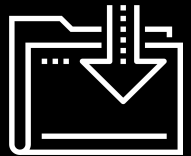


Twelve-Factor, Cloud-Native, and Microservices

Course: Java

S1



The background is a dark charcoal gray with a series of parallel diagonal lines running from the top-left to the bottom-right. Overlaid on this are several teal-colored geometric shapes: a large central triangle pointing right, a smaller triangle to its left, and a square to its right. Scattered around these shapes are various white line-art symbols, including a plus sign, a minus sign, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, a circle with a cross, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, a circle with a cross, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, and a circle with a cross.

WELCOME

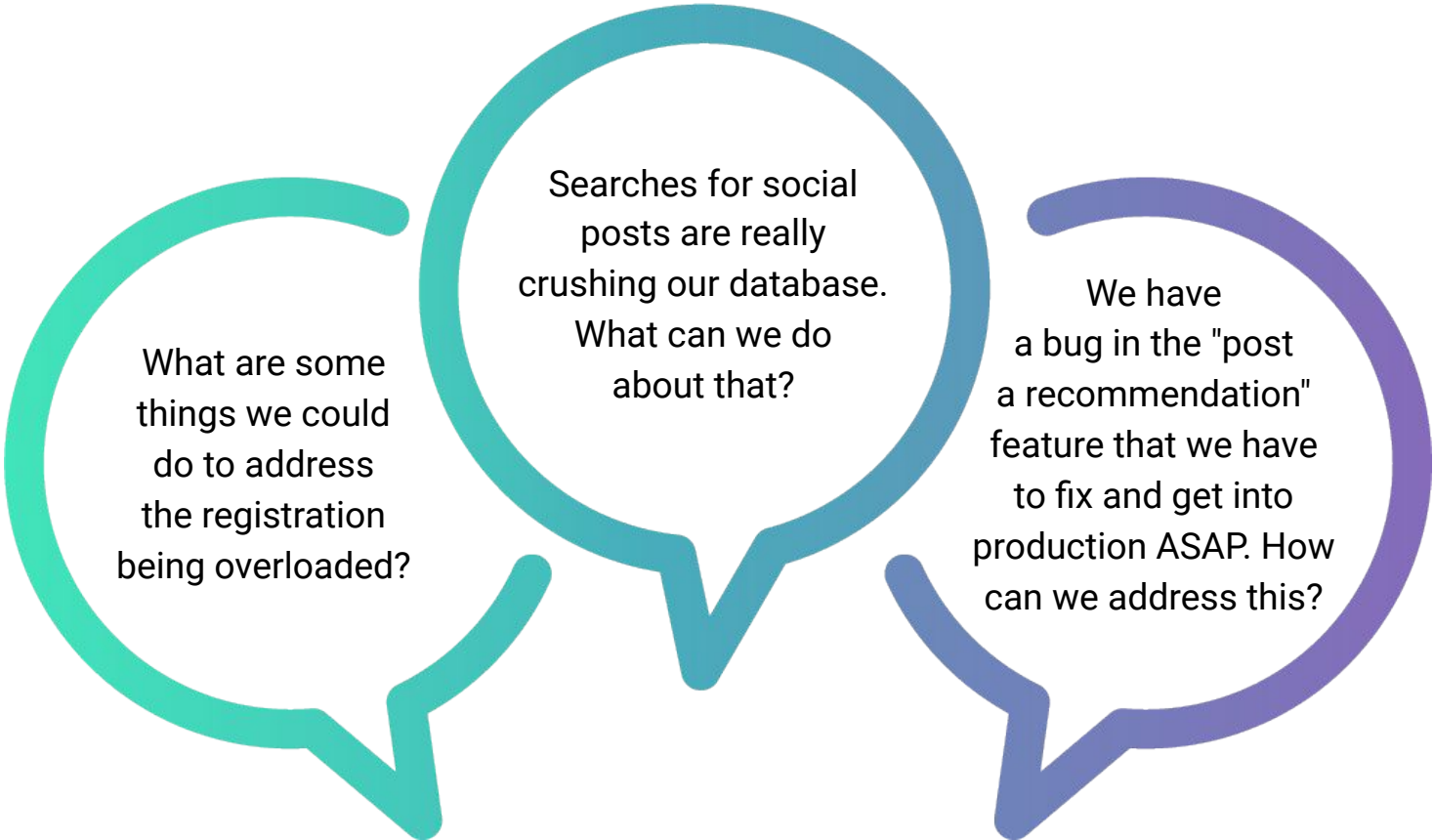


Activity: Brainstorm Solutions

Suggested Time:

5 Minutes

Discuss



What are some things we could do to address the registration being overloaded?

