RED HAT®
CONSULTING

redhat.

WHITEPAPER

PRINCIPLES OF CONTAINER-BASED APPLICATION DESIGN

By Bilgin Ibryam

EXECUTIVE SUMMARY

"Cloud native" is a term used to describe applications designed specifically to run on a cloud-based infrastructure. Typically, cloud-native applications are developed as loosely coupled microservices running in containers managed by platforms. These applications anticipate failure, and they run and scale reliably even when their underlying infrastructure is experiencing outages. To offer such capabilities, cloud-native platforms impose a set of contracts and constraints on the applications running on them. These contracts ensure that the applications conform to certain constraints and allow the platforms to automate the management of the containerized applications.

Many organizations understand the necessity and importance of becoming cloud native, but do not know where to start. Ensuring that cloud-native platforms and the containerized applications that run on them work seamlessly together provides the ability to anticipate failure and the reliability to run and scale even when the underlying infrastructure experiences outages. This whitepaper describes a number of principles that containerized applications must comply with in order to become good cloud-native citizens. Adhering to these principles will help ensure that your applications are suitable for automation in cloud-native platforms such as Kubernetes.

PRINCIPLES OF SOFTWARE DESIGN:

- Keep it simple, stupid (KISS)
- Don't repeat yourself (DRY)
- You aren't gonna need it (YAGNI)
- Separation of concerns (SoC)

RED HAT APPROACH TO CLOUD-NATIVE CONTAINERS:

- Single concern principle (SCP)
- High observability principle (HOP)
- Life-cycle conformance principle (LCP)
- Image immutability principle (IIP)
- Process disposability principle (PDP)
- Self-containment principle (S-CP)
- Runtime confinement principle (RCP)

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com

COMMON CONTAINER-RELATED BEST PRACTICES:

- Aim for small images.
- Support arbitrary user IDs.
- Mark important ports.
- Use volumes for persistent data.
- Set image metadata.
- Synchronize host and image.

PRINCIPLES OF SOFTWARE DESIGN

Principles exist in many areas of life, and they generally represent a fundamental truth or belief from which others are derived. In software, principles are rather abstract guidelines, which are supposed to be followed while designing software. They can be applied to any programming language, implemented using different patterns, and achieved following different practices.

Typically, patterns and practices are the tools used to achieve the desired outcome of the principles. There are fundamental principles for writing quality software from which all the other principles are derived. Examples of such principles include:

- **KISS**—Keep it simple, stupid.
- **DRY**—Don't repeat yourself.
- **YAGNI**—You aren't gonna need it.
- **SoC**—Separation of concerns.

Even if these principles do not specify concrete rules, they represent a language and common wisdom that many developers understand and refer to regularly.

There are also SOLID (**S**ingle responsibility, **O**pen/closed, **L**iskov substitution, **I**nterface segregation, **D**ependency inversion) principles that were introduced by Robert C. Martin, which represent guidelines for writing better object-oriented software. It is a framework consisting of complementary principles that are generic and open for interpretation but still give enough direction for creating better object-oriented designs. The expectation from SOLID principles is that, when applied, we are more likely to create a system that has higher-quality attributes and is more maintainable in the long term.

The SOLID principles use object-oriented primitives and concepts such as classes, interfaces, and inheritance for reasoning about object-oriented designs. In a similar way, there also principles for designing cloud-native applications in which the main primitive is the container image rather than a class. Following these principles, we are more likely to create containerized applications that are better suited for cloud-native platforms such as Kubernetes.

RED HAT APPROACH TO CLOUD-NATIVE CONTAINERS

Nowadays, it is possible to put almost any application in a container and run it. But to create a containerized application that can be automated and orchestrated effectively by a cloud-native platform such as Kubernetes requires additional effort.

The ideas below are inspired by many other works such as "The Twelve-Factor App," in which the scope ranges from source code management to application scalability models. However, the scope of the following principles is constrained to designing containerized microservices-based applications for cloud-native platforms such as Kubernetes.

The principles for creating containerized applications listed below use the container image as the basic primitive and the container orchestration platform as the target container runtime environment. Following these principles will ensure that the resulting containers behave like a good cloud-native citizen in most container orchestration engines, allowing them to be scheduled, scaled, and monitored in an automated fashion. These principles are presented in no particular order.

SINGLE CONCERN PRINCIPLE (SCP)

In many ways, this principle is similar to the single responsibility principle (SRP) from SOLID, which advises that a class should have only one responsibility. The motivation behind the SRP is that each responsibility is an axis of change and a class should have one—and only one—reason to change. The word "concern" in the SCP principle highlights concern as a higher level of abstraction than responsibility, and it better describes the scope as a container as opposed to a class. While the main motivation for SRP is to have a single reason for a change, the main motivation for SCP is container image reuse and replaceability. If you create a container that addresses a single concern, and it does it in a feature-complete way, the chances are higher of container image reuse in different application contexts.

Thus, the SCP principle dictates that every container should address a single concern and do it well. Achieving it is easier than achieving SRP in the object-oriented world, as containers usually manage a single process, and most of the time that single process addresses a single concern.

