SPRING -BOOT

A micro service Architecture approach

Powered by Java and Open Source Competency

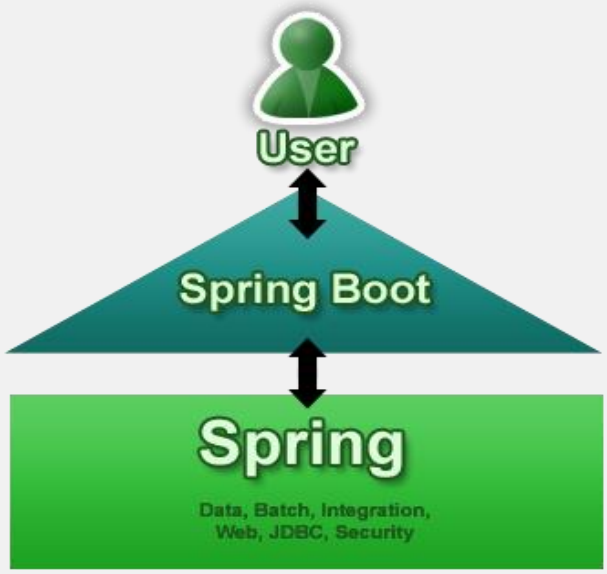
1/6/2017

Spring Boot –Micro Services

Table of Contents

1. Spring Boot introduction

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.



Features:

- Create stand-alone Spring applications.
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files).
- Provide opinionated 'starter' POMs to simplify your Maven configuration.
- Automatically configure Spring whenever possible.
- Provide production-ready features such as metrics, health checks and externalized configuration.
- Absolutely **no code generation** and **no requirement for XML** configuration
- It is well suited for developing Micro Service based applications

2. Monolithic Architecture

Characteristics

- One build and deployment unit.
- One code base.
- One technology stack (Linux, JVM, Tomcat, Libraries)

Benefits

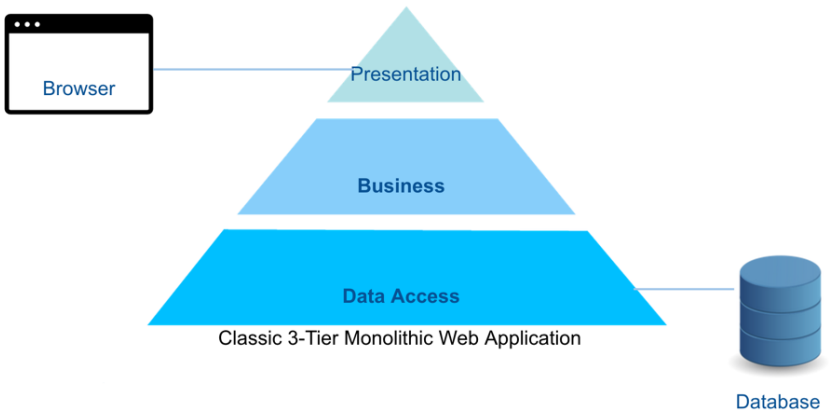
- Simple mental model for developers.
- One unit of access for coding, building, and deploying.
- Simple scaling model for operations.
- Just run multiple copies behind a load balancer.

Problems of Monolithic Software

- Huge and intimidating code base for developers.
- Development tools get overburdened.
- Refactoring take minutes.
- Builds take hours.
- Testing in continuous integration takes days.
- Scaling is limited.
- Running a copy of the whole system is resource-intense.
- It doesn't scale with the data volume out-of-the-box.
- Deployment frequency is limited.
- Re-deploying means halting the whole system.
- Re-deployments will fail and increase the perceived risk of deployment.

A monolithic application built as a single unit, but Enterprise Applications are often built in three main parts**.**

1. **Client-side user interface** consisting of HTML pages and JavaScript running in a browser on the user's machine.
2. **Database** consisting of tables inserted into a common, and usually relational, database management system
3. **Server-side application**. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a *monolith* - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application.



- A monolithic server is a natural way to approach building such a system. All your logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a deployment pipeline to ensure that changes are properly tested and deployed into production.

