Topics to Cover

Monday, February 11, 2019 11:39 AM

1. Java 8
   - Futures-Executor Service
   - Async
   - Rest Template-**
   - JDBC Template
   - JWT Authentication-**
   - Collections
   - Comparator and Comparable

2. Spring Boot
   - AOP
   - Custom Annotations
3. Angular JS-2
4. Redis
5. Database
6. Spring Cloud Config
7. Microservices-Service Discovery,
8. Spring Data JPA
9. Spring Security-OAUTH,JWT Tokens

Javascript Reference-

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

Spring :

1. What is IOC?
2. What is the use of IOC?
3. What is DI?
4. How we are achieving DI in Spring?
5. Spring Bean scope?
6. Annotation based/Java Based/XML Based configuration.
7. Bean life cycle, bean properties.
8. Setter,Getter/Constructor based DI, when to use, where to use.
9. Little bit of Spring MVC.
10. Basics of Spring AOP.
11. Different bean scopes
12. Transaction Management
13. Differences between the annotations- @component, @controller, @service, @repository.

From <https://www.quora.com/Which-are-the-important-concepts-of-spring-and-hibernate-framework-to-crack-interviews>

Write a sample Controller
Sorting
Reversal
Fibbonachi
Recursion

Angular 2

-Observables
-Deploying a SB(server) + Angular 2(clent) app on server
-

From Aditya Call

- Junit testing
- How do you test your API?
- How do you secure your API-JWT tokens
- Git commands-rebase etc
- Field based dependency injection drawbacks

Tuesday, January 22, 2019 5:17 PM

medium
dzone
https://howtodoinjava.com

Spring Boot Parent

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath />
</parent>
```

From <https://www.baeldung.com/spring-boot-start>

Application configuration

```
@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Notice how we're using *@SpringBootApplication* as our primary application configuration class; behind the scenes, that's equivalent to *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan* together.

From <https://www.baeldung.com/spring-boot-start>

Properties Configuration

```
@Configuration
@PropertySource("classpath:configprops.properties")
@ConfigurationProperties(prefix = "mail")
public class ConfigProperties {

  private String hostName;
  private int port;
  private String from;

  // standard getters and setters
}
```

We use *@Configuration* so that Spring creates a Spring bean in the application context.

We also use @PropertySource to define the location of our properties file. Otherwise Spring uses the default location (*classpath:application.properties*).

@ConfigurationProperties works best with hierarchical properties that all have the same prefix. So we add a prefix of *mail*.

From <https://www.baeldung.com/configuration-properties-in-spring-boot>

Spring Boot Sample Starters

- *spring-boot-starter:* core starter, including auto-configuration support, logging, and YAML
- *spring-boot-starter-aop:* starter for aspect-oriented programming with Spring AOP and AspectJ
- *spring-boot-starter-data-jpa:* starter for using Spring Data JPA with Hibernate
- *spring-boot-starter-jdbc:* starter for using JDBC with the HikariCP connection pool
- *spring-boot-starter-security:* starter for using Spring Security
- *spring-boot-starter-test:* starter for testing Spring Boot applications
- *spring-boot-starter-web:* starter for building web, including RESTful, applications using Spring MVC

From <https://www.baeldung.com/spring-boot-interview-questions>

System Requirements

Spring Boot 2.1.0.BUILD-SNAPSHOT requires Java 8 or 9 and Spring Framework 5.1.2.RELEASE or above.

Explicit build support is provided for the following build tools:

Build Tool	Version
Maven	3.3+
Name	**Servlet Version**
Tomcat 9.0	4.0

spring-boot-starter-parent

Spring Boot provides a number of "Starters" that let you add jars to your classpath.

The `spring-boot-starter-parent` is a special starter that provides useful Maven defaults. It also provides a `dependency-management` section so that you can omit `version` tags for "blessed" dependencies.

Other "Starters" provide dependencies that you are likely to need when developing a specific type of application. Since we ar e developing a web application, we add a `spring-boot-starter-web` dependency

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>

Adding Dependency

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

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>

@RestController and @RequestMapping

```
@RestController
@EnableAutoConfiguration
public class Example {
@RequestMapping("/")
      String home() {
            return "Hello World!";
      }
public static void main(String[] args) throws Exception {
            SpringApplication.run(Example.class, args);
      }
}
```

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>

`@RestController`. This is known as a *stereotype* annotation. It provides hints for people reading the code and for Spring that the class plays a specific role. In this case, our class is a web `@Controller`, so Spring considers it when handling incoming web requests.

The `@RequestMapping` annotation provides "routing" information. It tells Spring that any HTTP request with the `/` path should be mapped to the `home` method. The`@RestController` annotation tells Spring to render the resulting string directly back to the caller.

The @RestController and @RequestMapping annotations are Spring MVC annotations. (They are not specific to Spring Boot.) See the MVC section in the Spring Reference Documentation for more details.

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>

`EnableAutoConfiguration`. This annotation tells Spring Boot to "guess" how you want to configure Spring, based on the jar dependencies that you have added. e.g if `spring-boot-starter-web` added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>

Although a single *@RequestMapping* path value is usually used for a single controller method, this is just good practice, not a hard and fast rule – there are some cases where mapping multiple requests to the same method may be necessary. For that case, **the *value* attribute of *@RequestMapping* does accept multiple mappings**, not just a single one:

```
@RequestMapping(
  value = { "/ex/advanced/bars", "/ex/advanced/foos" },
  method = GET)
@ResponseBody
public String getFoosOrBarsByPath() {
   return "Advanced - Get some Foos or Bars";
}
```

From <https://www.baeldung.com/spring-requestmapping>

Auto Configuration

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 classpath, and you have not manually configured any database connection beans, then Spring Boot auto -configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

You should only ever add one @SpringBootApplication or @EnableAutoConfiguration annotation. We generally recommend that you add one or the other to your primary @Configuration class only.

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

If you find that specific auto-configuration classes that you do not want are being applied, you can use the exclude attribute of `@EnableAutoConfiguration` to disable them, as shown in the following example:

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

you can also control the list of auto-configuration classes to exclude by using the `spring.autoconfigure.exclude` property.

You can define exclusions both at the annotation level and by using the property.

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

Spring Beans and dependency injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using `@ComponentScan` (to find your beans) and using `@Autowired` (to do constructor injection) works well.

If you structure your code as suggested above (locating your application class in a root package), you can add `@ComponentScan` without any arguments. All of your application components (`@Component`, `@Service`, `@Repository`, `@Controller` etc.) are automatically registered as Spring Beans

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

@SpringBootApplication

The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes, as shown in the following example:

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

- `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that is started from a `main()` method. In many situations, you can delegate to the static `SpringApplication.run` method, as shown in the following example:

```
public static void main(String[] args) {
      SpringApplication.run(MySpringConfiguration.class, args);
}
```

From <https://docs.spring.io/spring-boot/docs/2.1.0.BUILD-SNAPSHOT/reference/htmlsingle/#using-boot>

Dependecny Injection

field-based injection should be avoided whenever possible due to its many drawbacks however elegant it may seem. The recommended ap proach is then to use constructor-based and setter-based dependency injection. Constructor-based injection is recommended for required dependencies allowing them to be immutable and preventing them to be null. Setter-based injection is recommended for optional dependencies

From <https://blog.marcnuri.com/field-injection-is-not-recommended/>

Constructor

```
public class SimpleMovieLister {
// the SimpleMovieLister has a dependency on a MovieFinder
    private MovieFinder movieFinder;
// a constructor so that the Spring container can inject a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
// business logic that actually uses the injected MovieFinder is omitted...
}
```

From <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection>

```
public class SimpleMovieLister {
// the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;
// a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
// business logic that actually uses the injected MovieFinder is omitted...
}
```

From <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection>

Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for *mandatory dependencies* and setter methods or configuration methods for *optional dependencies*. Note that use of the @Required annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not `null`. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code smell*, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values withi n the class.

From <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection>

Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a`BeanCurrentlyInCreationException`.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Altern atively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with sette r injection.

Unlike the *typical* case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

From <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection>

What is Spring

Spring container types – BeanFactory and ApplicationContext.

container and its types From <https://www.google.com/search?q=spring+bean&rlz=1C1GCEA_enUS845US846&oq=spring+bean+&aqs=chrome..69i57j0l2j69i65l2j69i60.2663j0j7&sourceid=chrome&ie=UTF-8>	From <https://www.google.com/search?q=spring+bean&rlz=1C1GCEA_enUS845US846&oq=spring+bean+&aqs=chrome..69i57j0l2j69i65l2j69i60.2663j0j7&sourceid=chrome&ie=UTF-8>

Lambda expression

In Java programming language, a Lambda expression (or function) is just an *anonymous function*, i.e., a function with no name and without being bounded to an identifier. They are written exactly in the place where it's needed, typically *as a parameter to some other function.*

The basic *syntax of a lambda expression* is:

```
either
(parameters) -> expression
or
(parameters) -> { statements; }
or
() -> expression
```

From <https://howtodoinjava.com/java-8-tutorial/>

Most OOP languages evolve around objects and instances and treat only them their first class citizens. Another important entity i.e. functions take back seat. This is specially true in java, where functions can't exist outside an object. A function itself does not mean anything in java, until it is related to some object or instance.

