An Architecture for Microservices using Spring

Rohit Kelapure

1 Motivation

We sincerely believe in the premise that, "On the back of software programming tools and Internet-based services Software is Eating the World" [andreesen]. Software is disrupting and eating much of the value chain of traditional major businesses and industries ranging from book sellers, retail, defense, agriculture etc., Industry analyst Gartner predicts that in a Bimodal IT world that has become ripe for digital disruption, by 2017, a significant disruptive digital business will be launched that was conceived by a computer algorithm. Given these stressors it is critical for enterprises to enable agile IT to iterate, prototype, develop and rapidly deliver software. Implementing a rapid feedback and rapid release loop is critical for survival. Microservices is a style of architecture that enables such an outcome for the digital business.

Microservices facilitate autonomy with responsibility. Microservices engender systems that provide flexibility and composability in terms of architecture, scalability and operational semantics. Microservices when done right increase the productivity of product teams by reducing developer cognitive load and separating concerns. Microservices, unlike monolith applications, decouple change cycles and enable frequent deploys of small well-tested cohesive components to production leading to quicker turnaround of features and a data driven analytic application.

A microservices based architecture for an application will allow large-scale architecture changes with minimal external impact providing insurance against unforeseen problems.

2 Characteristics of a microservice architecture [lewis-fowler, hughson]

Microservices is the first architectural style developed in a post continuous delivery world founded on the practices of continuous integration and continuous delivery. The key aspects of a microservices based architecture are listed below -

- **Componentization via Services**: Services are independently deployable, encapsulated, out-of-process components that separate concerns and can be composed into a system of systems with their interactions defined via service contracts (APIs).
- **Organized around Business Capabilities**: Services are vertical slices of technologies dedicated to particular business capability, ideally with coinciding service and team boundaries. Speed wins in the marketplace. Therefore leverage micros ervices for speed and availability.
- **Products not Projects**: The team should own the service over its entire lifecycle, both development and support. Establish a high trust low process culture where developers are given freedom and responsibility. Remove friction from product development.
- **Smart endpoints and dumb protocols**: Logic should reside within the services rather than the communications mechanisms that tie them together such as an HTTP request response style interaction or lightweight messaging.
- **Decentralized Governance**: Because of the partitioning inherent in this style, teams are not forced into one set of standards for their internal implementations.
- **Decentralized Data Management**: Because of the partitioning inherent in this style, data will be distributed across multiple services across availability zones and regions with a tendency towards eventual consistency.
- **Infrastructure and Test Automation**: Automated testing, automated deployment and automation of infrastructure. Auto scaled capacity and deployment updates.
- **Design for failure:** The distributed, partitioned nature of microservice architectures increases the need for system monitoring and resilience.
- **Evolutionary Design**: Smaller, modular services enable agile, controllable change. Isolate business logic in stateless microservices. Microservices scale the development process by reducing cognitive load.
- **Hexagonal Architecture (aka Ports and Adapters):** A way of structuring code for each service that separates the core domain model (What to do) from the ports and adaptors (How to do). Hexagonal architecture provides separation of concerns, ease of evolution and testability. Each of the hexagon's sides represents a different kind of port, for either input or output conversation with an external component. Sides consist of a port and an adaptor pair. By replacing the adaptors you can change how to the conversations happen. A hexagon handles 6 different types of conversations. Organize the methods on ports and adapters to reflect the conversations the core is trying to have. ports based on what conversation the core is trying to have with the outside world. Think of a Port as protocol endpoints like HTTP and the Adapter as a protocol handler like a Java Servlet or JAX-RS annotated class that receives method invocations from a Java EE container or framework (Spring, Jersey). [Skillcast & Implementing DDD].

