies



III. Config

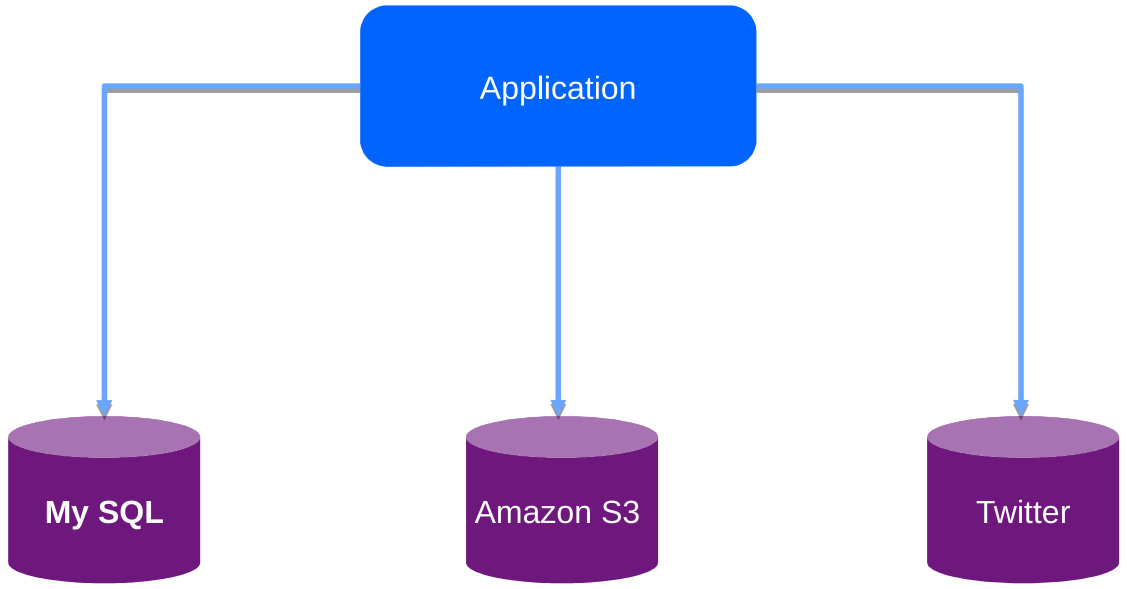
“Store config in the environment”

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



IV. Backing services

"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

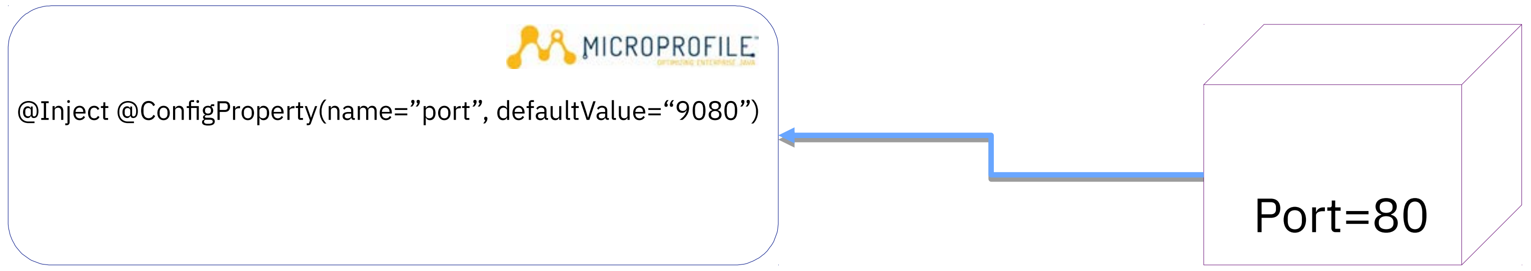
“Execute the app as one or more stateless processes”

- ➢Stateless and share-nothing
- ➢Any persistence uses a backing service
  - – Database
  - – Shared cache
  - – Message queue

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
- Use MP Config to inject ports to microservices to connect up microservices

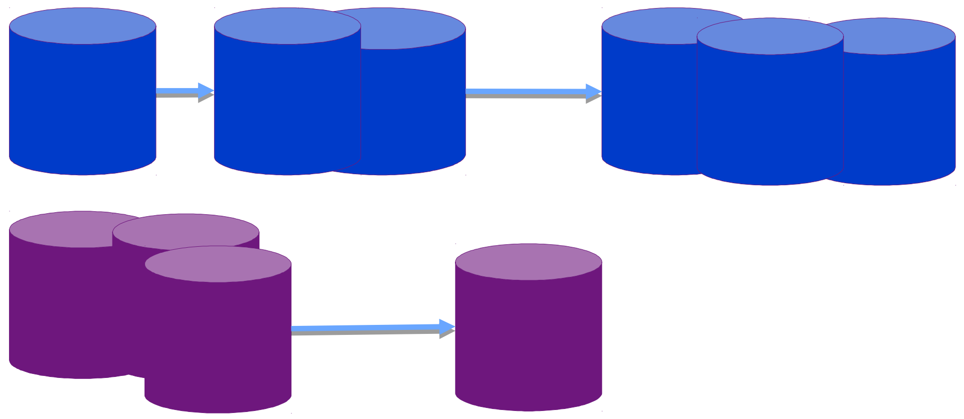


```
@Inject @ConfigProperty(name="port", defaultValue="9080")
```

