Building Microservices with Spring Boot

Chapter Content

1. Microservice disadvantages
2. The Roadmap

- Challenges in traditional waterfall methodologies:
    - Tightly coupled
    - Leaky
    - Monolithic
- Characteristics in microservice-based architecture:
    - Constrained
    - Loosely coupled
    - Abstracted
    - Independent

- Characteristics in Cloud-based development:
    - A large and diverse user base
    - Extremely high uptime requirements
    - Uneven volume requirements
- Successful microservice development of three critical roles:
    - Architect
    - Software developer
    - DevOps engineer

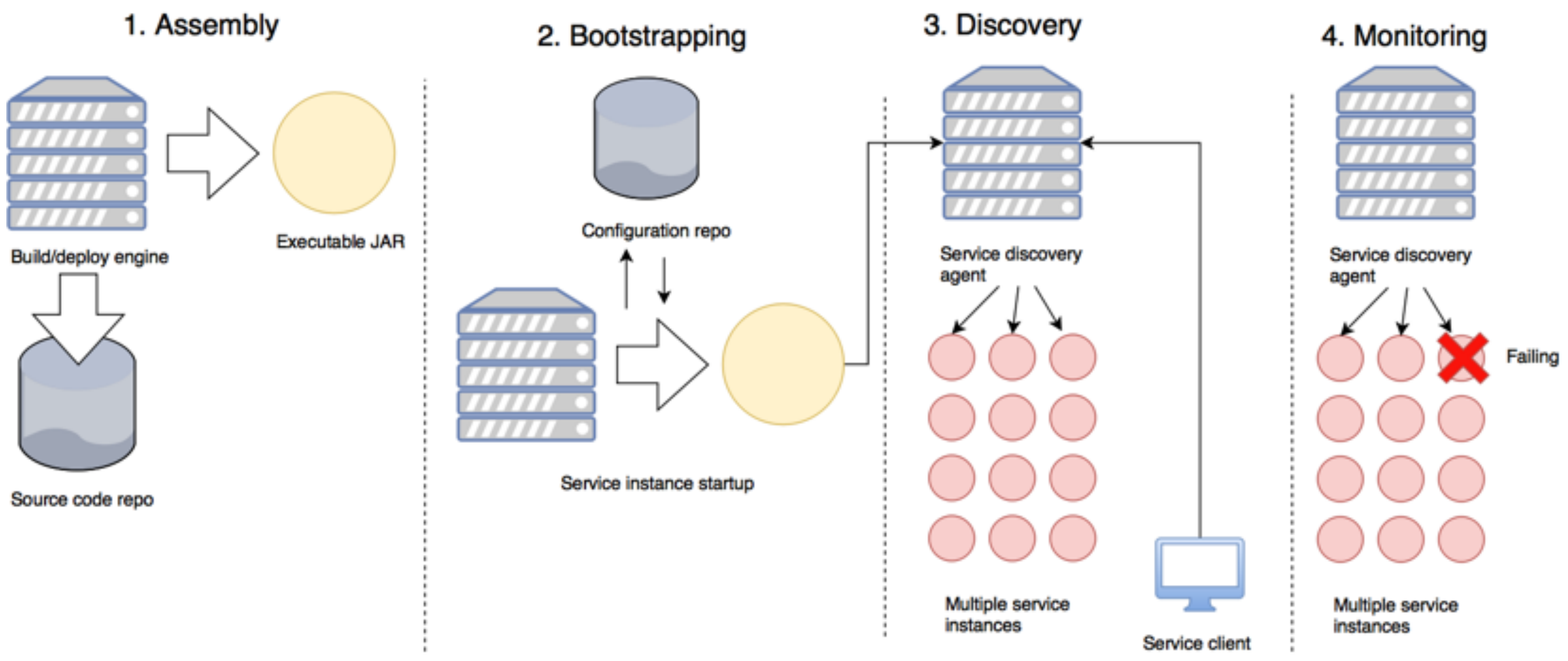
1. Microservice Disadvantages

- Complexity building distributed systems
  - Microservices are distributed and fine-grained, which introduce one more level of **complexity** compared to monolithic app
- Virtual server/container sprawl
  - In a large microservice-based application, you might end up with 50 to 100 servers or containers that have to be built and maintained in production. Even with cloud, the cost of **running**, the operational complexity of **managing** and **monitoring** can be a lot.
- Application type
  - Microservices are geared toward reusability and building large applications that need to be highly resilient and scalable. This means if you are building small, departmental level application with a small user base, it probably won't justify adopting microservice architecture.
- Data transactions and consistency
  - A microservice wraps around a small number of tables and works well as a mechanism for performing "operational" tasks such as CRUD. If the application needs to do complex data aggregation or transformation across multiple sources of data, the distributed nature of microservices will make this work difficult.

2. The Roadmap

- Microservice development principles:
  - A microservice should be **self-contained** and independently deployable with multiple instances of the service being started up and torn down with a single software artifact.
  - A microservice should be ***configurable***. When a service instance starts up, it should read the data it needs to configure itself from a central location or have its configuration information passed on as environment variables. No human intervention should be required to configure the service.
  - A microservice instance needs to be ***transparent*** to the client. The client should never know the exact location of a service. Instead, a microservice client should talk to a service discovery agent that will allow the application to locate an instance of a microservice without having to know its physical location
  - A microservice should ***communicate*** its health. This is a critical part of your cloud architecture. Microservice instances will fail and clients need to route around bad service instances.

- The four principles can be mapped to the following operational lifecycle steps:
    - **Service assembly** - package and deploy service to guarantee repeatability and consistency so that the same service code and runtime is deployed exactly the same way
    - **Service bootstrapping** - separate application and environment-specific configuration code from runtime code so that you can start and deploy a microservice instance quickly in any environment without human intervention to configure the microservice
