Introduction to Microservices

Java Accelerator 7

Lesson 5.1

WELCOME

Learning Outcomes

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

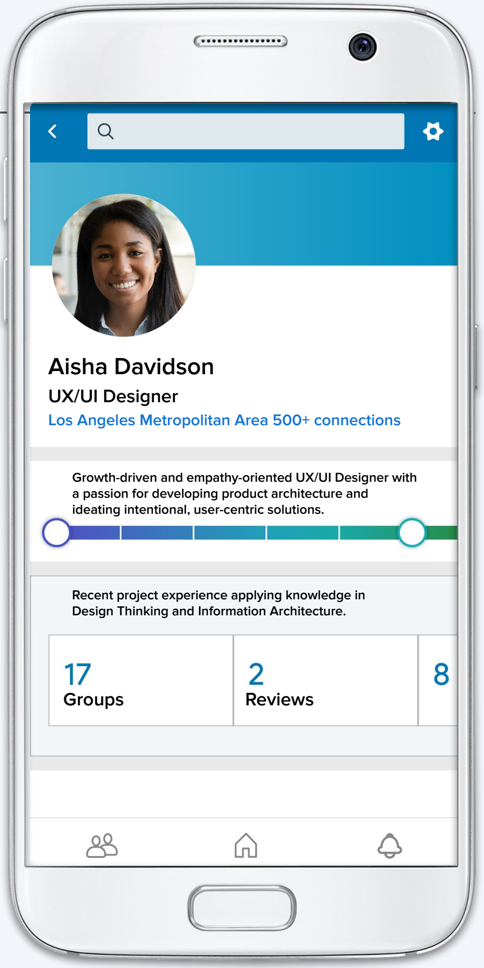
- Explain the purpose of the 12-factor app.
- Explain the cloud native approach to applications.
- Explain the purpose of microservices.
- Use a Spring configuration server to configure a REST web service.
- Use a service registry.

Spring Boot

Let's say we're working for a hot new internet social media company. We've built a really great web application with all the features the target audience wants, and we've rolled it out to great fanfare.

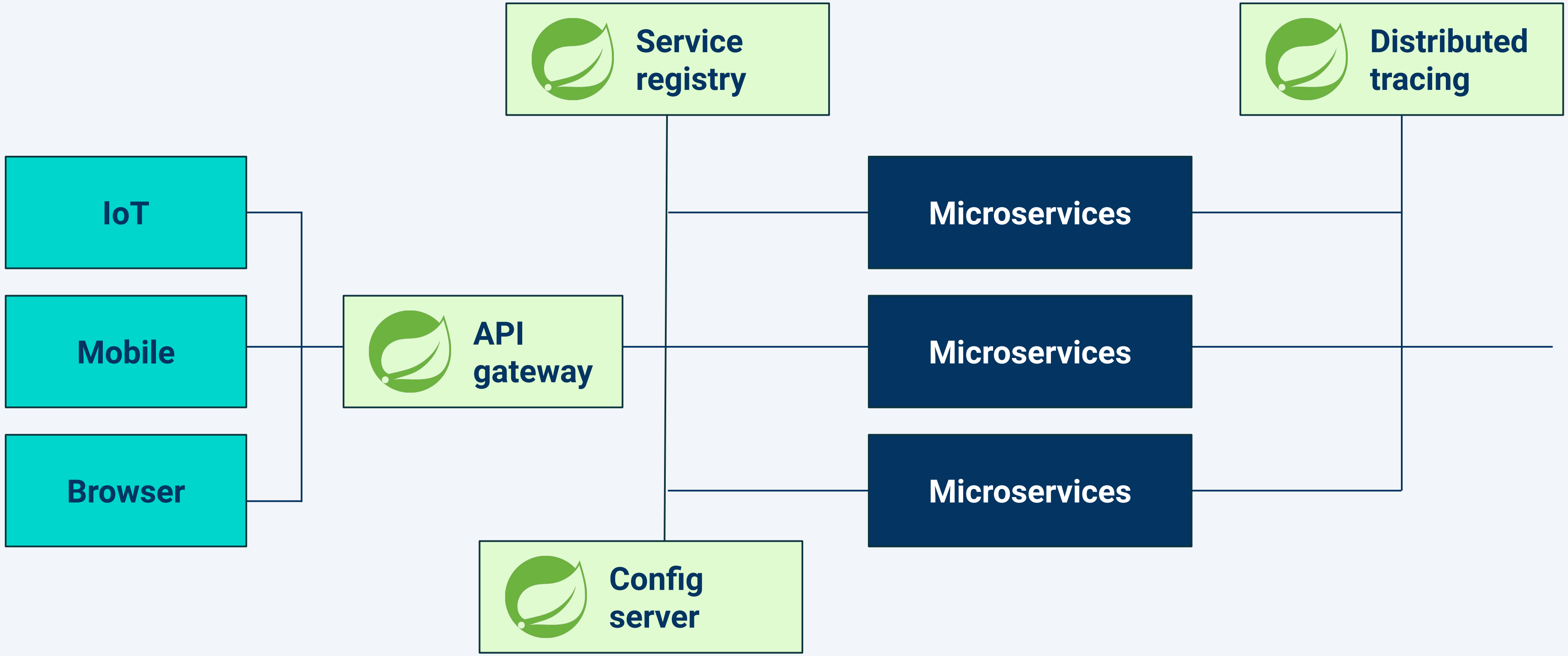
We're using Spring Boot, a relational database, and some super fancy JavaScript on the front end.