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

"Maximize robustness with fast startup and graceful shutdown"

- Processes start up fast.
- Processes shut down gracefully when requested.
- Processes are robust against sudden death
  - Ensure you won't lose data if part of your system dies
  - Use MicroProfile Fault Tolerance to make calls to other services resilient

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
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

DEMO

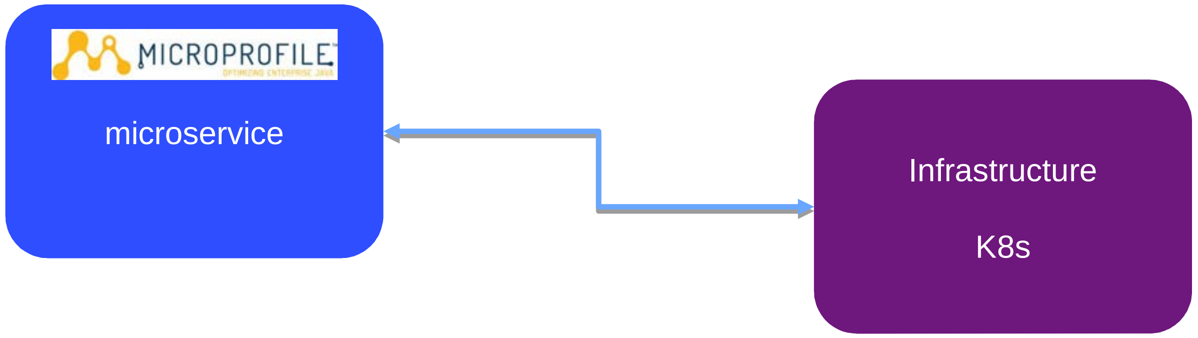
1	2	3	4	5	6	7	8	9	~~10~~	11	~~12~~
Codebase	Deps	Config	Backing Service	Build Release Run	Stateless	Port Binding	Scaling	Disposability	~~Prod, Dev, Staging~~	Logs	~~Admin Processes~~

1. Create a REST app (stateless, in one codebase)
2. Declare our dependencies using Maven and Dockerfile
   - Including OpenLiberty
3. Release and Run by deploying with config to Kubernetes
4. Update our app to call another service
5. Deploy again with scaling

12 FACTOR BINGO

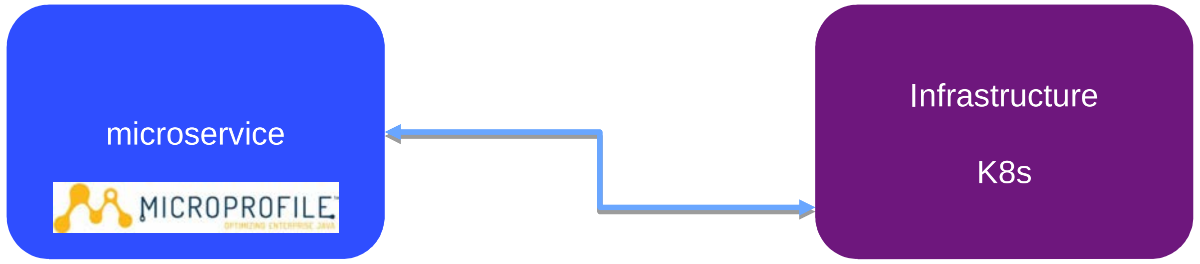
References

- Code sample to demonstrate 12-factor app
  - https://github.com/Azquelt/12factor-deployment
  - https://github.com/Azquelt/12factor-app-a
  - https://github.com/Azquelt/12factor-app-b
- http://microprofile.io
- http://openliberty.io (especially guides)
- https://www.12factor.net/



12 factor app

- Use MicroProfile and K8s to build a microservice => 12 factor app



MicroProfile Config

Why?

– Configure Microservice without repacking the application

How?

