Object-inspired container design patterns

Abhishek Tiwari

Citation: *A. Tiwari*, "Object-inspired container design patterns", Abhishek Tiwari, 2017. doi:10.59350/pmv7s-vf080

Published on: March 28, 2017

If we start thinking containers in terms of objects, it abstracts away the low-level details but reveals higher-level patterns that are common to a variety of applications and algorithms. A recent paper, published by Google Researchers Brendan Burns and David Oppenheimer, suggests that *containers are particularly well-suited as the fundamental "object" in distributed systems by virtue of the walls they erect at the container boundary*. In many ways, this paper is seminal research publication and authors believe that containers are destined to become analogous to objects in object-oriented software systems.

Burns and Oppenheimer describe three types of design patterns: 1) single-container patterns for container management, 2) single-node patterns of closely cooperating containers, and 3) multi-node patterns for distributed algorithms. Inspired by various well-documented object-oriented patterns, these container patterns encode best practices, simplify development, make distributed computing more reliable.

Single-container management patterns

Containers provide interfaces that can be used to define application-specific functionality (aka upward APIs) as well as to interact with management systems (aka downward API). Using upward APIs, developers can access a range of application details, including application-specific monitoring metrics (QPS, application health, etc.), profiling information (threads, stack, lock contention, network message statistics, etc.), component configuration information, and component logs. For instance, Kubernetes allows developers to define health checks via specified HTTP endpoints (e.g. `/health`).