- You can horizontally scale the monolith by running many instances behind a load-balancer. Monolithic applications can be successful, but increasingly people are feeling frustrations with them - especially as more applications are being deployed to the cloud.

- Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed.

- Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module.

- Scaling requires scaling of the entire application rather than parts of it that require greater resource.

- These frustrations have led to the ***micro service architectural style.*** *Will see later in the next section.*

3. Micro Service Architecture

On the logical level, micro service architectures are defined by ***Functional system decomposition into manageable and independently deployable components***. Functional system decomposition means vertical slicing in contrast to horizontal slicing through layers .Independent deployability implies no shared state and inter-process communication via HTTP REST interfaces.

In short, the micro service architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Componentization via Services

For as long as we've been involved in the software industry, there's been a desire to build systems by plugging together components, much in the way we see things are made in the physical world. During the last couple of decades we've seen considerable progress with large compendiums of common libraries that are part of most language platforms.

When talking about components we run into the difficult definition of what makes a component. Our definition is that a component is a unit of software that is independently replaceable and upgradeable.Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

- Line of separation is along functional boundaries, not along tiers

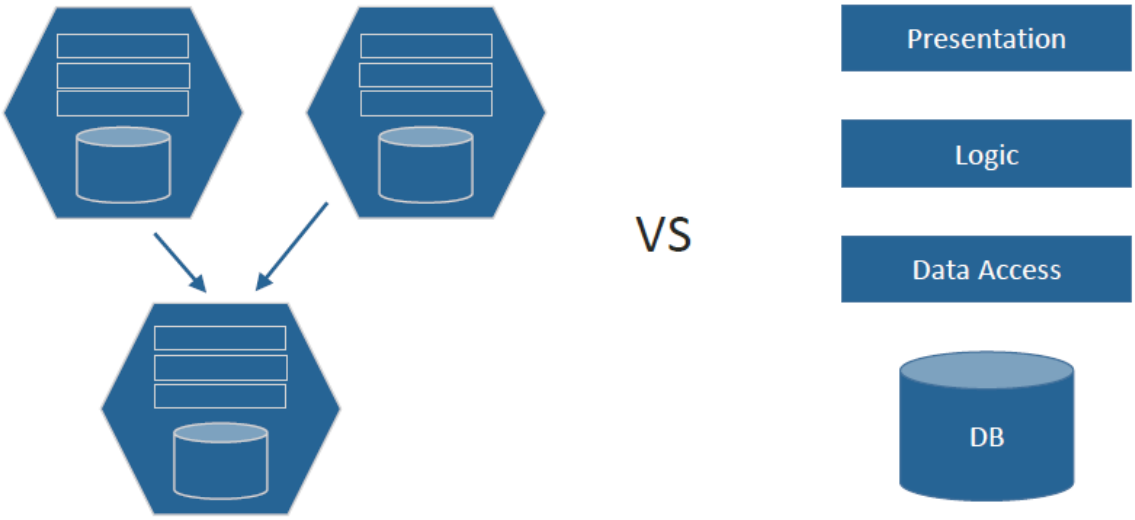
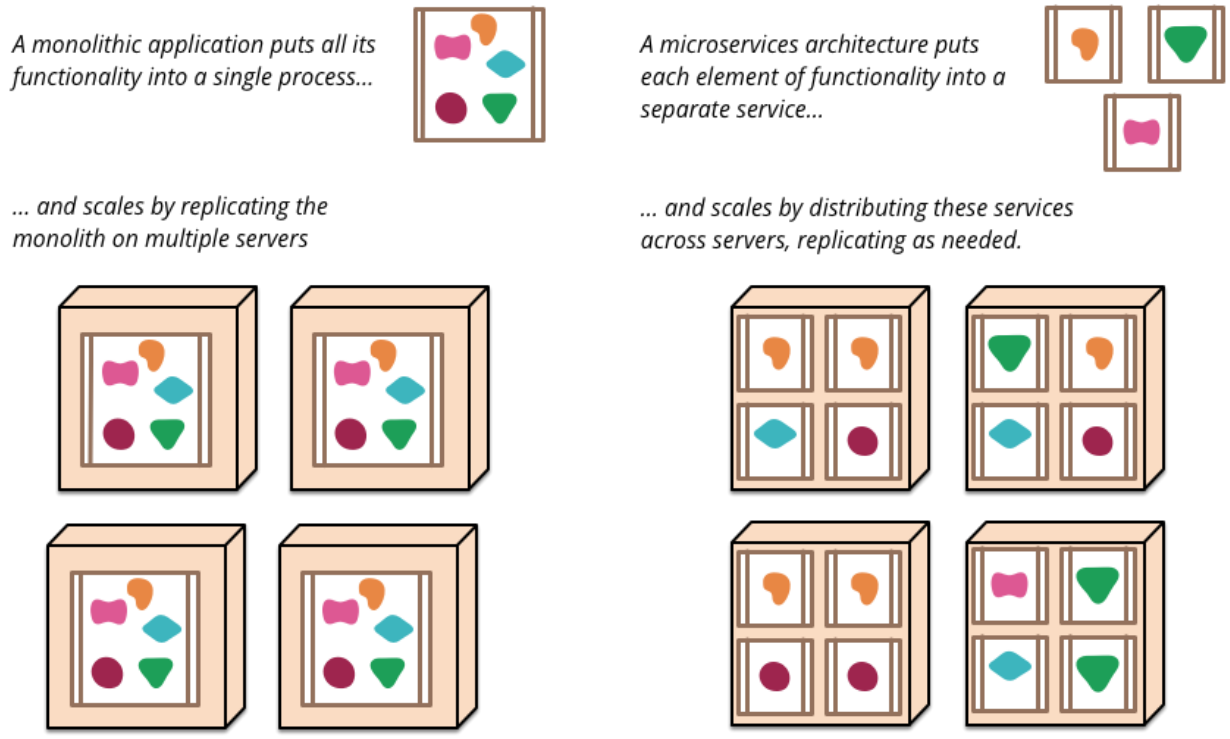