– Specify the configuration in configuration sources

– Access configuration via

- Programmatic lookup

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

config.getValue(“myProp”, String.class);
```

- CDI Injection

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

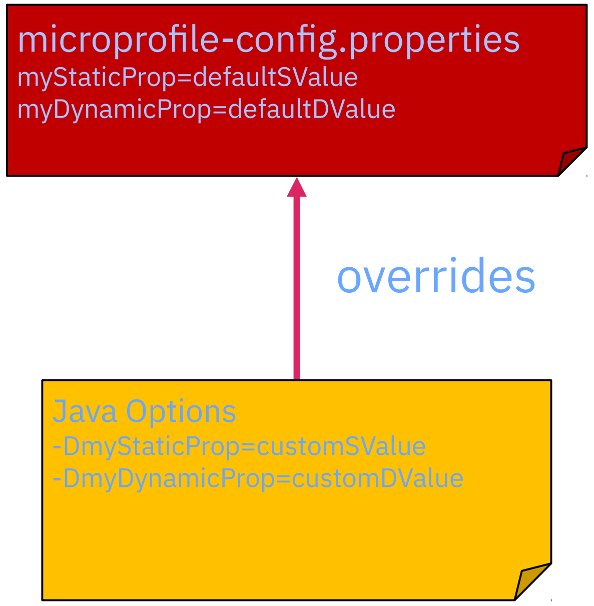
MicroProfile Config

Static Config

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

Dynamic Config

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


microprofile-config.properties