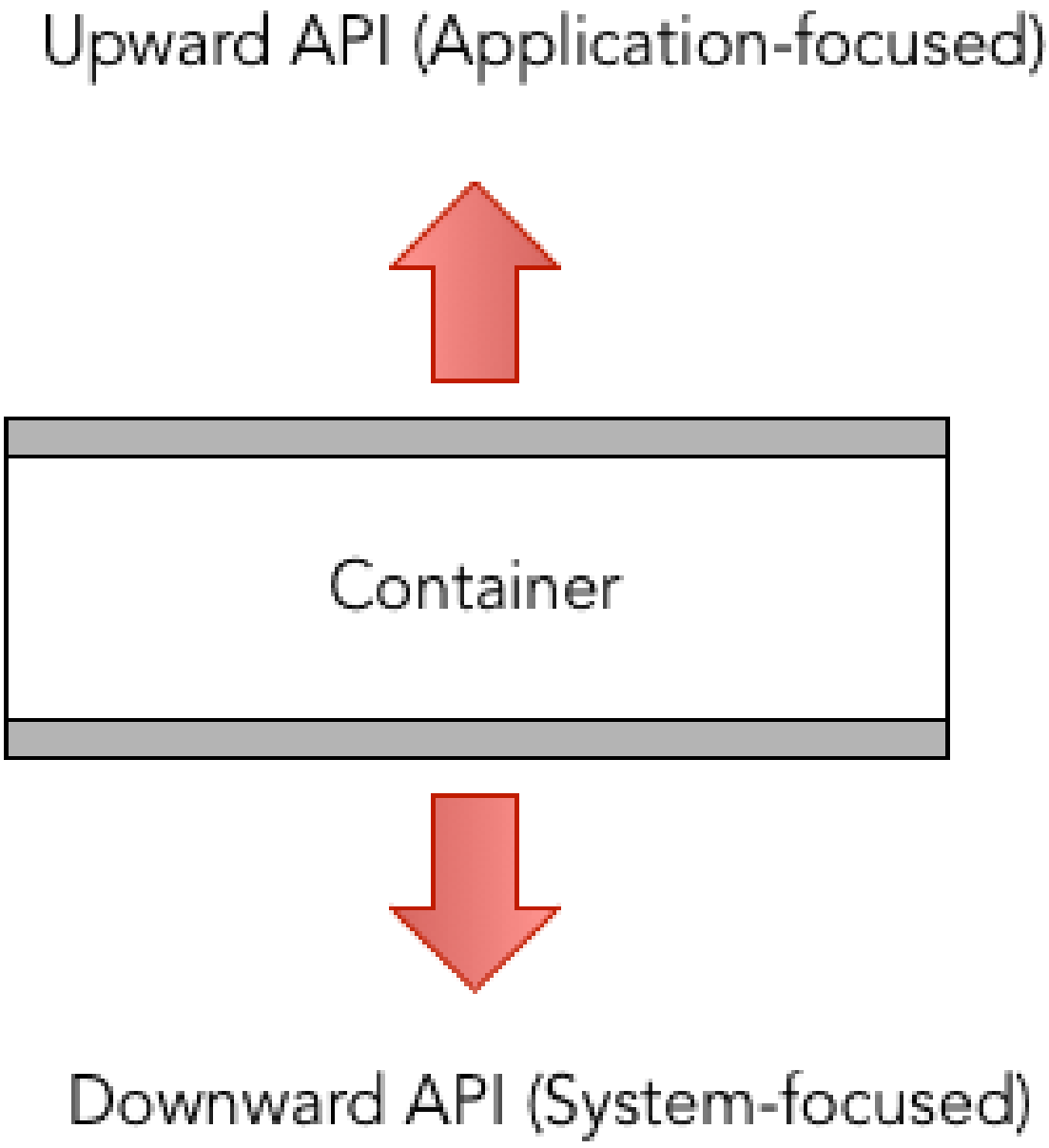

Figure 1: Upward and Downward API Pattern

Containers also provide downward APIs to manage container lifecycle which is controlled by a container cluster management system. Downward APIs also enables to create life cycle contracts between application and cluster management system which can be very useful to maintain application and container reliability. For example, Kubernetes issues `SIGTERM` signal before `SIGKILL` signal

which allows the application to terminate cleanly by finishing in-flight operations, flushing state to disk, etc. Similarly, Downward APIs are useful for a container to have information about itself or the system. For instance, downward API that a container might support is “replicate itself” in response to an auto scaling trigger.

Single-node, multi-container application patterns

In this pattern, multiple related or interdependent containers are co-scheduled onto a single host machine. In Kubernetes, this abstraction is called Pods. For instance, a Pod may have one container for Nginx as a proxy, a second container for the Application server, a third container for MySQL database server, and a persistent volume to store MySQL data.

Sidecar pattern

In a sidecar pattern, the functionality of the main container is extended or enhanced by a sidecar container without strong coupling between two. Although it is always possible to build sidecar container functionality into the main container, there are several benefits with this pattern,

- different resource profiles i.e., independent resource accounting and allocation
- clear separation of concerns at packaging level i.e., no strong coupling between containers
- reusability i.e., sidecar containers can be paired with numerous different “main” containers
- failure containment boundary, making it possible for the overall system to degrade gracefully
- independent testing, packaging, upgrade, deployment and if necessary roll back