We have a great team that develops the code and another that handles building and deploying the application.


Aisha Davidson
UX/UI Designer
Los Angeles Metropolitan Area 500+ connections
Growth-driven and empathy-oriented UX/UI Designer with a passion for developing product architecture and ideating intentional, user-centric solutions.
Recent project experience applying knowledge in Design Thinking and Information Architecture.
17 Groups
2 Reviews
8

Spring Boot

Spring Boot makes it easy—one build, one deploy.



Spring Boot

Just drop the executable JAR on a server and run it. Voila!

```
mjohn@FVFC917PL417 dynamodb % java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -shareDB
Initializing DynamoDB Local with the following configuration:
Port:  8000
InMemory:   false
DbPath: null
SharedDb:   true
shouldDelayTransientStatuses:      false
CorsParams:      *
```

What could go wrong now that we're in production?

Spring Boot

Possible issues that could arise:

- The registration part of the system is getting overloaded.
- Searches for social posts are really crushing the database.
- We have a bug!

The Twelve-Factor App

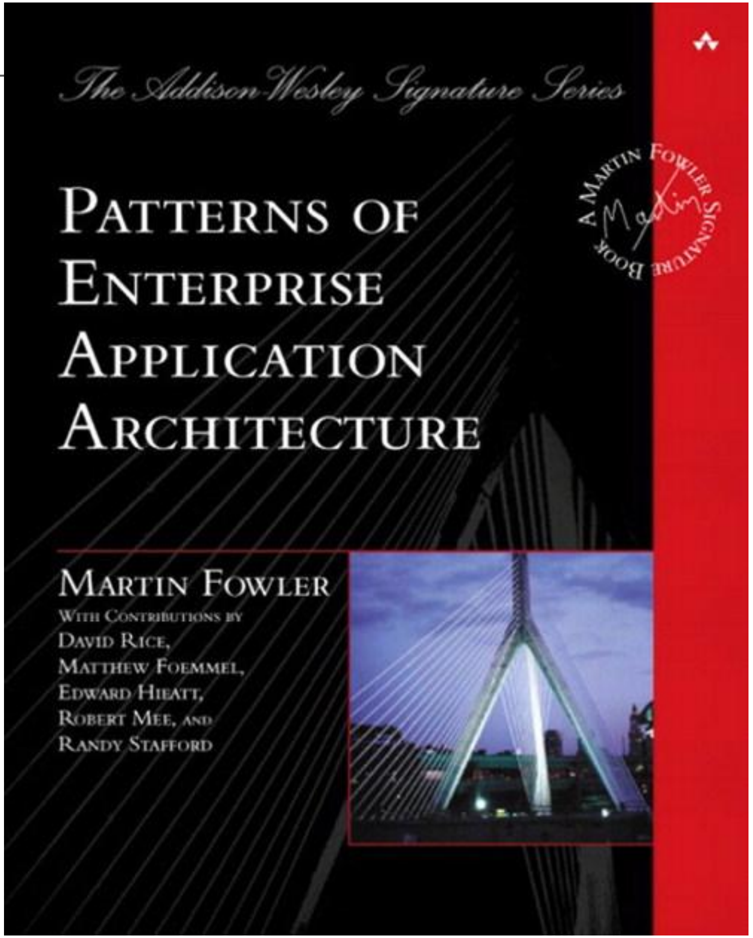
"Our motivation is to raise awareness of some systemic problems we've seen in modern application development, to provide a shared vocabulary for discussing those problems, and to offer a set of broad conceptual solutions to those problems with accompanying terminology."