```
myStaticProp=defaultSValue
myDynamicProp=defaultDValue
```

Java Options

```
-DmyStaticProp=customSValue
-DmyDynamicProp=customDValue
```

MicroProfile Fault Tolerance

A solution to build a resilient microservice

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

Backup: Using IBM Cloud Private

Codebase	Source: Github Enterprise, github Images: any registry, IBM Cloud private registry
Dependencies	Dependency management of language environment; container build process for repeatable inclusion of dependencies
Config	k8s configmaps and secrets
Backing services	Use configuration (see previous factor) to define target server as used by application
Build, release, run	UrbanCode Deploy UrbanCode Release Plus k8s mechanisms with CI tooling
Processes	To be considered in application design

Port binding	Application needs to expose ports. Ingress/service definition of k8s manages mapping of ports
Concurrency	App design ([auto]scaling built into k8s)
Disposability	App design
Dev/prod parity	Can use helm to deploy in same way. Namespaces for isolation of similar areas
Logs	ELK as part of ICP (or RYO)
Admin processes	App design; standard k8s tooling like “kubectl exec” or Kubernetes Jobs

Build a 12 Factor Microservice in an Hour

Andrew Rouse: Open Liberty Developer for MicroProfile and CDI, IBM

@azquelt

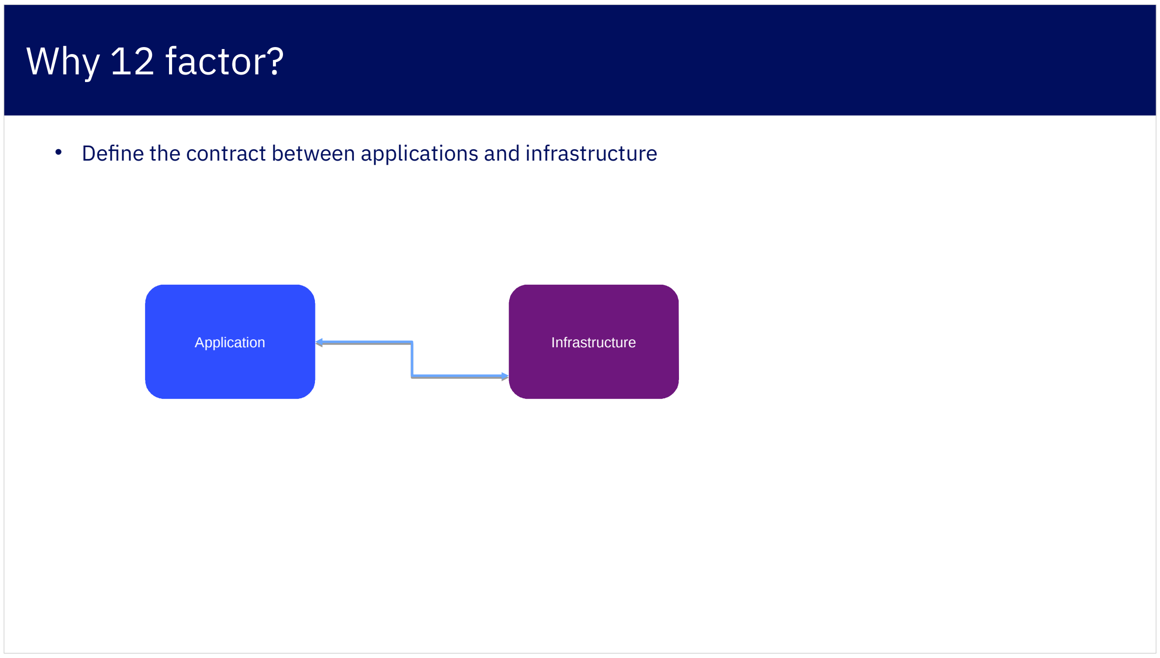
Contents

12 Factors in a nut shell

THE TWELVE-FACTOR APP

- A methodology
- Best Practices
- Manifesto

https://12factor.net/ by Heroku



Why 12 factor?

- Define the contract between applications and infrastructure

Application
Infrastructure

- Application design influences the infrastructure required (Kubernetes, Database selection, etc…)
- Application design guides price; a cloud-native application can be resilient without surplus infrastructure
- Some services are provided by an application but some are provided from the infrastructure
- Applications depend on the features and services of infrastructure to support agile development.
- Infrastructure requires applications to expose endpoints and integrations to be managed autonomously (e.g. Kubenetes asks applications to expose health endpoint)
- Increased resource and memory consumption of independently running components that

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

Why 12 factor?

In the modern era, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*. The twelve-factor app is a methodology for building software-as-a-service apps that:

Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;

Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;

Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;