2.1 What Microservices Architecture Is Not ?

Microservice architecture is not a layered architecture where presentation, business and data handling is realized as separate tiers to scale presentation and compute processing independently of the data tier. Microservices is also not an intelligently integrated architecture

where dumb services are integrated via a smart ESB or a centrally integrated architecture where anemic services are integrated with a central database. [thoughtworks-lessons-frontline]

3 Terms

Service is an extremely overloaded term in the cloud vocabulary. In the context of this document a microservice is synonymous with service i.e. an application deployed as a war or a jar file/s that exposes a REST API to provide a specific business capability. Cloud Foundry (CF) has its own notion of managed and user based services. These services when referred to, are explicitly called out as Cloud Foundry services.

A distinction is also made between Cloud Foundry and its commercial on premise distribution by Pivotal called Pivotal CF. By default the treatment of microservices covered in this document applies to the open source CF. Where applicable we explicitly call out Pivotal CF.

4 Applicability

4.1 Social

Before you embark on a microservices architecture for your enterprise you should establish a DevOps culture of production monitoring, rapid application deployment and provisioning implying close collaboration between developers and operations. This will often require a reorganization of teams. DevOps cannot be added like salt and pepper to a product. The CloudFoundry PaaS facilitates these baseline competencies enabling organizations to embrace microservices; however without the organizational shift to product centered teams that invert Conway's law, the long term benefits of microservices are negated. [References]

4.2 Technical

There is a certain threshold beyond which microservice style applications make sense. Cloud Foundry provides the baseline competencies to be successful with microservices i.e. 1. Rapid Provisioning, 2. Basic Monitoring and 3. Rapid Application Development. For your average enterprise application, a proven monolith architecture with clean component boundaries and the ability to scale in a cookie-cutter fashion using existing transactional paradigms like 2-phase commit makes sense. It is critical to balance speed of development/deployment while minimizing the risk of introducing change to critical systems. Microservices is not a panacea for architectural and scalability problems. In fact the same principles of SOLID design i.e. single responsibility, loose coupling, high cohesion, DRY, rigid interfaces, tolerant readers apply to both monolithic and microservices based systems. But a microservices architecture makes the process separation explicit, making it easier to separate bounded contexts.

5 Characteristics of Cloud Friendly applications

- Shared State is evil
    - Statelessness leads to scalability
    - Coordination of shared state across peers impacts performance

- No filesystem access
    - Local file system storage is short-lived
    - Instances of the same application do not share a local file system
- Configuration via environment or via an external configuration server (github/Zookeeper/etcd)
- Inject external dependency connection and credential information via services
- Console based logging
- Monitoring via services and frameworks
- Local development, cloud production
- All persistent data and state including HTTP sessions, caches etc., to be persisted and replicated by services like MySQL, REDIS cloud, etc., in external data stores.
- Execute application as one or more stateless processes enabling scale-out via a process model
- Explicitly declare and isolate dependencies
- One codebase tracked in revision control, multiple deploys
- Fast startup and graceful shutdown
- Make no distinction between local and third party services
- Treat logs as event streams
- Immutable code with instant rollback

6 Development Methodology - Local Development, Cloud Production

Develop microservices applications using Spring Boot and Spring Cloud projects. Spring Boot provides a nice abstraction layer for portability between local and cloud development. Spring Boot takes an opinionated - convention over configuration view of the Spring platform and third-party libraries so you can get started and deploy apps to production with minimum fuss. Spring Boot enables development in java, groovy or scala and packages apps as self contained jar and container deployable war files. Spring Boot tackles dependency hell via pre-packaging and the smart use of spring-boot-starter projects. A detailed treatment of spring cloud can be found in the following section.

1. Leverage the use of Spring programmatic configuration to configure the cloud and default profiles. Spring Boot automatically extracts the Cloud Foundry services connection and credential information from the VCAP_SERVICES environment variable and flattens the data into properties that can be accessed through Spring's Environment abstraction. Once the application is running in the cloud it is a best practice to switch to running with the "cloud" profile. Take a look at the SampleWebApplicationInitializer code from the rapwikiapp that dynamically switches the profile at runtime to "cloud" when running in CF.
2. Use the Spring Cloud local connector for local development and testing without mocking up `VCAP_SERVICES`. The connector provides the ability to configure Spring Cloud services locally for development or testing.