But in functional programming, you can define functions, give them reference variables and pass them as method arguments and much more. JavaScript is a good example of this where you can pass callback methods e.g. to Ajax calls.

From <https://howtodoinjava.com/java8/lambda-expressions/>

Functional interfaces

Functional interfaces are also called *Single Abstract Method interfaces (SAM Interfaces).* As name suggest, they permit exactly one abstract method inside them. Java 8 introduces an annotation i.e. `@FunctionalInterface` which can be used for compiler level errors when the interface you have annotated violates the contracts of Functional Interface.

A typical functional interface example:

```
@FunctionalInterface
public interface MyFirstFunctionalInterface {
    public void firstWork();
}
```

From <https://howtodoinjava.com/java-8-tutorial/>

The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

From <https://www.journaldev.com/2763/java-8-functional-interfaces>

Default methods

A default method is a method with an implementation – which can be found in an interface.

We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface:

```
1 public interface Vehicle {
2   public void move();
3   default void hoot() {
4     System.out.println("peep!");
5   }
6 }
```

Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method. In Java 8, this problem has been solved by the use of default method.

For example, *Collection* interface does not have *forEach* method declaration. Thus, adding such method would simply break the whole collections API.

Java 8 introduces default method so that *Collection* interface can have a default implementation of *forEach* method without requiring the classes implementing this interface to implement the same.

From <https://www.baeldung.com/java-8-interview-questions>

Streams

Java 8 Streams API, which provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or anyother way that may be useful to an application.

From <https://howtodoinjava.com/java-8-tutorial/>

Different ways to create streams-

From <https://howtodoinjava.com/java8/java-streams-by-examples/>

1. Stream.of(val1, val2, val3....) `Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);`
2. Stream.of(arrayOfElements) `Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );`
3. List.stream() `list.stream();`
4. String chars or String tokens `IntStream stream = "12345_abcdefg".chars();`

Convert Stream to List – Stream.collect(Collectors.toList())

```
Stream<Integer> stream = list.stream();
List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0).collect(Collectors.toList());
```

Convert Stream to array – Stream.toArray(EntryType[]::new)

```
Stream<Integer> stream = list.stream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
```

Intermediate operations

Stream.filter()

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                    .forEach(System.out::println);
```

From <https://howtodoinjava.com/java8/java-streams-by-examples/>

Stream.map()

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type.

memberNames.stream().filter((s) -> s.startsWith("A"))

```
        .map(String::toUpperCase)
        .forEach(System.out::println);
```

From <https://howtodoinjava.com/java8/java-streams-by-examples/>

Stream.sorted()

The elements are sorted in natural order unless you pass a custom Comparator.

```
memberNames.stream().sorted()
        .map(String::toUpperCase)
        .forEach(System.out::println);
```

From <https://howtodoinjava.com/java8/java-streams-by-examples/>

Terminal operations

Stream.forEach()

```
memberNames.forEach(System.out::println);
```

Stream.collect()

```
memberNames.stream().sorted()
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

Stream.match()

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.

```
boolean matchedResult = memberNames.stream()
        .anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

matchedResult = memberNames.stream()
        .allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

matchedResult = memberNames.stream()
        .noneMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);
```

Stream.count()

Count is a terminal operation returning the number of elements in the stream as a long.

```
long totalMatched = memberNames.stream()
        .filter((s) -> s.startsWith("A"))
        .count();

System.out.println(totalMatched);
```

Stream.reduce()

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

```
Optional<String> reduced = memberNames.stream()
        .reduce((s1,s2) -> s1 + "#" + s2);
```

Stream short-circuit operations

Though, stream operations are performed on all elements inside a collection satisfying a predicate, It is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, you will do with if-else block. In internal iteration, there are certain methods you can use for this purpose. Let's see example of two such methods:

Stream.anyMatch()

This will return true once a condition passed as predicate satisfy. It will not process any more elements.

```
boolean matched = memberNames.stream()
        .anyMatch((s) -> s.startsWith("A"));

System.out.println(matched);
```

Output: true

5.2. Stream.findFirst()

It will return first element from stream and then will not process any more element.

```
String firstMatchedName = memberNames.stream()
        .filter((s) -> s.startsWith("L"))
        .findFirst().get();

System.out.println(firstMatchedName);
```

Output: Lokesh

Date Time API

`Date` class has even become obsolete. The new classes intended to replace Date class are `LocalDate`, `LocalTime` and `LocalDateTime`.

From <https://howtodoinjava.com/java-8-tutorial/>

1. The `LocalDate` class represents a date. There is no representation of a time or time-zone.
2. The `LocalTime` class represents a time. There is no representation of a date or time-zone.
3. The `LocalDateTime` class represents a date-time. There is no representation of a time-zone.

From <https://howtodoinjava.com/java-8-tutorial/>

`Duration` class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

```
Duration duration = Duration.ofMillis(5000);
duration = Duration.ofSeconds(60);
duration = Duration.ofMinutes(10);
```

`Duration` deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code. To interact with human, you need to get bigger durations which are presented with `Period` class.

```
Period period = Period.ofDays(6);
period = Period.ofMonths(6);
period = Period.between(LocalDate.now(), LocalDate.now().plusDays(60));
```

From <https://howtodoinjava.com/java-8-tutorial/>

Method references

METHOD REFERENCE	DESCRIPTION	METHOD REFERENCE EXAMPLE
Reference to static method	Used to refer static methods from a class	`Math::max` equivalent to `Math.max(x,y)`
Reference to instance method from instance	Refer to an instance method using a reference to the supplied object	`System.out::println` equivalent to `System.out.println(x)`
Reference to instance method from class type	Invoke the instance method on a reference to an object supplied by the context	`String::length` equivalent to `str.length()`
Reference to constructor	Reference to a constructor	`ArrayList::new` equivalent to `new ArrayList()`

From <https://howtodoinjava.com/java8/lambda-method-references-example/#method-references>

Stream vs Collection

Streams differ from collections in several ways:

- **No storage**. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure,an array, a generator function, or an I/O channel, through a pipeline of computational operations.
- **Functional in nature**. An operation on a stream produces a result, but does not modify its source. For example, filtering a `Stream` obtained from a collection produces a new `Stream`without the filtered elements, rather than removing elements from the source collection.
- **Laziness-seeking**. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunitiesfor optimization. For example, "find the first `String` with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (`Stream`-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
- **Possibly unbounded**. While collections have a finite size, streams need not. Short-circuiting operations such as `limit(n)` or `findFirst()` can allow computations on infinite streams to complete in finite time.
- **Consumable**. The elements of a stream are only visited once during the life of a stream. Like an `Iterator`, a new stream must be generated to revisit the same elements of the source.

From <https://stackoverflow.com/questions/39432699/what-is-the-difference-between-streams-and-collections-in-java-8>

Predicate

In Java 8, Predicate is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference. So, where you think, we can use these true/false returning functions in day to day programming? I will say you can use predicates anywhere where you need to evaluate a condition on group/collection of similar objects such that evaluation can result either in true or false.

From <https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/>

As I said, `Predicate` is functional interface. It mean we can pass lambda expressions wherever predicate is expected. For example one such method is `filter()` method from `Stream` interface.

```
Stream<T> filter(Predicate<? super T> predicate);
```

From <https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/>

Date and Time

LocalDate

The LocalDate class represents a date. There is no representation of a time or time-zone.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

The LocalTime class represents a time. There is no representation of a date or time-zone.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

LocalDateTime

The LocalDateTime class represents a date-time. There is no representation of a time-zone.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

`OffsetDate`, `OffsetTime` and `OffsetDateTime`.`ZoneId`.

If you want to use the date functionality with zone information, then Lambda provide you extra 3 classes similar to above onei.e. `OffsetDate`, `OffsetTime` and `OffsetDateTime`. Timezone offset can be represented in "+05:30" or "Europe/Paris" formats. This is done via using another class i.e. `ZoneId`.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

Instant

For representing the specific timestamp ant any moment, the class needs to be used is Instant. The `Instant` class represents an instant in time to an accuracy of nanoseconds. Operations on an `Instant` include comparison to another `Instant` and adding or subtracting a duration.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

Duration

Duration class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

Period

To interact with human, you need to get bigger durations which are presented with Period class.Days,Months Year difference

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

ChronoUnit

java.time.temporal.ChronoUnit example to know the difference in days/months/years

Java program to get difference between two dates in months using ChronoUnit class.

```
public void difference_between_two_dates_java8()
{
     LocalDate dateOfBirth = LocalDate.of(1980, Month.JULY, 4);
     LocalDate currentDate = LocalDate.now();
     long diffInDays = ChronoUnit.DAYS.between(dateOfBirth, currentDate);
     long diffInMonths = ChronoUnit.MONTHS.between(dateOfBirth, currentDate);
     long diffInYears = ChronoUnit.YEARS.between(dateOfBirth, currentDate);
}
```

From <https://howtodoinjava.com/java8/calculate-difference-between-two-dates-in-java/>

Timezone Changes

Timezone related handling is done by 3 major classes. These are ZoneOffset, TimeZone, ZoneRules.

