

CSCI 5409 Adv. Topics in Cloud Computing – Fall, 2023
Week 13 – Lecture 1 (Nov 27, 2023)

Microservices (1)

Dr. Lu Yang
Faculty of Computer Science
Dalhousie University
luyang@dal.ca

Microservices

Cloud Computing

Slides prepared by Dr. Tami Meredith

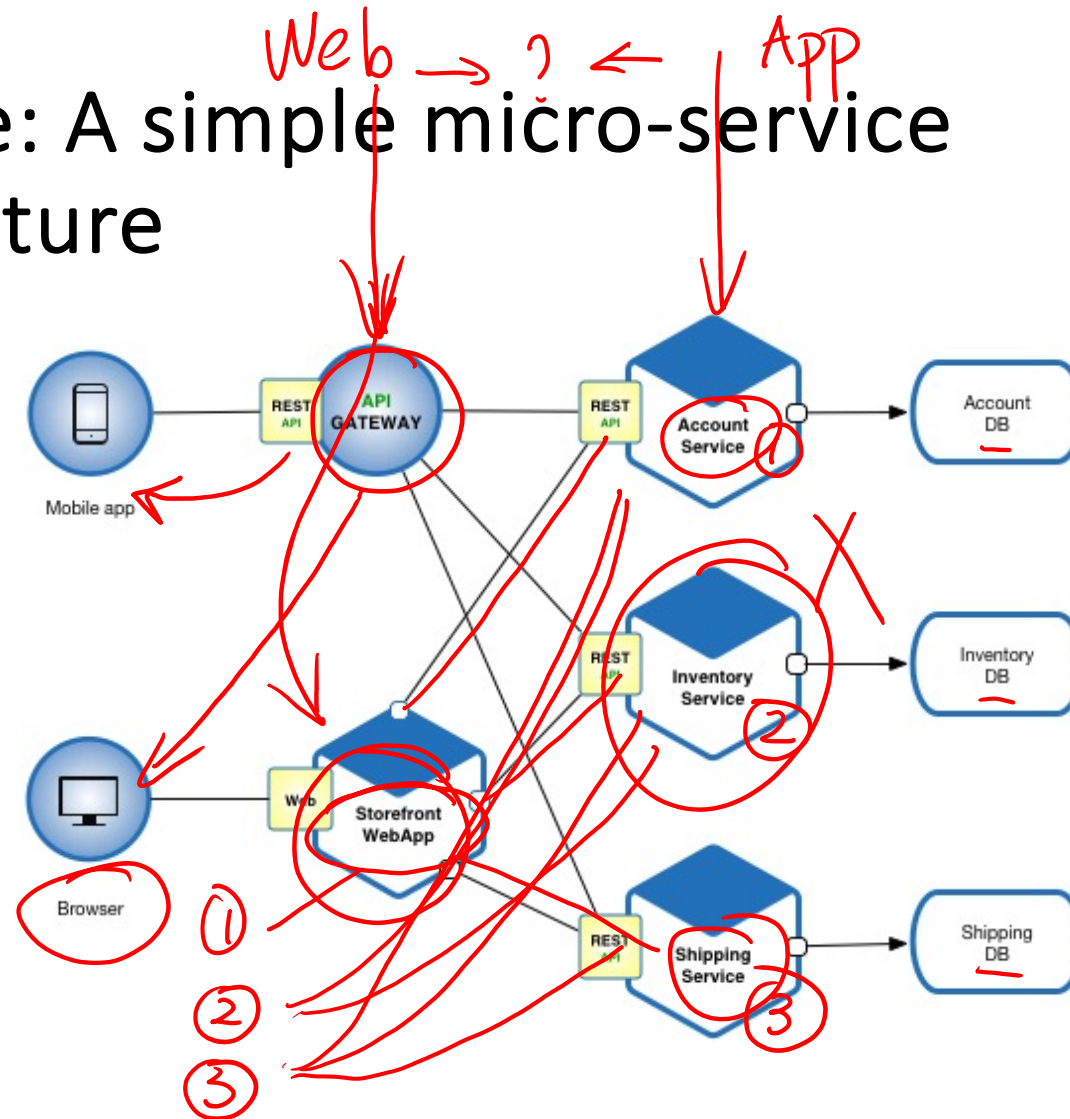
Housekeeping and Feedback

- Start recording
- Starting working on the term project ASAP. Ask Purvesh and Rahul questions.
- Term project report and CloudFormation video due at the due time. But the video demo is whenever it is scheduled.
- SLEQ
- PIER tour

What are Microservices?

- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:
 - Independently deployable,
 - Loosely coupled,
 - Organized around business capabilities,
 - Owned by a small team, and
 - Highly maintainable and testable.
- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

Example: A simple micro-service architecture



Characteristics of Micro-Services

1. Autonomous

- Each service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the functioning of other services.
- Services do not need to share any of their code or implementation with other services. Any communication between services happens via well-defined APIs.