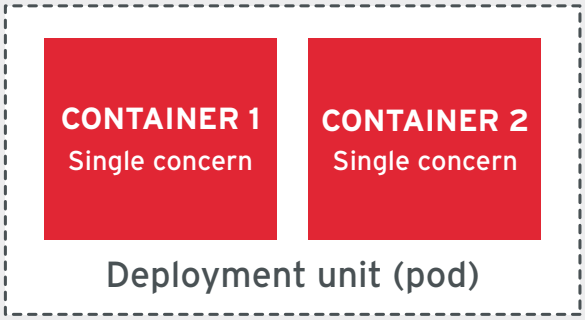

Figure 1. Multiple single-concern containers in one deployment unit

If your containerized microservice needs to address multiple concerns, it can use patterns such as sidecar and init-containers to combine multiple containers into a single deployment unit (pod), where each container still handles a single concern. Similarly, you can swap containers that address the same concern. For example, replace the web server container, or a queue implementation container, with a newer and more scalable one.

HIGH OBSERVABILITY PRINCIPLE (HOP)

Containers provide a unified way for packaging and running applications by treating them like a black box. But any container aiming to become a cloud-native citizen must provide application programming interfaces (APIs) for the runtime environment to observe the container health and act accordingly. This is a fundamental prerequisite for automating container updates and life cycles in a unified way, which in turn improves the system's resilience and user experience.

Figure 2. A container with multiple observability APIs (containers have multiple APIs to enable observability)

In practical terms, at a very minimum, your containerized application must provide APIs for the different kinds of health checks—liveness and readiness. Even better-behaving applications must provide other means to observe the state of the containerized application. The application should log important events into the standard error (STDERR) and standard output (STDOUT) for log aggregation by tools such as Fluentd and Logstash and integrate with tracing and metrics-gathering libraries such as OpenTracing, Prometheus, and others.

Treat your application as a black box, but implement all necessary APIs to help the platform observe and manage your application in the best way possible.

LIFE-CYCLE CONFORMANCE PRINCIPLE (LCP)

The HOP dictates that your container provide APIs for the platform to read from. The LCP dictates that your application have a way to read the events coming from the platform. Moreover, apart from getting events, the container should conform and react to those events. This is where the name of the principle comes from. It is almost like having "write API" in your application to interact with the platform.

Figure 3. A container providing APIs and conforming to platform events

There are all kind of events coming from the managing platform that are intended to help you manage the life cycle of your container. It is up to your application to decide which events to handle and whether to react to those events or not.

But some events are more important than others. For example, any application that requires a clean shutdown process needs to catch signal: terminate (SIGTERM) messages and shut down as quickly as possible. This is to avoid the forceful shutdown through a signal: kill (SIGKILL) that follows a SIGTERM.

There are also other events, such as PostStart and PreStop, that might be significant to your application life-cycle management. For example, some applications need to warm up before service requests and some need to release resources before shutting down cleanly.

IMAGE IMMUTABILITY PRINCIPLE (IIP)

Containerized applications are meant to be immutable, and once built are not expected to change between different environments. This implies the use of an external means of storing the runtime data and relying on externalized configurations that vary across environments, rather than creating or modifying containers per environment. Any change in the containerized application should result in building a new container image and reusing it across all environments. The same principle is also popular under the name of immutable server/infrastructure and used for server/host management, too.

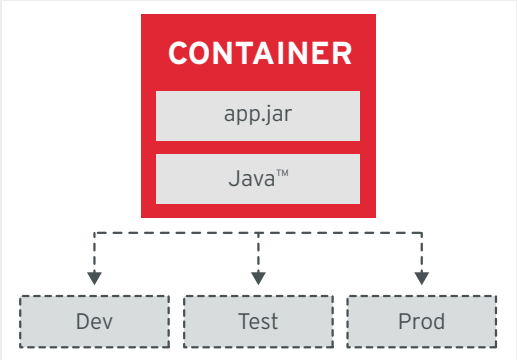

Figure 4. Immutable container images are used across all environments

Following the IIP principle should prevent the creation of similar container images for different environments, but stick to one container image configured for each environment. This principle allows practices such as automatic roll-back and roll-forward during application updates, which is an important aspect of cloud-native automation.

PROCESS DISPOSABILITY PRINCIPLE (PDP)

One of the primary motivations for moving to containerized applications is that containers need to be as ephemeral as possible and ready to be replaced by another container instance at any point in time. There are many reasons to replace a container, such as failing a health check, scaling down the application, migrating the containers to a different host, platform resource starvation, or another issue.

Figure 5. A containerized application with quick start up and shut down for easy replaceability

This means that containerized applications must keep their state externalized or distributed and redundant. It also means the application should be quick in starting up and shutting down, and even be ready for a sudden, complete hardware failure.

Another helpful practice in implementing this principle is to create small containers. Containers in cloud-native environments may be automatically scheduled and started on different hosts. Having smaller containers leads to quicker start-up times because before being restarted, containers need to be physically copied to the host system.