Figure: Differences between monolithic and Micro Service Architecture.



Characteristics

- Independent Deployability is the key and It enables separation and independent evolution of
  1. code base
  2. technology stacks
  3. Scaling and features too.

Independent code base

Each service has its own software repository

- Codebase is maintainable for developers – it fits into their brain.
- Tools work fast – building, testing, refactoring of code takes seconds.
- Service startup only takes seconds
- No accidental cross-dependencies between code bases.

Independent Technology Stack

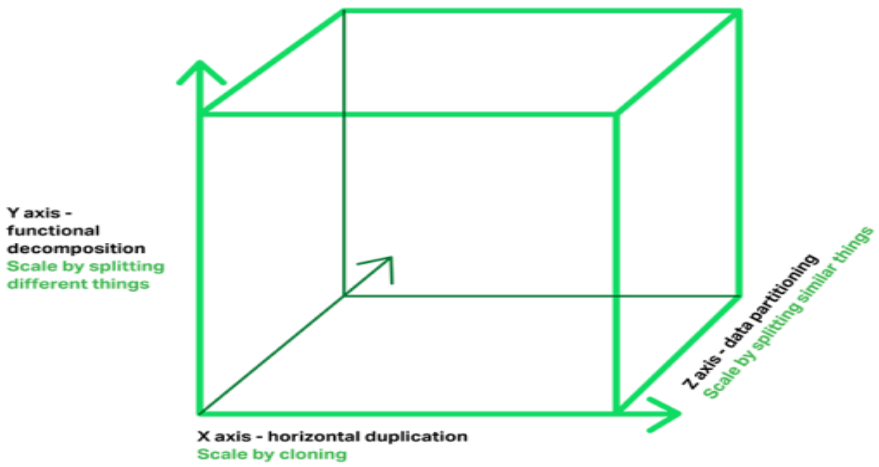
Each service is implemented on its own technology stacks

- The technology stack can be selected to fit the task best.
- Teams can also experiment with new technologies within a single micro service.
- No system-wide standardized technology stack also means
- No piggy-pack dependencies to unnecessary technologies or libraries
- Selected technology stacks are often very lightweight.