- The `ZoneOffset` class represents a fixed offset from UTC in seconds. This is normally represented as a string of the format "±hh:mm".
- The `TimeZone` class represents the identifier for a region where specified time zone rules are defined.
- The `ZoneRules` are the actual set of rules that define when the zone-offset changes.

From <https://howtodoinjava.com/java8/date-and-time-api-changes-in-java-8-lambda/>

String.join()

```
String join(CharSequence delimiter, CharSequence... elements)
```

This method can be used to join multiple strings which are not yet in form of collection or array.

```
String joinedString = String.join(", ", "How", "To", "Do", "In", "Java");
```

```
String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)
```

This method is used to join array of strings or list of strings.

Java program to join list of strings

```
List<String> strList = Arrays.asList("How", "To", "Do", "In", "Java");

String joinedString = String.join(", ", strList);

StringJoiner(CharSequence delimiter)
StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)
```

2.2. StringJoiner Example

Run the example with similar input as above example to join multiple strings. We want to format the output as `[How, To, Do, In, Java]`, then we can use below code:

```
StringJoiner joiner = new StringJoiner(", ", "[", "]");

joiner.add("How")
        .add("To")
        .add("Do")
        .add("In")
        .add("Java");

Output:

[How, To, Do, In, Java]
```

From <https://howtodoinjava.com/java8/java-8-join-string-array-example/>

While using Java 8 lambda, we can use Collectors.joining() to convert list to String.

```
List<String> numbers = Arrays.asList("How", "To", "Do", "In", "Java");

String joinedString =   numbers
                        .stream()
                        .collect(Collectors.joining(", ","[","]"));
```

From <https://howtodoinjava.com/java8/java-8-join-string-array-example/>

Comparator with Lambda

```
//Compare by Id
Comparator<Employee> compareById_1 = Comparator.comparing(e -> e.getId());

Comparator<Employee> compareById_2 = (Employee o1, Employee o2) -> o1.getId().compareTo( o2.getId() );

//how to use comparator
Collections.sort(employees, compareById);
```

--Reverse sorting

```
 Comparator<Employee> comparator = Comparator.comparing(e -> e.getFirstName());

employees.sort(comparator.reversed());
```

From <https://howtodoinjava.com/java8/using-comparator-becomes-easier-with-lambda-expressions-java-8/>

-- Sort on multiple fields – thenComparing()

```
Comparator<Employee> groupByComparator = Comparator.comparing(Employee::getFirstName)
                                                   .thenComparing(Employee::getLastName);
employees.sort(groupByComparator);
```

From <https://howtodoinjava.com/java8/using-comparator-becomes-easier-with-lambda-expressions-java-8/>

New features in Java 8

- Lambda expression – Adds functional processing capability to Java.
- Method references – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- Default method – Interface to have default method implementation.
- New tools – New compiler tools and utilities are added like 'jdeps' to figure out dependencies.
- Stream API – New stream API to facilitate pipeline processing.

- Date Time API – Improved date time API.
- Optional – Emphasis on best practices to handle null values properly.
- Nashorn, JavaScript Engine – A Java-based engine to execute JavaScript code.

Java 8 Method Reference

From <https://www.codementor.io/eh3rrera/using-java-8-method-reference-du10866vx>

A method reference is the shorthand syntax for a lambda expression that executes just ONE method. Here's the general syntax of a method reference:

```
Object :: methodName
```

We know that we can use lambda expressions instead of using an anonymous class. But sometimes, the lambda expression is really just a call to some method, for example:

```
Consumer<String> c = s -> System.out.println(s);
```

To make the code clearer, you can turn that lambda expression into a method reference:

```
Consumer<String> c = System.out::println;
```

In a method reference, you place the object (or class) that contains the method before the :: operator and the name of the method after it without arguments.

But you may be thinking:

- How is this clearer?
- What will happen to the arguments?
- How can this be a valid expression?
- I don't understand how to construct a valid method reference...

First of all, a method reference can't be used for any method. They can only be used to replace a single-method lambda expression.

So to use a method reference, you first need a lambda expression with one method. And to use a lambda expression, you first need a functional interface, an interface with just one abstract method.

In other words:

Instead of using

AN ANONYMOUS CLASS

you can use

A LAMBDA EXPRESSION

And if this just calls one method, you can use

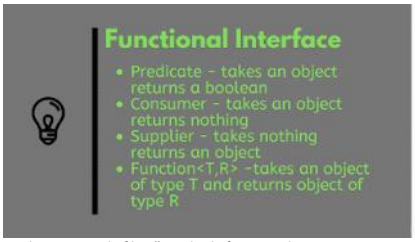
A METHOD REFERENCE

There are four types of method references:

- A method reference to a *static method*.
- A method reference to an *instance method of an object of a particular type*.
- A method reference to an *instance method of an existing object*.
- A method reference to a *constructor*.

From <https://www.codementor.io/eh3rrera/using-java-8-method-reference-du10866vx>

Functional Interface



Predicate example-filter() method of Stream class
Consumer example-forEach() method of Iterable in Java 8

BiFunction<T,U,R>-takes object as argument of type T and U and returns and object of type R---e.g map merge method

Unary Operator<T,T>-takes an object as argument of type T and returns T

Java 8 Optional

From <https://www.baeldung.com/java-optional>

It is a class that encapsulates an optional value, You can view Optional as a single-value container that either contains a value or doesn't (it is then said to be "empty").
From <https://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html>

There are several ways of creating *Optional* objects. To create an empty *Optional* object, we simply need to use its

From <https://www.baeldung.com/java-optional>

1. Optional<String> empty = Optional.empty();

From <https://www.baeldung.com/java-optional>

2. Optional<String> opt = Optional.of(name);

From <https://www.baeldung.com/java-optional>
he argument passed to the *of()* method can't be *null.* Otherwise, we'll get a *NullPointerException*

From <https://www.baeldung.com/java-optional>

3 . Optional<String> opt = Optional.ofNullable(name);

From <https://www.baeldung.com/java-optional>

By doing this, if we pass in a *null* reference, it doesn't throw an exception but rather returns an empty *Optional* object:

From <https://www.baeldung.com/java-optional>

Checking Value Presence: *isPresent()* and *isEmpty()*

From <https://www.baeldung.com/java-optional>

Default Value With *orElse()*

From <https://www.baeldung.com/java-optional>

```
@Test
public void whenOrElseWorks_thenCorrect() {
   String nullName = null;
   String name = Optional.ofNullable(nullName).orElse("john");
   assertEquals("john", name);
}
```

From <https://www.baeldung.com/java-optional>

Default Value With *orElseGet()*

From <https://www.baeldung.com/java-optional>

```
@Test
public void whenOrElseGetWorks_thenCorrect() {
   String nullName = null;
   String name = Optional.ofNullable(nullName).orElseGet(() -> "john");
   assertEquals("john", name);
}
```

From <https://www.baeldung.com/java-optional>

Exceptions with *orElseThrow()*

From <https://www.baeldung.com/java-optional>

```
@Test(expected = IllegalArgumentException.class)
public void whenOrElseThrowWorks_thenCorrect() {
   String nullName = null;
   String name = Optional.ofNullable(nullName).orElseThrow(
     IllegalArgumentException::new);
}
```

From <https://www.baeldung.com/java-optional>

Fetching values out of a OPTIONAL

Returning Value with *get()*

From <https://www.baeldung.com/java-optional>

```
@Test
public void givenOptional_whenGetsValue_thenCorrect() {
   Optional<String> opt = Optional.of("baeldung");
   String name = opt.get();
   assertEquals("baeldung", name);
}
```

From <https://www.baeldung.com/java-optional>

Conditional Return with *filter()*

From <https://www.baeldung.com/java-optional>

We can run an inline test on our wrapped value with the *filter* method. It takes a predicate as an argument and returns an *Optional* object. If the wrapped value passes testing by the predicate, then the *Optional* is returned as-is.