—The 12-Factor App

The 12-Factor Application 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.

The 12-Factor Application

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


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
A Martin Fowler Signature Book
Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

Patterns of Enterprise Application Architecture
by Martin Fowler
(Addison-Wesley Professional, 2002)

The 12-Factor Application

1	Codebase	We have a good start on this because we're checking the code into GitHub.
2	Dependencies	Maven and Spring Boot help us with this factor.
3	Config	We'll address this factor using configuration servers.
4	Backing Services	We'll address this by using a service registry (Eureka) for everything.

The 12-Factor Application

5	Build, release, run	We'll address this using Jenkins and pipelines.
6	Processes	We'll follow these guidelines when using platform-as-a-service (no disk, etc.). We'll set up the applications as a set of independent microservices.
7	Port binding	We'll address this by using config servers, service registries, and microservices.
8	Concurrency	The microservices approach allows us to scale horizontally and have the services be independent. Breaking the monolith allows us to scale only parts of the application where needed.

The 12-Factor Application

9	Disposability	We'll strive for this. Spring Boot helps, and the other services we'll use (queues, service registry, etc.) are built for this.
10	Development/ production parity	This must be designed into the whole system. We'll explore pieces that help with that.
11	Logs	We'll explore techniques for this approach using 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.

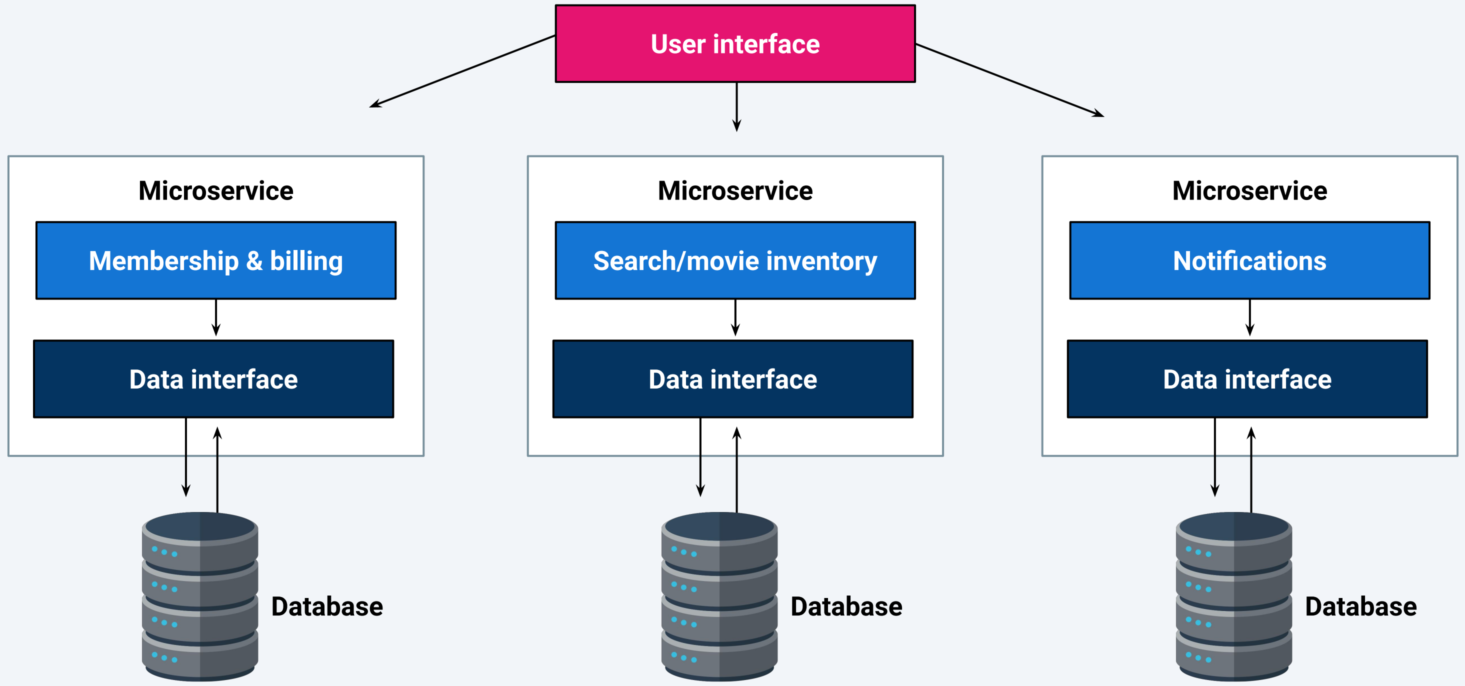
Cloud-Native Applications

Cloud-Native Applications

Cloud-native is related to 12-factor applications, and it solves some of the same problems.

Cloud-Native Applications

Cloud-native applications and teams take advantage of the patterns of 12-factor, DevOps, and microservices that improve scalability, reliability, and agility.



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

Cloud-Native Concepts

The main concepts of cloud-native applications:

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 being able to recover from those failures when they do occur.

Agility

The ability to deliver new features incrementally and consistently.