Independent Scaling

Each micro service can be scaled independently

- Identified bottlenecks can be addressed directly.
- Data sharding can be applied to micro services as needed.
- Parts of the system that do not represent bottlenecks can remain simple and un-scaled.



Applications typically use the three types of scaling together.
The Micro services Architecture pattern corresponds to the Y-axis scaling of the Scale Cube, which is a 3D model of scalability which decomposes the application into micro services. The other two scaling axes are X-axis scaling, which consists of running multiple identical copies (instances) of the application behind a load balancer, and Z-axis scaling (or data partitioning), where an attribute of the request (for example, the primary key of a row or identity of a customer) is used to route the request to a particular server.

Independent evolution of Features

Micro services can be extended without affecting other services.

- For example, you can deploy a new version of (a part of) the UI without re-deploying the whole system.

- You can also go so far as to replace the service by a complete rewrite

But you have to ensure that the service interface remains stable.

We have two technologies which will provide micro service based development capability.

1. Spring Boot.

2. Red Hat JBoss Fuse.

4. Spring Boot – Micro Services Approach

4.1 : Getting Started working with Spring Boot

The recommended way to get started using `spring-boot` in your project is with a dependency management system.

1.Create a maven Project and update the POM.xml file with below dependencies.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.4.3.RELEASE</version>
</parent>
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

2. Create a package "com.techm.cadt.springboot" and place this below main class.

```
package com.techm.cadt.springboot;
import org.springframework.boot.*;
@RestController
@SpringBootApplication
public class Example {
@RequestMapping("/")
   @ResponseBody
   String home() {
            return "Hello World!";
   }
   public static void main(String[] args) throws Exception {
      SpringApplication.run(SampleController.class, args);
   }
}
```

The final part of our application is the `main` method. This is just a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot's `SpringApplication` class by calling `run`. `SpringApplication` will bootstrap our

application, starting Spring which will in turn start the auto-configured Tomcat web server. We need to pass `Example.class` as an argument to the `run` method to tell `SpringApplication` which is the primary Spring component. The `args` array is also passed through to expose any command-line arguments.

Running the example

At this point our application should work. Since we have used the `spring-boot-starter-parent` POM we have a useful `run` goal that we can use to start the application. Type `mvn spring-boot:run` from the root project directory to start the application:

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.4.3.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.414)
```

If you open a web browser to localhost:8080 you should see the following output:

```
Hello World!
```

To exit the application hit `ctrl-c`.

NOTE: Here no XML configurations and no application server setup we have done but spring boot did some magic and it does everything what we need.

4.2: changing Default application port

Spring boot by default runs on 8080 port changing the below configuration in **application.properties** file we can change to any port of our interest.

Server.port = 8084.

Now application will be running from 8084 port.

4.3: Changing default Embedded Server

The Spring Boot starters (`spring-boot-starter-web` in particular) use Tomcat as an embedded container by default. You need to exclude those dependencies and include the Jetty one instead. Spring Boot provides Tomcat and Jetty dependencies bundled together as separate starters to help make this process as easy as possible. So update the Maven POM.XML file with below dependencies will change the server to jetty.

Example in Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

4.4 Changing the Java version

The `spring-boot-starter-parent` chooses fairly conservative Java compatibility. If you want to follow our recommendation and use a later Java version you can add a `java.version` property:

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

4.4: Changing Database Configuration

Update pom.xml file with below dependencies.

```
<dependency>
   <groupId>com.h2Database</groupId>
   <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>

<dependency>
   <groupId>org.springframework.boot </groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Update the below database properties in application.Properties file.

```
spring.datasource.url=jdbc:h2:file:./db/CustomerDb
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver=org.h2.Driver
```

4.6: Working with JPA to access Data

1. Define a simple entity

In this example, you store Customer objects, annotated as a JPA entity.

```
package com.techm.cadt.entity;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Customer {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;
   private String firstName;
   private String lastName;
  // setter getter methods
   protected Customer() {}
```

```
  public Customer(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  @Override
  public String toString() {
    return String.format(
        "Customer[id=%d, firstName='%s', lastName='%s']",
        id, firstName, lastName);
  }

}
```

Here you have a `Customer` class with three attributes, the `id`, the `firstName`, and the `lastName`. You also have two constructors. The default constructor only exists for the sake of JPA. You won't use it directly, so it is designated as `protected`. The other constructor is the one you'll use to create instances of `Customer` to be saved to the database.