2. Specialized

- Each service is designed for a set of capabilities and focuses on solving a specific problem.
- If developers contribute more code to a service over time and the service becomes complex, it can be broken into smaller services.

A few companies that use a Microservice Architecture:

- Coursera
- Localytics
- Remind
- Airtime
- Lyft, Uber
- Shippable
- Gilt
- Netflix
- Etsy
- Amazon

For more, see: <https://blog.dreamfactory.com/microservices-examples/>

BENEFITS of microservices




CI/cD

Enables the continuous delivery and deployment of large, complex applications with:

- Improved maintainability - each service is relatively small and so is easier to understand and change.
- Better testability - services are smaller and faster to test.
- Better deployability - services can be deployed independently. You don't have to deploy the entire app for an update, just the service that is updated.
- Improved fault isolation and tolerance. For example, if there is a memory leak in one service then only that service will be affected. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.

Programmers are happier and accomplish more because:

- Each microservice is relatively small:
 - Easier for a developer to understand – the service does one thing.
 - The application starts faster, which makes developers more productive, and speeds up deployments.
 - Enables you to organize the development effort around multiple, autonomous teams.
 - Each team owns and is responsible for one (or more) services.
 - Each team can develop, test, deploy and scale their services independently of the other teams.
 - Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a different technology stack.
- 

Microservices improve control:

per service

- Improved scaling with a finer level of control. Services can be scaled independently, allowing better control of costs – only scale the services that need scaling.
- Improved overall control. You can tailor the memory, environment, storage, etc., to the service.
- Deployment, updates, and maintenance can be done on a per-service level, minimizing the impact upon the overall system.
- Monitoring and logging can be independent for each service. You get a better view of where your application is experiencing issues and will be able to see and address issues faster.

Benefits: Summary

Agility: Microservices foster an organization of small, independent teams that take ownership of their services. Teams act within a small and well understood context and are empowered to work more independently and quickly which shortens development cycle times.

Flexible Scaling: Microservices allow each service to be independently scaled to meet demand for the application feature it supports and thus enables teams to right-size infrastructure needs, more accurately measure the cost of a feature, and maintain availability if a service experiences a spike in demand.

Easy Deployment: Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work. The low cost of failure enables experimentation, makes it easier to update code, and accelerates time-to-market for new features.

Technological Freedom: Microservices architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool to solve their specific problems. As a consequence, teams building microservices can choose the best tool for each job.

Resilience (Fault Tolerance): Service independence increases an application's resistance to failure. In a monolithic architecture, if a single component fails, it can cause the entire application to fail. With microservices, applications handle total service failure by degrading functionality and not crashing the entire application.



Issues of Concern



Things to be aware of:

Developers must deal with the additional complexity of creating a distributed system:

- Developers must implement the inter-service communication mechanism and deal with partial failure.
- Implementing requests that span multiple services is more difficult.
- Testing the interactions between services is more difficult.
- Implementing requests that span multiple services requires careful coordination between the teams.
- Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Managing "state" is hard! Services tend to be stateless for efficiency.

Stateless Communication

Communication must be stateless in nature, such that each request from client to the service must contain all of the information necessary to understand the request, and cannot take advantage of any stored communication context on the service.

In plain English → When you request data from a service it will send all the data and then end the connection.

More simply: The service has no "memory" of any interaction with a client.

Service has no: session or connection persistence, basket/shopping cart, etc. However, it must perform authentication of requests.

Stateless services do not need to provide "client stickiness" allowing better load balancing and scaling.

Complexity

Deployment complexity.

- In production, there is also the operational complexity of deploying and managing a system comprised of many different services.

Increased memory consumption.

- The microservice architecture replaces N monolithic application instances with NxM service instances. If each service runs in its own VM (EC2 instance or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many VM runtimes.
- Each service will likely require its own load balancer, scaling listener, and so on. There are simply more parts to deal with.

Issues

The architecture introduces additional complexity and new problems to deal with, such as:

- Internal network latency,
- message format design,
- backup/availability/consistency,
- increased network traffic (and potential performance issues as a result),
- increased number of interface points (which can increase architectural complexity),
- Increased load balancing, and
- fault tolerance.

All of these problems have to be addressed at scale.

The complexity of a monolithic application does not disappear if it is re-implemented as a set of microservices and may actually increase due to network and communications needs.

Excessive Independence

- Services, if not coordinated, can lead to poor code re-use. Use of shared libraries will help alleviate this issue.
- Teams tend to become “siloed.” Knowledge can be accidentally hidden from other teams due to each team’s independence.
- Excessive use of technology may occur. While “the right tool for the job” is important, teams do not tend to see what each other does, sometimes causing an unnecessary (and expensive) proliferation of tools and libraries. This issue, can in turn, make it harder to share code and knowledge.

Team coordination and oversight is critical to prevent these issues.