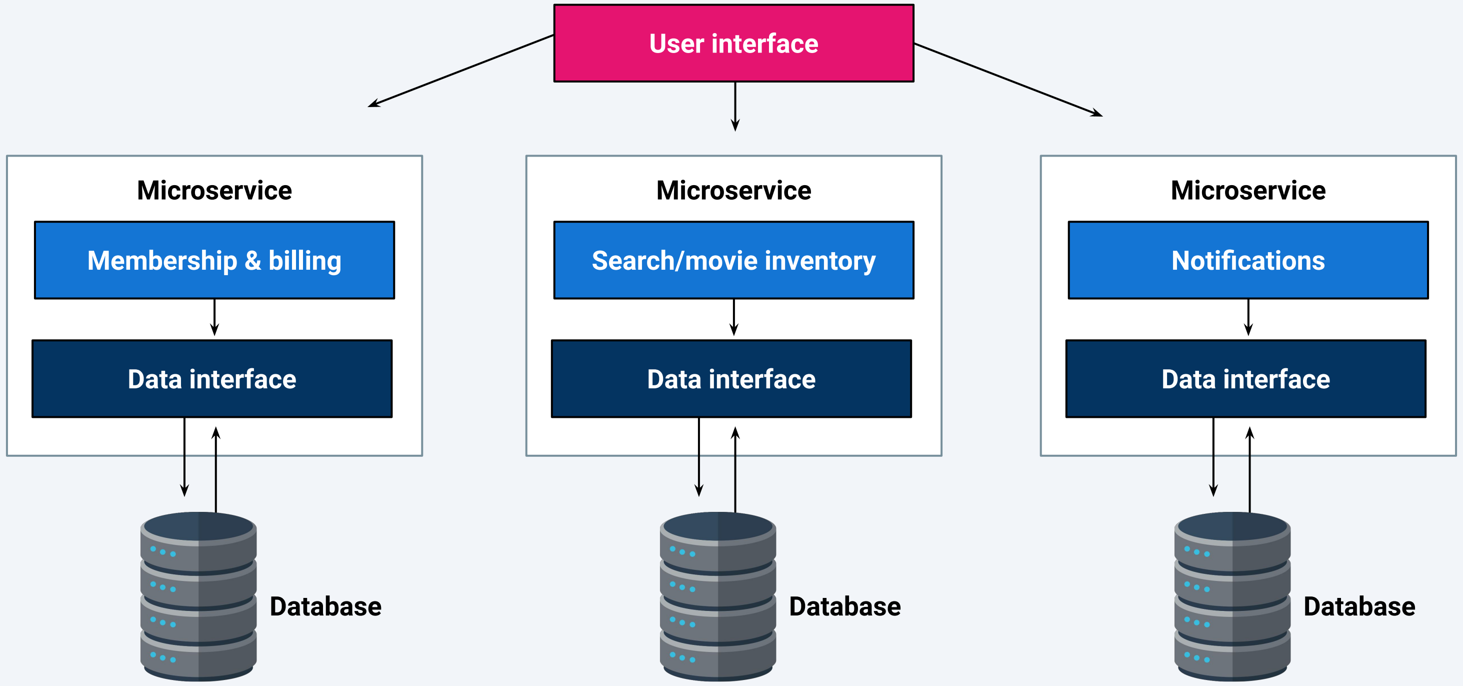
The Cloud-Native Approach

Ideas or principles	Define what we are optimizing for. The cloud-native approach values independence among components so that they can be quickly and easily scaled up or down and delivered individually.
Constraints	The ideas and principles help us form an approach to building the application. The approach to building the application must support the predefined ideas and principles. The approach is expressed as a group of constraints. These can resemble the twelve factors. For example, Factor #4, Backing Services, is a constraint.
Practices	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

Microservices

Small, independent services that work together to form a bigger application. They are usually web services—most often REST web services.



