Class 2 – Spring Boot Microservice Enterprise Applications with Java

Contents

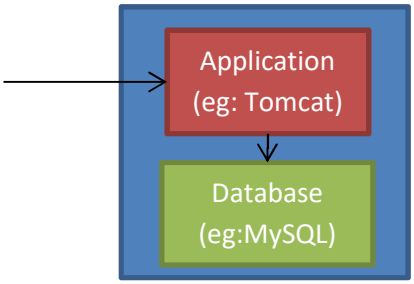
Review

- Class1
  - Movie Sites – Fandango and Movie Tickets review
- Q&A

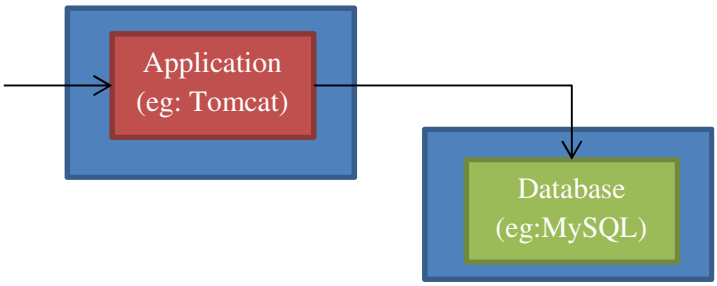
Break

Typical Deployments

1. Single Node – run all your processes in a single server (eg: application server, database etc). **Vertical scaling** by upgrading to more cores/more memory etc. Can get very expensive very quickly

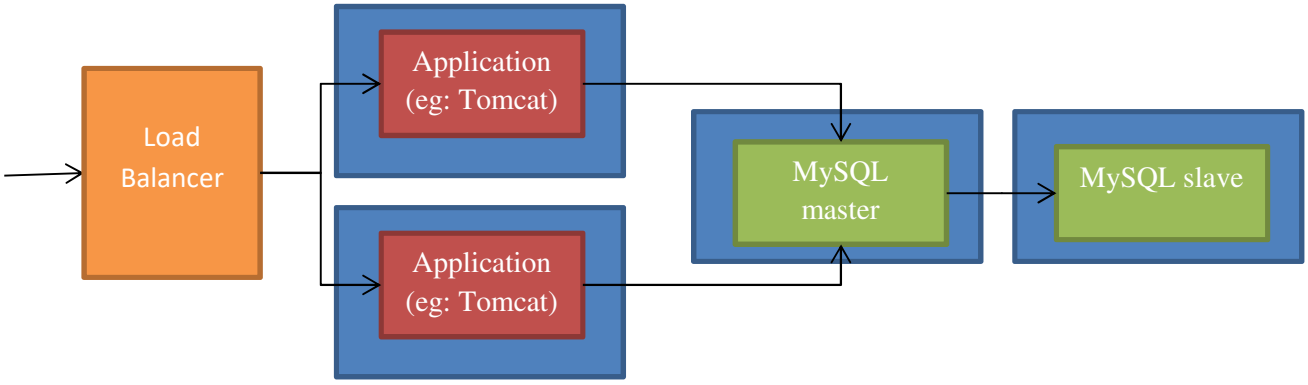


2. Multiple nodes – each node serves a specific purpose (eg: application server and database on separate servers). Each node can be independently vertically scaled to a limit. **Horizontal scaling** beyond that – **if your application can support it** AND additional infrastructure needed like a Load Balancer.