Searches for social posts are really crushing our database. What can we do about that?

We have a bug in the "post a recommendation" feature that we have to fix and get into production ASAP. How can we address this?

Learning Outcomes

By the end of this lesson, you will be able to:

01

Explain the origins and purpose of the twelve-factor app.

02

Explain the origins and purpose of cloud-native applications.

03

Explain the purpose of microservices.

The Twelve-Factor App

“Our motivation is to raise awareness of some systemic problems we’ve seen in modern application development, to provide a shared vocabulary for discussing those problems, and to offer a set of broad conceptual solutions to those problems with accompanying terminology.”

<https://www.12factor.net/>



General Points

01

This concept was originally proposed by developers at Heroku.

02

Adam Wiggins first presented it in 2011.

It is an approach for building web-based software as a service (SaaS) applications for maximum portability, scalability, resilience, and fit for deployment to cloud platforms.

03

The idea for the twelve-factor app is to outline some problems that can occur in modern application development and offer some pattern-based solutions for them. According to the website, the format is inspired by Martin Fowler's *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002).

The Twelve Factors (1-4)

01

Codebase: We have a good start on this because we're checking our code into GitHub.

02

Dependencies: Maven and Spring Boot help us out with this factor.

03

Config: We'll address this factor with configuration servers.

04

Backing Services: We'll address this by using a service registry (Eureka) for everything.

The Twelve Factors (5-8)

05

Build, release, run: We'll address this with Jenkins and pipelines.

06

Processes: We'll follow these guidelines when using Cloud Foundry (no disk, etc.). We'll set up our applications as a set of independent microservices.

07

Port binding: We'll address this by using config servers, service registries, and microservices.

08

Concurrency: The microservices approach allows us to scale horizontally and have our services be independent. Breaking the monolith allows us to scale only parts of the application where needed.

The Twelve Factors (9-12)

09

Disposability: We'll strive for this. Spring Boot helps, and the other services we will use (queues, service registry, etc.) are built for this.

10

Dev/prod parity: This is something that must be designed into the whole system. We'll look at pieces that help with that.

11

Logs: We'll look at techniques for this approach with Java, Spring Boot, and Cloud Foundry.

12

Admin processes: Again, part of the overall design and approach. A bit out of scope for this class.



**The twelve-factor app methodology
is programming-language agnostic.**

Cloud-Native Applications

Cloud-Native Concepts



Cloud-native is related to twelve-factor and solves some of the same problems.



Cloud-native applications and teams take advantage of the patterns of twelve-factor, devops, and microservices that improve scalability, reliability, and agility.