Microservices are tightly focused, have high cohesion, and follow the single-responsibility principle.

Microservices

Advantages of microservices architecture:

Independent scaling	We can scale up or down only the parts of the system that need to be scaled.
Independent release schedules/cycles	We can introduce new features or bug fixes on a per-microservice basis.
Right tech for right job	Because the contract between parts of the system is REST, we can use whatever technology 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

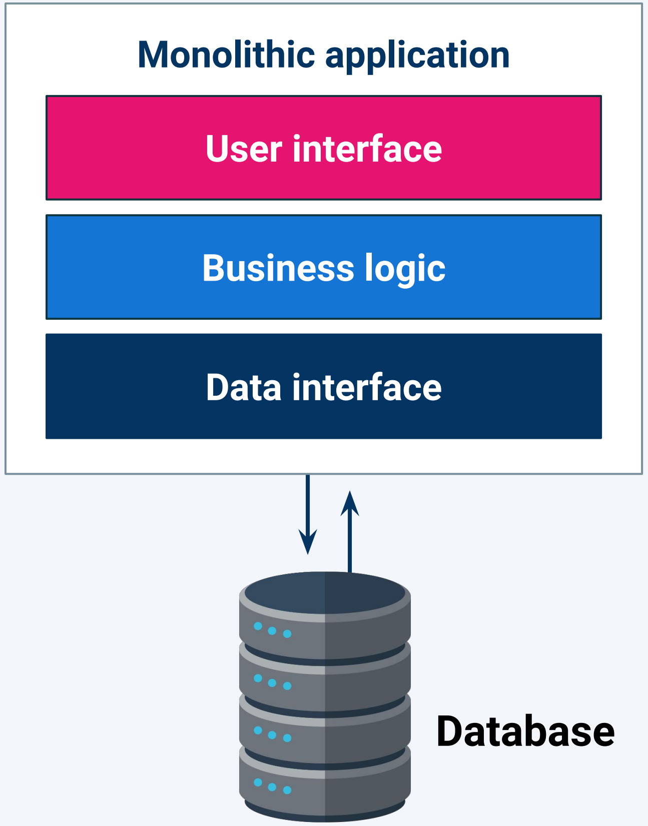
There are lots of moving pieces! We have config files, ports, host names, etc. This can be a lot to keep track of.

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.

Organizational Alignment

Microservices allow teams to divide work and responsibilities 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.