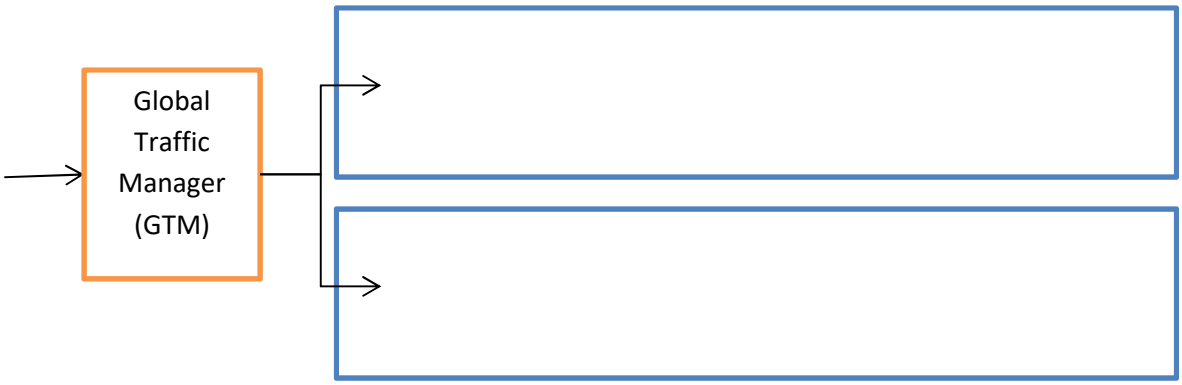


Typically multiple application servers and MySQL supports master-slave configuration. Some things to consider:

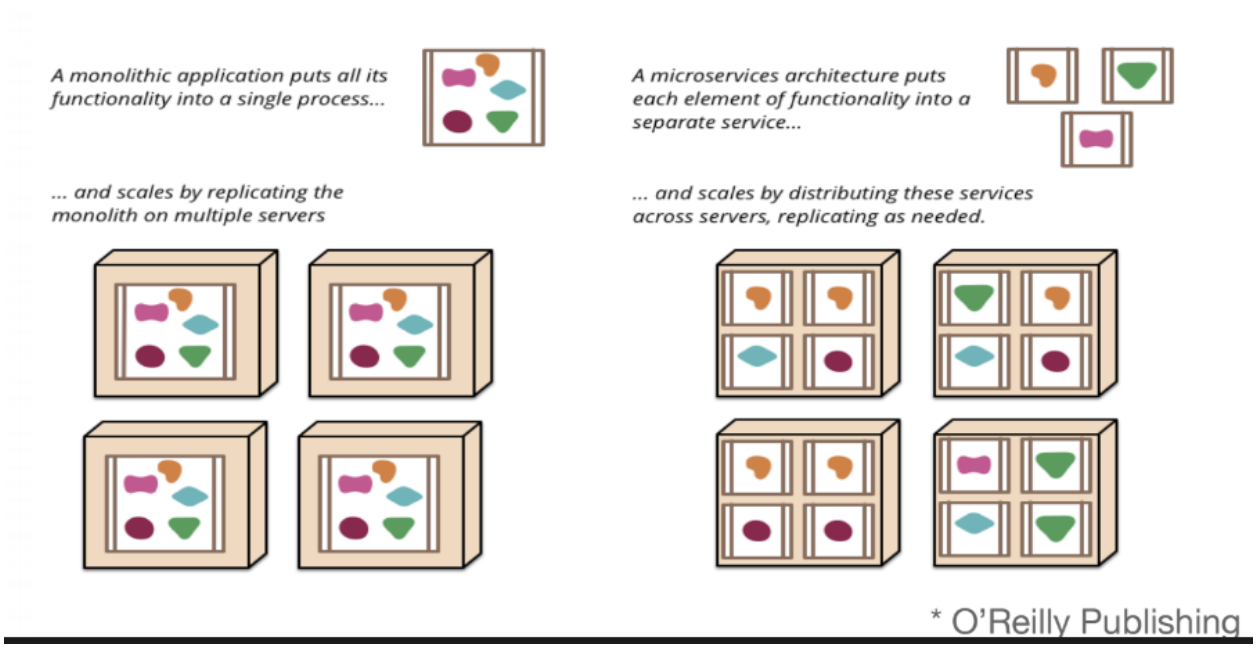
- What happens to stateful applications?
- Recovery from failure of different components?
- Load Balancer is single point of failure?



