Job Aid: Architectural Styles in microServices

Continuous Deployment Platform

Dewayne Hafenstein

Principal Technical Architect

Contents

Architectural Styles Overview

Architectural Styles

- Architectural styles are used to solve specific types of business and/or technical requirements related to running an application.
- Architectural styles are methods of organizing all of the parts of an application (user interface, application logic, data management).
- Each architectural style has its strengths and weaknesses.
- Each architectural style demonstrates different ways of constructing the application

Architectural Styles are used in the following ways:

- As approaches to structuring an application.
- To solve issues around scalability, availability, reliability, class-of-service, time-to-deliver, and others.

Architectural styles are not absolute; there is no one-size-fits-all solution.

Architectural Styles Overview, continued

Some factors that govern the choice of architectural style include:

- Ease of installation
- Maintenance
- Performance
- Scalability
- Ability to secure
- Control
- Monitoring
- Availability

…….and many others.

No single architectural style fits every application need.

Architectural Styles Overview, continued

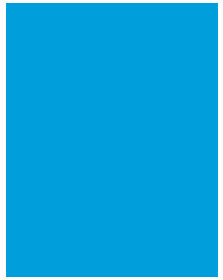
Many Different Implementation Technologies

- There are a lot of choices.
- There is a lot of overlap between technologies.
- Use of implementation technologies depends on needs as well as architectural style.
- No single architectural style fits every application need.
- Different architectural styles can require certain implementations that others do not.
  - For example, a monolithic implementation would not likely require an API, but a distributed component-based implementation would (more on the different architectural styles later)

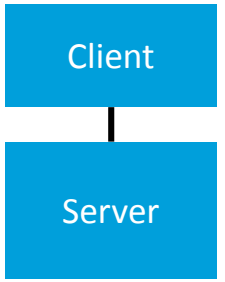
Common Architectural Styles

– We introduce here some of the more common architectural styles, just to compare and contrast them with the service oriented architecture (SOA) style, and the microService implementation of SOA in particular.
  - Monolithic
  - 2-tier / Client-Server
  - N-tier / Distributed
  - Peer-to-Peer / Networked
  - Services Oriented Architecture (SOA)
  - And there are others…

Common Architectural Styles, continued



Monolithic



2-Tier

Monolithic Applications FEATURES:
All aspects of the application, including its user interface, business application logic, and data management are bundled together in one package.

UPSIDE: Monolithic applications are generally easy to install and operate, and they generally have higher performance.

DOWNSIDE: Monolithic applications generally have more difficulty in data sharing, scalability, and monitoring.

2-Tier FEATURES:
Also known as client-server. The client component (usually the user interface) is separated from the business logic and replicated and distributed. The client communicates with the server over a network. The server contains all of the business logic and data management.

UPSIDE: 2-Tier applications generally have much better data sharing across multiple users, can better utilize hardware assets, and have better monitoring and security.

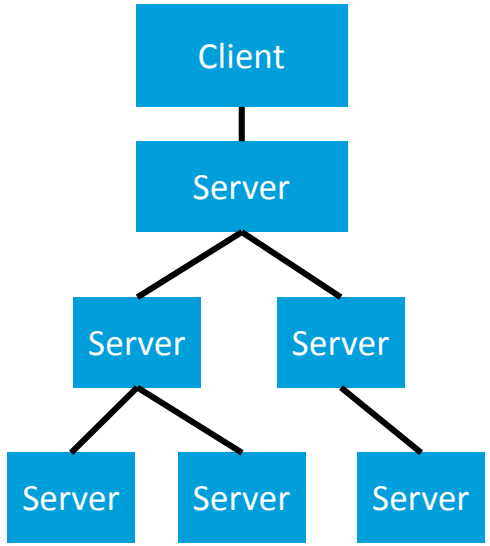
DOWNSIDE: Performance may not be quite as good and the installation and support is a little more complicated. The use of a browser-based UI tends to offset this complexity. Scalability can be difficult. Availability (fault-tolerance) can be difficult.

Common Architectural Styles, continued

N-Tier FEATURES:
N-Tier is a variation of 2-tier, where the business logic and data management are also broken up, replicated, and distributed. All of the components communicate with each other over the network.

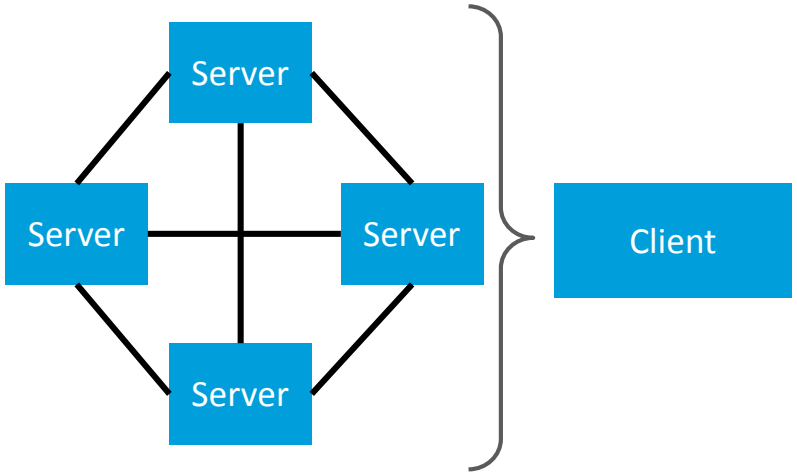
UPSIDE: This style tends to improve scalability and fault-tolerance (components can be replicated with automatic fail-over)

DOWNSIDE: This style adds complexity and complicates monitoring and support. This typically requires more advanced and complex runtime management frameworks such as application servers, which manage the deployed components. Performance may not be as good as monolithic or 2-tier.