    - **Service registration/discovery** - make a new microservice instance discoverable by other application clients after it's deployed
    - **Service monitoring** - when multiple instances of the same service are running, you need to monitor them and ensure any faults are routed around and ailing service instances are taken down



A successful microservice architecture requires strong application development and DevOps practices. One of the most succinct summaries of these practices can be found in Heroku's Twelve-Factor Application manifesto (https://12factor.net)

1. Service assembly - packaging and deploying microservices

- Multiple instances of one microservice should be able to be deployed quickly in response to a change of application environment

- To accomplish this, a microservice needs to be packaged and installable as **a single artifact** with all of its dependencies defined within it. This artifact can then be deployed to any server with a Java JDK installed on it. These dependencies will also include the runtime engine that will host the microservice.

- Spring boot uses maven to build an executable Java JAR file that has an embedded Tomcat engine in the JAR. The use of a single deployable artifact with the runtime engine embedded in the artifact eliminates much risk of configuration drift

2. Service bootstrapping - managing configuration of microservices:

- Service bootstrapping occurs when the microservice is first starting up and needs to load the application configuration information

- Traditionally, the configuration file is stored as **property file** with the application. However, with hundreds of microservice instances running on the cloud, it would be better off to store them **externally**

3. Service registration and discovery - how clients communicate with your microservices

- A microservice should be **location-transparent**, because a service on the cloud usually has shorter life. Cloud services can be started and torn down quickly with an entirely new IP address

- Due to the nature of large number of instances coming and going, manual managing this would be difficult so we should **automate** this process and thus service discovery

- When a microservice instance registers with a service discovery agent, it shows two things: the physical IP address or domain address of the service instance; a logical name that an application can use to look up the service

- Communicating a microservice’s health
  - Service instance failure happens all the time. The service discovery agent **monitors** the health of each service instance registered with it and removes any service instances from its routing tables to ensure that clients aren’t sent a service instance that has failed.
  - Service discovery agent will continue to monitor and ping the health check interface to ensure that the service is available
  - If a problem is discovered with an instance, it takes corrective action such as shutting down the ailing instance or bringing additional service instances up
  - In a microservice environment that uses REST, the simplest way to build a health check interface is to expose an HTTP end-point that can return a JSON payload and HTTP status code. This can be done with Spring Actuator.