3. Multiple nodes in multiple regions or data centers – to better serve traffic from around the world, servers can be located in multiple regions with the nearest one serving the traffic. This adds additional complexity to stateful applications and data that is stored. This data must now be replicated which has its own complexities (CAP theorem later).



Microservices Architecture



Microservices vs SOA? http://stackoverflow.com/questions/25501098/difference-between-microservices-architecture-and-soa

- Microservice is more about a deployment architecture
- Not for everyone, there are many tradeoffs to consider
- Automation pipelines are a MUST – not easy to build and maintain

Advantages

- **Single responsibility** per service - Easier to develop, build, test (lower cost, faster to deliver)
- Services can be independently scaled, auto-scaled (elasticity)
- Multiple versions could be deployed simultaneously
- Services are independently deployed and tested
- Helps with agility – delivering business value faster
- Data separation – **each Microservice will usually have its own DB** and could be *polyglot* (eg: one Microservice uses SQL and second uses no SQL). This is data ownership

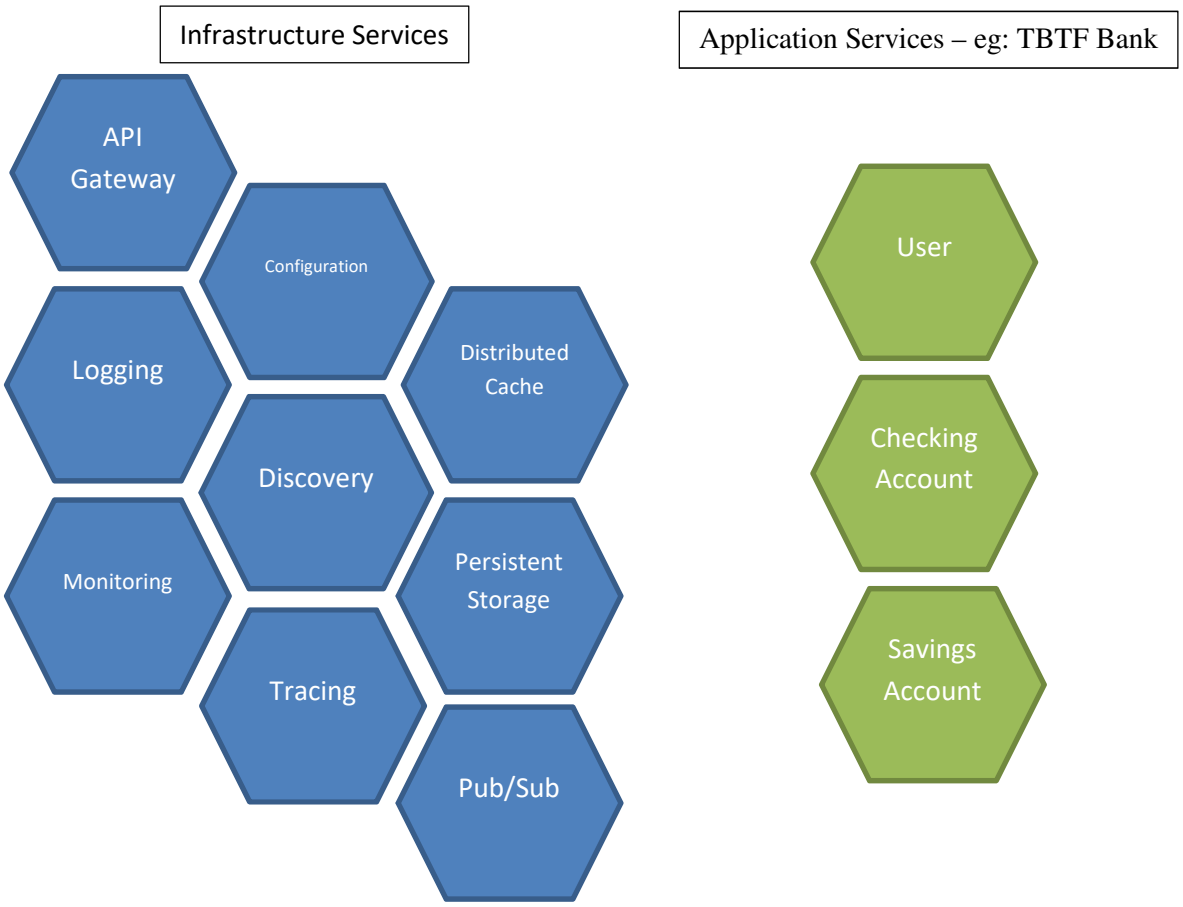
Disadvantages

- Increased deployment complexity
- Services are independently deployed and tested so needs organizational maturity
- Increased monitoring complexity

Microservices Deployment

Usually you will have the following components in your system.

Component Type	Examples
Legacy/External Dependencies	These are existing “monolith” systems that already exist or other existing dependencies that your new Microservices will need to interact with
Infrastructure Services	Services required for deployment and usually provided by cloud providers like AWS, GCP etc. eg: storage as a service
Application Services	Your applications
Development Support Services	To develop, test, deploy your code. Eg: git, container registry etc

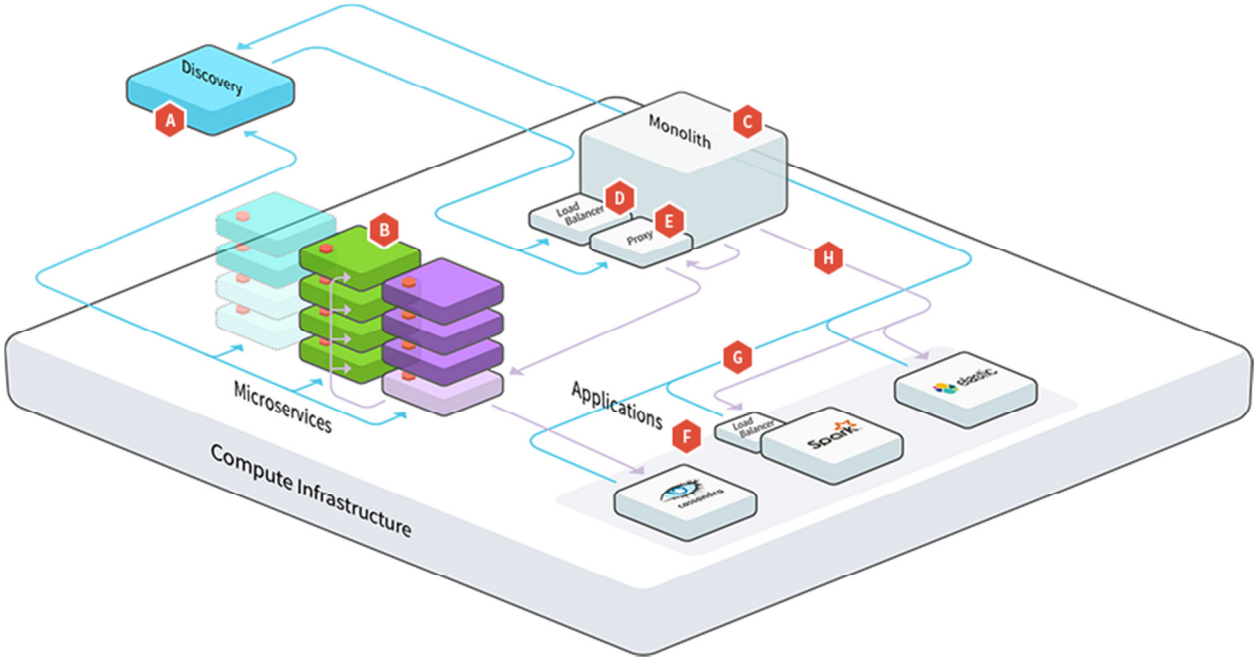


Component	Options
Configuration - manage properties for all services, refresh properties without restart, encrypted props	Spring Cloud Config Server Kubernetes Config Map/Secrets
Discovery - service registry to put together all components, enable load balancing, monitoring	Eureka Zoo Keeper