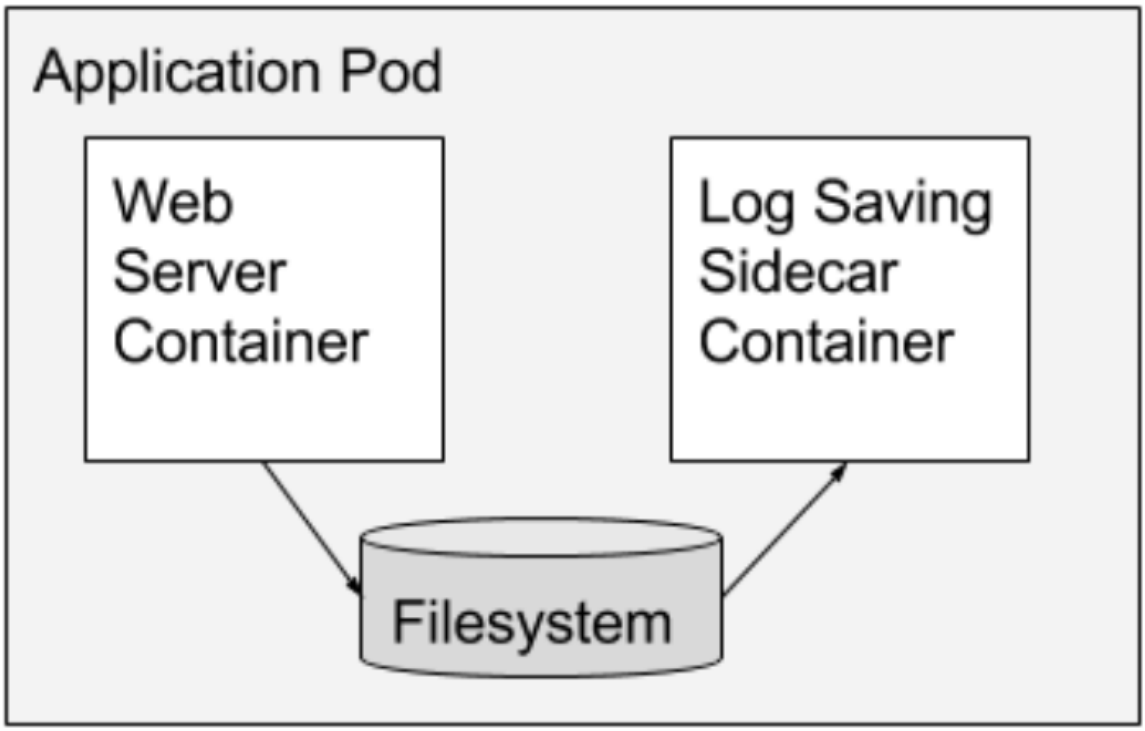

Figure 2: An example Sidecar pattern: here main container is a web server which is paired with a log saver sidecar container that collects the web server's logs from local disk and streams them to centralized log collector.

Ambassador pattern

In this pattern, an Ambassador container act as a proxy between two different types of main containers. Typical use case involves proxy communication related to load balancing and/or sharding to hide the complexity from the application.

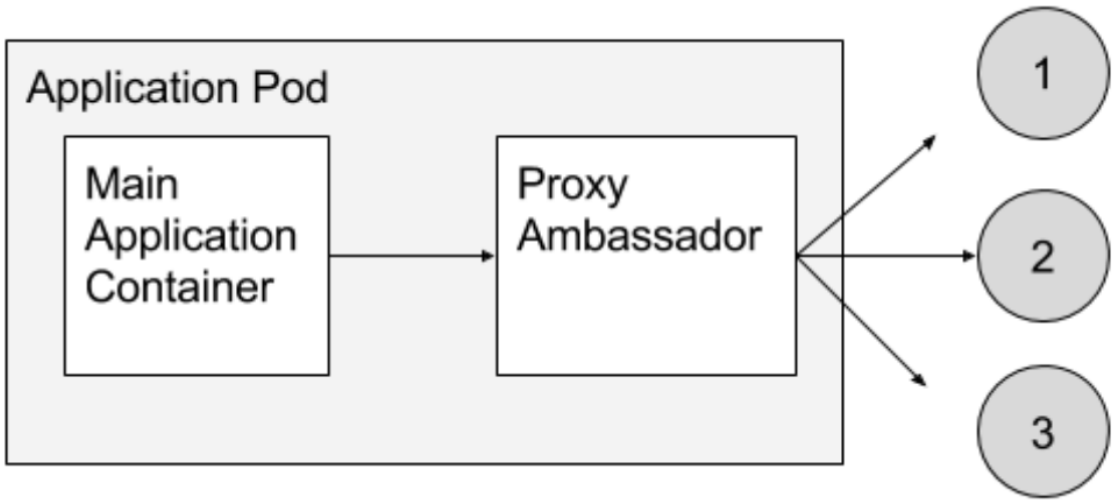

Figure 3: An example of the ambassador pattern: an application is speaking the memcache protocol with a twemproxy ambassador. In reality, twemproxy is sharding the requests across a distributed installation of multiple memcache nodes elsewhere in the cluster.