The `Customer` class is annotated with `@Entity`, indicating that it is a JPA entity. For lack of a `@Table` annotation, it is assumed that this entity will be mapped to a table named `Customer`.

The `Customer`'s `id` property is annotated with `@Id` so that JPA will recognize it as the object's ID. The `id` property is also annotated with `@GeneratedValue` to indicate that the ID should be generated automatically.

The other two properties, `firstName` and `lastName` are left unannotated. It is assumed that they'll be mapped to columns that share the same name as the properties themselves.

The convenient `toString()` method will print out the customer's properties.

2. Creating Sample Queries

Spring Data JPA focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

```
package com.techm.cadt.dao;
import com.techm.cadt.entity.Customer;
import java.util.List;
```

```
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long> {
  List<Customer> findByLastName(String lastName);
  List<Customer> findAll();
}
```

In a typical Java application, you'd expect to write a class that implements `CustomerRepository`.
Here You don't have to write an implementation of the repository interface. Spring Data JPA creates an implementation on the fly when you run the application.

Simply accessing this Repository interface will gives you the Customer data(if already db has some rows).

4.7: Spring Boot Maven Plug-In

Spring Boot includes a Maven plugin that can package the project as an executable jar. Add the plugin to your <plugins> section if you want to use it:

```
<build>
        <plugins>
                <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
</build>
```

To run executable jar from command prompt **java –jar <jar filename>.jar**

This will start the internal tomcat server, and then starts the application.

4.8: Spring Boot annotations

@Configuration

we generally recommend that your primary source is a @Configuration class. You don't need to put all your @Configuration into a single class.

@Import or @ComponentScan

The @Import annotation can be used to import additional configuration classes or else you can use @ComponentScan to automatically pickup all Spring components, including @Configuration classes.

@EnableAutoConfiguration

Spring Boot auto-configuration attempts to automatically configure your spring application based on the jar dependencies that you have added. For example, If HSQLDB is on your class path, and you have not manually configured any database connection beans, then we will auto-configure an in-memory database. You need to opt-in to auto-configuration by adding the @EnableAutoConfiguration annotation to one of your @Configuration classes.

4.9:Authentication and Authorization

Working with multiple services always we should authenticate and authorize the user before providing resource access.

We have following options to do this.

1.Auth2 Authentication.
2. JWT (JSON Web tokens).

5. JWT Overview

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7419) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties, because as they can be signed, for example using public/private key pairs, you can be sure that the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure?

JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following.

`xxxxx.yyyyy.zzzzz` Let's break down the different parts.

Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA246 or RSA.

For example:

```
{
  "alg": "HS246",
  "typ": "JWT"
}
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: *reserved*, *public*, and *private*claims.

- **Reserved claims**: These is a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub**(subject), **aud** (audience), and others. Notice that the claim names are only three characters long as JWT is meant to be compact.
- **Public claims**: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
- **Private claims**: These are the custom claims created to share information between parties that agree on using them.

An example of payload could be:

```
{
 "sub": "1234467890",
 "name": "John Doe",
 "admin": true
}
```

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
For example if you want to use the HMAC SHA246 algorithm, the signature will be created in the following way:

```
HMACSHA246(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 secret)
```

The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

Putting all together

The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

How do JSON Web Tokens work?

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used), instead of the traditional approach of creating a session in the server and returning a cookie.

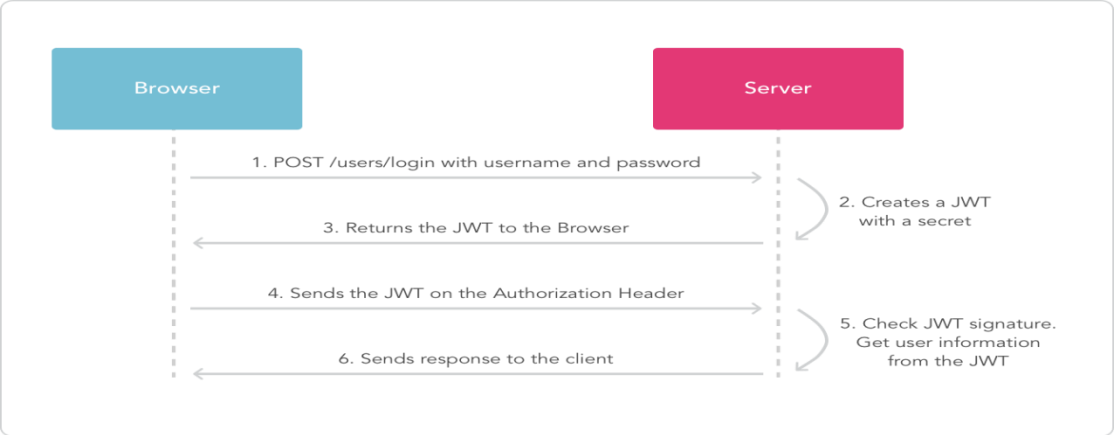
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

This is a stateless authentication mechanism as the user state is never saved in server memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. As JWTs are self-contained, all the necessary information is there, reducing the need to query the database multiple times.

This allows you to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn't matter which domains are serving your APIs, so Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

The following diagram shows this process:

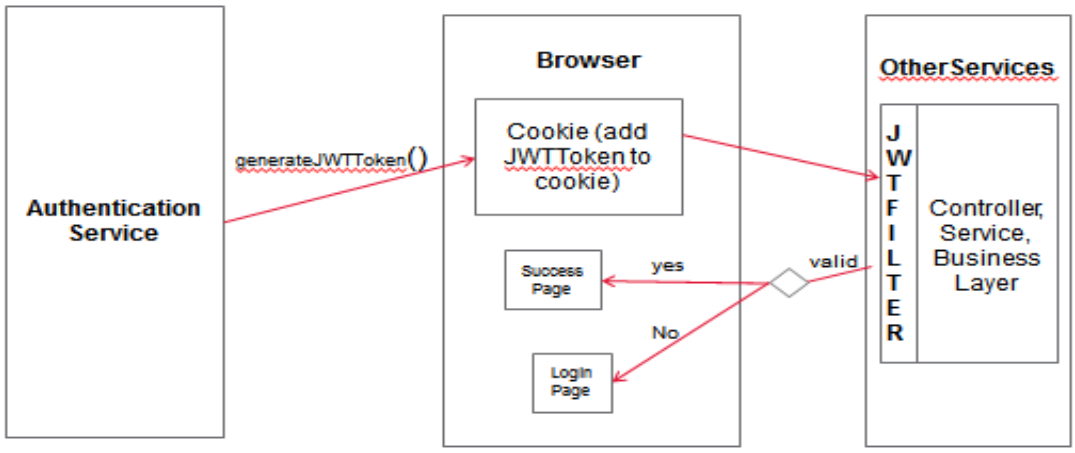


6. Spring Boot –JWT Integration

Update the pom.xnl file with below dependency.

```
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt</artifactId>
  <version>0.6.0  </version>
</dependency>
```

Approach 1:



Handling CORS Issues

To resolve the CORS issues while communicating among the multiple Micro services we can configure WebMvcConfigurer in the main class.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurerAdapter() {
      @Override
      public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("*").allowedOrigins("*")
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "HEAD")
                        .allowedHeaders("header1", "header2").allowCredentials(true);
      }
    };
  }
```

Figure : It Shows the JWT process flow in this approach.