Minimize divergence between development and production, enabling **continuous deployment** for maximum agility;

And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

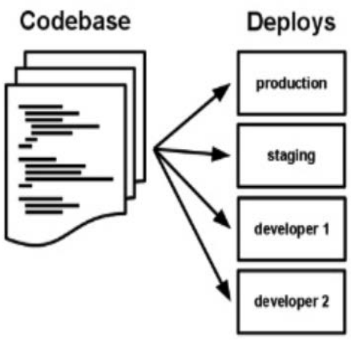
From https://12factor.net

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).
  - ➢ E.g. use a central git repo



A *codebase* is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized system like Git where changes are committed.) Also in Git – scenario of repository per package/service.

multiple repository advantages:

- Clear ownership: team that owns a service is clearly responsible for independently develop and deploy the full stack of that service
- Smaller code base: Separate repositories for a service leads to smaller code base and lesser complexity during code merge.
- Narrow clones: faster DevOps and automated build and releases as smaller code base means lesscode download/clone time

multiple repository disadvantages:

Difficult development and debugging: development, cross team communications and shared codes are difficult to maintain and thus development and debugging can be an issue
Abstracts the knowledge of the platform: Since each team is only responsible for a single service, integration becomes an issue and knowledge of the platform can decrease
Multiple Everything: Multiple dependencies, duplication and integrations

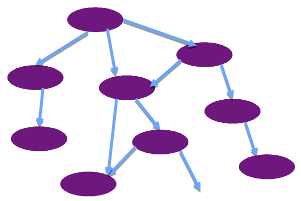
II. Dependencies

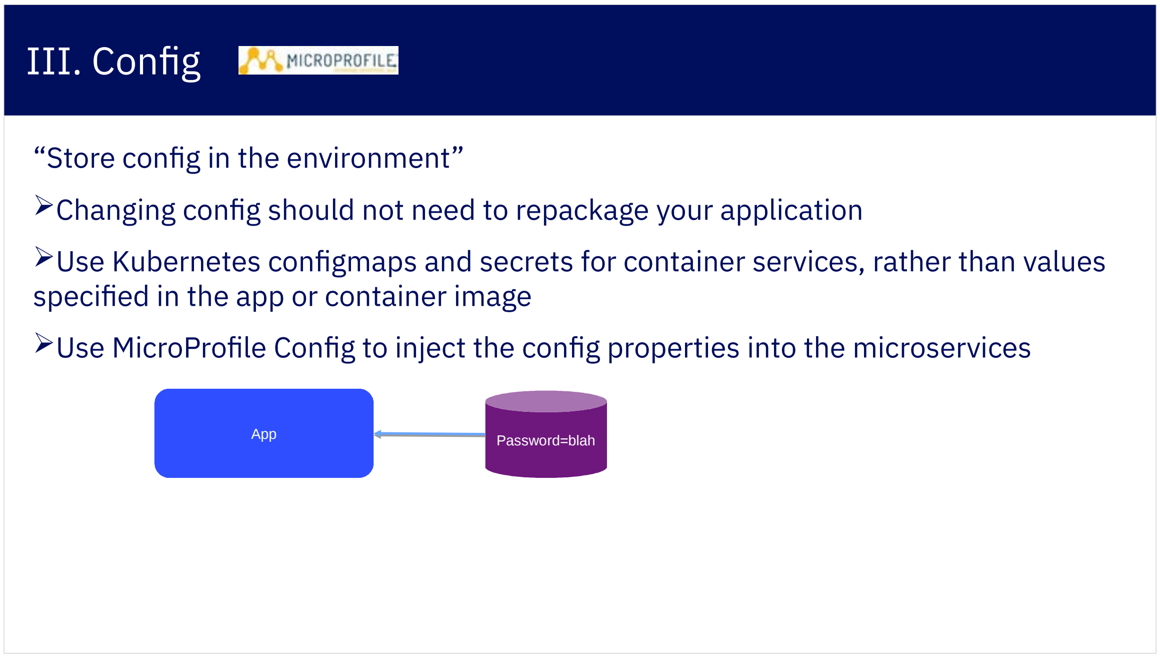
"Explicitly declare and isolate dependencies"

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 declare Java library dependencies
- Dockerfile declares dependencies on OS and other tools
- Each microservice declares its own dependencies





III. Config

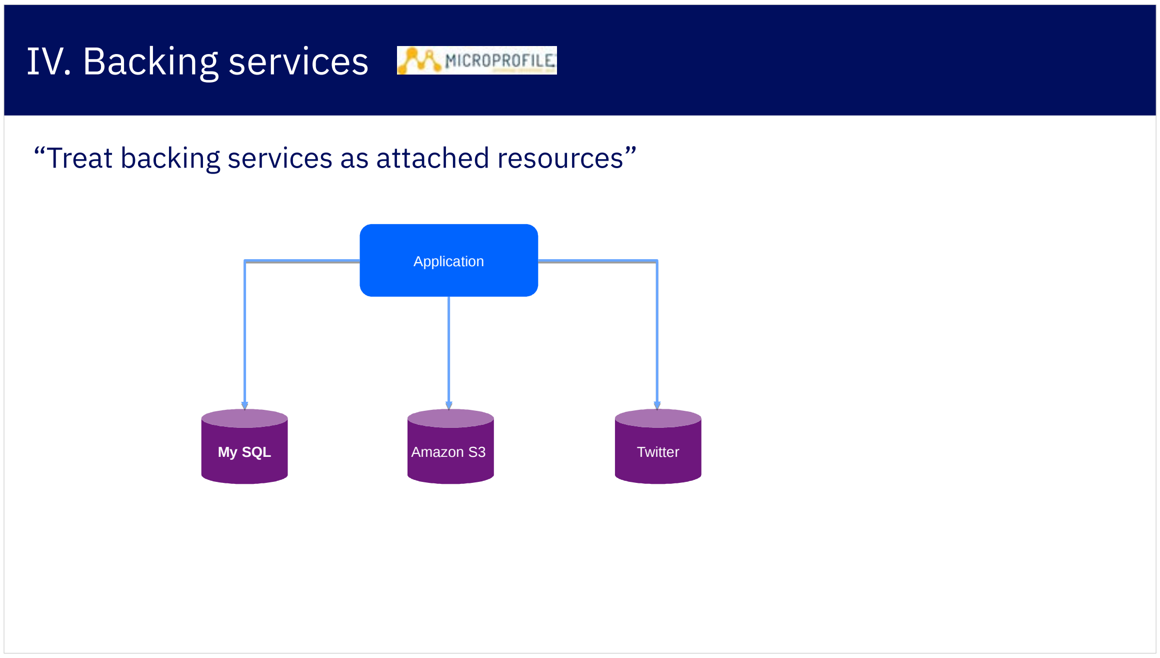
"Store config in the environment"

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

App
Password=blah

Do these make sense? –HJ
(struggling how these offerings help pushing configurations –ub)

Let's leave them because we have the space. I'm not really attached to them, so we can delete…



IV. Backing services

"Treat backing services as attached resources"

- A backing service is any service on which an application relies (data stores, messaging systems, caching systems, security services)
- Use configuration (see factor III) to define target service as used by application. Configuration defines access to a backing service
- Resource Binding should be done via external configuration.
- Attach and detach backing services from an application at will, without re-deploying the application.
- Fault Tolerance pattern: allow code to stop communicating with misbehaving backing services, providing a fallback path