etc	Consul Kubernetes services
API Gateway - the front for everything, token management, token translation, security gateway	Kong Traefik Apigee AWS Gateway and other similar cloud provider services
Monitoring - metrics on all aspects of the system, alerting	GCP Stackdriver and other similar cloud provider services Hystrix Dashboard Grafana
Logging - log aggregation, analysis	GCP Stackdriver and other similar cloud provider services Elastic Search (usually along with Logstash and Kibana – "ELK")
Tracing - trace individual request across all the micro-services it interacts with	Zipkin Spring Cloud Sleuth
Pub/Sub - queueing infra for async communication between microservices	Kafka JMS GCP Pub/Sub and other similar cloud provider services
Distributed Cache	Memcache Redis
Persistent Storage	AWS and GCP have multiple options Any database – MySQL, Cassandra etc

A good example for deployment[1]:

[1] https://www.microservices.com/reference-architecture/



TBTF Bank Example

The application services are the ones we saw in Class 1. Think of these being independently deployed. User and Account data could live in different databases.

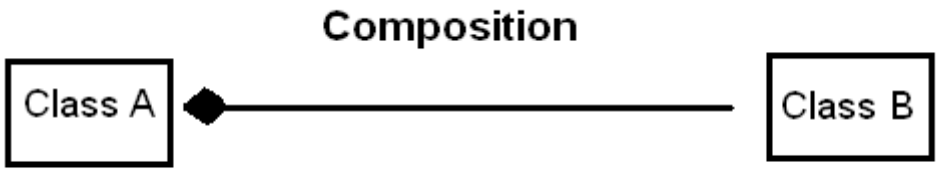
Agile and Microservices Mindset

Blog post: https://www.datawire.io/microservices-large-enterprises/

Microservices is about more than just a technology change.

Dependency Injection/Inversion of Control

Consider object composition: Class A has-a object of type B.



Typically:

```
class A {
    private B b;

    public A(){
       b = new B();
    }
}

class B {

}
```

If B had its own dependencies then those would be instantiated in the constructor of B and so on. In a different world suppose your classes are like this:

```
class A {
    private B b;
}

class B {

}
```

Somehow when an instance of A is created it is given the instance of B without having to construct it?

That leads to a Dependency Injection[2] container. Something like:

```
class Context {
    private A a;
    private B b;

    public void init(){
       b = new B();

       a = new A();
       a.setB(b);
    }
}
```

Now the instances "a" and "b" can be given to whoever needs them and they do not need to worry about how to create instances with all dependencies set. You could conceive of creating a mock B and setting that into A for a test. **The container takes care of the order of instantiation.**

Pros and Cons: http://en.wikipedia.org/wiki/Dependency_injection

- Easy to switch implementations

[2] http://martinfowler.com/articles/injection.html

- Easy to test
- Discoverability of instances in the Context
- Easy to apply aspect oriented programming (decorator pattern)

Break

Dependency and Build Management

Large software projects are composed of hundreds of thousands of files and millions of lines of code. What problems do you foresee?

The main goals we are trying to solve:

1. Dependency management: this involves decomposing large software projects into smaller, more manageable pieces that depend on each other. Each dependency may have multiple versions so accounting for **versions and transitive dependencies** is the major goal
2. Build management: this involves creating software for release – like jar or war files. The release process generally also includes testing and verification and managing which versions are available and how they are distributed
3. Continuous Integration/Continuous Deployment (CI/CD): Every check-in builds and verifies the software. This must be easy to manage

History

Make was the popular tool in the past and still is for C/C++.

For Java projects **Ant** became popular and it allows for tremendous flexibility. In Ant everything must be configured in a procedural way and every Ant script can be unique and do things its own way. Unfortunately there are no standards and for instance, different engineers may decide to use the words "clean", "clear", "delete", "init" etc to invoke the initialization process. So there is a steep learning curve if you join a new organization using Ant.