Adapter pattern

In this pattern, an adapter container offers standardized output and interfaces across multiple heterogeneous main application containers.

> In contrast to the ambassador pattern, which presents an application with a simplified view of the outside world, adapters present the outside world with a simplified, homogenized view of an application. They do this by standardizing output and interfaces across multiple containers.

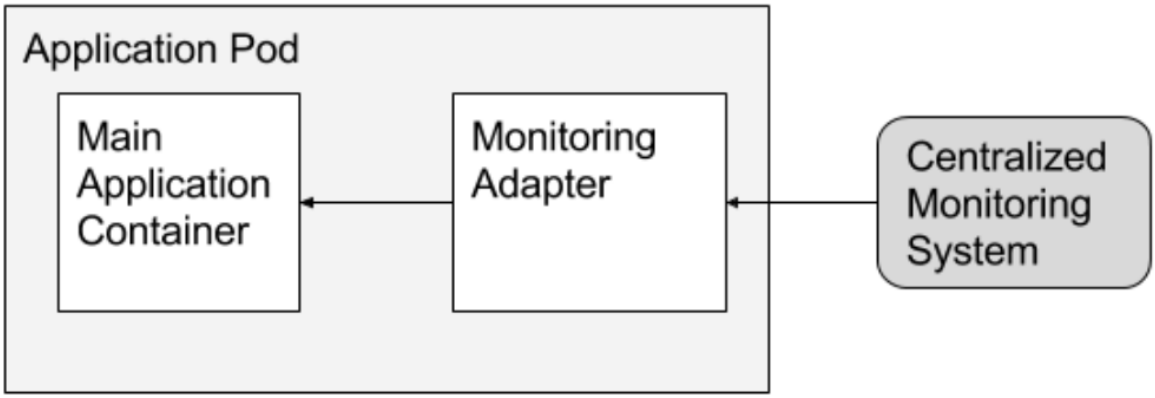

Figure 4: An example of the adapter pattern: adapters that ensure all containers in a system have the same monitoring interface. The main container can communicate with the adapter through localhost or a shared local volume.

Multi-node application patterns

Modular containers enable us to build coordinated multi-node distributed applications. Obviously, multi-node application patterns require Pod like abstraction.

Leader election pattern

In this pattern, the leader-election containers serve as sidecar containers and are co-scheduled with main application containers those require leader election. Thes leader-election containers can perform election amongst themselves, and they can present a simplified HTTP API over localhost to each application container that requires leader election (e.g. becomeLeader, renewLeadership, etc.). Leader-election containers can be re-used by different type of main applications

Work queue pattern

In this pattern, one can implement a generic work queue framework using containers that implement the `run()` and `mount()` interfaces and can take arbitrary processing code packaged as a container, and arbitrary data, and build a complete work queue system.