V. Build, release, run

“Strictly separate build and run stages”

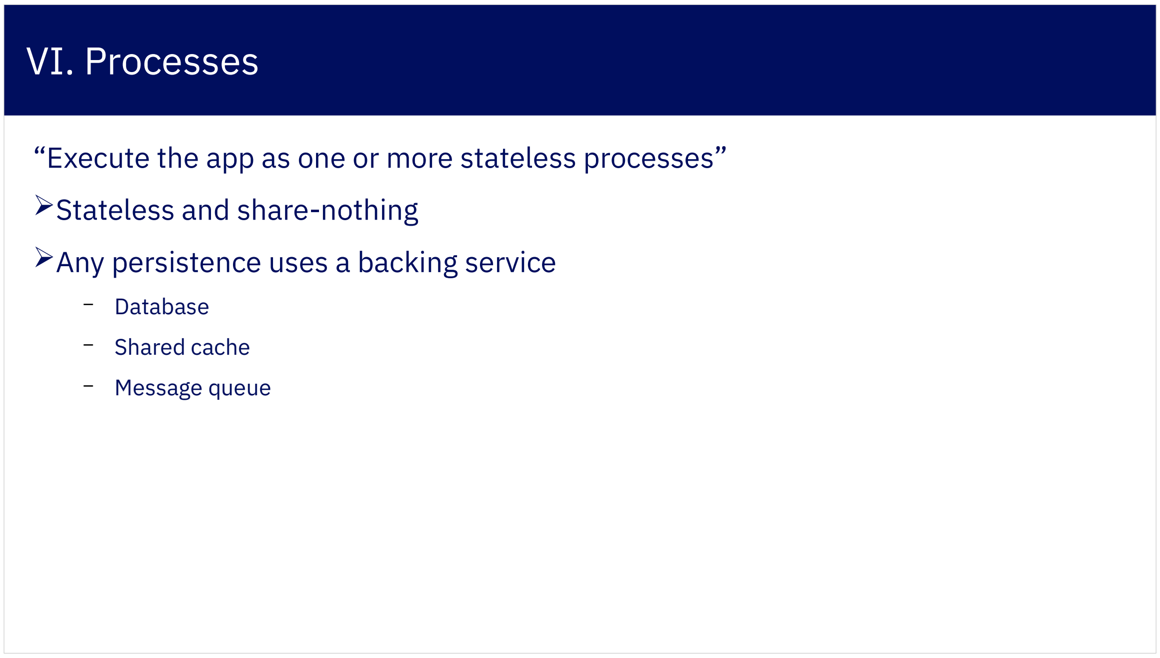
➢Source code is used in the build stage. Configuration data is added to define a release stage that can be deployed. Any changes in code or config will result in a new build/release

➢Needs to be considered in CI pipeline

IBM
- UrbanCode Deploy
- IBM Cloud Continuous Deli very Service

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

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



VI. Processes

"Execute the app as one or more stateless processes"

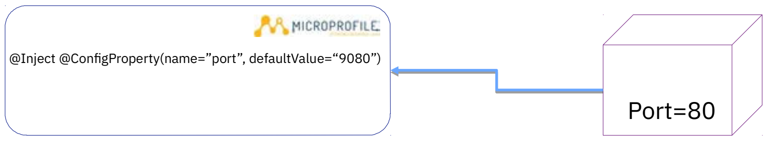
- ➢Stateless and share-nothing
- ➢Any persistence uses a backing service
  - Database
  - Shared cache
  - Message queue

- ➢ Components are stateless and shared-nothing. State can be put in a stateful backing service (database)
- ➢ Stateless components can be replaced quickly if they fail
- ➢ Avoid dependencies on sticky sessions and keep session data in a persistent store to ensure traffic can be routed to other processes without service disruption
- ➢ To be considered in application design by the developer, not in Cloud roll-out

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
- Use MP Config to inject ports to microservices to connect up microservices



VIII. Concurrency

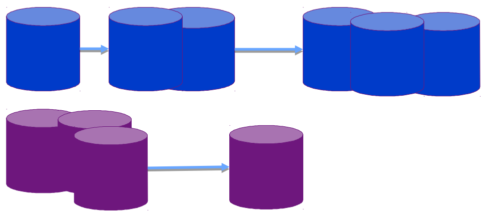
"Scale out via the process model"

➢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

"Maximize robustness with fast startup and graceful shutdown"

- Processes start up fast.
- Processes shut down gracefully when requested.
- Processes are robust against sudden death
  - Ensure you won't lose data if part of your system dies
  - Use MicroProfile Fault Tolerance to make calls to other services resilient

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

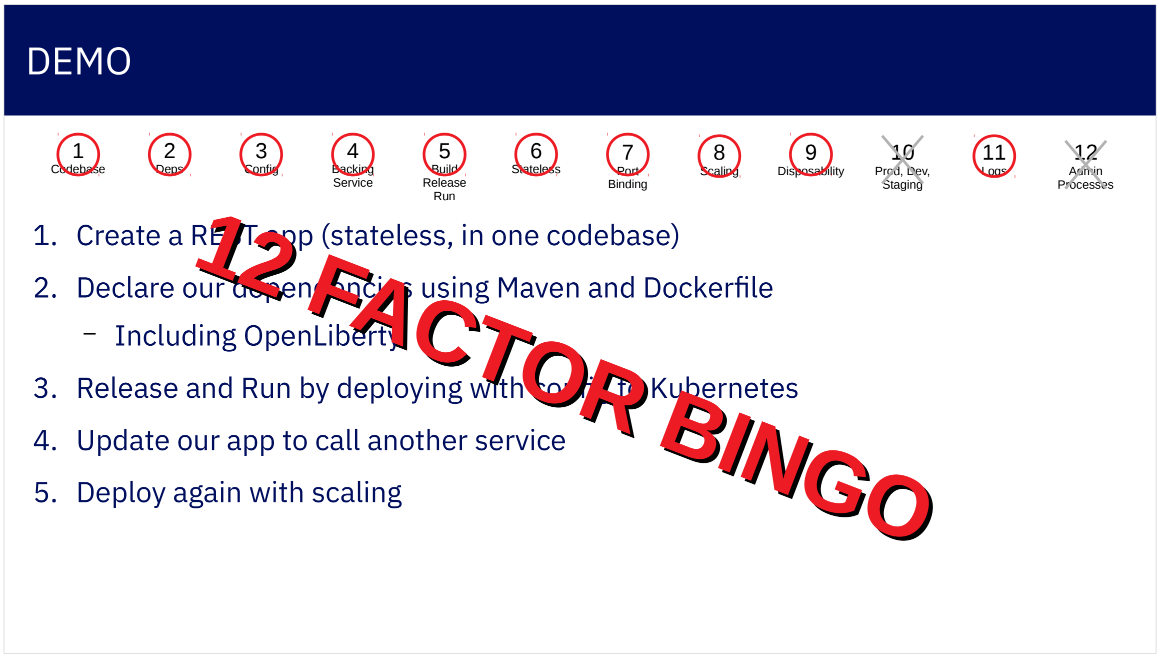
XII. Admin processes

“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
4. Backing Services
5. Build, Release, Run
6. Processes
7. Port binding MICROPROFILE
8. Concurrency
9. Disposability MICROPROFILE
10. Dev / Prod parity
11. Logs
12. Admin Processes



DEMO

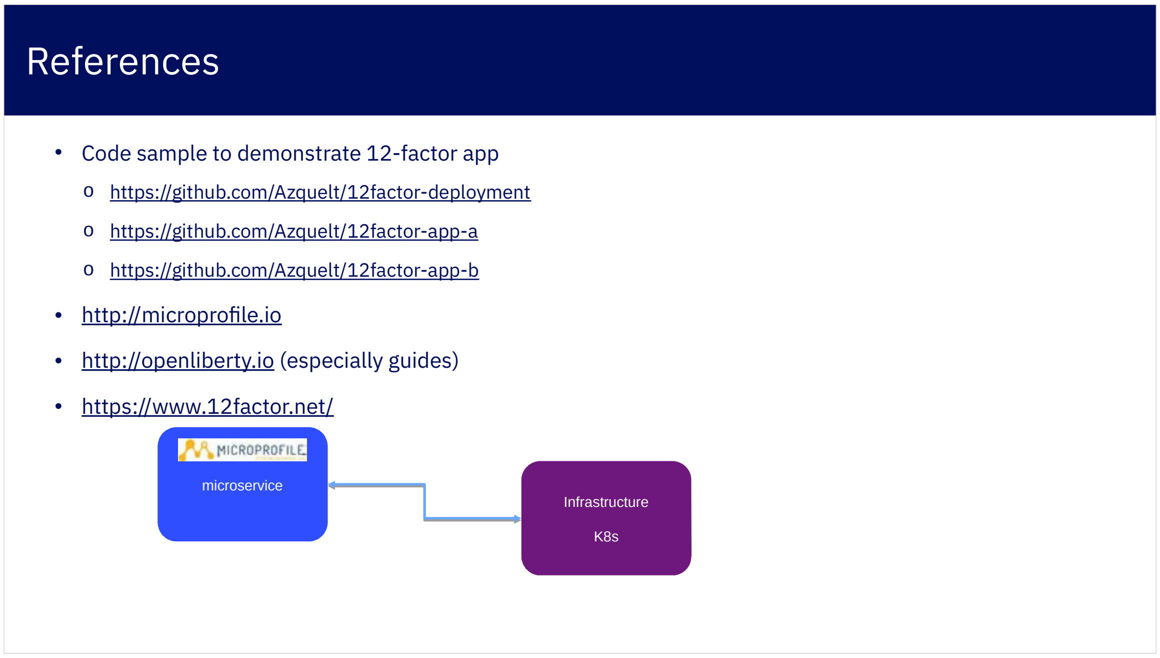
1 Codebase | 2 Deps | 3 Config | 4 Backing Service | 5 Build Release Run | 6 Stateless | 7 Port Binding | 8 Scaling | 9 Disposability | ~~10 Prod, Dev, Staging~~ | 11 Logs | ~~12 Admin Processes~~

1. Create a REST app (stateless, in one codebase)
2. Declare our dependencies using Maven and Dockerfile
   - Including OpenLiberty
3. Release and Run by deploying with config to Kubernetes
4. Update our app to call another service
5. Deploy again with scaling

12 FACTOR BINGO