SELF-CONTAINMENT PRINCIPLE (S-CP)

This principle dictates that a container should contain everything it needs at build time. The container should rely only on the presence of the Linux® kernel and have any additional libraries added into it at the time the container is built. In addition to the libraries, it should also contain things such as the language runtime, the application platform if required, and other dependencies needed to run the containerized application.

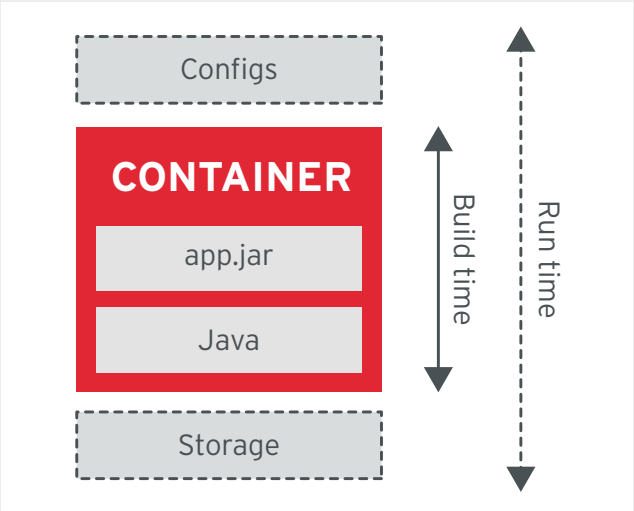

Figure 6. Containers should have baked in all dependencies at build time except environment-specific ones.

The only exceptions are things such as configurations, which vary between different environments and must be provided at runtime; for example, through Kubernetes ConfigMap.

Some applications are composed of multiple containerized components. For example, a containerized web application may also require a database container. This principle does not suggest merging both containers. Instead, it suggests that the database container contain everything needed to run the database, and the web application container contain everything needed to run the web application, such as the web server. At runtime, the web application container will depend on and access the database container as needed.

RUNTIME CONFINEMENT PRINCIPLE (RCP)

S-CP looks at the containers from a build-time perspective and the resulting binary with its content. But a container is not just a single-dimensional black box of one size on the disk. Containers have multiple dimensions at runtime, such as memory usage dimension, CPU usage dimension, and other resource consumption dimensions.

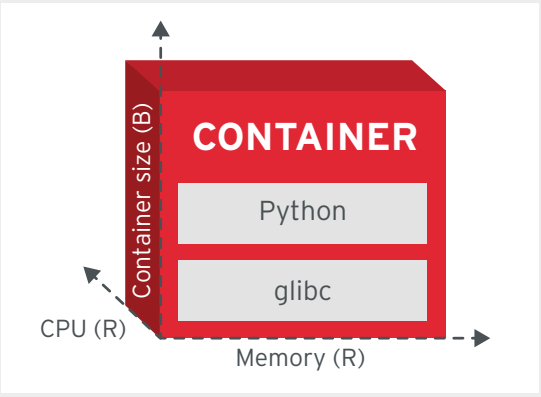

Figure 7. A container that declares its runtime resource requirements and respects them at runtime

This RCP principle suggests that every container declare its resource requirements and pass that information to the platform. It should share the resource profile of a container in terms of CPU, memory, networking, disk influence on how the platform performs scheduling, auto-scaling, capacity management, and the general service-level agreements (SLAs) of the container.

In addition to passing the resource requirements of the container, it is also important that the application stay confined to the indicated resource requirements. If the application stays confined, the platform is less likely to consider it for termination and migration when resource starvation occurs.

CONCLUSION

Cloud native is more than an end state—it is a way of working. This whitepaper described a number of principles that represent foundational guidelines that containerized applications must comply with in order to be good cloud-native citizens.

In addition to those principles, creating good containerized applications requires familiarity with other container-related best practices and techniques. While the principles described above are more fundamental and apply to most use cases, the best practices listed below require judgment on when to apply or not apply. Here are some of the more common container-related best practices:

- **Aim for small images.** Create smaller images by cleaning up temporary files and avoiding the installation of unnecessary packages. This reduces container size, build time, and networking time when copying container images.
- **Support arbitrary user IDs.** Avoid using the sudo command or requiring a specific userid to run your container.
- **Mark important ports**. While it is possible to specify port numbers at runtime, specifying them using the EXPOSE command makes it easier for both humans and software to use your image.
- **Use volumes for persistent data**. The data that needs to be preserved after a container is destroyed must be written to a volume.
- **Set image metadata.** Image metadata in the form of tags, labels, and annotations makes your container images more usable, resulting in a better experience for developers using your images.
- **Synchronize host and image.** Some containerized applications require the container to be synchronized with the host on certain attributes such as time and machine ID.

Here are links to resources with patterns and best practices to help you implement the above-listed principles more effectively:

- https://www.slideshare.net/luebken/container-patterns
- https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices
- http://docs.projectatomic.io/container-best-practices
- https://docs.openshift.com/enterprise/3.0/creating_images/guidelines.html
- https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_burns.pdf
- https://leanpub.com/k8spatterns/
- https://12factor.net/

NORTH AMERICA
1 888 REDHAT1

EUROPE, MIDDLE EAST, AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
info-latam@redhat.com

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#f8808_1017