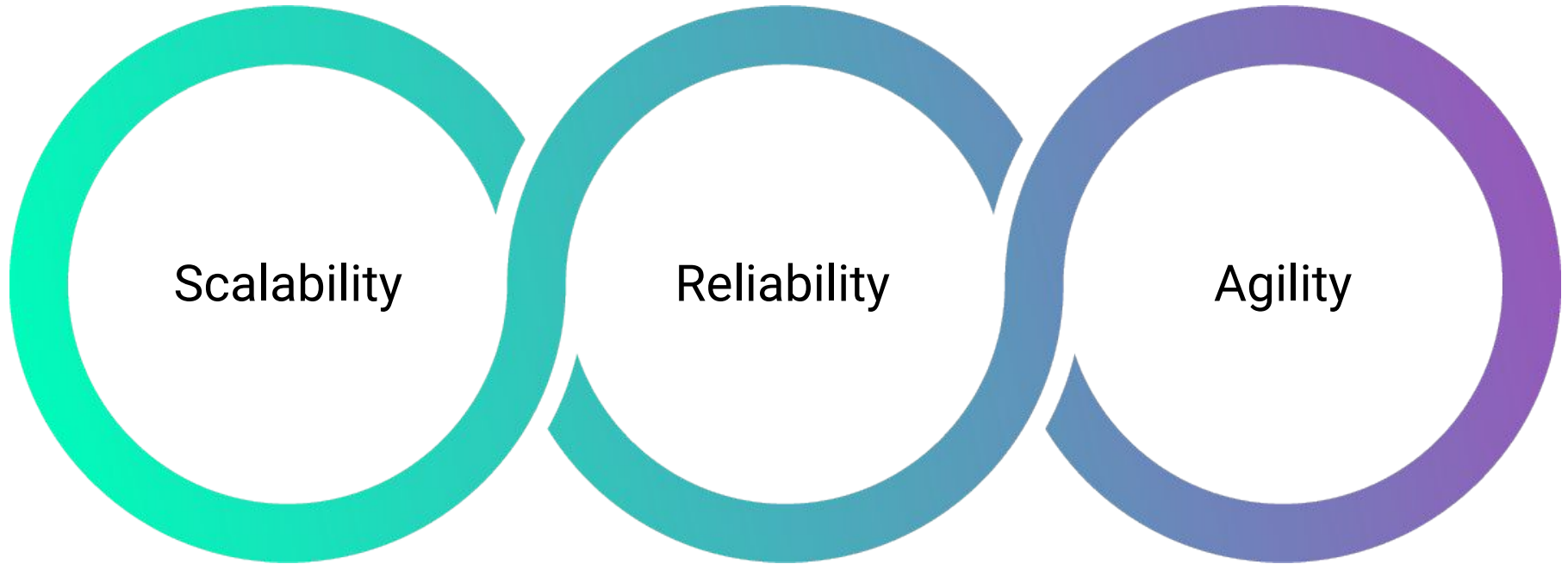
Cloud-native applications and teams take advantage of the patterns of twelve-factor, DevOps, and microservices that improve scalability, reliability, and agility.



Cloud-native applications and teams take advantage of the patterns of twelve-factor, devops, and microservices that improve scalability, reliability, and agility.

- Netflix has open-sourced many of their tools; we'll use some in class.

Cloud-Native Main Ideas



Scalability

The ability to scale capacity in production and to scale the development of more components.

Reliability

Increasing the predictability of a system, reducing failures in the system, and having the ability to recovery from those failures when they do occur.

Agility

The ability to deliver new features quickly and reliably.

The Cloud-Native Approach

Ideas or principles

These define what we are optimizing for. The cloud-native approach values independence among components so that they can be quickly and easily scaled up/down and delivered individually.

Constraints

The ideas and principles help us form an approach to building our applications. This approach must support the ideas and principles. The approach is expressed as a group of constraints. These can look a lot like the twelve factors. For example: Factor #4, Backing Services, is a constraint.

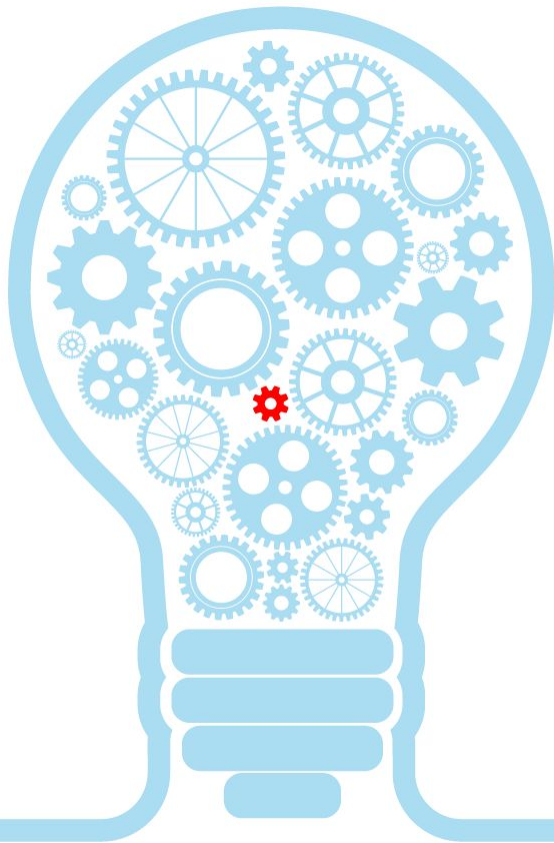
Practices

This is a list of repeatable actions that realize the constraints. These practices tell developers how we create microservices in our environment or how other components in the system interact with databases, etc.

Microservices

What Are **Microservices**?