Not covering 10 or 12 because…

We will create a REST app which is *stateless and has *one codebase
Declare our *dependencies using maven and a Dockerfile
Includes OpenLiberty which will take care of *binding to a port, starting up quickly and shutting down gracefully which is important for *disposability, and will *log to stdout
We will then build our app before doing a *release and run by deploying the app with its config to kubernetes
Then we'll update our app to call another service which we'll treat as an *attached backing service
And finally we'll deploy it again and demonstrate *scaling with Kubernetes
Hopefully allowing us to achieve *12 Factor Bingo!

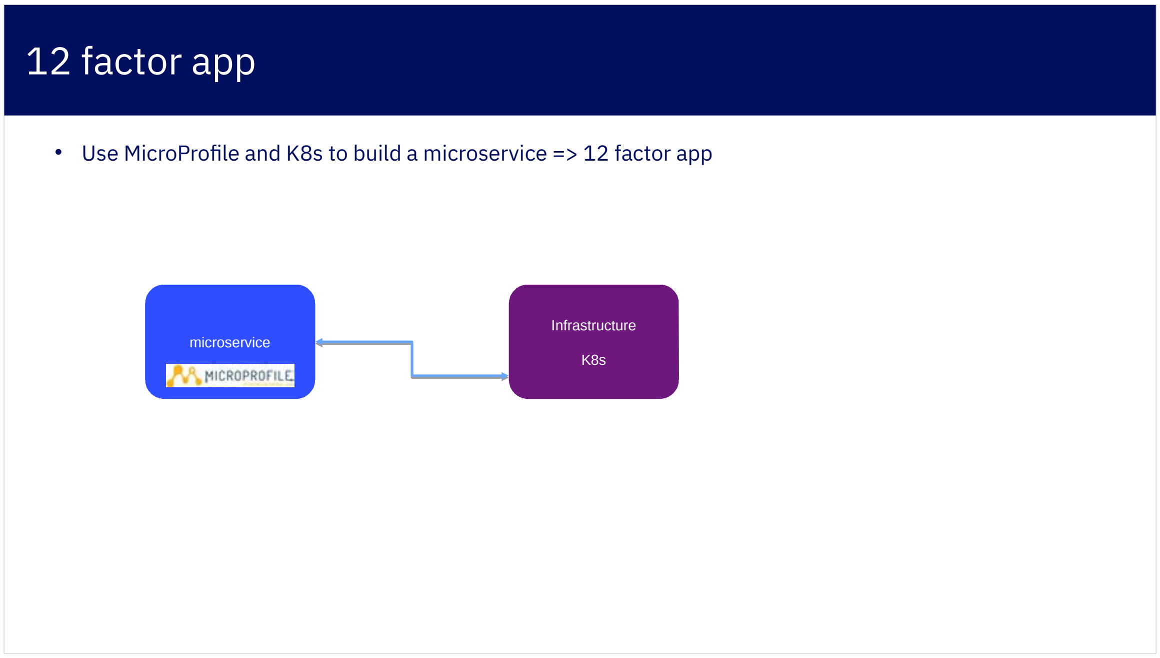


References

- Code sample to demonstrate 12-factor app
  - https://github.com/Azquelt/12factor-deployment
  - https://github.com/Azquelt/12factor-app-a
  - https://github.com/Azquelt/12factor-app-b
- http://microprofile.io
- http://openliberty.io (especially guides)
- https://www.12factor.net/

MICROPROFILE
microservice
Infrastructure
K8s

- Application design influences the infrastructure required (Kubernetes, Database selection, etc…)
- Application design guides price; a cloud-native application can be resilient without surplus infrastructure
- Some services are provided by an application but some are provided from the infrastructure
- Applications depend on the features and services of infrastructure to support agile development.
- Infrastructure requires applications to expose endpoints and integrations to be managed autonomously (e.g. Kubenetes asks applications to expose health endpoint)
- Increased resource and memory consumption of independently running components that



12 factor app

- Use MicroProfile and K8s to build a microservice => 12 factor app

microservice
MICROPROFILE
Infrastructure
K8s

- Application design influences the infrastructure required (Kubernetes, Database selection, etc…)
- Application design guides price; a cloud-native application can be resilient without surplus infrastructure
- Some services are provided by an application but some are provided from the infrastructure
- Applications depend on the features and services of infrastructure to support agile development.
- Infrastructure requires applications to expose endpoints and integrations to be managed autonomously (e.g. Kubenetes asks applications to expose health endpoint)
- Increased resource and memory consumption of independently running components that

MicroProfile Config

Why?

– Configure Microservice without repacking the application

How?

– Specify the configuration in configuration sources

– Access configuration via

- Programmatic lookup