N-Tier

Common Architectural Styles, continued



Peer-to-Peer
Replicated Servers

Peer-to-Peer:
This architectural style is a variation of monolithic where each peer is completely self-contained and implements **all** capabilities of the application. This is different from a component or service oriented architecture where the application logic is decomposed and distributed. In this architecture, every peer implements all functionality. Peers are then replicated and distributed across a network. Requests are distributed across the peers for processing.

UPSIDE: This is often used in clustering technologies where each member of the cluster is equally capable of processing any request. Performance on this architecture is generally good, as is scalability and fault-tolerance.

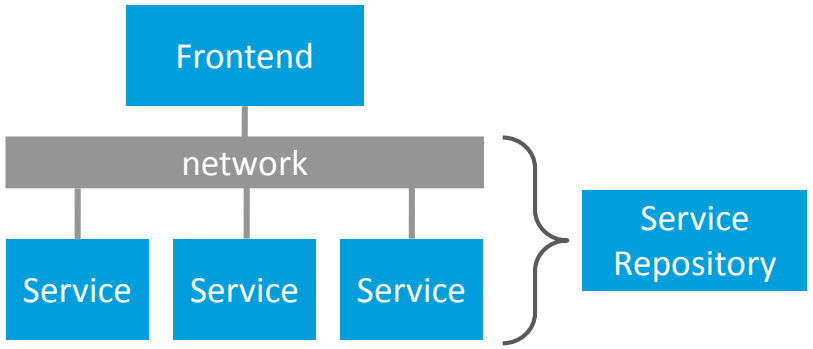
DOWNSIDE: Installation, monitoring, and support (as well as development) are somewhat more complicated.

Common Architectural Styles, continued

Service-Oriented Architecture (SOA):
Breaks up a business domain into reusable services that represent domain-specific functions like shipping, receiving, billing, order management, etc. Assembles different services to create applications by using the services in specific ways to perform business activities. Registers the services in a repository when instantiated, and locates them as needed via the repository. Services exist in a dynamic, changing environment with fault-tolerance, scalability, and other desirable goals. All service interaction is over the network to allow for location independence.

UPSIDE: Better fault-tolerance, load-balancing, and scalability and improvements in time-to-market because services can be developed and tested independently, and installed separately.

DOWNSIDE: High complexity, and monitoring and support can be complex. Performance is not generally as good as monolithic or 2-tier. Reuse of domain logic can be good.



Service Oriented Architecture

Service Oriented Architecture Overview

Service Oriented Architecture

- An application is *assembled* by integration of services (usually by a front-end) to coordinate and implement the application.
- Each service exposes an API to access that service.
- All interaction with the service is via the API.
- Services are discovered using a registry.
- Services are accessed over the network.
- Services are black-boxes.

In SOA, an application is created by assembling, or using, one or more domain services to perform activities needed by the application. An application in the typical sense becomes a coordinator of the different domain services, and usually is the front-end that the end user interacts with. The application itself may be a monolithic, 2-tier, or n-tier (or other styles as well) implementation, and could be another service. The interaction with the services is over a network, and the services are discovered using a lookup in a repository of available services. The end user rarely is even aware of the use of the services.

microServices are ***AN*** **implementation of the SOA architectural style**.

Challenges with SOA

SOA has some very promising benefits and a considerable amount of complexity as well.

In the past, there was no standard framework, so many organizations had to create their own. That introduces other complexities. Custom implementations of frameworks reduced the pool of knowledgeable resources because you could not hire people that were immediately productive in your framework. Costs of supporting the framework added to the costs of applications and services. Also, implementations may require service and application developers to interact with and be aware of the underlying frameworks and technologies.

In many cases the services had to handle their own lifecycles, registering and unregistering their existence, and interacting with the operational environment. Logging and monitoring was not standardized and varied widely. Diagnostics were difficult to obtain and to understand, especially if services move during analysis.

And to add more complexity, the promise of service reuse requires a different view of the business, called domain-driven design, with which many analysts and developers have somewhat novice experience. Implementation of services may require too much knowledge of the “framework”.

Challenges with SOA, continued

Here are some of the challenges with SOA:

- It is hard to implement automatic scalability and recovery.
- Portability may be difficult to achieve.
- Custom implementations can be very involved and complex.
- Reusability is hard to achieve without good domain-driven design.

microServices Implementations

microServices Addresses Challenges

microServices are an architectural style that defines how the services themselves are packaged, deployed, discovered, executed, secured, and monitored. Using standard frameworks can allow the service to be infrastructure neutral, or unaware.

- Using runtime management, services can be automatically scaled based on load and recovered after a failure.
- Using standard frameworks and runtime management can allow services to be deployed on any platform, physical or virtual, and of different operating systems. The standard frameworks are also being widely adopted across nearly all platforms, with portability being a major goal. This means it will be possible to create a service and deploy it on any Unix variant, as well as Windows™, and even on z-OS mainframe systems if needed.
- By using some standard frameworks, and developing some standard practices and tools around these technologies, many of the complexities of SOA (including cost) can be dramatically reduced.
- Also, these standard frameworks (such as Docker) utilize a container approach, and tend to greatly simply service code. The container removes most, if not all, of the interaction with the infrastructure from the service and handles that for it.

Some Challenges Remain

- Domain-Driven Design
- Automation (addressed by CDP)
- Development and testing processes (addressed by CDP)