However, if the predicate returns *false*, then it will return an empty *Optional*:

From <https://www.baeldung.com/java-optional>

```
@Test
public void whenOptionalFilterWorks_thenCorrect() {
   Integer year = 2016;
   Optional<Integer> yearOptional = Optional.of(year);
   boolean is2016 = yearOptional.filter(y -> y == 2016).isPresent();
   assertTrue(is2016);
   boolean is2017 = yearOptional.filter(y -> y == 2017).isPresent();
   assertFalse(is2017);
}
```

From <https://www.baeldung.com/java-optional>

Transforming Value with *map()*

From <https://www.baeldung.com/java-optional>

```
@Test
public void givenOptional_whenMapWorks_thenCorrect2() {
   String name = "baeldung";
   Optional<String> nameOptional = Optional.of(name);

   int len = nameOptional
    .map(String::length)
    .orElse(0);
   assertEquals(8, len);
}
```

From <https://www.baeldung.com/java-optional>

Transforming Value with *flatMap()*

From <https://www.baeldung.com/java-optional>

Just like the *map()* method, we also have the *flatMap()* method as an alternative for transforming values. The difference is that *map* transforms values only when they are unwrapped whereas *flatMap* takes a wrapped value and unwraps it before transforming it.

From <https://www.baeldung.com/java-optional>

Optional also supports a flatMap method. Its purpose is to apply the transformation function on the value of an Optional (just like the map operation does) and then flatten the resulting two-level Optional into a single one

From <https://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html>

e.g.

Optional<Optional<Squirrel>> -> Optional<Squirrel>

From <https://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html>

From <https://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html>

Comparable vs Comparator	Comparator is used when we want to sort a collection of objects which can be compared with each other. This comparison can be done using `Comparable` interface as well, but it restrict you compare these objects in a single particular way only. If you want to sort this collection, based on multiple criterias/fields, then you have to use `Comparator` only. From <https://howtodoinjava.com/java8/using-comparator-becomes-easier-with-lambda-expressions-java-8/> 1. Comparable interface can be used to provide single way of sorting whereas Comparator interface is used to provide different w ays of sorting. 2. For using Comparable, Class needs to implement it whereas for using Comparator we don't need to make any change in the class. 3. Comparable interface is in java.lang package whereas Comparator interface is present in java.util package. 4. We don't need to make any code changes at client side for using Comparable, Arrays.sort() or collection.sort() methods automatically uses the compareTo() method of the class. For Comparator, client needs to provide the Comparator class to use in compare() method. From <https://www.journaldev.com/780/comparable-and-comparator-in-java-example>

REST TEMPLATE

Tuesday, June 4, 2019 12:10 PM

RestTemplate	Synchronous client to perform HTTP requests, exposing a simple, template method API over underlying HTTP client libraries such as the JDK HttpURLConnection, Apache HttpComponents, and others. The RestTemplate offers templates for common scenarios by HTTP method, in addition to the generalized exchange and execute methods that support of less frequent cases. From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>
Methods	<T> **ResponseEntity**<T> **exchange**(**RequestEntity**<?> requestEntity, **Class**<T> responseType) Execute the request specified in the given **RequestEntity** and return the response as **ResponseEntity**. From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html> **getForObject**(**String** url, **Class**<T> responseType, **Map**<**String**,?> uriVariables) Retrieve a representation by doing a GET on the URI template. From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>

RequestEntity<T>
extends HttpEntity<T>
	Extension of HttpEntity that adds a method and uri. Used in RestTemplate and @Controller methods.
	In RestTemplate, this class is used as parameter in exchange():
	
	From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/RequestEntity.html>
	
	Example:
	
	`MyRequest body = ...`
	`RequestEntity<MyRequest> request = RequestEntity`
	`.post(new URI("https://example.com/bar"))`
	`.accept(MediaType.APPLICATION_JSON)`
	`.body(body);`
	`ResponseEntity<MyResponse> response = template.exchange(request, MyResponse.class);`
	
	From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/RequestEntity.html>

ResponseEntity<T>	Extension of HttpEntity that adds a HttpStatus status code. Used in RestTemplate as well @Controller methods.
	In RestTemplate, this class is returned by getForEntity() and exchange():
	
	From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>
	
	Example:
	
	`ResponseEntity<String> entity = template.getForEntity("https://example.com", String.class);`
	`String body = entity.getBody();`
	`MediaType contentType = entity.getHeaders().getContentType();`
	`HttpStatus statusCode = entity.getStatusCode();`
	
	From <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>

HTTP Methods -Safe and Idempotent	These are HTTP methods which don't change the resource on the server side. For example using a GET or a HEAD request on a resource URL should NEVER change the resource. Safe methods can be **cached** and **prefetched** without any repercussions or side-effect to the resource . Here is an example of safe method There are some HTTP methods e.g. GET which produce same response no matter how many times you use them e.g. sending multipleGET request to the same URI will result in same response without any side-effect hence it is known as idempotent. On the other hand, the POST is not idempotent because if you send multiple POST request, it will result in multiple resourcecreation on the server, but again, PUT is idempotent if you are using it to update the resource. GET,PUT-IDEMPOTENT POST-NOT IDEMPOTENT
@Controller and @RestController	Yes, both `@Controller` and `@RestController` are stereotypes. The `@Controller` is actually a specialization of Spring's `@Component` stereotype annotation. This means that class annotated with `@Controller` will also be automatically be detected by Spring container as part of container's component scanning process. And, @RestController is a specialization of `@Controller` for RESTful web service. It not only combines `@ResponseBody` and @Controller annotation but also gives more meaning to your controller class to clearly indicate that it deals with RESTful requests. Spring Framework may also use this annotation to provide some more useful features related to REST API development in future. Read more: https://javarevisited.blogspot.com/2018/02/top-20-spring-rest-interview-questions-answers-java.html#ixzz5ptjDYg8p From <https://javarevisited.blogspot.com/2018/02/top-20-spring-rest-interview-questions-answers-java.html> `@RestController` you get the `@ResponseBody` annotation automatically, which means you don't need to separately annotate your handler methods with `@ResponseBody` annotation Read more: https://javarevisited.blogspot.com/2018/02/top-20-spring-rest-interview-questions-answers-java.html#ixzz5ptjcSNdC From <https://javarevisited.blogspot.com/2018/02/top-20-spring-rest-interview-questions-answers-java.html>

Sample Rest API and Client	REST API Code
	
	`@RequestMapping(value = "/employees/{id}")`
	`public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") int id)`
	`{`
	`if (id <= 3) {`
	`EmployeeVO employee = new EmployeeVO(1,"Lokesh","Gupta","howtodoinjava@gmail.com");`
	`return new ResponseEntity<EmployeeVO>(employee, HttpStatus.OK);`

```
    }
    return new ResponseEntity(HttpStatus.NOT_FOUND);
}
```

REST Client Code

```
private static void getEmployeeById()
{
    final String uri = "http://localhost:8080/springrestexample/employees/{id}";

    Map<String, String> params = new HashMap<String, String>();
    params.put("id", "1");

    RestTemplate restTemplate = new RestTemplate();
    EmployeeVO result = restTemplate.getForObject(uri, EmployeeVO.class, params);

    System.out.println(result);
}
```

From <https://howtodoinjava.com/spring-restful/spring-restful-client-resttemplate-example/>

JWT

- Clients logs in by sending their credentials to the identity provider.
- The identity provider verifies the credentials; if all is OK, it retrieves the user data, generates a JWT containing user details and permissions that will be used to access the services, and it also sets the expiration on the JWT (which might be unlimited).
- Identity provider signs, and if needed, encrypts the JWT and sends it to the client as a response to the initial request with credentials.
- Client stores the JWT for a limited or unlimited amount of time, depending on the expiration set by the identity provider.
- Client sends the stored JWT in an Authorization header for every request to the service provider.
- For each request, the service provider takes the JWT from the `Authorization` header and decrypts it, if needed, validates the signature, and if everything is OK, extracts the user data and permissions. Based on this data solely, and again without looking up further details in the database or contacting the identity provider, it can accept or deny the client request. The only requirement is that the identity and service providers have an agreement on encryption so that service can verify the signature or even decrypt which identity was encrypted.

From <https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java>

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

Header(Algorithm and token type)

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
 "alg": "HS256",
 "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 data.

An example payload could be:

```
{
 "sub": "1234567890",
 "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 SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
```

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

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

spring-data-jpa

In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific *Repository* interface – *JpaRepository*. This will enable Spring Data to find this interface and automatically create an implementation for it.

By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO.

From <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>

by implementing one of the *Repository* interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.