```
Config config = ConfigProvider.getConfig();
config.getValue(“myProp”, String.class);
```

- CDI Injection

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

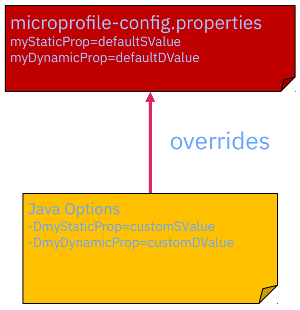
MicroProfile Config

Static Config

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

Dynamic Config

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


microprofile-config.properties

```
myStaticProp=defaultSValue
myDynamicProp=defaultDValue
```

Java Options

```
-DmyStaticProp=customSValue
-DmyDynamicProp=customDValue
```

MicroProfile Fault Tolerance

A solution to build a resilient microservice

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

Backup: Using IBM Cloud Private

Codebase	Source: Github Enterprise, github Images: any registry, IBM Cloud private registry
Dependencies	Dependency management of language environment; container build process for repeatable inclusion of dependencies
Config	k8s configmaps and secrets
Backing services	Use configuration (see previous factor) to define target server as used by application
Build, release, run	UrbanCode Deploy UrbanCode Release Plus k8s mechanisms with CI tooling
Processes	To be considered in application design

Port binding	Application needs to expose ports. Ingress/service definition of k8s manages mapping of ports
Concurrency	App design ([auto]scaling built into k8s)
Disposability	App design
Dev/prod parity	Can use helm to deploy in same way. Namespaces for isolation of similar areas
Logs	ELK as part of ICP (or RYO)
Admin processes	App design; standard k8s tooling like “kubectl exec” or Kubernetes Jobs