- Small, independent services that work together to form a bigger application. They are usually web services—most often REST web services.
- They are tightly focused, have high cohesion, and follow the single-responsibility principle.
- Does this sound familiar?
 - They focus on one thing and they do it well.
 - A lot like good class design.



Advantages of Microservices Architecture/Approach



Independent scaling

We can only scale up/down the parts of the system that need to be scaled.



Independent release schedules/cycles

We can introduce new features and bug fixes in a per-microservice basis.



Right tech for right job

Since the contract between parts of the system is REST, we can use the technology that is best suited for each individual web service.



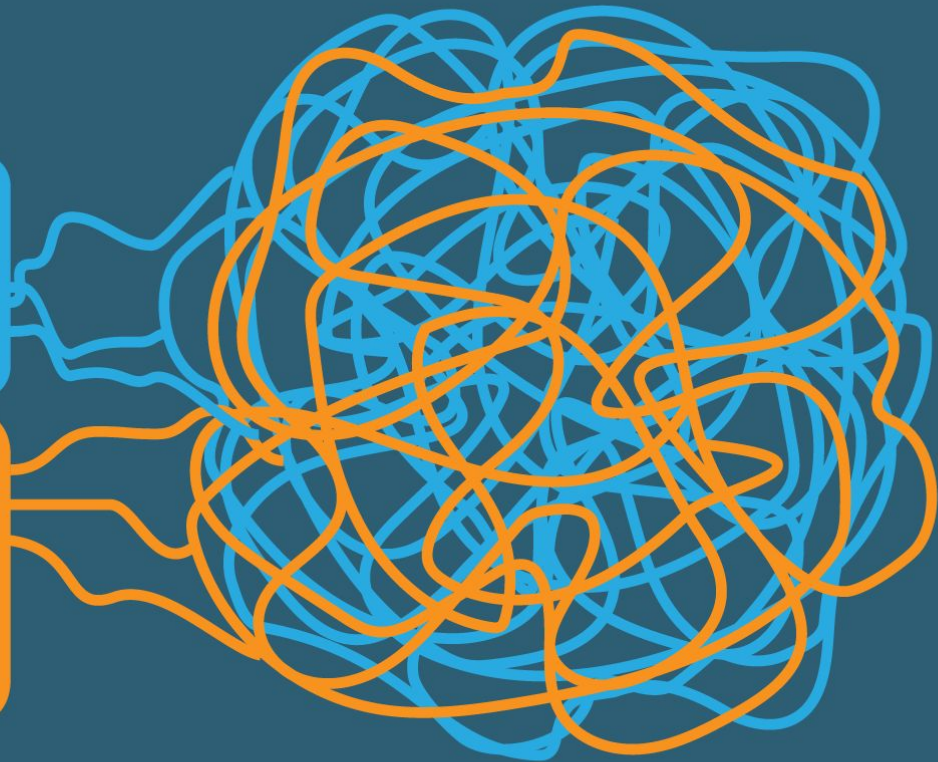
Resilience

We can have multiple copies of each service running at the same time.
Failures in one microservice won't take down other services.

Cost of Using Microservices

It does come with a cost:
COMPLEXITY

There are lots of moving pieces!
We have config files, ports, host names, etc. This can be a lot to keep track of.



About Monolithic Databases



Monolithic databases and web services are great for ACID, but everything is bound together. We lose the ability to scale and deliver independently.



Monolithic relational databases are good at consistency but at the expense of availability.



As we split the database into smaller tables fronted by separate web services, we lose ACID and must settle for eventual consistency. But we can gain availability and flexibility.

Microservices allow teams to divide work and responsibilities up along the same lines as the overall organization.

This allows us to follow Conway's Law:

Original quote:

“Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.”

Restated by Coplien and Harrison:

“If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product, or if the relationship between organizations do not reflect the relationships between product parts, then the project will be in trouble. Therefore: Make sure the organization is compatible with the product architecture.”

Microservices and Composability

Microservices allow for **composability**—we can stack them up.

This is a lot like composition in class design.





Recap

The main takeaways from this lesson are:

01

Most applications begin life as monoliths and many stay that way because they don't need to scale.

02

Twelve-factor, cloud-native, and microservice architecture and concepts allow teams to optimize for scalability, resilience, and agility.

03

None of these techniques are silver bullets. Each involves tradeoffs. Most of the tradeoffs have to do with increased application complexity in exchange for more scalability, resilience, or agility.

Questions?