To define more specific access methods, Spring JPA supports quite a few options:

- simply **define a new method** in the interface
- provide the actual **JPQ query** by using the *@Query* annotation
- use the more advanced **Specification and Querydsl support** in Spring Data
- define **custom queries** via JPA Named Queries

From <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>

To activate the Spring JPA repository support we can use the *@EnableJpaRepositories* annotation and specify the package that contains the DAO interfaces:

```
@EnableJpaRepositories(basePackages = "com.baeldung.jpa.dao"
public class PersistenceConfig { ... }
```

From <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-jpa</artifactId>
  <version>2.1.6.RELEASE</version>
</dependency>
```

From <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>

Aspect-Oriented Programming

From <https://www.baeldung.com/spring-interview-questions>

Aspects enable the modularization of cross-cutting concerns such as transaction management that span multiple types and objects by adding extra behavior to already existing code without modifying affected classes.

From <https://www.baeldung.com/spring-interview-questions>

Java Singleton

From <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for logging, drivers objects, caching and thread pool.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.
- Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

From <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

1. Singleton with eager initialization

This is a design pattern where an instance of a class is created much before it is actually required. Mostly it is done on system startup. In an eager initialization singleton pattern, the singleton instance is created irrespective of whether any other class actually asked for its instance or not.

```
public class EagerSingleton {
    private static volatile EagerSingleton instance = new EagerSingleton();

    // private constructor
    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

The above method works fine, but it has one drawback. The instance is created irrespective of it is required in runtime or not. If this instance is not a big object and you can live with it being unused, this is the best approach.

Let's solve the above problem in the next method.

2. Singleton with lazy initialization

In computer programming, lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process, until the first time it is needed. In a singleton pattern, it restricts the creation of the instance until it is requested for first time. Lets see this in code:

```
public final class LazySingleton {
    private static volatile LazySingleton instance = null;

    // private constructor
    private LazySingleton() {
    }
```

```
    public static LazySingleton getInstance() {
        if (instance == null) {
            synchronized (LazySingleton.class) {
                instance = new LazySingleton();
            }
        }
        return instance;
    }
}
```

On the first invocation, the above method will check if the instance is already created using the instance variable. If there is no instance i.e. the instance is null, it will create an instance and will return its reference. If the instance is already created, it will simply return the reference of the instance.

From <https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/>

Describe and compare fail-fast and fail-safe iterators.

From <https://stackify.com/java-interview-questions/>

The main distinction between fail-fast and fail-safe iterators is whether or not the collection can be modified while it is being iterated. Fail-safe iterators allow this; fail-fast iterators do not.
Fail-fast iterators operate directly on the collection itself. During iteration, fail-fast iterators fail as soon as they realize that the collection has been modified (i.e., upon realizing that a member has been added, modified, or removed) and will throw a **ConcurrentModificationExceptio**n. Some examples include **ArrayList**, **HashSet**, and **HashMap** (most JDK1.4 collections are implemented to be fail-fast).
Fail-safe iterates operate on a cloned copy of the collection and therefore do not throw an exception if the collection is modified during iteration. Examples would include iterators returned by **ConcurrentHashMap** or **CopyOnWriteArrayList**.

From <https://stackify.com/java-interview-questions/>

Runnable and *Callable* interfaces

From <https://www.baeldung.com/java-concurrency-interview-questions>

The *Runnable* interface has a single *run* method. It represents a unit of computation that has to be run in a separate thread. The *Runnable* interface does not allow this method to return value or to throw unchecked exceptions.

The *Callable* interface has a single *call* method and represents a task that has a value. That's why the *call* method returns a value. It can also throw exceptions. *Callable* is generally used in *ExecutorService* instances to start an asynchronous task and then call the returned *Future* instance to get its value.

From <https://www.baeldung.com/java-concurrency-interview-questions>

Spring Cloud Config Server(Micro services)

The server project is relying on the *spring-cloud-config-server* module, as well as the *spring-boot-starter-security* and *spring-boot-starter-web* starter bundles:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
  <version>1.1.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>1.4.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>1.4.0.RELEASE</version>
</dependency>
```

From <https://www.baeldung.com/spring-cloud-configuration>

The main part of the application is a config class – more specifically a *@SpringBootApplication* – which pulls in all the required setup through the *auto-configure* annotation *@EnableConfigServer:*

From <https://www.baeldung.com/spring-cloud-configuration>

```
@SpringBootApplication
@EnableConfigServer
@EnableEncryptableProperties----///for enabling encryption of properties e.g.jasypt-java simplified encryption
public class ConfigServer {

  public static void main(String[] arguments) {
    SpringApplication.run(ConfigServer.class, arguments);
  }
}
```

From <https://www.baeldung.com/spring-cloud-configuration>

Now we need to configure the server *port* on which our server is listening and a *Git*-url which provides our version-controlled configuration content. The latter can be used with protocols like *http*, *ssh* or a simple *file* on a local filesystem.

From <https://www.baeldung.com/spring-cloud-configuration>

```
server.port=8888
spring.cloud.config.server.git.uri=ssh://localhost/config-repo
spring.cloud.config.server.git.clone-on-start=true
security.user.name=root
security.user.password=s3cr3t
```

From <https://www.baeldung.com/spring-cloud-configuration>

6. The Client Implementation

From <https://www.baeldung.com/spring-cloud-configuration>

The configuration, to fetch our server, must be placed in a resource file named *bootstrap.application*, because this file (like the name implies) will be loaded very early while the application starts:

```
1 @SpringBootApplication
2 @RestController
```

```
public class ConfigClient {

    @Value("${user.role}")
    private String role;

    public static void main(String[] args) {
        SpringApplication.run(ConfigClient.class, args);
    }

    @RequestMapping(
      value = "/whoami/{username}",
      method = RequestMethod.GET,
      produces = MediaType.TEXT_PLAIN_VALUE)
    public String whoami(@PathVariable("username") String username) {
        return String.format("Hello!
          You're %s and you'll become a(n) %s...\n", username, role);
    }
}
```

In addition to the application name, we also put the active profile and the connection-details in our b*ootstrap.properties*:

```
spring.application.name=config-client
spring.profiles.active=development
spring.cloud.config.uri=http://localhost:8888
spring.cloud.config.username=root
spring.cloud.config.password=s3cr3t
```

From <https://www.baeldung.com/spring-cloud-configuration>

5. Querying the Configuration

From <https://www.baeldung.com/spring-cloud-configuration>

/{application}-{profile}.properties

From <https://www.baeldung.com/spring-cloud-configuration>

Discovery(Microservices)

Since our applications could be running on any ip/port combination we need a central address registry that can serve as an application address lookup.

From <https://www.baeldung.com/spring-cloud-bootstrapping>

we need a way for all of our servers to be able to find each other. We will solve this problem by setting the *Eureka* discovery server up.

From <https://www.baeldung.com/spring-cloud-bootstrapping>

When a new server is provisioned it will communicate with the discovery server and register its address so that others can communicate with it. This way other applications can consume this information as they make requests.

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Setup

From <https://www.baeldung.com/spring-cloud-bootstrapping>

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Spring Config

From <https://www.baeldung.com/spring-cloud-bootstrapping>

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApplication {...}
```

@EnableEurekaServer will configure this server as a discovery server using *Netflix Eureka*. *Spring Boot* will automatically detect the configuration dependency on the classpath and **lookup the configuration from the config server**.

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Properties

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Now we will add two properties files:

bootstrap.properties in src/main/resources **(Referring to cloud config url from discovery bootstrap.properties)**

```
spring.cloud.config.name=discovery
spring.cloud.config.uri=http://localhost:8081
```

These properties will let discovery server query the config server at startup.

discovery.properties in our Git repository

```
spring.application.name=discovery
server.port=8082

eureka.instance.hostname=localhost


eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Add Dependency to the Config Server

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Add this dependency to the config server POM file:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

From <https://www.baeldung.com/spring-cloud-bootstrapping>

Eureka Client

From <https://www.baeldung.com/spring-cloud-netflix-eureka>

For a *@SpringBootApplication* to be discovery-aware, we've to include some *Spring Discovery Client* (for example *spring-cloud-starter-netflix-eureka-client*) into our *classpath.*

Then we need to annotate a *@Configuration* with either *@EnableDiscoveryClient* or *@EnableEurekaClient*

From <https://www.baeldung.com/spring-cloud-netflix-eureka>

Junit testing

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

From <https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/>

Mockito Testing
Annotate class with
`@RunWith(MockitoJUnitRunner.class)`

From <https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/>

1. To mock any dependencies added by Spring using @Autowired

- The @Mock annotation creates a mock implementation for the class it is annotated with.
- @InjectMocks also creates the mock implementation, additionally injects the dependent mocks that are marked with the annotations `@Mock` `into it.`

From <https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/>

2. add the set-up method that initializes all of the mocked objects together when the test runs. The method annotated with `@Before` gets ran before each test method. The `init()` method runs `MockitoAnnotations.initMocks(this)` using `this` instance as the argument. This sets up our mocks before each test.

From <https://stackabuse.com/how-to-test-a-spring-boot-application/>

```
@InjectMocks
  EmployeeManager manager;

  @Mock
  EmployeeDao dao;