Create JwtUtil.java file and is responsible for generate JWTtoken .

```
public static String generateToken(String signingKey, String subject) {
      long nowMillis = System.currentTimeMillis();
      Date now = new Date(nowMillis);
      JwtBuilder builder = Jwts.builder()
            .setSubject(subject)
            .setIssuedAt(now)
            .signWith(SignatureAlgorithm.HS246, signingKey);
      System.out.println("generated token inside JwtUtil ::::"+builder.compact());
        return builder.compact();
   }
```

Once token generation successful add this token to browser cookie in another file as mentioned below.

```
token = JwtUtil.generateToken(signingKey, username);
CookieUtil.create(httpServletResponse,jwtTokenCookieName,token, false, -1,"localhost");
```

Configure JWTFilter in the main class of second micro service . This will read the cookie which has be added by Authentication Service.

```
@Bean
   public FilterRegistrationBean jwtFilter() {
    final   FilterRegistrationBean registrationBean = new FilterRegistrationBean();
      registrationBean.setFilter(new JwtFilter());
      registrationBean.setInitParameters(Collections.singletonMap("next.page", orderpg));
     return registrationBean;
   }
```

Approach 2:

Steps involved in this approach

1.Create JWT Token for combining both username and password.

2.Include this generated token in each and every request headers.

3.In the target micro service once you reach the request headers data decode the jwt token.

4. Compare the Logged in user details against decoded JWT token values.

4. On successful validation of these credentials provide access to resource.

Step1: JWT token generation.

```
public static String generateToken
                  (String signingKey, String username, String password) {
     long nowMillis = System.currentTimeMillis();
           JwtBuilder builder = Jwts.builder()
                     .setIssuedAt(new Date(nowMillis))
                     .setSubject(String.valueOf(username))
                     .setIssuer(signingKey)
                     .claim("password", password)
                     .signWith(SignatureAlgorithm.HS246, signingKey);
                           return builder.compact();

   }
```

Step 2: Once JWT Token generated include this token in headers.

```
this.authenticate = function(username, password, callback) {
                    var user = {
                                "username" : username,
                                "password" : password };
var responsePromise = $http({
                                url : "http://localhost:8086/customerService/login",
                                method : "POST",
                                data : user,
                                headers : {
                                            'Content-Type' : 'application/json',
                                            'jwtToken' : jwttoken }
                    });
responsePromise.success(function(data, status, headers, config) {
                                callback(data); });
responsePromise.error(function(data, status, headers, config) {
alert("AJAX failed! because no webservice is attached yet");
                    });

          }
```

Step3: In the target Micro Service decode the jwt token as mentioned below.

```
public static String getSubject(HttpServletResponse httpServletResponse,
String token, String signingKey) {
Jws<Claims> claims = Jwts.parser().setSigningKey(signingKey).parseClaimsJws(token);
                    String username = claims.getBody().getSubject();
                    LOGGER.info("Subject :::"+username);
                    return username;
          }



public static String getPassword(HttpServletResponse httpServletResponse,String token, String signingKey) {
Jws<Claims> claims = Jwts.parser().setSigningKey(signingKey).parseClaimsJws(token);
                    Object password = claims.getBody().get("password");
                    LOGGER.info("password :::"+password);
                    return password.toString();
          }
```

Step 4 :Validate the decoded JWT tokenvalues on successful validate provide the access to resource.

```
 String jwtUsername = JwtUtil.getSubject(httpServletResponse, listObj, signingKey);
 String jwtPassword = JwtUtil.getPassword(httpServletResponse, listObj, signingKey);
          if (userCredentials != null)
                    userCredjson = new JSONObject(userCredentials.toString());
                    if (userCredjson != null)
                      username = userCredjson.optString("username");
                    if (username != null)
                      customerObj = customerService.getCustomerDetails(username);
```

```
                    if (customerObj != null) {
// Comapare the object from database and de copuled details from JWT Token details ..
if ((jwtPassword.equals(customerObj.getPassword())) &&
            jwtUsername.equals(customerObj.getUsername())) {
            validation = true;
            Step 4 : Provide the access to resource
            LOGGER.info("JWT Token Validataion Sucessful in customer Service.");
            } else {
            LOGGER.info("JWT Token Validataion failed in customer Service.");
            validation = false;
            Step 4 : Won't  Provide the access to resource
            }
  }
```

Summary :

Based on this document ,now you are able to build applications in micro service architecture with Spring boot and by using JWT for authentication.

References:

http://martinfowler.com/articles/microservices.html
https://projects.spring.io/spring-boot/
https://jwt.io/

Thank you