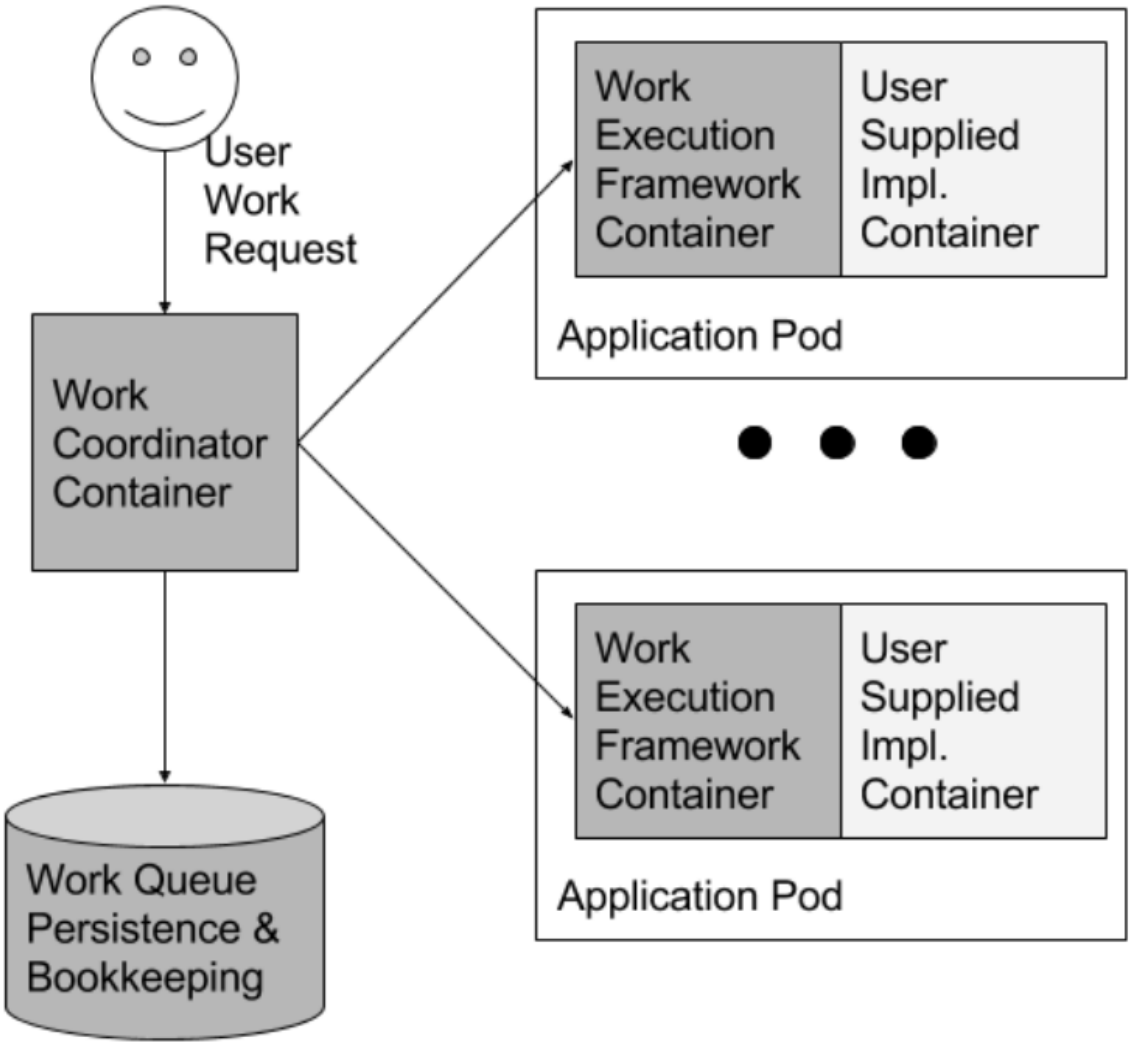

Figure 5: An example of the Work queue pattern: reusable work execution framework containers pull task and associated data to local filesystem which is used by user-supplied implementation container to perform actual task.

Scatter/gather pattern

This pattern is quite similar to MapReduce framework. Here an external client sends an initial request to a framework supplied `root` container. This root fans out the request to a large number of developer-supplied `leaf` containers to perform computations in parallel. Then another developer-supplied `merge` container responsible for merging or aggregate the results. You can think `root` container as map step and `merge` container as a reduce step in MapReduce paradigm. You can extend

this pattern, by introducing a `shuffle` container which is similar to a shuffle step in MapReduce process.