  @Before
  public void init() {
      MockitoAnnotations.initMocks(this);
  }

  @Test
  public void getAllEmployeesTest()
  {
      List<EmployeeVO> list = new ArrayList<EmployeeVO>();
      EmployeeVO empOne = new EmployeeVO(1, "John", "John", "howtodoinjava@gmail.com");
      EmployeeVO empTwo = new EmployeeVO(2, "Alex", "kolenchiski", "alexk@yahoo.com");
      EmployeeVO empThree = new EmployeeVO(3, "Steve", "Waugh", "swaugh@gmail.com");

      list.add(empOne);
      list.add(empTwo);
      list.add(empThree);

      when(dao.getEmployeeList()).thenReturn(list);
```

```
        //test
        List<EmployeeVO> empList = manager.getEmployeeList();

        assertEquals(3, empList.size());
        verify(dao, times(1)).getEmployeeList();
    }
```

From <https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/>

REFER TO :https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/

Transactions with Spring and JPA

From <https://www.baeldung.com/transaction-configuration-with-jpa-and-spring>

***@EnableTransactionManagement* annotation** that we can use in a *@Configuration* class and enable transactional support:

```
@Configuration

@EnableTransactionManagement
public class PersistenceJPAConfig{
```

From <https://www.baeldung.com/transaction-configuration-with-jpa-and-spring>

List out the dependency scope in Maven?

From <https://career.guru99.com/top-20-maven-interview-questions/>

The various dependency scope used in Maven are:
- Compile: It is the default scope, and it indicates what dependency is available in the classpath of the project
- Provided: It indicates that the dependency is provided by JDK or web server or container at runtime
- Runtime: This tells that the dependency is not needed for compilation but is required during execution
- Test: It says dependency is available only for the test compilation and execution phases
- System: It indicates you have to provide the system path
- Import: This indicates that the identified or specified POM should be replaced with the dependencies in that POM's section

From <https://career.guru99.com/top-20-maven-interview-questions/>

Design Patterns

SINGELTON

Singleton pattern is a design solution where an application wants to have one and only one instance of any class, in all possible scenarios without any exceptional condition

From <https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/>

- A private constructor
- A static field containing its only instance
- A static factory method for obtaining the instance

We'll also add an info property, for later usage only. So, our implementation will look like this:

```
public final class ClassSingleton {

    private static ClassSingleton INSTANCE;
    private String info = "Initial info class";

    private ClassSingleton() {
    }

    public static ClassSingleton getInstance() {
        if(INSTANCE == null) {
            INSTANCE = new ClassSingleton();
        }