Config Servers

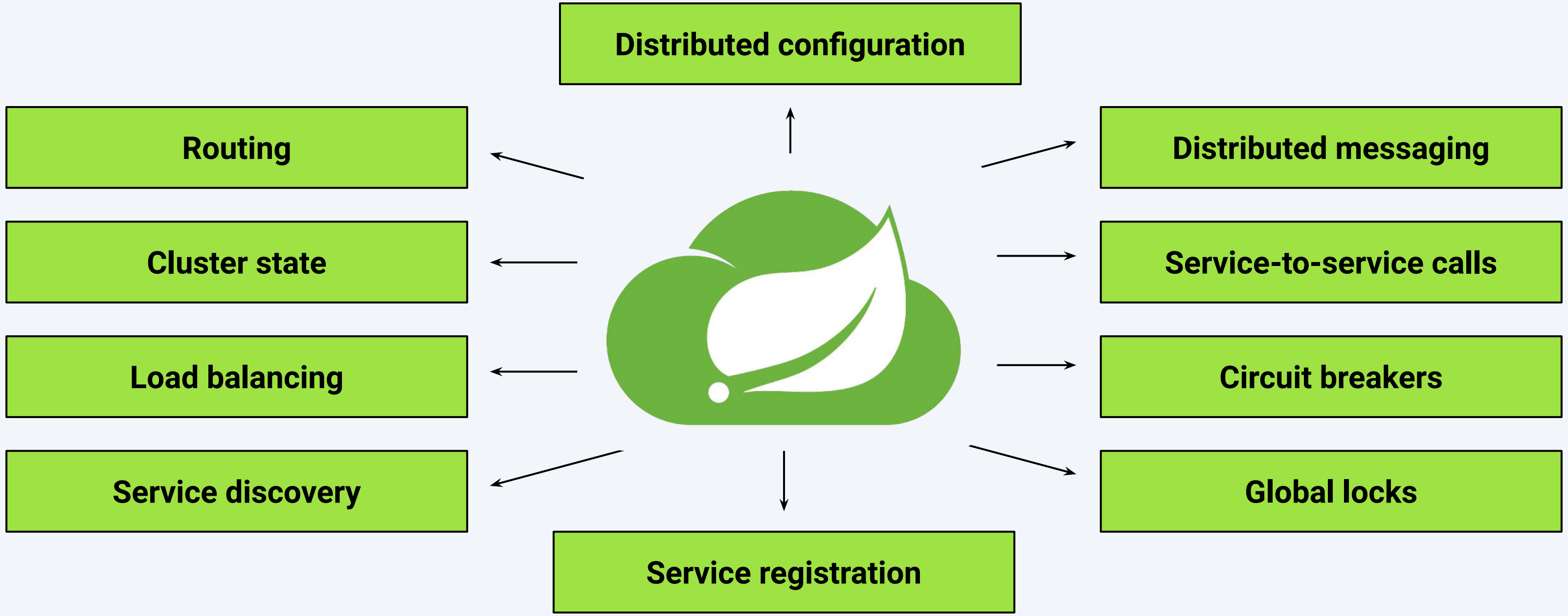
Config Servers

The advantages of externalizing configuration settings:

- Settings can be changed without having to rebuild code.
- Settings can be changed without having to restart the application.
- Settings are centralized.
- We can trace changes to settings.
- We can provide encryption and decryption of sensitive settings.

Configuration Server

Configuration server creation is similar to the creation of any Spring Boot project. We'll use the Initializr and select the appropriate starter pack. In this case, it's the Config Server starter pack.



Configuration Server Setup

Configuration Server

01

The **configuration server** is a web service. We start it up, and it runs on a particular port serving configuration settings to configured clients (we'll configure the client in the next step).

02

Configuration files for client applications are kept in a Git repository.

Setting Up the Config Server

01 Set up the repo.

02 Add the `@EnableConfigServer` annotation to the `main` class of the application.

03 Set the following values in the `application.properties` file:

`server.port`	The port that the config server will listen on.
`spring.cloud.config.server.git.uri`	The location of the Git repository that holds all the client configuration files.
`spring.cloud.config.server.git.username`	The username for the Git repository.
`spring.cloud.config.server.git.password`	The password for the Git repository.

For simplicity, our repositories will not have username/password.