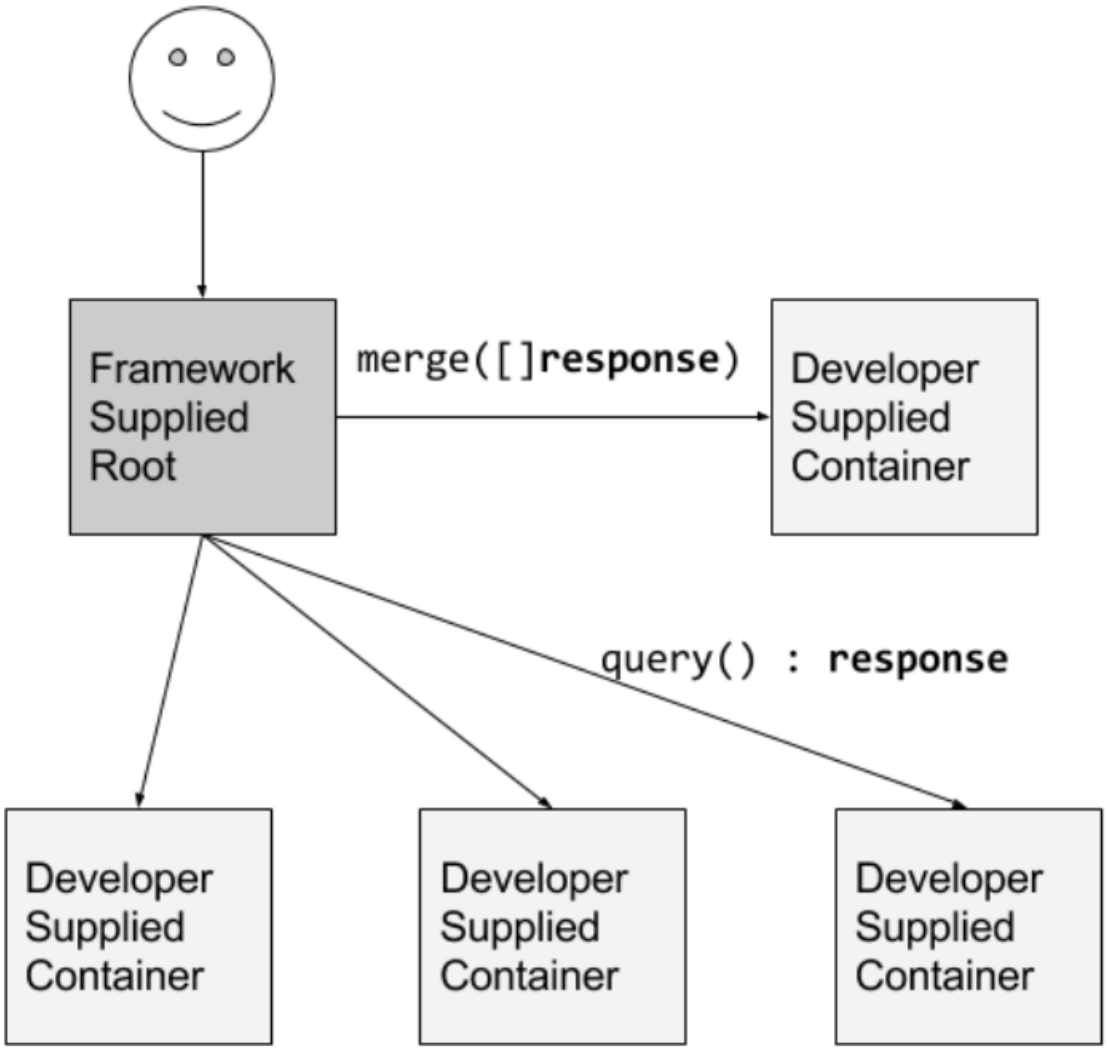

Figure 6: An example of the scatter/gather pattern: a reusable root container implements client interactions and request fan-out to developer-supplied leaf containers and to a developer-supplied merge container responsible for aggregating the results

Conclusion

As you can see here, containers provide many of the same benefits as objects in object-oriented systems, i.e. abstraction, encapsulation, reusability and so on.