        return INSTANCE;
    }

    // getters and setters
}
```

From <https://www.baeldung.com/java-singleton>

- Singleton pattern is used for logging, drivers objects, caching and thread pool.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.
- Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

From <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

Factory Method Design Pattern

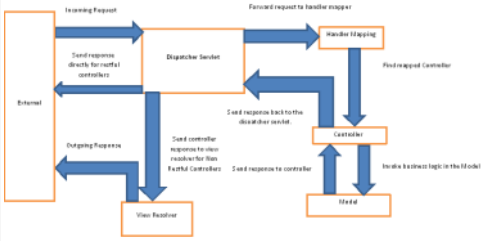
From <https://www.baeldung.com/creational-design-patterns>

	“defines an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses”. This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor. From <https://www.baeldung.com/creational-design-patterns>

Spring Topics

Wednesday, May 24, 2017 4:54 PM

Topic	Description
Dependency Injection	The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The **Inversion of Control (IoC)** is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control. When writing a complex Java application, application classes should be as independent as possible of other Java classes to in crease the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent. What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC. Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, we will explain this concept in a separate chapter with relevant example. Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies. The act of connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves. From <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
Dependency Management	The process of dependency management involves locating those resources, storing them and adding them to classpaths. Dependencies can be direct (e.g. my application depends on Spring at runtime), or indirect (e.g. my application depends on `commons-dbcp` which depends on `commons-pool`). The indirect dependencies are also known as "transitive" and it is those dependencies that are hardest to identify and manage.
Artifactory	Maven Central, which is the default repository that Maven queries, and does not require any special configuration to use. Many of the common libraries that Spring depends on also are available from Maven Central and a large section of the Spring community uses Maven for dependency management, so this is convenient for them. The names of the jars here are in the form `spring-*-<version>.jar` and the Maven groupId is `org.springframework`.
IOC	IoC is also known as *dependency injection*(DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern. The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. The `BeanFactory`interface provides an advanced configuration mechanism capable of managing any type of object. `ApplicationContext` is a sub-interface of `BeanFactory`. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the `WebApplicationContext` for use in web applications. From <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html>
The Spring IoC Container From <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>	An IoC container is a common characteristic of frameworks that implement IoC. In the Spring framework, the IoC container is represented by the interface *ApplicationContext*. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their lifecycle. From <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring> *ApplicationContext* is an interface representing a container holding all information, metadata, and beans in the application. It also extends the *BeanFactory* interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for individual beans. From <https://www.baeldung.com/spring-interview-questions> *Figure 1. The Spring IoC container*
Dependency Injection	DI exists in 3major variants, Constructor-based dependency injection , Setter-based dependency injection,Field-based dependency injection **Field-Based Dependency Injection** In case of Field-Based DI, we can inject the dependencies by marking them with an *@Autowired* annotation: From <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
Spring Bean From <https://www.baeldung.com/spring-interview-questions>	he Spring Beans are Java Objects that are initialized by the Spring IoC container. From <https://www.baeldung.com/spring-interview-questions> By default, a Spring Bean is initialized as a *singleton*. From <https://www.baeldung.com/spring-interview-questions>
Bean Scopes	To set Spring Bean's scope, we can use *@Scope* annotation or "scope" attribute in XML configuration files. There are five supported scopes: • **singleton** • **prototype** • **request** • **session** • **global-session** From <https://www.baeldung.com/spring-interview-questions> By default, a Spring Bean is initialized as a *singleton*. From <https://www.baeldung.com/spring-interview-questions> For a bean with the default *singleton* scope, Spring first if a cached instance of the bean already exists and only creates a new one if it doesn't. The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made. From <https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s05.html>

	As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans. From <https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s05.html> From <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
@Autowired	@Autowired can be applied on a bean's constructor, field, setter method or a config method to autowire the dependency using Spring's dependency injection.
@Resource vs @Autowired	Standard @Resource annotation marks a resource that is needed by the application. It is analogous to @Autowired in that both injects beans by type when no attribute provided. But with name attribute, @Resource allows you to inject a bean by it's name, which @Autowired does not. From <http://websystique.com/spring/spring-dependency-injection-annotation-beans-auto-wiring-using-autowired-qualifier-resource-annotations-configuration/> No optionality in @Resource and no autowiring by bean name in @Autowired. All in all, @Autowired is the most widely used option compare to @Resource and autowire attribute in XML. From <http://websystique.com/spring/spring-dependency-injection-annotation-beans-auto-wiring-using-autowired-qualifier-resource-annotations-configuration/>
@Component Scan	@ComponentScan which will make Spring auto detect the annotated beans via scanning the specified package and wire them wherever needed (using @Resource or @Autowired). From <http://websystique.com/spring/spring-dependency-injection-annotation-beans-auto-wiring-using-autowired-qualifier-resource-annotations-configuration/>
@Qualifier	is useful for the situation where you have more than one bean matching the type of dependency and thus resulting in ambiguity.
@Controller	marks this class as spring bean which may handle different HTTP requests based on mapping specified on class or individual controller methods. From <http://websystique.com/springmvc/spring-4-mvc-helloworld-tutorial-full-example/>
@RequestMapping	marks this class as spring bean which may handle different HTTP requests based on mapping specified on class or individual controller methods. From <http://websystique.com/springmvc/spring-4-mvc-helloworld-tutorial-full-example/>
ModelMap	is a Map implementation, which saves you from old request.getAttribute/ request.setAttribute. It provides a way to set/get attributes from/to request or session. From <http://websystique.com/springmvc/spring-4-mvc-helloworld-tutorial-full-example/>
@EnableWebSecurity From <https://www.baeldung.com/spring-interview-questions>	To use Spring Security in web applications, you can get started with a simple annotation: *@EnableWebSecurity*. From <https://www.baeldung.com/spring-interview-questions>
Design Patterns used in the Spring Framework From <https://www.baeldung.com/spring-interview-questions>	• **Singleton Pattern:** Singleton-scoped beans • **Factory Pattern:** Bean Factory classes • **Prototype Pattern:** Prototype-scoped beans • **Adapter Pattern:** Spring Web and Spring MVC • **Proxy Pattern:** Spring Aspect Oriented Programming support • **Template Method Pattern:** *JdbcTemplate*, *HibernateTemplate*, etc. • **Front Controller:** Spring MVC *DispatcherServlet* • **Data Access Object:** Spring DAO support • **Model View Controller:** Spring MVC From <https://www.baeldung.com/spring-interview-questions>
***Model View Controller* architecture** From <https://www.baeldung.com/spring-controllers>	 Fig 1 MVC Architecture flow In the traditional approach, *MVC* applications are not service-oriented hence there is a *View Resolver* that renders final views based on data received from a *Controller*. *RESTful* applications are designed to be service-oriented and return raw data (JSON/XML typically). Since these applications do not do any view rendering, there are no *View Resolvers* – the *Controller* is generally expected to send data directly via the HTTP response. From <https://www.baeldung.com/spring-controllers> Spring boot-Only below properties are needed instead of defining a DispatchedServlet etc. spring.view.prefix:/WEB-INF/ spring.view.suffix:.jsp spring.view.view-names:jsp/*
checked and an unchecked exception From <https://www.baeldung.com/java-exceptions-interview-questions>	A checked exception must be handled within a *try-catch* block or declared in a *throws* clause; whereas an unchecked exception is not required to be handled nor declared. Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively. All exceptions are checked exceptions, except those indicated by *Error*, *RuntimeException*, and their subclasses. From <https://www.baeldung.com/java-exceptions-interview-questions> An exception is an event that represents a condition from which is possible to recover, whereas error represents an external situation usually impossible to recover from. All errors thrown by the JVM are instances of *Error* or one of its subclasses, the more common ones include but are not limited to: • *OutOfMemoryError* – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available • *StackOverflowError* – occurs when the stack space for a thread has run out, typically because an application recurses too deeply • *ExceptionInInitializerError* – signals that an unexpected exception occurred during the evaluation of a static initializer • *NoClassDefFoundError* – is thrown when the classloader tries to load the definition of a class and couldn't find it, usually because the required *class* files were not found in the classpath • *UnsupportedClassVersionError* – occurs when the JVM attempts to read a *class* file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java Although an error can be handled with a *try* statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown. From <https://www.baeldung.com/java-exceptions-interview-questions>

try-with-resources
 	
From <https://www.baeldung.com/java-flow-control-interview-questions>
	The *try-with-resources* statement declares and initializes one or more resources before executing the *try* block and closes them automatically at the end of the statement regardless of whether the block completed normally or abruptly. Any object implementing *AutoCloseable* or *Closeable* interfaces can be used as a resource:
	
	`try (StringWriter writer = new StringWriter()) {`
	`  writer.write("Hello world!");`
	`}`
	
	From <https://www.baeldung.com/java-flow-control-interview-questions>

- *ngFor
- *ngIf
- Interpolation {{ }}
- Property binding [ ]
- Event binding ( )

From <https://angular.io/start>

STEP 1: GLOBALLY UPGRADE ANGULAR CLI FROM 1.X TO 6

- Install the angular-cli 6 globally using below command:

npm install -g @angular/cli

- If the above command throws permission denied error then run the following command:

sudo npm install -g @angular/cli

Note: Ensure Node.js V8+ is already installed as mentioned in prerequisites. Lower version of Node.js will cause issues while upgradation of cli version, however, while execution of cli you will see errors.

STEP 2: UPGRADE ANGULAR CLI VERSION IN PROJECT

- Go to project the source directory
- Run the following command to install latest angular/cli at project level:

npm install @angular/cli@latest

- Run the following command to upgrade the angular-cli at the project:

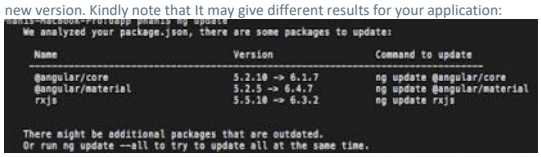
ng update @angular/cli

STEP 3: IDENTIFY THE PACKAGES THAT NEED AN UPGRADE

- Run the below command to identify the packages that are needed to be upgraded:

ng update

- Above command produced output like below. It gives package name along with current and new version. Kindly note that It may give different results for your application:



STEP 4: UPGRADE PACKAGES

- Run all the below-listed commands to upgrade the packages:

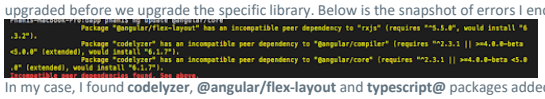
ng update @angular/core
ng update @angular/material
ng update rxjs

- Do not worry if upgrade throws an error. If you get an error, kindly skip this step and move on to next step.
- You may have to repeat this step after finishing the next step.

Notes: Kindly note that above commands may change based on the application you are trying to upgrade

STEP 5: SOLVE PEER DEPENDENCY ISSUES

- If there are no errors in the previous step, then you don't need to anything in this step
- In case of an error, kindly read the error carefully to identify the packages that need to be upgraded before we upgrade the specific library. Below is the snapshot of errors I encountered



- In my case, I found **codelyzer**, **@angular/flex-layout** and **typescript@** packages added need upgradation
- Install **codelyzer, @angular/flex-layout** to the latest version and **typescript@** to 2.6.2 with following commands:

npm install codelyzer@latest
npm install @angular/flex-layout@latest
npm install typescript@ 2.6.2

- After this run 'Upgrade packages' again. These two steps i.e. 'Upgrade packages' and 'Solve peer dependency issues' will be repeated until all the dependencies are resolved.

AFTER FINISHING ALL THE 5 STEPS WE ARE DONE WITH THE UPGRADE OF AN APPLICATION TO ANGULAR6!

From <https://walkingtree.tech/upgrading-angular-4-5-projects-to-angular-6/>

Spring AOP

Friday, July 19, 2019 11:35 AM

In AOP, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed crosscutting concerns).

AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well. You can add/remove concerns without recompiling complete source code simply by changing configuration files (if you are applying aspects suing XML configuration).

From <https://howtodoinjava.com/spring-aop-tutorial/>

1. An important term in AOP is advice. It is the action taken by an aspect at a particular join-point.
2. Joinpoint is a point of execution of the program, such as the execution of a method or the handling of an exception. In Spring AOP, a joinpoint always represents a method execution.
3. Pointcut is a predicate or expression that matches join points.
4. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut.
5. Spring uses the AspectJ pointcut expression language by default.

From <https://howtodoinjava.com/spring-aop-tutorial/>

What is Spring AOP Proxy?

A proxy is a well-used design pattern. To put it simply, a proxy is an object that looks like another object, but adds special functionality behind the scene.

Spring AOP is proxy-based. AOP proxy is an object created by the AOP framework in order to implement the aspect contracts in runtime.

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied. Spring AOP can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces.

CGLIB is used by default if a business object does not implement an interface.

From <https://howtodoinjava.com/interview-questions/top-spring-aop-interview-questions-with-answers/>

Types of AOP advices

There are five types of advice in spring AOP.

1. Before advice: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
2. After returning advice: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
3. After throwing advice: Advice to be executed if a method exits by throwing an exception.
4. After advice: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
5. Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

From <https://howtodoinjava.com/spring-aop-tutorial/>

Sample Code

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>4.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.6.11</version>
</dependency>
<dependency>
```

From <https://howtodoinjava.com/spring-aop-tutorial/>

```
@Aspect
public class EmployeeCRUDAspect {

    @Before("execution(* EmployeeManager.getEmployeeById(..))")         //point-cut
expression
    public void logBeforeV1(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logBeforeV1() : "+
joinPoint.getSignature().getName());
    }
}
```

From <https://howtodoinjava.com/spring-aop-tutorial/>

```
@Component
public class EmployeeManager
{
    public EmployeeDTO getEmployeeById(Integer employeeId) {
        System.out.println("Method getEmployeeById() called");
        return new EmployeeDTO();
    }
}
```

From <https://howtodoinjava.com/spring-aop-tutorial/>

```
ublic class TestAOP
{
    @SuppressWarnings("resource")
    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext
                            ("com/howtodoinjava/demo/aop/applicationContext.xml");

        EmployeeManager manager = context.getBean(EmployeeManager.class);

        manager.getEmployeeById(1);
    }
}
```

From <https://howtodoinjava.com/spring-aop-tutorial/>

Program output:

```
EmployeeCRUDAspect.logBeforeV1() : getEmployeeById
Method getEmployeeById() called
Console
```

From <https://howtodoinjava.com/spring-aop-tutorial/>

RMQ

Thursday, August 1, 2019 5:39 PM

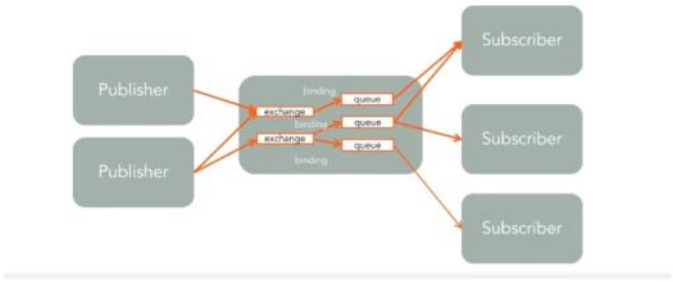
AMQP (Advanced Message Queuing Protocol) is a protocol that RabbitMQ uses for messaging.