04 After these values are set, we can start the config server as we would any web service.

Time to Code

Create Config Server

Suggested Time:

Using the Configuration Server

Using the Config Server

01

We enable a web service to use the config server by adding the Cloud Config Client starter dependencies.

02

To be able to refresh the configuration without having to restart the application, we must include the Actuator starter dependencies.

03

We add setup information to the `application.properties` file so that the application can use the config server.

Property Values in bootstrap.properties

We set the following property values in `bootstrap.properties`

`spring.application.name`

This is the name of the web service. It must match the name of the associated properties file in the Git repository used by the config server.

`spring.config.import`

This is the host and port of the config server.

Additional Configuration Settings

One advantage of using the config server is that we can change the config settings without having to restart the application. This requires some additional configuration:

The first step is to include the Actuator dependencies (we did that earlier).

We must include the `@RefreshScope` annotation on the controller class.

We must include the following entry in our application property file in the Git repo backing the config server:

```
Management.endpoints.web.exposure.
include=*
```

In order to force the application to RefreshScope, we must send an empty POST to the /actuator/refresh endpoint of the client application. The curl command is:

```
$ curl localhost:8080/actuator/
refresh -d {} -H"Content-Type:
application/json"
```

Time to Code

Hello Cloud

Suggested Time:

Time to Code

Quote Service

Suggested Time:

Service Registry

Service Registry

What is a service registry?

01

A **service registry** is a database of services, their instances, and their locations.

02

Applications can use the service registry to dynamically discover and call registered services.

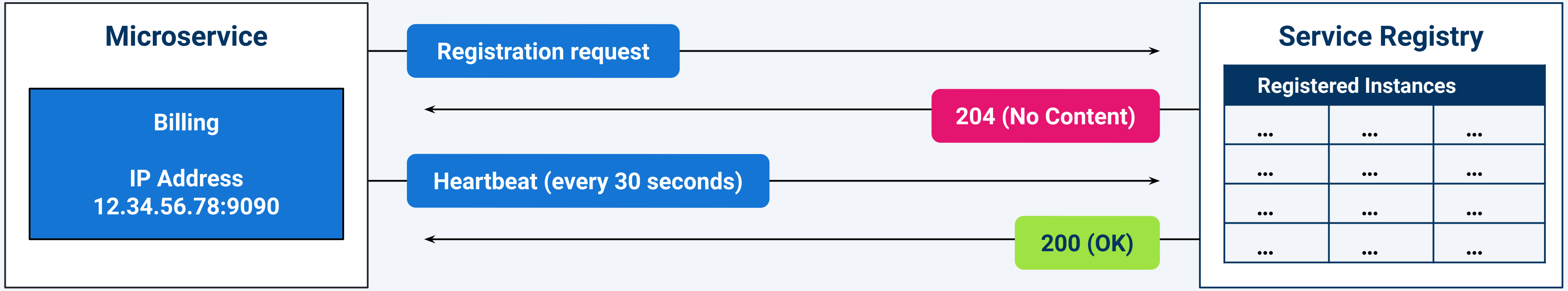
03

Clients use the service registry to know where to send their requests.

Service Registry

When the client registers itself as a service, it includes metadata such as its host and port. After it is registered, the client sends a renewal heartbeat at a regular interval (default: 30 seconds).

If the service registry does not receive a consistent heartbeat from an instance, that instance will be removed from the registry.



Service Registry

Advantages of using a service registry:

- Service instances are registered at startup and deregistered at shutdown.
- Allows the client to find available instances of a service.
- Can use the Health Check API to verify that the service is available to handle the request.

Service Registry

Disadvantages of using a service registry:

- Must be set up.
- Must be configured.
- Must be managed.
- Must be highly available and up-to-date, as a critical system component.

What happens if the service registry fails?

Similar to how DNS is used to find the IP address of a site, the service registry leads to the discovery of the service, aka **service discovery**.

Time to Code

Hello Cloud v2

Suggested Time:

Time to Code

Quote Service v2

Suggested Time:

Recap

Recap

What we learned in this lesson:

- Explain the purpose of the 12-factor app.
- Explain the cloud native approach to applications.
- Explain the purpose of microservices.
- Use a Spring configuration server to configure a REST web service.
- Use a service registry.