Maven

"Convention over Configuration"[3]

Maven is a project of the Apache Foundation and almost all Java related open source projects use it! The Project Object Model (POM)[4] is central to it (uses a XML file to define it). You can use Maven as a standalone tool or integration with your IDE (more on that later when we look at Eclipse).

In Maven there is one standard way – the *convention*. The project lifecycle[5] is defined by Maven "phases" and each phase has a "goals." In the above example you *must* invoke the "clean" goal and have it do whatever it needs to do. This makes defining projects uniform and consistent. For example public

[3] http://en.wikipedia.org/wiki/Convention_over_configuration
[4] http://maven.apache.org/pom.html
[5] http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

open-source projects like Spring. The folder structure – which folders store which files is also well defined and it must be followed. This allows anyone who knows the Maven conventions to immediately know where certain files will be in general for any project they encounter.

Customization using "plugins" is possible and there are many "standard" plugins that we will use. Maven has led to other projects like Ivy and Gradle.

Let us create a project in the Hands On section and take a look.

Hands On

Are these installed (see details in Class 1 Homework)?

- Java SE
- Maven

Let us create a Maven project and see what it looks like. Go to your terminal/command prompt.

Check version:

$mvn -version

- You should see a Maven and Java version

$cd <folder where you want to create a project>

Next create a blank project:
$mvn archetype:generate -DgroupId=<see below> -DartifactId=<see below> -DinteractiveMode=false -DarchetypeArtifactId=maven-archetype-quickstart

- You will see a bunch of things getting downloaded the first time! (We will talk about them)
- groupId: this is your root for the package hierarchy you use in your classes. For example com.sun, org.springframework, org.apache etc. This forms the root java package.
- artifactId: this is the name of the project. For example http-client, project1 etc
- What is an archetype[6] – think of it as a template (more in homework)

$cd <artifactId>

- You should see a "src" folder and pom.xml file
- We will look inside them later, first do this:

$mvn clean install

- Again bunch of downloads (we will talk about them)
- Should end in "Build Success"

[6] http://maven.apache.org/guides/introduction/introduction-to-archetypes.html

- What other new things got created in your folder?

Maven FAQ

1. The project folder structure
2. The pom.xml file
3. What are all the downloads?
4. ~/.m2
5. The target folder and artifact
6. Goals - what happened on mvn clean install?
7. Local and remote repositories
8. Versioning – SNAPSHOT and releases

Folders[7] - what we will be using

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/test/java	Test sources
src/test/resources	Test resources

Maven Terms

1. POM
    a. Project information – groupId, artifactId, version, packaging (how its distributed)
    b. Dependencies – direct dependencies only (Maven automatically gets the transitive dependencies)
    c. Build – plugins
    d. Repositories – we are currently using the default (Maven Central)
2. Repository ("repo")
3. Maven Coordinates – Group:Artifact:Version

STS/Eclipse

I will be using STS which is an Eclipse flavor and very popular and open-source IDE, IntelliJ is the popular commercial option. Best way to learn is to use it!

Homework for Class 2

Download, install and run STS or Eclipse for Java EE. Alternately you can use your favorite IDE like IntelliJ, NetBeans etc.

Try out these important features:

1. Create some Maven projects to play around

[7] http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

2. Load your Maven project in Eclipse (File > Import > Existing Maven Project)
3. Connect to your Git test repo and manage files in Git using Eclipse. (Eclipse terminology is "Team")
    a. **Binaries MUST NOT be committed to Git (everything in target, eclipse files etc)**
    b. Create a .gitignore file[8] (see example in hello-spring[9])
4. Maven commands from within Eclipse
5. Maven command on command line (the m2e Eclipse plugin is often flaky so I prefer to have a command prompt open and run Maven commands there instead)

[8] https://help.github.com/articles/ignoring-files
[9] https://github.com/rahulaga/hello-spring