From <https://dzone.com/articles/all-you-need-to-know-about-asyncronous-messaging-u>

Code Example

http://candidjava.com/tutorial/spring-boot-rabbitmq-example-using-maven/

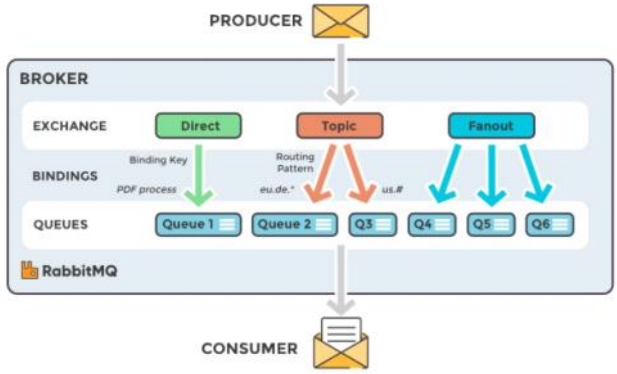
RabbitMQ Architectural Design



Exchange: Takes a message and routes it to one or more queues. Routing algorithms decides where to send the message from the exchange. Routing algorithms depends on the exchange type and rules called "bindings."

Exchange Type	Routing Algorithms	Purpose
Direct	It routes messages with a routing key equal to the routing key declared by the binding queue	This is a Default exchange type. It is used when a message needs to send to a queue
Fanout	It routes messages to all the queues from the bound exchange. If routing key is provided then it will be ignored	Useful for broadcast feature using publish subscribe pattern
Topic	It routes messages to queues based on either full or a portion of routing key matches	Useful for broadcast to specific queues based on some criteria
Headers	Routes messages based upon matching of the message header to specified header based on binding queue	Useful for directing messages which may contain a subset of known criteria

From <https://dzone.com/articles/all-you-need-to-know-about-asyncronous-messaging-u>



Topics: Topics are the subject part of the messages. These are the optional parameters for message exchange.

Bindings: "Bindings" is the glue that holds exchanges and queues together. These are the rules for routing algorithms.

Queue: Queue is a container for messages. It is only bound by the host's memory and disk limit. Queues are the final destination for messages before being popped up by subscribers.

Property Name	Description
Name	Name of the queue
Durable	Either persists the queue to the disk or not
Exclusive	Delete the queue if not used anymore
Auto-Delete	Delete the queue if consumer unsubscribes

Producer: Producer is a program that sends message to a queue.

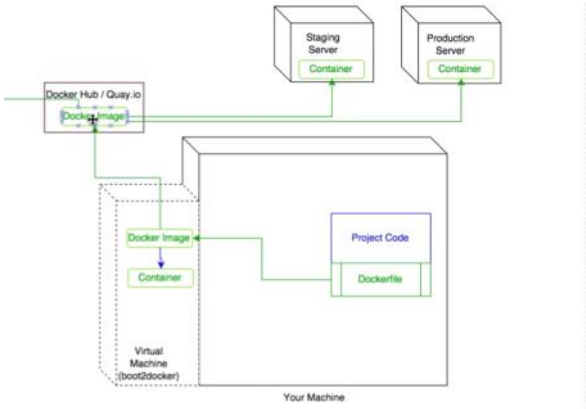
Consumer: A consumer is a program which receives messages from the

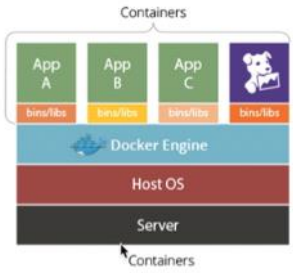
queue.

From <https://dzone.com/articles/all-you-need-to-know-about-asyncronous-messaging-u>

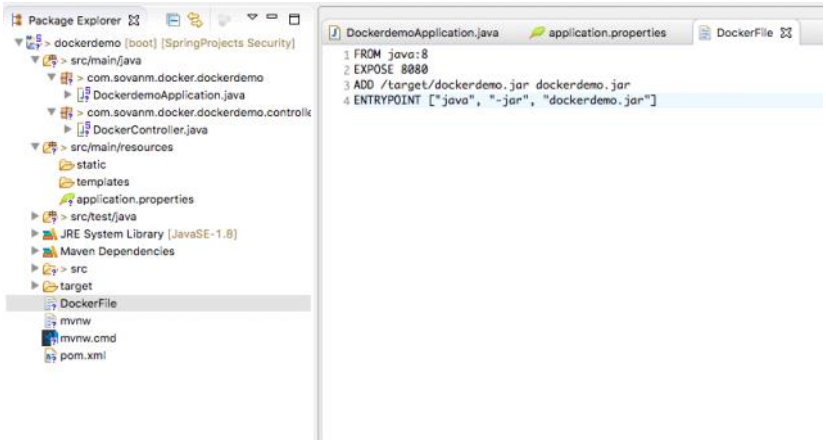
Docker

Saturday, August 3, 2019 1:28 PM



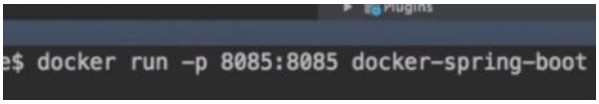


Here is our Dockerfile. Create a simple file in the project folder and add these steps in that file:



1. **FROM java:8** means this is a Java application and will require all the Java librariesk so it will pull all the Java-related libraries and add them to the container.
2. **EXPOSE 8080** means that we would like to expose 8080 to the outside world to access our application.
3. ADD /target/dockerdemo.jar dockerdemo.jar
   ADD <source from where Docker should create the image> <destination>
4. **ENTRYPOINT [“java”, “-jar”, “dockerdemo.jar”]** will run the command as the entry point as this is a JAR and we need to run this JAR from within Docker.

From <https://dzone.com/articles/deploying-spring-boot-on-docker>



docker build -t spring-boot-websocket-chat-demo .

From <https://www.callicoder.com/spring-boot-docker-example/>

Once you have a docker image, you can run it using `docker run` command like so -

```
$ docker run -p 5000:8080 spring-boot-websocket-chat-demo
```

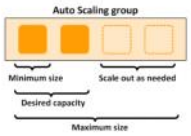
In the `run` command, we have specified that the port `8080` on the container should be mapped to the port `5000` on the Host OS.

Once the application is started, you should be able to access it at `http://localhost:5000`.

From <https://www.callicoder.com/spring-boot-docker-example/>

Saturday, August 3, 2019 3:03 PM

- EC2 (Elastic Compute Cloud)
- VPC (Virtual Private Cloud)
- S3 (Simple Storage Service)
- Relational Database Service
- Route 53
- ELB (Elastic Load Balancing)
- Autoscaling



13 How do you setup a system to monitor website metrics in real-time in AWS?

AWS CloudWatch

- CloudWatch events helps us to monitor application status of various AWS services and custom events
- Using CloudWatch we can monitor:

1. State changes in Amazon EC2
2. Auto-scaling lifecycle events
3. Scheduled events
4. AWS API calls
5. Console sign-in events

9 What services can be used to create a centralized logging solution?

- Log management helps organizations to track a relationship between operational, security and change management events
- It also helps you to understand the infrastructure
- We can create a centralized logging solution using the following:

S3 vs EBS vs EFS

EBS may be good for setting up a drive for virtual machines, and S3 is good for storage, but what if you want to run an application with high workloads that need scalable storage and relatively fast output? Amazon Elastic File System was created to fulfill those needs.

From <https://www.cloudberrylab.com/resources/blog/amazon-s3-vs-ebs-vs-efs/>

AMAZON S3	AMAZON EBS	AMAZON EFS
Can be publicly accessible Web interface Object Storage Scalable Slower than EBS and EFS	Accessible only via the given EC2 Machine File System interface Block Storage Hardly scalable Faster than S3 and EFS	Accessible via several EC2 machines and AWS services Web and file system interface Object storage Scalable Faster than S3, slower than EBS
Good for storing backups	Is meant to be EC2 drive	Good for shareable applications and workloads

From <https://www.cloudberrylab.com/resources/blog/amazon-s3-vs-ebs-vs-efs/>

18 What are the different types of EC2 instances based on their costs?

- There are three types of Amazon EC2 instances based on costs:

CLOUD WATCH

Auto Scaling is enabled by Amazon CloudWatch and is available at no extra cost. AWS CloudWatch can be used to measure CPU utilization, network traffic, etc.

From <https://www.tutorialspoint.com/amazon_web_services/amazon_web_services_auto_scaling.htm>

Amazon ECS is a highly scalable Docker container management service that allows you to run and manage distributed applications that run in Docker containers.

AWS Lambda is an event-driven task compute service that runs your code in response to "events" such as changes in data, website clicks, or messages from other AWS services without you having to manage any compute infrastructure.

From <https://aws.amazon.com/ecs/faqs/>